A Graphic Simulator for Robotic Workcells

Faycal Kahloun

B Eng., Université du Québec à Trois-Rivières

Department of Electrical Engineering

McGill University

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

M. Eng., McGill University, 1987

March 12, 1987

RESEARCH THESES/REPORTS

THESIS NO. 87 3T

TITLE: A Graphic Simulation for Robotic Workcells

AUTHOR(S) Faycal Kahloun

Department of Electrical Engineering

McGill University

Montreal, CANADA

DATE: March 1987

GRANT OR CONTRACT: FCAR

NO. OF PAGES: 111

SUPERVISOR: Dr. Malowany

This thesis is dedicated to the memory of my dear friend and valued companion Mounir Slama.

Abstract

As robotic applications are becoming more and more complex, robotic workcells are evolving to include varied and complicated configurations involving multiple robots working in conjunction with sensing devices. Programming such a system is a difficult and tedious task. The use of a simulator provides a testbed for debugging incremental changes and developing new application programs. Furthermore, debugging becomes safer, since actual equipment is not used, and program development can proceed during the production cycle. Moreover, the simulator usually requires a graphics interface, to create a more powerful and user-friendly programming environment. This thesis presents a solution to the problem of robotic workcell simulation by decomposing it into smaller problems of database, data sublanguage, solid modelling, motion of solids in 3-D, robot kinematics, robot programming and graphics.

Résumé

Comme les applications en robotique deviennent de plus en plus complexes, les cellules de travail robotisées sont appelées à inclure des configurations compliquées impliquant des robots multiples utilisant des systèmes sensoriels. La programmation de tels systèmes est une tâche difficile et laborieuse. L'utilisation d'un simulateur procure un banc d'essai permettant de mettre au point des programmes d'application. De plus, le développement des tâches de travail devient plus securitaire puisque l'equipement réel n'est pas utilisé. Cette approche présente également l'avantage de ne pas interrompre le cycle de production pour des fins de développement de nouvelles strategies. Le simulateur nécessite généralement une interface graphique afin d'amiliorer la facilité d'utilisation et de produire un environnement de programmation plus performant. Cette thèse présente une solution au problème de la simulation des cellules de travail robotisées en le décomposant en des problèmes plus simples de base de données, language d'interface, modélisation des solides, mouvements des corps rigides en 3-D, cinématique des robots, programmation des robots et graphisme.

Acknowledgements

I would like to thank all the members of the Computer Vision and Robotics Laboratory for their constant support during the course of this work. I would like to thank my supervisor, Dr. Malowany, who provided me with guidance throughout this project. I would also like to thank Dr. Angeles for his help concerning the kinematics aspects of this work, and Dr. Hayward for providing me with useful literature.

I am very thankful to my colleagues Moshe Cohen, Bruno Blais, Baris Demir, and Clement Gosselin for their help in proofreading and correcting this manuscript. Special thanks to Mike Parker who helped me overcome difficult programming hurdles, and to Sylvain Juneau, Abdol-Reza Mansouri, Stéphane Aubry, and Iskender Paylan, whose encouragements were of great value in completing this work. I would like to acknowledge my immediate family for their continued support and understanding throughout the years. The financial support of FCAR and of the Tunisian government are also gratefully acknowledged.

Contents

List of Figures

Chapter 1 Introduction

1.1 Robotics

Robots are among the most advanced automated machines built since the industrial revolution. Many definitions have been assigned to them, but the one we think applies best is the RIA's (Robotics Institute of America): " A robot is a programmable, multifunctional manipulator designed to move material, parts, tools, or specialized devices, through variable programmed motions for the performance of a variety of tasks" [Holland83]. From this definition we can conclude that tasks are performed through a variety of motions. The basic question in robotics in general is 'how to perform motion?'. To this end, problems of architecture, kinematics, dynamics, and control arise. From the same definition we can also see the importance of programming considerations in robotics.

1.1.1 Robot Architectures

Inspired by the human body, a robot is made of many joints and links in order to be able to cover a desired workspace. One big difference between humans and robots from an architectural point of view is that parts of the human body are flexible or piecewise rigid such as the spine, which is rare even though possible for robots. This limitation is especially due to the difficulty of controlling a flexible body. Thus, most robots today are made of rigid bodies called links connected together by joints. There are many types of joints: Prismatic,

revolute, cylindrical, spherical and others. In practice, however, the prismatic and revolute joints are dominant because they form the basis for any other type of joint, and are easier to fabricate mechanically. The ability to achieve any type of motion is a positioning problem, as it should be possible to reach any point in the workspace of the manipulator. However, in order to manipulate material characterized by dimensional properties, the direction with which the manipulator approaches the material is also of interest. Hence, in order to fulfil the definition given above, the position and orientation of the hand or end effector of the manipulator are of interest. This separation of position and orientation has led to a popular class of manipulators, namely wrist partitioned manipulators. This latter class of manipulators separates to a certain extent the position from the orientation of the manipulators. Wrist partitioned manipulators are usually six dgrees of freedom (6 d.o.f), of which three degrees are needed for positioning and three for direction. In some tasks, however, only five degrees of freedom are necessary as explained by Angeles [Angeles86a]. Examples of such tasks are the ones used in manipulating objects which have one axis of symmetry, which is a rather current topic.

1.1.2 Robot Kinematics

The kinematics problem in robotics can be divided in two, forward and inverse kinematics. The forward kinematics problem is defined as follows: Given the joint values and rates, find the position, the orientation, and the speed of the end effector of the manipulator. The inverse kinematics is the reverse problem and is more complex to solve in the general case. In order to solve these problems, certain means have been standardized, such as the choice of the coordinate frames to use in order to facilitate the solution. The Hartenberg and Denavit parameters are by far the most popular definitions used to define those frames, particularly due to the uniformity of the transformations relating the frame attached to the $(i+1)^{th}$ joint to the one attached to the i^{th} joint [Paul81a].

1.1.3 Robot Dynamics

When a manipulator is moving to a particular position and orientation of its end effector, many considerations enter into account beside the position, velocity, and acceleration. Gravity has an effect on the desired motion, as does friction and other forces and torques applied to the manipulator. The problem of dynamics is complicated due to the high non-linearity of manipulator motion. The dynamical equations relate forces and torques to positions, velocities, and accelerations and are usually solved in order to obtain the equations of motion of the manipulator. Lagrangian mechanics is widely used in robotics to formulate the dynamical equations since it bypasses the physical properties of the problem. The dynamical equations are then formulated in a purely mathematical way. Generally in robotics, the forward dynamics calculations are not needed, since the desired motion is usually known. An inverse computation is needed to determine the forces and torques to apply at the joint level for control. The problem of dynamics is the motive for the development of control theory in robotics [Paul81a].

1.1.4 Robot Control

Once a trajectory is specified in terms of time-based functions defining the joint positions, rates, and accelerations, a control scheme must be developed to assure tracking of the desired trajectory. A feedback control system is needed to approximate with minimal error the desired positions, rates, or accelerations, depending on the type of control, regardless of the varied torques resulting from the robot's configuration [Craig86, Paul81a].

1.1.5 Robot Programming

In order for a robot manipulator to be of any use, means have to be provided to specify the desired configurations it is to attain. Robot languages have been introduced concurrently with the introduction of robots themselves. Although this goes back farther

than a decade, developments have been slow and robot languages have been progressed in a rather haphazard manner. The languages that are currently in use can be divided into three categories. These are manipulator level languages, object level languages, and task level languages. The manipulator level languages are the lowest level in a robotic programming environment and deal in terms of joints values or end effector coordinates for motion tasks. Object level programming is very much in use, with the idea being to define objects and their coordinates in some form of a database. The tasks would therefore be performed in terms of objects, and not directly in terms of coordinates [Faverjon86] (e.g grasp_tray, where the coordinates of the tray are defined in the database). The last category is an extension to the object level, and deals in higher terms of tasks, which are somehow interpreted into lower level commands [Sata81] (e.g assemble pump which is interpreted as a series of lower level commands such as move, grasp and others).

1.2 Why Simulation?

Many problems may arise in programming a robot or a complete workcell. For example, the programmer can ask the robot to go out of its workspace, and thus error detection becomes mandatory in every robot program. This is cumbersome and is usually not the case in practice. Robots are often placed in constrained workspaces, such as on a table, so the tasks are very dependant on the robot environment, and a program could be prefectly suitable for one environment and not at all for another. Many other problems are encountered in robotics, so as a precaution, simulation is necessary in any robotic environment. Simulation can have many uses, of which we shall name several:

1- Graphic simulation of a robotic workcell may be used as a means for teaching new entrants to the field, while safeguarding against possible damage to the robotic equipment.

2- In some instances, it may be desired to experiment with manipulators of specific architectures due to environmental or task-related constraints; this could also

be accomplished using a simulator.

3- Programming a manipulator for a particular task is an everyday problem in robotics, and instead of trying out the task on the manipulator, it can be tried on a simulated replication of the manipulator.

4- In some industries, the placement of many manipulators is very important in terms of efficiency, and the possible solutions can be tried on a simulator, thus avoiding inconvience and high cost.

5- Contructing new manipulators specialized for different tasks can also be simplified enormously by using a simulator.

6- Before making a decision such as buying a particular robot for a well defined task, a simulator can be used for comparison between competing manipulators.

There are, of course, many other considerations which justify the usefulness of a robotic workcell simulator. Some of them will be encountered in the following chapters of this thesis.

