Sensory Processing and World Modeling for an Active Ranging Device

by

Tsai-Hong Hong

Angela Y. Wu

September 1991

Contents

1. INTRODUCTION

The NASA Standard Reference Model (NASREM) architecture is a hierarchical model structured into six control levels such that at each level, a different fundamental mathematical transformation is performed. The six levels are: Operation Control, Service Bay Control, Object/Task level, Elementary Moves (e-move), Primitive level, and Servo/Coordinate Transfer level.

Each level of the hierarchy is functionally partitioned into three modules : task decomposition, world modeling and sensory processing. The world modeling module is the knowledge base which has the internal representation of the external world. It maintains geometric models of the world and stores lists of objects and their attributes in each level. It generates predictions and evaluation functions to be used by the sensory processing module. The sensory processing module computes temporal and spatial correlations, convolutions, differences and integrations. The sensory module's output will confirm or deny the prediction provided by the world model, thus the information stored in the world model will be updated.

In this project, we studied world modeling and sensory processing for laser range data. World Model data representation and operation were defined. Sensory processing algorithms for point processing and linear feature detection were designed and implemented. The interface between world modeling and sensory processing in the Servo and Primitive levels was investigated and implemented. In the primitive level, linear features detectors for edges were also implemented, analyzed and compared.

Section 2 of this report surveys the existing world model representations. It also presents the design and implementation of the Y-frame model, a hierarchical world

model. Section 3 contains the interfaces between the world model module and the sensory processing module. Section 4 describes the linear feature detectors designed and implemented.

2. WORLD MODEL REPRESENTATION

In order to design a World Model representation of space platforms, different existing methods proposed by various researchers in the field were studied, compared, and contrasted. In this section, we present an analysis of the advantages and disadvantages of the different World Model Representations. A data structure suitable for the modeling space platform and range data was designed and implemented. This World Model is a hierarchical feature based representation. It is also compatible with NASREM (NASA Standard Reference Model) system and HARPS (Hierarchical Ada Robot Programming System). The hierarchical feature based data structures and programs were implemented in the programming language Ada.

2.1 Design Criteria of World Models

As the world model is a key component of any intelligent machine, much research is focused on it and many different representation methods have been proposed, tested, and implemented. No matter, however, which exact scheme is used, the modeling system must be able to adequately model the complexity of the objects in the environment, and it must contain enough structure to allow the low level sensory data to map into the model during the robot operation. A good world model representation scheme should posses three properties amongst others: *validity*, *completeness*, and *uniqueness*. These properties assure that a representation does not generate nonsense objects (*validity*), that a given representation gives rise to only one object (*completeness*), and that a given object possesses only one representation (*uniqueness*).

Hence, according to Peter K. Allen [4], when designing a world model for an autonomous system, the following criteria have to be taken under consideration:

1) Computability from sensors. A model must be in some way computable from the sensory information provided by the low level sensors. If the model representation scheme is very different from the sensory information, then transformations which may not be information preserving are necessary. These transformations can also make the recognition process slow and inefficient. A better situation is one in which the model representation scheme is directly related to the sensors scheme.

2) Preserving structure and relations. Models of complex objects need to be broken

down into manageable parts, and maintaining relationships between these parts in the model is important. In recognizing the environment, relational information becomes a powerful constraint. As an object is decomposed, it should retain its "natural" segmentation. This is important in identifying partial matches of a workspace.

3) Explicit specification of features. Feature based identification has been a useful prototype in recognition tasks. If features of objects are computable, then they need to be modelled explicitly as an aid in the recognition process. Most object recognition systems are model-based discrimination systems which attempt to find evidence consistent with a hypothesized model, for which there is no contradictory evidence. The more features that are modelled, the better the chances of a correct interpretation.

4) Ability to model curved surfaces. Some domains may be constrained enough to allow polyhedral models or simple cylindrical objects. However, most domains need the ability to model curved surface objects. The models must be rich enough to handle doubly curved surfaces as well as cylindrical and planar surfaces. This complexity precludes many representation schemes, particularly polygonal networks, which have simple computational properties, but become difficult to work with as the number of faces increases.

5) Modeling ease. Very rich, complicated models of objects are desired. However, unless these models can be built using a simple, efficient and accurate procedure, it may be prohibitive to built large data bases of objects. Modeling is done once, so there is an acceptable amount of effort that can be expended in the modeling effort. As designs change and different versions of an object are created, incremental changes are desired, not a new modeling effort.

6) Attributes easily computed. Whatever representation is used, it is important that geometric and topological measures are computed efficiently and accurately. For surfaces,

this means measures such as area, surface normal and curvature. For holes and cavities, this means axes, boundary curves and cross sections. Analytical surface representations are well suited for computing these measures.

2.2 World Model and Sensory Data

The world model builds its internal representation of the workspace based on the data returned by the sensory system of the robot. As the sensing means are so essential in the update of the world model, a brief introduction to the current tendencies in robotic sensing systems will be beneficial in understanding the basic principles of world modeling.

Work environments are not static and can not always be constrained. There is much uncertainty in the world, and humans are equipped with powerful sensors to deal with this uncertainty. Robots need to have this ability also. There is at present much work going on in the area of sensor design. Range finders, tactile, force/torque, and other sensors are being developed.

Much of the sensor related work in robotics has tried to use a single sensor, typically vision, to determine environmental properties. However, not all sensors are able to detect many of the properties of the environment that are deemed important. As a consequence, the world model is fed inadequate and inaccurate data. This requires the use of complex, time-consuming algorithms in order to improve the quality of the input. Still, a wrong environment representation may be obtained, which will eventually lead to mistaken robot operation.

A much more promising approach is to supplement the single sensor data (in most cases the visual information) with other sensory inputs. To increase the capabilities and performance of robotic systems, in general, requires a variety of sensing devices to support

the various tasks to be performed. Since different sensor types have different operational characteristics and failure modes, they can, in principle, complement each other. This is particularly important, because multiple sensor systems can be used to generate improved world models and provide higher levels of safety and fault tolerance. More specifically, the tendency today is to additionally use tactile sensing to supplement the sparse data. While vision remains the primary sensing modality in robotics, interest in tactile sensing is increasing. Vision systems are unable to deal effectively with occlusion, uncontrolled illumination and reflectance properties. At the same time, tactile information can directly measure shape and surface properties.

However, although adding sensors to a robotic system can produce more accurate sensing, it also introduces complexity due to the added problems of control and coordination of the different sensing systems. It is difficult enough to regulate and organize the activities of a single sensor system, let alone those of a multiple sensory system with different bandwidth, resolution, accuracy, and response time that must be integrated in one world model.

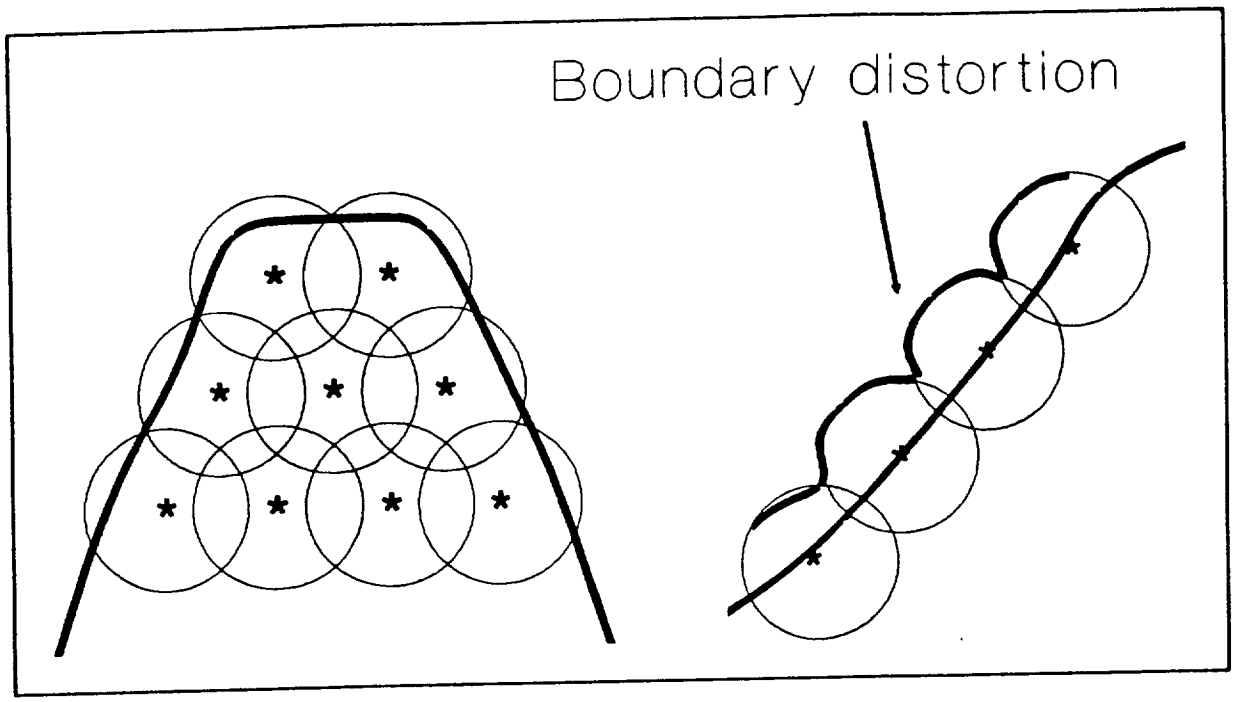
2.3 Combination Geometry World Model

When designing an autonomous system, a researcher always has a specific application in mind. The first component he has to select is the form of input. In other words, he has to decide about the sensory system of the robot. Once the sensory means are known, the world model can be developed. However, the choice of sensory devices and the selection of a specific application impose strong constraints in the design of the world model. As a consequence, each researcher comes up with his own variation of world modeling, leading to a plethora of world models.

One approach is the model supported by M. Goldstein, F. G. Pin, G. de Saussure, and C. R. Weisbin [19]. This scheme describes the shape of objects using spheres. The whole idea is based on combinatorial geometry, also known as Constructive Solid Geometry (CSG), where solids are represented as combinations of primitive solids or building blocks, using Boolean operations of union, intersection, and difference. The data structure used for its representation is a binary tree, where the terminal nodes are instances of primitives and the branching nodes represent Boolean operators.

Using range data, each measured point on the objects surface is surrounded by a solid sphere with a radius determined by the range to that point. Then, the 3-D shapes of the visible surfaces are obtained by taking the Boolean union of the spheres. In more detail, the result of a range scan is a matrix of distances from the sensor focal plane to an object surface. In other words, the coordinates of discrete points on the visible parts of the

boundary surfaces of different objects in the external world are known. Let a be the small angle between two successive reading directions of the sensor. First each discrete point i, is surrounded by a small sphere with a radius $r_i = \max(R_i * \sin * R_i)$, where R_i is the associated measurement error, R_i is the range sensor measurement, and the subscript i is a reference to a specific object point. The approximate 3-D shape of the visible boundary surface is obtained directly by taking the union of all the spheres.



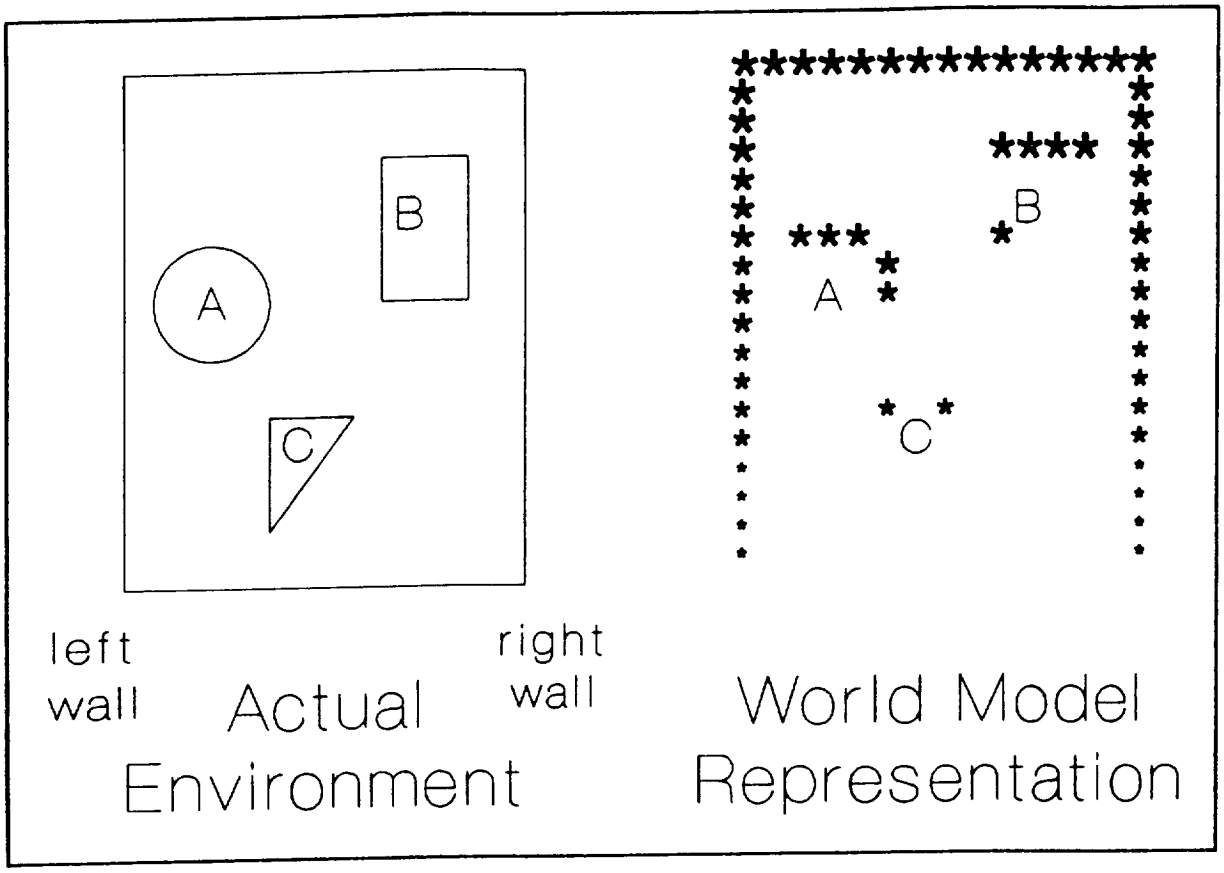
Sphere Representation

The reason for using spheres as primitive solids, is to keep the representation as compact as possible. Describing the sphere for a particular discrete point in space means adding only one additional parameter, the radius, to the coordinates of the discrete point which are provided by the sensor.

To avoid the appearance of "holes" in the geometry and to take into account the range uncertainty, r_i is defined in such a way, that neighboring spheres are highly

overlapping one another. Thus, the boundary surface of the union of all spheres is continuous (without "holes") from the robot's point of view. Still, it is obvious that by using spheres, the shape of the boundary surfaces is distorted. However, the distortion is proportional to the range at each point, which means that the resolution of the model is improved as the range to the surface is decreased.

A very useful feature of this combinatorial geometry representation is its efficiency in calculating distances to 3-D surfaces in a desired direction. The range data provided by the sensor quantify the distances from the sensor focal plane (the center of the robot) to the object surfaces.



Combinatorial Geometry Environment Representation

This model was developed by emphasizing the following aspects: minimal fast memory for storage, efficiency in navigation, minimal computation, and no a priori knowledge. It is ideal for fast building up of world models, but is not very accurate in the sense that surface boundaries are distorted. In addition, although Constructive Solid Geometry (CSG) is complete in its representation, it is not unique. However, boundary distortion of the type involved in this scheme, will not affect the performance of navigation.

2.4 Polygonal Planar Hulls World Model

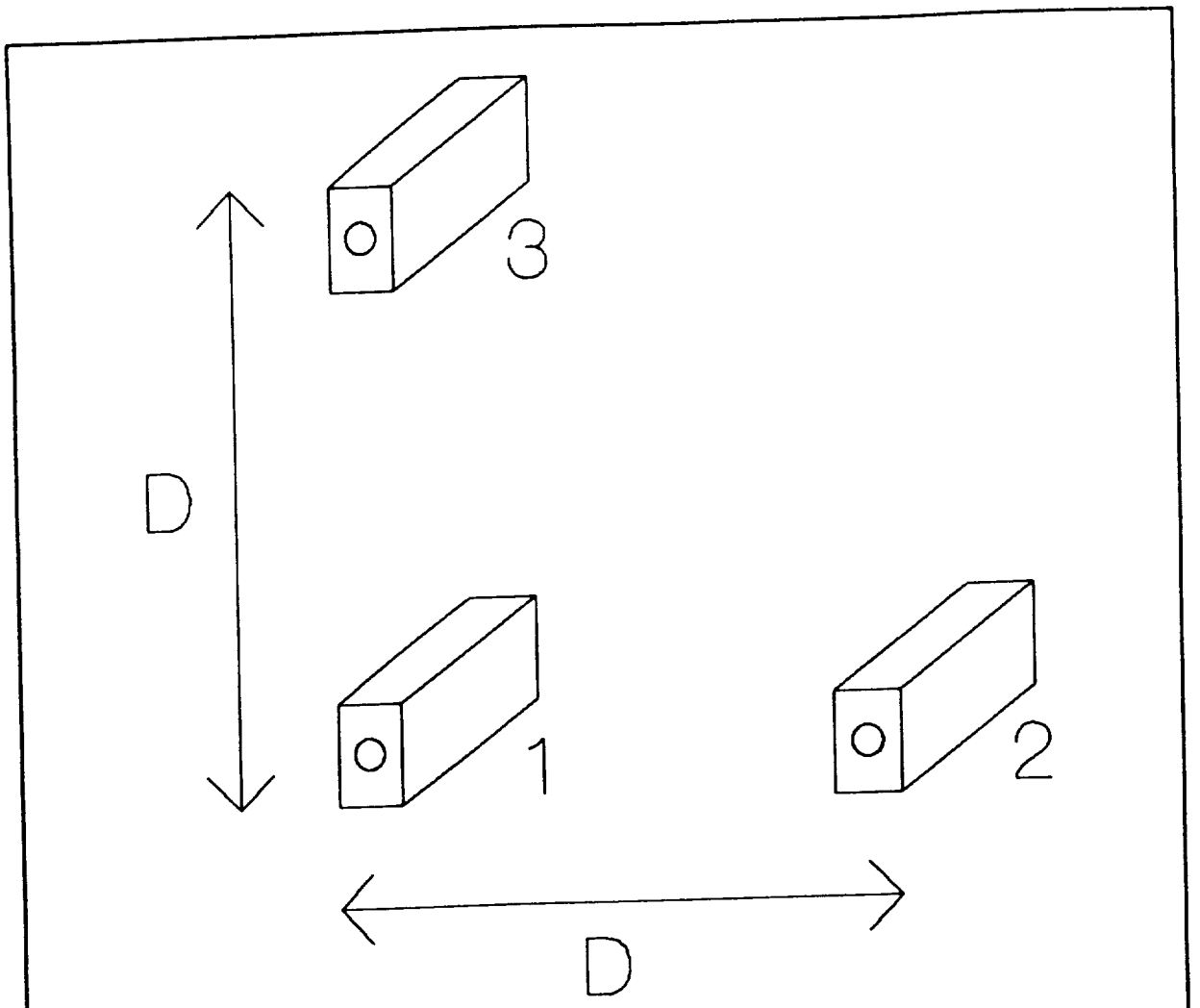
Arnaud R. de Saint Vincent [12], on the other hand, proposed a different world model representation that produces a planar description of the occupied space consisting of a set of non-convex polygonal hulls enclosing the ground projected 3-D features. The map is built from 3-D stereo data obtained from the robot's standpoint.

For the stereo vision, three cameras positioned at the vertices of a right triangle are used. Stereo-correspondences are searched twice, between the images produced by camera 1 and camera 2 (horizontal epipolar lines), and between images produced by camera 1 and camera 3 (vertical epipolar lines).

In order to provide data for a higher-level understanding of the scene (detection of main features such as walls, doors, etc.) and for easier recognition of already seen parts of the environment, vertical planes are searched among the 3-D segments. This is done by a prediction and verification algorithm or/and by use of a priori knowledge of the world, when available and applicable.

Then, the geometric map of the occupied space is built. The construction of the model must take into account not only the previously detected vertical planes, but also a set of sparse 3-D features (segments) which belong to unmodeled obstacles.

In this case of sparse depth measurements, it is in general impossible to determine the exact free space, because the position of the physical surfaces linking the perceived segments can not be predicted. However, it is possible to compute a description of



Camera Setup
for Stereo Acquisition

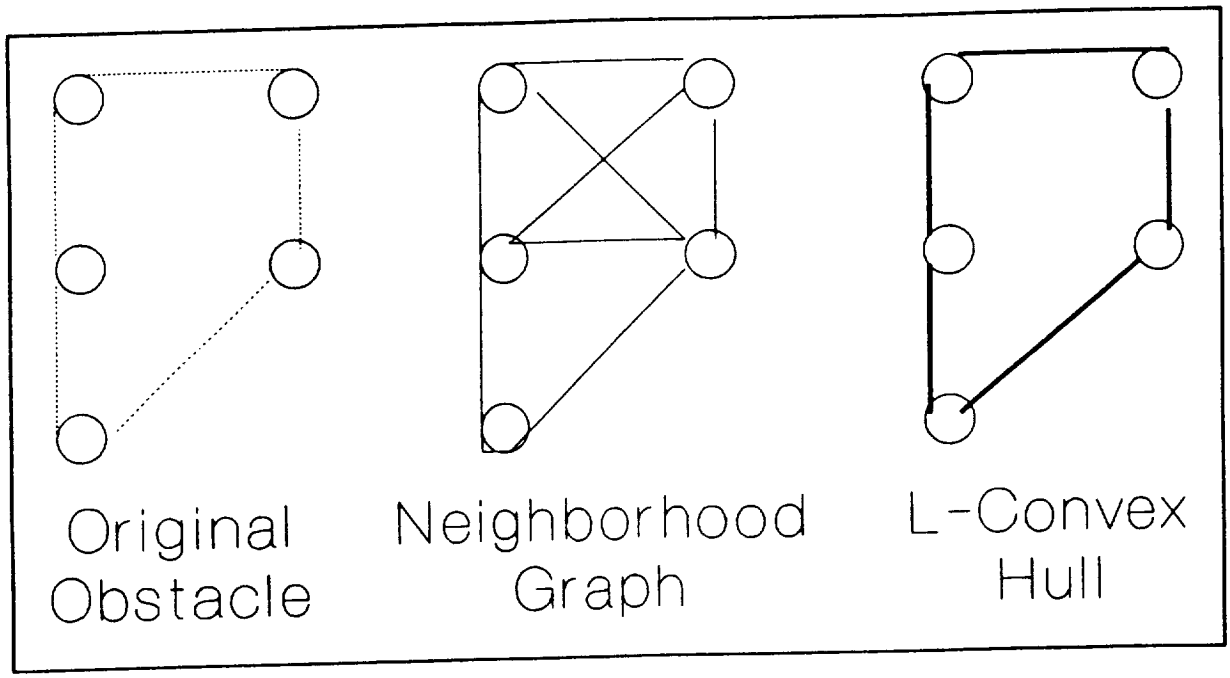
"certainly occupied" areas, relying on other sources of information, such as other sensory devices, to further remove ambiguities.

Nevertheless, this representation scheme enables to avoid this problem by building an intrinsic geometric description of the occupied space. This model uses a set of planar polygonal nonconvex hulls which enclose the ground-projected perceived segments.

This representation is based on a new family of hulls called *L-convex hulls*. The definition of these hulls is purely geometric, and the resulting model of the occupied space is thus independent of any assumption on the world structure. The main property of the model is that, though it does not, in general, represent the real shapes of the objects (as this is unpredictable with a single sensory system), the topological properties of the free space are preserved. What this means is that, given a collision-free trajectory of the robot in an environment, every point of this trajectory will be in the *free space* as described in this model.

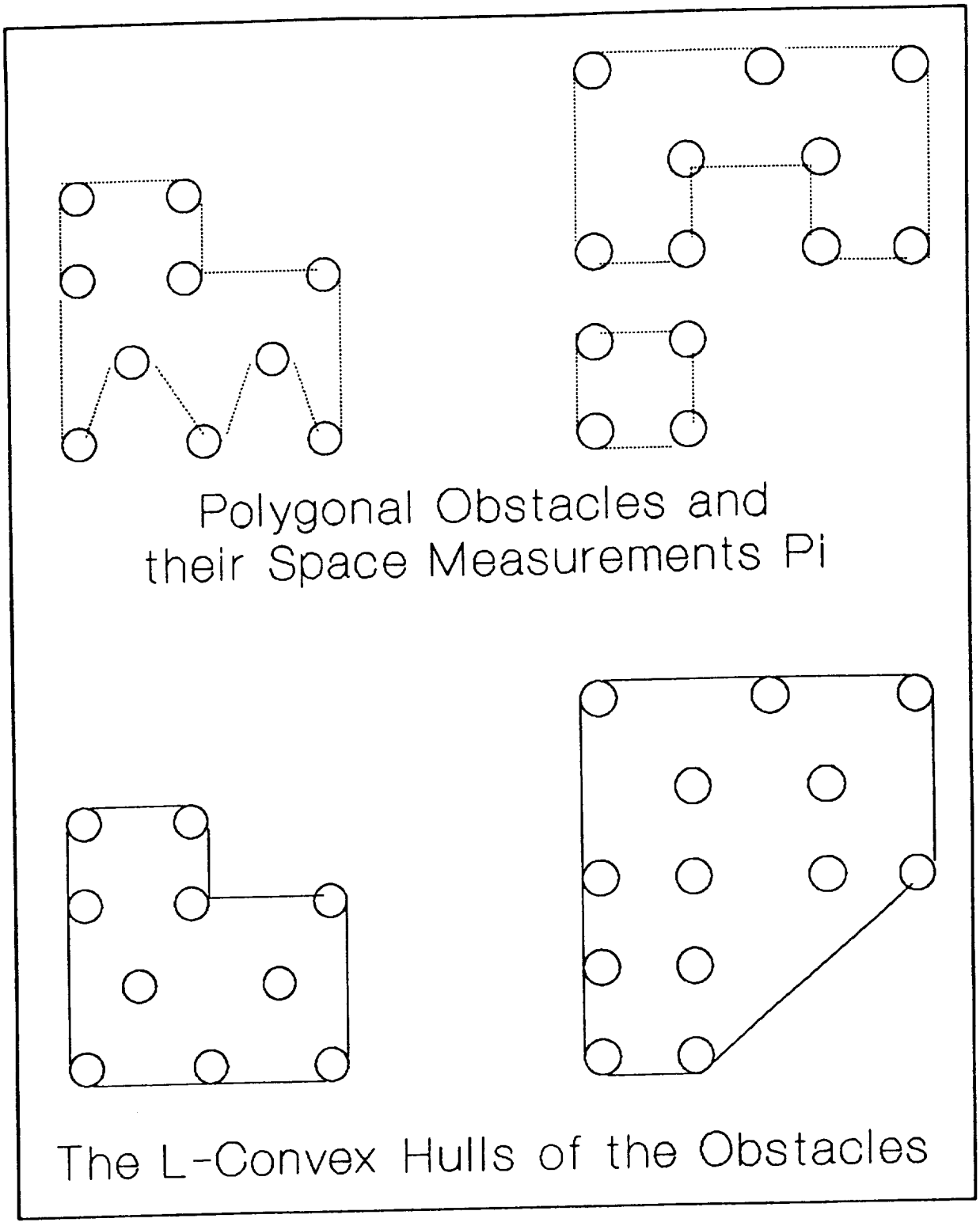
For constructing the *L-convex hull* of an obstacle, the input data consists of the coordinates of the vertices P_i of the obstacle. For each vertex its neighborhood graph is calculated. This neighborhood graph is a set of all the couplets (P_i,P_j), where P_i and P_j are neighboring vertices. Let L be the diameter of the robot. The *L-convex hull* is the smallest set C such that, for any couple (P_i,P_j), if (P_i,P_j) belongs to the list of external arcs and $D(P_i,P_j)<L$, then the segment $[P_i,P_j]$ is included in C.

As no exact description of obstacles is represented, but the topological properties of the free space are preserved, this model is best suited for navigation projects. The fact that no environment assumptions are made, gives flexibility to the scheme. In addition, the employment of a priori knowledge, if applicable, is an extra advantage. However, for the convex hulls to be constructed, the neighborhood graph of each and every vertex of all the



Construction of an L-Convex Hull

obstacles must be calculated, so the algorithms involved in this model are time consuming.



Obstacle Representation Using L-Convex Hulls

2.5 Geometric World Model

Another researcher, James L. Crowley [10], suggested the use of a geometric world model which illustrates the environment in form of line segments. The local model, as well as the raw ultrasonic range data are described as line segments, represented with the following data structure.

In this structure, the minimal set of parameters is:

PM : mid-point of the line segment in external coordinates,

θ : orientation of the line segment,

h : half-length of the line segment,

σ_θ : uncertainty (standard deviation) in the orientation,

σ_c : uncertainty in position perpendicular to line segment.

In addition, there is a set of redundant parameters that can be used like:

a,b : for the line equation $a=\sin(\theta)$, $b=-\cos(\theta)$,

c : perpendicular distance to the origin, $c=-ax-by$,

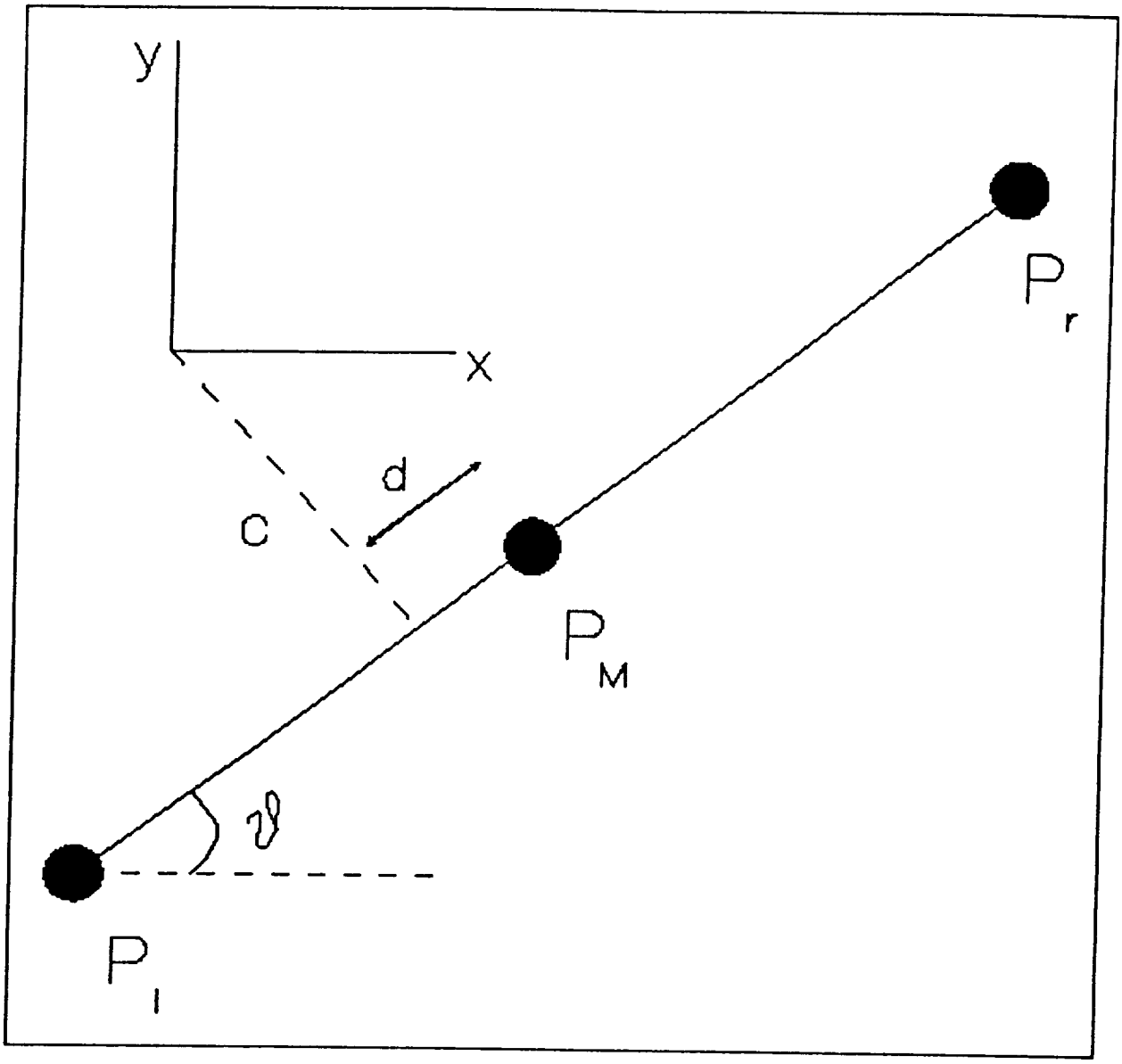
d : distance from the perpendicular intercept to the origin, to the midpoint of the segment,

P_r : end-point to the right of the segment,

P_l : end-point to the left of the segment.

