

2 UNDERSTANDING LINE DRAWINGS OF SCENES WITH SHADOWS

David Waltz

2.1 INTRODUCTION

How do we ascertain the shapes of unfamiliar objects? Why do we so seldom confuse shadows with real things? How do we “factor out” shadows when looking at scenes? How are we able to see the world as essentially the same whether it is a bright sunny day, an overcast day, or a night with only streetlights for illumination? In the terms of this paper, how can we recognize the identity of Figs. 2.1 and 2.2? Do we use learning and knowledge to interpret what we see, or do we somehow automatically see the world as stable and independent of lighting? What portions of scenes can we understand from local features alone, and what configurations require the use of global hypotheses?

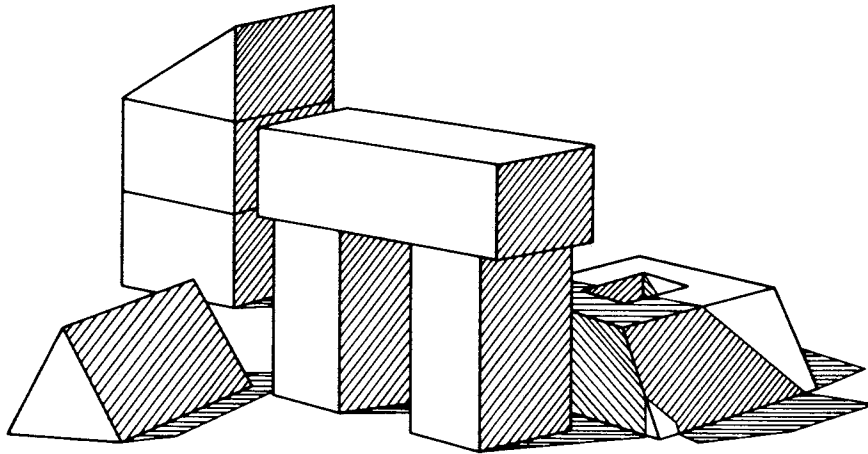


Fig. 2.1

In this essay I describe a working collection of computer programs which reconstruct three-dimensional descriptions from line drawings which are obtained from scenes composed of plane-faced objects under various lighting conditions. The system identifies shadow lines and regions, groups regions which belong to the same object, and notices such relations as contact or lack of contact between the objects, support and in-front-of/behind relations between the objects as well as information about the spatial orientation of various regions, all using the description it has generated.

2.1.1 Descriptions

The overall goal of the system is to provide a precise description of a plausible scene which could give rise to a particular line drawing. It is therefore

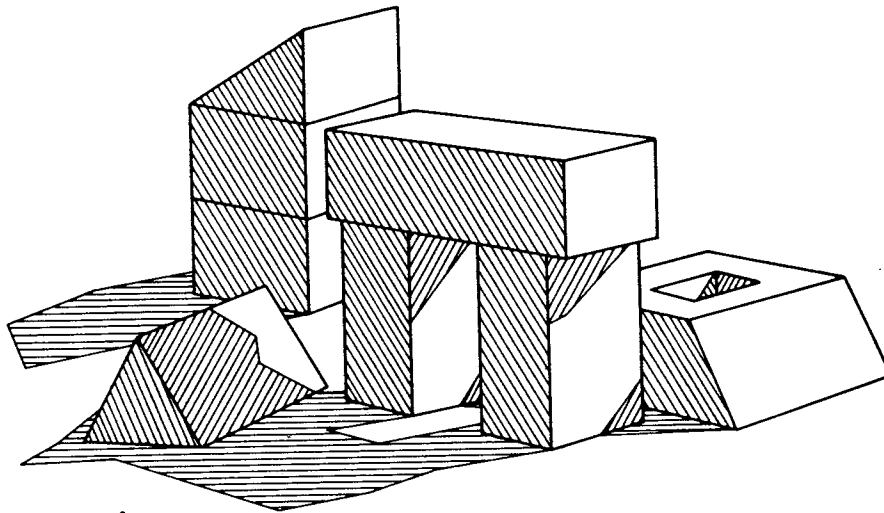


Fig. 2.2

important to have a good language in which to describe features of scenes. Since I wish to have the program operate on unfamiliar objects, the language must be capable of describing such objects. The language I have used is an expansion of the labeling system developed by Huffman¹ in the United States and Clowes² in Great Britain.

The language employs labels which are assigned to line segments and regions in the scene. These labels describe the edge geometry, the connection or lack of connection between adjacent regions, the orientation of each region in three dimensions, and the nature of the illumination for each region (illuminated, projected shadow region, or region facing away from the light source). The goal of the program is to assign a single label value to each line and region in the line drawing, except in cases where humans also find a feature to be ambiguous.

This language allows precise definitions of such concepts as supported-by, in-front-of, behind, rests-against, is-shadowed-by, is-capable-of-supporting, leans-on, and others. Thus, if it is possible to label each feature of a scene uniquely, then it is possible to directly extract these relations from the description of the scene based on this labeling.

2.1.2 Junction Labels

Much of the program's power is based on access to lists of possible line label assignments for each type of junction in a line drawing. Depending on the amount of computer memory available, it may either be desirable to store the complete lists as compiled knowledge or to generate the lists when they are needed. In my current program the lists are for the most part precompiled.

The composition of the dictionary is interesting in its own right. While some junction types require many dictionary entries, others require relatively few. Moreover, in some cases local information about the relative brightness of the surrounding regions and about the directions of the lines may severely limit the number of relevant dictionary entries for any particular junction. In other cases such information has little effect.

Figure 2.3 shows all the junction types which can occur in the universe of the program. The dictionary is arranged by junction type, and a standard ordering is assigned to all the line segments which make up junctions (except FORKs and MULTIs). There is a considerable amount of local information which can be used to select a subset of the total number of junction configurations which are consistent with physical reality.

For example the program can use local region brightness and line segment direction to preclude the assignment of certain labels to lines. If it

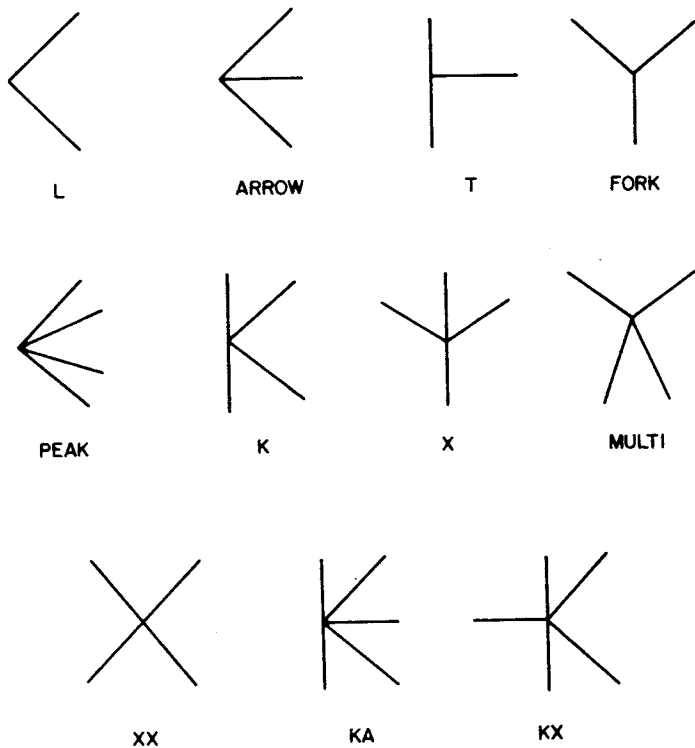


Fig. 2.3

knows that one region is brighter than an adjacent region, then the line which separates the regions can be labeled as a shadow region in only one way. There are other rules which relate region orientation, light placement and region illumination as well as rules which limit the number of labels which can be assigned to line segments which border the support surface for the scene. The program is able to combine all these types of information in finding a list of appropriate labels for a single junction.

2.1.3 Combination Rules

Combination rules are used to select the label, or labels, which correctly describe the scene features that could have produced each junction in the given line drawing. The simplest type of combination rule merely states that a label is a possible description for a junction if and only if there is at least one label which "matches" it assigned to each adjacent junction. Two junction labels "match" if and only if the line segment which joins the junctions gets the same interpretation from both of the junctions at its ends.

I thought at the outset of my work that it might be necessary to construct models of hidden vertexes or features which faced away from the eye in order to find unique labels for the visible features. The difficulty in this is that unless a program can find which lines represent obscuring edges, it cannot know where to construct hidden features, but if it needs the hidden features to label the lines, it may not be able to decide which lines represent

obscuring edges. As it turns out, no such complicated rules and constructions are necessary in general; most of the labeling problem can be solved by a scheme which only compares adjacent junctions.

2.1.4 Experimental Results

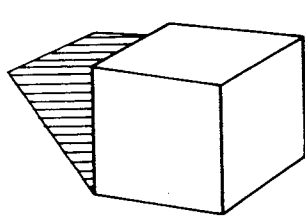
The program computes the full list of dictionary entries for each junction in the scene, eliminates from the list those labels which can be precluded on the basis of local features, assigns each reduced list to its junction, and then a filtering program computes the possible labels for each line, using the fact that a line label is possible if and only if there is at least one junction label at each end of the line which contains the line label. Thus, the list of possible labels for a line segment is the intersection of the two lists of possibilities computed from the junction labels at the ends of the line segment. If any junction label would assign an interpretation to the line segment which is not in this intersection list, then that label can be eliminated from consideration. The filtering program uses a network iteration scheme to systematically remove all the interpretations which are precluded by the elimination of labels at a particular junction.

Initially I had intended to have a tree search program follow the filtering program, but to my amazement I found that in the first few scenes I tried, this program alone found a unique label for each line. Even when I tried considerably more complicated scenes, there were only a few lines in general which were not uniquely specified, and some of these were essentially ambiguous, i.e. I could not decide exactly what sort of edge gave rise to the line segment myself. The other ambiguities, i.e. the ones which I could resolve myself, in general require that the program recognize lines which are parallel or collinear or recognize regions which meet along more than one line segment and hence require more global agreement.

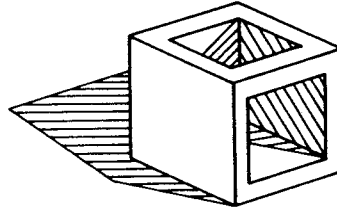
I have been able to use this system to investigate a large number of line drawings, including ones with missing lines and ones with numerous accidentally aligned junctions. From these investigations I can say with some certainty which types of scene features can be handled by the filtering program and which require more complicated processing. Whether or not more processing is required, the filtering system provides a computationally cheap method for acquiring a great deal of information. For example, in most scenes a large percentage of the line segments are unambiguously labeled, and more complicated processing can be directed to the areas which remain ambiguous.

Figure 2.4 shows some of the scenes which the program is able to handle. The segments which remain ambiguous after its operation are marked with stars, and the approximate amount of time the program requires to label each scene is marked below it. The computer is a PDP-10, and the program is written partially in MICRO-PLANNER³ and partially in compiled LISP.

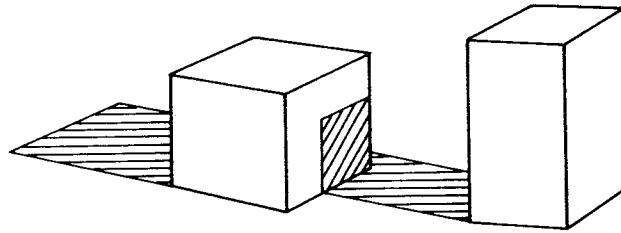
24 The Psychology of Computer Vision



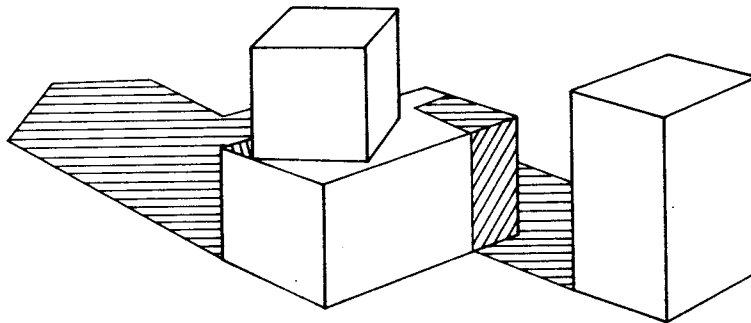
(5 seconds)



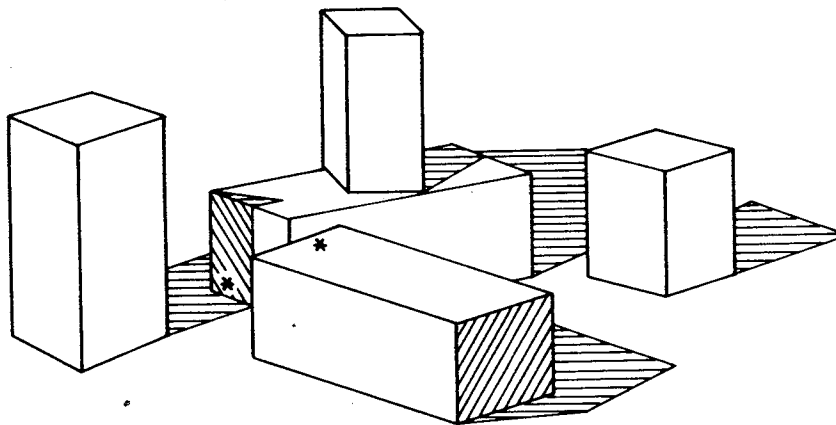
(15 seconds)



(15 seconds)

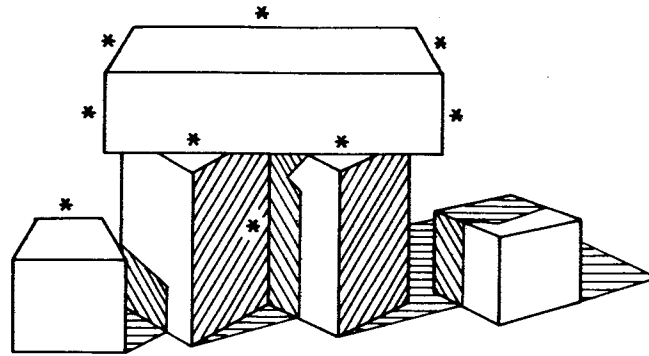


(22 seconds)



(39 seconds)

Fig. 2.4



(48 seconds)

Fig. 2.4 (continued)

2.2 LINE LABELS

In what follows I frequently make a distinction between the scene itself (objects, table, and shadows) and the retinal representation of the scene as a two-dimensional line drawing. I will use the terms vertex, edge and surface to refer to the scene features which map into junction, line and region respectively in a line drawing.

Our first subproblem is to develop a language that allows us to relate these two worlds. I have done this by assigning names called labels to lines in the line drawing, after the manner of Huffman¹ and Clowes.² Thus, in Fig. 2.5 line segment J1-J2 is labeled as a shadow edge, line J2-J3 is labeled as a concave edge, line J3-J14 is labeled as a convex edge, line J4-J5 is labeled as an obscuring edge and line J12-J13 is labeled as a crack edge. Thus, these terms are attached to parts of the drawing, but they designate the kinds of things found in the three-dimensional scene.

Pay particular attention to the notation used to label the lines. When I talk of junction labels I refer to the various possible combinations of such line labels around a junction. Each such combination is thought of as a particular junction labeling.

When we look at a line drawing of this sort, we usually can easily understand what the line drawing represents. In terms of a labeling scheme either (1) we are able to assign labels uniquely to each line, or (2) we can say that no such scene could exist, or (3) we can say that although it is impossible to decide unambiguously what the label of an edge should be, it must be labeled with one member of some specified subset of the total number of labels. What knowledge is needed to enable the program to reproduce such labeling assignments?

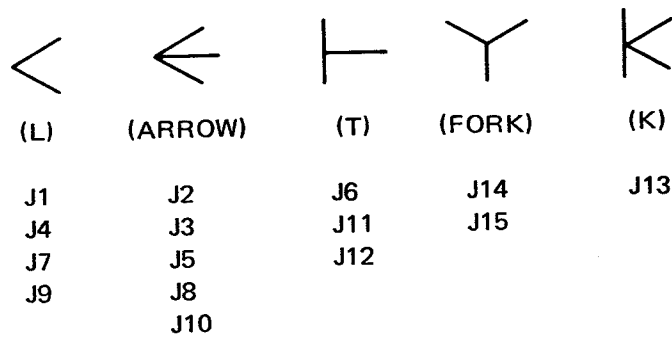
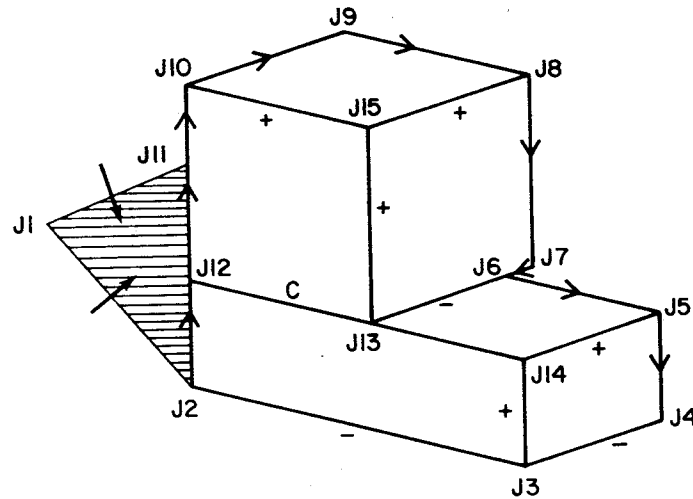
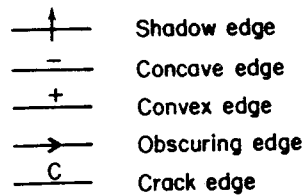


Fig. 2.5

2.2.1 System Knowledge

The knowledge of this system is expressed in several distinct forms:

1. A list of possible junction labels for each type of junction geometry includes the a priori knowledge about the possible three-dimensional interpretations of a junction.
2. Selection rules which use junction geometry, knowledge about which region is the table, and region brightness. These can easily be extended to use line segment directions to find the subset of the total list of possible junction labelings which could apply at a particular junction in a line drawing.
3. A program to find the possible labelings; it knows how to systematically eliminate impossible combinations of labels in a line drawing and, as such, contains implicit knowledge about topology.

4. Optional heuristics which can be invoked to select a single labeling from among those which remain after all the other knowledge in the program has been used. These heuristics find a "plausible" interpretation if required. For example, one heuristic eliminates interpretations that involve concave objects in favor of ones that involve convex objects, and another prefers interpretations which have the smallest number of objects; this heuristic prefers a shadow interpretation for an ambiguous region to the interpretation of the region as a piece of an object.

In this section I show how to express the first type of knowledge and give hints about some of the others. A large proportion of my energy and thought has gone into the choice of the set of possible line labels and the sets of possible junction labels. In this I have been guided by experiment with my program, since there are simply too many labels to hand simulate the program's reaction to a scene. The program, the set of edge labels, and the sets of junction labelings have each gone through an evolution involving several steps. At each step I noted the ambiguities of interpretation which remained, and then modified the system appropriately.

The changes have generally involved (1) the subdivision of one or more edge labels into several new labels embodying finer distinctions and (2) the recomputation of the junction label lists to include these new distinctions. In each case I have been able to test the new scheme to make sure that it solves the old problems without creating any unexpected new ones. For example, the initial data base contained only junctions (1) which represented trihedral vertexes (i.e., vertexes caused by the intersection of exactly three planes at a point in space) and (2) which could be constructed using only convex objects.

The present data base has been expanded to include all trihedral junctions and a number of other junctions caused by vertexes where more than three planes meet.

Throughout this evolutionary process I have tried to systematically include in the lists every possibility under the stated assumptions. In this part of the system I have made only one type of judgement: if a junction can represent a vertex which is physically possible, include that junction in the data base.

Each type of junction (L, ARROW, FORK) can only be labeled in a relatively small number of ways; thus if we can say with certainty what the label for a particular line must be, we can greatly constrain all other lines which intersect that line segment. As a specific example, if one branch of an L junction is labeled as a shadow edge, then the other branch must be labeled as a shadow edge as well.

Moreover shadows are directional, i.e., in order to specify a shadow edge, it must not only be labeled "shadow" but must also be marked to indicate which side of the edge is shadowed and which side is illuminated.

Therefore, not only the type of edge but the nature of the regions on each side can be constrained.

2.2.2 Better Edge Description

So far I have classified edges on the basis of geometry (concave, convex, obscuring, or planar) and have subdivided the planar class into crack and shadow subclasses. Suppose that I further break down each class according to whether or not each edge can be the bounding edge of an object. Objects can be bounded by obscuring edges, concave edges, and crack edges. Figure 2.6

Interpretation

$\frac{R1}{R2} \text{ --- } -$ An inseparable concave edge; the object of which R1 is a part [OB(R1)] is the same as [OB(R2)].

$\frac{R1}{R2} \text{ --- } \leftarrow$ A separable two-object concave edge; if [OB(R1)] is above [OB(R2)] then [OB(R2)] supports [OB(R1)].

$\frac{R1}{R2} \text{ --- } \rightarrow$ Same as above; if R1 is above R2, then [OB(R2)] obscures [OB(R1)] or [OB(R1)] supports [OB(R2)].

$\frac{R1}{R2} \text{ --- } \times$ A separable three-object concave edge; neither [OB(R1)] nor [OB(R2)] can support the other.

$\frac{R1}{R2} \text{ --- } \overset{C}{\rightarrow}$ A crack edge; [OB(R2)] is in front of [OB(R1)] if R1 is above R2.

$\frac{R1}{R2} \text{ --- } \overset{C}{\leftarrow}$ A crack edge; [OB(R2)] supports [OB(R1)] if R1 is above R2.