1.3 Previous Simulators

Previous work on robot off-line programming and simulation systems is abundant. As examples we mention the McAuto system [Shumaker80, Kretch82], the GRASP program [Derby82a], the MIRE system [Liegeosis et al. 80], the SIMULATOR program [Soroka80], and the SAMMIE system [Heginbotham et al. 73]. An overview of the existing systems is presented by Derby in [Derby82b], from which we concluded that most robot simulators use commercial solid modellers such as PADL [Voelcker et al.78], CATIA [Borrel et al. 83], NONAME [Staff from GMP 81], and others. However, due to the generality of those solid modellers, the robot simulation process tends to be cumbersome and slow. Moreover, the solid modellers mentioned above can not be easily integrated with a workcell

programming environment. We have therefore decided to develop a solid modeller suitable for rigid body manipulation and particularly robotics applications. Thus, the source code is provided, and flexibility is one of the prominant features of our design.

1.4 Project's Outline

Our goals in this project are listed as follows:

1- To create any workcell interactively and easily.

2- To program the workcell for different tasks to be accomplished by a variety of motions.

3- To view graphically the results of the tasks.

There are many problems which we shall introduce now and will be solved later when encoutered. The first goal is divided in two major parts, namely, a data structure for easy interactive use, and a solid modelling part which is used for modelling workcells. In any type of simulation where a variety of objects, such as tools, parts, robots and others, are to be used, certain organized data is needed in order to facilitate the storage, retrieval, and modification of any type of information related to the objects of interest. In a robotic workcell simulation, the information of interest is either descriptive or geometric in general, and thus the data structure should be capable to handle both. Moreover, this data structure changes with time due to motion, and should be dynamic. After the design of the data structure, the user needs some way to interact with it; this means of interaction is called a data sublanguage. Both the data structure and the data sublanguage are the subject of chapter 2, where they are designed and discussed in detail. The second solid modelling part of the first goal is basically an extension to the data structure and data sublanguage. The creation of a particular object can be done at the lowest level by using the data structure and the data sublanguage. However this is difficult, as will be shown

in chapter 2. Hence, some other means have been developed to facilitate the modelling of a workcell, as discussed in chapter 3. The first part of chapter 3 is an overview of the existing methods of solid modelling, while the latter part is a presentation of what was developed for the simulator and a discussion of the advantages and disadvantages of the methods in use. The second goal is the programming aspect of the workcell. This comes down to the simulation of a motion applied to one solid in 3-D space, as introduced and discussed in chapter 4. In robotics, however, the motion is applied simultaneously to many solids, namely the different links of a robot plus the grasped object, if any, and this is more difficult to solve. Moreover, the motion is usually known in cartesian space rather than joint space, and thus the inverse kinematics solution is needed. The whole notion of motion in robotics is discussed in chapter 4, and solutions are presented with means of speeding up the entire process. The last goal is obtaining and viewing results; this is presented in chapters 2, 3, and 4, whenever a result is required. The graphics aspect of the result is discussed briefly at the end of chapter 3, as a way of verifying the solid modelling. Chapter 5 presents some results in which we create a workcell, program it, and view the output of the program. Some ways of debugging a robot program will also be introduced. As we can notice from this introduction to the project's outline, the dynamics and control aspects of robotics are beyond the scope of this thesis. Thus the results are a good approximation of reality as long as the control scheme is successful enough to make the dynamical effects negligeable. The architecture, kinematics, and programming aspects in a robotics workcell are the major interest for this thesis.

Chapter 2 Database and Data Sublanguage

The design of a data-structure has become more and more important in solid modelling, and hence in any kind of graphic simulation involving solids. Depending on the application, some representations are more suitable than others; thus, a decision has to be made on the structure to be used for the world modelling. Once this is done, a data sublanguage (user interface) is developed to perform the interaction between the user and this data-structure or database. This chapter is divided in two major sections, namely, database and data sublanguage.

2.1 Database

In a simulation environment, the designer or programmer has to deal with different components to construct the world model, and there is therefore a need to develop a database which contains the descriptive and geometric information of these components. In general the components have relations between them, which should also be stored in the same database[Date81].

The need for this type of database appeared when world modelling became important in the field of robotics in general. This is due to the fact that most robots work in a known environment, and hence a dynamic database which could handle the initial model of the workspace and the changes during the manipulation would be very helpful. This concept of dynamic database found use in fields other than robotics and vision such

as in flight simulators. The reasons mentioned above, and some other research results, led us to believe that a good design of a datamodel (DM), the user's view of a database [Date81], would be very helpful for the simulator.

In order to make this good design, we have to define the criteria which make an adequate datamodel. One of these criteria is that the datamodel should provide ways to retrieve or derive all the required characteristics of a selected object, in an acceptable amount of time. For engineering applications it has been noted that the objects to be manipulated are fairly complicated [Dittrich85, Haskin82]. This imposes an additional criterion of simplicity for representing these objects in the datamodel. The third criterion which defines a good database design is its compactness, which we should keep in mind while we are developing the software for the datamodel. Looking at what has been done before, we can notice that there are three traditional favored design approaches to represent entities and their relationships, namely, the relational approach, the hierarchical approach and the network approach [Date81].

We shall investigate these approaches one by one, before building our design. The best way to discuss these approaches is, probably, to consider an example. Suppose we are asked to represent some typical robot manipulators as part of an assembly chain; one of them could be a 6 degree of freedom with 6 revolute pairs. The first moving link is geometrically cylindrical, the second and third links could be constructed easily out of surfaces connected together to form a certain volume (volume2 and volume3), and the last three links constitute a spherical joint and are geometrically represented with one sphere. In other words we can consider this particular manipulator as being made of four different solids, a cylinder, volume2 and volume3 and a sphere. This is geometrically similar to a Puma. Let us suppose that we have three manipulators, and like the one described above, they are made of many links decomposable geometrically in smaller entities such as cylinders, spheres, and blocks, as shown in figure 2.1.

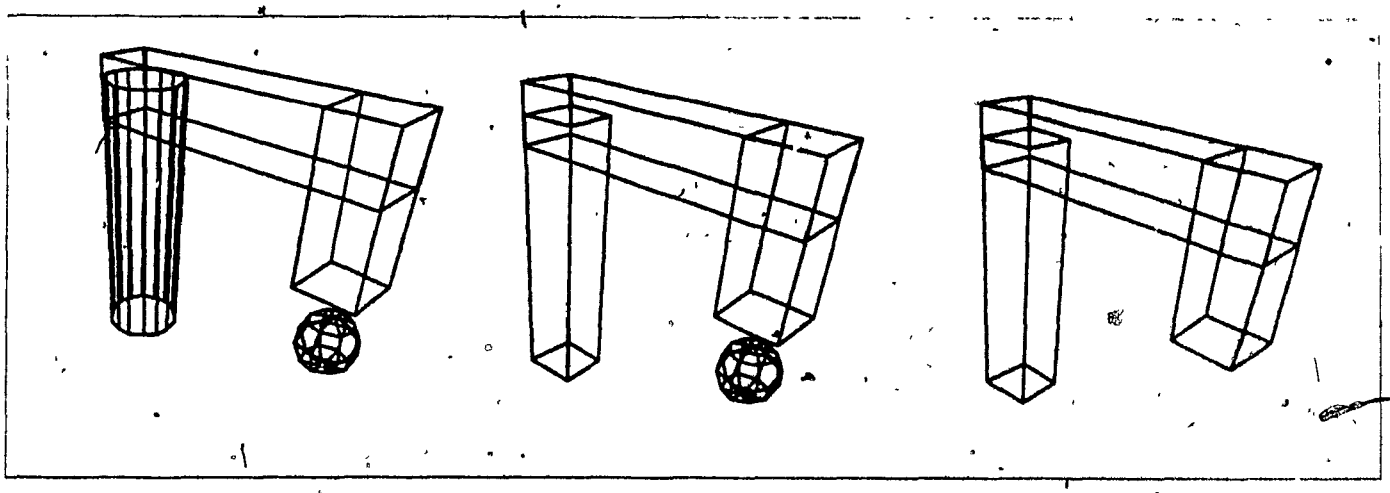


Figure 2.1 Robot Manipulators.

2.1.1 The Relational Approach

As mentioned before, a database system must be able to represent entities and relations between them [Freedman86]. In the relational approach, there is no explicit distinction between the entities and the relations [Ullman82]. Considering our example of robots and the solids which construct them, we can define two relations: ROBOT and SOLID. These two relations are represented using a table, which is the typical way of representing a model by the relational approach [Astrahan76], this is shown in figure 2.2

ROBOT

Serial number	Solid_1	Solid_2	Solid_3
1	1	2	1
2	0	3	1
3	0	3	0

SOLID

Serial number	Surfaces	Cylinders	Spheres
1	0	1	0
2	6	0	0
3	0	0	1

Figure 2.2 The relational approach.

To determine the components which make up a given solid, the appropriate solid is located using the serial number, and then we read across its row in the table. To determine which solid contains a given component or a certain quantity of a given component, we locate first the component's column and then read across to the serial number. Since robots are constructed from solids and solids from components, an extra level of searching is required to know what components (spheres, cylinders, surfaces) make one particular robot. This extra level of searching is also needed for the reverse operation of finding which robot contains a given component. More complex queries could also be answered, for example, which solid contains components '1' AND '2', '1' OR '3' and so on; in fact, any type of query based on the relational algebra can easily be answered, and hence the name relational database [Codd72].

2.1.2 The Hierarchical Approach

In the hierarchical approach, relationships are entirely implicit, and the relationship between two entities is presented by the relative position of the two nodes defining the entities [Date81, Ullman82]. Continuing with our example, the representation in this approach would be as shown in figure 2.3.