Line segments are also labeled with a confidence factor, CF. A segment with $CF<0$ is removed from the model.

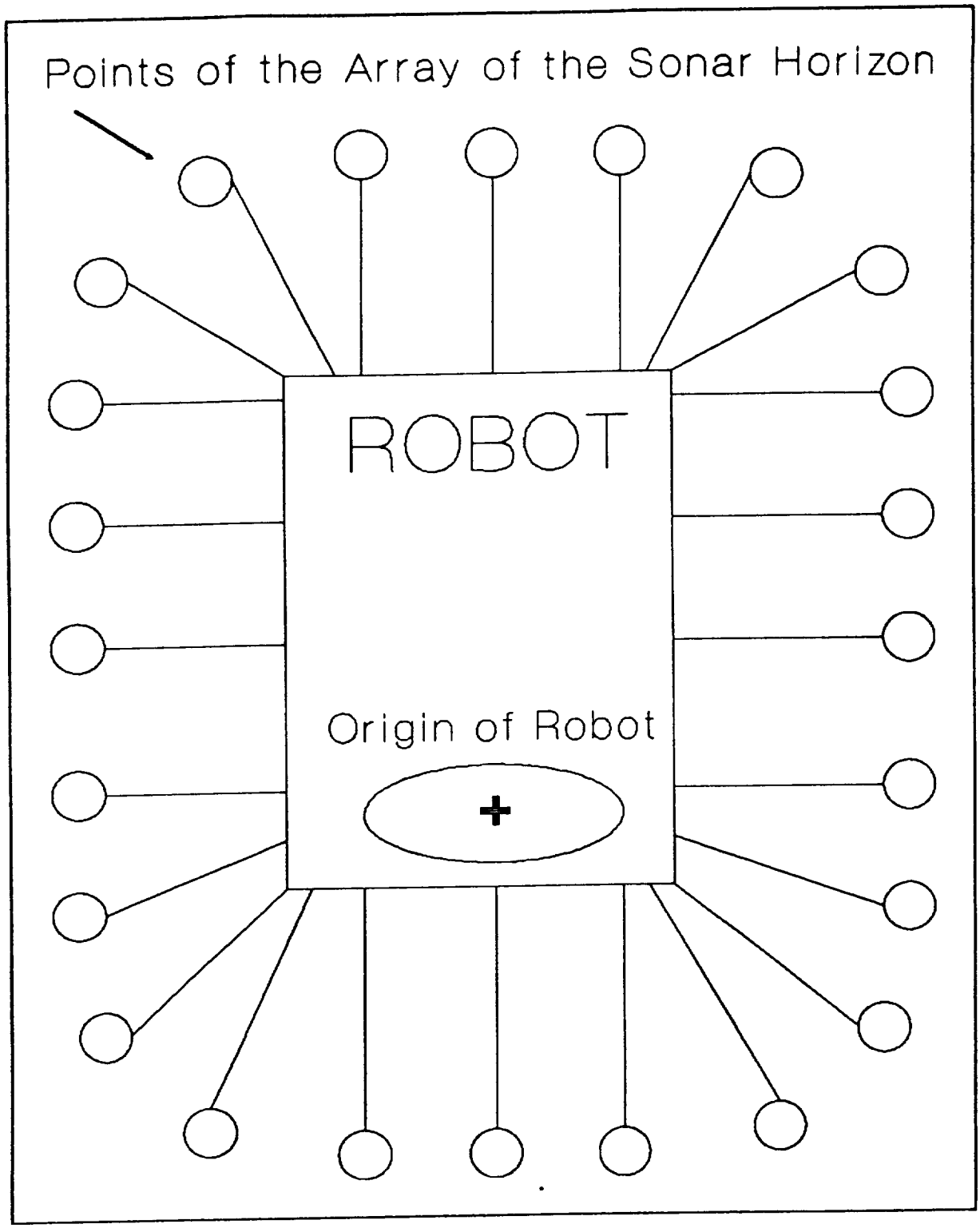
For constructing the line segments, the information is extracted from the visible free-



A Geometric Model Line Segment

space around the robot, known as sonar horizon. The sonar horizon is an array of 24 positions in external Cartesian coordinates. The points in this array are the vertices of a polygon of immediately visible free-space around the autonomous system. An uncertainty is stored along each point in the sonar horizon.

By detecting range measurements that are mutually consistent, sensor noise is



Robot Configuration

filtered.

Line segments are formed in terms of external coordinates to permit the integration of range measurements while the robot is moving. After a segment has been detected and formed, the uncertainty of the robot's position is added to the segment.

Small line segments, just obtained from ultrasound data, are matched to the composite model. Matching is a process of comparing each of the segments in the composite local model against the observed segment to detect similarity in orientation, colinearity and overlap. The longest line segment in the composite model that passes all three tests is selected as the matching segment. This segment is then used to correct the estimated position of the robot and to update the model.

As a conclusion, a geometric model can be implemented at cases where sensor observations are noisy and imprecise, by using an explicit model of uncertainty. This model provides a technique for a vehicle to maintain an estimate of its position as it travels, even in the case where the environment is unknown.

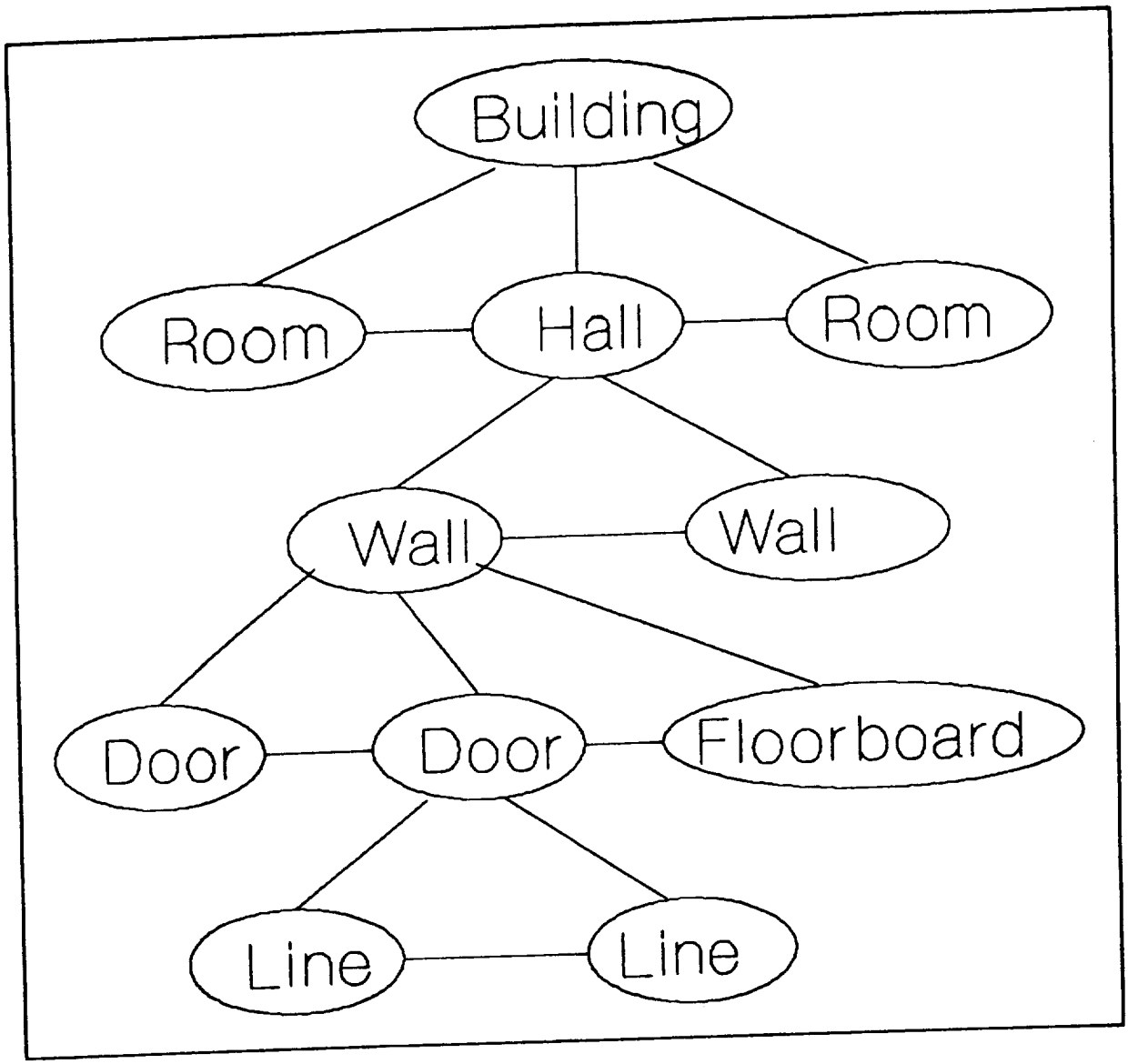
On the other hand, the geometric model leads to sparse and brittle world representations. This scheme requires early decisions in the interpretation of the sensor data for the instantiation of specific model primitives. Additionally, it does not provide adequate mechanisms for handling sensor uncertainty and errors (compared to other models), while it relies heavily on the adequacy of the precompiled world models and the heuristic assumptions used. All these factors introduce strong domain-specific dependencies. Thus, geometric world models may be useful in highly structured domains, but have limited capabilities in more complex environments.

2.6 Hierarchical Geometric World Model

This hierarchical geometric world model used by David J. Kriegman, Ernst Triendl, and Thomas Binfold [28], employs a map which is built bottom up. The lowest level of information is closest to the actual sensor measurement, while higher levels of the hierarchy become more abstract and symbolic. At the lowest level are the points and lines detected from the sensor data. This information is fit to a model of generic objects such as walls, doors, and windows. Higher level structures are composed of lower level patterns. For example, two parallel walls that bound an elongated region of free space would be a hall, and hallways are found in buildings. So, especially in robot navigation, when searching for a route between rooms in a building, search would start at the building level for route and then find paths along successive levels of the map.

The interesting point in the model is that it uses four sensing modalities: vision, acoustics, tactile, and odometry. Each of them returns different environmental information, using different representations, which are combined in a common world model.

Stereo vision uses two onboard cameras and returns three dimensional location of vertical lines within its field of view. The data gathered is generally the most accurate sensed measurement available. However, stereo has a high computational cost and covers only a rather narrow field of view. In addition, the uncertainty in distance measurement from stereo, even at moderate distances, becomes larger than the angular uncertainty which is complementary to the acoustic sensing system.



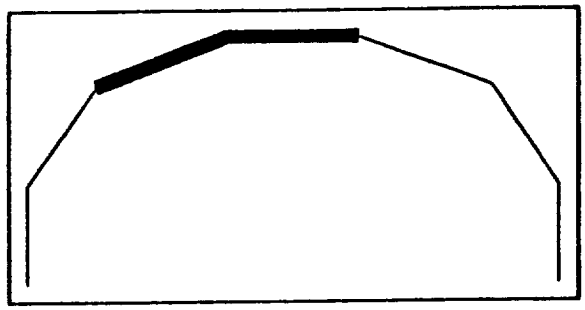
Map of Hierarchical Geometric Model

This acoustic sensing system is composed of twelve Polaroid ultrasonic sensors surrounding the robot, and provides direct range information at a speed of 10 readings per second. After a scan, straight line segments can be extracted. If the length of the line is on the order of a beam width, then there are two possible interpretations: either a straight line, or a corner. Additionally, these straight line readings inform the model that the region between the intelligent machine and the segment is free space. The consistency of theses

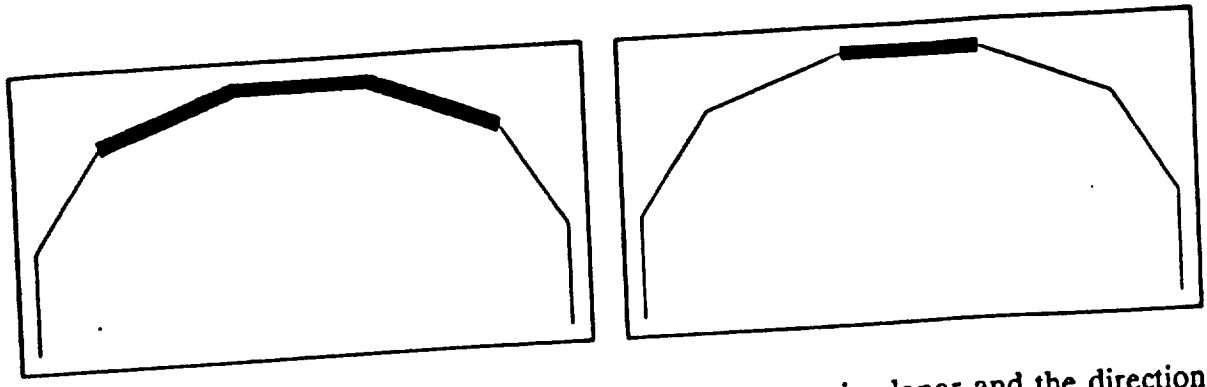
features with the map can be ascertained, and the map can be updated. Finally, those readings which can not be modelled as either straight lines, or corner points can be added to the model as representing a surface patch that could lie anywhere along a 30 arc. Because of its very low angular resolution but accurate depth measure, the acoustic system is useful in guarded moves.

As a last line of defense, if the vision and sensing system miss an object, there is a tactile sensing system, composed of twelve bumpers with internal contact switches along the edges of a nonregular dodecagon. In addition to protection, when an autonomous system accidentally crashes, the bumpers provide very definite information about the presence and location of an object. This form of data can be added to the model. Assuming that only one object is contacted at a time, the geometry of the bumpers allow the following interpretations.

1) If two adjacent bumpers are contacted, then the contact is a corner and the corner point can be localized with a fair degree of certainty. If the contact point is part of a wall, according to the already created map, then the bounds of that wall are detected.



2) If only one bumper is contacted, then the point of contact is a uniform distribution along the length and depth of the bumper.



3) If three adjacent bumpers are contacted, the contact is planar and the direction of the contact is that of the central bumper with an uncertainty based on the depth of the bumpers.

Finally, odometry determines the intelligent machine's position. As all sensing devices are on the robot, all the measurements are relative to the position of the whole autonomous system. Thus, the robot's orientation must be accurately measured for the creation of a reliable world model. For this purpose, as the intelligent machine moves, shaft encoder readings are mapped through the kinematics, to determine robot velocity, and are integrated to calculate vehicle location.

This model requires no a priori knowledge of the environment, but needs the preexistence of some object definitions, in order to combine low level data to higher level information. The uncertainty involved in this method can be reduced by applying a Kalman filter to the data. The hierarchical geometric world model was initially developed for navigation in buildings, and thus the higher level objects were walls, rooms, buildings etc.. In other environments, like manufacturing, the same model can be used but the objects this time will be cylinders, nuts, and other manufacturing parts.

2.7 Attributed Graph World Model

This world model described by A. C. Kak, A. J. Vayda, R. L. Cromwell, W. Y. Kim, and C. H. Chen [24] is an attributed relational graph in which both nodes and arcs have attributes. The nodes are surfaces and the arcs are relations between surfaces. Each individual object model is a connected graph and the scene description is a graph but it is not necessarily connected.

The first step is to derive a boundary representation of the object model from the Constructive Solid Geometry (CSG) representation. As already mentioned, the CSG representation of a complex object has a tree structure, where the leaf nodes are primitive objects and the non-leaf nodes are primitive operations. The set of primitive objects used, depends on the particular system, but the most common ones are block, cylinder, sphere, and cone. The primitive operations are union, intersection, and difference. With a sufficient set of primitive objects and these three primitive operations, any arbitrary complex object may be defined. For example, for primitive objects, the surface representation is simple: a block has 6 surfaces and 12 adjacency relations, a cylinder has 3 surfaces and 2 adjacency relations. Complex objects have larger graph representations.

This attributed graph world model is actually implemented by using Prolog clauses. The justification for this selection is that Prolog's declarative structure lends itself well to this type of task. The database consists of three type of facts which specify objects, surfaces and relations.

An object has an identifying name, a specification of its type (block, cylinder, toroid, etc.), a list of surfaces that it is comprised of, and a list of relations between those surfaces.

object(Name,Type,Surfaces,Relations)

Every surface has at least five attributes: the identifying name, the type based on curvature properties (planar, cylindrical, conical, spherical, ellipsoidal, and toroidal), the area, and the position and orientation of the surface which is derived differently for each type of surface. Other attributes may be specified as necessary.

surface(Name,Type,Area,Position,Orientation,Attributes)

Each relation has a name, a type, and two surfaces that share in the relation. The most useful type of relation is the adjacency relation. Attributes are specified as necessary. Two useful attributes are the angle between the orientation values of the surfaces and the type of edge which separates the surfaces (concave, convex, jump).

relation(Name,Type,Attributes,Surface1,Surface2)

As an example, what follows is the object definition of a specific cylinder with diameter = 3.5" and length = 4".

```
object(cylinder1,
        cylinder,
        [surface(top,planar,9.6,_15,_16,[[depth,4]]),
         surface(cyl,cylindrical,44,_23,_24,[[depth,3.5]]),
         surface(bottom,planar,9.6,_31,_32,[[depth,4]])],
        [relation(_96,adjacent,[[angle,90],[edgetype,convex]],
              surface(top,planar,9.6,_15,_16,[[depth,4]]),
              surface(cyl,cylindrical,44,_23,_24,[[depth,3.5]])),
         relation(_103,adjacent,[[angle,90],[edgetype,convex]],
```

surface(cyl,cylindrical,44,_23,_24,[[depth,3.5]]),

surface(bottom,planar,9.6,_31,_32,[[depth,4]]))]).

The represented objects can be even more detailed by the addition of other properties.

For scene analysis, the problem is rephrased in partitioning the scene into subgraphs such that each subgraph is also a subgraph of a known object model. By replacing each subgraph by the corresponding complete object model graph, a 3-D description of the scene is formed. Inference checking can be used to ensure that the model of the scene is valid.

The input from the structured light scanning gives the x, y, z coordinates of a set of points on surfaces. The range map determined from these coordinates allows computations of surface curvatures and surface normals. With range, curvature, and surface normal information, the segmentation of the scene into distinct surfaces can be accomplished. Next, the attributes of these surfaces are found, and relationships between surfaces are determined.

This graph scheme requires some a priori knowledge. Once surfaces are totally described and the relations among them are detected, the information for recognizing the represented object must preexist in some database. Still, for an intelligent machine to make a map or generally picture the environment, it is not absolutely necessary to identify the type of obstacles that are surrounding it. What is actually required depends on the robot's application. The model is flexible and expandable, as there is no limit in the number of different types of objects that can be identified and in the number of properties that can be included in relation, surface, and object descriptions. However, to extract all this information from only light scanning involves a high degree of computation, and

vulnerability to error.

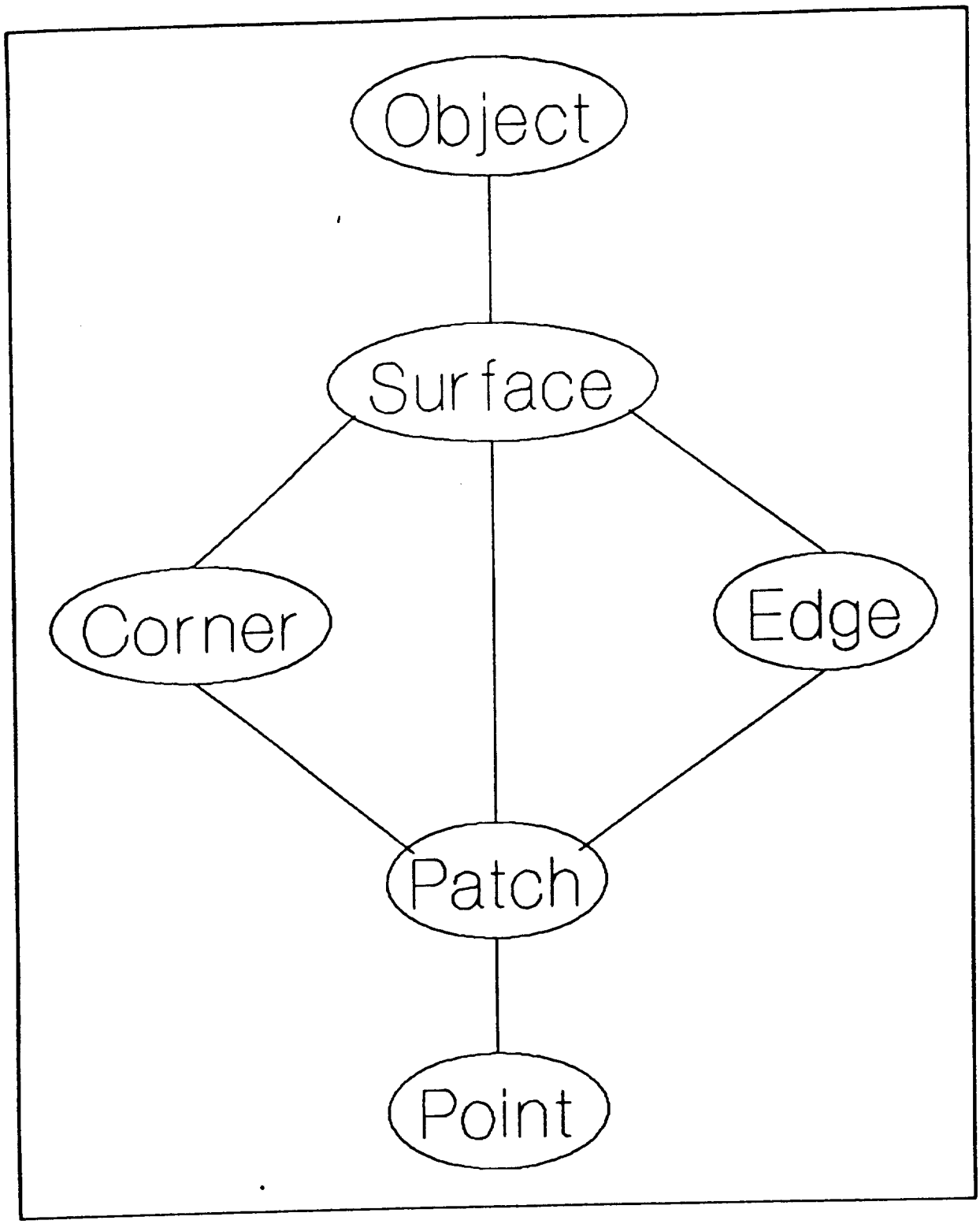
2.8 Feature Space Graph World Model

Francis Merat and Hsianglung Wu [34], on the other hand, came up with another representation scheme, the feature space graph world model. Their idea is to describe objects in terms of features, where a feature is some relation defined on a closed set of points. To represent an object a feature space like the one in the following figure is defined.

Points are entities containing position, normal, and curvature measurements. A *patch* is a small area on a surface and is denoted by: the centroid of the patch, the curvatures in various directions at the centroid, the patch class, and the neighboring relations between the patch and the neighboring patches. A *surface* is a closed set (graph) of connected patches, which have uniform properties. An *object* is a set (graph) of connected surfaces with a set size greater than one.

Based on the feature space hierarchy, objects are described in terms of features. The description of an object includes the surface equation, the orientation, and the centroid of the surface or object under examination. The properties of a patch or point can be easily derived from the surface equation and are left out in the final object description.

As an example, the feature space graph model would represent the information for the following object as described further on.



World Feature Space

Object Name: Sphere-on-Block

Object Id: 1 *Feature Level: Object*

Centroid: (0,0,0) *Orientation: (0,0,0)*

Surface_Graph: (0,1), (1,2), (1,4), (1,5), (1,6), (2,3),
(2,5), (2,6), (3,4), (3,5), (3,6), (4,5), (4,6)

Reference Surface: 0

Surface Name: Sphere

Surface Id: 0 *Feature Level: Surface*

Centroid: (0,0,0) *Orientation: (0,0,0)*

Surface Equation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -2.25 \end{bmatrix}$$

Neighboring Surface: 1

Surface Name: Plane

Surface Id: 1 *Feature Level: Surface*

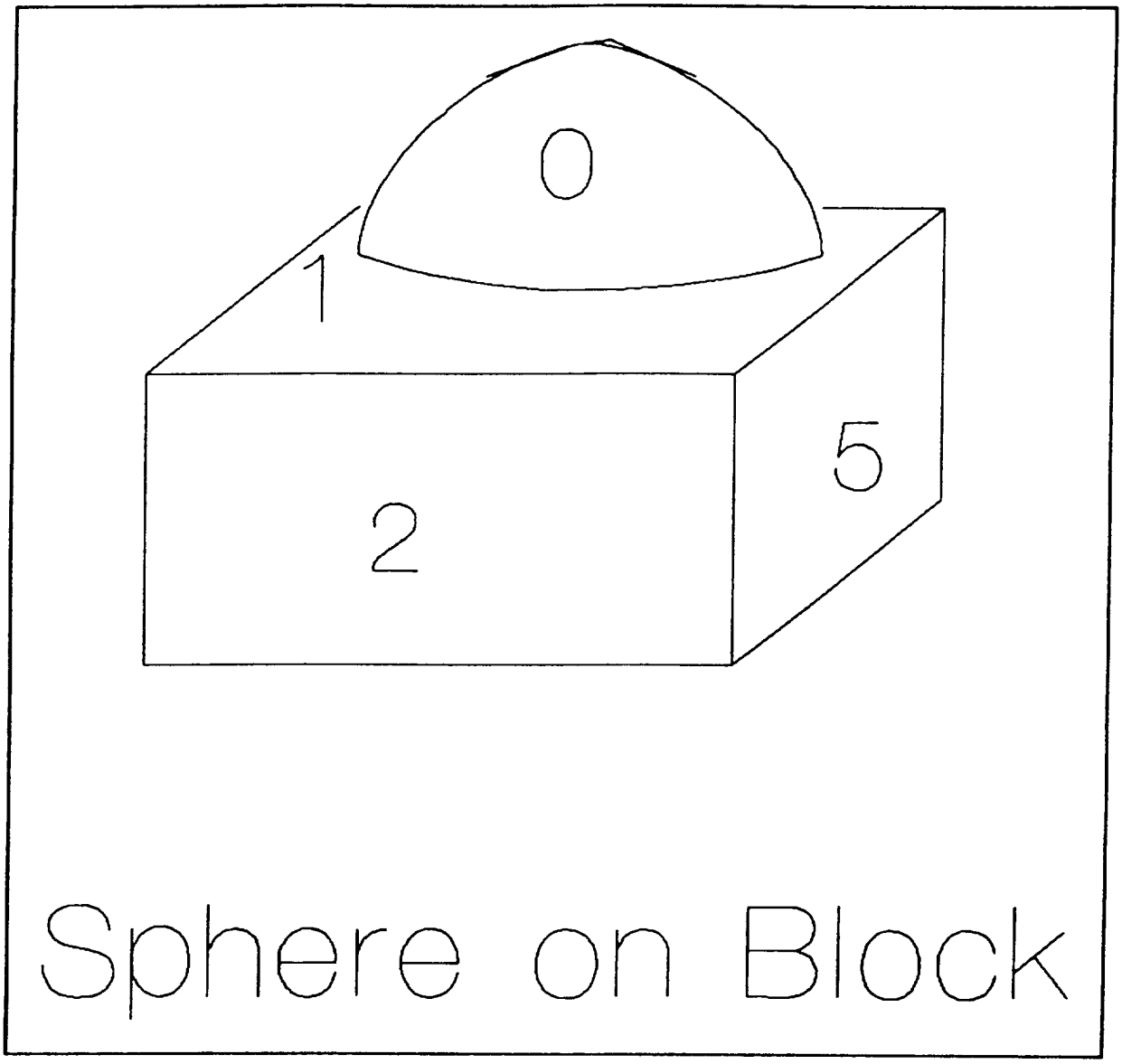
Centroid: (0,0,0) *Orientation: (0,0,0)*

Surface Equation:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0 \end{bmatrix}$$

Neighboring Surface: 0, 2, 4, 5, 6

Other Surfaces are similar to surface 1.



The process of creating the model is innovative. As a first step, low level features are extracted from sparse range vision data, which gives the capacity of generating partial object descriptions. What follows is *Feature Extraction by Demands* (FED). This method feeds back the partial descriptions to guide the feature extraction process to extract more detailed information from interesting areas, which can be used to refine the object

description. Regions which are not perceived to contain useful information will be ignored in further processing. As a more complete object description is generated, FED converges from bottom-up image processing to top-down hypotheses verification to generate complete hierarchical object descriptions.

This technique, FED, together with a concurrent processing scheme, can generate object descriptions more efficiently than sequential methods. The method is very robust because features can be extracted from local analysis and verified globally which means a smaller chance of missing features. Finally, the feature space graph model is general and expandable in the sense that many man-made (i.e. manufactured) objects can be modeled with objects containing quadric surfaces and that the processing is independent of the specific type of range sensor employed. This representation scheme, requires no a priori knowledge. The object description can be generated in the early phases of operation, called the learning period, or in the processing phase, where the object may have extrinsic information.

2.9 Visibility Graph World Model

This world model by Nageswara S. V. Rao, S. S. Iyengar, C. C. Jorgensen and C. W. Weisbin [38], uses a visibility graph of an environment O, denoted as VG(O). Formally, VG(O) is defined as a graph (V,E) where,

1) V is a set of all vertices of all obstacles, and

2) The line joining two vertices u and v, u,vϵV, forms an edge (u,v)ϵE, if it is not obstructed by an obstacle.

VG(O) is an undirected graph and is unique for a given environment.

However, for this model to work, a couple of assumptions have to be made. First of all, a finite sized robot is placed in an obstacle workspace, called terrain, populated by unknown but finite number of polygonal objects of varied sizes and locations in the plane. In addition, the environment is considered to be of finite size, which means that there exists a circle of radius $R>0$ which contains all the obstacles. Finally, the sensory devices, which can be of any type, should be such as to be capable of detecting all the object vertices and edges that are visible from the present location of the intelligent machine.

The exploration of the environment and the creation of the world model starts at any arbitrary point in the obstacle terrain. The robot scans and moves to the nearest obstacle vertex. This is considered the starting vertex. The autonomous system then moves from vertex to vertex in a systematic manner. When a vertex is visited for the first time, a "scan" operation is performed. Let the robot be located for the first time at vertex v. The

adjacency list of v, in VG(O), is built by detecting all the vertices visible from v using the scan operation. The vertex v is marked as *visited* and then pushed onto a stack. There are two cases:

1) If v has unvisited adjacent nodes, then the robot moves to a node, say w, which is nearest to v among the unvisited adjacent nodes. From w the same process starts again.

2) If all adjacent nodes of v are *visited*, then the nodes on the stack are repeatedly popped till a node x with at least one unvisited adjacent node is obtained. Then the shortest paths to each of the unvisited adjacent nodes of x are computed using Dijkstra's shortest path algorithm. The robot chooses the shortest path among the computed ones, and moves to the corresponding unvisited node w. From w the same process starts again.

The complete world model is built when the robot is located at vertex u such that, all nodes adjacent to u are visited, and the adjacent nodes of each node on the stack are visited. At this point the autonomous system moves back to the starting vertex along the shortest path.

For this process to work, it is assumed that the visibility graph of the environment of polygonal objects in the plane is connected, which means that there exists a path between any two nodes. It is also assumed that the order in which the unexplored vertices of obstacles are visited by the robot is exactly the same as the order in which the new nodes of VG(O) are visited by a depth-first-search algorithm (if VG(O) were available).

Although the visibility graph world model seems simple to perceive, and graph traversal and creation algorithms are well established, it makes too many assumptions, which make its implementation too dependent on the existence of a specific environment.

In addition, although for each obstacle vertex only its position needs to be stored (except its adjacency list), graph search and shortest path algorithms are neither simple nor fast.

2.10 Face-to-Face Composition Graph Model

Two other researchers, Leila De Floriani and George Nagy [11], proposed a formal representation of a family of solid objects for advanced engineering applications, called the Face-to-Face Composition (FFC) graph. This is a multi-rooted hierarchical structure based on boundary representation and is capable of accommodating different conceptual views of the same object.

The FFC graph of an object is a directed acyclic multigraph. Each node represents a valid single-shell volumetric component (a shell is any maximally connected set of faces on the building surface of an object). Arcs between nodes correspond to pairs of perfectly abutting connection faces. If an object consists of disconnected, non-contiguous components, then these components correspond to different connected components of the FFC graph. However, a single connected component of the FFC graph can describe an object consisting of multiple shells.