Separations

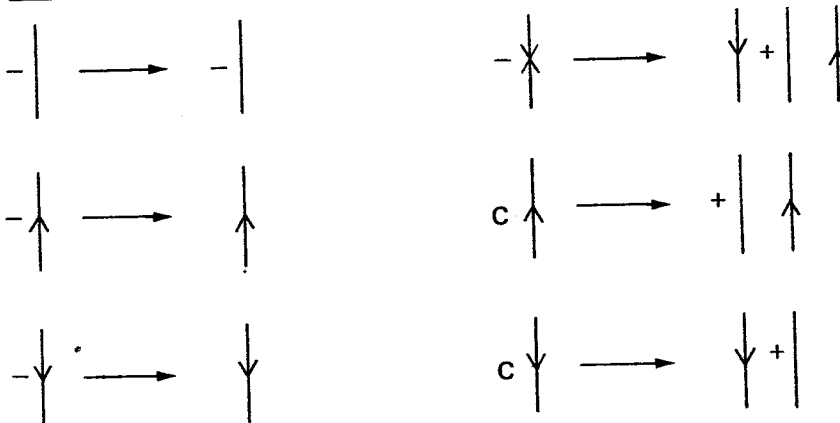


Fig. 2.6

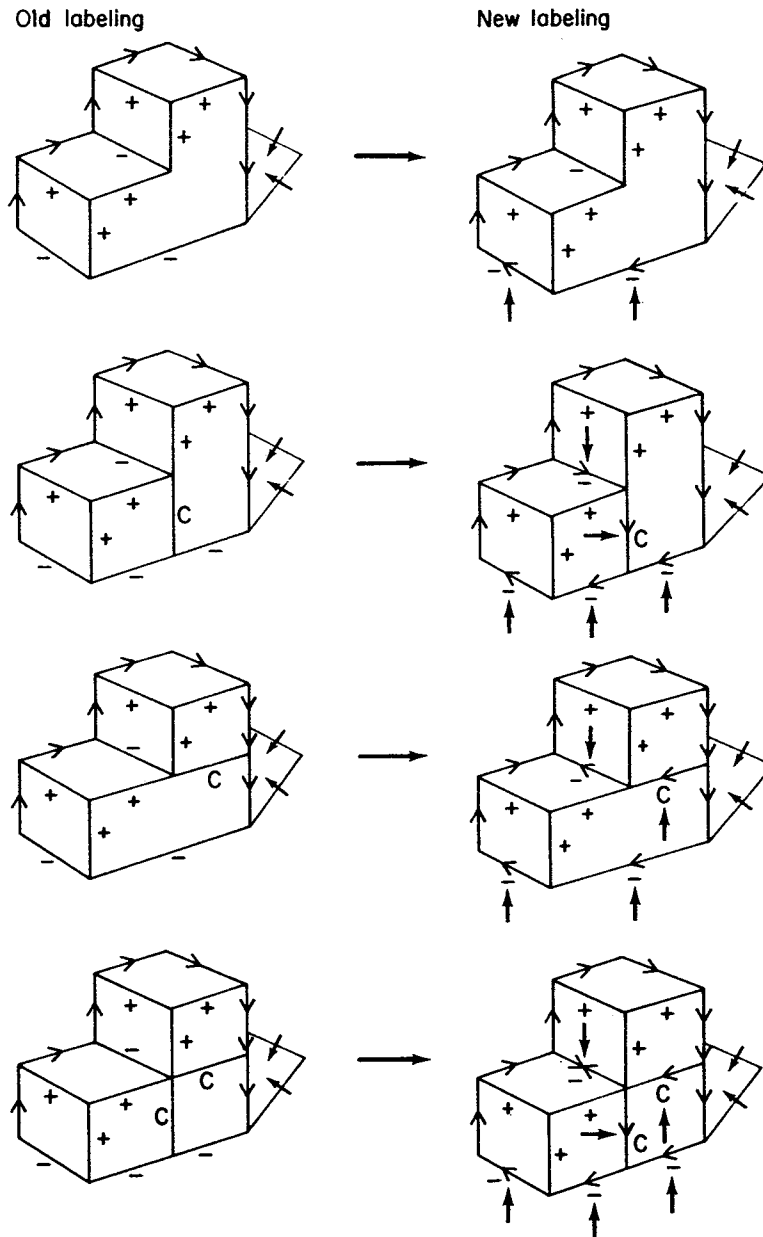


Fig. 2.6 (continued)

shows the results of appending a label analogous to the “obscuring edge” mark to crack and concave edges. This approach is similar to one first proposed by Freuder.⁴

2.2.3 Edge Geometry

The first problem is to find all possible trihedral vertexes. Huffman observed that three intersecting planes, whether mutually orthogonal or not, divide space into eight parts so that the types of trihedral vertex can be

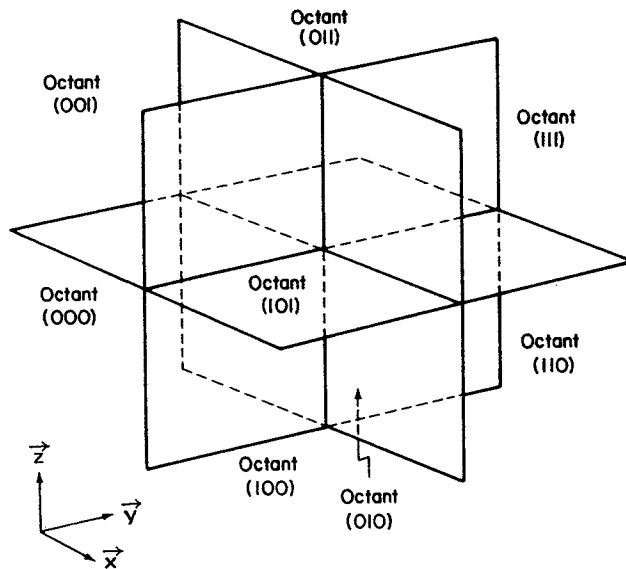


Fig. 2.7

characterized by the octants of space around the vertex which are filled by solid material.¹

Consider the general intersection of three planes shown in Fig. 2.7. These planes divide space into octants, which can be uniquely identified by three-dimensional binary vectors $(x\ y\ z)$ where the x , y , and z directions are specified as shown. The vectors make it easy to describe the various geometries precisely. I can then generate all possible geometries and nondegenerate views by imagining various octants to be filled in with solid material. There are junctions which correspond to having 1, 2, 3, 4, 5, 6, or 7 octants filled. Figure 2.8 shows the ten possible geometries that result from filling various octants; when considered from all possible viewing positions these ten geometries produce 196 different junction labelings. There are some other geometries which I have chosen not to use to generate junction labels. I have not included these geometries because each involves objects which touch only along one edge, and whose faces are nonetheless aligned, an extremely unlikely arrangement when compared to the other geometries. (In addition, some of the geometries are physically impossible unless one or more objects are cemented together along an edge or supported by invisible means.)

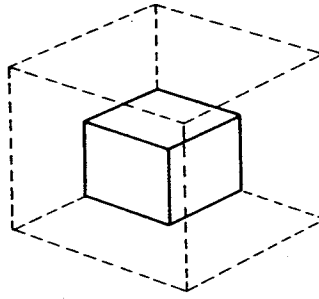
The four geometries recognized by Huffman and Clowes correspond to my numbers 1, 3, 5, and 7 in Fig. 2.8.

In Fig. 2.9 I show how the 20 different labels with type 3 geometry can be generated. Basically this process involves taking a geometry from Fig. 2.8, finding all the ways that the solid segments can be connected or separated, and finding all the possible views for each partitioning of the octants. To generate all the possible views one can either draw or imagine the particular geometry as it appears when viewed from each octant. From some viewing octants the central vertex is blocked from view by solid material, and therefore not every viewing position adds new labelings.

Octants filled

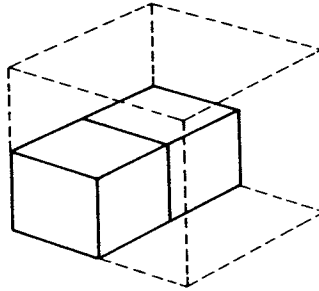
Total number of junction labels

1



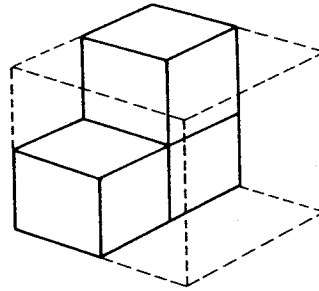
3

2



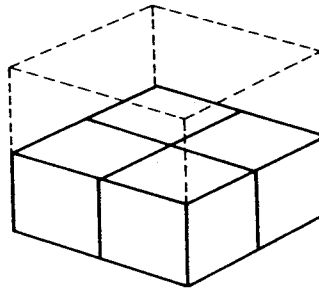
3

3



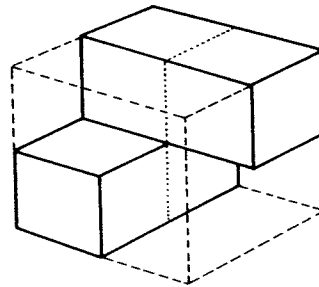
20

4
(Case A)



9

4
(Case B)



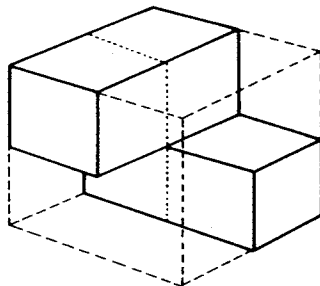
15

Fig. 2.8

Octants filled

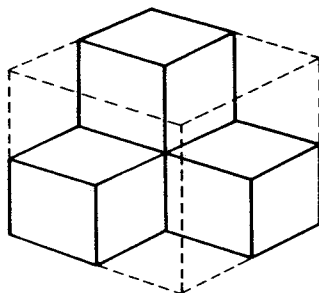
Total number of junction labels

4
(Case C)



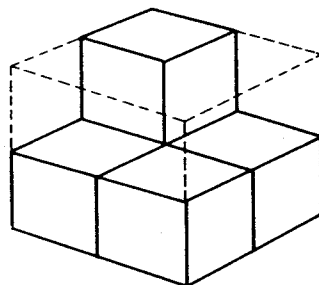
15

4
(Case D)



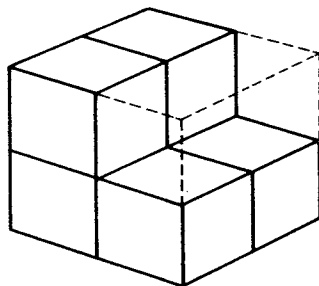
8

5



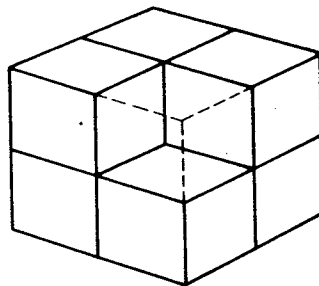
56

6



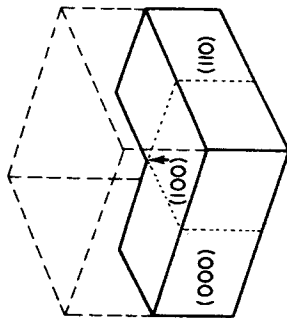
46

7



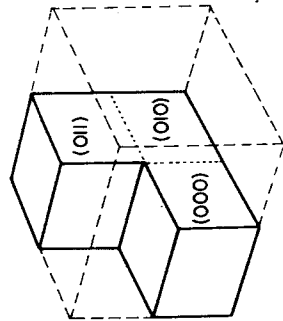
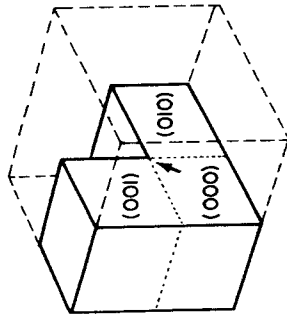
21

Fig. 2.8 (continued)



Name of junction	Appearance and labeling of junction	Number of objects at vertex	Objects at vertex are	Appearance of corresponding object(s)
L-3A		1	$A = (000) \cup (100) \cup (110)$	
T-3A		2	$A = (000) \cup (100)$ $B = (110)$	
T-3B		2	$A = (000)$ $B = (100) \cup (110)$	
XX-3A		3	$A = (000)$ $B = (100)$ $C = (110)$	

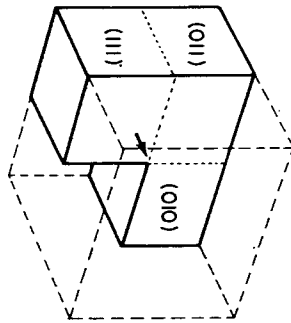
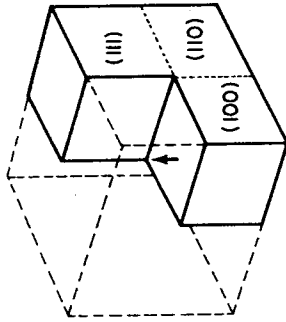
Fig. 2.9



Name of junction	Appearance and labeling of junction	Number of objects at vertex	Objects at vertex are	Appearance of corresponding object(s)
L-3B		1	$A = (001) \cup (000) \cup (010)$	
T-3C		2	$A = (001) \cup (010)$	
T-3D		2	$A = (001) \cup (000)$ $B = (010)$	
XX-3B		3	$A = (001)$ $B = (010)$ $C = (000)$	

Name of junction	Appearance and labeling of junction	Number of objects at vertex	Objects at vertex are	Appearance of corresponding object(s)
Arrow-3A		1	$A = (000) \cup (010) \cup (011)$	
K-3A		2	$A = (010) \cup (011)$ $B = (000)$	
K-3B		2	$A = (011)$ $B = (000) \cup (010)$	
KXX-3A		3	$A = (011)$ $B = (010)$ $C = (000)$	

Fig. 2.9 (continued)



Name of junction	Appearance and labeling of junction	Number of objects at vertex	Objects at vertex are	Appearance of corresponding object(s)
Fork-3A		1	$A = (111) \cup (110) \cup (100)$	
Fork-3B		2	$A = (111)$ $B = (110) \cup (100)$	
Fork-3C		2	$A = (111) \cup (110)$ $B = (100)$	
Fork-3D		3	$A = (111)$ $B = (100)$ $C = (110)$	

Name of junction	Appearance and labeling of junction	Number of objects at vertex	Objects at vertex are	Appearance of corresponding object(s)
L-3C		1	$A = (111) \cup (110) \cup (010)$	
T-3E		2	$A = (111)$ $B = (110) \cup (010)$	
T-3F		2	$A = (111) \cup (110)$ $B = (010)$	
XX-3C		3	$A = (111)$ $B = (010)$ $C = (110)$	

Fig. 2.9 (continued)

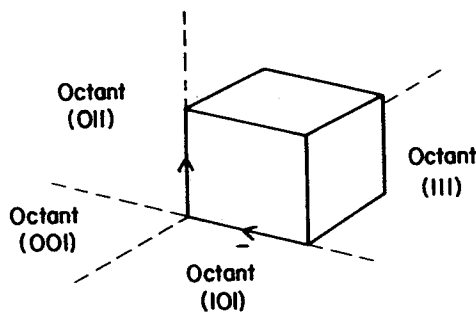
Whenever one of the regions at a junction could correspond to the background it is marked with an asterisk. A noteworthy fact which I will exploit later is that only 37 out of the 196 junction labels can occur on the scene/background boundary.

2.2.4 Shadows at Trihedral Vertexes

To find all the variations of these vertexes which include shadow edges, first note that vertexes with 1, 2, 6, or 7 octants filled cannot cause shadows such that the shadow edges appear as part of the vertex. This can be stated more generally: in order to be a shadow-causing vertex (i.e., a vertex where the caused shadow edge radiates from the vertex) there must exist some viewing position for the vertex from which either two concave edges and one convex edge or one concave edge and two convex edges are visible. Consider the geometries listed in Fig. 2.8. First, a shadow-causing edge must be convex. Second, unless there is at least one concave edge adjacent to this convex edge, there can be no surface which can have a shadow projected onto it by the light streaming by the convex edge. Finally, a junction which has one convex and one concave edge must have at least one other convex or concave edge, since the convex edge and concave edge define at least three planes which cannot meet at any vertex with only two edges.

This immediately eliminates 73 out of 196 of the labelings from consideration. A listing of all the shadow-casting junctions can be constructed in the manner illustrated in Fig. 2.10; for each potential shadow-causing

To find all the shadow possibilities for a junction, first imagine it as part of an object, and define a coordinate system centered at the junction.



Then imagine the light source to be in each of the four octants.

Light in (001)
(no shadow edges
visible at vertex)

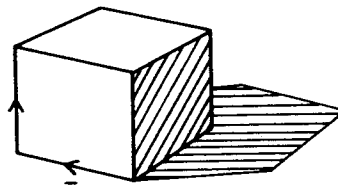
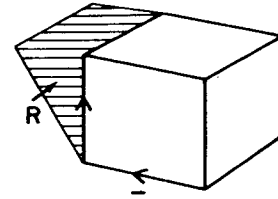
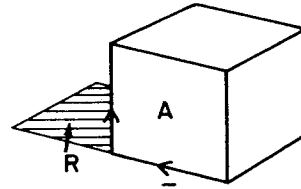


Fig. 2.10

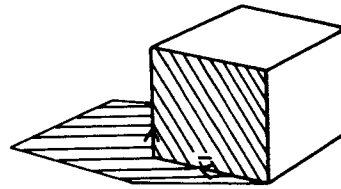
Light in (101)



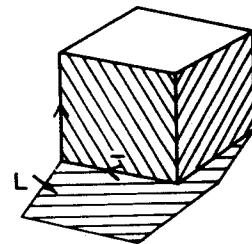
Light on boundary of (101) and (111) (degeneracy; light in plane of A)



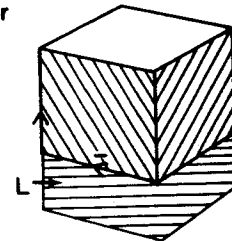
Light in (111)



Light in (011)



or



or

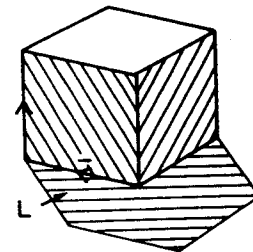


Fig. 2.10 (continued)

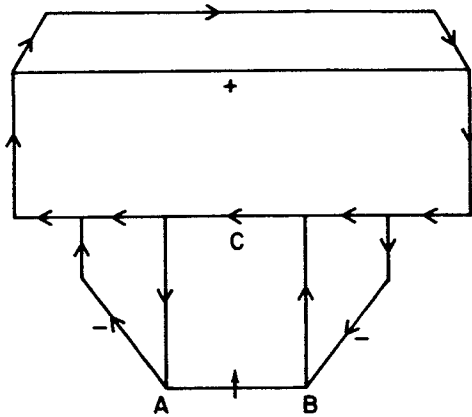


Fig. 2.11

This interpretation is prevented by adding L and R marks to the shadow junctions at A and B respectively so they no longer can match.

vertex, imagine the light source to be in each of the octants surrounding the vertex, and record all the resulting junctions. I have marked each shadow edge which is part of a shadow-causing junction with L or R, according to whether the arrow on the shadow edge points counterclockwise or clockwise respectively.

Any junction which contains either a clockwise shadow edge, marked R, or a counterclockwise shadow edge, marked L, is defined as a shadow-causing junction. The reason for distinguishing between the L and R shadow edges is that this prevents labeling an edge as if it were a shadow caused from both its vertexes. Without this device there would be no way to prevent Fig. 2.11 from being labeled as shown, with line segment L-A-B interpreted as a shadow edge. (I use L- as a prefix to mean "line segment(s) joining the following points"; thus L-A-B is the line segment joining points A and B.) When the L and R marks are attached to each shadow-causing junction, then the two shadow-causing junctions at A and B in Fig. 2.11 are no longer compatible, and therefore the labeling shown will not be considered possible by the program.

2.2.5 Other Nondegenerate Junctions

I now must describe vertexes which do not fall into the categories I have described so far. These include (1) all the rest of the combinations that shadow edges can form and (2) obscured edges.

In Fig. 2.12 I show all the other nondegenerate vertexes which involve shadow edges, and in Fig. 2.13 I show all the obscured edges.