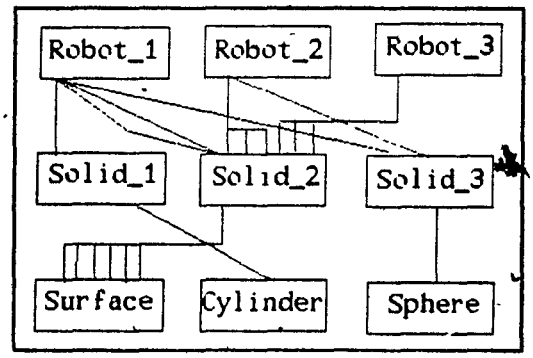


Figure 2.3 The hierarchical approach.

In this approach, responding to a query is necessarely divided in two distinct operations: First, traverse the whole hierarchy to find the entity on which the query is to be applied, second, apply the query [Kunwoo85]. As an example, the query could be 'what

are the components of solid_1', in order to answer this, the solid 'solid_1' would have to be located, and then all the nodes attached to this entity's node would constitute the answer.

2.1.3 The Network Approach

The network datamodel allows for more flexible associations by replacing the hierarchical structure with a network [Freedman86]. In this approach, the nodes represent the tuples of data, and the arcs represent the relations. Using the example of robots and solids, the structure in this approach would be similar to the one shown in figure 2.4.

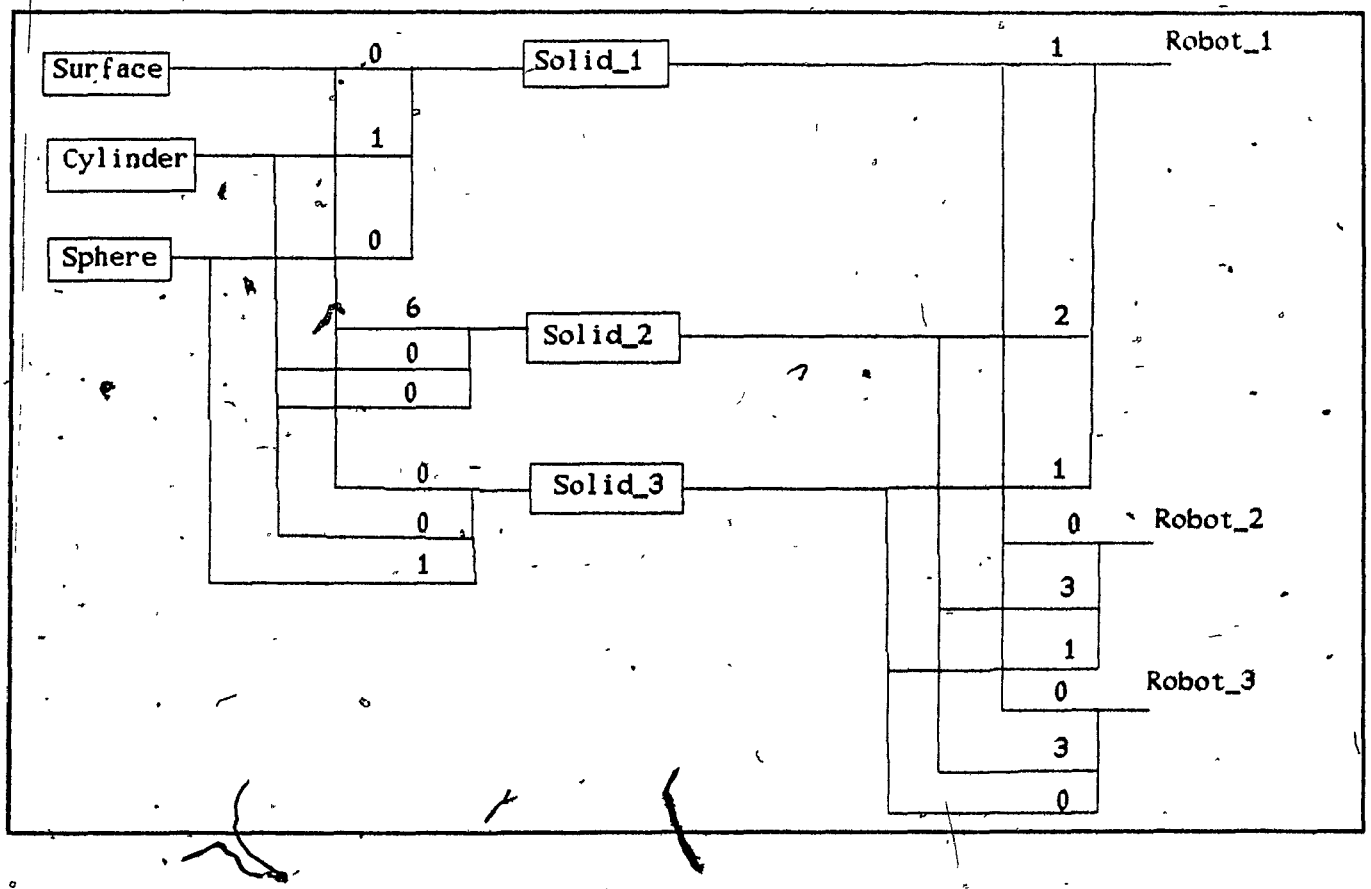


Figure 2.4 The network approach.

All relations in a network datamodel are explicit, and thus the positions of the nodes are irrelevant. Answering a query means searching the network for all possible paths to apply the operation. Some special care should ba taken if the arcs form a chain [Date81].

2.1.4 Approach Selected

We have presented three known datamodel approaches, none of which is really perfect In this paragraph, their respective drawbacks will be shown. An attempt to construct a datamodel avoiding these drawbacks will be made for this simulator study.

The relational approach shows its weakness, when considering engineering applications, where the representation of complex entitities and relations is needed, a large amount of relations must then be kept [Fenves85]. The ability to represent these complex objects requires a hierarchical approach, but, in its turn, the hierarchical approach has other disavantages. Since the relationships are implicit, the only way the user can interact with the information contained in a relationship is by building it up dynamically while traversing the hierarchical tree, [Date81]. For example, to know if a sphere A is part of one solid S, we should locate the solid and then fetch all the components which build it. In the network approach, on the other hand, the relations are explicit and are present by means of pointers. However, the relations and the entities are considered as different objects and are stored differently.

The fact that the relational datamodel is not suitable for representing the variety of objects we meet in engineering applications does not make it a bad representation, since it has advantages such as the explicity of the relations. The network and hierarchical approaches both have positive points as well, namely the simplicity of representing complex objects. In our design we aimed at using the advantages of all these approaches. The network approach is, in practice, very similar to the hierarchical one, and so there is no advantage in considering it seperately, especially since it introduces the chain complexity.

The hierarchical datamodel is very suitable for engineering applications, however its biggest disadvantage is the implicity of the relations. If we make these relations explicit by giving them identities and by supplying means to access them, the problem would be partly solved. The second disadvantage of a hierarchical approach is the fact that for each query we have to traverse the whole hierarchy to find the entity or relation (once made

explicit) on which the query is to be applied. This requires further optimization, such as allowing the queries to be specified at intermediate levels of the hierarchy rather than just at the top. With the two changes mentioned above, we avoid the disadvantages of the hierarchical datamodel.

To recapitulate, we can say that our representation would basically be a hierarchical datamodel with explicit relations and a user interface allowing access to every level of explicit relations or entities.

2.1.5 World Representation

The term world representation is used to signify the datamodel approach we used for world modelling. The chosen datamodel is made of entities and relations as described earlier. In practice, the entities include xy_pairs (XY), surface_contours (SCT), surfaces (S), blocks (B), objects (O) and scenes (SC). The explicit relations are basically homogeneous transformations with identifications, these include transform_surfaces (TS), transform_blocks (TB) and transform_objects (TO). Thus, we have six different level of entities and three levels of explicit relations. Every entity from the level block up has explicit relations attached to it. All the entities at the same level are stored as linked lists bounded by a begin_list and an end_list pointer. The relations at the same level are also stored as linked lists, and differ from the storage of entities in that we can have many linked lists at the same level of relations defined as transformations, these linked lists are also bounded by begin and end marks, this is shown in figure 2.5.