Single nodes are internally described according to one of the accepted boundary models. The definition of the FFC graph is independent of the particular model chosen to represent individual components. This model is, therefore , modular as any geometric or topological modification of a single component, which does not affect its connection entities, is local to that particular component.

Each node has one or more parents, except for an (arbitrary) set of root-nodes called the base of the FFC graph. The base may be the largest component, the baseplate, or the

floor, on which everything rests, or any other component chosen as the starting point. The resulting hierarchy defines a valid partial order for constructing the object starting at the base by successive addition or subtraction operations.

At an abstract level, each component of the FFC graph can be viewed as the collection of its connection faces, which define the interface of such a component. Connection loops, edges and vertices of a single component of the FFC graph are attached to their connecting faces.

An FFC graph of an object can be constructed by building its single components separately and then combining them by successive pairwise composition of distinct FFC graphs. A node in the FFC graph contains the boundary description of a component and thus can be constructed from sensory data. With each primitive topological entity (face, edge, vertex) an appropriate geometric descriptor is associated. The adjacency topology and the geometry are well separated and in principle, a parametric representation of the surface can be accomplished. Complex objects can be constructed from distinct models of simpler models through merging operations.

What is new and different in this scheme from other graph models is the imposition, either by the user or by an algorithm, of an arbitrary, but valid, partial order of object components. The model allows for flexibility in the representation used in single-shell components. As a graph, it does not make a very efficient use of storage. No a priori knowledge is needed. This method is better suited for design and manufacturing applications than navigation and recognition projects.

2.11 Topological World Model

This method, introduced by Benjamin J. Kuipers and Yung-Tai Byun [30], is inspired by the study of cognitive maps which humans use. The topological model consists of nodes and arcs, corresponding to *distinctive places* (DPs) and *local travel edges* linking nearby distinctive places. A place in the environment corresponding to a node in the topological model must be logically distinctive within its immediate neighborhood by one geometric criterion or another. Each distinctive place has its signature, which is defined to be: the subset of features, the distinctiveness measures, and the feature values, which are maximized at the place. A hill-climbing search is used to identify and recognize a distinctive place when the robot is in its neighborhood. While exploring, both the signature and the local maximum must be found. While returning to a known place, a robot is guided by the known signature. *Travel edges* corresponding to arcs are defined by *Local Control Strategies (LCS)*, which describe how the autonomous system can follow the link connecting two distinctive places.

A *set of rules* is used to decide whether a robot instance is in the neighborhood of a *distinctive place* (DP) and what distinctive features can be maximized in the neighborhood. Each rule consists of assumptions and a decision for the distinctive features. Once the robot instance knows what distinctive features can be maximized locally in the neighborhood of a distinctive place (DP), a hill-climbing search is performed around the neighborhood looking for the point of maximum distinctiveness. When a distinctive place

is identified, it is added to the topological model with its distinctiveness measures, connectivity to edges, and metrical information. Some measures include the following:

1) Extent of distance differences to near objects.

2) Extent and quality of symmetry across the center of the robot or a line.

3) Temporal discontinuity in one or more sensors, given a small step.

4) Number of directions of reasonable motion provided by the distinct open spaces, with a small step.

5) The point along a path that minimizes or maximizes lateral distance readings.

Travel edges, on the other hand, are defined in terms of Local Control Strategies (LCS). Once a distinctive place has been identified, the robot moves to another place by choosing an appropriate control strategy. While following an edge with a chosen strategy, the robot continues to analyze its sensory input for evidence of new distinctive features. Once the next place has been identified and defined, the arc connecting the two distinctive places is procedurally defined in terms of the LCS required to follow it.

Another *set of production rules* is used to decide a proper *Local Control Strategy (LCS)* depending on the current sensory information. The current LCSs are:

1) Follow-Midline. Follow the midline of a corridor.

2) Walk-Along-Object-Right. Walk along the right side of a large space.

3) Walk-Along-Object-Left. Walk along the left side of a large space.

4) Blind-Step. Walk blindly.

The current position is described topologically, rather than metrically. When a robot instance is at a distinctive place, the current position is described by: the current place name, the current orientation in degrees, and the travel edge through which the intelligent machine instance has come to the current place from the previous place. When a robot

instance is on an edge, the current position is described by: the previous place name, the current orientation, and an "On-Edge" indication.

While the autonomous machine explores, it uses an exploration agenda to keep information about where and in which direction it should explore further to complete its exploration. If the exploration agenda is empty, it means that there is no known place with directions requiring further exploration.

Topological world modeling was tried in the same environment with 0%, 5%, and 10% error rates in sensor readings. In all cases the correct map was constructed, but as the error level increased, the correct path was found in repeated trials, making the process much slower.

Generally, this modeling scheme overcomes the high vulnerability to metrical inaccuracy in sensory devices and movement stimulators. This method does not depend critically on the choice of sensors and movement actuators. In environments dominated by obstacles and extended landmarks, a topological map provides a more robust environmental representation than, for example, regions related by adjacency.

Still, local geometry, shape of near objects, distances and directions to obstacles etc. is metrical information and as such subject to error. However, averaging and continuous accumulation of this data in the exploration and navigation stage minimizes metrical error. In addition, continuous sensory feedback is used to eliminate cumulative error.

2.12 Space-Time Octree World Model

The main characteristic of this world model designed by Kikuo Fujimura and Hanan Samet [16] is that it includes time as one of its dimensions. In other words, a three dimensional space representation is used, where time is the third dimension. (Of course, in other applications, a four dimensional space representation can be used, where time will be the fourth dimension). An object, say O, moving in a two-dimensional plane can be regarded as a three-dimensional stationary object whose volume is the trajectory that is swept as it moves. If a point (x,y,t) is inside that volume in space-time, then the two-dimensional point (x,y) is occupied by object O at time t. Therefore, an interference between two objects in three-dimensional space means that a collision has occurred in the two-dimensional plane. Note that two different objects which occupy the same location at different times don't collide, and will occupy different locations in space-time.

Assuming that the motion of the obstacles doesn't involve rotation, as long as a polygon moves at a constant speed, the trajectory (i.e. the volume swept by the polygon) becomes a polyhedron in three dimensions. A polyhedron can be modeled in terms of its vertices, edges, and surfaces. A tree structure, serving as an index to the world model yields efficient access to a location.

Everything in the workspace is defined in a world with bounded x, y, and t values. A point in the space is represented by (x,y,t) where $x_1 < x < x_2$, $y_1 < y < y_2$, and $t_1 < t < t_2$. x and y are measured in terms of distance, while t corresponds to time. Usually, it is convenient

to let $x_1 = y_1 = t_1 = 0$ and $x_2 = y_2$. Note, that time is also bounded. In this world, every motion of an object on the two-dimensional plane during the time period t_1 and t_2 is represented as a three dimensional object.

The index tree is built by repeatedly subdividing three dimensional space-time into eight subspaces of equal size called cells, until each cell satisfies one of the following conditions:

1) A cell contains part of the trajectory of a *vertex* of an obstacle.

2) A cell doesn't contain any part of the trajectory of a vertex, but contains part of the trajectory of an *edge* of an obstacle.

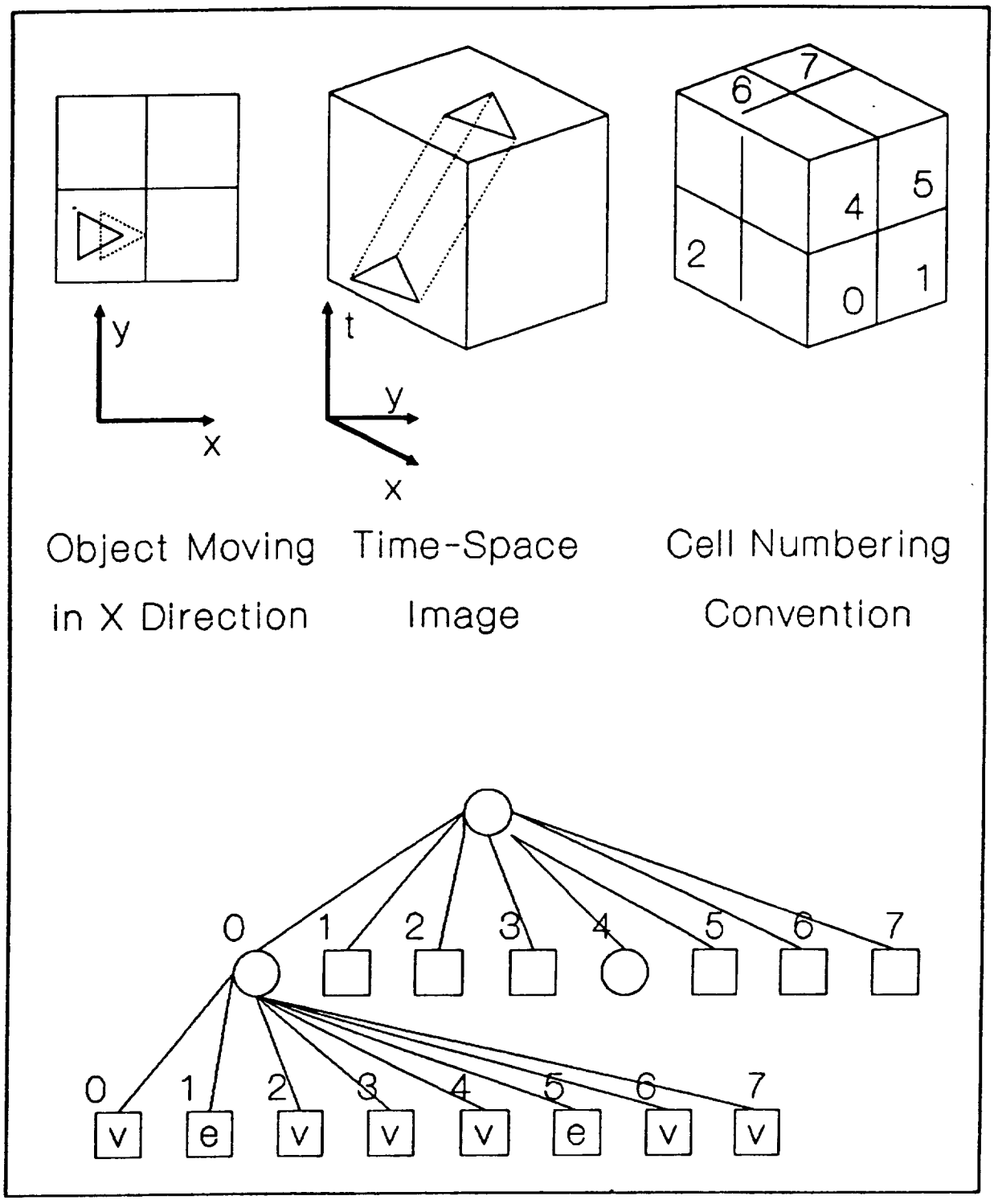
3) A cell doesn't contain any part of the trajectory, so it is *empty*.

4) A cell is entirely contained in the trajectory, and thus is *full*.

The cells defined by these criteria are called respectively *vertex* cells, *edge* cells, *empty* cells, and *full* cells. This decomposition of space is similar to the one followed in the octree representation.

Building this space-time tree is also performed in a way similar to simple octree creation. Initially, the entire workspace is treated as a single cell which is represented as a tree containing one node. If any of the conditions 1 through 4 are violated by this cell, then the cell is subdivided in eight equal sized cells, and these resulting cells are checked for violation of conditions 1 through 4. This process is applied recursively.

The space-time octree representation is based on a cell decomposition scheme, in which each cell, in other words each leaf node, has a simple geometry, i.e. it contains at most the (x,y,t) coordinates of one vertex, or one edge of an obstacle. As the time stamp is added, this world model is especially useful in the representation of environments where moving objects exist. In these cases, this method allows to regard the moving obstacles as



Space-Time Octree Construction

being stationary in the extended world.

This octree scheme is an expandable representation. No a priori knowledge is needed, but information regarding obstacles would improve the performance of the model. Still, thinking of time as the third dimension is not a very familiar concept and is useful only in time sensitive applications. If time information is not needed, then time should not be used as an extra dimension, especially in 3-D representations, where time would be the fourth dimension.

2.13 Occupancy Grid World Model

The occupancy grid model, promoted by Alberto Elfes and Larry Matthies [13,33], is a multidimensional random field which maintains stochastic elements of the occupancy state of the cells in a spatial lattice. It employs probabilistic sensor interpretation models and random field representation schemes. Operations are performed directly on the occupancy grid for a variety of robot tasks.

This representation employs a multidimensional (usually either 2-D or 3-D) tessellation of space into cells, where each cell stores a probabilistic estimate of its state. The cell states are exclusive and exhaustive, (P[S(C)]=OCCupied) + (P[S(C)]=EMPty) = 1, where S(C): state variable associated with cell C of grid.

The range data obtained from a given sensor r, is related to the true parameter space range value z, by a probability density function p(r/z). This density function is subsequently used in Bayesian estimation procedure to determine the occupancy grid cell state probabilities.

Cells that have not been observed before, have an occupancy probability of 0.5. There is an incremental composition of sensory information. Given a current estimate of the state of a cell C_i, $P[S(C_i)=OCC/\{r\}_t]$, based on observations $\{r\}_t=\{r_1,r_2,..,r_{t-1},r_t\}$ and given a new observation r_{t+1}, the improved estimate is given by

$$P[S(C_i)=OCC/\{r\}_{t+1}]=(p[r_{t+1}/S(C_i)=OCC]*P[S(C_i)=OCC/\{r\}_t])/$$
$$\sum_{S(Ci)}(p[r_{t+1}/S(C_i)]*P[S(C_i)/\{r\}_t])$$

In this recursive formulation, the previous estimate of the cell state, $P[S(C_i)=OCC/\{r\}_t]$, serves as the prior and is obtained directly from the occupancy grid. The new cell state estimate, $P[S(C_i)=OCC/\{r\}_{t+1}]$ is subsequently stored again in the map.

A combination of different sensing devices can be used, as the same occupancy grid can be updated by multiple sensors operating independently. Still, using a different estimation method, separate occupancy grids can be maintained for each sensor system and in some later stage all these sensor maps are integrated.

The occupancy grid model can be used in both unknown environments, and in environments for which some prior knowledge is available. In this second case, the occupancy grid framework incorporates information from precompiled maps.

An optimal estimate of the state of a cell is given by the maximum a posteriori (MAP) decision rule:

- a cell C is *occupied* if $P[S(C)=OCC] > P[S(C)=EMP]$;
- a cell C is *empty* if $P[S(C)=OCC] < P[S(C)=EMP]$;
- a cell C is *unknown* if $P[S(C)=OCC] = P[S(C)=EMP]$.

Other decision making criteria that can be used are minimum-cost estimates, or employment of an unknown band (instead of a threshold value.)

Occupancy grids can also be used in a different mode, in three-map world models, where many local maps are combined in a global map. A single sensor's data is called a sensor view. Various sensor views can be composed into a local sensor map. Different local sensor maps might correspond to different sensor types. Finally, local maps from multiple data gathering locations are composed into a global map of the environment.

Thus, occupational grids can take advantage of the existence of a priori knowledge, but can be used as efficiently with no precompiled geometric models. No runtime

segmentation decisions are necessary. The updating method of this representation scheme allows observations performed in the remote past to become increasingly uncertain, while recent observations suffer little blurring. The occupancy grid is a stochastic spatial world model. It is possible to derive higher-level geometric representations or voxel models from the grid. In addition, better world models and disambiguous sensor data are achieved because of additional sensing, not because of additional assumptions or finer tuned heuristics. Generally, the occupancy grid representation is simple to manipulate, and treats different sensors uniformly. Since all sensor readings have a common interpretation and make comparable statements in the grid framework, the sensor integration problem becomes relatively straightforward. This model can be applied for the detection of moving objects over sequences of maps. The occupancy grids represent a fundamental departure from traditional approaches to intelligent machine perception and spatial reasoning.

However, this model has the drawback of fixed size representation, which makes expandability hard and storage space consuming. Its major shortcoming is that the size of the representation and the cost of the update increases linearly with the surface of the world, and quadratically with the accuracy of the representation.

2.14 Volumetric World Model

In the volumetric world model, proposed by Yuval Roth-Tabak and Ramesh Jain [40], space is partitioned into a 3-D matrix of cubic voxels. Dense range data from multiple viewpoints in an environment refines the 3-D voxel based volumetric model of that environment. This world model permits annotations, additions and temporary overlays, so that unexpected information and interesting features can be registered with both object and sensory information.

The voxels in the 3-D volumetric grid are assigned three possible values: *void*, for empty voxels that represent an open piece of space; *full*, for occupied voxels; and *unknown*, for voxels for which no meaningful information has yet been obtained. In this model, no certainty levels are assigned to the attributes for the following reasons:

1) Dense range sensors, unlike ultrasonic sensors, do not impose any uncertainty on the location of the actual obstacles.

2) Dense range data provides readings for all the pixels in the image, and hence there are no spatial gaps of the depth information.

3) The updating technique is model-driven, which means it uses knowledge already stored in the model. If certainty levels were employed, in each updating step the whole grid would be scanned, and the whole operation would be much slower.

4) Uncertainties are treated globally by using certain thresholds that can be altered adaptively.

The model is initially entirely unknown. The successive exploring algorithm can be described as follows:

1) Only voxels within the scope of the sensor are checked.

2) Only voxels not yet void are checked.

3) For each of the voxels actually checked, all of the eight vertices are checked and compared to the actual pixel in the range image that points to their position in space.

4) If the maximum distance to any of the eight vertices is smaller than the minimum range pointed by any of the range pixels, then the voxel is *void*.

5) If the range of the vertices' distances intersects with the range pixels, and the difference between the maximum and the minimum range pixels is within a certain threshold, then a voxel is *full*.

6) Else it is unchanged.

The fact that eight vertices are being checked has an inherent smoothing effect on the result. In most cases, not all vertices will fall within the same range pixel. Hence, to a certain extent, noisy images will not have a strong impact on the result. In the fourth step, a certain threshold margin can be added to the above requirement in cases with some location uncertainty of known extent. This margin represents the worst case error that might result from such a location uncertainty. The threshold on the fifth step is introduced to avoid assigning full values to voxels which lie on, or near sharp range discontinuities.

Experiments pointed out that the method is not susceptible to noise. Whereas the original design requires no previous knowledge of the environment, precompiled maps could be used when available. Although only dense range sensors were used in the initial installation of the model, other types of sensors can be used and are being implemented. By comparing information between the expected scene and the viewed scene, detection of

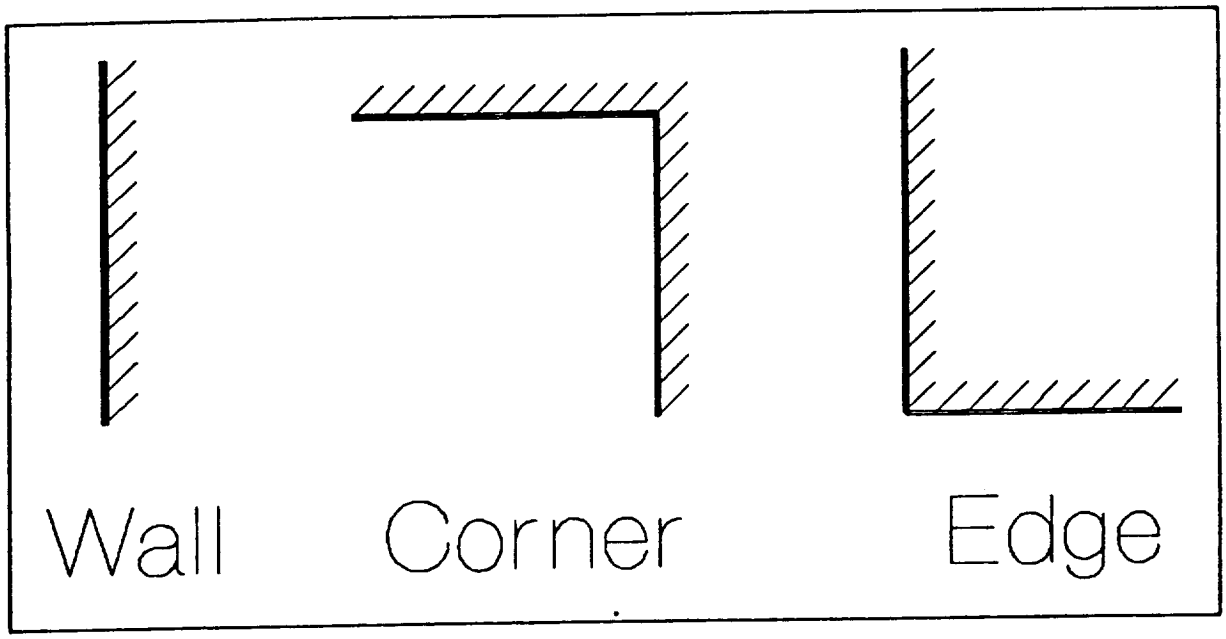
changes and movements in scene can be achieved.

On the other hand, volumetric world model is a static representation, that can not expand in size as the environment expands. If a large maximum workspace is specified, too much storage is wasted. In addition, the updating algorithm, in order to avoid storing uncertainties, checks all eight vertices, resulting in time consuming algorithms.

2.15 Visible Grid World Model

The visibel grid world model, introduced by Roman Kuc and M. W. Siegel [29] represents the floor plan of the environment as a two dimensional grid of *visibels*, which indicate which elements are visible from a particular location. A *visibel* is represented by a word in computer memory, each bit of which is assigned to a particular element.

In the initial implementation of the model three different elements are used: corners, edges, and walls. These three components compose a cew world. Walls are simple planes, while corners and edges are located at the intersections of planes. As acoustic sensors are used, corners like walls produce reflections, while edges produce diffracted signals.



A wall is detected only by the reflection that bounces directly back to the transducer. For corners and edges to be visible, the transducer must have reflections bounces back to

it from both planes defining the element.

Once an element in some position is detected, the *visibel* corresponding to that position is updated. Each bit of the word corresponds to a particular element, and that bit is set to 1, if the corresponding element is detected, or is set to 0 otherwise. The members of the grid on the boundary have their visibels set to -1. A grid set to all 0s means either an unexplored space, or an empty space. If the visibels however, have some special value (a dedicated bit) to differentiate between empty and unexplored cells, the model can become even more accurate.

As a conclusion, this model is conceptually simple. It is expandable as by changing (increasing the number of bits) in the internal representation of *visibels*, a bigger number of different objects can be stored. Bit manipulation is not easy, but makes an efficient use of storage. In addition, if wanted, a priori knowledge can be used. Finally, the model is flexible as it can be applied to higher-level, and lower-level models by simply changing the level of the elements composing the world.

2.16 Three-Map World Model

The model presented by Minoru Asada [5] consists of three kinds of maps, a *sensor* map, a *local* map and a *global* map. Any kind of sensory data can be used. Each sensor has its own coordinate system. Each sensor builds its own sensor map, which is nothing more than the recorded input sensory data.

Then the local map builder builds local maps from sensor maps. While building the local maps all the sensor maps are transformed, if needed to a robot centered coordinate system. Then the *local* map, or otherwise called height map is segmented into *unexplored*, *occluded*, *traversable*, and *obstacle* regions. Initially, the height map consists of two types of regions: those in which information is available and those for which no data is obtainable. The latter regions are classified into unexplored or occluded regions. Unexplored regions are outside the visual field of the sensors. The remaining regions in this category are labelled as occluded regions. Some regions, which are not actually occluded may be classified as such, due to inadequate information. Finding traversable regions is straightforward. As regions occupied by obstacles have high slope and high curvature, while traversable regions have low slope and low curvature, the process of differentiating between them is not complex.

If needed, a further refinement is to classify the obstacle regions into *artificial objects* or *parts of natural objects*. For obstacle classification both the local map and the sensor map of the intensity image are used. Each segmented region is classified according to the

following criteria:

1) If a region has sufficient size (larger than a predetermined threshold), constant slope (small variance of slope), and low curvature (low mean curvature and small variance of the curvature), then the region is an *artificial object.*

2) If a region has sufficient size and high curvature (high mean curvature and large variance of the curvature) and large variance of the brightness of the intensity image, then the region is a *part of a natural object.*

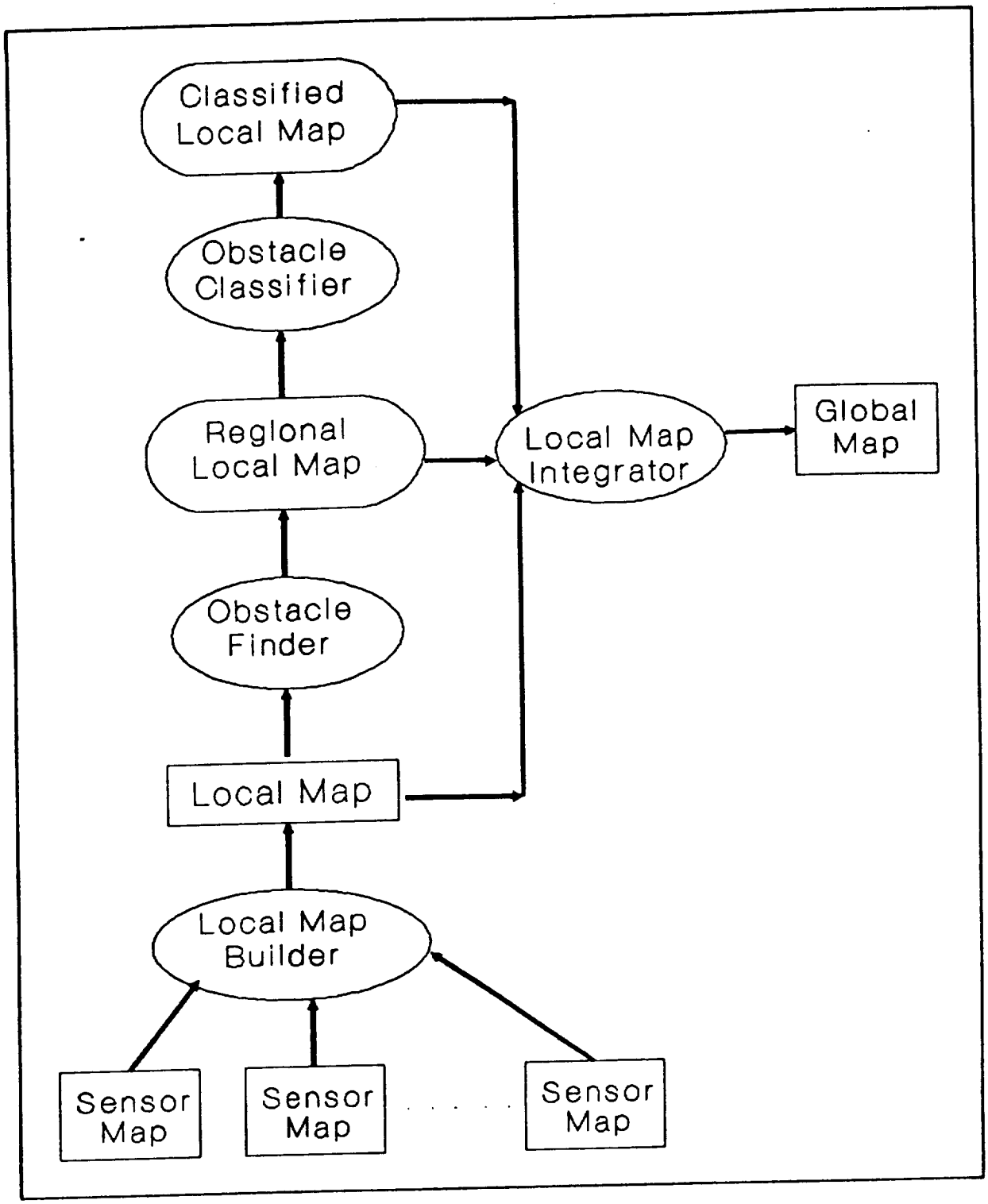
3) Otherwise, the region is regarded as *uncertain* in the current system.

Finally, the global map is constructed. During the motion of the intelligent machine, the world model produces a sequence of local maps built at different observation stations. These maps are integrated into a global map in the robot centered coordinate system. The local map integrator consists of two parts, the first one matches two different local maps to determine the correct motion parameters of the robot, and the second updates the description of region properties.

This three-map model has the advantage of storing all sensor information, all intermediate information, and the final global representation. This allows the model to use and combine different data to extract more information about the environment with greater accuracy and less uncertainty.

However, all these maps (sensor, local, global) and their refinements (regional local map, classified local map) use extensive storage. Even when all this information is needed, manipulating it for obtaining greater detail and accuracy, means more complex and time-consuming algorithms.

Above all, the concept of segmenting the information into different levels gives the flexibility of better manipulated workspace knowledge according to the desired level for the



Map Building System

specific task to be accomplished.

2.17 Generic World Model

The idea behind the generic model is to have a single model which describes a broad class of objects. This model was highly supported by David J. Kriegman and Thomas O. Binfold [27]. A generic model should not represent a particular environment. Instead, one model for a specific class exists, which includes the class of features that can be found in any object of that class.

Given sensor information, the generic model can be partially constrained until, ultimately, there is an instantiation that represents the actual workspace. In general, it will not be possible to instantiate the generic model fully, but instead, sensing will impose enough constraints necessary for the task.

A generic object should be described in terms of its function, or purpose, as well as physical constraints. A generic model is composed of the following five aspects: *classes*, *sets*, *numbers*, *mappings*, and *constraints*.

The generic model of an object is named *class* and is made up of named components and constraints. Components are typed, and the types may be either another class, a set, or an element of a set. A *set* may have elements which are either classes, or themselves sets. Sets need not be finitely enumerated but may be infinite sets, where membership is determined by the set theoretic definition of membership, which is satisfaction of a constraint (predicate). Set operations on infinite sets are represented as Boolean operations on the constraints of the set.

Additionally, classes can represent *mappings* from a domain to a range. This is used to represent geometric objects according to their mathematical definitions.

Constraints describe the relationship between components and subcomponents. They may simply be algebraic and Boolean constraints between components with a numeric type (e.g. pythagorean theorem). Still, other constraints may have to do with the element types, such as two planar faces being parallel. Symbolic constraints are defined by name along with the type of the objects which are being constrained. This allows for geometric reasoning without having to do more costly symbolic algebraic reasoning. Symbolic constraints may be expandable into algebraic, Boolean, or even further geometric constraints. By expanding symbolic constraints into algebraic constraints, algebraic constraint manipulation can be used. Furthermore, constraints may be quantified over sets.

Since classes are named, one must be careful about their scope; a class is always named with respect to a particular namespace. Finally, classes are defined in an object-oriented manner; a class may be defined as a subclass of another class or classes leading to taxonomies describable by a directed acyclic graph. A subclass is a strict specialization of the parent classes. Any constraint that is true for a parent class, is true for a child. A subclass will inherit components and constraints of the parent classes, leading to issues of multiple inheritance. If a component is defined by multiple ancestors, with different types, then the type is determined by iteratively comparing ancestor types. If the multiply inherited component type is a class, then the more specialized type is used as determined by the specialization directed acyclic graph of all classes. If neither class is a specialization of the other, then an attempt is made to create automatically a new class, which is a specialization of the types of the component from the two ancestor classes. However, this is not always possible because certain types may be incompatible. If the multiply inherited

components have set types, then the specialization of the type is the intersection of the two type sets.

While David J. Kriegman and Thomas O. Binfold [27] developed the theory of the generic world model and explained all its components and the relationships among them, the actual implementation was carried out by S. A. Stansfield [42].