Later I return to the topic of junction labels and show how it is possible to also include junctions representing common degeneracies and accidental alignments as well as junctions with missing lines. In the degenerate cases I do not include every labeling possibility; instead I include the most common

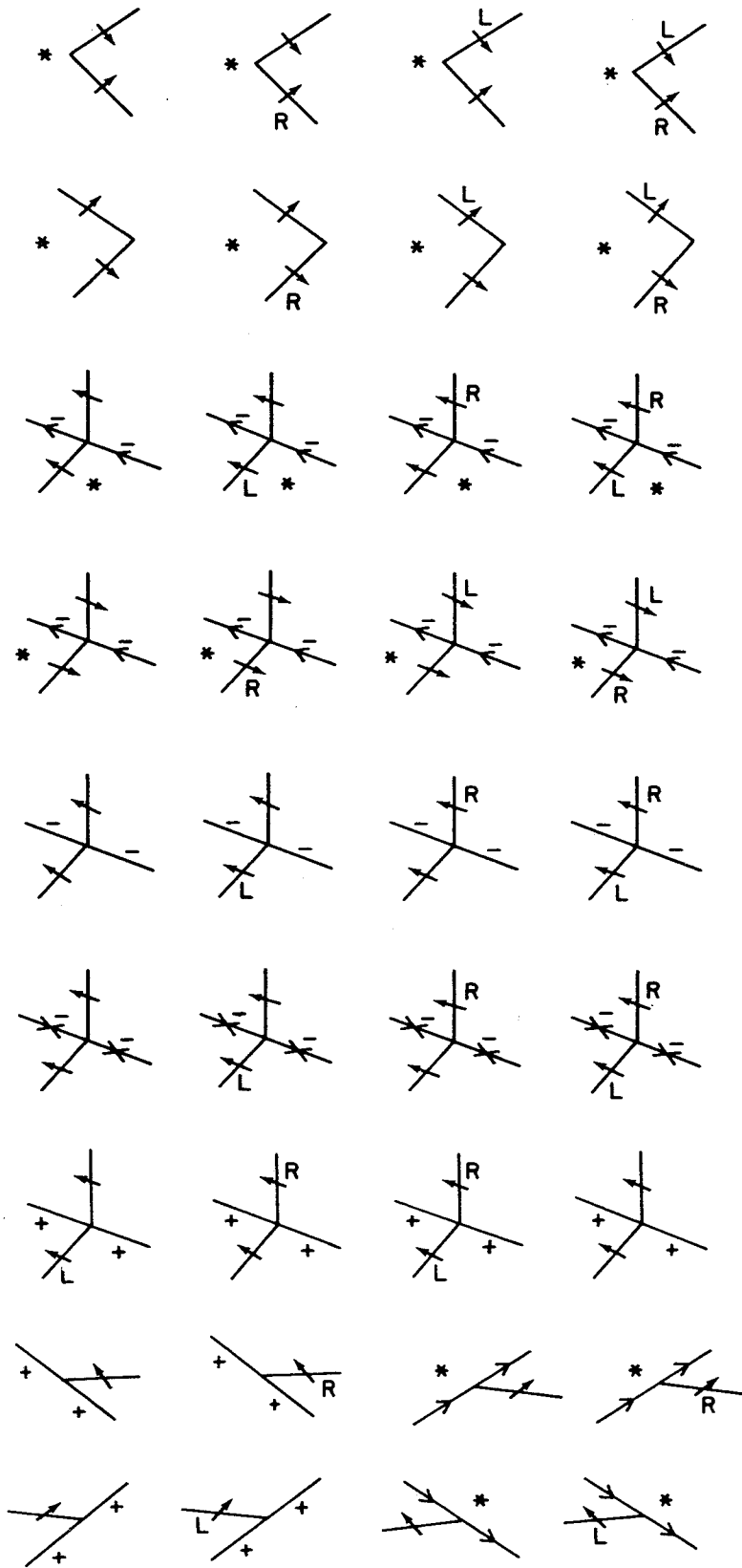


Fig. 2.12

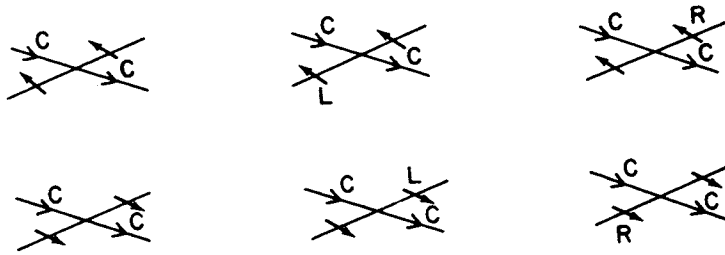
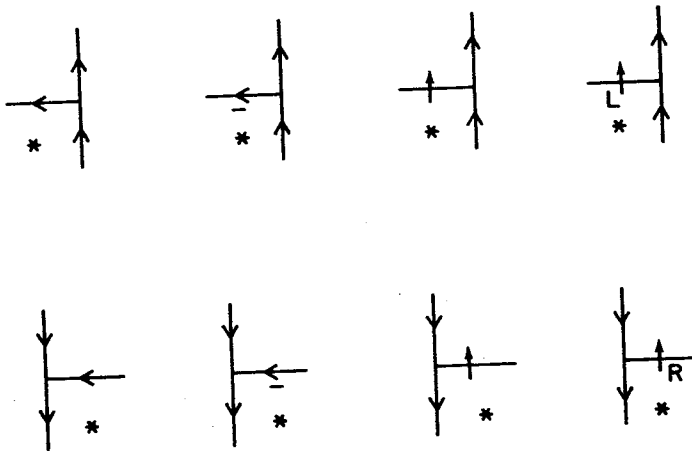


Fig. 2.12 (continued)

occurrences using certain observations about junctions. This is important since I do not want to limit the program to any particular set of objects. Fortunately certain types of junctions are rare no matter what types of objects are in a scene. For example, many junctions can only occur when the eye, light, and object are aligned to within a few degrees; when these

These can occur on the scene/background boundary.



This can occur only in the scene interior.

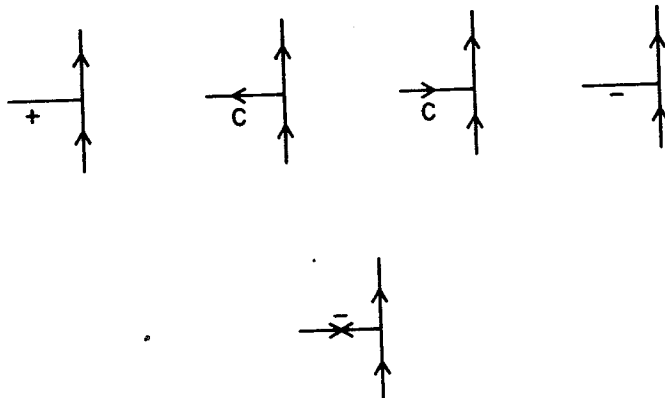


Fig. 2.13

junctions also contain unusual or aligned edges the combined likelihood of the junctions is low enough so that they can be safely omitted. It will be shown that the program can still give information about junctions even if they do not have proper labelings listed in the data base, provided that not too many of these occur together in a single scene. This approach is reasonable since any additional ability to use stereo images or to move the eye or range-finding ability will allow a program to disambiguate most of these types of features.

2.2.6 A Class of Degeneracies

As a final topic, I include one type of degeneracy which cannot be resolved by eye motion or stereo. This type of degeneracy results when the light source is placed in the plane defined by one of the faces of an object. In this case, shadows are aligned with edges to produce junctions which are unlabelable given only the normal set of labels described so far. Two examples of such alignment are shown in Fig. 2.14(a) and (b); all junctions of this type

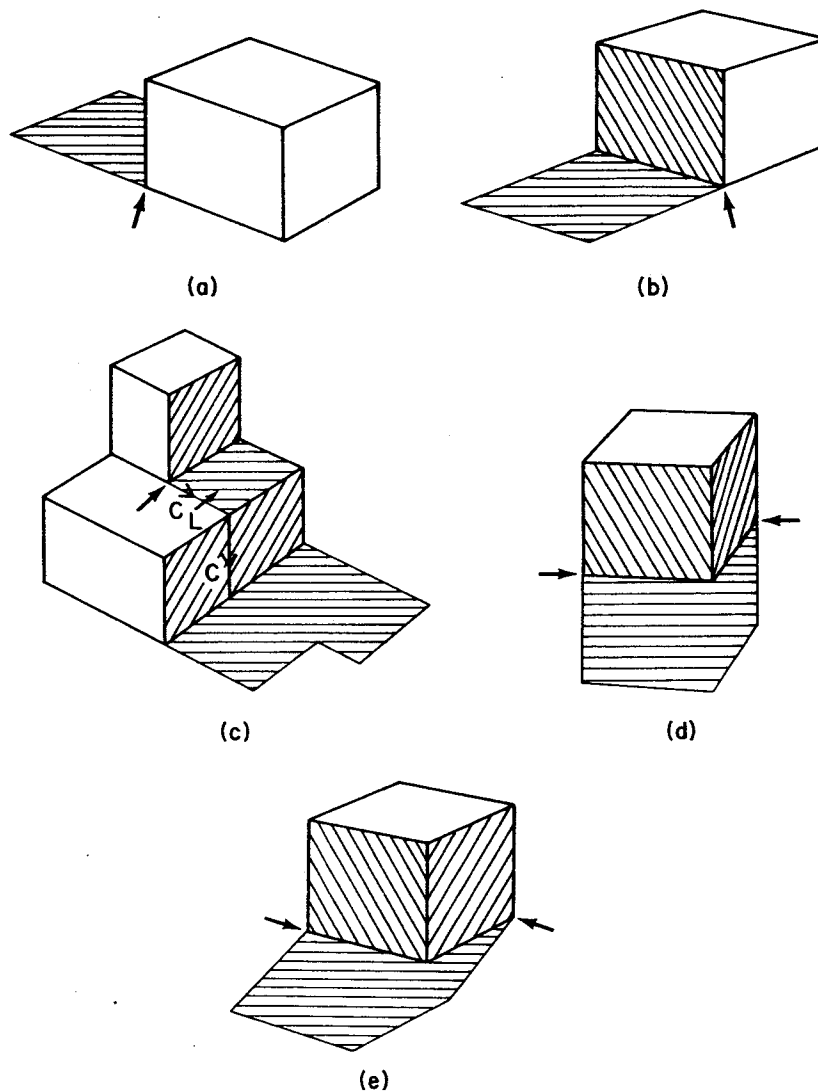


Fig. 2.14

are included in the data base except those cases where a shadow edge is projected directly onto an edge of some other type as in Fig. 2.14(c). These cases are excluded since they would require me to define new edge labels which are of very limited value, although there is no technical difficulty in defining such edges and junctions. I also have excluded, for the time being, cases like the one shown in Fig. 2.14(d), since the two junctions marked only appear to be T junctions when the eye is in the plane defined by the light source and the shadow-causing edge (L-A-B or L-C-D in Fig. 2.14(d)). If the eye is moved to the right, the shadow-causing junctions change to ARROWS or FORKS as illustrated in Fig. 2.14(e). In contrast, notice that for the scenes shown in Figs. 2.14(a) and (b), no change in eye position can make any difference in the apparent geometry of the shadow-causing junctions.

Later I consider some of the common nontrihedral junctions which the program is likely to encounter. Some of these require me to define extra labels.

The grand total number of legal trihedral junctions listed in this section is 505. The interesting thing in my estimation is that the number of junction labels, while fairly large, is very small compared to the number of possibilities if the branches of these junctions were labeled independently; moreover, even

Some common nontrihedral vertexes

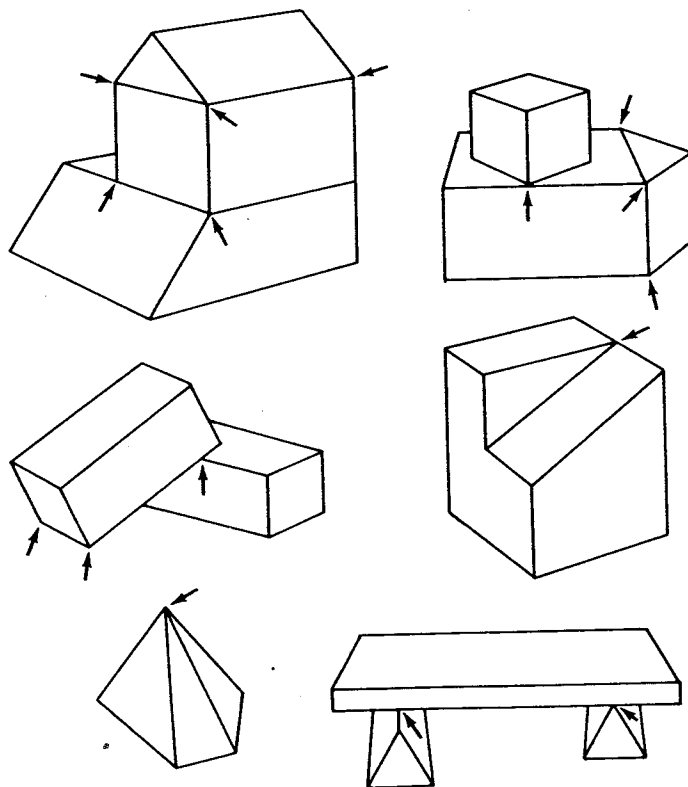


Fig. 2.15

though I have not yet shown how to include various degeneracies and alignments, the set I have described already is sufficient for most scenes which a person would construct out of plane-faced objects, provided that he did not set out to deliberately confuse the program.

Since it may not be obvious what types of common vertexes are nontriheral, Fig. 2.15 contains a number of such vertexes. Later sections show how to handle all of them.

2.3 USING ILLUMINATION

It would be hard to devise a program which could start with a few pieces of information and eventually yield the list of junctions described in the last section. Moreover, even if such a program were written (which would indeed be theoretically interesting), it would be rather pointless to generate labels with it every time the labels are needed in an analysis. Instead the generating program could run once and save its results in a table. In this form the junction labelings table is a sort of compiled knowledge, computed once using a few general facts and methods. The knowledge in the current program is almost totally in this compiled form; this is the reason for its rapid operation, but I have paid a price for this speed in that I require a large amount of memory (about 14,000 words) to store the junction labelings. (All the rest of the labeling program occupies only about 4,000 words of memory even though it is written in MICRO-PLANNER and LISP, neither of which is particularly noted for space efficiency.)

2.3.1 Region Illumination Values

Each region can be labeled as belonging to one of the three following classes:

- I—Illuminated directly by the light source.
- SP—A projected shadow region; such a region would be illuminated if no object were between it and the light source.
- SS—A self-shadowed region; such a region is oriented away from the light source.

Given these classes, I can define new edge labels which also include information about the lighting on both sides of the edge. Notice that in this way I can include at the edge level (a very local level) information which constrains all edges bounding the same two regions. Put another way, whenever a line can be assigned a single label which includes this lighting information, then a program has powerful constraints for the junctions which can appear around either of the regions which bound this line.

Figure 2.16 is made up of tables which relate the region illumination types which can occur on both sides of each edge type. For example, if either side of a concave or crack edge is illuminated, both sides of the edge must be illuminated.

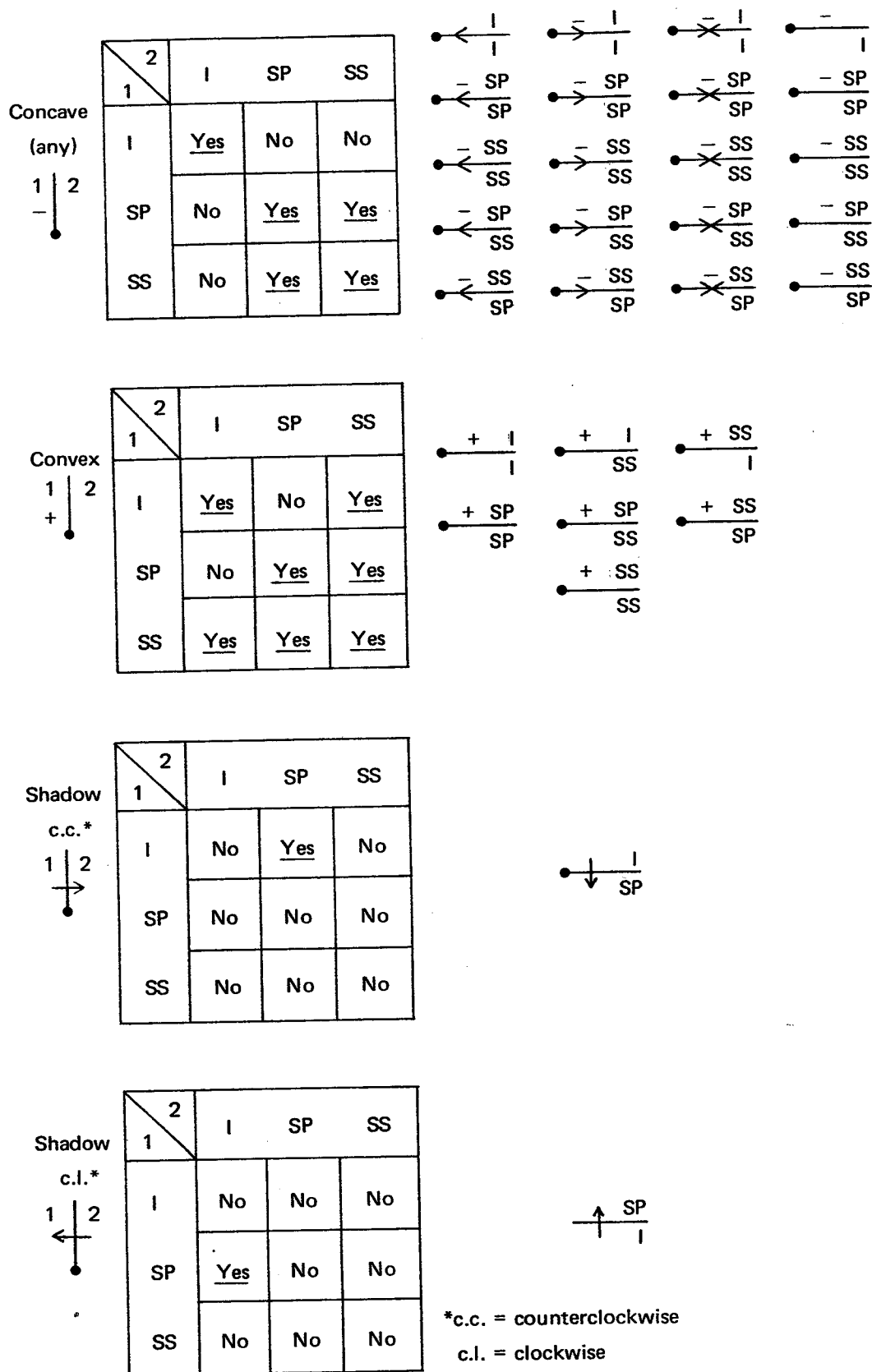


Fig. 2.16

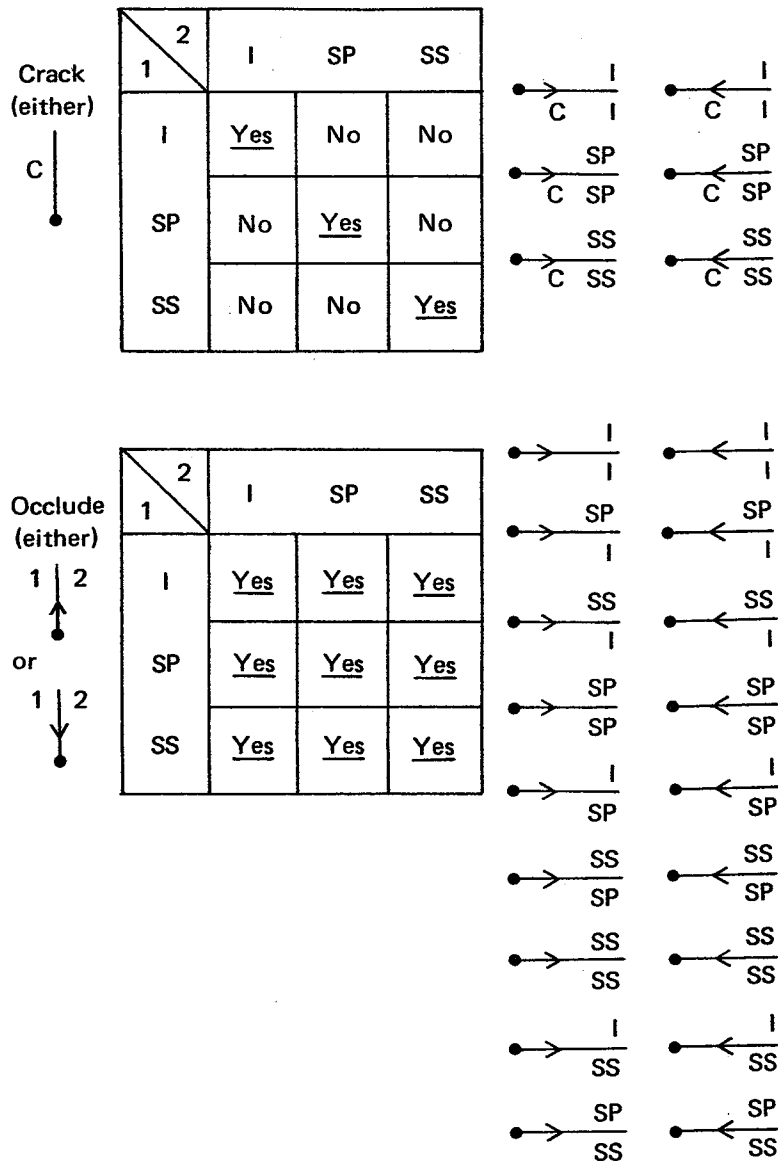


Fig. 2.16 (continued)

These tables can be used to expand the set of allowable junction labels; the new set of labels can have a number of entries which have the same edge geometries but which have different region illumination values. It was very easy to write a program to expand the set of labelings; the principles of its operation are (1) each region in a given junction labeling can have only one illumination value of the three, and (2) the values on either side of each line of the junction must satisfy the restrictions in the tables of Fig. 2.16.

There are two extreme possibilities that this partitioning may have on the number of junction labelings now needed to describe all real vertexes:

1. Each old junction label which has n concave edges, m crack edges, p clockwise shadow edges, q counterclockwise shadow edges, s

- obscuring edges and t convex edges will have to be replaced by $20^n 6^m 3^p 3^q 9^s 8^t$ new junctions, or
- Each old junction will give rise to only one new junction (as in the shadow-causing junction cases).