Let us now start with an xy_pair and build up a typical model. The xy_pair is a structure determining the coordinates of one point in a two dimensional space. Many xy_pairs constitute a surface_contour, and a group of surface_contours make up a surface. The feature that a surface could be made of many surface_contours is provided to allow holes in a surface. A surface is then made of the union of different sets of surface_contours referred to as main_countours and hole_contours, which are stored as linked lists. Up

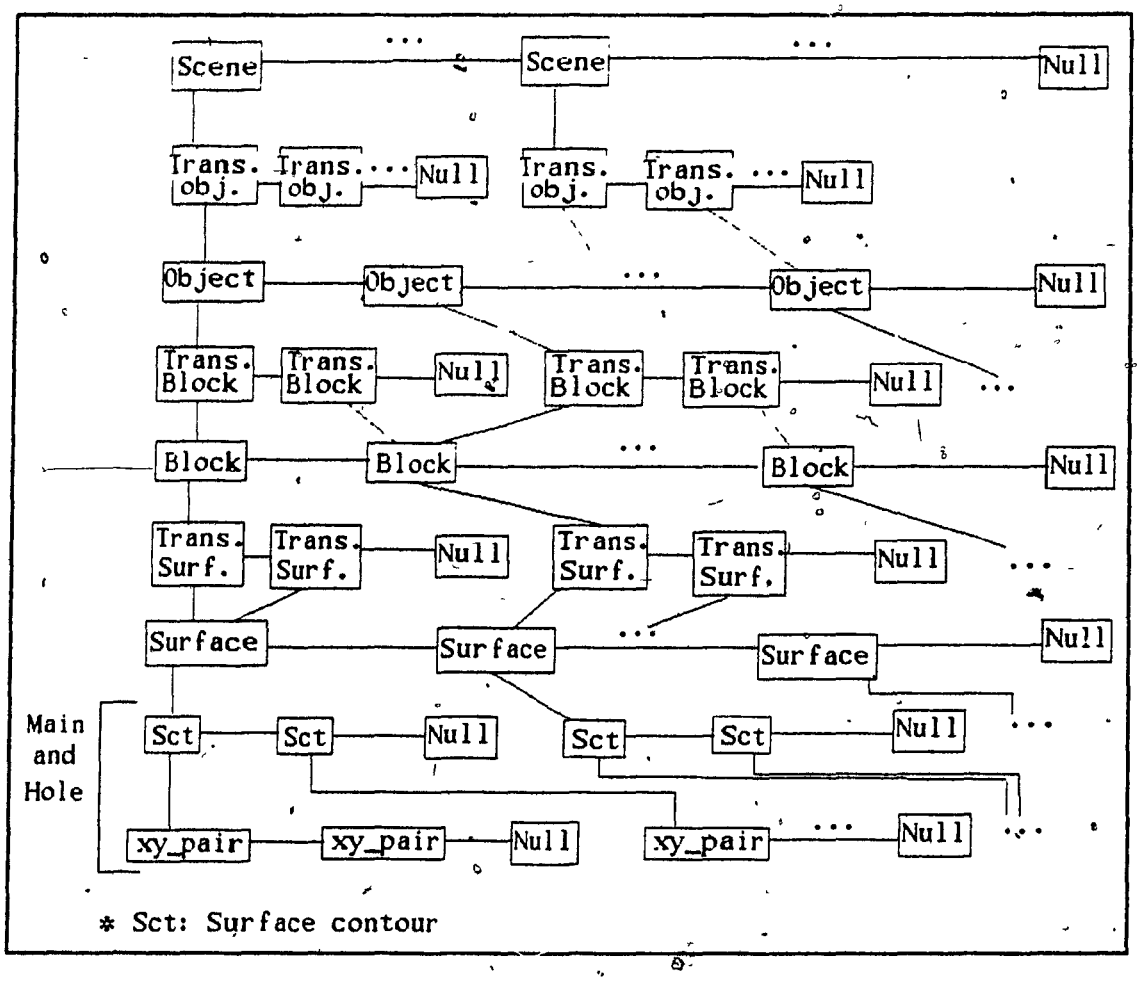


Figure 2.5 Database architecture.

to the surface level, we deal with two dimensional representations. An xy_pair is part of a surface_contour, which in turn is part of a surface. Both of the aforementioned relations, namely "part of" relations are stored implicitly. We can not therefore access them directly. This is done to make us think of a surface as being the lowest entity that we can manipulate in the hierarchy. Manipulation or query of surface_contours or xy_pairs is accomplished indirectly through the surface to which they belong. The next level above surfaces is a level of explicit relations, namely transform_surfaces. Every transform_surface has an "id", a homgeneous transformation and points to a surface entity. This level of relations is identified and explicit, and thus may accept queries. The transform_surfaces are grouped together, ending with an end_mark of a null pointer, to build a higher level entity, the block. A block is made by applying the transform_surfaces in question on the

surfaces to which they are pointing; the transformations are specified in a three dimensional space and thus a block is the first 3-D entity. The next level above the blocks is made of transform_blocks; again these relations are explicit. Without going through the entire model, we can say that this structure is propagated up to the highest entity level called, a scene. Adding other levels is unecessary, since the data sublanguage offers facilities to copy from the same entity level, with an option of specifying a relation while copying. However, if deemed necessary, facilities to go to higher levels are easy to implement. Theoretically, this approach is capable of representing complicated solids. In reality, some practical considerations enter into account. For example, is it acceptable to ask the user to enter a homogeneous transformation every time he wants to attach a lower level entity to higher level one. The answer to this question is obviously no, and therefore we introduced other features. For example, specifying the attachement transformation is achieved from a composition of scalings, rotations and translations with respect to some coordinate system. The rotations in particular could be specified in different ways such as using the euler angles, using 3 points of a rigid solid to describe the start and end position and orientation or using an axis of rotation, a point of the axis and an angle; of course a rotation could be specified in other ways as well.

Looking at the datamodel, we can notice that there are no storage reserved for templates or instances at any level; this is not explicit in the datamodel, but is implemented in the data sublanguage using the copy facility. A drawback to this datamodel is the fact that one particular entity could be shared by many higher level entities [Cardenas79], and thus any changes applied to it may affect entities that we did not want to affect. This is a characteristic of any model based on a hierarchical approach, this risk could, of course, be avoided by replicating information, however this is space consuming. The type of change that could cause the most damage is certainly the deletion, that is why, we included a descriptive field called "pointed_by" in each entity. This field gives information about the number of relations that are pointing to the entity in question. Deletion takes place only if the "pointed_by" field is zero, otherwise, it is refused.

The datamodel we have chosen has a good characteristic of a relational model, in that all essential relations are explicit and can therefore be "queried". It also embodies an advantage of the hierarchical model, namely the hierarchy of building solids. One major difference between our datamodel and the relational model is the fact that the relations (transformations) and the entities are different, but, accessed similarly. The major difference with the hierarchical approach is that a query would not have to traverse the whole hierarchy but is, instead, applied at one level, and the searching or traversing is achieved horizontally through linked lists, of course there is also the difference caused by the explicity of relations.

2.2 Data Sublanguage

The data sublanguage (DSL) is the user's language to interact with the datamodel. Practically the data sublanguage has been limited to three types of queries which are addition, deletion and update [Date81]. To facilitate interaction with the datamodel, some other features such as copying and showing were added. Moreover, in order to speed up the editing phase of the datamodel and in order to avoid problems such as typing an incorrect user command, the user interface is managed by a command line parser and syntatic checker called a "key tree matcher".

A good data sublanguage is one which best accomodates the use of the datamodel. In order to accomplish this, in view of the hierarchical aspect of our datamodel, the user interface is implemented at every level of entities or explicit relations. Furthermore, the data sublanguage is implemented so that the explicit relations and the entities are accessed in a similar manner. The next paragraph will explain in greater detail the implementation of the data sublanguage.

2.2.1 data sublanguage Implementation

The facilities offered by the data sublanguage are all accessible to the user via the key tree matcher; the basic and most useful operations are creation, removal, modification,

copying, showing, attachment, detachment and model filing. The tools that were developed to allow use of these facilities include memory allocation facilities, searching and insertion of entities and relations. The identification of an object is done using its name and the level to which it belongs, and thus all queries will require at least those two specifications, in order to be applied. Let us now present the facilities offered by the data sublanguage trying as much as possible to omit programming details.

2.2.1.1 Creation

The creation process applies only to entities and takes a level specification and a name. First the specified level is verified to assure it does not already contain an entity of the same name. The memory allocation, if needed, and insertion in the entity list then take place. The flowchart for this query is shown in figure 2.6.

2.2.1.2 Removal

This process also only applies to entities and takes a level and name specification. The entity to be removed is first located at the level specified. If the entity is not being pointed to by any higher level relation, it is unlinked, and the memory previously allocated for it is saved in a garbage collection for the level in question; otherwise the query is refused. A flow chart is shown in figure 2.7.

2.2.1.3 Modification

This query applies both to entities and relations. In the case of entities it needs a name and level specification. If the search is successful the query enters a smaller key tree matcher, in order to make the modification more user friendly; for example we can see the possible kinds of allowed changes, such as changing the type , the name and other characteristics. The modification can also be applied at a level of relations. Every relation is identified by its name, its level, and the name of the entity to which it is attached. Modification of relations operates in basically the same manner as modification of entities.

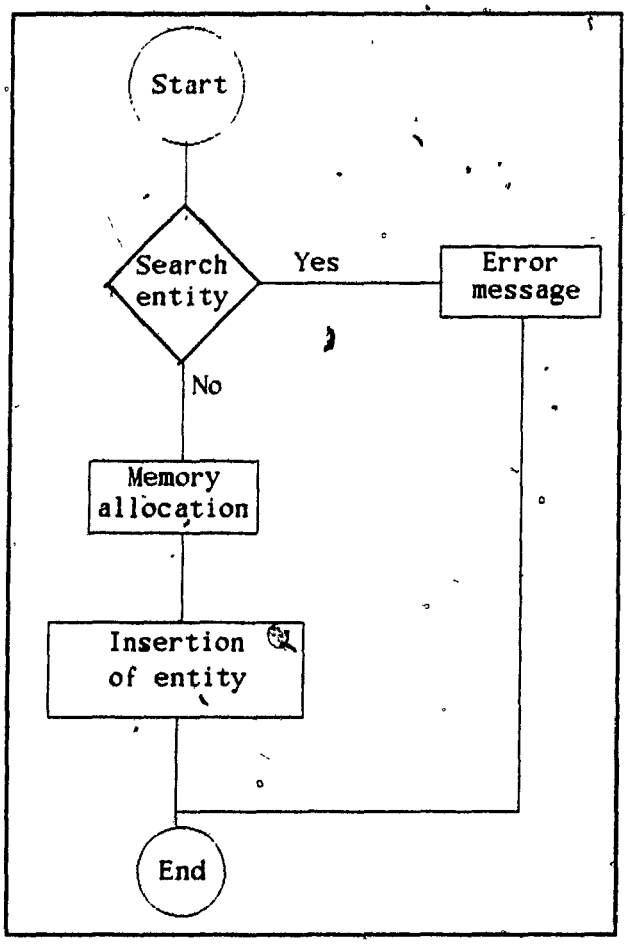


Figure 2.6 The creation process.