In this installation, as the model must be able to handle the variations of the generic models, both spatial/geometric, as well as symbolic information must be stored. Taking these requirements into account, along with the premise of category theory that "people represent and reason about objects based upon features," a feature based model representation was used. This scheme consists of a *hierarchy of frames* and a spatial/geometric model called the *spatial polyhedron.*

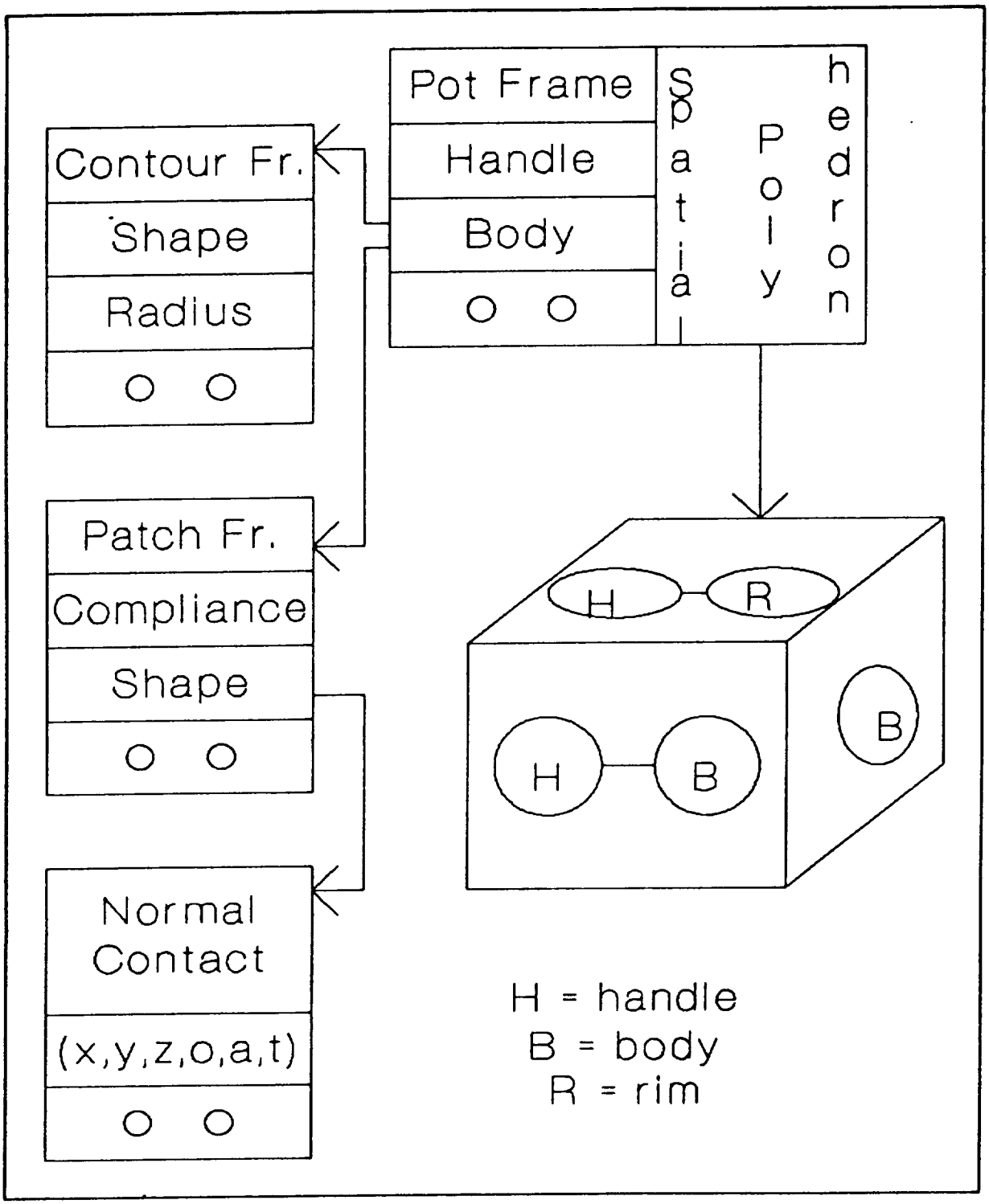
The idea in spatial polyhedron is that all of the infinite 2-D views of a 3-D object can be grouped into a finite set of equivalence classes. Informally, the *spatial polyhedral representation* may be described as follows. Imagine an object at the center of an n-sided polyhedron. If the object were to be viewed, or sensed, along a line normal to each face of this polyhedron, then certain components and features of the object would be viewable, while others would not. Slight changes in attitude as the viewer moves around the object will not result in any new features coming into view. When the viewer has moved sufficiently, however, then he will be sensing the object from a different perspective (or face of the spatial polyhedron) and different components and features will be viewable. Thus, an object is modeled by mapping to each face of the spatial polyhedron all of the features which are expected to be viewable along that face. This mapping consists of a list of these features and their appearance from the specified view.

The remainder of the object representation consists of a *hierarchy of frames.* At the

highest level, information about the object as a whole is stored. Intermediate levels contain the components which define the object. The features which parameterize these components are incorporated into the spatial polyhedron. This frame representation can carry such non-perceptual knowledge as function, ownership, etc.. Simpler objects use spatial polyhedron with fewer sides, while for more complex objects with larger numbers of components and features, more faces will be needed. In general, *frame hierarchy* contains perceptual information about the object, while the *spatial polyhedron* provides the spatial and relational information.

In more detail, the object to be identified is first processed visually to obtain 3-D edges and 2-D regions. These edges are then used to invoke a set of haptic (or touch) modules which do a further exploration of the object via a fixed set of Exploratory Procedures (i.e. hand movement strategies), to obtain a final set of features and components for the explored object.

The exploration is not object model-driven. The Exploratory Procedures are invoked based upon an initial, tactile, local exploration of the extracted visual features. This visual data is sparse and highly inaccurate and does not provide enough information. The sensed object is then matched against the object database using a form of prototype matching. reasoning is feature based. To determine if an instance is a member of the category, it is compared to the prototype for that category. It is not necessary for any of the objects in the category to have all of the defining attributes of the prototype. A similarity metric of some sort is applied to determine whether or not the object belongs to the category. The object is matched against the modeled prototypes using the extracted components, features, and their spatial relations. The matching requirements are, that each feature of the unknown object be present in the instantiated model, that it fit within the bounds of the



Representation of a Pot

upper and lower limits stored in the model, and that the relations between the instantiated model and the extracted features be the same. Simultaneously, the orientation of the spatial polyhedron is fixed for each matched model.

All reasoning modules are represented in Prolog. The model reasons from the more complex hypothesis to the less complex one. Thus it looks first for missing components and then for non-visible features of present components. For example, the Prolog implementation of a pot is:

```
object(one_handled_pot,50,300,80,150,450,300,3,
    [body,part],[body,handle]).
component(one_handled_pot,body,40,250,50,250,250,100,body).
component(one_handled_pot,part,50,10,10,200,20,20,handle).
face(one_handled_pot,2,
    [[body,contour,[rim,curved,0,[60,150,60,150]],rim],
     [handle,fpart,[large,one_extended],handle]],
    side1).
face(one_handled_pot,2,
    [[body,surface,[nonelastic,noncompliant,smooth,planar,
      [border,curved,0,[60,150,60,150]]],bottom_surface],
     [handle,fpart,[large,one_extended],handle]],
    side2).
face(one_handled_pot,2,
    [[body,surface,[nonelastic,noncompliant,smooth,
      curved,[]],side_surface],
     [handle,fpart,[small,stubby],handle]],
```

side3).

face(one_handled_pot,1,

[[body,surface,[nonelastic,noncompliant,smooth,

curved,[]],side_surface],

side4).

face(one_handled_pot,2,

[[body,surface,[nonelastic,noncompliant,smooth,

curved,[]],side_surface],

[handle,fpart,[large,one-extended],handle]],

side5).

face(one_handled_pot,2,

[[body,surface,[nonelastic,noncompliant,smooth,

curved,[]],side_surface],

[handle,fpart,[large,one-extended],handle]],

side6).

The generic world model, overall, allows flexibility. Under different applications, different features are emphasized. It also allows emphasizing on the function of an object or a workspace. The system is proven to be fast and robust. Additionally, beyond parameterizing the model of objects, this representation scheme allows for gross changes in object geometry and topology. Another advantage of this model is that it tries to follow human reasoning and recognition, and thus it is simple to conceive. People tend to divide the world into categories. When humans speak of cups or screwdrivers, they do not have, most of the times, a specific object in mind; they rather refer to the class of cups or screwdrivers. Above all, it is less time and space consuming to model the concept of a class

of objects (screwdriver), than to model each and every instantiation of the class (every different screwdriver). However, a priori knowledge and processing is required. The generic model must preexist for the sensory data to instantiate a specific object.

2.18 Multiple World Model

The latest trend in world modeling, as Peter K. Allen [3] emphasizes is to combine many different world models in one global scheme. The main characteristic of this world model is the use of multiple shape representations. Influenced by the tendency of using multiple sensory means, the basic idea behind a multiple world model is to use the best suited world representation for each sensory system, and then merge all these independent models in order to get an overall depiction of the environment.

In his implementation, Peter K. Allen [3] uses tactile sensory systems because of their ability to recognize attributes of three-dimensional objects quickly and accurately. Among these attributes are global shape, hardness, temperature, weight, size, articulation, and function. The objective is to identify hand movement strategies which are used by humans in discovering different attributes of three-dimensional objects. These hand movement strategies are called *Exploratory Procedures* (EPs). So far, EPs have reported success rates, 96-99%, in identifying different object properties using two handed, haptic exploration.

One major EP is *grasping by containment.* This exploratory technique derives sparse, but global, shape information. The recovered shape is represented in *superquadratics.* The main reasons for choosing superquadratics for this EP are:

1) The representation is volumetric by nature, which maps directly into the psychophysical perception processes suggested by grasping by containment.

2) The models can be constrained by the volumetric constraint implied by the joint positions on each finger.

3) The representation can be recovered with sparse amounts of point contact data since only a limited number of parameters need to be recovered. There are five parameters related to shape, and six related to position and orientation in space. Global deformations, like tapering and bending, add a few more.

4) In addition to the use of contact points of fingers on a surface, the surface normals from contacts can be used to describe a dual superquadric, which has the same analytical properties as the model itself.

5) The recovery process uses a non-linear least-squares estimate of a fit function. This approach is especially relevant with touch sensing, in which there is evidence that the human tactile system serves essentially as a low-pass filter.

This Exploratory Procedure obtains a number (typically 30-100) of finger contact points by encompassing the fingers of the hand around the object. The data is from all the sides of an object. Using superquadrics makes the shape estimator efficient, stable in the presence of noise and uncertainty, and able to use sparse, partial data. Thus, a good initial shape estimate is generated.

Another EP is the *lateral extent*. This is used to explore a continuous, homogeneous surface, such as a planar face, and to determine its extends. This EP uses the hand's index finger. An initial contact with the surface is made, and the Cartesian coordinates of the contact point are noted. The hand and arm then begin an iterative search for the boundaries of the surface by performing the following sequence:

1) Lift the finger off the surface until tactile contact is lost;

2) Move the arm in a direction parallel to the surface;

3) If the finger is in contact after the movement, note the new contact location, else lower the index finger until it makes contact with the surface again;

4) Repeat steps 1 through 3 until the finger fails to make contact in step 3.

A failure of contact can either be an edge, or a big distance between the surface and the finger. In the latter case, rechecking has to take place by moving the arm towards the surface.

Then a second mapping in the opposite direction follows, until an edge is detected. Finally, a third and fourth mapping takes place, by repeating the same search in both directions of a track perpendicular to the first two traces.

This procedure is able to map out a set of contact points on the surface, describing its extend. Each time a fingertip contacts the surface, the Cartesian coordinates of the contact are retained.

The data extracted from this procedure is mapped into a winged-edge type of *Face-Edge-Vertex model.*

Finally, another EP is the *contour follower.* This exploratory procedure is a dynamic procedure in which the hand maintains contact with a contour of the object. This EP reports information that can be used to recover a shape which can be represented as a class of generalized cylinders. (The class that Peter K. Allen [3] used was surfaces of revolution). The arm is moved to a location near one end of the explored object. The thumb and the index finger are opened enough to allow them to encompass the object without making contact with it. Then, first the thumb, and then the index finger, are slowly moved toward the object until the sensors detect contact between the finger and the object. The positions of the two contact locations are noted, and the fingers are retracted from the object so that no contact exists. The arm and hand are moved a small distance along the

axis of the explored object and the process is repeated, until the other edge of the object is reached.

The above EPs can be considered a set of primitive haptic functions to be used as the building blocks for an active, autonomous haptic recognition system. However, tactile sensing is not powerful enough, as any other single sensory system, to solely perceive and recognize the environment.

Using a multiple world model has many advantages. Fist of all, it is the ideal scheme for multiple sensory system, as each system will be updating the world model which is most compatible with its structure. Each sensor can act independently and work in parallel with others. Sensors can share information by non-destructively accessing the other sensor models. This means more data in *greater speeds*. Vital information can be collected, manipulated and double checked by many sensors, producing a more accurate world model.

However, updating concurrently different models involves the execution of many complex and time consuming algorithms. Additionally, too much storage is required, and up to a certain degree there is duplication of information, leading to undesired redundancy. Furthermore, if a priori knowledge is to be used, all different models have to be updated.

2.19 Comparison of Different World Models

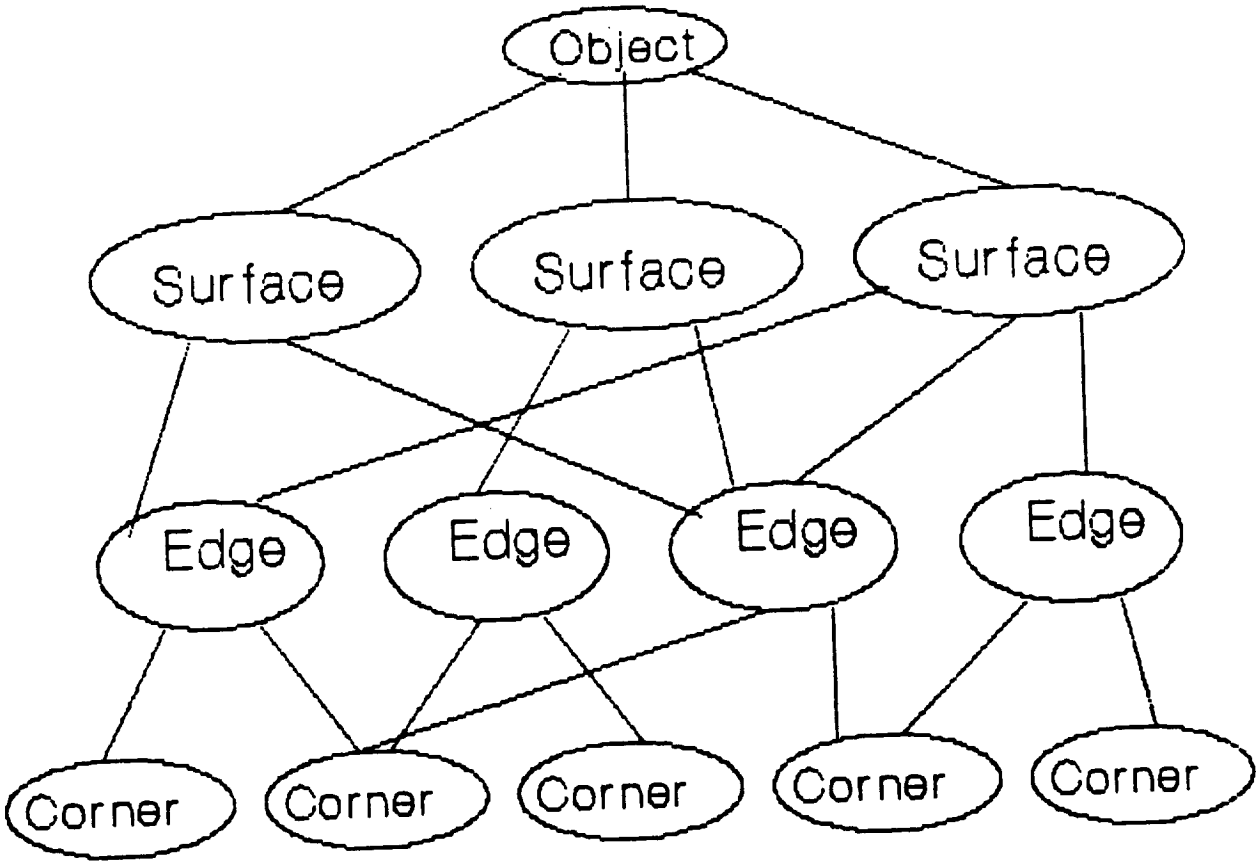
Most of the different world model representations are designed having a specific application in mind. Hence, the same criteria that are of major importance for one situation, might be of lesser significance for another, and vice versa. This variation in the employed criteria justifies the number and diversity among various world models. Having always in mind that the choice and development of world models is application specific, the different world models can be compared according to: Accuracy of representation, Storage requirements, Speed in updating the scheme, Simplicity of the basic concepts of the model, ease of Implementation, Expandability, need of a Priori Knowledge, and Application for which this model is best suited. Using these criteria, the following table compares the performance of the analytically described world models.

World Model Evaluation Table

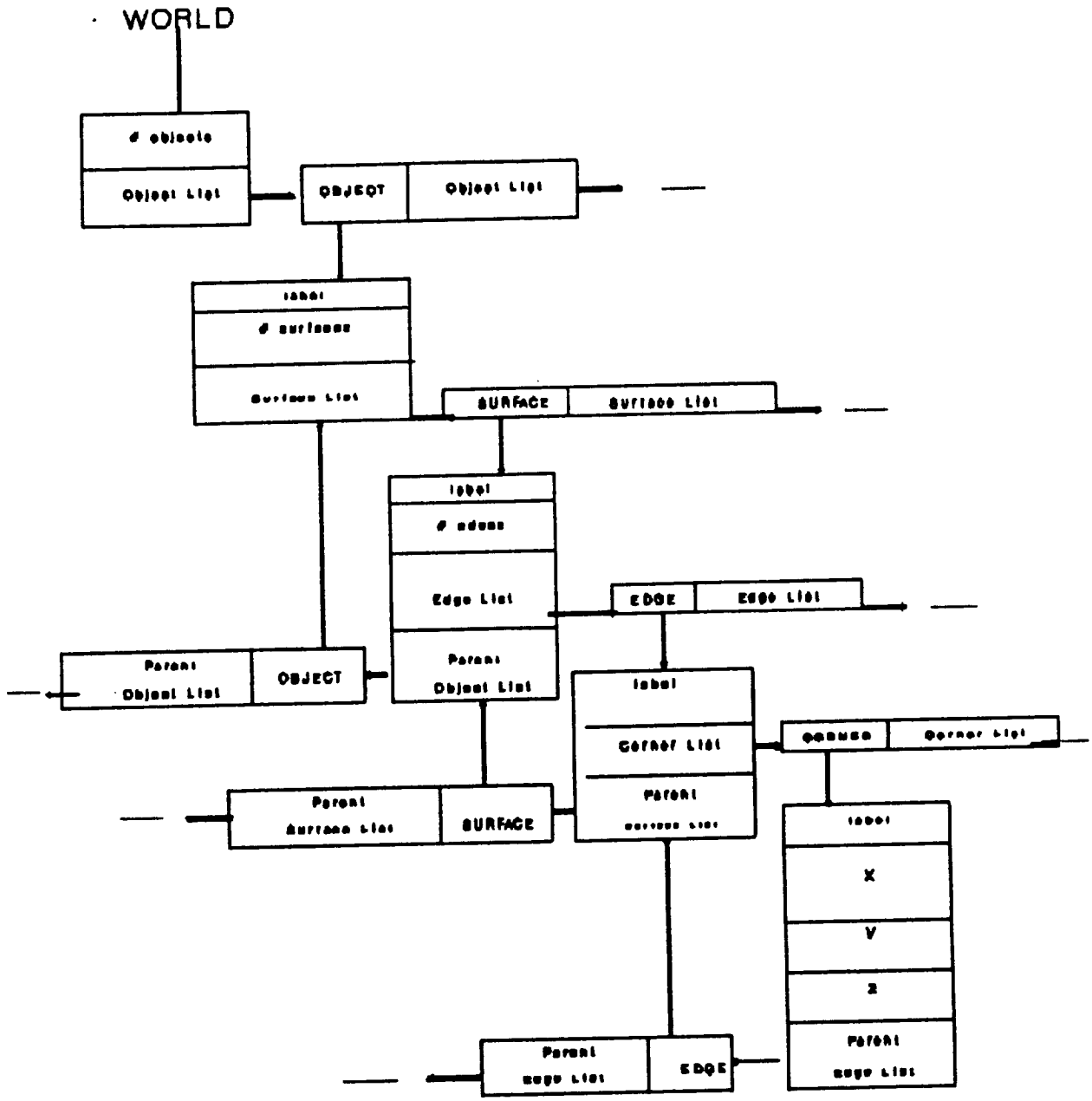
World Models	Acc	Sto	Sp	Sim	Imp	Exp	PriorK	Application
Combinatorial Geometry	*	***	***	***	**	***	No	Navigation
Polygonal Planar Hulls	**	**	*	**	**	**	No	Navigation
Geometric	***	**	**	**	**	**	No/Yes	Navigation
Hierarchical Geometric	***	**	**	**	***	***	Yes	Navigation
Attributed Graph	*	*	*	**	**	***	Yes	Recognition
Feature Space Graph	***	**	***	**	**	**	No	Recognition
Visibility Graph	**	**	*	***	*	*	Yes	Navigation
FFC Graph	**	**	**	**	**	***	No	Manufacture
Topological	***	***	*	**	**	***	No	Navigation
Space Time Octree	**	**	**	*	**	***	No/Yes	Movement
Occupancy Grid	***	***	**	***	***	*	No/Yes	Navigation
Volumetric	***	**	**	**	**	*	No/Yes	Navigation
Visibel Grid	**	***	**	*	*	**	No/Yes	Navigation
Three-Map	***	*	**	**	***	**	No/Yes	Navigation
Generic	***	**	***	*	***	***	Yes	Recognition
Multiple	***	*	*	**	***	***	No/Yes	Anything

2.20 Y-Frame World Model Design

Y-Frame Feature Model



Y-Frame Data Structure



2.21 Implementation/Ada Programs

objmain.a

Y-Frame World Model Representation

```
  This is the main procedure that calls the procedures that create
  and print the contents of the object database.

th OBJECT_DATABASE; use OBJECT_DATABASE;
th OBJECT_LISTING; use OBJECT_LISTING;

ɔcedure OBJECT_HANDLING is

      WORKSPACE: WORLD_MODEL_TYPE;

gin
 OBJECT_INFO (WORKSPACE);
 SURFACE_INFO (WORKSPACE);
 EDGE_INFO (WORKSPACE);
 CORNER_INFO (WORKSPACE);
 PRINT_DATABASE (WORKSPACE);
d;
```

Y - Frame World Model Representation

This package enters all the user information to the object database.
This version of the package uses singly linked data strucures.

```
ckage OBJECT_DATABASE is

   type OBJECT_LIST_TYPE;
   type OBJECT_MODEL_TYPE;
   type SURFACE_LIST_TYPE;
   type SURFACE_MODEL_TYPE;
   type EDGE_LIST_TYPE;
   type EDGE_MODEL_TYPE;
   type CORNER_LIST_TYPE;
   type CORNER_MODEL_TYPE;

   type OBJECT_LIST_POINTER is access OBJECT_LIST_TYPE;
   type OBJECT_MODEL_POINTER is access OBJECT_MODEL_TYPE;
   type SURFACE_LIST_POINTER is access SURFACE_LIST_TYPE;
   type SURFACE_MODEL_POINTER is access SURFACE_MODEL_TYPE;
   type EDGE_LIST_POINTER is access EDGE_LIST_TYPE;
   type EDGE_MODEL_POINTER is access EDGE_MODEL_TYPE;
   type CORNER_LIST_POINTER is access CORNER_LIST_TYPE;
   type CORNER_MODEL_POINTER is access CORNER_MODEL_TYPE;

-  The following data structure is used for storing the workspace information

-  The root of the structure is a node of WORLD_MODEL_TYPE.  This contains
-  the number of objects that are present in the world and a pointer to a
-  linked list of objects.

-  The linked list of objects is composed of nodes with two elements.  One
-  element is a pointer to a node containing all the object-related data.
-  The other element is a pointer to the next object, or in other words to
-  the next item in the linked list.

-  The node containing all the object-related data is of OBJECT_MODEL_TYPE.
-  It contains three elements.  The first one is the label by which the objec
-  is referenced to.  The second one is a field holding the number of surface
-  that compose this object.  Finally, the third element is a pointer to a
-  linked list of the surfaces that compose this object.
```

The linked list of surfaces is composed of nodes with two elements. One element is a pointer to a node containing all the surface-related data. The other element is a pointer to the next surface, or in other words to the next item in the linked list.

The node containing all the surface-related data is of SURFACE_MODEL_TYPE. It contains four elements. The first one is the label by which the surfac is referenced to in the workspace. The second one is a field holding the number of edges that define this surface. The next field is a pointer to a linked list of the edges that define this surface. Finally, the last element is a pointer to a linked list of the objects to which this surface belongs.

The linked list of edges is composed of nodes with two elements. One of them is a pointer to a node containing all the edge-related data. The other element is a pointer to the next edge, or in other words to the next item in the linked list.

The node containing all the edge-related data is of EDGE_MODEL_TYPE. It contains three elements. The first one is the label by which the surface is referenced in the workspace. The second element is a pointer to a linked list of the two corners that define the edge. Finally, the last element is a pointer to a linked list of the surfaces to which this edge belongs.

The linked list of corners is composed of nodes with two elements. One of them is a pointer to a node containing all the corner-related data. The other element is a pointer to the next corner, or in other words to the next item in the linked list.

Finally, the node containing all the corner-related data is of CORNER_MODE _TYPE. It is the leaf node in the data structure, and contains five elements. The first one is the label by which the corner is referenced in the workspace. The second, third, and fourth elements are correspondingly the x, y, z coordinates of the corner. The fifth element is a pointer to a linked list of the edges to which this corner belongs.

Data strucure of the root world node.

```
type WORLD_MODEL_TYPE is record
    NUM_OBJECTS              : integer;
    OBJECT_LIST              : OBJECT_LIST_POINTER;
end record;
```

Data structure of the nodes in the linked list of objects.

```
type OBJECT_LIST_TYPE is record
    OBJECT_MODEL             : OBJECT_MODEL_POINTER;
    OBJECT_LIST_NEXT         : OBJECT_LIST_POINTER;
end record;
```