If (1) were true then the partition would be worthless, since no new information could be gained. If (2) were true, the situation would be greatly improved, since in a sense all the much more precise information was implicitly included in the original junctions but was not explicitly stated. Because the information is now more explicitly stated, many matches between junctions can be precluded; for example, if in the old scheme some line segment L1 of junction label Q1 could have been labeled concave, as could line segment L2 of junction label Q2, a line joining these two junctions could have been labeled concave. But in the new scheme, if each junction label gives rise to a single new label, both L1 and L2 would take on one of the twenty possible values for a concave edge. Unless both L1 and L2 gave rise to the same new label, the line segment could not be labeled concave using Q1 and Q2. The truth lies somewhere between the two extremes, but the fact that it is not at the extreme of (1) means that there is a net improvement. In Fig. 2.17 I compare the number of labels before and after

Total number of labels in data base for each junction type		
Junction type	Number of labels before adding region illumination	Number of labels including region illumination
L	24	92
ARROW	24	86
T	91	623
FORK	116	826
PEAK	10	10
K	42	213
X	129	435
XX	40	128
MULTI	96	160
KA	20	20
KX	60	76
KXX	25	121
SPECIAL	40	466
Totals	717	3,256

Fig. 2.17

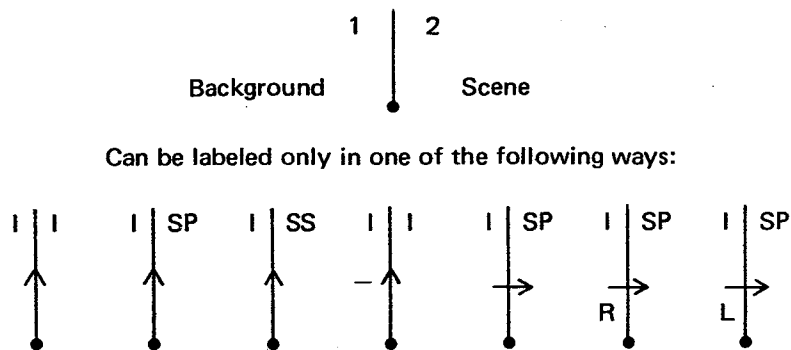


Fig. 2.18

adding region illumination information. Although there are 505 distinct labelings before adding illumination, the actual total number of labels shown in larger. This is because different permutations of labels count as different elements in some of the label lists for the junctions. The total number of list elements needed to represent the 505 labelings is 717, and this number expands to 3,256 when the region illumination information is added to the labelings.

I have also used the better descriptions to express the restriction that each scene is assumed to be on a horizontal table which has no holes in it and which is large enough to fill the retina. This means that any line segment which separates the background (table) from the rest of the scene can only be labeled as shown in Fig. 2.18, Because of this fact the number of junction labels which could be used to label junctions on the scene/background boundary can be greatly restricted.

2.3.2 Labeling Junctions with Illumination

Given tables of allowable region illumination values (Fig. 2.16), it is easy to show how to write a program which expands the data base to include this information.

In order to include illumination information in the data base, I merely append the region illumination value names to the name of each label. Thus I subdivide each label type (except shadow edge labels) into a number of possibilities. Expanding the number of line labels does not increase the total number of junction labels as much as one might imagine. (See Fig. 2.17.) The largest possible number of illumination interpretations for any junction is 3^n , where n is the number of junction branches. A number of T junctions actually have 27 interpretations (for example, this is true of any T made up of three occluding edges).

A little cleverness is required to avoid duplicate labelings when including the different permutations of X junctions. This is because some X junctions give rise to two elements in the X labelings list, while the rest add only one element. Figure 2.19(b) shows an X junction which requires two elements to

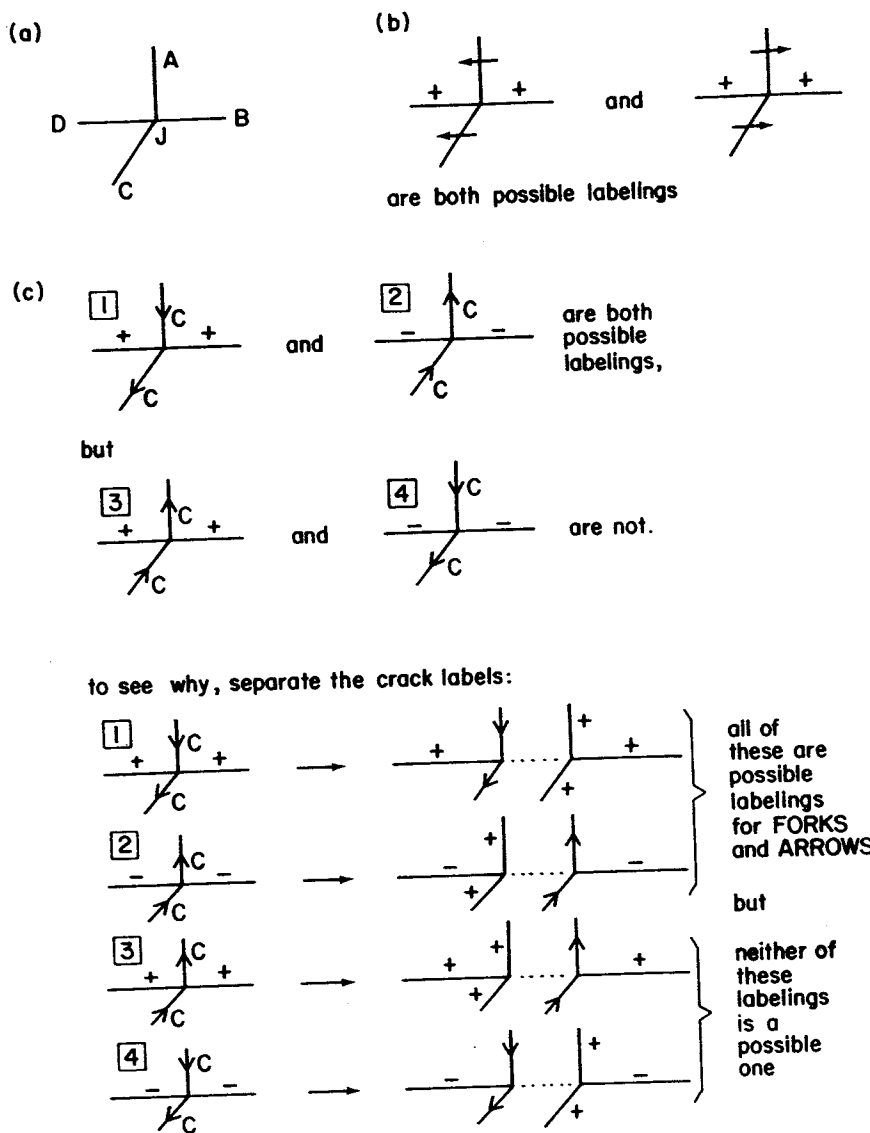


Fig. 2.19

be added to the list, while Fig. 2.19(c) shows two labelings which each add only one element to the data base. Most shadow X junctions give rise to two elements in the data base, and most junctions without shadows give rise to one.

It is now possible to describe how the program handles each junction it encounters:

1. If the junction is an L, ARROW, T, K, PEAK, X, KX, or KXX, it uniquely orders the junction's line segments (by choosing a particular line segment and considering the rest as ordered in a clockwise direction from this line segment).
2. If the junction is a FORK, MULTI, or XX, it chooses one line segment arbitrarily.

3. It then fetches a list of labels which contains every possible set of assignments for the lines (excluding the possibilities of accidental alignments and degeneracies and junctions with missing lines) and associates this list with the junction.

It makes absolutely no difference whether the program obtains this list from a table (the compiled knowledge case) or whether it must perform extensive computations to generate the list (the generated knowledge case). Similarly, it does not matter at all that various members of the list bear a particular relation to each other, e.g., as in the case of a FORK junction, where most elements of the list have two other elements which are permutations of the element. When I return to the issues of degeneracies, accidental alignments and missing lines, all I need to show is how the labelings corresponding to these cases can be added to the appropriate junction lists. The machinery to choose a particular element operates independently of just what the labelings actually are.

2.4 SELECTION RULES

Now that I have shown how to generate a large number of possible labels for a junction, I will show how to go about eliminating all but one of them. The strategy for doing this involves:

1. Using selection rules to eliminate as many labels as possible on the basis of relatively local information such as region brightness or line segment directions.
2. Using the main portion of the program to remove labels which cannot be part of any total scene labeling.

2.4.1 Region Brightness

If I know only that line segment L-A-B is a line in a scene, then it can theoretically be assigned any of the 57 possible labels. Once I know that L-A-B has an ARROW at one of its ends as shown in Fig. 2.20(b), the number of possibilities drops to 19. Suppose that I know, in addition, the relative brightness of R1 and R2 in the neighborhood of L-A-B in Fig. 2.20(c). There are three possibilities:

1. R1 is darker than R2,
2. R2 is darker than R1, or
3. the brightness of R1 is equal to the brightness of R2.

If (1) is true, I know for certain that if L-A-B is a shadow edge, then R1 must be the shadowed side and R2 the illuminated side. Obviously if (2) is true, then the opposite holds, i.e., R2 must be the shadowed side and R1 must be the illuminated side. If (3) is true, then it is impossible for L-A-B to be a shadow edge at all. (If I happen to also know that each object in a scene has all its faces painted identically with a nonreflective finish, then I can also

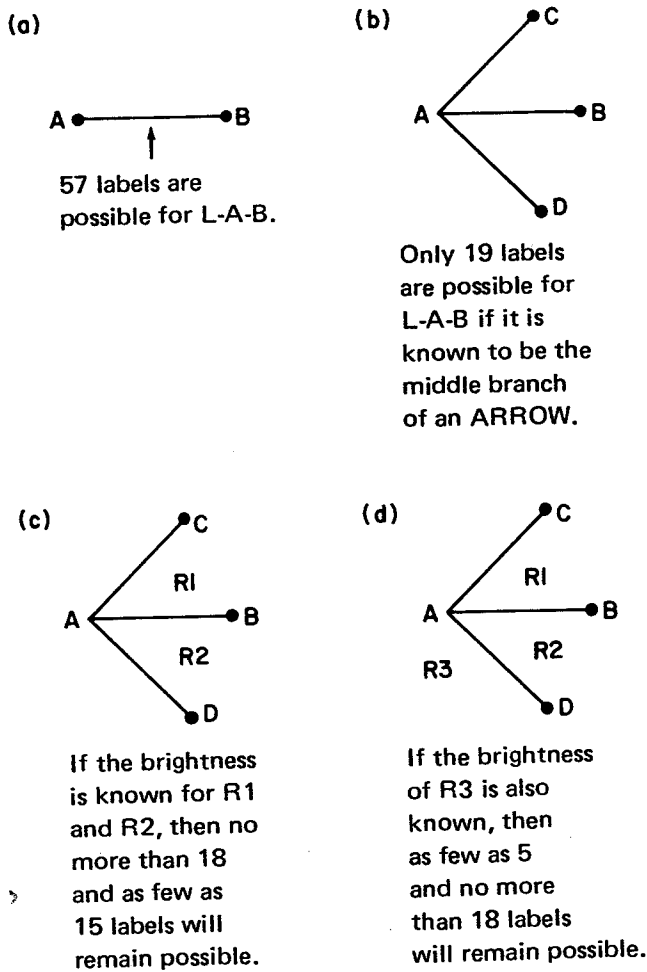


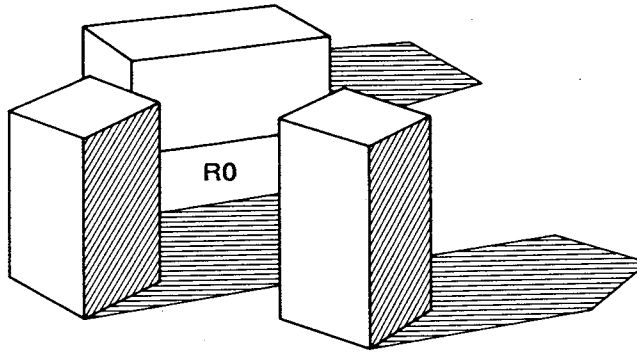
Fig. 2.20

eliminate more labels. In this case, if (1) is true, then L-A-B cannot be labeled as a convex edge with region R1 illuminated and R2 shadowed type SS; if (2) is true, then L-A-B cannot be labeled as convex with R2 illuminated and R1 shadowed type SS, and if (3) is true, then neither of these labels is possible.)

2.4.2 Scene/Background Boundary Revisited

It is easy to find all the junctions which can occur around the scene/background boundary. All that is necessary is to make a list of all the line segments which can occur along the boundary and then look for segments of junctions which are bounded by two members of this set.

Junctions which can occur on the scene/background boundary are listed separately from junctions which have the same geometry but which cannot occur on the scene/background boundary. Thus the list of ARROW labels is divided into ARROW-B, a list made up of those labels which can occur on the scene/background boundary, and ARROW-I, made up of those which



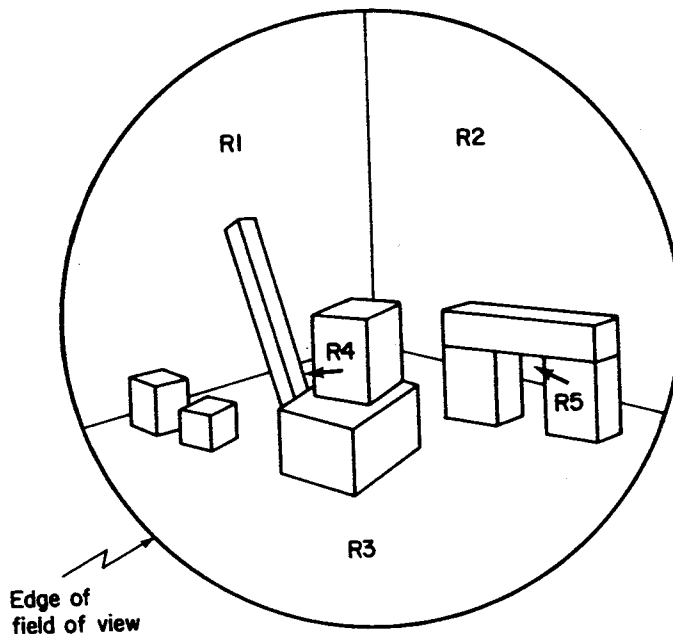
The same junctions and edges can border R0 as can appear on the scene/background boundary.

Fig. 2.21

must occur on the interior of a scene. The total list of junctions which can also appear in the interior of a scene is found by appending ARROW-B to ARROW-I, since the scene/background labelings can appear on the interior of the scene as shown in Fig. 2.21. Figure 2.22 lists the number of trihedral junction labels which can occur on the interior and on the scene/background

Type of junction	Total number of trihedral junction labelings which can appear on	
	The interior of a scene	The scene/background boundary
L	92	16
ARROW	86	12
T	623	96
FORK	826	26
PEAK	10	2
K	213	2
X	435	72
XX	128	3
MULTI	160	8
KA	20	-
KX	76	8
KXX	121	-
SPECIAL	466	-
Totals	3,256	245

Fig. 2.22



By appending all the regions which touch the edge of the field of view, we obtain all of the background except the small regions R4 and R5. By finding and continuing collinear obscured line segments (Guzman's matched Ts) these regions can be found and added to the background also.

Fig. 2.23

boundary for each type of junction. The assumption that the light source is positioned in one of the four octants of space above the support surface guarantees that the background is an illuminated region.

Obviously, if I can determine which lines in the line drawing are part of the scene/background boundary, this knowledge can be used to great advantage. It is, in fact, not difficult to determine this boundary; any of several strategies will work. Two examples are: (1) look for regions which touch the edge of the field of view and append them all together, or (2) find the contour which has the property that every junction lies on or inside it.⁵

Both of these methods require that the scene be completely surrounded by the background region or regions. As shown in Fig. 2.23, method (1) works even if the background is made up of more than one region.

Once the program has found which region is the background region, it can also find how each junction is oriented on the scene/background boundary. Some junctions always appear in the same orientation; for example, ARROW and PEAK junctions can only be oriented so that the background region is the region whose angle is greater than 180 degrees, and K junctions can only have the region whose angle is 180 degrees as the background region.

Of course there is no way to easily define the orientations of FORK, XX, or MULTI junctions. However, as shown in Fig. 2.24, the L, T, X and

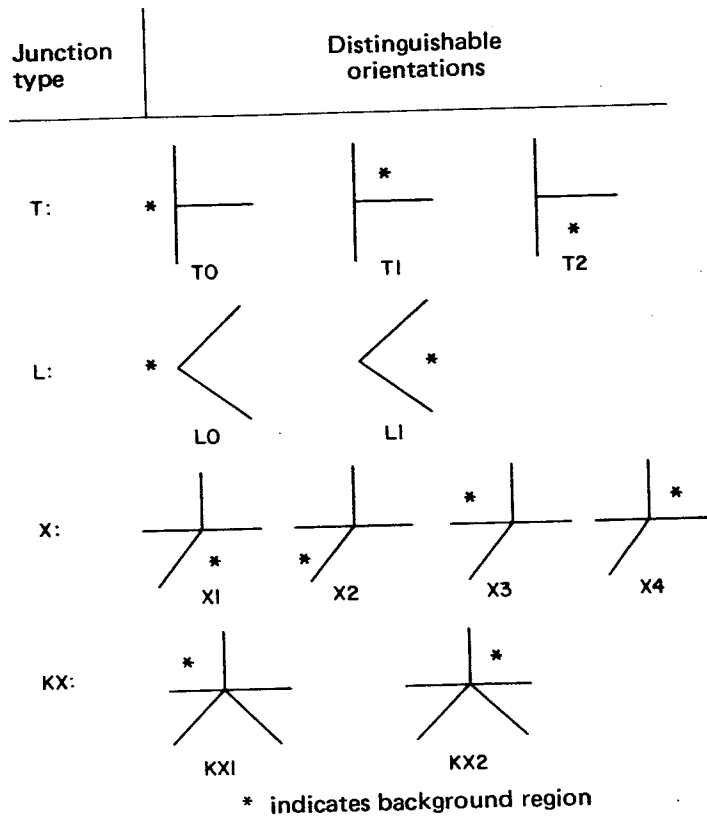


Fig. 2.24

KX junctions which appear on the scene/background boundary can be sorted according to which of their segments is the background region.

Consider Fig. 2.25. Each of the L, T, and X junctions is marked to indicate which orientation it has. Figure 2.26 shows that this distinction makes a significant reduction in the size of the starting list of label assignments for these junctions.

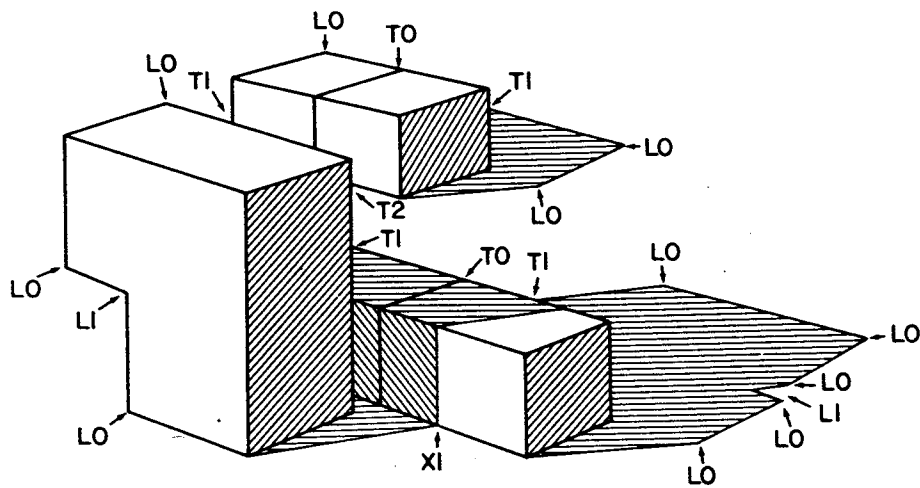


Fig. 2.25

	Number of labelings if in the scene interior	Number of labelings if on the scene/background boundary	Number if on scene/background boundary and orientations distinguished
T:	623	96	—
T0:	*	—	14
T1:	*	—	38
T2:	*	—	38
L:	92	16	—
L0:	*	—	9
L1:	*	—	7
X:	435	72	—
X1:	*	—	8
X2:	*	—	28
X3:	*	—	28
X4:	*	—	8
KX:	76	8	—
KX1:	*	—	4
KX2:	*	—	4

*There is no way to distinguish a preferred orientation in the interior of the scene.

Fig. 2.26

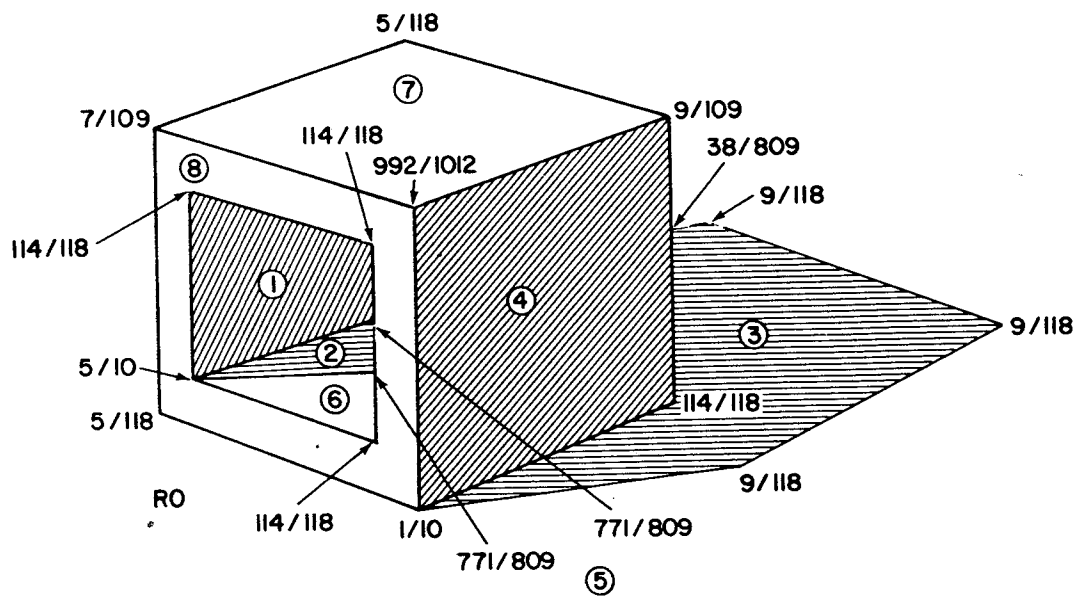


Fig. 2.27

2.4.3 An Example

I have now shown how to use selection rules to narrow down the choices for junction labels on the basis of various kinds of cues from the line drawing. To give an idea of how much these rules help, look at Fig. 2.27. Next to each junction I have listed the numbers of labels which are possible for it before and after applying the selection rules. I have assumed that the program knows that R0 is the support surface and that the circled numbers in each region indicate the relative brightness (the higher a number, the brighter the region). Notice that one junction, the peak on the scene/background boundary, can be uniquely labeled using only selection rules. Most of the interior junctions remain highly ambiguous.