In both cases, if the results of the search are negative, the user is notified and the action is refused. The control then goes back to a higher level database manager (DBM) i.e key tree matcher, and waits for new commands.

2.2.1.4 Copying

This facility is used in order to make use of instances of entities at every level. It takes as arguments a level specification, a name of an existing entity at that level and a new name for the desired entity. The utility verifies certain parameters, and copies the old entity to the new one, with an option of specifying a relation to be applied. The creation of the new entity, interrogates the garbage collection to determine if there is enough memory to accomodate the new entity; if the space is found to be insufficient , new memory is

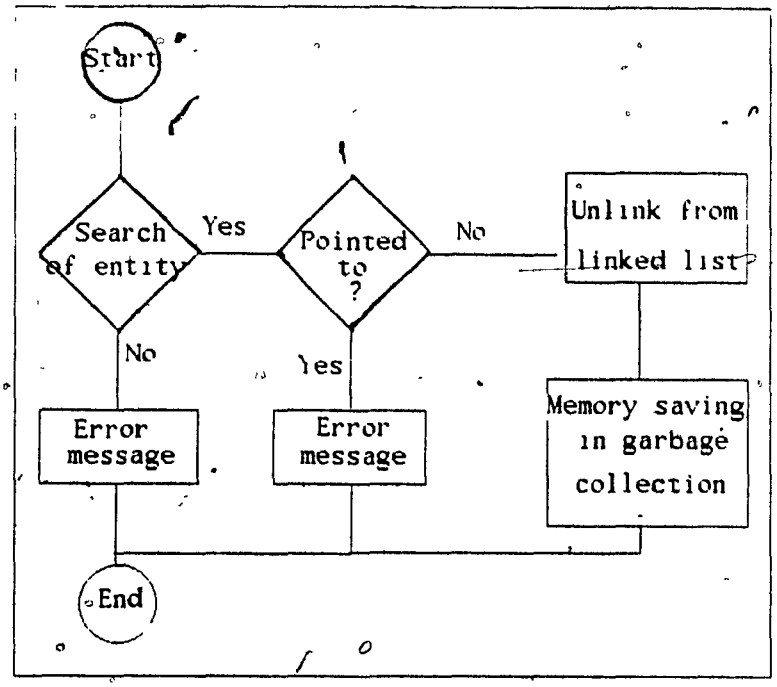


Figure 2.7 The removal process

allocated.

2.2.1.5 Showing

Showing is a query that may be applied to both entities and relations. This query provides all of the possible information about an object. Different versions of showing have been implemented depending on the amount of detail wanted about a particular object.

2.2.1.6 Attachment

This query installs a relation between two entities. It takes as arguments the name of the relation, the level and name of the entity that this relation is to be attached to, and the name of the entity that the relation should point to. It first verifies the existence of both entities and the non existence of a relation with the same name at the level specified Then memory allocation, if needed, and insertion occur and the "pointed_by" field of the entity being pointed to by this relation is incremented. The relation (transformation) could be specified in different ways, depending on the ability of the user. For more details we

provide example 1 at the end of this chapter.

2.2.1.7 Detachment

Detachment removes a relation between two entities. After all the necessary searching and verifications are performed, a specified relation is unlinked from the linked list attached to a particular entity. The space that the relation was occupying is saved in a garbage collection at the same level of relations. The "pointed_by" field of the entity that was pointed to by the relation in question, is decremented.

2.2.1.8 Model Filing

For model filing we have two basic facilities, namely, save and load. These two facilities are used so that we would not have to re-create any object (relation or entity) once it has been created in the datamodel.

In the preceeding sections we have discussed the most useful queries in our datamodel. This user interface was developed keeping ease of use and compactness as primary goals, to allow convenient use of the datamodel. The user interface is written in "C" and could reside on any UNIX machine; it is actually implemented on a SUN running the UNIX 4.3 BSD. The execution time of the queries is not very critical since the development of the model is done at the editing phase of the database . We provide two simple examples here to clarify the datamodel and its data sublanguage.

2.2.2 Datamodel Examples

2.2.2.1 Example 1: A Cube

This example is about constructing a unit cube as a block in the datamodel. First we should construct a unit square, this is done by invoking the creation process at the surface level where we create a surface called "square1", this surface has just one

main contour, without holes. Then, we invoke the creation command at the block level for "cube1". Once this is done the attachment takes place, we attach the surface "square1" to block "block1" six times, using six transform_surfaces, each transform_surface is specified using the following scaling (S), rotating (R) and translating (T) factors:

$$S_x = 1, S_y = 1, S_z = 1, R_x = 90, R_y = 0, R_z = 0, T_x = 0, T_y = 0, T_z = 0$$

$$S_x = 1, S_y = 1, S_z = 1, R_x = 90, R_y = 0, R_z = 0, T_x = 0, T_y = 1, T_z = 0$$

$$S_x = 1, S_y = 1, S_z = 1, R_x = 0, R_y = -90, R_z = 0, T_x = 0, T_y = 0, T_z = 0$$

$$S_x = 1, S_y = 1, S_z = 1, R_x = 0, R_y = -90, R_z = 0, T_x = 1, T_y = 0, T_z = 0$$

$$S_x = 1, S_y = 1, S_z = 1, R_x = 0, R_y = 0, R_z = 0, T_x = 0, T_y = 0, T_z = 0$$

$$S_x = 1, S_y = 1, S_z = 1, R_x = 0, R_y = 0, R_z = 0, T_x = 0, T_y = 0, T_z = 1$$

If our datamodel was empty before this operation, then as a result we now have one block pointing six times to one surface.

2.2.2.2 Example 2: A Parallelepiped

This example is about how to construct a parallelepiped, using what has been done in the previous example. Let us suppose we want to construct a parallelepiped of length 2, 3, and 1.5 along the x, y, and z axes. One way of doing so is by creating an object called "parallelepiped1", and attaching the block "cube1" to it by specifying a transform_block which has the proper scaling factors. Another way would be by using the block "cube1" as an instance and copy it at the block level to form another block for the parallelepiped, while copying we have to specify the scaling factors properly.

2.3 Summary

For solids such as cubes, pyramids, parallelepipeds and others it is acceptable to ask the user of the system to construct them by specifying transformations in 3-D applied on 2-D surfaces. However, constructing such solids as spheres, cones, and ellipsoids is an extremely difficult task which is practically impossible to perform in the above manner

Thus some further extensions are required, which is why we investigated the solid modelling techniques. The next chapter will introduce those techniques, discuss them and present what we have developed in this simulator for the purpose of solid modelling.

Chapter 3 Solid Modelling

3.1 Solid Modelling Techniques

A solid modelling system is defined by the following four properties, as stated by Requicha [Requicha80]:

1- Data structures which represent solids.

2- Input facilities for creating, removing, and modifying the representation of solids.

3- Output facilities for the results of the representations.

4- Facilities for answering geometric questions.

In the previous chapter, we developed the datamodel to accomodate the representation of solids. The data sublanguage was developed to fulfil the second property of a solid modelling system. The third and fourth properties will be investigated at the end of this chapter in two sections devoted to the graphics and geometric properties. In this section, we develop means of representing solids which are more suitable to our needs than those demonstrated in the examples of chapter 2. In order to do so, some traditional representations will be presented and discussed, and then a representation will be chosen and developed in greater detail.

3.1.1 Sweep Representations

Sweep representations are by far the most mathematically structured schemes for representing solids [Hayward86]. Intuitively the idea is easy to understand, it is based on the fact that by moving a point along a certain trajectory we create a curve, by moving a curve we create a surface and by moving a surface we create a solid. Sweep representations are very practical for modelling constant cross-section solids as demonstrated by Lossing [Lossing74], and they also proved to be suitable for detecting collision between solids in a workcell, where a moving solid S_1 collides with a fixed solid S_2 if the volume swept by S_1 while moving intersects that of S_2 [Boyse79]. These representations have also been used in material removal applications, in which, the material found in the intersection of the volume swept by the tool while moving and a fixed solid is removed [Voelcker74, Voelcker77]. These representations have been successful because they are well defined mathematically. The entity used for the motion is usually called the **generator** and the trajectory to be followed is the **director**. Once the director is defined, a generator is chosen and swept along to represent the desired solid. The most common curves used to fully describe the director are the PD curves (P for position and D for direction). These curves are a general form of six component curves, in which the first three components define a continuous parametric equation of position that associates a point $P_i = \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$ on the curve; the last three components define a continuous direction equation that associates a direction vector $\mathbf{d}_i$. At each point of the curve, assuming that the parameter is u, we can then define a frame as follows :

$$\mathbf{t}_i = \frac{dP_i/du}{\|dP_i/du\|} \tag{3.1a}$$

$$\mathbf{n}_i = \frac{\mathbf{d}_i \times \mathbf{t}_i}{\|\mathbf{d}_i \times \mathbf{t}_i\|} \tag{3.1b}$$

$$\mathbf{o}_i = \mathbf{t}_i \times \mathbf{n}_i \tag{3.1c}$$

A PD curve is shown in figure 3.1.