Data strucure of the object node.

```
type OBJECT_MODEL_TYPE is record
    LABEL                       : integer;
    NUM_SURFACES                : integer;
    SURFACE_LIST                : SURFACE_LIST_POINTER;
end record;

- Data structure of the nodes in the linked list of surfaces.
type SURFACE_LIST_TYPE is record
    SURFACE_MODEL               : SURFACE_MODEL_POINTER;
    SURFACE_LIST_NEXT           : SURFACE_LIST_POINTER;
end record;

- Data structure of the surface node.
type SURFACE_MODEL_TYPE is record
    LABEL                       : integer;
    NUM_EDGES                   : integer;
    EDGE_LIST                   : EDGE_LIST_POINTER;
    PARENT_OBJECTS              : OBJECT_LIST_POINTER;
end record;

- Data strucure of the nodes in the linked list of edges.
type EDGE_LIST_TYPE is record
    EDGE_MODEL                  : EDGE_MODEL_POINTER;
    EDGE_LIST_NEXT              : EDGE_LIST_POINTER;
end record;

-- Data structure of the edge node.
type EDGE_MODEL_TYPE is record
    LABEL                       : integer;
    CORNER_LIST                 : CORNER_LIST_POINTER;
    PARENT_SURFACES             : SURFACE_LIST_POINTER;
end record;

-- Data structure of the nodes in the linked list of corners.
type CORNER_LIST_TYPE is record
    CORNER_MODEL                : CORNER_MODEL_POINTER;
    CORNER_LIST_NEXT            : CORNER_LIST_POINTER;
end record;

-- Data strucure of the leaf corner node.
type CORNER_MODEL_TYPE is record
    LABEL                       : integer;
    X, Y, Z                     : float;
    PARENT_EDGES                : EDGE_LIST_POINTER;
end record;


    procedure OBJECT_INFO (WORLD: in out WORLD_MODEL_TYPE);
    procedure SURFACE_INFO (WORLD: in out WORLD_MODEL_TYPE);
```

```
      procedure EDGE_INFO (WORLD: in out WORLD_MODEL_TYPE);
      procedure CORNER_INFO (WORLD: in out WORLD_MODEL_TYPE);

i OBJECT_DATABASE;



th TEXT_IO; use TEXT_IO;
th INTEGER_IO; use INTEGER_IO;
th FLOAT_IO; use FLOAT_IO;


ckage body OBJECT_DATABASE is

      CORNER                                  : CORNER_MODEL_POINTER;
      EDGE                                    : EDGE_MODEL_POINTER;
      SURFACE                                 : SURFACE_MODEL_POINTER;
      OBJECT                                  : OBJECT_MODEL_POINTER;

      TMP_CORNER_LIST_POINTER1,
      TMP_CORNER_LIST_POINTER2                : CORNER_LIST_POINTER;
      TMP_EDGE_LIST_POINTER1,
      TMP_EDGE_LIST_POINTER2                      : EDGE_LIST_POINTER;
      TMP_SURFACE_LIST_POINTER1,
      TMP_SURFACE_LIST_POINTER2                : SURFACE_LIST_POINTER;
      TMP_OBJECT_LIST_POINTER                  : OBJECT_LIST_POINTER;

      DATA_OUT                                : file_type;

rocedure OBJECT_INFO (WORLD: in out WORLD_MODEL_TYPE) is

      TEMP_LABEL : integer;
      CURRENT_POINTER, PREVIOUS_POINTER : SURFACE_LIST_POINTER;

egin
      create (DATA_OUT, out_file, "obj_input_to_output.dat");

      -- Updating the root world node accessed by the variable WORLD
      put ("How many objects are in the workspace? ");
      get (WORLD.NUM_OBJECTS);
      put (DATA_OUT, WORLD.NUM_OBJECTS);
      new_line (DATA_OUT);
      WORLD.OBJECT_LIST := new OBJECT_LIST_TYPE;

      -- Updating the first (and only) node in the linked list of objects
      TMP_OBJECT_LIST_POINTER := WORLD.OBJECT_LIST;
      TMP_OBJECT_LIST_POINTER.OBJECT_MODEL := new OBJECT_MODEL_TYPE;
      TMP_OBJECT_LIST_POINTER.OBJECT_LIST_NEXT := NULL;
```

```
    -- Updating the object node for the first (and only) object
    OBJECT := TMP_OBJECT_LIST_POINTER.OBJECT_MODEL;
    put ("How is the object labeled? ");
    get (OBJECT.LABEL);
    put (DATA_OUT, OBJECT.LABEL);
    new_line (DATA_OUT);
    put ("How many surfaces does this object have? ");
    get (OBJECT.NUM_SURFACES);
    put (DATA_OUT, OBJECT.NUM_SURFACES);
    new_line (DATA_OUT);
    OBJECT.SURFACE_LIST := null;

    -- Creating the linked list of surfaces for the referenced object
    PREVIOUS_POINTER := null;
    TMP_SURFACE_LIST_POINTER2 := OBJECT.SURFACE_LIST;
    for i in 1..OBJECT.NUM_SURFACES loop
       put ("How is this surface labeled? ");
       get (TEMP_LABEL);
       put (DATA_OUT, TEMP_LABEL);
       new_line (DATA_OUT);
       -- this is the first surface in the linked list of surfaces for
       -- the referenced object
       if  OBJECT.SURFACE_LIST = null  then
          OBJECT.SURFACE_LIST := new SURFACE_LIST_TYPE;
          CURRENT_POINTER := OBJECT.SURFACE_LIST;
       --  this object has at least another surface already stored in its
       --  linked list of surfaces
       else
          CURRENT_POINTER.SURFACE_LIST_NEXT := new SURFACE_LIST_TYPE;
          CURRENT_POINTER := CURRENT_POINTER.SURFACE_LIST_NEXT;
       end if;
       CURRENT_POINTER.SURFACE_MODEL := new SURFACE_MODEL_TYPE;
       SURFACE := CURRENT_POINTER.SURFACE_MODEL;
       SURFACE.LABEL := TEMP_LABEL;
       SURFACE.EDGE_LIST := null;
       SURFACE.PARENT_OBJECTS := new OBJECT_LIST_TYPE;
       SURFACE.PARENT_OBJECTS.OBJECT_MODEL := OBJECT;
       SURFACE.PARENT_OBJECTS.OBJECT_LIST_NEXT := null;
       CURRENT_POINTER.SURFACE_LIST_NEXT := null;
       if  PREVIOUS_POINTER /= null  then
          PREVIOUS_POINTER.SURFACE_LIST_NEXT := CURRENT_POINTER;
       end if;
       PREVIOUS_POINTER := CURRENT_POINTER;
    end loop;

end OBJECT_INFO;
```

```
ocedure SURFACE_INFO (WORLD: in out WORLD_MODEL_TYPE) is

      TEMP_LABEL : integer;
      FOUND : boolean;
      TMP_SURFACE : SURFACE_MODEL_POINTER;
      CURRENT_POINTER, PREVIOUS_POINTER : EDGE_LIST_POINTER;

egin

      put ("Start entering surface information");
      new_line;

      -- Updating the surface nodes of the referenced surfaces
      for i in 1..OBJECT.NUM_SURFACES loop
        put ("Which surface are you referring at? ");
        get (TEMP_LABEL);
        put (DATA_OUT, TEMP_LABEL);
        new_line (DATA_OUT);

        -- Checking if the user typed the correct surface label, in other
        -- words checking if a surface node for the referenced surface
        -- already exists
        TMP_SURFACE_LIST_POINTER2 := OBJECT.SURFACE_LIST;
        SURFACE := TMP_SURFACE_LIST_POINTER2.SURFACE_MODEL;
        while  (SURFACE.LABEL /= TEMP_LABEL) and
               (TMP_SURFACE_LIST_POINTER2 /= null)  loop
            TMP_SURFACE_LIST_POINTER2 :=
                    TMP_SURFACE_LIST_POINTER2.SURFACE_LIST_NEXT;
            if TMP_SURFACE_LIST_POINTER2 /= null then
                 SURFACE := TMP_SURFACE_LIST_POINTER2.SURFACE_MODEL;
            end if;
        end loop;
        -- SURFACE is now pointing to the referenced surface
        if  TMP_SURFACE_LIST_POINTER2 = null  then
           put (" ERROR IN LABELING ");
        -- Once the surface is located and accessed through variable
        -- SURFACE, its updating proceeds.
        else
           put ("How many edges does this surface have? ");
           get (SURFACE.NUM_EDGES);
           put (DATA_OUT, SURFACE.NUM_EDGES);
           new_line (DATA_OUT);
           SURFACE.EDGE_LIST := null;
           PREVIOUS_POINTER := null;

           -- Creating the linked list of edges for the referenced surface
           TMP_EDGE_LIST_POINTER2 := SURFACE.EDGE_LIST;
           for k in 1..SURFACE.NUM_EDGES loop
             put ("How are these edges labeled? ");
             get (TEMP_LABEL);
```

```
put (DATA_OUT, TEMP_LABEL);
new_line (DATA_OUT);
-- Checking if this edge is already referenced, and thus an
-- edge node already exists for that edge
FOUND := false;
TMP_SURFACE_LIST_POINTER1 :=
          WORLD.OBJECT_LIST.OBJECT_MODEL.SURFACE_LIST;
while (TMP_SURFACE_LIST_POINTER1/=null) and (not FOUND) loop
   TMP_SURFACE := TMP_SURFACE_LIST_POINTER1.SURFACE_MODEL;
   TMP_EDGE_LIST_POINTER1 := TMP_SURFACE.EDGE_LIST;
   while (TMP_EDGE_LIST_POINTER1 /= null) and (not FOUND) loop
      EDGE := TMP_EDGE_LIST_POINTER1.EDGE_MODEL;
      if  EDGE.LABEL = TEMP_LABEL  then
         FOUND := true;
         -- EDGE points to the already existing edge node
      else
         TMP_EDGE_LIST_POINTER1:=
               TMP_EDGE_LIST_POINTER1.EDGE_LIST_NEXT;
      end if;
   end loop;
   TMP_SURFACE_LIST_POINTER1 :=
         TMP_SURFACE_LIST_POINTER1.SURFACE_LIST_NEXT;
end loop;
-- this is the first edge in the linked list of edges for
-- the referenced surface
if  SURFACE.EDGE_LIST = null then
   SURFACE.EDGE_LIST := new EDGE_LIST_TYPE;
   CURRENT_POINTER := SURFACE.EDGE_LIST;
-- this surface has at least another edge already stored in its
-- linked list of edges
else
   CURRENT_POINTER.EDGE_LIST_NEXT := new EDGE_LIST_TYPE;
   CURRENT_POINTER := CURRENT_POINTER.EDGE_LIST_NEXT;
end if;
-- If the edge is one for which no edge node already exists
-- a new edge node is created and labeled
if TMP_EDGE_LIST_POINTER1 = null then
    CURRENT_POINTER.EDGE_MODEL := new EDGE_MODEL_TYPE;
    EDGE := CURRENT_POINTER.EDGE_MODEL;
    EDGE.CORNER_LIST := null;
    EDGE.PARENT_SURFACES := new SURFACE_LIST_TYPE;
    EDGE.PARENT_SURFACES.SURFACE_MODEL := SURFACE;
    EDGE.PARENT_SURFACES.SURFACE_LIST_NEXT := null;
    EDGE.LABEL := TEMP_LABEL;
-- Otherwise, if an edge node is already created, the edge
-- pointer in the current node of the linked list of edges
-- for the referenced surface is redirected to the specific
-- edge node, and the parent list of that edge node is updated
else
    CURRENT_POINTER.EDGE_MODEL := EDGE;
```

```
                TMP_SURFACE_LIST_POINTER1 := EDGE.PARENT_SURFACES;
                while TMP_SURFACE_LIST_POINTER1.SURFACE_LIST_NEXT
                                /= null  loop
                    TMP_SURFACE_LIST_POINTER1 :=
                        TMP_SURFACE_LIST_POINTER1.SURFACE_LIST_NEXT;
                end loop;
                TMP_SURFACE_LIST_POINTER1.SURFACE_LIST_NEXT :=
                    new SURFACE_LIST_TYPE;
                TMP_SURFACE_LIST_POINTER1 :=
                    TMP_SURFACE_LIST_POINTER1.SURFACE_LIST_NEXT;
                TMP_SURFACE_LIST_POINTER1.SURFACE_MODEL := SURFACE;
                TMP_SURFACE_LIST_POINTER1.SURFACE_LIST_NEXT := null;
            end if;
            CURRENT_POINTER.EDGE_LIST_NEXT := null;
            if  PREVIOUS_POINTER /= null  then
               PREVIOUS_POINTER.EDGE_LIST_NEXT := CURRENT_POINTER;
            end if;
            PREVIOUS_POINTER := CURRENT_POINTER;
          end loop;
        end if;
      end loop;

nd SURFACE_INFO;


rocedure EDGE_INFO (WORLD: in out WORLD_MODEL_TYPE) is

      TEMP_LABEL : integer;
      FOUND      :  boolean;
      MORE_EDGES :  character;
      TMP_SURFACE : SURFACE_MODEL_POINTER;
      TMP_EDGE : EDGE_MODEL_POINTER;
      CURRENT_POINTER, PREVIOUS_POINTER : CORNER_LIST_POINTER;

egin
      put ("Start entering the edge information ");
      new_line;
      MORE_EDGES := 'y';

      -- Updating the edge nodes of the referenced edges
      while (MORE_EDGES = 'y') or (MORE_EDGES = 'Y')  loop
         put ("Which edge are you referring at? ");
         get (TEMP_LABEL);
         put (DATA_OUT, TEMP_LABEL);
         new_line (DATA_OUT);

         -- Checking if the user typed the correct edge label, in other
         -- words checking if an edge node for the referenced edge already
         -- exists
```

```
FOUND := false;
TMP_SURFACE_LIST_POINTER2 :=
        WORLD.OBJECT_LIST.OBJECT_MODEL.SURFACE_LIST;
while (TMP_SURFACE_LIST_POINTER2 /= null) and (not FOUND) loop
    SURFACE := TMP_SURFACE_LIST_POINTER2.SURFACE_MODEL;
    TMP_EDGE_LIST_POINTER2 := SURFACE.EDGE_LIST;
    while  (TMP_EDGE_LIST_POINTER2 /= null) and (not FOUND) loop
        EDGE := TMP_EDGE_LIST_POINTER2.EDGE_MODEL;
        if  EDGE.LABEL = TEMP_LABEL then
           FOUND := true;
           -- EDGE is pointing to the referenced edge node
        else
           TMP_EDGE_LIST_POINTER2 :=
                      TMP_EDGE_LIST_POINTER2.EDGE_LIST_NEXT;
        end if;
    end loop;
    if  (not FOUND) then
       TMP_SURFACE_LIST_POINTER2 :=
             TMP_SURFACE_LIST_POINTER2.SURFACE_LIST_NEXT;
    end if;
 end loop;

if  TMP_EDGE_LIST_POINTER2 = null  then
   put (" ERROR IN LABELING ");
-- the referenced edge node is located, accessed through the variabl
-- EDGE, and its updating proceeds
else
   PREVIOUS_POINTER := null;
   -- Creating the linked list of corners for the referenced edge
   TMP_CORNER_LIST_POINTER2 := EDGE.CORNER_LIST;
   for i in 1..2 loop
      put ("How are the corners defining the edge labeled? ");
      get (TEMP_LABEL);
      put (DATA_OUT, TEMP_LABEL);
      new_line (DATA_OUT);
      -- Checking if the corner is already referenced and thus
      -- a corner node already exists for that corner
      FOUND := false;
      TMP_SURFACE_LIST_POINTER1 :=
            WORLD.OBJECT_LIST.OBJECT_MODEL.SURFACE_LIST;
      TMP_EDGE_LIST_POINTER1 :=
            TMP_SURFACE_LIST_POINTER1.SURFACE_MODEL.EDGE_LIST;
      while (TMP_SURFACE_LIST_POINTER1/=null) and (not FOUND) loop
        while (TMP_EDGE_LIST_POINTER1 /= null) and (not FOUND) loop
           TMP_EDGE := TMP_EDGE_LIST_POINTER1.EDGE_MODEL;
           TMP_CORNER_LIST_POINTER1 := TMP_EDGE.CORNER_LIST;
           while (TMP_CORNER_LIST_POINTER1 /= null)
               and (not FOUND) loop
              CORNER := TMP_CORNER_LIST_POINTER1.CORNER_MODEL;
              if  CORNER.LABEL = TEMP_LABEL  then
```

```
            FOUND := true;
            -- CORNER points to the already existing corner
            -- node of the referenced corner
         else
            TMP_CORNER_LIST_POINTER1:=
               TMP_CORNER_LIST_POINTER1.CORNER_LIST_NEXT;
         end if;
      end loop;
      if (not FOUND) then
          TMP_EDGE_LIST_POINTER1:=
              TMP_EDGE_LIST_POINTER1.EDGE_LIST_NEXT;
      end if;
   end loop;
   if (not FOUND) then
      TMP_SURFACE_LIST_POINTER1 :=
          TMP_SURFACE_LIST_POINTER1.SURFACE_LIST_NEXT;
      if TMP_SURFACE_LIST_POINTER1 /= null then
          TMP_SURFACE :=
             TMP_SURFACE_LIST_POINTER1.SURFACE_MODEL;
          TMP_EDGE_LIST_POINTER1 := TMP_SURFACE.EDGE_LIST;
      end if;
   end if;
end loop;
-- this is the first corner in the linked list of corners for
-- the referenced edge
if  EDGE.CORNER_LIST = null  then
   EDGE.CORNER_LIST := new CORNER_LIST_TYPE;
   CURRENT_POINTER := EDGE.CORNER_LIST;
-- this edge has its first corner already stored in its linked
-- list of corners
else
   CURRENT_POINTER.CORNER_LIST_NEXT := new CORNER_LIST_TYPE;
   CURRENT_POINTER := CURRENT_POINTER.CORNER_LIST_NEXT;
end if;
-- If the corner is one for which no corner node already exists
-- a new corner node is created and labeled
if  TMP_EDGE_LIST_POINTER1 = null  then
    CURRENT_POINTER.CORNER_MODEL := new CORNER_MODEL_TYPE;
    CORNER := CURRENT_POINTER.CORNER_MODEL;
    CORNER.PARENT_EDGES := new EDGE_LIST_TYPE;
    CORNER.PARENT_EDGES.EDGE_MODEL := EDGE;
    CORNER.PARENT_EDGES.EDGE_LIST_NEXT := null;
    CORNER.LABEL := TEMP_LABEL;
-- Otherwise, if a corner node is already created, the corner
-- pointer in the current node in the linked list of corners
-- for the referenced edge is redirected to the specific corner
-- node, and the parent list of that corner node is updated
else
    CURRENT_POINTER.CORNER_MODEL := CORNER;
    TMP_EDGE_LIST_POINTER1 := CORNER.PARENT_EDGES;
```

```
                while TMP_EDGE_LIST_POINTER1.EDGE_LIST_NEXT
                                /= null  loop
                    TMP_EDGE_LIST_POINTER1 :=
                        TMP_EDGE_LIST_POINTER1.EDGE_LIST_NEXT;
                end loop;
                TMP_EDGE_LIST_POINTER1.EDGE_LIST_NEXT :=
                     new EDGE_LIST_TYPE;
                TMP_EDGE_LIST_POINTER1 :=
                    TMP_EDGE_LIST_POINTER1.EDGE_LIST_NEXT;
                TMP_EDGE_LIST_POINTER1.EDGE_MODEL := EDGE;
                TMP_EDGE_LIST_POINTER1.EDGE_LIST_NEXT := null;
            end if;
            CURRENT_POINTER.CORNER_LIST_NEXT := null;
            if  PREVIOUS_POINTER /= null then
               PREVIOUS_POINTER.CORNER_LIST_NEXT := CURRENT_POINTER;
            end if;
            PREVIOUS_POINTER := CURRENT_POINTER;
          end loop;
        end if;
        put ("Are there more edges to be processed? (y/n) ");
        get (MORE_EDGES);
        put (DATA_OUT, MORE_EDGES);
        new_line (DATA_OUT);
      end loop;
nd EDGE_INFO;


rocedure CORNER_INFO (WORLD: in out WORLD_MODEL_TYPE) is

      TEMP_LABEL : integer;
      FOUND      :  boolean;
      MORE_CORNERS :  character;
      TMP_SURFACE : SURFACE_MODEL_POINTER;
      TMP_EDGE : EDGE_MODEL_POINTER;
      TMP_CORNER :CORNER_MODEL_POINTER;

egin
      put ("Start entering the corner information ");
      new_line;
      MORE_CORNERS := 'y';

      -- Updating the corner nodes of the referenced corners
      while (MORE_CORNERS = 'y') or (MORE_CORNERS = 'Y')  loop
         put ("Which corner are you referring at? ");
         get (TEMP_LABEL);
         put (DATA_OUT, TEMP_LABEL);
         new_line (DATA_OUT);

         -- Checking if the user typed the correct corner label, in other
```

```
-- words checking if a corner node for the referenced corner
-- already exists
FOUND := false;
TMP_SURFACE_LIST_POINTER2 :=
        WORLD.OBJECT_LIST.OBJECT_MODEL.SURFACE_LIST;
 while (TMP_SURFACE_LIST_POINTER2 /= null) and (not FOUND) loop
    SURFACE := TMP_SURFACE_LIST_POINTER2.SURFACE_MODEL;
    TMP_EDGE_LIST_POINTER2 := SURFACE.EDGE_LIST;
    while  (TMP_EDGE_LIST_POINTER2 /= null) and (not FOUND) loop
        EDGE := TMP_EDGE_LIST_POINTER2.EDGE_MODEL;
        TMP_CORNER_LIST_POINTER2 := EDGE.CORNER_LIST;
        while (TMP_CORNER_LIST_POINTER2 /= null)
              and (not FOUND) loop
            CORNER := TMP_CORNER_LIST_POINTER2.CORNER_MODEL;
            if  CORNER.LABEL = TEMP_LABEL then
               FOUND := true;
               -- CORNER is now pointing to the corner node of
               -- referenced corner
            else
               TMP_CORNER_LIST_POINTER2 :=
                     TMP_CORNER_LIST_POINTER2.CORNER_LIST_NEXT;
            end if;
        end loop;
        if  (not FOUND) then
           TMP_EDGE_LIST_POINTER2 :=
                TMP_EDGE_LIST_POINTER2.EDGE_LIST_NEXT;
        end if;
    end loop;
    if  (not FOUND) then
       TMP_SURFACE_LIST_POINTER2 :=
             TMP_SURFACE_LIST_POINTER2.SURFACE_LIST_NEXT;
    end if;
 end loop;

if  TMP_CORNER_LIST_POINTER2 = null  then
   put (" ERROR IN LABELING ");
-- The corner node for the referenced corner [illegible] accessed
-- through the variable CORNER, and its updating proceeds
else
   put ("What is the X coordinate of this corner? ");
   get (CORNER.X);
   put (DATA_OUT, CORNER.X);
   put ("What is the Y coordinate of this corner? ");
   get (CORNER.Y);
   put (DATA_OUT, CORNER.Y);
   new_line (DATA_OUT);
   put ("What is the Z coordinate of this corner? ");
   get (CORNER.Z);
   put (DATA_OUT, CORNER.Z);
   new_line (DATA_OUT);
```

```
        end if;
        put ("Are there more corners to be processed? (y/n) ");
        get (MORE_CORNERS);
        put (DATA_OUT, MORE_CORNERS);
        new_line (DATA_OUT);
      end loop;
      close (DATA_OUT);

d CORNER_INFO;

id OBJECT_DATABASE;
```

```
-                    Y-Frame World Model Representation
-
-
-
- This package contains the procedures that perform a Depth First Traversal
- in the object database.

ith OBJECT_DATABASE; use OBJECT_DATABASE;
ith TEXT_IO; use TEXT_IO;
ith INTEGER_IO; use INTEGER_IO;
ith FLOAT_IO; use FLOAT_IO;

ackage OBJECT_LISTING is
    procedure PRINT_DATABASE (WORLD: in out WORLD_MODEL_TYPE);
nd OBJECT_LISTING;

ackage body OBJECT_LISTING is

        CORNER                          : CORNER_MODEL_POINTER;
        EDGE                            : EDGE_MODEL_POINTER;
        SURFACE                         : SURFACE_MODEL_POINTER;
        OBJECT                          : OBJECT_MODEL_POINTER;

        TMP_CORNER_LIST_POINTER1,
        TMP_CORNER_LIST_POINTER2        : CORNER_LIST_POINTER;
        TMP_EDGE_LIST_POINTER1,
        TMP_EDGE_LIST_POINTER2                  : EDGE_LIST_POINTER;
        TMP_SURFACE_LIST_POINTER1,
        TMP_SURFACE_LIST_POINTER2       : SURFACE_LIST_POINTER;
        TMP_OBJECT_LIST_POINTER         : OBJECT_LIST_POINTER;


rocedure PRINT_DATABASE (WORLD: in out WORLD_MODEL_TYPE) is

egin
   new_line;
   put (" The world model is composed of ");
   put (WORLD.NUM_OBJECTS);
   put (" objects.");
   new_line;
   TMP_OBJECT_LIST_POINTER := WORLD.OBJECT_LIST;
```

```
-- Loop going through every object in the world
while TMP_OBJECT_LIST_POINTER /= null loop
     OBJECT := TMP_OBJECT_LIST_POINTER.OBJECT_MODEL;
     put ("Object Nr ");
     put (OBJECT.LABEL);
     put (" has the following characteristics..");
     new_line;

     put (" The object is composed of ");
     put (OBJECT.NUM_SURFACES);
     put (" surfaces.");
     new_line;
     TMP_SURFACE_LIST_POINTER2 := OBJECT.SURFACE_LIST;
     -- Loop going through every surface in the current object
     while TMP_SURFACE_LIST_POINTER2 /= null loop
          new_line;
          SURFACE := TMP_SURFACE_LIST_POINTER2.SURFACE_MODEL;
          put ("     Surface Nr ");
          put (SURFACE.LABEL);
          put (" has the following characteristics..");
          new_line;

          put ("     This surface is composed of ");
          put (SURFACE.NUM_EDGES);
          put (" edges.");
          new_line;
          TMP_EDGE_LIST_POINTER2 := SURFACE.EDGE_LIST;
          -- Loop going through every edge in the current surface
          while TMP_EDGE_LIST_POINTER2 /= null loop
               new_line;
               EDGE := TMP_EDGE_LIST_POINTER2.EDGE_MODEL;
               put ("          Edge Nr ");
               put (EDGE.LABEL);
               -- put (" has the following characteristics..");
               new_line;
               TMP_CORNER_LIST_POINTER2 := EDGE.CORNER_LIST;
               -- Loop going through every corner in the current edge
               while TMP_CORNER_LIST_POINTER2 /= null  loop
                    new_line;
                    CORNER := TMP_CORNER_LIST_POINTER2.CORNER_MODEL;
                    put ("               Corner Nr ");
                    put (CORNER.LABEL);
                    new_line;
                    put ("                    X coordinate = ");
                    put (CORNER.X);
                    new_line;
                    put ("                    Y coordinate = ");
                    put (CORNER.Y);
                    new_line;
                    put ("                    Z coordinate = ");
```

```
                    put (CORNER.Z);
                    new_line;
                    TMP_CORNER_LIST_POINTER2 :=
                        TMP_CORNER_LIST_POINTER2.CORNER_LIST_NEXT;
                end loop;
                TMP_EDGE_LIST_POINTER2 :=
                        TMP_EDGE_LIST_POINTER2.EDGE_LIST_NEXT;
            end loop;
            TMP_SURFACE_LIST_POINTER2 :=
                        TMP_SURFACE_LIST_POINTER2.SURFACE_LIST_NEXT;
        end loop;
        TMP_OBJECT_LIST_POINTER := TMP_OBJECT_LIST_POINTER.OBJECT_LIST_NEXT;
    end loop;

nd PRINT_DATABASE;

nd OBJECT_LISTING;
```

2.22 References

[1] Aggarwal, J. K., Wang, Y. F.. "Analysis of a Sequence of Images Using Point and Line Correspondences." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1987. pp. 1275-1280.

[2] Albus, J. S.. "Data Storage in the Cerebellar Model Articulation Controller (CMAC)." Journal of Dynamic Systems, Measurement, and Control. September 1975. pp. 228-233.

[3] Allen, Peter K.. "Mapping Haptic Exploratory Procedures of Multiple Shape Representations." Robotics and Automation. 1990. pp. 1679-1682.

[4] Allen, Peter K.. Robotic Object Recognition Using Vision and Touch. Massachusetts: Kluwer Academic Publishers. 1987.

[5] Asada, Minonu. "Building a 3-D World Model for a Mobile Robot from Sensory Data." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1988. pp. 918-923.

[6] Boissonat, J. D., Faugeras, O. D., Le Bras-Mehlman, E.. "Representing Stereo Data with the Delaunay Triangulation." Proceedings IEEE International Conference on Robotics and Automation. vol. 3. 1988. pp. 1798-1803.

[7] Chaconas, Karen, Nashman, Marilyn. "Visual Perception in a Hierarchical Central System: Level I." NIST Technical Note 1260. June 1989.

[8] Chen, Yao-Chon, Vidyasagar, Mathukumalli. "Optimal Trajectory Planning for Planar n-Link Revolute Manipulators in the Presence of Obstacles." Proceedings IEEE International Conference on Robotics and Automation. vol. 1. 1988. pp. 202-208.

[9] Crowley, James L.. "Using the Composite Surface Model for Perceptual Tasks." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1987. pp. 929-934.

[10] Crowley, James L.. "World Modeling and Position Estimation for a Mobile Robot Using Ultrasonic Ranging." Publication of the National Polytechnic

Institute of Grenoble. France. February 1989.

[11] De Floriani, Leila, Nagy, George. "An Alternative Goal-Oriented Hierarchical Representation of Solid Objects for Computer-Integrated Manufacturing." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1988. pp. 1101-1106.

[12] de Saint Vincent, Arnaud R.. "Visual Navigation for a Mobile Robot: Building a Map of the Occupied Space from Sparse 3-D Stereo Data." Proceedings IEEE International Conference on Robotics and Automation. vol. 3. 1987. pp. 1429-1435.

[13] Elfes, Alberto. "Using Occupancy Grids for Mobile Robot Perception and Navigation." Computer. June 1989. pp. 46-57.

[14] Fan, Ting-Jun, Medioni, Gerard, Nevatia, Ramakant. "Matching 3-D Objects Using Surface Descriptions." Proceedings IEEE International Conference on Robotics and Automation. vol. 3. 1988. pp. 1400-1406.

[15] Faugeras, O. D., Hebert, M.. "The Representation, Recognition and Positioning of 3-D Shapes from Range Data." Techniques for 3-D Machine Perception. ed. A. Rosenfeld. North-Holland: Elsevier Science Publishers B. V.. 1986. pp. 13-51.

[16] Fujimara, Kikuo, Sanet, Hanan. "Path Planning among Moving Obstacles Using Spatial Indexing." Proceedings IEEE International Conference on Robotics and Automation. vol. 3. 1988. pp. 1662-1667.

[17] Gennery, Donald B.. "Stereo Vision for the Acquisition and Tracking of Moving Three-Dimensional Objects." Techniques for 3-D Machine Perception. ed. A. Rosenfeld. North-Holland: Elsevier Science Publishers B. V.. 1986. pp. 53-74.

[18] Gigus, Ziv, Malik, Jitendra. "Computing the Aspect Graph for Line Drawings of Polyhedral Objects." Proceedings IEEE International Conference on Robotics and Automation. vol. 3. 1988. pp. 1560-1566.

[19] Goldstein, M., Pin, F. G., de Saussure, G., Weisbin, C. R.. "3-D World Modeling Based on Combinatorial Geometry for Autonomous Robot Navigation." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1987. pp. 727-733.

[20] Harmson, S. Y.. "A Report on the NATO Workshop on Mobile Robot Implementation." Proceedings IEEE International Conference on Robotics and Automation. vol. 1. 1988. pp. 604-610.

[21] Henderson, Thomas C., Weitz, Eliot, Hansen, Chuck, Grupen, Rod, Ho, C. C.,

Bhanu, Bir. "CAD-Based Robotics." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1987. pp. 631-635.

[22] Jun, Sungtaeg, Shin, Kang G.. "A Probabilistic Approach to Collision-Free Robot Path Planning." Proceedings IEEE International Conference on Robotics and Automation. vol. 1. 1988. pp. 220-225.

[23] Kak, A. C., Roberts, B. A., Andress, K. M., Cromwell, R. L.. "Experiments in the Integration of World Knowledge with Sensory Information for Mobile Robots." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1987. pp. 734-740.

[24] Kak, A. C., Vayda, A. J., Cromwell, R. L., Kim, W. Y., Chen, C. H.. "Knowledge-Based Robotics." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1987. pp. 637-646.

[25] Kelman, Laura. "Manipulator Servo Level World Modeling." NIST Technical Note 1258. December 1989.

[26] Kent, Ernest W., Albus, James S.. "Servoed World Models as Interfaces between Robot Control Systems and Sensory Data." NIST Publication. December 1983.

[27] Kriegman, David J., Binfold, Thomas O.. "Generic Models for Robot Navigation." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1988. pp. 746-751.

[28] Kriegman, David J., Triendl, Ernst, Binfold, Thomas O.. "A Mobile Robot: Sensing, Planning and Locomotion." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1987. pp. 402-408.

[29] Kuc, Roman, Siegel M. W.. "Efficient Representation of Reflecting Structures for a Sonar Navigation Model." Proceedings IEEE International Conference on Robotics and Automation. vol. 3. 1987. pp. 1916-1923.

[30] Kuipers, Benjamin J., Byun, Yung-Tai. "A Robust, Qualitative Method for Spatial Learning in Unknown Environments." Proceedings of AAAI-88. 1988. pp. 1-12.

[31] Luo, Ren C., Kay, Michael G.. "Multisensor Integration and Fusion in Intelligent Systems." IEEE Transactions on Systems, Man, and Cybernetics. vol. 19. no. 15. September/October 1989. pp. 901-929.

[32] Magee, M. J., Aggarwal, J. K.. "Using Multisensory Images to Derive the Structure of Three-Dimensional Objects -A Review." Computer Vision, Graphics and Image Processing, 32. 1985. pp. 145-157.

[33] Matthies, Larry, Elfes, Alberto. "Integration of Sonar and Stereo Range Data Using a Grid-Based Representation." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1988. pp. 727-733.

[34] Merat, Francis, Wu, Hsianglung. "Generation of Object Descriptions from Range Data Using Feature Extraction by Demands." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1987. pp. 941-946.

[35] Nasr, Hatem, Bhanu, Bir. "Landmark Recognition for Autonomous Mobile Robots." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1988. pp. 1218-1223.

[36] Ponce, Jean, Chellberg, David. "Localized Intersections Computation for Solid Modeling with Straight Homogeneous Generalized Cylinders." Proceedings IEEE International Conference on Robotics and Automation. vol. 3. 1987. pp. 1481-1484.

[37] Rao, K., Medioni, G., Liu, H., Bekey, G. A.. "Robot Hand-Eye Coordination: Shape Description and Grasping." Proceedings IEEE International Conference on Robotics and Automation. vol. 1. 1988. pp. 407-411.

[38] Rao, Nageswara S. V., Iyengar, S. S., Jorgensen, C. C., Weisbin, C. R.. "On Terrain Acquisition by a Finite-Sized Mobile Robot in Plane." Proceedings IEEE International Conference on Robotics and Automation. vol. 3. 1987. pp. 1314-1319.

[39] Rembold, Ulrich. "The Karlsruhe Autonomous Mobile Assembly Robot." Proceedings IEEE International Conference on Robotics and Automation. vol. 1. 1988. pp. 598-603.

[40] Roth-Tabak, Yuval, Jain, Ramesh. "Building an Environment Model Using Depth Information." Computer. June 1989. pp. 85-90.

[41] Schneier, Michael O., Lumia, Ronald, Kent, Ernst W.. "Model-Based Strategies for High-Level Robot Vision." Computer Vision, Graphics and Image Processing, 33. 1986. pp. 293-306.

[42] Stansfield, S. A.. "Representing Generic Objects for Exploration and Recognition." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1988. pp. 1090-1095.

[43] Triendl, Ernst, Kriegman, David J. "Stereo Vision and Navigation within Buildings." Proceedings IEEE International Conference on Robotics and Automation. vol. 3. 1987. pp. 1725-1730.

[44] Wang, Y. F., Aggarwal, J. K.. "On Modelling 3-D Objects Using Multiple Sensory

Data." Proceedings IEEE International Conference on Robotics and Automation. vol. 2. 1987. pp. 1098-1102.

3. WORLD MODEL AND SENSORY PROCESSING MODEL INTERFACE

Using the hierarchical feature based representation, the interface between the World Model and the Sensory Processing Model was designed. It is implemented for the Servo and Primitive levels of the hierarchy. The design was structured in such a way that it can be generalized to the higher levels.

```
:he interface between world modeling ---> sensory processing */
ıct io_mld2 {
   unsigned int time;
   int instance;            /* name of object */
   int partname;
   int num_feat; /* num of corners */
   double *O_W;  /* object to world matrix */
   double *C_W;                  /* camera to world */
   double *C_O;        /* camera to object */
   double init_view[3];     /* initial view of object */
   struct vertex *vertices;/* vertices of model */
   struct feature *corn;    /* corners of object in the image */
   struct io_mld2 *next;    /* link to next object */

* world modeling contains object lists and spatial volum matrix
  representation of the world */

                        /* structure storing the contents of the world */
:ruct worldcontents {
  int num_inst;                          /* number of instances in the world */
  struct object *pobj;                   /* pointer to object lists */
  struct octnode *world_map;             /* pointer to spatial representation of
                                            the world */
:
```

```
                                        *
structures for blobs and corners        *
                                        *
****************************************/


uct objects {
      double area;              /* the area of the component */
      double xcntr, ycnt,       /* the centroid of the object */
             zcnt;
      double m20, m11;          /* object moments */
      double mm10, mm01, mm20, mm02, mm11, mm21, mm12, mm03, mm30 ;
      float perim;              /* index to length of boundary (not including holes) '
      struct feature *equation;  /* index of structure containing surface equation '
      int holes;                /* number of holes */
      struct objects *olinks[2]; /* links to other objects */
      struct objects *future;                 /* link to next frame */
      struct objects *past;                   /* link to previous frame */
      int *odescription;         /* further information about the object */
      int ox3d;                 /* 3d position: x, y, z */
      int oy3d;                 /* (or a vector pointing to centroid, if */
      int oz3d;                 /* the 3d information is not known) */
      int oyaw;                /* 3d position: yaw, pitch, roll (degrees) */
      int opitch;
      int oroll;
      struct edgcoords *st_list;  /* start address of edge list */
      struct feature *s_corn, *e_corn;      /* start and end pointers for corners */
      int color;                /* object or hole */
      int xmin, xmax, ymin, ymax, /* bounding rectangle */
          zmin, zmax;
      struct objects *h_area, *l_area;      /* links to next largest and next smalle
      int otype;                /* object type or number */
      int oname;                /* name from model database */
      int oconfidence;          /* is this really the right object? */
      int fingered;             /* set to one if a feature on this object */
                                /* matched with an expected feature */

;

****************************************\
                                        *
  structure for features in database    *
                                        *
****************************************/

truct feature {
       double surfeqn[4];       /* surface equation if feature is a surface */
       int fx3d;                /* 3d position: x, y, z, yaw, pitch, roll */
       int fy3d;                /* (or a vector pointing to centroid, if */
       int fz3d;                /* the 3d information is not known) */
       int fyaw;                /* 3d position: yaw, pitch, roll (degrees) */
       int fpitch;
       int froll;
       struct feature *flinks[2];          /* links to other features */
                                        /* Note that the corners are linked through t
                                        /* flinks[0] is anti-clockwise, flinks[1] is
       char *fdescription;        /* further information about the feature */
nt fedgenum;             /* number of edge points if edge feature */
       int ftype;               /* feature type or number */
       int fname;               /* name from model database */
       struct frame *fframe;    /* pointer to header structure for picture */
       int fconfidence;         /* is this really the right feature? */
       };
```

```
                                                                    *
 structure for table entries used in mldi (dynamically allocated)   *
 the data will be used in table maintainer
                                                                    *
 ********************************************************************/

 uct featentry {                          /* a column in the table */
      int feattype;                       /* type of feature described */
      int numfeats;                       /* number in list */
      struct feature *featlist;           /* pointer to list of features */
      struct featentry *nextcol;          /* the next column */
      };

 ruct tablentry {                         /* a row in the table */
      int instname;                       /* instance identifier */
      int genericobj;                     /* genericobj num. */
      double *O_W;                        /* object to world matrix */
      int obj_confidence;
      struct featentry *entry;            /* list of "table columns" */
      struct tablentry *nextrow;          /* the next row in the Table */
      };
```

```
*************************\
                         *
  structures for edges   *
                         *
*************************/

truct edgcoords {      /* structure for edges in mldi */
             int x;
             int y;
              };

truct edges {
       int l_link, x_coord, y_coord, r_link;
;

*********************************************\
                                             *
  structure for frame-dependent information  *
                                             *
*********************************************/

truct frame {
       unsigned int ferrorstat; /* system errors */
       unsigned int ftod;       /* time of day when picture was taken */
       unsigned int fsequence;  /* sequence number for picture */
       unsigned int fpictype;   /* the picture that was requested */
       unsigned int fnumnodes;  /* number of data entries found */
       char *ffirstnode; /* pointer to first node */
       unsigned int fnummatches;/* number of matches with expectations */
       unsigned int fnumedges;  /* number of edges */
       unsigned int ftabentries; /* number of table entries */
       };

 ************************\
                        *
  Chebyshev structure   *
                        *
```

```
:uct cheby {  /* The Chebyshev coefficients, errors, and line endpoints */
      int firstx;                  /* image coordinate start and end points of segme
      int lastx;
      int cymin;               /* min y for bounding rectangle */
      int cymax;               /* max y for bounding rectangle */
      double coeffa;             /* coefficients of polynomial */
      double coeffb;             /* origin is at firstx */
      double coeffc;
      double coeffbb;           /* (alternate coefficients) */
      double coeffcc;           /* origin at midpoint of curve */
      double cerror;             /* fitting error */
      unsigned int *cequation;         /* equation of surface */
      int ctype;                       /* object type or number */
      int cname;                       /* name from model data base */
      unsigned int cnumfeats;  /* number of features */
      struct feature *chebfeats;       /* corners of segment */
      struct cheby *nextcheb;          /* next structure of this type */
      struct frame *cpicture;  /* pointer to frame info */
      int touched;                     /* flag for unpredicted blobs */
      };

*struct  flgstruct {                       /* previous user and open flag */
*        unsigned char fstatus;          */
*        unsigned char fprevuser;        */
*        };                              */
```

4. LOCAL LINEAR FEATURE EXTRACTION FROM LASER RANGE DATA

In order to construct the hierarchical representation of an object in the world model, labeled linear features must be obtained from depth data. The depth data obtained from the laser range sensor are not evenly spaced when mapped into the Cartesian Coordinate. Most of the existing vision algorithm, including the linear feature detectors, cannot be used because they generally assume equal distance existed between neighboring sampled points. An algorithm to convert range data from the laser range sensor to evenly spaced Cartesian Coordinate depth data was designed and implemented. This would allow the local linear feature detectors to be more accurate and effective. Three local linear feature detection algorithms were implemented and tested on the range image taken from the range sensor. This section describes the scheme, the algorithms, and includes the C programs for linear feature extraction.