2.5 THE MAIN LABELING PROGRAM

You will recall that I described at some length a "filter program" which systematically removes junction labels whenever there are no possible matches for the labels at adjacent junctions. Now that I have shown a good deal more about the junction labels and the use of the selection rules, I would like to treat this program again from a somewhat different perspective.

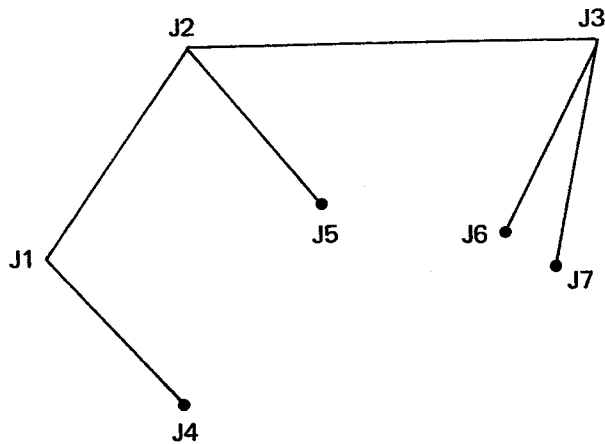
2.5.1 An Example

Suppose that the program is working on a scene, a portion of which is shown in Fig. 2.28. Assume that the selection rules eliminate all labels for each type of junction except those shown at the bottom of the figure. Remember that the selection rules operate only locally, i.e., they give the same list of possibilities no matter how the labeling has proceeded or in what order the junctions are taken. All the step numbers refer to Fig. 2.29, which summarizes the successive lists attached to each junction:

Step 1 Suppose that the program starts with J2, and that all of the other junctions are unlabeled. Then the program assigns list L2 to J2, and since all the other junctions are unlabeled, it has no basis on which to eliminate any of the labels in L2. As far as the program knows, all of these labelings are still possible.

Step 2 Now suppose that it next labels J1 by attaching to it the list L1. When it checks the junctions adjacent to J1 it now can see that J2 has already been labeled.

Step 3 Therefore the program looks at J2 to find what restrictions, if any, have already been placed on line segment L-J1-J2. In this case, the restrictions are that L-J1-J2 must be labeled with either B or C or A or D or F, i.e., with any letter which appears third in an element of L2. Each element of L1 which does not have B, C, A, D, or F as its first letter can then be eliminated. Therefore the program drops (G H), (E A) and (E B) as possibilities and L1 becomes



Results of selection rules for J1. (first element of each labeling refers to L-J1-J2, second to L-J1-J4)

- L1 = ((A B)
- (A C)
- (A D)
- (B B)
- (B E)
- (C F)
- (F A)
- (G H)
- (E A)
- (E B))

Results of selection rules for J2. (first element refers to L-J2-J3, second to L-J2-J5, third to L-J2-J1)

- L2 = ((A B B)
- (A B C)
- (B C A)
- (F A D)
- (D B F))

Results of selection rules for J3. (first element refers to L-J3-J7, second to L-J3-J6, third to L-J3-J2)

- L3 = ((A B A)
- (B C A)
- (G H I)
- (F B C)
- (D B F)
- (A B E)
- (D C E))

Fig. 2.28

((A B) (A C) (A D) (B B) (B E) (C F) (F A))

Step 4 Now the program uses this same reasoning in the opposite direction. In what ways, if any, does the fact that J1 must be labeled from the list restrict the labels of adjacent junctions? Only J2 of the adjacent junctions has been labeled so far, so only J2 can be affected. The only labels which are possible for J2 are those elements of L2 which have as a third letter A or B or C or F. Therefore, the program eliminates (F A D) as a possible label and L2 becomes

((A B B) (A B C) (B C A) (D B F))

Can the program eliminate any other labels because (F A D) has been eliminated? No, since no other neighbors of J2 except J1 have been labeled, and the reason (F A D) was eliminated was because it had no counterpart at J1.

Step 5 The program now can move on to J3 and label it with L3.

Step 6 Each label for J3 must have a third letter equal to one of the first letters from a label in L2. These letters are A, B and D. Therefore the program eliminates (G H I), (F B C), (D B F), (A B E) and (D C G) from L3 and sets L3 to

	Labels assigned to		
	L1	L2	L3
Start	—	—	—
Step 1	—	((ABB)(ABC) (BCA)(FAD) (DBF))	—
Step 2	((AB)(AC)(AD) (BB)(BE)(CF) (FA)(GH)(EA) (EB))	(unchanged)	—
Step 3	((AB)(AC)(AD) (BB)(BE)(CF) (FA))	(unchanged)	—
Step 4	(unchanged)	((ABB)(ABC) (BCA)(DBF))	—
Step 5	(unchanged)	(unchanged)	((ABA)(BCA) (GHI)(FBC) (DBF)(ABE) (DCE))
Step 6	(unchanged)	(unchanged)	((ABA)(BCA))
Step 7	(unchanged)	((ABB)(ABC))	(unchanged)
Step 8	((BB)(BE)(CF))	(unchanged)	(unchanged)
No more labelings can be eliminated			
Final Result	((BB)(BE)(CF))	((ABB)(ABC))	((ABA)(BCA))
↓ Time			

Fig. 2.29

((A B A) (B C A))

Step 7 What labels now are possible for J2? Since the only remaining labels for J3 both set L-J2-J3 to A, the program eliminates (B C A) and (D B F) from L2 so that L2 becomes

((A B B) (A B C))

Step 8 This time, a neighbor of J2, namely J1, has been labeled already, so the program must check to see whether eliminating the element of L2 has placed further restrictions on L1. Only elements of L1 which have a first letter B or C are possible labels now, so the program eliminates (A B), (A C), (A D), and (F A). L1 thus becomes

((B B) (B E) (C F))

Since no other neighbors of J1 are labeled, the effects of this change cannot propagate any further.

2.5.2 Discussion

I think it is easiest to view the process of the program at each junction as having three actions:

1. attaching labels,
2. removing any of these labels which are impossible given the current context of this junction, and
3. iteratively removing labelings from the context by allowing the new restrictions embodied in the list of labels for the junction to propagate outward from the junction until no more changes in the context can be made.

There are two points of importance:

1. The solution the program finds is the same no matter where it begins in the scene, and
2. the program is guaranteed to be finished after one pass through the junctions, where it performs the three actions listed above at each junction.

Given a line drawing with N junctions, a data base which has no more than M possible labelings for any junction, and a situation where any number of junctions from 0 to N have already been labeled, let condition C be one where for each possible line label which can be assigned to a line segment either

1. there is at least one matching line label assigned to the junction at the other end of this line segment, or else
2. the junction at the other end of the line segment has not been labeled.

Condition C must be satisfied before the program moves on to a new junction; the program keeps track of the line segments on which the condition may not be satisfied.

When the program begins labeling a junction J, assume that C holds throughout the line drawing. When the junction, previously unlabeled, has labels added, the only line segments along which C can be violated are the line segments which join J to its neighbors, and it is possible for C to be unsatisfied in both directions on these segments (i.e., both J and J's neighbors may have unmatched line labels). Therefore, to make sure that the program needs to consider each line segment a minimum number of times, the program first uses the lists of possible labels specified by J's neighbors to eliminate all impossible labels from J.

To see why this is the correct way to proceed, suppose that the program used J's initial set of labels to eliminate some labels from one of J's neighbors, J1. It is then possible that the set of labels for J can be reduced further because neighbor J2 has no match for one or more labels still attached to J. The program would then have to go back to line L-J-J1 again to see whether more labels could be eliminated from J1. By considering the effects of each of J's neighbors on J's labels first, the program guarantees that as many labels as possible have been eliminated from J's label list before using this list to recompute the lists for J's neighbors.

Condition C can now only be untrue along line segments joining J with its neighbors and, moreover, can only be untrue in one direction, i.e., J's neighbors may have unmatched labels, but not vice versa. When the program eliminates the unmatched labels from each of J's neighbors, C is now satisfied on each line segment joining J to its neighbors and C can only be unsatisfied along the line segments joining J's neighbors with the neighbors of J's neighbors, and again only in an "outward" direction, i.e., the junctions two line segments away from J can have unmatched labels, but all those junctions one line segment away (J's neighbors) cannot have unmatched labels.

The line segments on which C does not hold continue to spread outward to the neighbors of junctions two segments away from J, then junctions three segments away from J, etc., but only as long as labels are being removed from any junctions. As soon as the program reaches a step where no labels are removed from any junction, then the program knows that condition C must be satisfied everywhere in the scene, and it can move on to the next unlabeled junction.

The violations of C can spread outward to eventually touch any line segment of a line drawing, but only if the number of labels can be reduced at each junction on some path between the junction the program is currently labeling and the line segment.

One final point: the process is guaranteed to terminate, since if there are N junctions and no more than M labels possible for any one junction, the process can never go on for more than $M \times N$ steps at the very worst. This is

important since the restrictions can propagate back to the junction which initiated the process. To see that the possibility of cycles does not create any difficulties, consider the following trick. Suppose that as soon as the starting junction has been checked against each of its neighbors, that all the remaining labels are removed from it. The restrictions can then spread outward only until no more changes can be made; now look at the process as though the junction were being labeled for the first time with the set of junctions just removed as its starting junction set. This process can then be repeated as often as necessary, but the number of times can never be greater than the initial number of labelings assigned to the junction, since the process terminates if no more labels can be removed from the list of possibilities.

2.5.3 Control Structure

While the program can start at any junction and still arrive at the same solution, the amount of time required to understand a scene does depend on the order in which the junctions are labeled. The basic heuristic for speeding up the program is to eliminate as many possibilities as early as possible. Two techniques which help accomplish this end are to

1. label all the junctions on the scene/background boundary first, since these have many fewer interpretations than interior junctions do, and
2. next label all junctions which bound regions that share an edge or junction with the background.

I mentioned at the beginning of this paper that the amount of time (and therefore computation) is roughly proportional to the number of line segments in a scene. This may not seem to fit with the obvious fact that there is really nothing to prevent the effects caused by labeling a single junction to propagate to every portion of a line drawing.

There are good physical reasons why this seldom happens. The basic reason is that some junctions simply do not propagate effects to all their neighbors, and so the effects tend to die out before getting too far. The prime type of junction which stifles the spreading effects is the T junction.

In most T junctions, the labelings of the upright and crossbar portions are independent. Even if we know the exact labeling of the crossbar portion we are unlikely to be able to draw any conclusions about the labeling of the upright and vice versa. Since objects are most commonly separated by T junctions, the effects of labeling a junction are for the most part limited to the object of which the junction is a part and to the object's shadow edges, if any.

Another reason why effects do not propagate far is that when junctions are unlabeled or when they are uniquely labeled, they do not propagate effects at all. Thus when few junctions are labeled and when most junctions are labeled the effects of adding restrictions tend to be localized.

2.5.4 Program Performance

The program portions I have now described are adequate for labeling scenes without accidental alignments, nontriangular vertexes or missing lines. Within this range there are still certain types of features which confuse the program, but before showing its limits, I will show some of its complete successes. In all the scenes which follow, I assume that the program knows which region is the background region, and that it also knows the relative brightness of various regions. The program operates nearly as well without these facts but not as rapidly. Figure 2.30 shows a number of scenes for which the program produces unique labelings or is only confused about the illumination type of one or two regions as in Fig. 2.30(d) and (i). By varying some of the region brightness values or omitting them, the program could also be confused in a similar way about the tops of objects in Fig. 2.30(a), (b), (e), (g), and (h). In general, the program is not particularly good at finding the illumination types

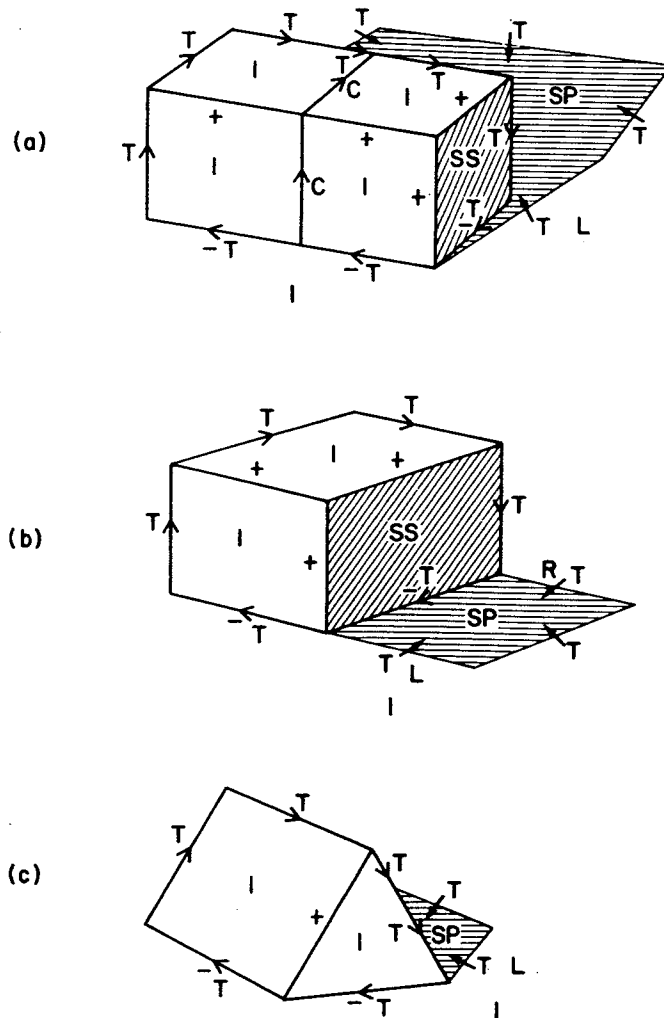


Fig. 2.30

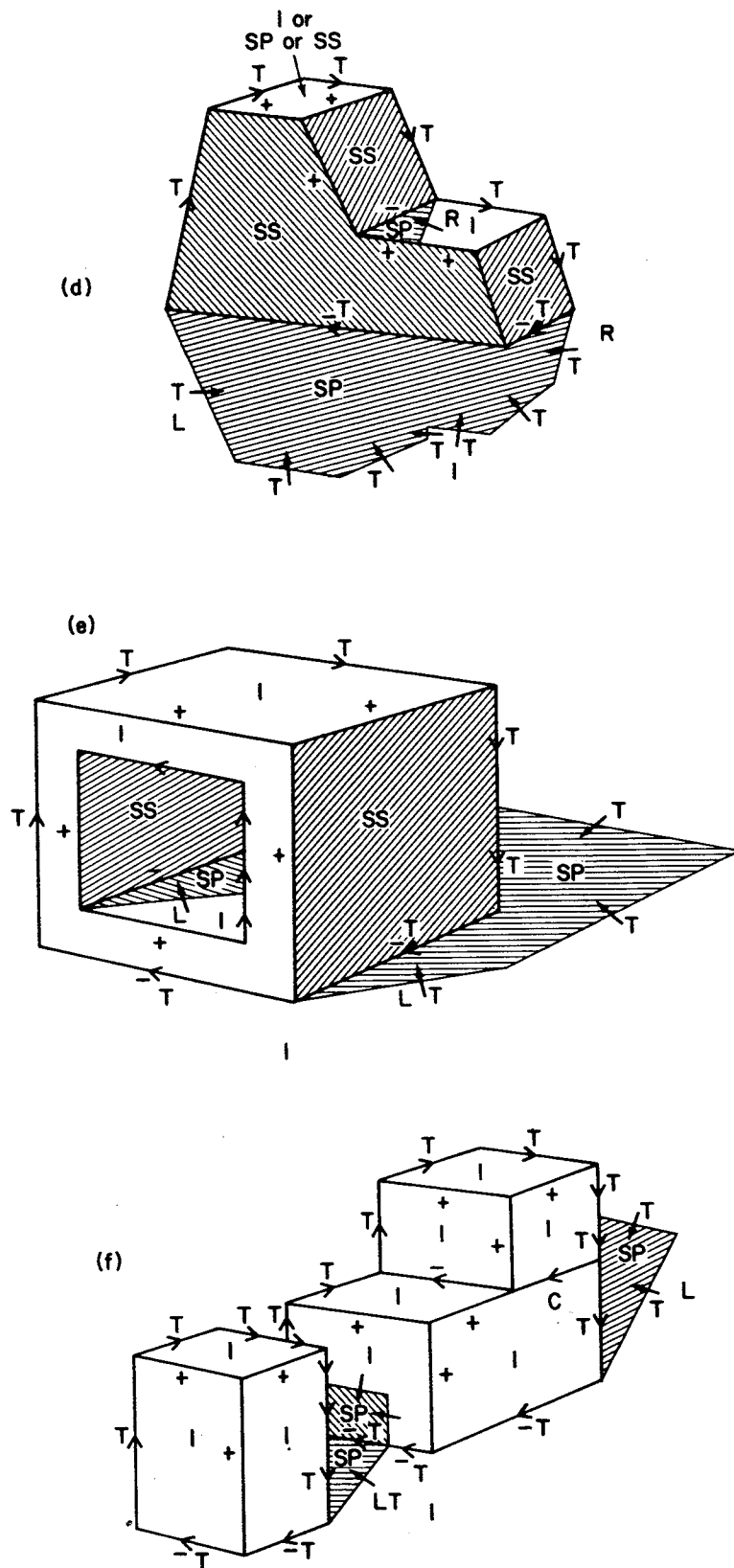


Fig. 2.30 (continued)

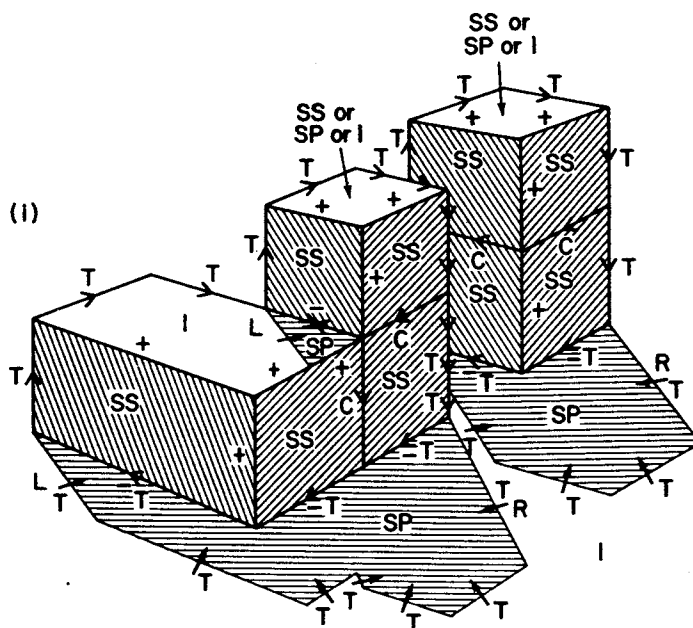
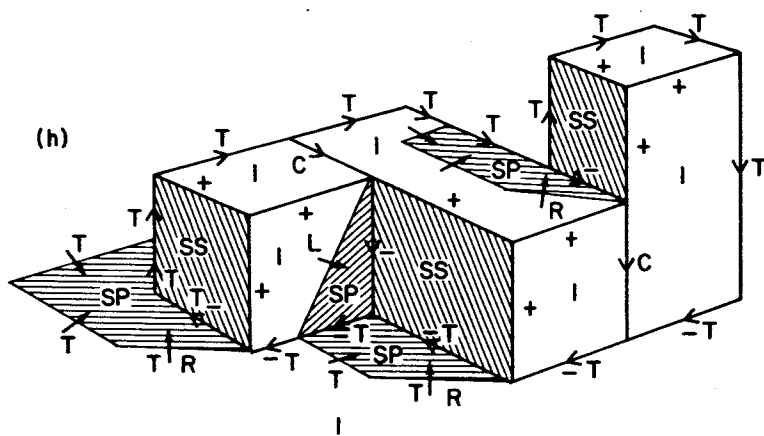
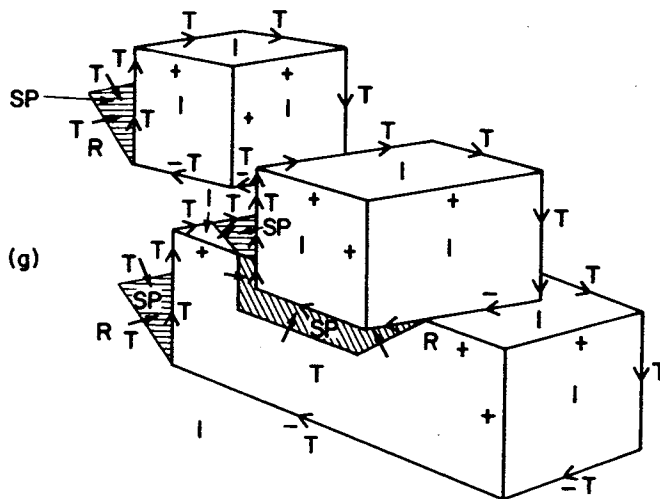


Fig. 2.30 (continued)

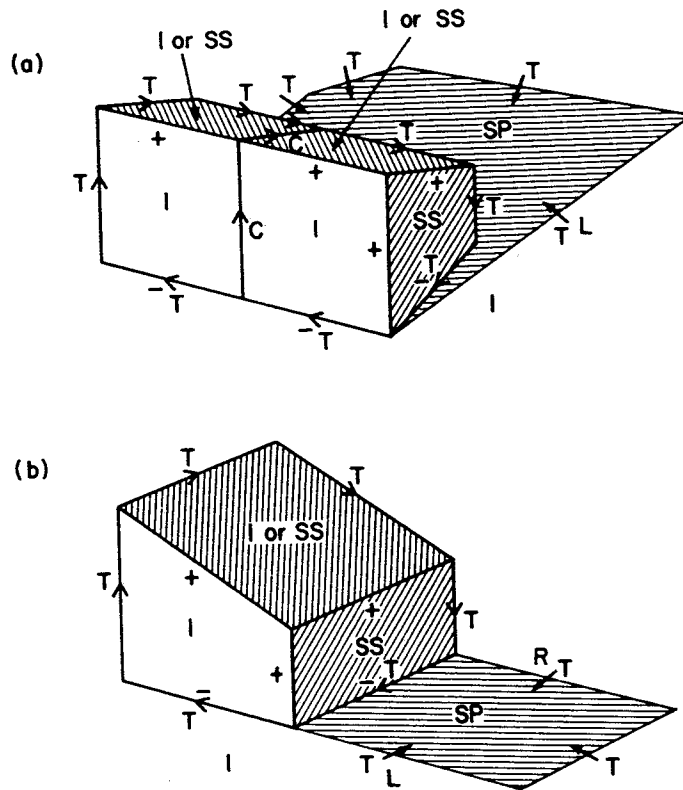


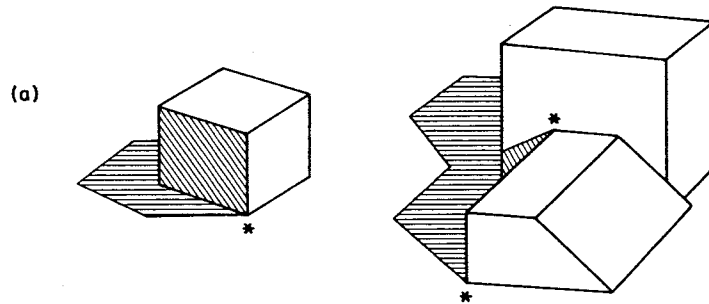
Fig. 2.31

for regions unless the regions are bounded by one or more concave edges. This confusion has a physical basis as well. In all these diagrams I have drawn the top surfaces as though they were parallel to the table so they all should be labeled as type I (illuminated), but since the program I have described so far uses only the topology of the line drawings, it has no way to distinguish the scenes I have drawn from other topologically equivalent scenes which should be labeled differently. For example, in Fig. 2.31 I have redrawn (a) and (b) so that the top surfaces are type SS (self-shadowed), but the figures are topologically identical.