Other extensions could be added to the PD curves such as scaling factors which would then make them 9 components curves. Figure 3.2 shows examples of various PD

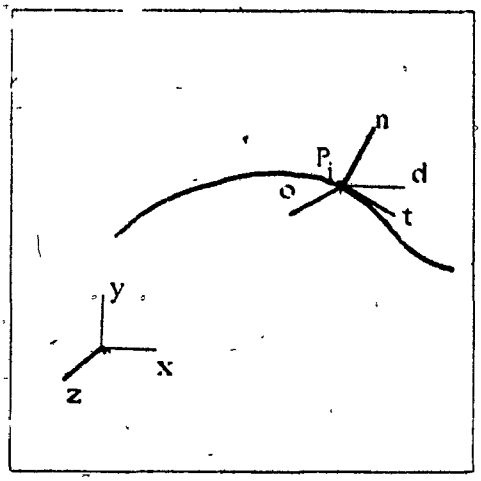


Figure 3.1 PD curves.

curves. The first one (fig. 3.2a) is just a translational sweep of a given generator following a given straight director. Figure 3.2b uses the same generator following a different director, where we can notice the twisting. Figure 3.2c shows the scaling effect using the same generator and director as in figure 3.2b, this last curve is described by 9 components.

In practice, the most common techniques in the sweep representations are the translational and rotational sweep [Requicha80]. For example, the translational sweep could be used to model a cube starting with a square surface generator, and the rotational sweep could be used to model a cone given a rectangular triangle generator. The sweep approach is prefered when dealing with flexible solids instead of rigid bodies due to the structural way of adding the scaling factors in the PD curves. In the current work, the translational and rotational sweep have been implemented and will be explained in greater mathematical detail later.

3.1.2 Constructive Solid Geometry

Constructive solid geometry is another representation approach for solid modelling. This representation uses three entities in order to model a given solid: primitives, regularized boolean operators and rigid body transformations [Requicha77]. The constructive solid geometry (CSG) techniques could be modelled by binary trees: the nontermi-

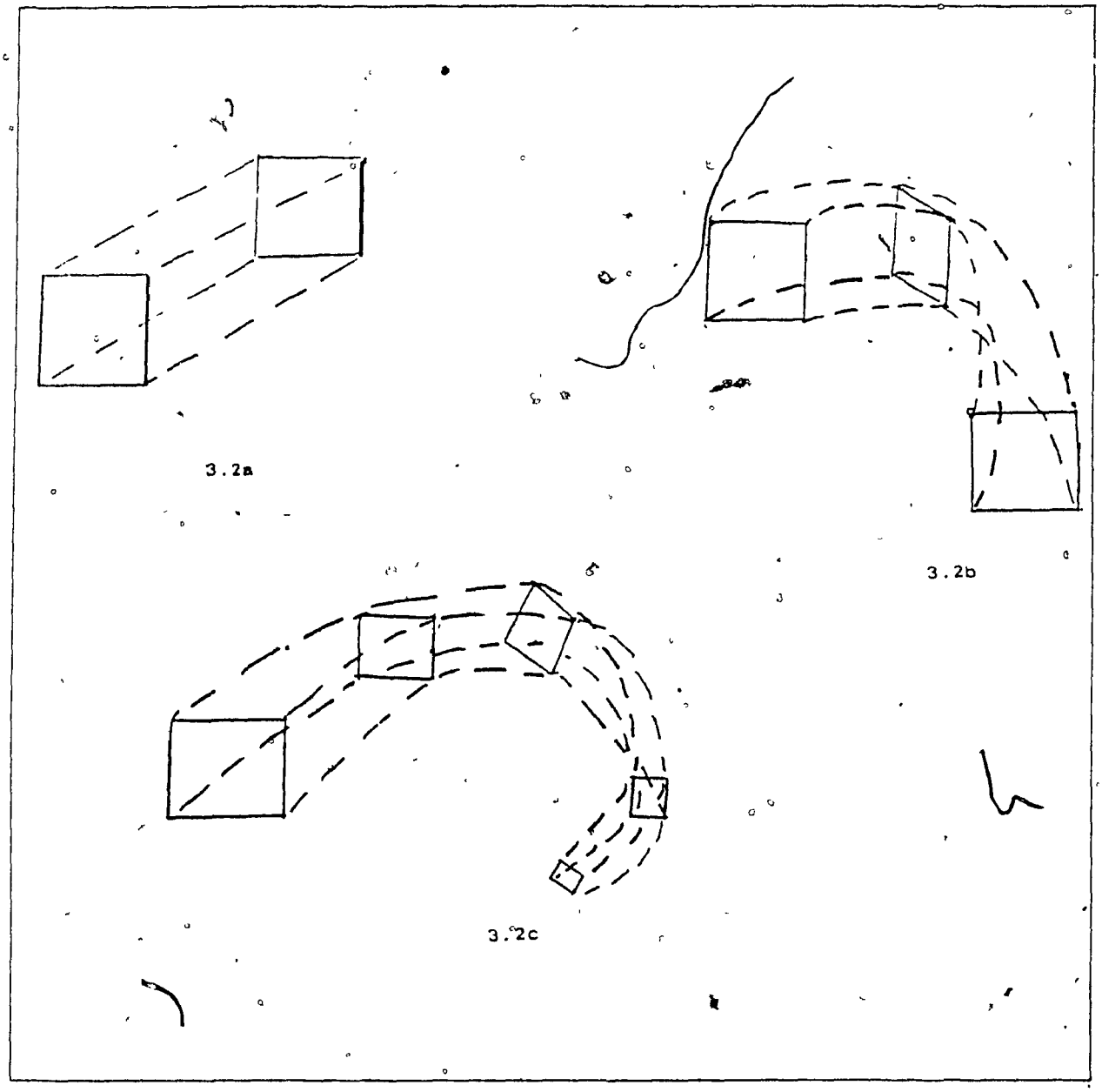


Figure 3.2 Sweep Representations.

nal nodes represent either rigid motions (translation, and rotation) or regularized boolean operators (in order to ensure the non construction of dangling deges or surfaces): the terminal nodes are either primitives or arguments defining the rigid body transformations [Requicha80], this is shown in figure 3.3.

The basic boolean operators being used are the regularized union, intersection, and difference. The CSG representations are very attractive in many ways. Primarily, the

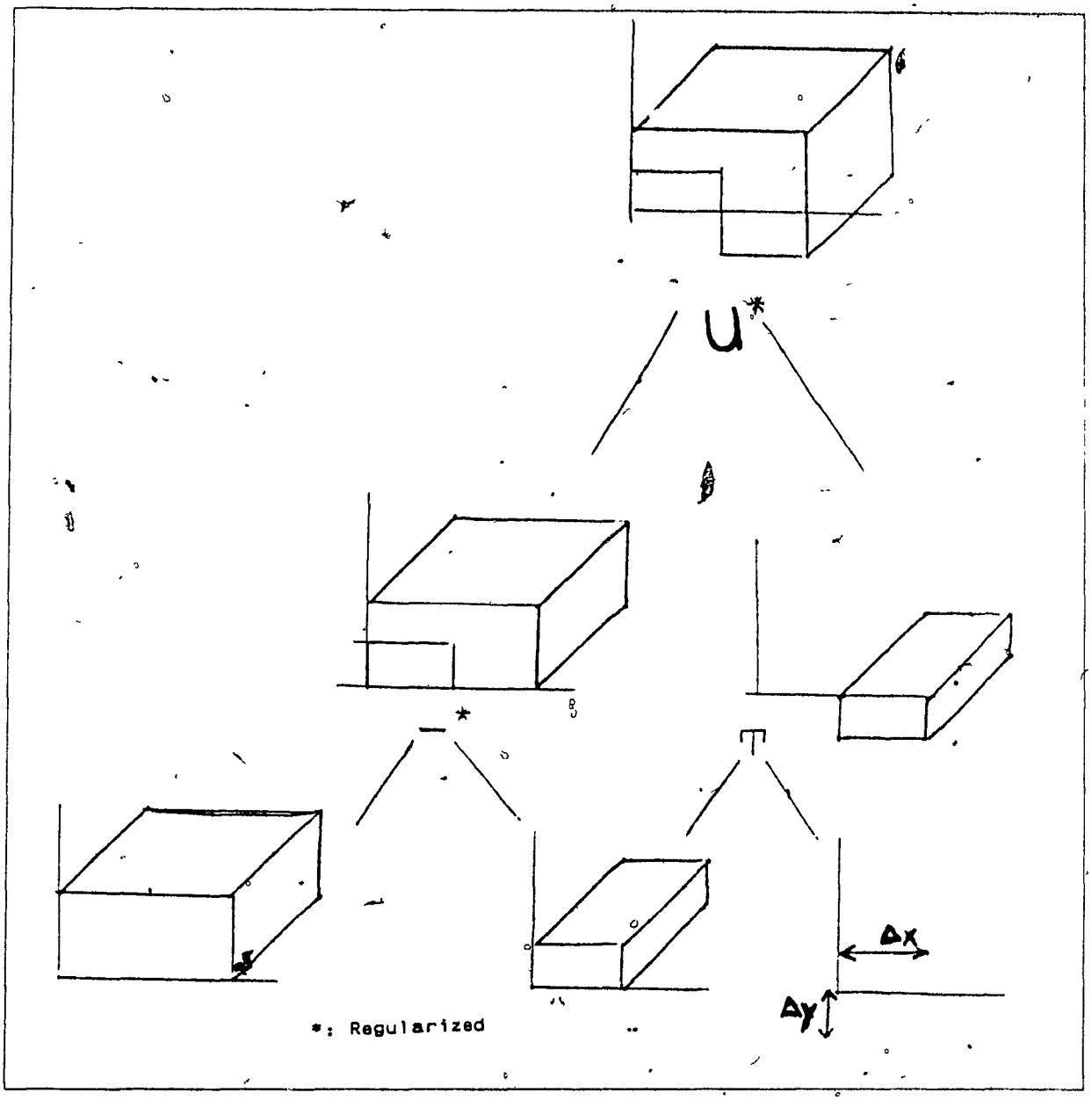


Figure 3.3 CSG Representations

problem of representing a particular solid could be stated rigorously through the use of boolean operators. Another advantage of the CSG is that if the primitives are valid and bounded and if the boolean operators are regularized then the resulting solids are valid and bounded, this is verified in PADL [Voelcker78]. The most common primitives are the cube, the cylinder, the sphere, the cone and the torus. In general, however, practical systems offer facilities to create quasi-primitives using techniques such as sweep representations.

and it is the user's responsibility to verify their validity. The primitives which are part of a CSG modelling system are usually represented using half-spaces. The half-spaces are defined as two unbounded regions of a cartesian space divided by an unbounded surface. As an example, a cylinder is represented by four half-spaces; two of them cylindrical and obeying the relation $0 \leq x^2 + y^2 \leq r^2$ and two planar half-spaces obeying the relation $a \leq z \leq b$, as shown in figure 3.4.