4.1. The scheme

In our implementation, the linear feature extractions scheme contains the following steps :

1 Map range data into evenly space grid points.

2 Apply local edge operators to detect depth changes that correspond to object boundaries in the 3-D world.

3 Apply non-maxima suppression to produce eight-connected edges that are one pixel wide.

4 Find connected components in the thinned edge data.

4.2 Range data mapping algorithm

The depth data obtained from the laser range sensor are in scattered 3-D (x,y,z) form. Most of the existing vision algorithm, including the linear feature detectors, cannot be used because they generally assume equal distance existed between neighboring sampled points (i.e. in raster form). The algorithm to convert range data from the laser range sensor to evenly spaced Cartesian Coordinate depth data is broken down into the following steps :

1 Find the minimum and maximum x and y values. In practices (max x - min x) has almost the same value as (max y - min y).

2 Construct a 64 By 64 grid based on minimum and maximum values of x and y.

3 Map each range point into the grid point closest to its x, y values.

4 Evaluate each point in the grid by averaging all range values mapped to the grid point.

5 Propagate the range value to neighbors with no range values mapped to it.

4.3 Local edge detectors

An edge in a depth image corresponds to a depth discontinuity in the object or scene. The primary reason for using edges is to reduce of information to be processed while preserving spatial information. There are a large variety of edge detection algorithms in the literature, see [Dav75][Abd79][Bli84] for surveys. We implemented two local 2D edge detectors to be applied to the mapped range data obtained in step 2. The Sobel edge detector was chosen because it is one of the edge detectors most com-

monly used by researchers in robot vision. The Canny operator[Can 86] was chosen for its robustness. In addition, they are faster and simpler than other detectors that can provide the same desired combination of good detection, good localization and good response to a single edge.

We have also designed and implemented a 3-D range algorithm to detect edges. The concepts and formula used in the design are listed below :

1 Boundary edges : An edge is called a boundary edge if any of its eight neighbors' data is not available (e.g. exceeds a sensor's maximum range).

2 Discontinuous edge : a discontinuous edge exists when the value of depth changes abruptly in the neighborhood. In our implementation, eight-neighbors were used.

$$D(i,j;0) = r(i+1, j) - r(i,j); \{\text{represent right direction}\}$$

$$D2(i,j) = \max \{D(i, j; k * \Pi/4) ; k = 0, 1, 2, ..., 7\}$$

First, D2(i,j) is calculated for every point in a range image. In the second pass, all the point (i,j) such that D2 (i,j) is above a threshold value is determined as a discontinuous edge.

3 Corner edge : A corner edge is defined when two different surfaces meet. A typical detection method is to compute the difference of the surface orientations in neighbors.

$$n(i,j) = (\partial r(i,j) / \partial x, \partial r(i,j) / \partial y, -1)$$

$$\partial r(i,j) = (r(i+k,j) - r(i-k, j)) / 2k)$$

$$\partial r(i,j)/ \partial y = (r(i, j+k) - r(i, j-k)) / 2k$$

where k is a parameter, $0 < k < 3$

$\cos \alpha(i,j) = (n\ (i+k,j)\ .\ n(i-k,\ j)) / (|n\ (i+k,j)| * |n(i-k,\ j)|)$;

$\cos \beta(i,j) = (n\ (i,\ j+k)\ .\ n(i,\ j-k)) / (|n\ (i,j\ +k)| * |n(i,\ j-k)|)$;

$DD2\ (i,j) = \max\ (\ \alpha(i,j),\ \beta(i,j))$

Similar to discontinuous edge detection, first DD2(i,j) for each point (i,j) is computed. Then every point (i,j) such that DD(i,j) is above a threshold value is declared as a corner edge.

Figure 1-3. Show edge images after applying Sobel, Canny, and 3-D edge detectors respectively.

Figure 1. Edge image after applying Sobel edge detector.

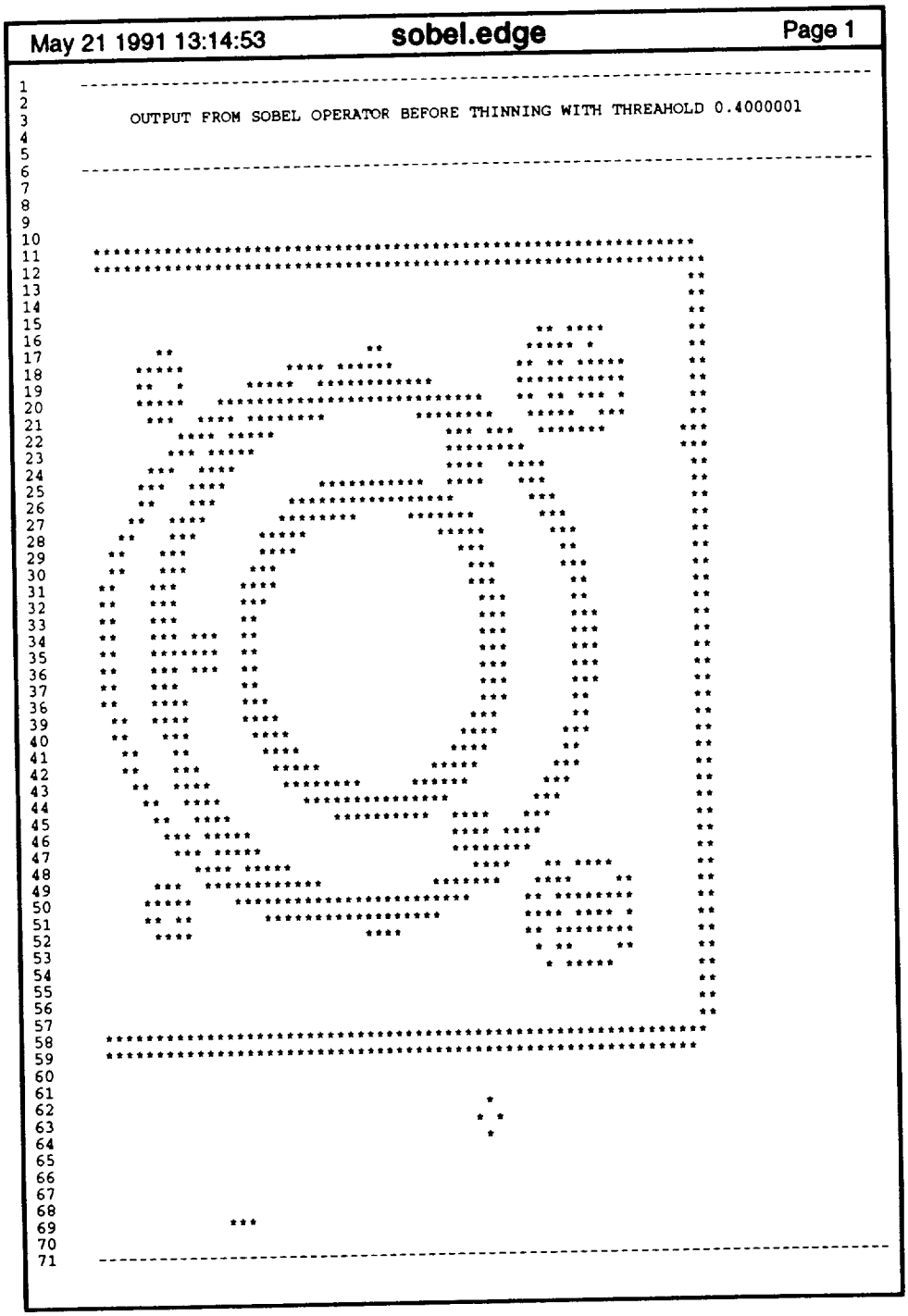


Figure 2. Edge image after applying Canny edge detector.

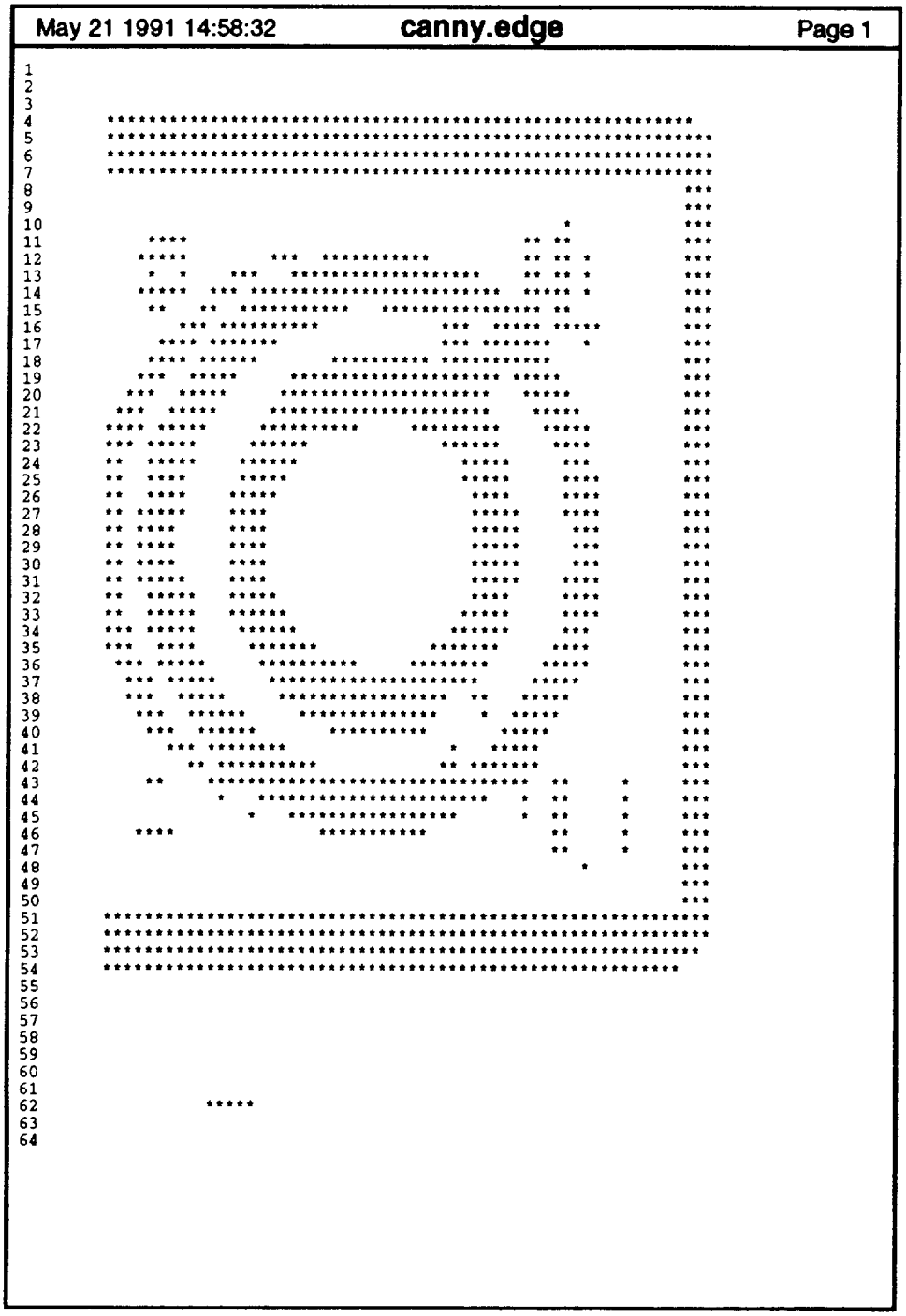
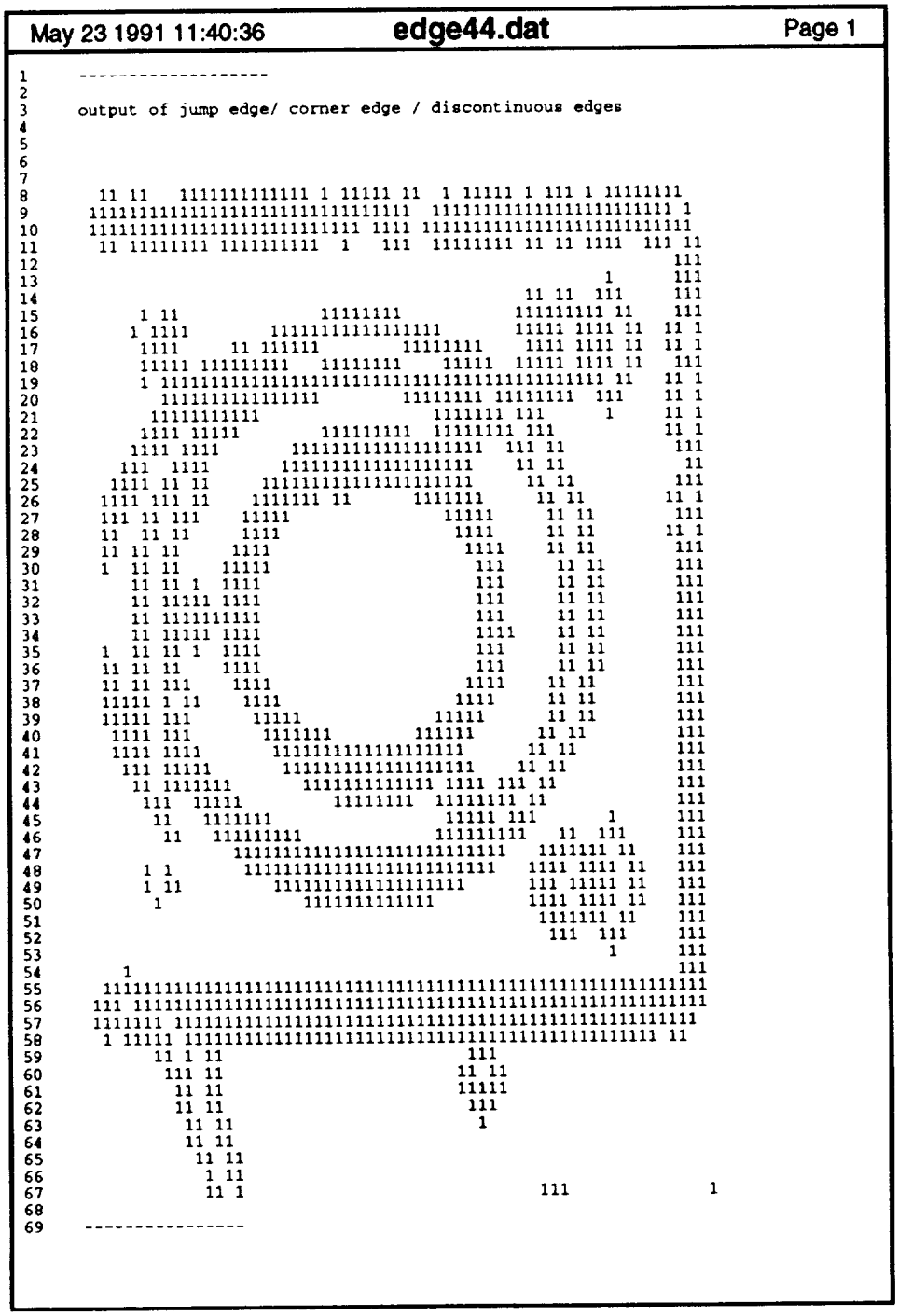


Figure 3. Edge image after applying 3-D edge detector.



4.4. Non-maximum suppression for thinning edge data

One of problem in local edge detectors is that they re quite sensitive to noise, and they usually produce thick edges. In order to localize edge reliably, non-maximum suppression is applied to the edge images.

Figure 4-6 show the results of non-maximum suppression applied to Figure 1-3 respectively.

Figure 4. Edge image after thinning Figure 1.

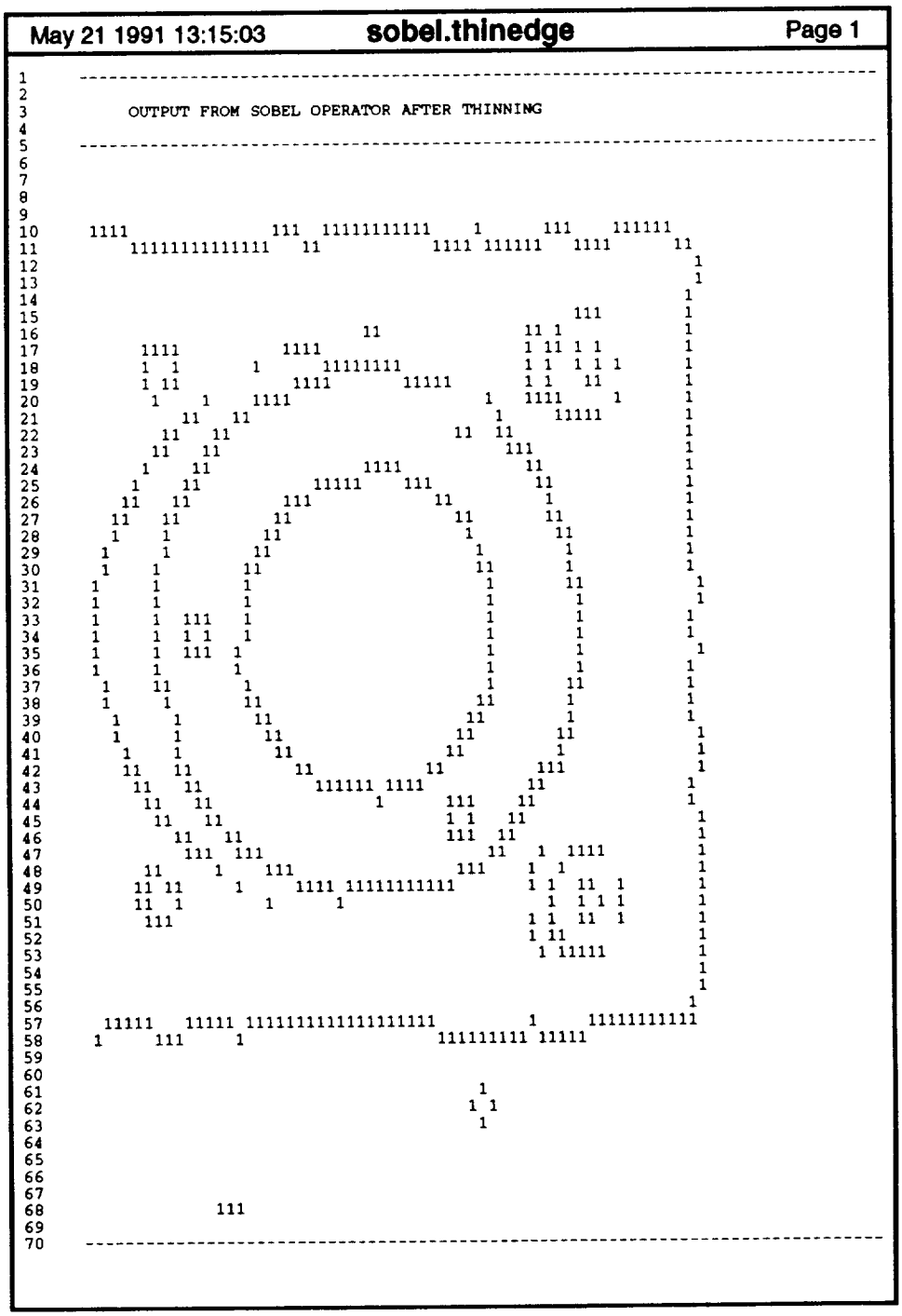
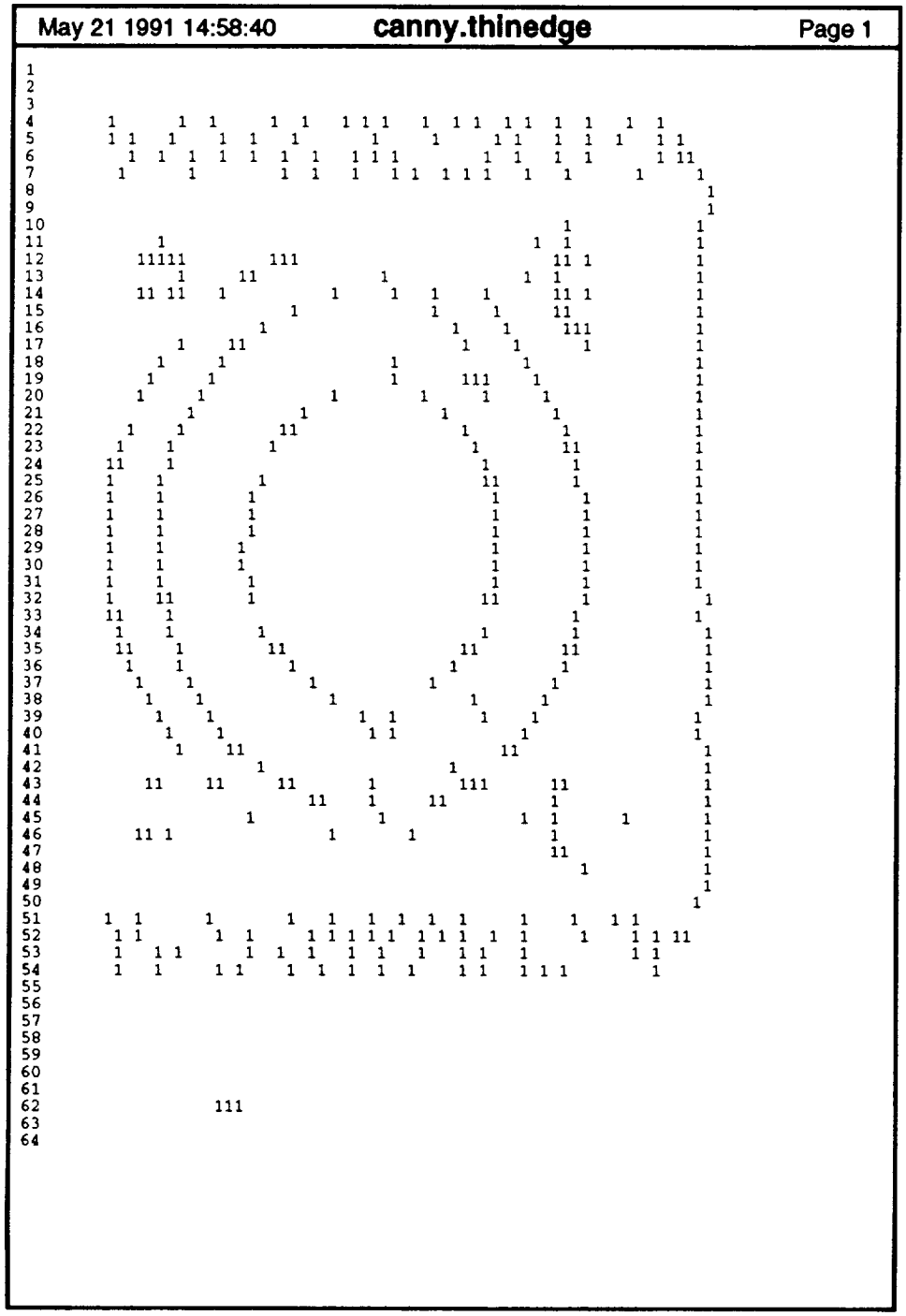


Figure 5. Edge image after thinning Figure 2.



4.5 Connected components labeling

The Connected Components (CM) algorithm labels the edge points by making raster scans of the non-maximum suppressed edge image. Edge points that are connected will have the same label. Hence the points on the same boundary of an object have the same label.

Figure 7-9 show the labeled connected components for Figures 4-6.

Figure 7. Labeled connected components of Figure 4.

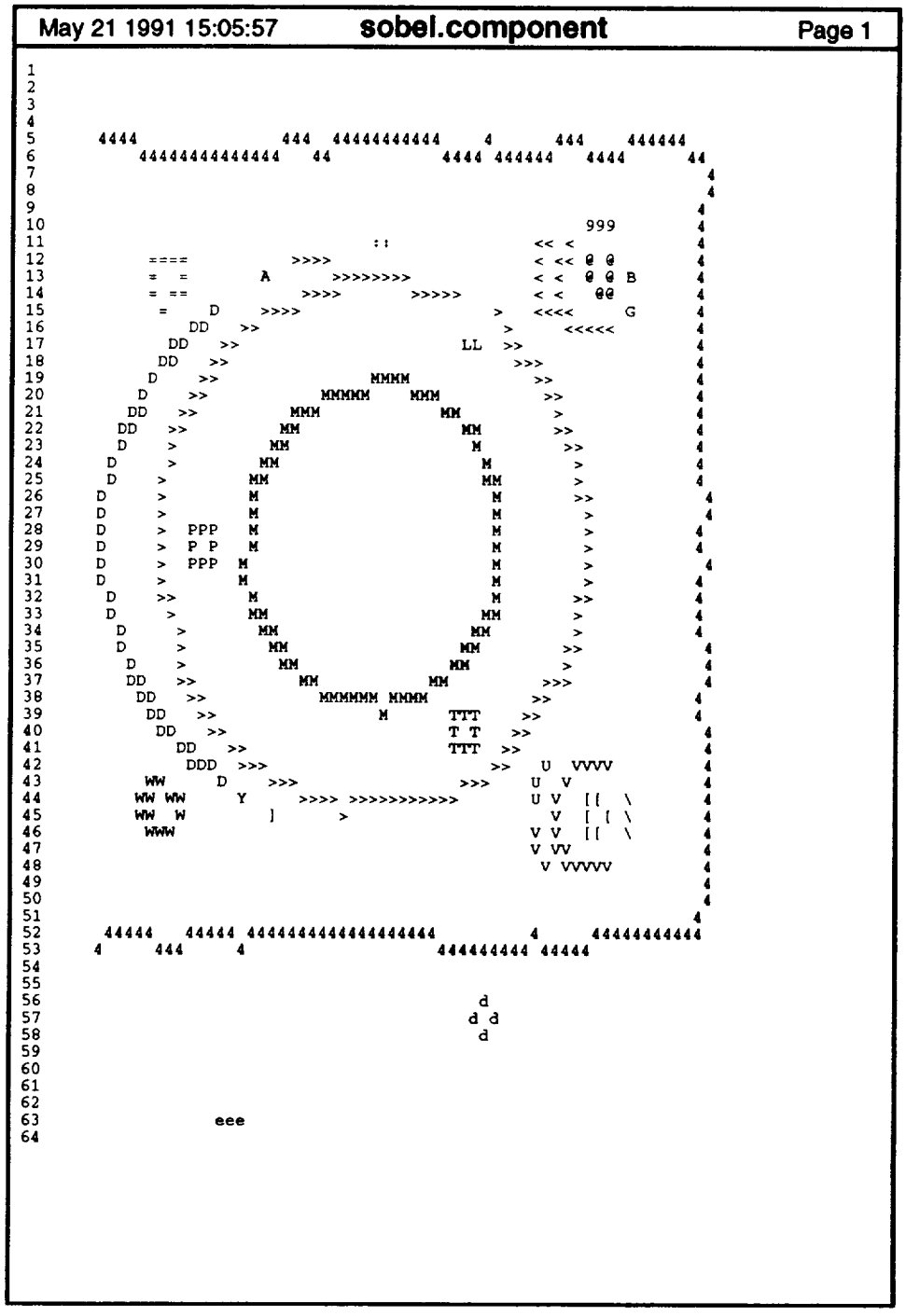
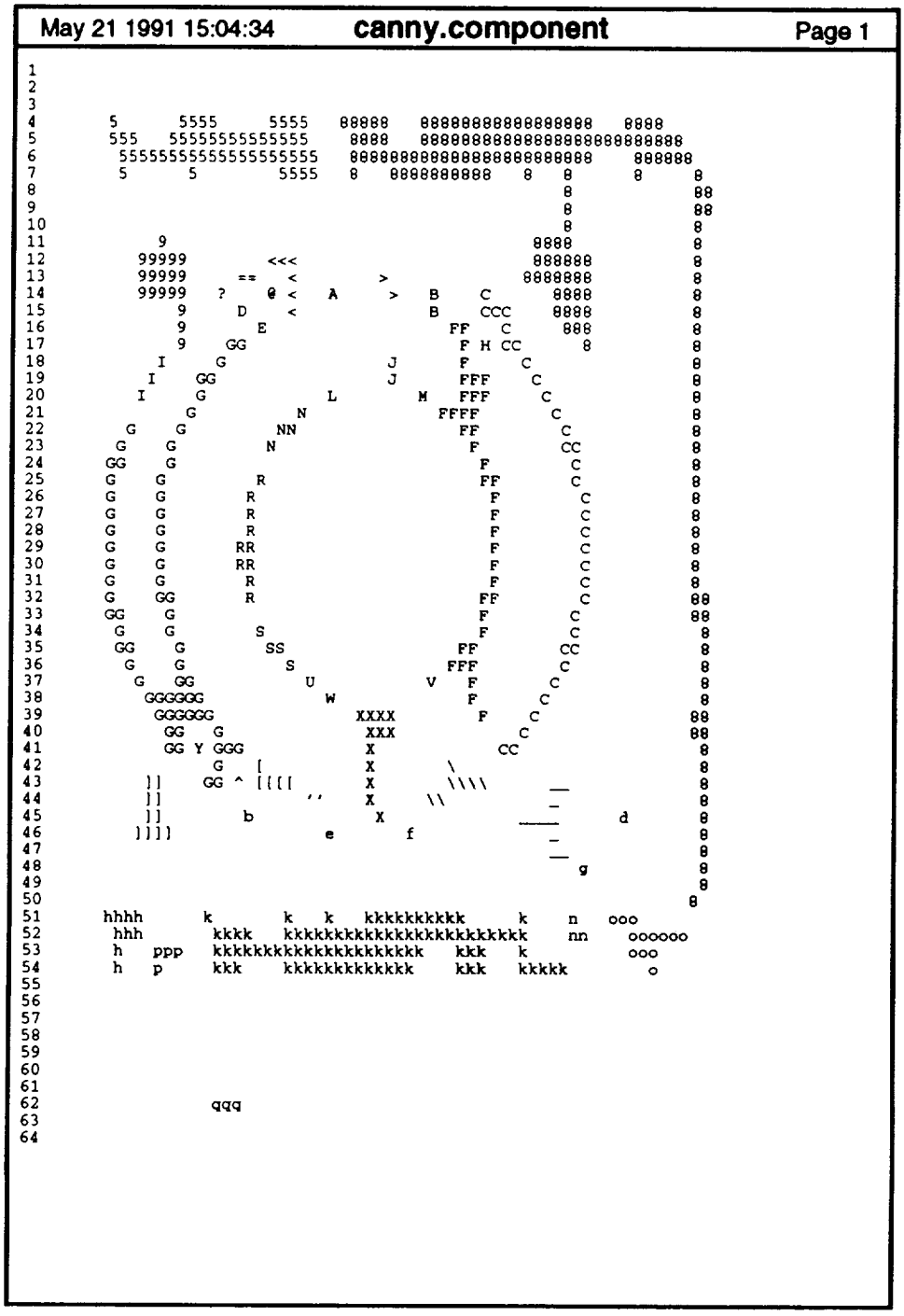


Figure 8. Labeled connected components of Figure 5.