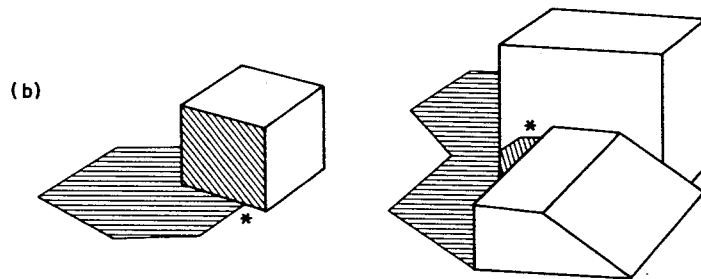
2.5.5 Performance Problems

Shadows convey a considerable amount of information about which edges of an object touch a surface since a shadow edge can only intersect the edge which causes it if the surface the shadow is cast on touches the shadow-causing edge as shown in Fig. 2.32(a). As long as shadows are present, a program can find relations between the objects in a scene and the background, as shown in Fig. 2.32(b). However, if there are no shadows, then it is impossible to decide how the pieces of a scene are related. For example, in Fig. 2.32(c) the block on the left could be stuck to a wall, or sitting on a table, or sitting on a smaller block which suspends it off the table; there is

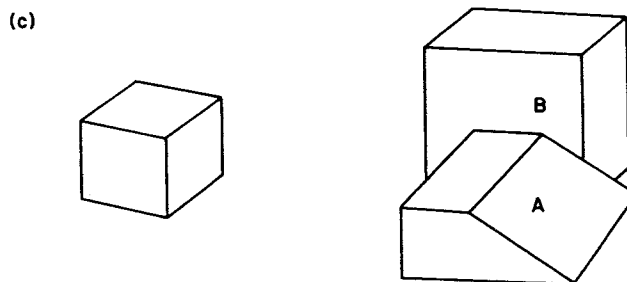
simply no way to tell which of these cases is the true one given only a shadow-free line drawing. Moreover, the program does not use (at this point) knowledge of the line segment directions in a scene, so it cannot even distinguish which way is up. If you turn Fig. 2.32(c) about one-third of a turn clockwise, there is a reasonable interpretation of the two blocks where A is supported by B. Without line segment direction information the program finds all these interpretations in the absence of shadows.



In these scenes the starred junctions provide evidence the two objects or the object and table *touch*:





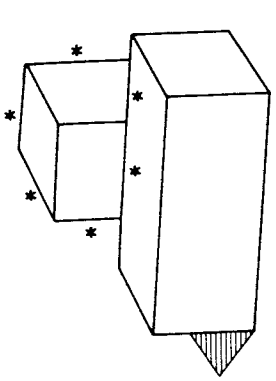
In these scenes the starred junctions provide evidence that the two objects or the object and table *do not touch*.



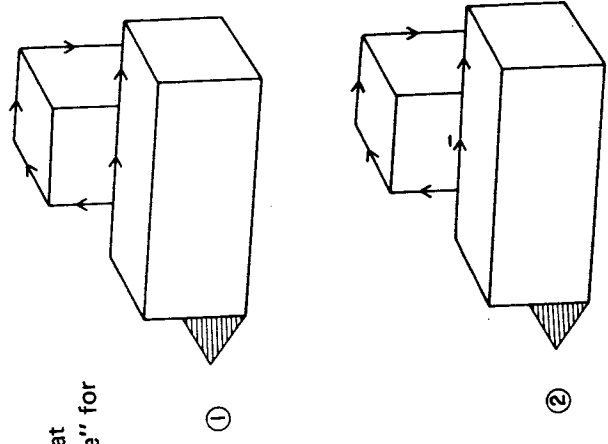
In these scenes there is no evidence to use to relate the objects to each other or to the table; it is not possible to decide whether they touch or not.

Fig. 2.32

(a) Each starred line can be labeled as  or  or all unmarked lines are labeled uniquely.



(b) Two labelings that seem "reasonable" for this scene:



All the remaining "unreasonable" labelings

(c)

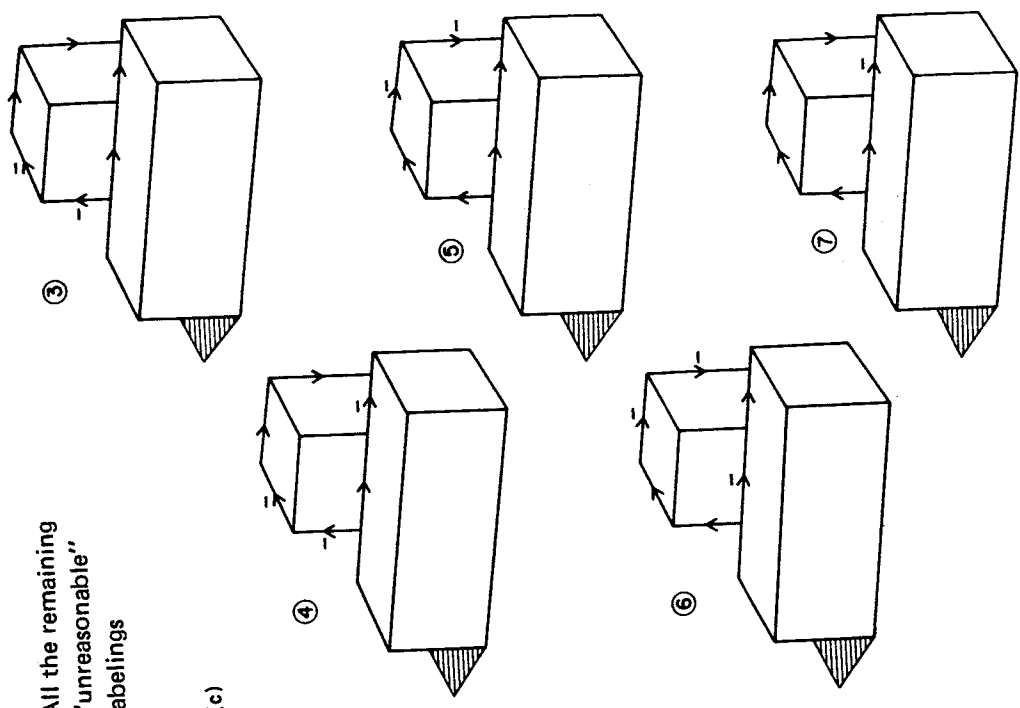


Fig. 2.33

In Fig. 2.33, each of the segments marked with a star can be interpreted either as an obscuring edge or as a concave edge, though in most cases choosing one or the other for some line segment forces other segments to be interpreted uniquely, as shown in Fig. 2.33(b) and (c). To show why the program finds all these labelings as reasonable interpretations, I have constructed the five scenes in Fig. 2.34 to be topologically identical to the scenes in Fig. 2.33(c); this time, however, the labelings shown seem at least plausible if not the most reasonable.

Figure 2.35 shows another problem case. Such a case occurs when we can only see enough of an object so that it is not possible to tell whether the region is a shadow or an object. If it happens that the ambiguous region is brighter than the background (or what would be the illuminated portion of a partly shadowed surface if the feature occurs on the interior of a scene), then the program can eliminate the possibility that the region is a shadow. Unfortunately, if the ambiguous region is darker than its neighbor, the program cannot tell whether the region is a shadow or a dark object. In Fig.

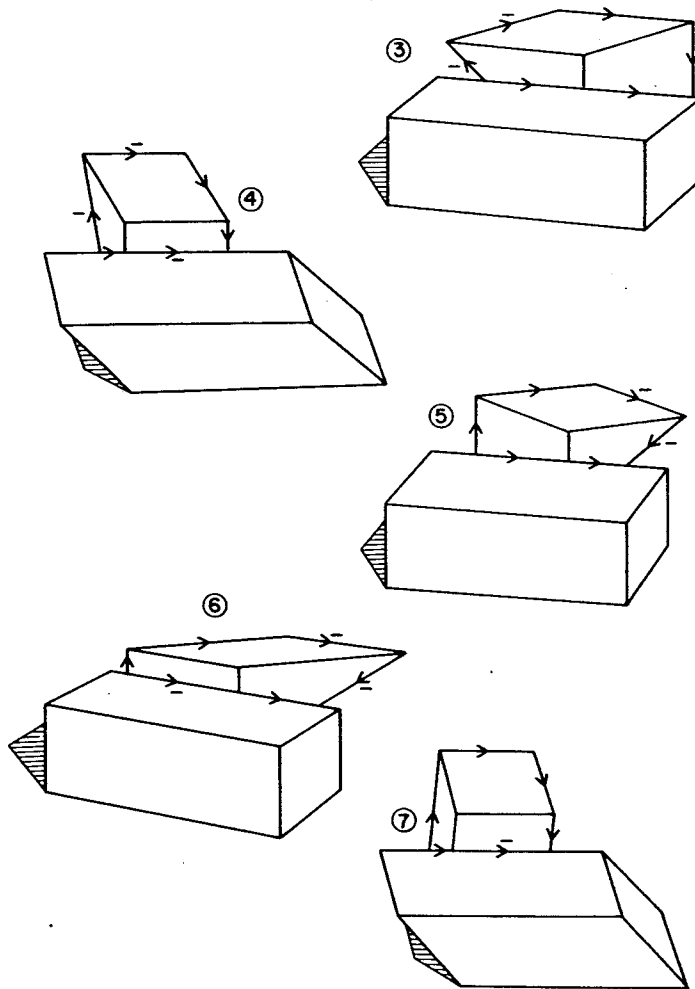
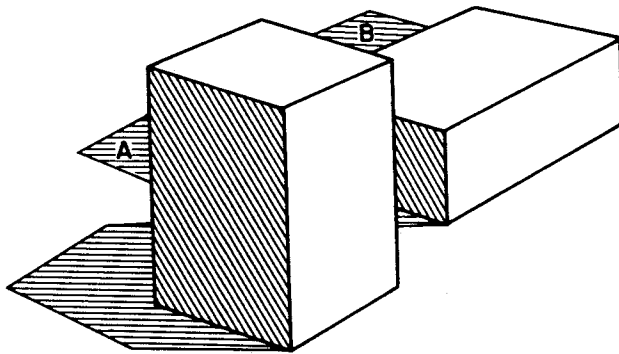


Fig. 2.34

2.35 do you think that both A and B should really be labeled as shadow regions? In fact neither A nor B can be shadows! You can prove this for yourself by finding the characteristic light source slope for the scene, using the front object and its shadow. Then note that there can be no hidden objects which could project A and B.

The labelings which the program finds must be made up of local features, each one of which is physically possible, but it is not obvious that the features which remain should each be part of a total labeling of the scene which is physically possible. After all, the only conditions I impose are that each of these features must agree with at least one other feature at each neighboring junction. On the basis of the fact that the main labeling program does not leave extraneous labels on junctions, it seems clear that topology provides a major portion of the cues necessary to understand a scene.



According to the program A and B can be labeled as follows:

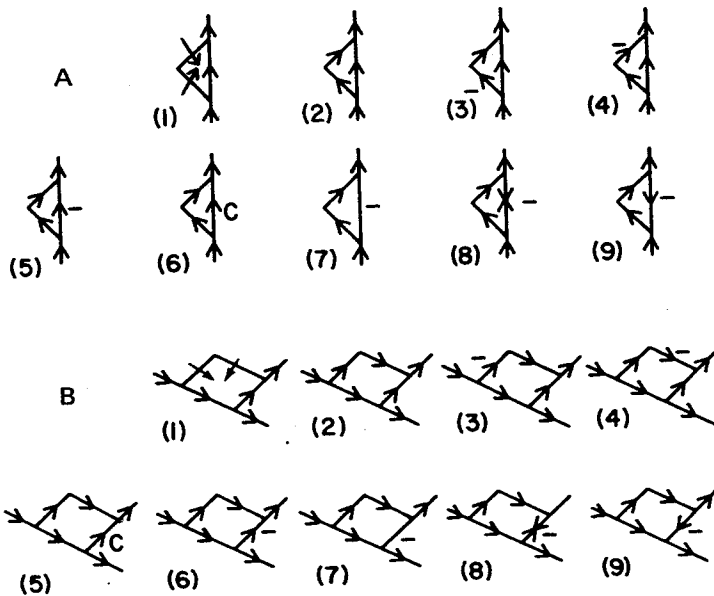


Fig. 2.35

In the next section I show some heuristic rules which can be used to eliminate some of the labelings which people usually consider unlikely. In fact the true case is that these labelings are not unlikely, but the scenes which have these labelings as reasonable ones (to our eyes) do not often arise in our experience. Unfortunately, heuristics sometimes reject real interpretations, and indeed would reject each of the interpretations shown in Fig. 2.34 in favor of the ones in Fig. 2.33(b). Nonetheless, in the absence of solid rules, these heuristics can be useful. In the last section I will deal with techniques which enable a program to find the labelings which we would assign to these line drawings without resorting to heuristics.

2.6 NONTRIHDRAL VERTEXES AND RELATED PROBLEMS

So far I have assumed that all the junctions I am given are normal trihedral junctions and essentially that the line drawing which I am given is "perfect." When a program has to be able to accept data from real line finders and from arbitrarily arranged scenes, these criteria are rather unrealistic.

In this section, I show how to correct some of these problems in a passive manner. By passive I mean that the program is unable to ask a line finding program to look more carefully or to use alternative predicates at a suspicious junction, and similarly that it cannot move its eye or camera, or direct a hand to rearrange part of a scene.

Instead I handle these types of problems by including labels for a number of the most common of these junctions in the regular data base. In cases where the program confuses these junction labelings with the regular labelings and where I want a single parsing, I can easily remove these new types of junction labels first, since I have included special markers for each labeling of this type. Moreover, depending on the reliability of the program which generates the line drawing, I may wish to remove labels in different orders. For example, if a line finding program rarely misses edges, missing edge interpretations can be removed first; if a line finding program tends to miss short line segments, then accidental alignments are probably being generated by the program, and these interpretations can be retained until last. Therefore the labels for each type of problem are marked with different indicators in the data base.

2.6.1 Nontrihedral Vertexes

Some nontrihedral vertexes must be included in the data base; indeed some are much more common than many of the trihedral vertexes. I will limit the number by including only those nontrihedral vertexes which can be formed by convex trihedral objects.

The first type of vertex is formed by the alignment of a vertex with a convex edge as shown in Fig. 2.36(a) and (b). In Fig. 2.36(c) a similar set of

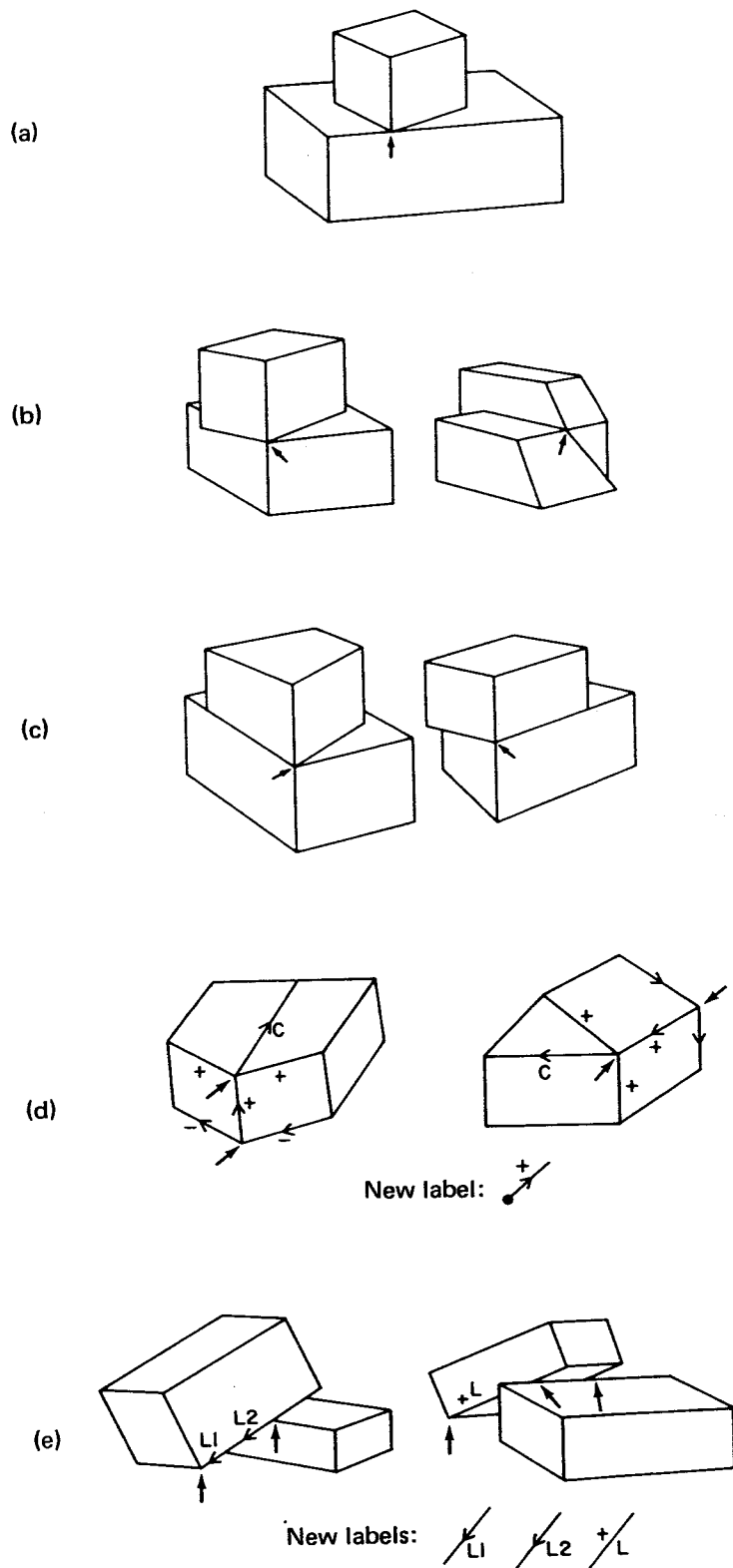


Fig. 2.36

junctions is shown for objects which MARRY (i.e., have coplanar faces separated by a crack edge; see Winston⁶) along one edge, but which have different face angles.

Figure 2.36(d) illustrates another common nontriangular vertex which again results from objects with dissimilar face angles. This time I need a new type of edge (a separable convex edge) labeled as shown in that figure.

Figure 2.36(e) illustrates the types of non-triangular vertexes which can occur when one block leans on another. In order to keep these cases from being confused with other triangular junctions, I have introduced three new edge types. These types only can occur in a very limited number of contexts.

In the data base each of the labelings of the types shown in Fig. 2.36, and any other junction labels involving the leaning edges or the separable convex edges, are marked as nontriangular. Later, if I wish to find a single parsing for a scene where there are still ambiguous labels, removing these nontriangular junctions, if possible, may be a good heuristic.