In CSG representations, as in sweep representations, the rigid body transformations could be extended to include flexible bodies, by adding scaling factors. Furthermore it is possible to add other kinds of transformations such as symmetric transformations, however, this is limited by the boolean algorithms used in the CSG. The CSG also has disadvantages such as the complexity in computing the solid's boundaries. This problem is called CSG to Boundary representations conversion, and is very useful if we plan to detect interference between solids.

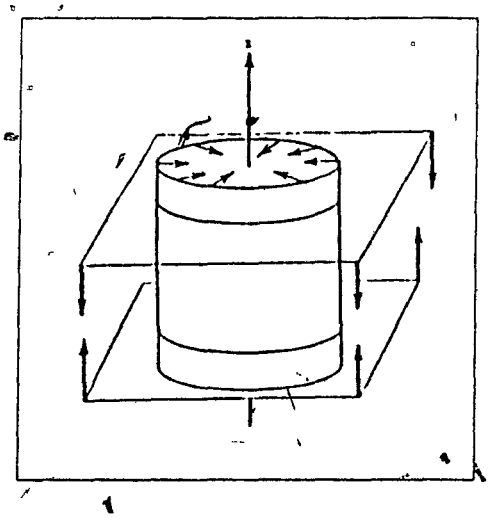


Figure 3.4 Half Spaces

3.1.3 Boundary Representations

The boundary of a solid is a closed region in 3-D which seperates the inside from the outside of the solid. The region has to be well formed which means closed, orientable, non-self-intersecting, bounded and connected [Mortenson85]. The bounding

surface is composed of the union of faces composed themselves of edges and vertices. The edges could be interpreted as bounded intersections of half-spaces. Then we can define a solid as:

$$S = \bigcup_{i=1}^{n} \bigcap_{j=1}^{m} \text{half-spaces} \tag{3.2}$$

The faces or patches (2-D) of an object must satisfy the following conditions as stated by Requicha [Mortenson85] :

1- A finite number of faces defines the boundary of a solid.

2- A face of a solid is a subset of the solid's boundary.

3- The union of all faces of a solid defines its boundary.

4- A face is itself a subset or limited region of some more extensive surface.

5- A face must have a finite area and be dimensionally homogeneous.

The five conditions mentioned above guarantee the unambiguity of the faces and thus the unambiguity of the solid made of these faces. Boundary representations (B-reps) are not unique, since the faces could be chosen arbitrarily as long as they cover the solid completely without overlapping. This non uniqueness is demonstrated in figure 3.5.

Boundary representations are suitable for representing complex solids. Let us suppose we want to represent a certain solid S, this task could be decomposed recursively by stating S as the union of two less complicated solids A and B, $S = A \cup B$ as found in [Barnhill74]. This scheme is further decomposed until we reach subsolids that we already have created, or we can create. The problem is then to define the bounding surface of S as a function of the bounding surfaces of the final subsolids. Some algorithms have been developed in order to solve this problem [Baumgart74]. A typical boundary representation could be modelled by a tree, not necessarily binary, as shown in figure 3.6. Boundary representations have found a lot of success in applications involving computer graphics.

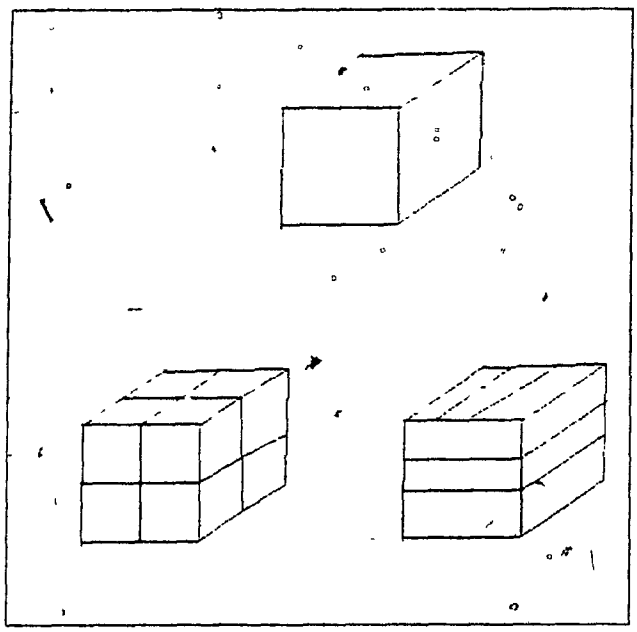


Figure 3.5 Non Uniqueness of B-reps

This is due to the fact that they are easily handled for graphics manipulations because every solid, at the lowest level, is represented as a list of vertices which are suitable for displaying. Boundary representations have also been extensively used for robotics applications where collision detection is a major aspect. Lately some further development in mechanics have rendered these representations suitable for representing assemblies. This is due to the simplicity with which the volume, first and second moments of inertia could be calculated using line integrals, since lines are part of the boundary representations structure. The methods in order to do so have been developed by many as explained in [Lee82a, Lee82b, Woodwark82]. However, many unresolved problems remain in these schemes such as the validity of a given representation: is a representation closed, oriented, dimensionally homogeneous, and so on?. Boundary representations have been used in this simulator, for reasons that will be mentioned later.

3.1.4 Cell Decomposition and Spatial Occupancy Enumeration

Cell decomposition is the most natural way to represent solids. The idea is simple: Starting with a complex solid, decompose it into different pieces called subsolids

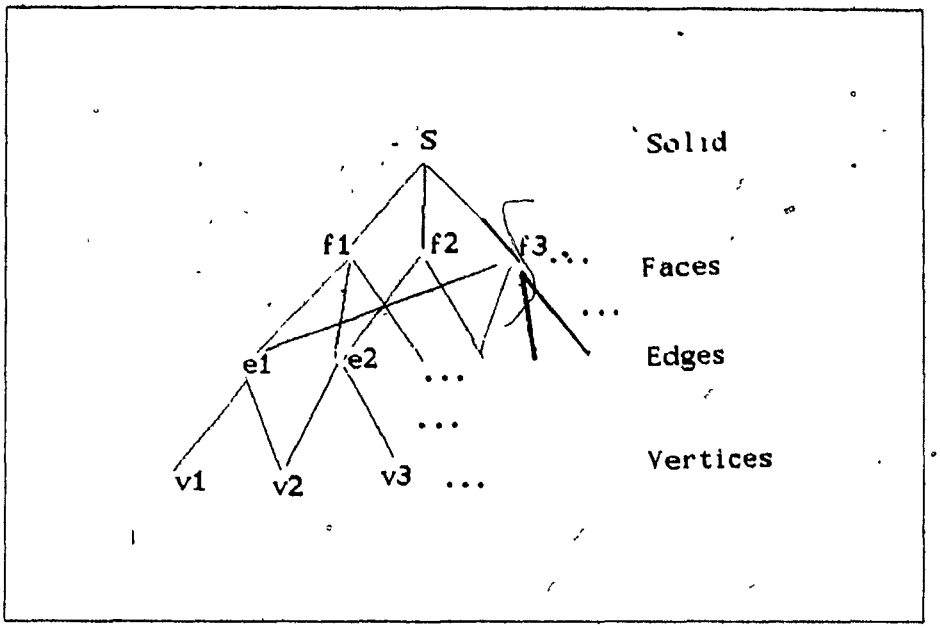


Figure 3.6 B-reps tree

or cells, where each piece should be easier to represent than its own parent. If necessary, we continue to decompose the subsolids until we reach a set of subsolids which we have the tools to represent. The solid we started with, is then the union of all the decomposed subsolids or cells. This representation approach is, of course, non unique but unambiguous. The cell decomposition schemes could be considered as particular cases of CSG schemes, where the only operator used is the "**glue**" operator, which is a restricted union operator. The "**glue**" operator could be defined as the union operator between two cells meeting exactly at a common face, edge or vertex. In the cell decomposition techniques, the final cells need not to be identical. A particular case of cell decomposition is spatial occupancy enumeration, where the final cells are of the same shape, usually cubical. Some other schemes have appeared where the final cells are spherical for example as in [O'roorke79]. In our discussion we will use cubical final cells, and the characteristics of the cubical example should be applicable to other types of final cells. Usually the final cells (cubes), of predefined size, are placed on a grid in a three dimensional space. Representing a solid means placing it on the grid and filling it with these cubes. These representations have two major advantages compared to the general cell decomposition techniques, of which the first is the simplicity with which we can access a given point in space, and the second

is the uniqueness of the representation [Baer79]. However, there are disadvantages, of which the biggest is probably the amount of storage needed for these representations, especially as the amount of storage increases considerably with the predefined resolution of the representation. The smaller the final cell is, the bigger the resolution. The coding of these representations is done the following way: Any given cell is either empty (0) or contains part of a solid (1). This coding is very redundant [Redd78], because there is a high probability of finding large streams of 1's or 0's before finding any change. The changes are just around the boundary of the solid. In order to avoid this redundancy, and at the same time save memory, another technique appeared and was applied primarily in 2-D. It is called the **quadtree**. The quadtree is a method which uses spatial occupancy enumeration more efficiently. It is based on recursive subdivision of a square in quadrants [Hunter79], as shown in figure 3.7. At any stage a quadrant could have three states, and different decisions then have to be made:

1- Full quadrant, no further subdivision.