4.6 Implementations and C programs

Range Mapping C Programs

```
#include <stdio.h>
#define ARRAY_INITIAL  (0.0)
#define LEVEL (65)
#define LARGEST_LENGTH  (4500)

double depth_image[LEVEL][LEVEL];


main () {
/* this program transforms the scattered data (x,y,z) into z(x,y) */
/* input is surf.dat ; output is depth.dat */

int length=LARGEST_LENGTH/10;
double largest[3], smallest[3], step[3];
double x_cor[LARGEST_LENGTH], y_cor[LARGEST_LENGTH], z_cor[LARGEST_LENGTH];

initialization( &length, x_cor, y_cor, z_cor );
analysis( length, largest, smallest, step, x_cor, y_cor, z_cor );
rearrange_data( length, smallest, step, x_cor, y_cor, z_cor );

}


/*************************INITIALIZATION*************************/
int initialization ( int *length, double x_cor[],
                          double y_cor[], double z_cor[] )  {
int i,j,c;
FILE *ifp;
float temp;

depth_image[0][0]=0;
for (i=1; i < LEVEL; ++i)  {
  depth_image[0][i]=i-1;
  depth_image[i][0]=i-1;
}
for (i=1; i< LEVEL; ++i)
  for (j=1; j< LEVEL; ++j)
    depth_image[i][j] = ARRAY_INITIAL;

ifp=fopen("surf.dat","r");

for (i=0; i < LARGEST_LENGTH; ++i){
  if ( fscanf(ifp,"%f", &temp) != EOF )  {
    x_cor[i] = (double)temp;
    fscanf(ifp, "%f", &temp);
    y_cor[i] = (double)temp;
    fscanf(ifp, "%f", &temp);
    z_cor[i] = (double)temp;
  }
  else  {
    *length=i;
    break;
  }
}
fclose(ifp);

}

/*************************ANALYSIS*************************************/
int analysis ( int length, double largest[],
                double smallest[], double step[],
                double x_cor[], double y_cor[], double z_cor[] )   {
int i;

largest[0] = smallest[0] = x_cor[0];
largest[1] = smallest[1] = y_cor[0];
```

```
largest[2] = smallest[2] = z_cor[0];

for (i=1; i < length; i++)  {
  if (x_cor[i] < smallest[0])
    smallest[0] = (double)x_cor[i];
  else if (x_cor[i] > largest[0])
    largest[0] = (double)x_cor[i];

  if (y_cor[i] < smallest[1])
    smallest[1] = (double)y_cor[i];
  else if (y_cor[i] > largest[1])
    largest[1] = (double)y_cor[i];

  if (z_cor[i] < smallest[2])
    smallest[2] = (double)z_cor[i];
  else if (z_cor[i] > largest[2])
    largest[2] = (double)z_cor[i];
}

for (i=0; i < 2; ++i)
     step[i] = largest[i]-smallest[i];
}


/***********************REARRANGE_DATA*********************************/
int rearrange_data(int length, double smallest[], double step[],
                   double x_cor[], double y_cor[], double z_cor[] )  {
int i,j,col,row;
int count[LEVEL][LEVEL];
FILE *ofp;

for (i=0; i< LEVEL; ++i)
  for (j=0; j< LEVEL; ++j)
       count[i][j] = 0;

for (i=0; i < length; ++i) {
       row = (int)(((LEVEL-1)* x_cor[i]-(LEVEL-1)* smallest[0])/step[0])+1;
       col = (int)(((LEVEL-1)* y_cor[i]-(LEVEL-1)* smallest[1])/step[1])+1;
       count[row][col] = count[row][col] + 1;
       depth_image[row][col] = depth_image[row][col] + z_cor[i] ;
       }

for (i=1; i< LEVEL; ++i)
  for (j=1; j< LEVEL; ++j)
       if ( count[i][j] > 1 )
           depth_image[i][j] = depth_image[i][j] / count[i][j];

ofp = fopen("depth.dat","w");
for (i=0; i<LEVEL; ++i)  {
    for (j=0; j < LEVEL; ++j)
        fprintf(ofp, "%f1 ", depth_image[i][j]);
    fprintf(ofp, "\n");
    }
fclose(ofp);
}
```

3-D Edge Detector C Programs

```
#include <stdio.h>
#define LEVEL  (65)
#define ARRAY_INITIAL  (0.0)


main()  {

double depth_image[LEVEL][LEVEL];
int edge_image[LEVEL][LEVEL];

       initialization( edge_image, depth_image );
       find_jump_edge( edge_image, depth_image );

}



/***************************INITIALIZATION*********************************/

int initialization( int edge_image[LEVEL][LEVEL],
                    double depth_image[LEVEL][LEVEL] ) {
int i,j;
FILE *ifp;
float temp;

ifp = fopen( "depth.dat", "r" );

for ( i=0; i < LEVEL; i++ )
   for ( j=0; j < LEVEL; j++ )  {
      fscanf(ifp,"%f1", &temp);
      depth_image[i][j] = (double) temp;
    }
fclose(ifp);


/* Initialize edge_image[LEVEL][LEVEL] */
for ( i=1; i < LEVEL; i++ )  {
   edge_image[0][i] = i-1;
   edge_image[i][0] = i-1;
   }
for ( i=1; i < LEVEL; i++ )
  for ( j=1; j < LEVEL; j++ )
     edge_image[i][j] = 0;

}


/***************************JUMP EDGE*********************************/

int find_jump_edge( int edge_image[LEVEL][LEVEL],
                                double depth_image[LEVEL][LEVEL] )  {
                                     /* find all jump edges points */


int i,j,k,l,jump;
FILE *ofp, *ofp2;
double jump_image[LEVEL][LEVEL];

for ( i = 2; i < (LEVEL-1); i++ )
  for ( j = 2; j < (LEVEL-1); j++ ) {              /* current is (i,j) */
    jump = 0;
    if ( depth_image[i][j] != ARRAY_INITIAL )
       for ( k = i-1; k <=  i+1; k++ )
          for ( l = j-1; l <= j+1; l++ )
            jump =  ( jump || (depth_image[k][l]==ARRAY_INITIAL) );
    if ( jump )
       edge_image[i][j] = 0;
```

```
/*        edge_image[i][j] = 1;*/
  }


                                    /* merge edge_image with depth_image, save the
                                                  resulting image as jump.dat */

 jump_image[0][0]=0;
 for ( i=1; i < LEVEL; i++ ) {
    jump_image[0][i] = i-1;
    jump_image[i][0] = i-1;
 }
 for ( i=1; i < LEVEL; i++ )
    for ( j=1; j < LEVEL; j++ )
        jump_image[i][j] = depth_image[i][j] * edge_image[i][j];

ofp = fopen( "jump.dat", "w" );
ofp2 = fopen( "edge2.dat", "w" );

for ( i=0; i < LEVEL; i++ )  {
   for ( j=0; j < LEVEL; j++ ) {
      fprintf( ofp, "%f1  ", jump_image[i][j] );
      fprintf( ofp2, "%d  ", edge_image[i][j] );
    }
   fprintf ( ofp, "%\n" );
   fprintf ( ofp2, "%\n" );
 }

fclose ( ofp );
fclose ( ofp2 );

}
```

```
#include <stdio.h>
#define DIMENSIONS   (3)
#define K 1
#define LEVEL  (65)
#define ARRAY_INITIAL  (0.0)
#define DIRECTIONS      (8)



main()  {

double depth_image[LEVEL][LEVEL];
int edge_image[LEVEL][LEVEL];

       initialization( edge_image, depth_image );
       find_discontinuous_edge( edge_image, depth_image );

}



/**************************INITIALIZATION**********************************/

int initialization( int edge_image[LEVEL][LEVEL],
                           double depth_image[LEVEL][LEVEL] )  {

int i, j, temp2;
FILE *ifp, *ifp2;
float temp;

ifp = fopen( "depth.dat", "r" );
ifp2 = fopen ( "edge2.dat", "r" );

for ( i=0; i < LEVEL; i++ )
   for ( j=0; j < LEVEL; j++ )  {
      fscanf(ifp,"%fl", &temp);
      depth_image[i][j] = (double) temp;
      fscanf( ifp2, "%d", &temp2 );
      edge_image[i][j] = temp2;
    }

fclose ( ifp );
fclose ( ifp2 );


}



/************************DISCONTINOUS EDGE*********************************/

                          /* find the maximum of the absolute eight numbers */
double maxi8 ( double d[] ) {
double max = d[0];
int i;

for ( i = 1; i < DIRECTIONS; i++ )
  if ( d[i] > max )
      max =  d[i];
return max;
}

                                  /* compute the differences in each of
                                     the eight directions, (i,j) is the
```

```
                                        current position. k is the
                                        direction. k=0: 0;  k=1: pi/4;
                                        k=2: pi/2; ...                  */

double diff (int i, int j, int k,
                        double depth_image[LEVEL][LEVEL] )  {
double temp;
switch (k) {
case 0: temp = depth_image[i][j+1]   - depth_image[i][j] ; break;
case 1: temp = depth_image[i-1][j+1] - depth_image[i][j] ; break;
case 2: temp = depth_image[i-1][j]   - depth_image[i][j] ; break;
case 3: temp = depth_image[i-1][j-1] - depth_image[i][j] ; break;
case 4: temp = depth_image[i][j-1]   - depth_image[i][j] ; break;
case 5: temp = depth_image[i+1][j-1] - depth_image[i][j] ; break;
case 6: temp = depth_image[i+1][j]   - depth_image[i][j] ; break;
case 7: temp = depth_image[i+1][j+1] - depth_image[i][j] ; break;
   }

if ( temp > 0.0 )
     return temp;
else return (-temp);

}


int find_discontinuous_edge( int edge_image[LEVEL][LEVEL],
                                 double depth_image[LEVEL][LEVEL] )  {
                                        /* find all discontinuous edges */

int i, j, k;
double d[DIRECTIONS], temp;
FILE *ofp;

ofp = fopen ( "thres1.dat", "w" );
for ( i = 2; i < (LEVEL-2); i++ )
   for ( j = 2; j < (LEVEL-2); j++ ) {
                                        /* current is at (i,j) */
      if ((depth_image[i][j] != ARRAY_INITIAL)
                        && (edge_image[i][j] == 0))  {
         for ( k=0; k < DIRECTIONS; k++ )
            d[k] = diff( i, j, k, depth_image );
         temp = maxi8( d );
         fprintf ( ofp, "%f\n",  temp );
       }
      }
   fclose ( ofp );


}
```

```
#include <stdio.h>
#include <math.h>
#define DIMENSIONS   (3)
#define K 1
#define LEVEL  (65)
#define ARRAY_INITIAL  (0.0)
#define DIRECTIONS      (8)

/* function prototypes */
double cos_sita( int, int, double[][] );
double cos_beta( int, int, double[][] );
double maximum2(double, double);



main()  {

double depth_image[LEVEL][LEVEL];
int edge_image[LEVEL][LEVEL];

       initialization( edge_image, depth_image );
       find_corner_edge( edge_image, depth_image );

}

/***************************INITIALIZATION*********************************/
int initialization ( int edge_image[LEVEL][LEVEL],
                         double depth_image[LEVEL][LEVEL] ) {

float temp1;
FILE *ifp, *ifp2;
int i, j, temp2;

ifp  = fopen( "depth.dat", "r" );
ifp2 = fopen( "edge33.dat", "r" );

for ( i=0; i < LEVEL; i++ )
   for ( j=0; j < LEVEL; j++ )  {
      fscanf(ifp,"%f1", &temp1);
      depth_image[i][j] = (double) temp1;
      fscanf ( ifp2, "%d", &temp2 );
      edge_image[i][j] = temp2;
    }
fclose ( ifp );
fclose ( ifp2 );

}


/***********************************CORNER EDGE*****************************/
find_corner_edge( int edge_image[LEVEL][LEVEL],
                           double depth_image[LEVEL][LEVEL]) {
                                  /* Find all corner edge points */

FILE * ofp;
double sita, beta, temp;
int i,j;

ofp = fopen ( "thres2.dat", "w" );

for ( i = 2 * K+1; i < (LEVEL-1) - 2 * K; i++ )
   for (j = 2 * K+1; j < (LEVEL-1) - 2 * K; j++ )  {
                                                  /* current is (i,j) */
      if (( depth_image[i][j] != ARRAY_INITIAL )
                           && (edge_image[i][j]==0)) {
          sita = cos_sita( i, j, depth_image );
```

```
          beta = cos_beta( i, j, depth_image );
          temp = maximum2 ( sita, beta ) * 180.0 / 3.1415926;
          fprintf ( ofp, "%f1\n" , temp );
        }
    }
fclose ( ofp );


}

/*******************************/

int normal_vector( int i, int j, double normal[],
                        double depth_image[LEVEL][LEVEL] ) {

  normal[0] = ( depth_image[i+K][j] - depth_image[i-K][j] ) / (2*K);
  normal[1] = ( depth_image[i][j+K] - depth_image[i][j-K] ) / (2*K);
  normal[2] = -1.0;
}


/*******************************/
double magnitude ( double n[] ) {
double temp;
temp = n[0] * n[0] + n[1] * n[1] + n[2] * n[2];
return ( temp );
}


/*******************************/
double dot_product( double n1[], double n2[] )  {
double temp;
   temp = n1[0] * n2[0] + n1[1] * n2[1] + n1[2] * n2[2] ;
   return temp;
}


/*******************************/
double cos_sita ( int i, int j,
                      double depth_image[LEVEL][LEVEL] ) {

double temp1[DIMENSIONS], temp2[DIMENSIONS], m1,m2;
double temp;
double fabs();

normal_vector ( i+K, j, temp1, depth_image );
normal_vector ( i-K, j, temp2, depth_image );
m1 = magnitude (temp1);
m2 = magnitude (temp2);
if (fabs(m1*m2*10)> 0.00001)
    temp = dot_product( temp1, temp2 ) / ( m1 * m2 );
else
     printf("divided by zero\n");
return temp;

}


/*******************************/
double cos_beta( int i, int j,
                double depth_image[LEVEL][LEVEL] ) {

double temp1[DIMENSIONS], temp2[DIMENSIONS], m1,m2;
double temp;
double fabs();
```

```
normal_vector( i, j+K, temp1, depth_image );
normal_vector( i, j-K, temp2, depth_image );
m1 = magnitude ( temp1 );
m2 = magnitude ( temp2 );

if (fabs(m1*m2*10)> 0.00001)
    temp = dot_product( temp1, temp2 ) / ( m1 * m2 );
else
    printf("diveded by zero\n");
return temp;
}



/*******************************/
double maximum2 ( double x, double y ) {  /* find maxinum of two number */

double t1, t2;

t1 = acos ( x );
t2 = acos ( y );
if ( t1 > t2 )
     return t1;
else return t2;

}
```

```
#include <stdio.h>
#define DIMENSIONS    (3)
#define LEVEL   (65)
#define ARRAY_INITIAL   (0.0)
#define DIRECTIONS       (8)
#define THRESHOLD1       (0.380)



main()  {

double depth_image[LEVEL][LEVEL];
int edge_image[LEVEL][LEVEL];

        initialization( edge_image, depth_image );
        find_discontinuous_edge( edge_image, depth_image );

}



/**************************INITIALIZATION**********************************/

int initialization( int edge_image[LEVEL][LEVEL],
                          double depth_image[LEVEL][LEVEL] )   {

int i, j, temp2;
FILE *ifp, *ifp2;
float temp;

ifp = fopen( "depth.dat", "r" );
ifp2 = fopen ( "edge2.dat", "r" );

for ( i=0; i < LEVEL; i++ )
   for ( j=0; j < LEVEL; j++ )  {
      fscanf(ifp,"%fl", &temp);
      depth_image[i][j] = (double) temp;
      fscanf( ifp2, "%d", &temp2 );
      edge_image[i][j] = temp2;
    }

fclose ( ifp );
fclose ( ifp2 );


}



/************************DISCONTINOUS EDGE*********************************/

                          /* find the maximum of eight numbers */
double maxi8 ( double d[] ) {
double max = d[0];
int i;

for ( i = 1; i < DIRECTIONS; i++ )
  if ( d[i] > max )
      max = d[i] ;
return max;
}
```



```
                          /* compute the differences in each of
                             the eight directions, (i,j) is the
```

```
                                        current position. k is the
                                        direction. k=0: 0;  k=1: pi/4;
                                        k=2: pi/2; ...                  */

double diff (int i, int j, int k,
                        double depth_image[LEVEL][LEVEL] )  {
double temp;

switch (k) {
case 0: temp = depth_image[i][j+1]   - depth_image[i][j] ; break;
case 1: temp = depth_image[i-1][j+1] - depth_image[i][j] ; break;
case 2: temp = depth_image[i-1][j]   - depth_image[i][j] ; break;
case 3: temp = depth_image[i-1][j-1] - depth_image[i][j] ; break;
case 4: temp = depth_image[i][j-1]   - depth_image[i][j] ; break;
case 5: temp = depth_image[i+1][j-1] - depth_image[i][j] ; break;
case 6: temp = depth_image[i+1][j]   - depth_image[i][j] ; break;
case 7: temp = depth_image[i+1][j+1] - depth_image[i][j] ; break;
   }

if ( temp > 0.0 )
      return temp;
else  return (-temp);

}


int find_discontinuous_edge( int edge_image[LEVEL][LEVEL],
                                double depth_image[LEVEL][LEVEL] )  {
                                        /* find all discontinuous edges */


int i, j, k;
double d[DIRECTIONS], temp;
FILE *ofp, *ofp2;
double discontinuous_image[LEVEL][LEVEL];

ofp = fopen ( "edge33.dat", "w" );
ofp2 = fopen ( "discon.dat", "w" );

for ( i = 2; i < (LEVEL-2); i++ )
   for ( j = 2; j < (LEVEL-2); j++ )                    /* current is at (i,j) */
      if ((depth_image[i][j] != ARRAY_INITIAL)
                           && (edge_image[i][j] == 0))  {
         for ( k=0; k < DIRECTIONS; k++ )
            d[k] = diff( i, j, k, depth_image );
         temp = maxi8( d );
         if ( temp > THRESHOLD1)
                edge_image[i][j] = 1;
          }

                                /* merge edge_image with depth_image, and
                                        save the result to discon.dat*/


discontinuous_image[0][0]=0;
for (i=1; i < LEVEL; i++) {
   discontinuous_image[0][i]=i-1;
   discontinuous_image[i][0]=i-1;
  }
for (i=1; i < LEVEL; i++)
   for (j=1; j < LEVEL; j++)
      if ( edge_image[i][j] == 2 )
            discontinuous_image[i][j] = depth_image[i][j] ;
      else  discontinuous_image[i][j] = 0.0;

for (i=0; i < LEVEL; i++)  {
   for (j=0; j < LEVEL; j++)  {
```

```
      fprintf ( ofp, "%d  ", edge_image[i][j] );
      fprintf( ofp2, "%fl ",discontinuous_image[i][j] );
    }
   fprintf ( ofp, "%\n" );
   fprintf ( ofp2, "%\n" );
   }
fclose ( ofp );
fclose ( ofp2 );

}
```

```
#include <stdio.h>
#include <math.h>
#define DIMENSIONS    (3)
#define K 1
#define LEVEL  (65)
#define ARRAY_INITIAL  (0.0)
#define DIRECTIONS      (8)
#define THRESHOLD2     (15.0)

/* function prototypes */
double cos_sita( int, int, double[][] );
double cos_beta( int, int, double[][] );
double maximum2(double, double);



main()  {
/*this program find the corner edges */

double depth_image[LEVEL][LEVEL];
int edge_image[LEVEL][LEVEL];

       initialization( edge_image, depth_image );
       find_corner_edge( edge_image, depth_image );

}

/***************************INITIALIZATION*********************************/
int initialization ( int edge_image[LEVEL][LEVEL],
                          double depth_image[LEVEL][LEVEL] ) {

float temp1;
FILE *ifp, *ifp2;
int i, j, temp2;

ifp  = fopen( "depth.dat", "r" );
ifp2 = fopen( "edge33.dat", "r" );

for ( i=0; i < LEVEL; i++ )
   for ( j=0; j < LEVEL; j++ )  {
      fscanf(ifp,"%f1", &temp1);
      depth_image[i][j] = (double) temp1;
      fscanf ( ifp2, "%d", &temp2 );
      edge_image[i][j] = temp2;
    }
fclose ( ifp );
fclose ( ifp2 );

}


/***********************************CORNER EDGE*****************************/
find_corner_edge( int edge_image[LEVEL][LEVEL],
                           double depth_image[LEVEL][LEVEL]) {
                                   /* Find all corner edge points */

FILE * ofp;
double sita, beta, temp;
int i,j;


for ( i = 2 * K+1; i < (LEVEL-1) - 2 * K; i++ )
   for (j = 2 * K+1; j < (LEVEL-1) - 2 * K; j++ )  {
                                                    /* current is (i,j) */
      if (( depth_image[i][j] != ARRAY_INITIAL )
                           && (edge_image[i][j]==0)) {
```

```
            sita = cos_sita( i, j, depth_image );
            beta = cos_beta( i, j, depth_image );
            temp = maximum2 ( sita, beta ) * 180.0 / 3.1415926;
            if ( temp > THRESHOLD2 )
               edge_image[i][j] = 3;
         }
      }

ofp = fopen( "edge44.dat", "w" );

fprintf ( ofp, "--------------------\n\n" );
fprintf ( ofp, "output of jump edge/ corner edge / discontinuous edges\n\n" );
for ( i = 1; i < LEVEL; i++ ) {
   for ( j = 1; j < LEVEL; j++ ){
     if ( edge_image[i][j] <= 0)
       fprintf ( ofp, " " );
     else fprintf ( ofp, "1" );
     }
   fprintf ( ofp, "\n" );
 }

fprintf ( ofp, "----------------\n" );
fclose ( ofp );


}

/*******************************/

int normal_vector( int i, int j, double normal[],
                          double depth_image[LEVEL][LEVEL] ) {

  normal[0] = ( depth_image[i+K][j] - depth_image[i-K][j] ) / (2*K);
  normal[1] = ( depth_image[i][j+K] - depth_image[i][j-K] ) / (2*K);
  normal[2] = -1.0;
}


/*******************************/
double magnitude ( double n[] ) {
double temp;
temp = n[0] * n[0] + n[1] * n[1] + n[2] * n[2];
return ( temp );
}


/*******************************/
double dot_product( double n1[], double n2[] )  {
double temp;
   temp = n1[0] * n2[0] + n1[1] * n2[1] + n1[2] * n2[2] ;
   return temp;
}


/*******************************/
double cos_sita ( int i, int j,
                      double depth_image[LEVEL][LEVEL] ) {

double temp1[DIMENSIONS], temp2[DIMENSIONS], m1,m2;
double temp;
double fabs();

normal_vector ( i+K, j, temp1, depth_image );
normal_vector ( i-K, j, temp2, depth_image );
m1 = magnitude (temp1);
m2 = magnitude (temp2);
```

```
if (fabs(m1*m2*10)> 0.00001)
    temp = dot_product( temp1, temp2 ) / ( m1 * m2 );
else
     printf("divided by zero\n");
return temp;

}


/*******************************/
double cos_beta( int i, int j,
                 double depth_image[LEVEL][LEVEL] ) {

double temp1[DIMENSIONS], temp2[DIMENSIONS], m1,m2;
double temp;
double fabs();

normal_vector( i, j+K, temp1, depth_image );
normal_vector( i, j-K, temp2, depth_image );
m1 = magnitude ( temp1 );
m2 = magnitude ( temp2 );

if (fabs(m1*m2*10)> 0.00001)
    temp = dot_product( temp1, temp2 ) / ( m1 * m2 );
else
    printf("diveded by zero\n");
return temp;
}



/*******************************/
double maximum2 ( double x, double y ) {  /* find maxinum of two number */

double t1, t2;

t1 = acos ( x );
t2 = acos ( y );
if ( t1 > t2 )
     return t1;
else return t2;

}
```

```
#include <stdio.h>
#define LEVEL  (65)
#define ARRAY_INITIAL  (0.0)

double average_neighbor(int, int, double [][]);

main()  {

/* this program fill the miss data with interpolation*/
double depth_image[LEVEL][LEVEL];

       initialization( depth_image );
       fill_in( depth_image);
}



/**************************INITIALIZATION**********************************/

int initialization( double depth_image[LEVEL][LEVEL] ) {
int i,j;
FILE *ifp;
float temp;

ifp = fopen( "depth.dat", "r" );

for ( i=0; i < LEVEL; i++ )
   for ( j=0; j < LEVEL; j++ )  {
      fscanf(ifp,"%fl", &temp);
      depth_image[i][j] = (double) temp;
    }
fclose(ifp);

}


/**************************FILL_IN**********************************/

int fill_in( double depth_image[LEVEL][LEVEL]) {

/* use neighbor average to fill in miss data*/

int i,j,done;
FILE *ofp;
double temp_image[LEVEL][LEVEL], smooth_image[LEVEL][LEVEL];
double fabs( double );


for(i=0; i<LEVEL; i++)
   for (j=0; j<LEVEL; j++)
         temp_image[i][j] = depth_image[i][j];

smooth_image[0][0] = 0.0;
for(i=0; i<LEVEL; i++) {
        smooth_image[i][0] = i-1;
        smooth_image[0][i] = i-1;
      }

done = 0;
while (!done) {
for (i=1; i<LEVEL; i++)
   for(j=1; j<LEVEL; j++){
     if ( temp_image[i][j] == ARRAY_INITIAL ) {
        if ( i==1 )
           smooth_image[i][j] = temp_image[i+1][j];
        else if( j==1 )
```

```
            smooth_image[i][j] = temp_image[i][j+1];
        else if ( i == LEVEL-1 )
            smooth_image[i][j] = temp_image[i-1][j];
        else if ( j == LEVEL-1 )
            smooth_image[i][j] = temp_image[i][j-1];
        else
            smooth_image[i][j] = average_neighbor(i,j, temp_image);
      }
     else smooth_image[i][j] = temp_image[i][j];
   }

done = 1;
/*check to see if it really done*/
for (i=1; i<LEVEL; i++)
   for(j=1; j<LEVEL; j++)
      if(fabs( smooth_image[i][i]-ARRAY_INITIAL ) < 0.0001)
         done = 0;

/*prepare for the next loop*/
for (i=1; i<LEVEL; i++)
   for(j=1; j<LEVEL; j++)
     temp_image[i][j] = smooth_image[i][j];

}

ofp = fopen("smooth.dat", "w");
for (i=0; i<LEVEL; i++) {
   for(j=0; j<LEVEL; j++)
    fprintf( ofp, "%fl ", smooth_image[i][j]);
   fprintf( ofp, "\n" );
 }

close(ofp);
}

/**********************AVERAGE_NEIGHBOR*************************/
double average_neighbor(int i, int j, double image[LEVEL][LEVEL]){
/*fill in image[i][j] with local average*/

int row, col, count;
double sum, avg;

count=0;
sum=0;
for(row=i-1; row<=i+1; row++)
    for(col=j-1; col<=j+1; col++)
        if(image[row][col]!= ARRAY_INITIAL) {
            count=count+1;
            sum=sum+image[row][col];
          }
if (count != 0) {
   avg = sum / count;
   return( avg );
 }
else return( ARRAY_INITIAL);
}
```