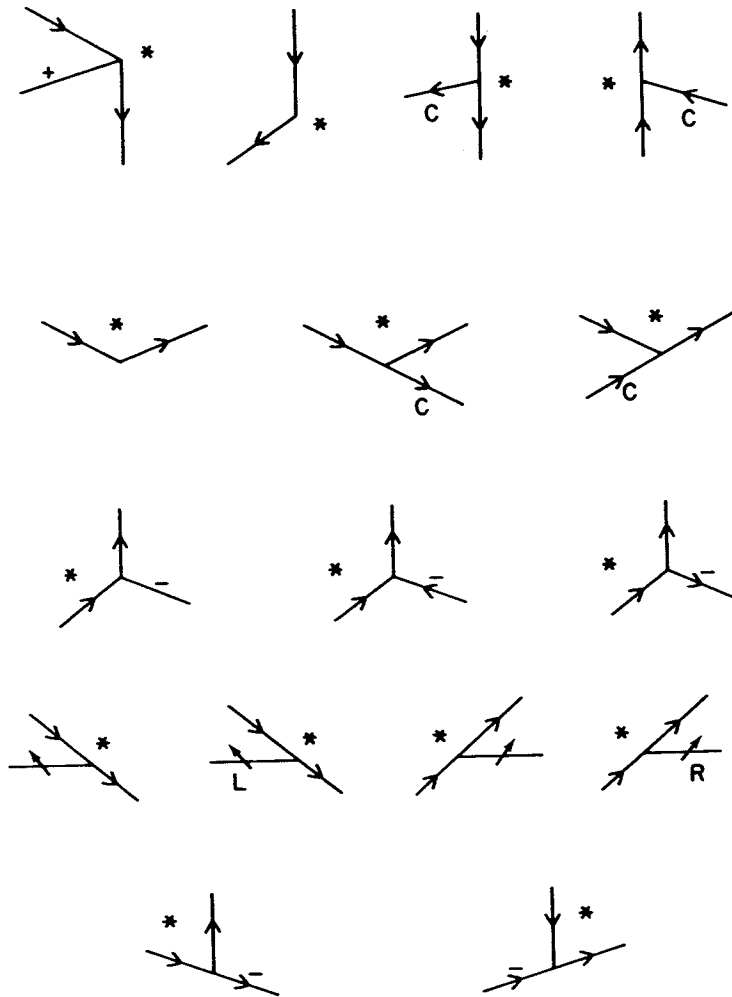
2.6.2 Accidental Alignments (First Type)

In this section I have not attempted to exhaustively list every possible junction labeling which results from accidental alignment, but have concentrated on including only the most common cases. There is some justification for this, in that ambiguities caused by accidental alignments can be resolved by simply moving with respect to the scene.

Figure 2.37 lists all the junctions which can take part in the first type of accidental alignment I will consider. This type of alignment occurs when a vertex is closer to the eye than an edge which appears to be but is not part of the vertex. Thus the set of vertexes in Fig. 2.37 is exactly that subset of the scene/background boundary junctions which contain only obscuring edges on the scene/background boundary. Figure 2.37 shows only those junctions which I include as sufficiently common. The rest are excluded because they involve unusual concave geometries like those found in Soma cube pieces (Soma cubes are the three-dimensional puzzles manufactured by Parker Bros., Inc., Salem, Mass.) or because they involve three-object edges or because the resulting junction would have enough line segments to require a designation of SPECIAL or because the junction would require the alignment of the eye with three points in space.

At this writing, I have not included all these accidental alignment types in the program's data base, but I have included most of the scene/background boundary cases and a number of the interior cases. In general, I have assumed that no nontriangular edges or three-object edges will be among those obscured since both the alignment itself and the edge types are relatively unlikely, so their coincidence at a single junction is extremely unlikely.

Junctions which are used to make up accidental alignment list



(Extra edges are in starred regions)

Fig. 2.37

2.6.3 Accidental Alignment (without Obscuring Edges)

Figure 2.38 shows some alignments which have shown up frequently in scenes I have worked with. These junctions have occurred because (1) our line finding program misses short line segments (and therefore tends to include more lines than it should in a single junction), (2) our line finding program has a tolerance angle within which it will call edges collinear, so some edges are called collinear even when they are not, and (3) edges which lie in a plane parallel to the surface on which they cast shadows are parallel to the shadows they cast, so that alignments become particularly likely when we use bricks, cubes, and prisms.

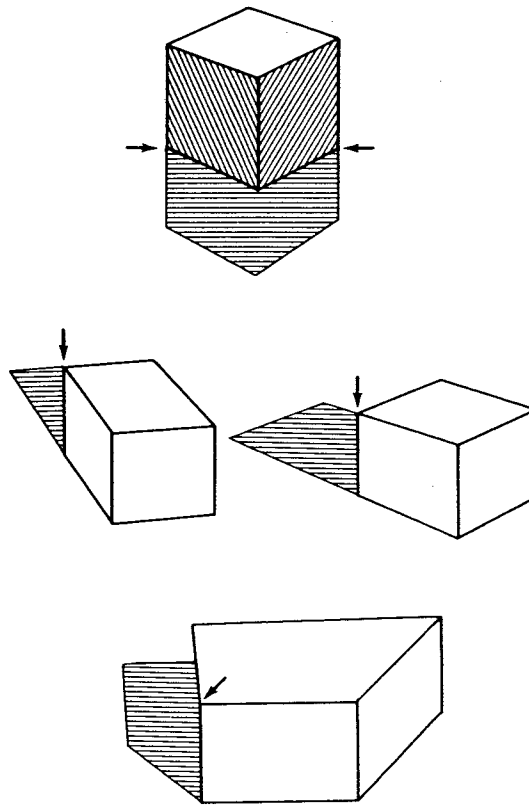
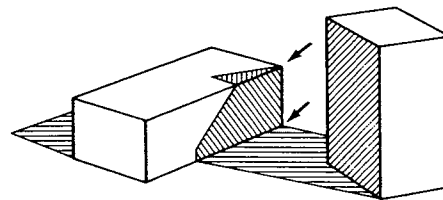


Fig. 2.38



2.6.4 Accidental Alignments (Third Type)

The worst type of accidental alignment, in terms of the number of new junctions it can introduce, occurs when an edge between the eye and a vertex appears to be part of the vertex. Fortunately, all of the types of junctions which these alignments introduce are either K_s , KA_s or $SPECIAL_s$. To see why this is so, look at Fig. 2.39. All these labelings can be quite easily generated by a program which operates on the regular data base. Notice that for each obscured vertex labeling three new labelings are generated since the near region can have any of the three illumination values.

Also notice that any of these junctions which appear on the scene/background boundary can only be oriented with the background in a junction segment type $K1$, $K2$, $K3$, $KA1$, $KA2$, $KA3$, or $KA4$. (See Fig. 2.40.) Therefore it is not difficult to recognize the cases where accidental alignments of this type occur on the scene/background boundary since none of the regular trihedral junctions can ever appear on the scene/background boundary in any of these

If the vertex is	Then the possible accidental alignments with an edge are
L	
ARROW	
FORK	
T	
PEAK	

Fig. 2.39

If the vertex is	Then the possible accidental alignments with an edge are
MULTI	
XX	
X	
K	

Fig. 2.39 (continued)

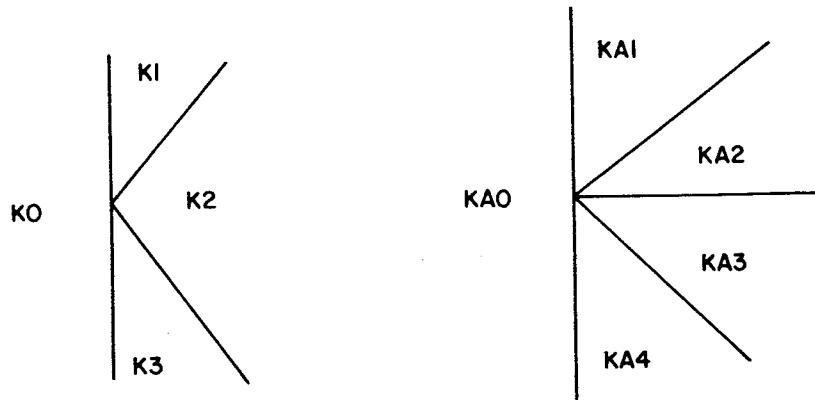


Fig. 2.40 Default condition.

orientations. (The background can only appear normally in segments of type KO or KAO.)

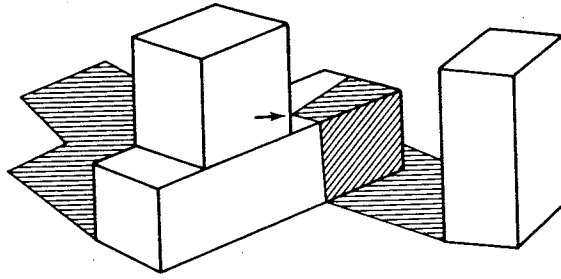
2.6.5 More Control Structure

In this section I return again to the main labeling program and describe what happens when the program is unable to label a scene consistently, using the set of labels with which it has been equipped.

If a junction cannot be labeled from the normal set, instead of marking it unlabelable I generate possible labelings by modifying the line drawing so that it contains equivalent junctions which are not accidentally aligned, and then I label these junctions in the normal manner. Thus, as shown in Fig. 2.41, if the normal set of junctions is inadequate to label a K, the most reasonable alternative is that the junction is actually an obscured L vertex. Therefore I change the line drawing (saving the original of course) and try to label the new line drawing. This change is equivalent to moving the eye slightly to see what type of junction is obscured, except that since the program is unable to move its eye and therefore does not know what the real vertex type is, it keeps trying various alternatives until one works, or until it hits a default case. In the example shown, the program finds a reasonable interpretation on the first try. If it had not, then the program would next have tried to label the junction as an obscured ARROW, since ARROWS are the next most common type of junction after Ls.

This solution is not guaranteed to contain the correct one; the program will be satisfied with the first set of modifications for the K and KA junctions which gives a complete labeling. To be certain of including the correct solution, the program would have to try every combination of interpretations for every K and KA and save all the ones which give complete labelings.

I have lumped a number of junction types together into a default case for two reasons: this lessened the possibility of stopping before getting the desired "correct" solution, and it enabled the program to run much faster and



This K junction cannot be labeled from the normal labelings list for Ks. Therefore the program modifies the line drawing, assuming that the K is really an obscured L, and now the line drawing can be labeled.

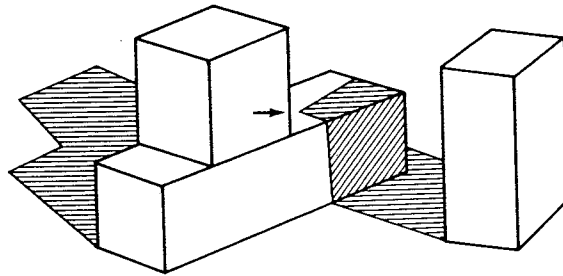
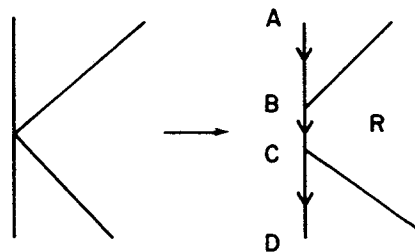


Fig. 2.41

required a much smaller program than would have been needed if I had included separate machinery for each type of junction. The program tries the possibilities for a K in the following order:

1. Try to label the K from the normal label lists.
2. Try to label the K as an obscured L vertex.
3. Try to label the K as an obscured ARROW vertex.
4. If all these fail, label the K as two T junctions. (See Fig. 2.42.)

The default condition represents the exact opposite of the previous conditions. The two Ts result if, instead of moving the eye (by imagination) to see what vertex is behind the obscuring edge, the program moves its eye (by imagination) to completely cover the vertex and eliminate the accidental



Default condition

alignment. Notice that the default condition gives much weaker constraints than could be obtained by trying all the rest of the junction types explicitly. The only relation that must hold for the two T uprights is that the region between them (marked R in Fig. 2.42) have an illumination value which matches both uprights. Nonetheless this is a much stronger condition than is imposed by leaving the junction totally unlabeled and, in addition, the collinear segments (L-A-B, L-B-C, L-C-D in Fig. 2.42) can all be labeled unambiguously as occluding edges. The information I throw away requires that the two uprights be adjacent segments of the same vertex, where this vertex can presumably be labeled from the normal label lists.

2.6.6 Missing Edges

Missing edges usually occur when the brightness of adjacent regions is nearly the same, since most line finding programs depend heavily on steps in brightness to define edges. I have made no attempt to treat missing edges systematically, but have only included a few of the most common cases in the data base. Clearly missing edge junction labels could be systematically generated by a program merely by listing all possibilities for eliminating one edge from each junction label. This procedure would generate $(n - 1) \times$ (old number of regular labels) for each junction type (where n is the number of line segments which make up the junction), and clearly this would be a rather unmanageable number of new labels. The number of new labels could be lessened somewhat by noting that certain types of edges such as cracks are likely to be missed, whereas certain other edges such as shadows are relatively unlikely to be missed.

Even if a program such as mine can recognize that a junction must be labeled as having a missing edge, problems still remain about exactly how the line drawing should be completed. This difficulty is illustrated in Fig. 2.43. Depending on the line segment directions and lengths, the missing edge junction D can be connected to vertex A, vertex B or vertex C, even though the topology of all the line drawings is identical.

The missing edge junctions which are included in the program's data base are all L junctions which result from deleting one of the branches of a FORK junction with three convex edges.

A rule which can be helpful in removing impossible missing edge interpretations is that if a region is bounded by only one junction which can be interpreted as having a missing edge in that region, then that missing edge interpretation is impossible. (There must be another junction to connect with the missing edge.) A similar rule depends on including the label that the missing edge would have had in each missing edge labeling. In this case, the rule is that not only must there be a pair of missing edge junctions around a region in order for either of them to be possible, but this pair must also match in the label that each gives to the missing edge. One final rule is that the previous rules only hold if the pair of missing edge junctions are not

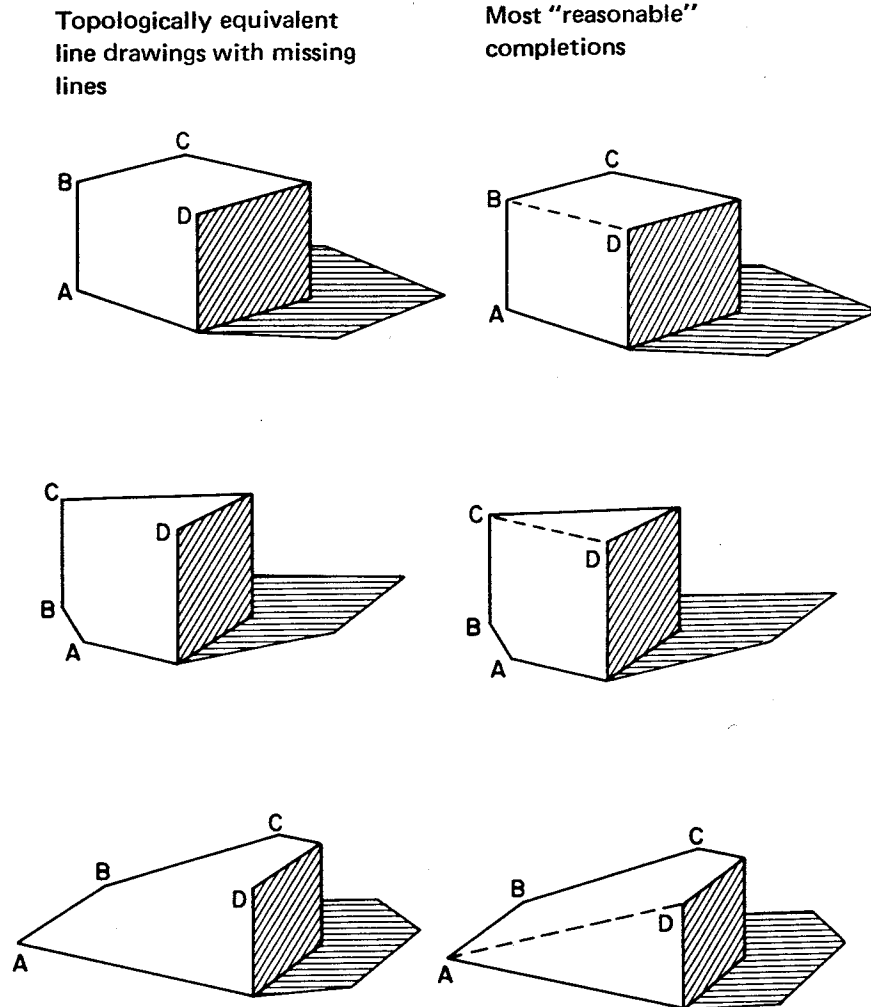


Fig. 2.43

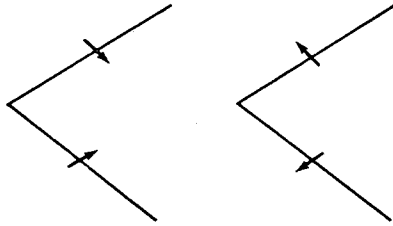
adjacent to one another (i.e. each pair of junctions can be connected by only one straight line).

2.6.7 Heuristics

As I have mentioned earlier in several places, the program is able to remove junction labels selectively according to a crude probability measure of the relative likelihood of various individual feature interpretations. These heuristics are a poor substitute for foolproof rules; in essence I view the heuristics as an expedient method for handling problems I have not yet been able to solve properly. As I explained earlier, these heuristics may nonetheless be of considerable value in guiding programs which find sound solutions.

There is not much to say about the heuristics themselves. The ones I am using currently lump all the "unlikely" junction labels into one class, the "likely" ones into another, and simply eliminate all the "unlikely" labels as long as there are "likely" alternatives.

(a) Shadow L junctions



(b) Contact ARROW junction

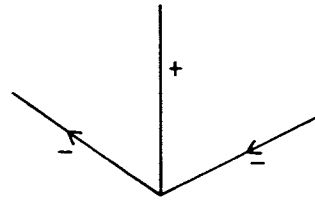


Fig. 2.44

However there are some interesting cases where I have found that I can usually handle the problem scenes.

Heuristic 1 Try to minimize the number of objects in a scene interpretation.

Implementations:

1. Make shadow L junction labels (Fig. 2.44[a]) more likely than any other type of L junction.
2. Make labels representing interior table regions more likely than the equivalent labels that do not involve table regions.
3. If regions can be interpreted either as shadows or as objects, make shadow interpretations more likely.

Heuristic 2 Eliminate interpretations that have points of contact between objects or between objects and the TABLE unless there is solid evidence of contact.

Implementation: Make ARROW junction labels which have two concave edges and one convex edge (Fig. 2.44(b)) less likely than ARROW labels of other types.

These heuristics select interpretations (1), (2), and (7) from Fig. 2.33 and interpretations A(1) and B(1) from Fig. 2.35.

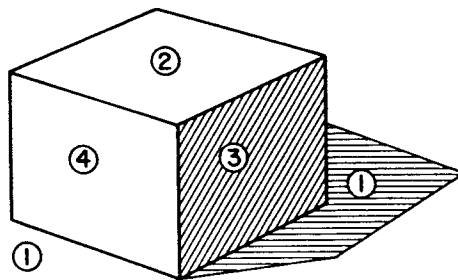
2.7 REGION ORIENTATIONS

What has obviously been missing from all that I have shown so far is a connection between line segment directions on the retina and possible labelings for these lines. Such a connection is extremely useful if the program is to understand gravity and support. In this section I describe approaches to this problem which I have not yet included in my program. There is probably as much work required to properly add the ability to handle direction information as I have already invested in my program. Nonetheless, I believe that this section provides a good idea of the work that needs to be done as well as the physical knowledge that these additions will allow one to include in the program.

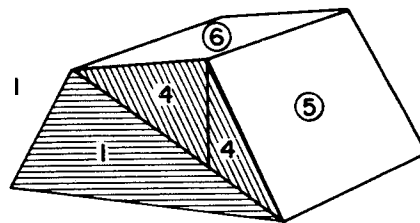
2.7.1 General Region Orientations

In this section I define another scheme which assigns to each visible region one of 16 values. The regions are named in as sensible and simple a manner as I could devise, and are defined with respect to a coordinate system which is itself defined by the TABLE surface and the position of the eye viewing the scene. The region orientation values are each shown in Fig. 2.45. I assume that this figure will serve as an adequate specification for the meaning of the different orientation values. If the scene is moved with respect to the eye or vice versa, then the region values (except table and horizontal) may change, and regions previously invisible may become visible. Thus the region orientation values are not inherent properties of the surfaces, but are only defined with respect to a particular eye-table arrangement.