Canny Edge Detector C Programs

```
#include <stdio.h>
#include <math.h>
#include  <sys/file.h>
#define CANNY_THRESHOLD  0.580

#define SIZE    65                      /* size of input image  */
#define TSIZE   8                       /* size of templates */
#define GSIZE   5
#define HALFGSIZE 2
#define SQ2PI   2.506628275             /*  = sqrt(2.0*PI) */

/* Macros */
#define    MAX_SHORT  32767
#define    CNST       (MAX_SHORT / (255. * 25.))
                                        /* Constant for maximum res */
#define    SQSIG      (6.25 / log(CNST))      /* 0.5 * 2 * 6.25 = 6.25 */
#define    maskfun(x,s)    ( -( x / s ) * exp( -x * x / ( 2.0 * s ) ) )
                                          /* Make Gaussian mask */
#define    NBRS       24                    /* Number of neighbours */
#define    ALFA       3.14159/8.          /* angle increment in radian */

/* Coordinates of neighbours, numbered clockwise from zero at X-axis */
char nx[NBRS] = { 3,3,2,1,1,1,2,3,4,4,4,3,2,1,0,0,0,0,0,1,2,3,4,4 };
char ny[NBRS] = { 2,3,3,3,2,1,1,1,2,3,4,4,4,4,4,3,2,1,0,0,0,0,0,1 };

int enter_g( float g[TSIZE][GSIZE][GSIZE], float n[TSIZE] );
int read_data( float pic[SIZE][SIZE], int *rows, int *cols );
int edge_detector( float inputpic[SIZE][SIZE], int rows, int cols,
               float g[TSIZE][GSIZE][GSIZE], float n[TSIZE],
               int dir[SIZE][SIZE], float mag[SIZE][SIZE] );
int output_pic ( int dir[SIZE][SIZE], float mag[SIZE][SIZE] );
int nonmaxima_suppression ( float mag[SIZE][SIZE], int dir[SIZE][SIZE] );

/**************************MAIN*************************/
main() {
  float g[TSIZE][GSIZE][GSIZE];         /* gaussian filter */
  float pic[SIZE][SIZE];                /* input image */
  int   cols, rows;                     /* size of image */
  float n[TSIZE];
  float mag[SIZE][SIZE];
  int   dir[SIZE][SIZE];

  enter_g( g, n );                      /* enter canny operators */
  read_data( pic, &rows, &cols );       /* open files and read images */
  printf ( "applying edge detector to the  pic...\n" ); /* find edges */
  edge_detector( pic, rows, cols, g, n, dir, mag );
  output_pic ( dir, mag );
  nonmaxima_suppression ( mag, dir );
  printf( "All done\n" );

}


/*********************************ENTER_G*************************/
enter_g( float g[TSIZE][GSIZE][GSIZE], float n[TSIZE] ){
     /* g is gaussian filter, n is norm */
  int i,j, k ;          /* index */
  int o_height, o_width, o_y_cntr, o_x_cntr;
  float s, x;
  short dx, dy;

  s = SQSIG;       /* enter Gaussian templates */
  o_height = o_width = GSIZE;
  o_y_cntr = o_x_cntr = HALFGSIZE;

  for ( k = 0; k < TSIZE; k++ ) {     /* Make 8 directional templates */
```

```
    for ( j = 0; j < GSIZE; j++ ) {
      dy = j - o_y_cntr;
      for ( i = 0; i < GSIZE; i++ ) {
        dx = i - o_x_cntr;
        x  = dx * cos ( k * ALFA ) - dy * sin ( k * ALFA );
        g[k][j][i] = CNST * maskfun ( x, s );
      }
    }
  }

  for ( k = 0; k < TSIZE; k++ ) {      /* Compute a norm of each template */
    n[k] = 0.0;
    for ( j = 0; j < GSIZE; j++ )
      for ( i = 0; i < GSIZE; i++ )
          n[k] += fabs( g[k][j][i] );
  }

}


/*************************READ_DATA*****************************/
read_data( float pic[SIZE][SIZE], int *rows, int *cols )
/* the image contains floating points data */{

  int i, j;
  FILE *ifp;
  float temp;

  ifp = fopen ( "smooth.dat", "r" );

  *rows = 65;
  *cols = 65;

  for ( i = 0; i < *rows; i++ )
    for ( j = 0; j < *cols; j++ ){
      fscanf ( ifp, "%f1", &temp );
      pic[i][j] = temp;
    }


  fclose ( ifp );

  printf( "done reading\n" );
}



/***********************EDGE_DETECTOR*************************/
/* smooth image with gaussian, doing rows first,
   then columns.
*/
edge_detector( float inputpic[SIZE][SIZE], int rows, int cols,
               float g[TSIZE][GSIZE][GSIZE], float n[TSIZE],
               int dir[SIZE][SIZE], float mag[SIZE][SIZE] ){

  int d, dir_point, ii, jj, i, j, dx, dy, temp;
  float mag_point, sum, curr_mag, a;

  for ( i = 0; i < SIZE; i++ )
    for ( j = 0; j< SIZE; j++ ){
      mag[i][j] = 0.0;
      dir[i][j] = 0.0;
    }

  for ( ii = HALFGSIZE + 1; ii< SIZE - HALFGSIZE; ii++ )
    for ( jj = HALFGSIZE + 1; jj< SIZE - HALFGSIZE; jj++ ){
```

```
      /* Find template with strongest match */

      mag_point = -MAX_SHORT;  /* Ridiculously low value */

      for( d = 0;  d < TSIZE;  d++ ) {
        /* Compute scalar product of rotated template with neighborhood */
        sum = 0.0;
        for( i = 0;  i < GSIZE;  i++ )
          for ( j = 0; j < GSIZE; j++ )
            sum += inputpic[ii-2+i][jj-2+j] * g[d][i][j];
        curr_mag = ((sum < 0.0) ? (sum - n[d]/2)/n[d] :
                                                (sum + n[d]/2)/n[d]);
        if((a = fabs( curr_mag )) > mag_point ) {
          mag_point = a;
          dir_point = d;  /* Remember strongest & where */
         }
      }/*for d*/
          mag[ii][jj] = mag_point;
          dir[ii][jj] = dir_point;

    }

printf("done edge operator \n");
}


/*************************************OUTPUT_PIC*****************/
output_pic ( int dir[SIZE][SIZE], float mag[SIZE][SIZE] ){

  float temp;
  FILE *ofp;
  int i, j;

  ofp = fopen ( "canny.edge", "w" );

  for ( i = 1; i < SIZE; i++ ){
    for ( j = 1; j < SIZE; j++ )
      if ( mag[i][j] > CANNY_THRESHOLD )
        fprintf( ofp, "*" );
      else {
        fprintf ( ofp, " " );
        mag[i][j] = 0.0;
      }

      fprintf ( ofp, "\n" );
  }

  fclose ( ofp );

}

/***************NONMAXIMA_SUPPRESSION************************/
int nonmaxima_suppression ( float mag[SIZE][SIZE], int dir[SIZE][SIZE] ){
  int i,j;
  float x1, x2, temp;
  FILE *ofp;
  FILE *ofp1;
  int thinedge[SIZE][SIZE];

  for ( i = 0; i < SIZE; i++ )
    for ( j = 0; j < SIZE; j++ )
      thinedge[i][j] = 0;

  for ( i = 2; i < SIZE - 1; i++ )
    for ( j = 2; j < SIZE - 1; j++ )
```

```
      if ( mag[i][j] > 0 ){
        switch ( dir[i][j] ) {
              case 0 :
              case 4 : x1 = mag[i][j-1];
                       x2 = mag[i][j+1];
                       break;
              case 1 :
              case 5 : x1 = mag[i+1][j-1];
                       x2 = mag[i-1][j+1];
                       break;
              case 2 :
              case 6 : x1 = mag[i-1][j];
                       x2 = mag[i+1][j];
                       break;
              case 3 :
              case 7 : x1 = mag[i-1][j-1];
                       x2 = mag[i+1][j+1];
                       break;
             default : printf( " error\n" );
        }

        x1 = ( x2 > x1 )? x2 : x1;
        temp = mag[i][j];
        if (( temp > CANNY_THRESHOLD ) && ( temp > x1 ))
          thinedge[i][j] = 1;
      }

  fill_edge ( thinedge, dir );

  ofp  = fopen ( "canny.thinedge", "w" );
  ofp1 = fopen ( "canny.thinnedge", "w" );

  for ( i = 1; i < SIZE; i++ ){
    for ( j = 1; j < SIZE; j++ )
      if ( thinedge[i][j] ){
        fprintf( ofp1, "%d ", 1 );
        fprintf ( ofp, "1" );
      }
      else {
        fprintf ( ofp1, "%d ", 0 );
        fprintf ( ofp, " " );
      }
      fprintf ( ofp1, "\n" );
      fprintf ( ofp, "\n" );
  }

  fclose ( ofp );
  fclose ( ofp1 );


}


/***************************FILL_EDGE*********************************/
fill_edge( int thinedge[SIZE][SIZE], int dir[SIZE][SIZE] ){

  int temp[SIZE][SIZE];
  int i,j;


  for ( i = 0; i < SIZE; i++ )
    for ( j = 0; j < SIZE; j++ )
      temp[i][j] = 0;


  for ( i = 2; i < SIZE - 1; i++ )
```

```
    for ( j = 2; j < SIZE - 1; j++ ){
      if ( thinedge[i][j] == 0 ){
        if ( thinedge[i][j+1] && (( dir[i][j+1] = 2) || ( dir[i][j+1] = 6)))
          temp[i][j]++;
        if ( thinedge[i][j-1] && (( dir[i][j-1] = 2) || ( dir[i][j-1] = 6)))
          temp[i][j]++;
        if ( thinedge[i-1][j+1]&&(( dir[i-1][j+1]=3)||( dir[i-1][j+1] = 7)))
          temp[i][j]++;
        if ( thinedge[i+1][j-1]&&(( dir[i+1][j-1]=3)||( dir[i+1][j-1] = 7)))
          temp[i][j]++;
        if ( thinedge[i-1][j] && (( dir[i-1][j] = 0) || ( dir[i-1][j] = 4)))
          temp[i][j]++;
        if ( thinedge[i+1][j] && (( dir[i+1][j] = 0) || ( dir[i+1][j] = 4)))
          temp[i][j]++;
        if ( thinedge[i-1][j-1]&&(( dir[i-1][j-1]=1)||( dir[i-1][j-1] = 5)))
          temp[i][j]++;
        if ( thinedge[i+1][j+1]&&(( dir[i+1][j+1]=1)||( dir[i+1][j+1] = 5)))
          temp[i][j]++;
      }
    }

  for ( i = 1; i < SIZE; i++ )
    for ( j = 1; j < SIZE; j++ )
      if (( thinedge[i][j] == 0 ) && ( temp[i][j] > 1 ))
        thinedge[i][j] = 1;
}
```

```
#include <stdio.h>
#include <math.h>
#define MAX_COMPONENT    100   /* max number of components */
#define LEVEL 65
#define COMPONENT_SIZE 3   /* smallest size of one component */
#define LOOP_COUNT 1     /* number of times to expand and shrink */

int data_in ( int edge_image[LEVEL][LEVEL] );
int expansion_shrinking ( int edge_image[LEVEL][LEVEL] );
int expand ( int i, int j, int image[LEVEL][LEVEL] );
int shrink ( int i, int j, int image[LEVEL][LEVEL] );
int connected_component ( int edge_image[LEVEL][LEVEL],
                          int component_image[LEVEL][LEVEL] );
int find ( int i, int parent[MAX_COMPONENT] );
int uunion ( int x, int y, int parent[MAX_COMPONENT] );


/************************************MAIN*****************************/
main (){

  int edge_image[LEVEL][LEVEL],
      component_image[LEVEL][LEVEL];

  data_in ( edge_image );
  expansion_shrinking ( edge_image );
  connected_component ( edge_image , component_image );
  output_component ( component_image );

}


/*****************************DATA IN*****************************************/
int data_in ( int edge_image[LEVEL][LEVEL] ){

  FILE *ifp;
  int i,j, tempi;

  for ( i = 0; i < LEVEL; i++ )
    for ( j = 0; j < LEVEL; j++ )
      edge_image[i][j] = 0;

  ifp = fopen ( "canny.thinnedge", "r" );

  for ( i = 1; i < LEVEL; i++ )
    for ( j = 1; j < LEVEL; j++ ){
      fscanf ( ifp, "%d", &tempi );
      edge_image[i][j] = tempi;
    }

}


/*****************************EXPANSION AND SHRINKING****************/
int expansion_shrinking ( int edge_image[LEVEL][LEVEL] ) {

  int temp_image [LEVEL][LEVEL];
  int i,j,loop_count;

  for ( loop_count = 1; loop_count <= LOOP_COUNT; loop_count++ ){
    for ( i = 2; i < LEVEL-1; i++ )
      for ( j = 2; j < LEVEL-1; j++ )
        temp_image[i][j] = expand ( i, j, edge_image );
    for ( i = 1; i < LEVEL; i++ )
      for ( j = 1; j < LEVEL; j++ )
        edge_image[i][j] = temp_image[i][j];
  }
```

```
  for ( loop_count = 1; loop_count <= LOOP_COUNT; loop_count++ ){
    for ( i = 2; i < LEVEL-1; i++ )
      for ( j = 2; j < LEVEL-1; j++ )
        temp_image[i][j] = shrink ( i, j, edge_image );
    for ( i = 1; i < LEVEL; i++ )
      for ( j = 1; j < LEVEL; j++ )
        edge_image[i][j] = temp_image[i][j];
  }

}

/*****************************EXPAND*******************************/

int expand ( int i, int j, int image[LEVEL][LEVEL] ){

  if ( image[i][j] )
    return ( 1 );
  else
    if ( image[i-1][j-1] || image[i-1][j] || image[i-1][j+1] || image[i][j-1]
         || image[i][j+1]|| image[i+1][j-1]|| image[i+1][j]
         || image[i+1][j+1] )
      return ( 1 );
    else
      return ( 0 );
}


/***************************SHRINK*********************************/

int shrink ( int i, int j, int image[LEVEL][LEVEL] ){

  if (  image[i-1][j-1] && image[i-1][j] && image[i-1][j+1] && image[i][j-1]
        && image[i][j+1] && image[i+1][j-1] && image[i+1][j]
        && image[i+1][j+1] )
    return ( 1 );
  else
    return ( 0 );

}



/*****************************FIND CONNECTED COMPONENT*****************/
int connected_component ( int edge_image[LEVEL][LEVEL],
                          int component_image[LEVEL][LEVEL] ){

  int i, j, set_count, flag;
  int parent[MAX_COMPONENT];
  int count[MAX_COMPONENT];

  for ( i = 0; i < MAX_COMPONENT; i++ )
    parent[i] = -1;

/* when parent[i] > 0, it points to its parent node. when it is negative,
   it is the root of a connected component, the absolute value is the
   number of nodes in that component. */

  set_count = 1;

  for ( i = 0; i < LEVEL; i++ )
    for ( j = 0; j < LEVEL; j++ )
      component_image[i][j] = 0;

  for ( j = 1; j < LEVEL; j++ )
    if (  edge_image[1][j] )
```

```
    component_image[1][j] = set_count;
else
  if ( j > 1 )
    if ( edge_image[1][j-1] )
      set_count++;

for ( i = 2; i < LEVEL; i++ )
  if ( edge_image[i][1] )
    component_image[i][1] = set_count;
else
  if ( edge_image[i-1][1] )
    set_count++;

if (( component_image[1][1]) &&  (component_image[2][1]) )
  uunion ( component_image[1][1], component_image[2][1], parent );

for ( i = 2; i < LEVEL; i++ )
  for ( j = 2; j < LEVEL - 1; j++ ){

    flag = 0;
    if ( edge_image[i][j] ){ /*******/
      if ( edge_image[i][j-1] ){
        component_image[i][j] = component_image[i][j-1];
        flag = 1;
      }
      else
        if ( edge_image[i-1][j-1]  ){
          component_image[i][j] = component_image[i-1][j-1];
          flag = 2;
        }
        else
          if ( edge_image[i-1][j]  ){
            component_image[i][j] = component_image[i-1][j];
            flag = 3;
          }
          else
            if ( edge_image[i-1][j+1]  ){
              component_image[i][j] = component_image[i-1][j+1];
              flag = 4;
            }
            else /* start a new component */ {
              set_count++;
              component_image[i][j] = set_count;
            }
    } /******/

    if ( flag == 1 ){
      if ( component_image[i-1][j]  )
        uunion(component_image[i][j-1], component_image[i-1][j], parent);
      if ( component_image[i-1][j+1]  )
        uunion(component_image[i][j-1], component_image[i-1][j+1], parent);
    }
    else
      if ( flag == 2 )
        if ( component_image[i-1][j+1] )
          uunion( component_image[i-1][j-1],component_image[i-1][j+1],
                  parent );

  }/* j loop */

for ( i = 1; i < LEVEL; i++ )
  for ( j = 1; j < LEVEL; j++ )
    if ( component_image[i][j] )
      component_image[i][j] = find ( component_image[i][j], parent );
```

```
  for ( i = 0; i < MAX_COMPONENT; i++ )
    count[i] = 0;

  for ( i = 1; i < LEVEL; i++ )
    for ( j = 1; j < LEVEL; j++ )
      if ( component_image[i][j] )
        count[component_image[i][j]]++;

    for ( i = 1; i < LEVEL; i++ )
      for ( j = 1; j < LEVEL; j++ )
        if ( count[ component_image[i][j] ] < COMPONENT_SIZE )
          component_image[i][j] = 0;
}

/*************************************FIND*****************************/
int find ( int x, int parent[MAX_COMPONENT] ){

  int p, prep;

  prep = p = x;
  while ( p > 0 ){
    prep = p;
    p = parent[p];
  }
  return prep;
}

/*********************************UNION*****************************************/
int uunion ( int x, int y, int parent[MAX_COMPONENT] ){

  int r1, r2, c1, c2;

  r1 = find ( x, parent );
  r2 = find ( y, parent );

  if ( r1 != r2 ){
    c1 = abs ( parent[r1] );
    c2 = abs ( parent[r2] );
    if ( c1 > c2 ){
      parent[r1] = - ( c1 + c2 );
      parent[r2] = r1;
    }
    else {
      parent[r2] = - ( c1 + c2 );
      parent[r1] = r2;
    }
  }
}


/*********************************OUTPUT_COMPONENT*************************/
output_component ( int component_image[LEVEL][LEVEL] ){

  int i,j;
  FILE *ofp;

  ofp = fopen ( "canny.component", "w" );

  for ( i = 1; i < LEVEL; i++ ){
    for ( j = 1; j < LEVEL; j++ )
      if ( component_image[i][j] == 0 )
        fprintf ( ofp, " " );
      else
        fprintf ( ofp, "%c",49 +  component_image[i][j] );
    fprintf ( ofp, "\n" );
```

```
  }

  fclose ( ofp );
}
```

Sobel Edge Detector C Programs

```
/*****************************************************************************

  THIS PROGRAM USE SOBEL OPERATOR TO FIND EDGES

   TEMPLATE1 :                    TEMPLATE2 :
   -1  0  1                        1  2   1
   -2  0  2                        0  0   0
   -1  0  1                       -1 -2  -1


   INPUT : SMOOTH.DAT
   OUTPOUT : SOBEL.EDGE ----raw edge
             SOBEL.THINEDGE ---thin edge
             SOBEL.THINNEDGE ---thin edge in numerical form

5/21/91
 *****************************************************************************/


#include <stdio.h>
#include <math.h>
#define ARRAY_INITIAL  ( 0.0 )
#define SIZE  ( 65 )
#define SOBEL_THRESHOLD  ( 0.4 )
#define PI  ( 3.1415926 )


int initialization ( double depth_image [ ][ SIZE ],
                     double edge_image  [ ][ SIZE ],
                     double angle_image [ ][ SIZE ]);
/* read in the depth data and initialize edge_image and angle_image to be blank */

double template1 ( int i, int j, double depth_image [ ][ SIZE ] );
/* calculate the increment in x direction */

double template2 ( int i, int j, double depth_image [ ][ SIZE ] );
/* calculate the increment in y direction */

double max4 ( double x1, double x2, double x3, double x4 );
/* calculate the maximum of the 4 numbers */

int sobel_operator ( double depth_image [ ][ SIZE ],
                     double edge_image  [ ][ SIZE ],
                     double angle_image [ ][ SIZE ] );
/* apply sobel operator to depth_image and return edge_image and angle_image for la

int nonmaxima_suppression ( double image_image  [ ][ SIZE ],
                            double angle_image [ ][ SIZE ] );
/* apply non-maxima suppression to the edge image */




/************MAIN************/
main ( ){

  double depth_image [ SIZE ][ SIZE ];
  double edge_image  [ SIZE ][ SIZE ];
  double angle_image [ SIZE ][ SIZE ];

  initialization ( depth_image, edge_image, angle_image );
  sobel_operator ( depth_image, edge_image, angle_image );
  nonmaxima_suppression ( edge_image, angle_image );

}
```

```
/**************INITIALIZATION********/
int initialization ( double depth_image [ ][ SIZE ],
                     double edge_image  [ ][ SIZE ],
                     double angle_image [ ][ SIZE ] ){

  int i, j;
  FILE *ifp;
  float temp;

  ifp = fopen ( "smooth.dat", "r" );

  for ( i = 0; i < SIZE; i++ )
    for ( j = 0; j < SIZE; j++ ){
      fscanf ( ifp, "%f1", &temp );
      depth_image [i][j] = temp;
    }

  fclose ( ifp );

  for ( i = 1; i < SIZE; i++ ){
    edge_image [0][i] = i - 1;
    edge_image [i][0] = i - 1;
    angle_image[0][i] = i - 1;
    angle_image[i][0] = i - 1;
  }

  for ( i = 1; i < SIZE; i++ )
    for ( j = 1; j < SIZE; j++ ){
      edge_image [i][j] = 0.0;
      angle_image[i][j] = 0.0;
    }

}/* INITIALIZATION */


/********TEMPLATE1********/
double template1 ( int i, int j, double depth_image[SIZE][SIZE] ){

  double temp1, temp2;

  temp1  =     depth_image[i-1][j-1];
  temp1 += 2 * depth_image[i  ][j-1];
  temp1 +=     depth_image[i+1][j-1];

  temp2  =     depth_image[i-1][j+1];
  temp2 += 2 * depth_image[i  ][j+1];
  temp2 +=     depth_image[i+1][j+1];

  return ( temp2 - temp1 );

}


/********TEMPLATE2********/
double template2 ( int i, int j, double depth_image[SIZE][SIZE] ){

  double temp1, temp2;

  temp1  =     depth_image[i+1][j-1];
  temp1 += 2 * depth_image[i+1][j  ];
  temp1 +=     depth_image[i+1][j+1];

  temp2  =     depth_image[i-1][j-1];
  temp2 += 2 * depth_image[i-1][j  ];
```

```
  temp2 +=     depth_image[i-1][j+1];

  return ( temp2 - temp1 );

}


/********SOBEL_OPERATOR*******/
int sobel_operator ( double depth_image [ ][ SIZE ],
                     double edge_image  [ ][ SIZE ],
                     double angle_image [ ][ SIZE ]  ){

  int i, j;
  FILE *opf;
  double magnitude, dx,dy;

  for ( i = 2; i < SIZE - 1; i++ ){

    for ( j = 2; j < SIZE - 1; j++ ){

      dx = template1 ( i, j, depth_image );
      dy = template2 ( i, j, depth_image );
      magnitude = dx * dx + dy * dy;

      if ( magnitude > SOBEL_THRESHOLD ){
        edge_image[i][j] = magnitude;
        angle_image[i][j] = atan2 ( dy, dx ) ;
      }
    }

  }

/* output the current result */
  opf = fopen ( "sobel.edge", "w" );

  for ( i = 1; i < 80; i++ )
    fprintf ( opf, "-" );
  fprintf ( opf, "\n\n" );
  fprintf ( opf,
  "     OUTPUT FROM SOBEL OPERATOR BEFORE THINNING WITH THREAHOLD %f1
                                        \n\n", SOBEL_THRESHOLD );
  for ( i = 1; i < 80; i++ )
    fprintf ( opf, "-" );
  fprintf ( opf, "\n" );

  for ( i = 1; i < SIZE; i++ ){
    for ( j = 1; j < SIZE; j++ )
      if ( edge_image[i][j] > SOBEL_THRESHOLD )
        fprintf ( opf, "*" );
      else fprintf ( opf, " " );
    fprintf ( opf, "\n" );
  }
    for ( i = 1; i < 80; i++ )
    fprintf ( opf, "-" );


  fclose ( opf );

}


/***********MAX4**********/
double max4 ( double x1, double x2, double x3, double x4 ){

  double t1,t2;
```

```
  if ( x1 > x2 )
    t1 = x1;
  else t1 = x2;
  if ( x3 > x4 )
    t2 = x3;
  else t2 = x4;
  if ( t1 > t2 )
    return t1;
  else return t2;

}


/************NON-MAXIMA SUPPRESSION***********/
int nonmaxima_suppression ( double edge_image  [ ][ SIZE ],
                            double angle_image [ ][ SIZE ]){

  int i, j, flag;
  double alfa, x1, x2, temp;
  FILE *opf;
  FILE *opf1;
  int thinedge[ SIZE ][ SIZE ];

  for ( i = 0; i < SIZE; i++ )
    for ( j = 0; j < SIZE; j++ )
      thinedge[i][j] = 0;

  for ( i = 2; i < SIZE - 1; i++ )
    for ( j = 2; j < SIZE - 1; j++ ){

      alfa = angle_image[i][j];
      while ( alfa < 0 )
        alfa += 2 * PI;
      while ( alfa > 2 * PI )
        alfa -= 2 * PI;
      /* alfa is in [0, 2*PI] */

      flag = 0;
      while ( alfa > PI / 4 ){
        alfa -= PI / 4;
        flag++;
      }
      if ( alfa > PI / 8 )
        flag++;

      switch ( flag ){
            case 0 :
            case 4 :
            case 8 : x1 = edge_image[i][j-1];
                     x2 = edge_image[i][j+1];
                     break;
            case 1 :
            case 5 : x1 = edge_image[i+1][j-1];
                     x2 = edge_image[i-1][j+1];
                     break;
            case 2 :
            case 6 : x1 = edge_image[i-1][j];
                     x2 = edge_image[i+1][j];
                     break;
            case 3 :
            case 7 : x1 = edge_image[i-1][j-1];
                     x2 = edge_image[i+1][j+1];
                     break;
            dafault : printf ( " error\n " );
      }
```

```
      x1 = ( x2 > x1 )? x2 : x1 ;
      temp = edge_image[i][j];
      if (( temp > SOBEL_THRESHOLD ) && ( temp >= x1 ))
        thinedge[i][j] = 1;

    }

  opf  = fopen ( "sobel.thinedge", "w" );
  opf1 = fopen ( "sobel.thinnedge", "w" );

  for ( i = 1; i < 80; i++ )
    fprintf ( opf , "-" );

  fprintf ( opf, "%\n\n" );
  fprintf ( opf,
      "     OUTPUT FROM SOBEL OPERATOR AFTER THINNING  " );
  fprintf ( opf, "\n\n" );

  for ( i = 1; i < 80; i++ )
    fprintf ( opf, "-" );
  fprintf ( opf, "\n" );

  for ( i = 1; i < SIZE; i++ ){
    for ( j = 1; j < SIZE; j++ ){
      if ( thinedge[i][j] )
        fprintf ( opf, "%d", thinedge[i][j] );
      else fprintf ( opf, " " );
      fprintf ( opf1, "%d ", thinedge[i][j] );
    }
    fprintf ( opf, "\n" );
    fprintf ( opf1, "\n" );
  }

  for ( i = 1; i < 80; i++ )
    fprintf ( opf, "-" );

  fclose ( opf );
  fclose ( opf1 );
}
```

```
#include <stdio.h>
#include <math.h>
#define MAX_COMPONENT    100   /* max number of components */
#define LEVEL 65
#define COMPONENT_SIZE 0   /* smallest size of one component */
#define LOOP_COUNT 1     /* number of times to expand and shrink */

int data_in ( int edge_image[LEVEL][LEVEL] );
int expansion_shrinking ( int edge_image[LEVEL][LEVEL] );
int expand ( int i, int j, int image[LEVEL][LEVEL] );
int shrink ( int i, int j, int image[LEVEL][LEVEL] );
int connected_component ( int edge_image[LEVEL][LEVEL],
                          int component_image[LEVEL][LEVEL] );
int find ( int i, int parent[MAX_COMPONENT] );
int uunion ( int x, int y, int parent[MAX_COMPONENT] );


/*************************************MAIN*****************************/
main (){

  int edge_image[LEVEL][LEVEL],
      component_image[LEVEL][LEVEL];

  data_in ( edge_image );
  expansion_shrinking ( edge_image );
  connected_component ( edge_image , component_image );
  output_component ( component_image );

}


/*******************************DATA IN*****************************************/
int data_in ( int edge_image[LEVEL][LEVEL] ){

  FILE *ifp;
  int i,j, tempi;

  for ( i = 0; i < LEVEL; i++ )
    for ( j = 0; j < LEVEL; j++ )
      edge_image[i][j] = 0;

  ifp = fopen ( "sobel.thinnedge", "r" );

  for ( i = 1; i < LEVEL; i++ )
    for ( j = 1; j < LEVEL; j++ ){
      fscanf ( ifp, "%d", &tempi );
      edge_image[i][j] = tempi;
    }

}


/*****************************EXPANSION AND SHRINKING***************/
int expansion_shrinking ( int edge_image[LEVEL][LEVEL] ) {

  int temp_image [LEVEL][LEVEL];
  int i,j,loop_count;

  for ( loop_count = 1; loop_count <= LOOP_COUNT; loop_count++ ){
    for ( i = 2; i < LEVEL-1; i++ )
      for ( j = 2; j < LEVEL-1; j++ )
        temp_image[i][j] = expand ( i, j, edge_image );
    for ( i = 1; i < LEVEL; i++ )
      for ( j = 1; j < LEVEL; j++ )
        edge_image[i][j] = temp_image[i][j];
  }
```

```
  for ( loop_count = 1; loop_count <= LOOP_COUNT; loop_count++ ){
    for ( i = 2; i < LEVEL-1; i++ )
      for ( j = 2; j < LEVEL-1; j++ )
        temp_image[i][j] = shrink ( i, j, edge_image );
    for ( i = 1; i < LEVEL; i++ )
      for ( j = 1; j < LEVEL; j++ )
        edge_image[i][j] = temp_image[i][j];
  }

}

/*****************************EXPAND*******************************/

int expand ( int i, int j, int image[LEVEL][LEVEL] ){

  if ( image[i][j] )
    return ( 1 );
  else
    if ( image[i-1][j-1] || image[i-1][j] || image[i-1][j+1] || image[i][j-1]
         || image[i][j+1]|| image[i+1][j-1]|| image[i+1][j]
         || image[i+1][j+1] )
      return ( 1 );
    else
      return ( 0 );
}


/*************************SHRINK*******************************/

int shrink ( int i, int j, int image[LEVEL][LEVEL] ){

  if (  image[i-1][j-1] && image[i-1][j] && image[i-1][j+1] && image[i][j-1]
        && image[i][j+1] && image[i+1][j-1] && image[i+1][j]
        && image[i+1][j+1] )
    return ( 1 );
  else
    return ( 0 );

}







/*****************************FIND CONNECTED COMPONENT*****************/
int connected_component ( int edge_image[LEVEL][LEVEL],
                          int component_image[LEVEL][LEVEL] ){

  int i, j, set_count, flag;
  int parent[MAX_COMPONENT];
  int count[MAX_COMPONENT];

  for ( i = 0; i < MAX_COMPONENT; i++ )
    parent[i] = -1;

/* when parent[i] > 0, it points to its parent node. when it is negative,
   it is the root of a connected component, the absolute value is the
   number of nodes in that component. */

  set_count = 1;

  for ( i = 0; i < LEVEL; i++ )
```

```
  for ( j = 0; j < LEVEL; j++ )
    component_image[i][j] = 0;

for ( j = 1; j < LEVEL; j++ )
  if (  edge_image[1][j] )
    component_image[1][j] = set_count;
else
  if ( j > 1 )
    if ( edge_image[1][j-1] )
      set_count++;

for ( i = 2; i < LEVEL; i++ )
  if ( edge_image[i][1] )
    component_image[i][1] = set_count;
else
  if ( edge_image[i-1][1] )
    set_count++;

if (( component_image[1][1]) &&  (component_image[2][1]) )
  uunion ( component_image[1][1], component_image[2][1], parent );

for ( i = 2; i < LEVEL; i++ )
  for ( j = 2; j < LEVEL - 1; j++ ){

    flag = 0;
    if ( edge_image[i][j] ){ /*******/
      if ( edge_image[i][j-1] ){
        component_image[i][j] = component_image[i][j-1];
        flag = 1;
      }
      else
        if ( edge_image[i-1][j-1]  ){
          component_image[i][j] = component_image[i-1][j-1];
          flag = 2;
        }
        else
          if ( edge_image[i-1][j]  ){
            component_image[i][j] = component_image[i-1][j];
            flag = 3;
          }
          else
            if ( edge_image[i-1][j+1]  ){
              component_image[i][j] = component_image[i-1][j+1];
              flag = 4;
            }
            else /* start a new component */ {
              set_count++;
              component_image[i][j] = set_count;
            }
    } /******/

    if ( flag == 1 ){
      if ( component_image[i-1][j]  )
        uunion(component_image[i][j-1], component_image[i-1][j], parent);
      if ( component_image[i-1][j+1]  )
        uunion(component_image[i][j-1], component_image[i-1][j+1], parent);
    }
    else
      if ( flag == 2 )
        if ( component_image[i-1][j+1] )
          uunion( component_image[i-1][j-1],component_image[i-1][j+1],
                  parent );

  }/* j loop */

for ( i = 1; i < LEVEL; i++ )
```

```
    for ( j = 1; j < LEVEL; j++ )
      if ( component_image[i][j] )
        component_image[i][j] = find ( component_image[i][j], parent );



  for ( i = 0; i < MAX_COMPONENT; i++ )
    count[i] = 0;

  for ( i = 1; i < LEVEL; i++ )
    for ( j = 1; j < LEVEL; j++ )
      if ( component_image[i][j] )
        count[component_image[i][j]]++;

    for ( i = 1; i < LEVEL; i++ )
      for ( j = 1; j < LEVEL; j++ )
        if ( count[ component_image[i][j] ] < COMPONENT_SIZE )
          component_image[i][j] = 0;
}

/*************************************FIND*****************************/
int find ( int x, int parent[MAX_COMPONENT] ){

  int p, prep;

  prep = p = x;
  while ( p > 0 ){
    prep = p;
    p = parent[p];
  }
  return prep;
}

/*********************************UNION*****************************************/
int uunion ( int x, int y, int parent[MAX_COMPONENT] ){

  int r1, r2, c1, c2;

  r1 = find ( x, parent );
  r2 = find ( y, parent );

  if ( r1 != r2 ){
    c1 = abs ( parent[r1] );
    c2 = abs ( parent[r2] );
    if ( c1 > c2 ){
      parent[r1] = - ( c1 + c2 );
      parent[r2] = r1;
    }
    else {
      parent[r2] = - ( c1 + c2 );
      parent[r1] = r2;
    }
  }
}


/*********************************OUTPUT_COMPONENT*****************************/
output_component ( int component_image[LEVEL][LEVEL] ){

  int i,j;
  FILE *ofp;

  ofp = fopen ( "sobel.component", "w" );

  for ( i = 1; i < LEVEL; i++ ){
    for ( j = 1; j < LEVEL; j++ )
```

```
      if ( component_image[i][j] == 0 )
        fprintf ( ofp, " " );
      else
        fprintf ( ofp, "%c",49 +  component_image[i][j] );
    fprintf ( ofp, "\n" );
  }

  fclose ( ofp );
}
```

4.7 References

[Can86]

J. Canny, "A computational approach to edge detection," IEEE PAMI, pp679-698, Nov. 1986.

[Abd79]

I. E. Abdou and W. K. Pratt, " Quantitative design and evaluation of enhancement/thresholding edge detectors," Proc. IEEE, pp753-763, May 1979.

[Bli84]

P. Blicher, " Edge detection and geometric methods in computer vision," Ph.D. dissertation, Dept. Math. Univ. California, Berkeley, Oct. 1984.

[Dav75]

L. S. Davis, "A survey of edge detection techniques," CGIP, pp248-270, Sept. 1975.

5. CONCLUDING REMARKS

The model and representation designed and implemented for the Servo and Primitive levels can be easily extended to the higher levels. This will be useful in finding surface features in higher levels, and in building a global model to be used for local path planning, object tracking, object recognition and navigation.

HARPS (Hierarchical Ada Robot Programming System) uses camera input (light intensity data). We believe that our result complements and enhance HARPS. The Y-frame model and data structures were implemented in ADA, which can be incorporated into HARPS easily.

The results and experience from this research project will help guide future research in world modeling and sensor processing with range data. The laser sensor results can be used in studies on sensor fusion.