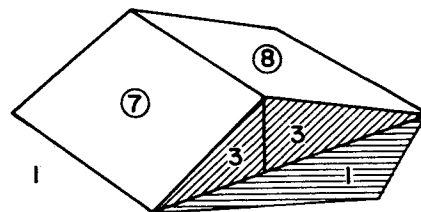
- ① Table (TA)
- ② Horizontal
- ③ FRV (front right vertical)
- ④ FLV (front left vertical)



- 1 TA
- 4 FLV
- ⑤ FRU (front right up)
- ⑥ BLU (back left up)



- 1 TA
- 3 FRV
- ⑦ FLU (front left up)
- ⑧ BRU (back right up)



- 1 TA
- ⑨ LU (left up)
- ⑩ RU (right up)
- ⑪ FU (front up)
- ⑫ BU (back up)

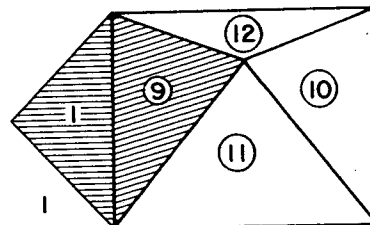
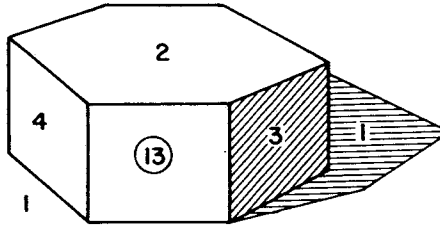
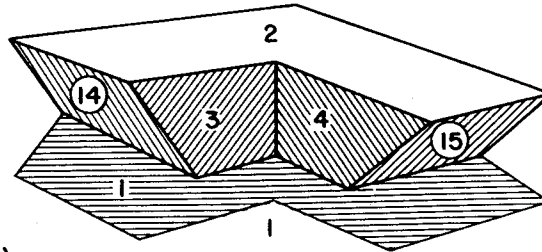


Fig. 2.45

- 1 TA
- 2 H
- 3 FRV
- 4 FLV
- ⑬ FV (front vertical)



- 1 TA
- 2 H
- 3 FRV
- 4 FLV
- ⑭ FLD (front left down)
- ⑮ FRD (front right down)



- 1 TA
- 2 H
- 3 FRV
- 4 FLV
- ⑯ FD (front down)

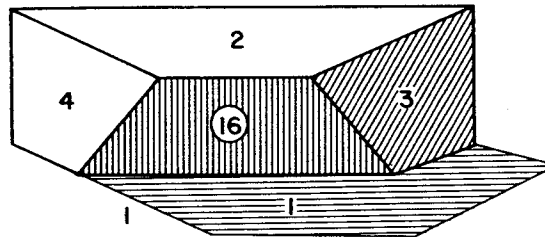


Fig. 2.45 (continued)

2.7.2 General Line Directions

Before I can carry out this type of association in general, I must

1. define line directions on the retina
2. define line directions in the scene domain
3. show how to find the scene direction values, given a labeled line drawing and the retinal line directions

Throughout this section I assume that the eye is far enough away from the scene so that vertical edges in the scene project into North/South lines on the retina. Since the definition of North/South edges includes a tolerance angle, the eye does not need to be at infinity for this condition to hold. By the same reasoning I assume that parallel edges can be recognized without resorting to perspective or vanishing point considerations.

First I define the retinal line directions in terms of compass points as shown in Fig. 2.46.

Next, in Fig. 2.47, I define the names for the directions of lines in the scene by showing examples for each type possible direction. These names resemble the names for region orientations, but I will always use lower case

lp/rp

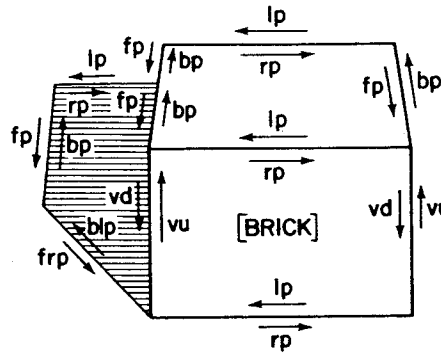
(left parallel /
right parallel)

fp/bp

(front parallel /
back parallel)

vu/vd

(vertical up /
vertical down)

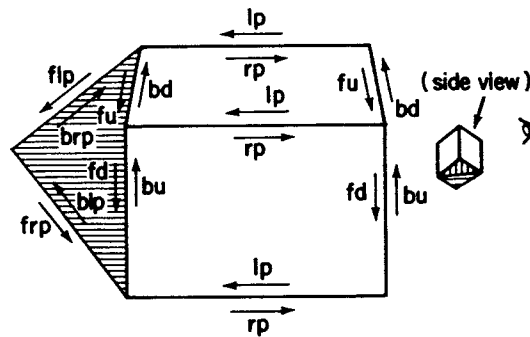


fu/bd

(front up /
back down)

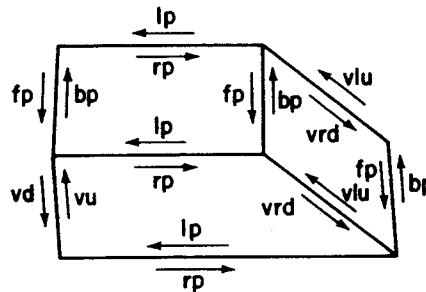
fd/bu

(front down /
back up)



vlu/vrd

(vertical left up /
vertical right down)



vld/vru

(vertical left down /
vertical right up)

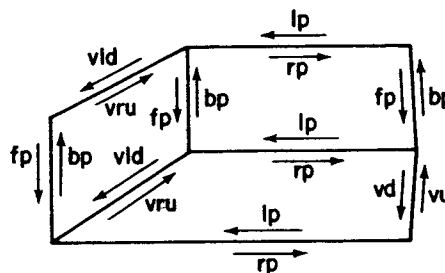


Fig. 2.47 (continued)

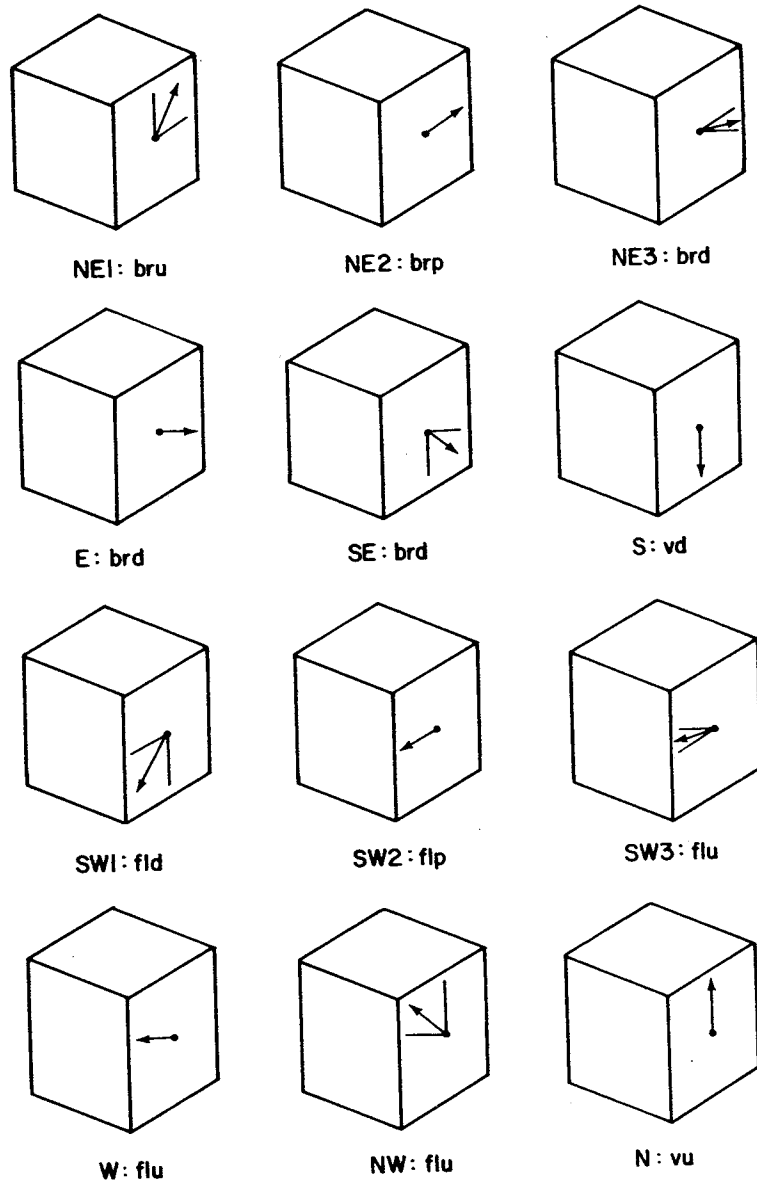


Fig. 2.48

which bounds a type FRV region can only be caused by a type brd edge, etc. Figure 2.49 is a summary of the types of scene edges which can cause lines of each type on the retina, arranged according to the types of regions that each edge can bound.

Now to tie everything together, notice that an edge can only separate two regions if the edge could have the same direction in both regions bounding the edge. Therefore, to find all the region pairs that an N (North) edge (as seen on the retina) could separate, look down the N column in Fig. 2.49 and find all the pairs of regions which can share an edge which points in a particular direction. A north pointing edge can thus separate any of the following pairs of region types (this is not a complete list):

Region type ↓	Direction of line on retina							
	N	NE	E	SE	S	SW	W	NW
H or TA	bp	brp	rp	frp	fp	flp	lp	blp
FU	bu	bru	rp	frd	fd	fid	lp	blu
FV	vu	vru	rp	vrd	vd	vld	lp	vlu
FD	fu	fru	rp	brd	bd	bld	lp	flu
FRU	bu	bru brp brd	brd	brd vrd frd	fd	fid flp flu	flu	flu vlu blu
FRV	vu	bru brp brd	brd	brd	vd	fid flp flu	flu	flu
FRD	fu	fru, vru brp, bru brd	brd	brd	bd	bld, vld fid, flp flu	flu	flu
RU	bp	brd	brd	brd vrd frd	fp	flu	flu	flu vlu blu
BRU	bd	brd	brd	brd, vrd frd, frp fru	fu	flu	flu	flu, vlu blu, blp bld
BU	bd	brd	rp	fru	fu	flu	lp	bld
BLU	bd	brd, brp bru, vru fru	fru	fru	fu	flu, flp fid, vld bld	bld	bld
LU	bp	bru vru fru	fru	fru	fp	fid vld bld	bld	bld
FLU	bu	bru vru fru	fru	fru frp frd	fd	fid vld bld	bld	bld blp blu
FLV	vu	fru	fru	fru frp frd	vd	bld	bld	bld blp blu
FLD	fu	fru	fru	fru, frp frd, vrd brd	bd	bld	bld	bld, blp blu, vlu flu

Fig. 2.49

((TA TA) (HH) (TA LU) (H LU) (H RU)
 (RU TA) (RU H)
 (FRV FRV) (FRV FLV) (FRV FV)
 (FLV FRV) (FLV FV) (FLV FLV)
 (FV FV) (FV FLV) (FV FRV)
 (LU H) (LU RU) (LU LU)
 (RU H) (RU LU) (RU RU)
 (BLU BRU) (BRU BLU))

Not all these pairs can be separated by the same types of edges; shadows and cracks can only separate regions with the same orientation values, and convex edge pairs become concave edge pairs if the order of the pairs is reversed. For example, a North line separating regions with orientation values (FLV FRV) represents a convex edge (where the ordering of the regions is in a clockwise direction), but if the orientation values are (FRV FLV) for a North line, this must represent a concave edge.

A program can use this information in the following ways:

1. If there are ambiguities remaining after the regular labeling program has finished, pick a single labeling, assign region values and see whether this labeling can represent a possible interpretation; if the interpretation is not possible, then the program will be unable to assign orientation values to every region.
2. Region illumination values can be tied in with the region orientation values. For example, if a scene is lit from the left, and the light-eye angle is less than 90° (in Fig. 2.50, the light-eye angle is the angle

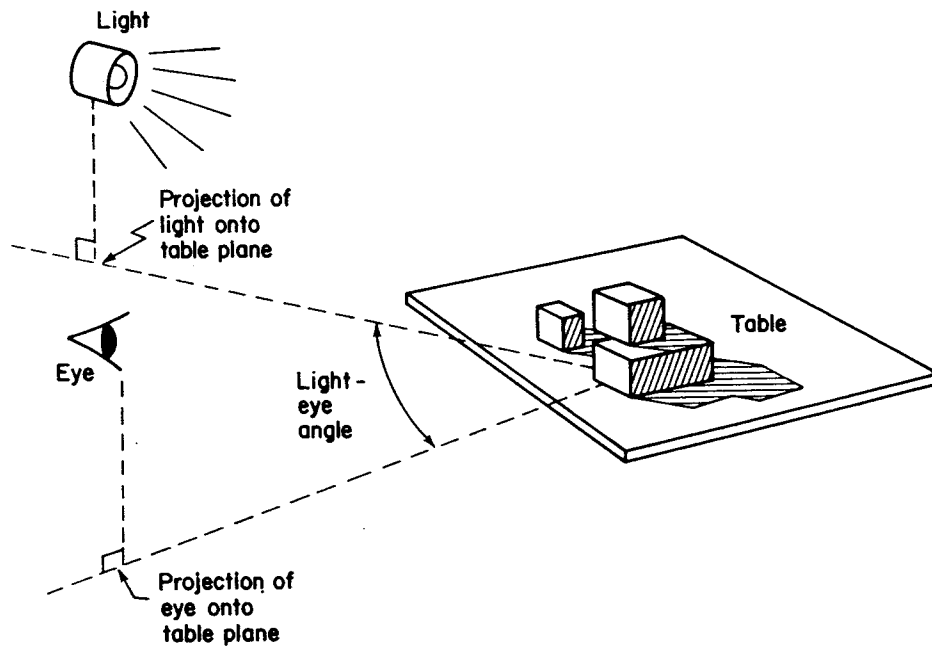


Fig. 2.50

between the projections of the eye and the light onto the plane of the TABLE, as measured from the center of the scene), then a region cannot be labeled simultaneously as orientation type FLV and illumination type SS (self-shadowed).

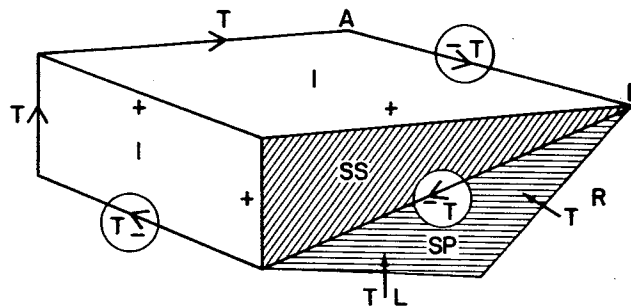
3. All these facts provide a neat way to integrate stereo information into a scene description. For example, if an edge is truly vertical (type vu) then it must appear as N (North) in any retinal projection of a stereo system. However an edge which is of type bp (back parallel) can appear to be N on the retina because of the particular placement of the eye with respect to the scene. If the eye is shifted slightly to the right, this edge will now appear to point NE (Northeast) and if the eye is shifted to the left, the edge will appear to point NW (Northwest). Clearly this knowledge would enable a program to much more severely restrict the region orientation pairs, and consequently the labelings, that can be assigned to a line drawing of a scene.
4. All the possibilities for region orientations can be generated by the function I use to build up sets of region illumination values. For each labeling which the program finds, region pairs can be selected according to the line directions and line labels, and a set of region orientation values can be built up. The difference is that there are far too many region orientation values in general to possibly include them in precompiled form; the values must be generated from the greatly reduced set of possibilities that remain after the regular labeling program has completed its work. The reason why there are so many possibilities is that there are so many possible region orientations. Each edge can potentially have $16 \times 16 = 256$ region orientation pairs as opposed to the nine possible region illumination pairs.

2.7.3 Support

Using the region orientation values, I can now define the set of edges along which support must hold, the set of edges along which support can hold, and the set of edges along which support cannot hold. By support I mean what is commonly termed either resting on or leaning on.

To start with, I can eliminate from consideration any edges which are shadows, convex edges, obscuring edges, or concave edges made up of one object or of three objects, and I can say for certain that support is exhibited along any concave edge which has the TABLE as a bounding region.

The important fact is that these edges exhibit support regardless of their directions on the retina, so that there is no problem with edges such as L-A-B in Fig. 2.51. The best previous rules to find where support holds in a scene (Winston⁶) are not able to handle cases like this; Winston's rules were biased toward finding ARROWS, Ks, and Xs which have vertical (or at least upward



Support relations are circled.

Fig. 2.51

pointing) lines. In addition, Winston's rules failed to find support relations for leaning blocks; his rules assumed that objects would be supported by face contact only.

Although my program can find support in cases like Fig. 2.51, it is important to note that, in general, it is not possible to use my regular labelings and line directions alone to find which edges exhibit support and which do not. Suppose that on the basis of the frequency of crack edges like the ones shown in Fig. 2.52(a) I decided to label as supporting/crack edges ones in which the arrow of the crack label points SW, W, or NW, and to class all the others together as being crack edges without support relations. Then in Fig. 2.52(b) edges L-B-C and L-C-D would be correctly marked but L-A-B would not. I could patch up the rule by saying that if support holds for one noncollinear line in an X junction it must hold for the other noncollinear line of the X as well. Unfortunately this rule causes the program to assert that support holds between the two objects in Fig. 2.52(c) since support would be transferred by the rule from L-B-C to L-A-B.

Similarly, for concave edges I cannot use line directions and the direction of the arrow on the label to define support. As an example, observe that while L-A-B in Fig. 2.52(d) does not exhibit support, L-C-D in Fig. 2.52(e) does.

Region orientation values can help to avoid these problems, at least for some cases. (There are some cases, such as the one in Fig. 2.52[f], where I do not know whether to say that support holds along L-A-B and L-B-C or not.) Interestingly enough, with region orientations specified, I do not necessarily need line directions, although I certainly need line directions to find the region orientation values to begin with.

An example of an edge where support must hold is any concave edge which has a horizontal surface on its left when one looks along the edge in the direction of its "arrow", as does L-C-D in Fig. 2.52(e).

Some examples of edges where support cannot hold are concave edges which have vertical surfaces (FRV, FV, or FLV) or downward pointing

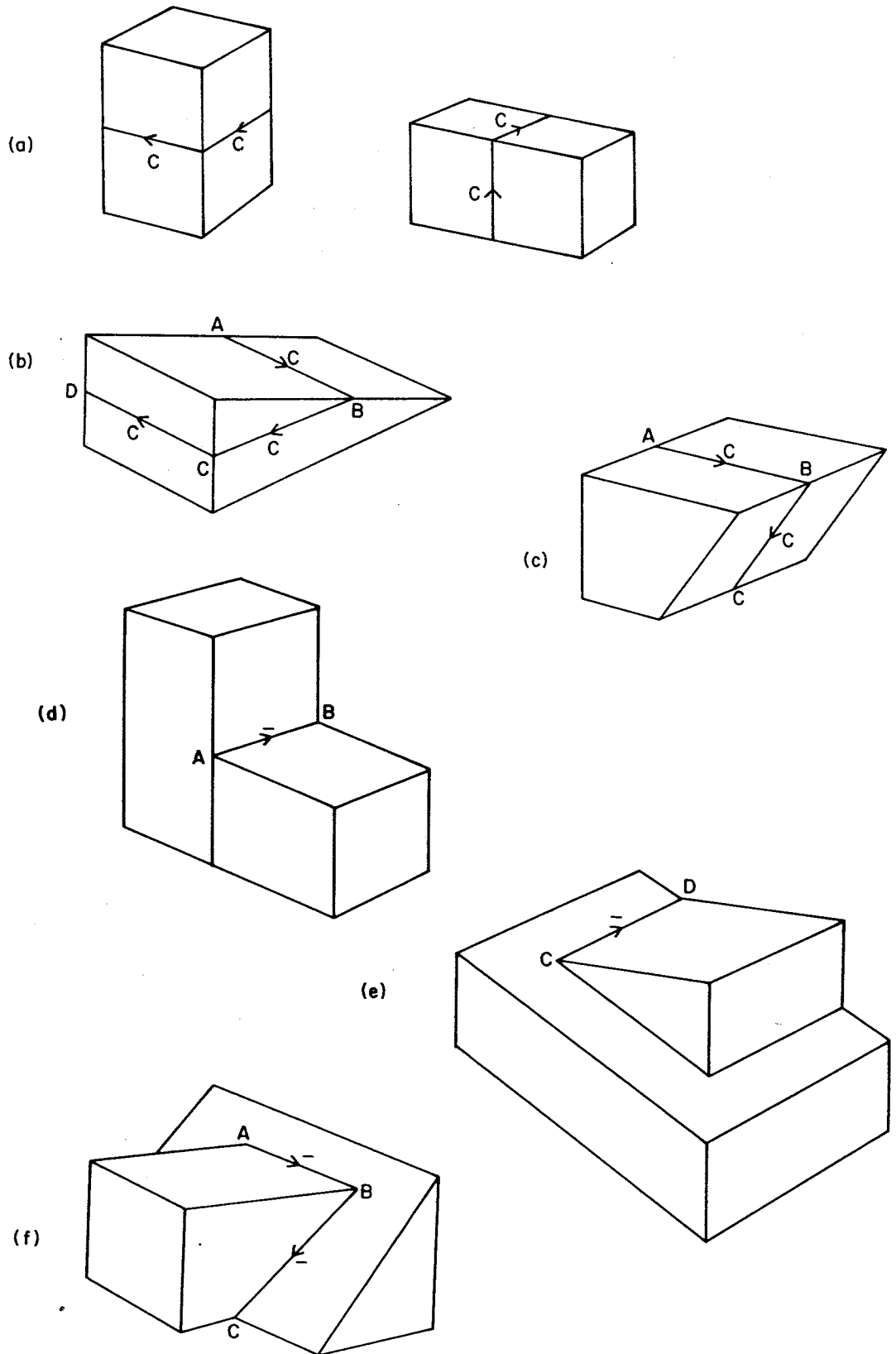


Fig. 2.52

surfaces (FRD, FD, or FLD) on the left of the edges when looking along the direction of the "arrow"; line L-A-B in Fig. 2.52(d) is an edge of this type.

REFERENCES

1. Huffman, David: Impossible Objects as Nonsense Sentences, in B. Meltzer and D. Michie (eds.), "Machine Intelligence 6," Edinburgh University Press, Edinburgh, Scotland, 1971.
2. Clowes, Maxwell: On Seeing Things, *Artif. Intel.*, 2(1):79-116 (1971).
3. Sussman, G., T. Winograd, and D. McDermott: MICRO-PLANNER Reference Manual, *M.I.T. Artificial Intelligence Laboratory Memo 203A*, 1971.
4. Freuder, Eugene: The Object Partition Problem, *M.I.T. Artificial Intelligence Laboratory Working Paper 4*, 1971.
5. Mahabala, H. N. V.: Preprocessor for Programs which Recognize Scenes, *M.I.T. Artificial Intelligence Laboratory Memo 177*, 1969.
6. Winston, Patrick H.: "Learning Structural Descriptions from Examples," Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, Mass., 1970 (included as Chap. 5 of this book).