2- Empty quadrant, no further subdivision.

3- Partially full quadrant, subdivision.

We continue to subdivide the quadrants until they are either full or empty, or until we reach a predetermined resolution. As we can see from figure 3.7, each node of the tree is subdivided in four, if it represents a partially full quadrant. In this representation, the nodes and their state are stored in partial arrays: the maximum size of these arrays is $2^n \times 2^n$ [Mortenson85], where n represents the height of the tree. In our example of figure 3.7, $n = 3$ and thus the maximum potential array size is 64, but with the quadrants method, the array's size is 45. The quadtree method converges very quickly to the objects' details, because from the very beginning, the totally empty or totally full quadrants are eliminated. The redundancy problem is also eliminated. In the example shown in figure 3.7, square 25 is represented by just one node instead of four taking the resolution to be

the same as square 21. In order to accurately represent one 2-D entity, we have to go down to very small squares, where the size of the smallest square in this representation is directly related to the surface curvature and the range between the fine and coarse features of the surface to be represented [Mortenson85].

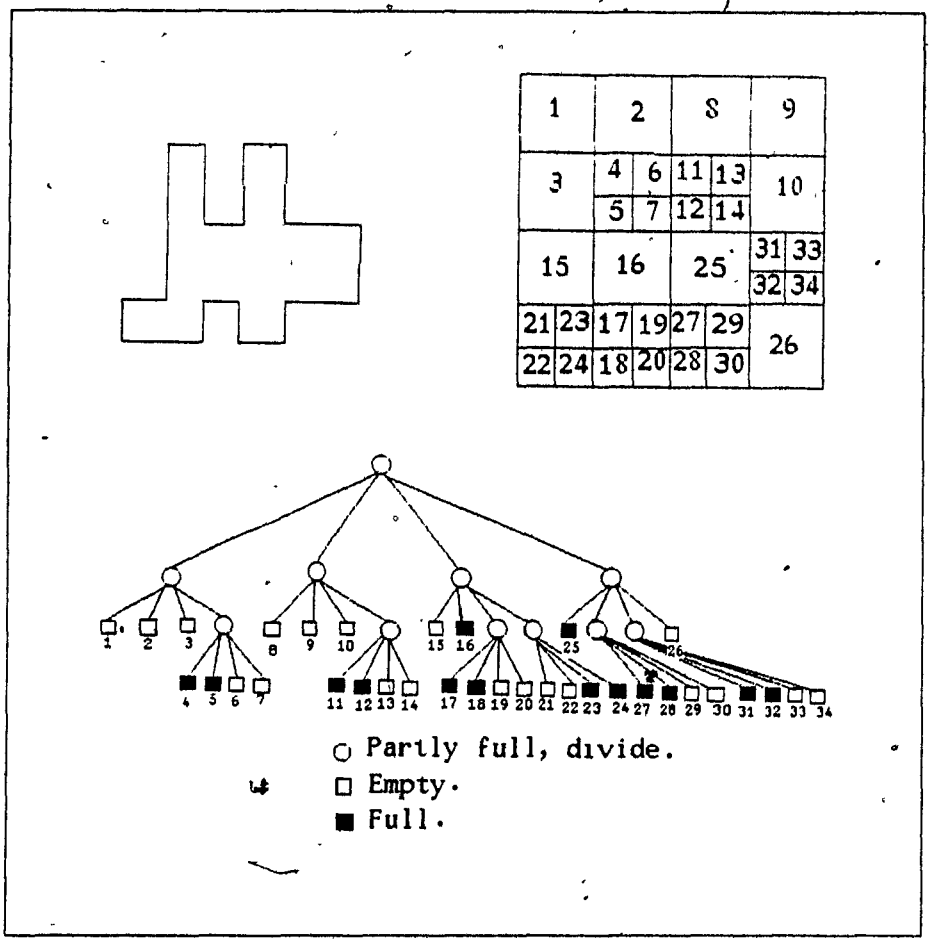


Figure 3.7 Quadtree Representations

The success of these representations has led to its 3-D analog, called the **octree** representations [Meagher80]. The name comes from the fact that each cube (instead of square in 2-D) is subdivided into octants. A subdivision of the cube is made if the cube in question is partially full; otherwise if totally empty or totally full, there is no further subdivision. The representational tree looks like the one of figure 3.7, except that every node, if divisible, has eight descendants instead of four. The maximum potential array size in these representations is $2^n \times 2^n \times 2^n$, where n is the height of the tree. Using the octree

coding however, the amount of memory needed is always less than the maximum size, as was shown in the 2-D analog.

Octree and quadtree representations are very useful and easy to manipulate. Meagher [Meagher82] developed some algorithms in order to translate, rotate, scale solids represented using spatial occupancy enumeration techniques and in particular octree models, he aslo developed techniques to compute the geometric properties and perform the interference analysis. Octree representations are particularly suitable for the computation of certain geometric properties of solids such as volume, first and second moments of inertia, because these computations are usually reduced to the computation of geometric properties over cubes. These representations are also suitable for interference detection since the conversion to B-reps is not difficult. The spatial occupancy enumeration techniques are not very structured however, and can even be described as heuristic, compared to the other approaches discussed above. The success of the octree representations do not depend only on the solid to represent but on its position and orientation as well.

3.1.5 Existing Solid Modellers

There are other representation schemes used for solid modelling, but the representations mentioned above are the ones which most avoid problems such as: Ambiguity, incoherence and creating non-sence objects. Therefore most commercial solid modellers use CSG (PADL[VOEL78], GMSolid[Boyse82]), B-reps (ROMOLUS[Mortenson85]), and-sweep representations (TRUCE[Wang86]). A survey on the exsisting solid modellers can be found in [Baer79].

3.2 Approach Selected

In this section we will compare the previously mentioned approaches and talk about some conversion techniques from one approach to another. At the end of this section we will make a choice on the approach to be used after defining the goals we are seeking.

3.2.1 Comparison

The most common approaches used in solid modelling are sweep representations, CSG, B-reps and spatial occupancy enumeration techniques. Each approach has advantages which make it the most suitable for a particular application. Sweep representations are very structured mathematically and easy to extend; they have found success in applications where the shape of solids changes during a certain task, as in the material removal applications. Sweep representations are theoretically very general, so they can supposedly handle any type of solid. However, it is difficult to analytically describe the volume swept by a certain generator following a certain director. In most solid modellers the rotational and translational sweep are usually the only ones implemented. This is due to the fact that any trajectory could be decomposed into different segments for position and different arcs for twisting. The second most structured technique in solid modelling is CSG. This approach gets its popularity mainly for two reasons: first, its use of boolean operators to hierarchically represent complex solids and second, the certainty of modelling valid and bounded solids. CSG representations have been extensively used in assembly representations, and are certainly the most user friendly approaches. However, they are usually meant for systems where the question of consistency arises, and they are difficult to convert to other representations. Boundary representations are third in mathematical structure, and their aim is to represent a solid by bounding it with a 3-D surface. These representations have found great success in robotics, especially if interference detection is an important objective [Boyse79]. B-reps suffer from validity checking, and the algorithms developed to this end are usually heuristic and time consuming, so that the problem of validity checking is left generally to the user. The final approach is spatial occupancy enumeration, and particularly the octree representation, which are based on the following concept: given a rough approximation of the object by a cube enveloping it, gradually refine the description of the object's parts by providing hierarchically smaller cubes. These approaches are finding widespread use because of their hierarchical structure, but are still heuristic and not well defined mathematically.

As mentioned before, practical solid modellers usually use more than just one approach interacting with the same datamodel. In order to make this possible, a conversion algorithm from one approach to the other is embedded in the solid modeller. In the next section we will present some conversion algorithms.

3.2.2 Conversion Algorithms

Many conversion techniques have appeared in the literature in order to establish compatibility between different approaches [Light82]. In this section we will introduce three conversion algorithms. The aim of this section is not to go into implementation details but to present typical examples of conversion for the sake of demonstration.

The first algorithm we introduce here is the pattern recognition decomposition algorithm which is used in the sweep-to-CSG conversion. In this type of conversion, the solids represented using sweep representations are given, and the algorithm tries to identify patterns which will construct the primitives in the CSG approach. This first operation is called the search for pattern. Once a pattern is found, it is classified. This pattern classification operation is done by taking one point of the pattern and testing it to establish whether it is inside or outside of the current sweep outline. The next step is the construct pattern solid operation, which constructs the subsolid to be considered later as a primitive. Once the subsolid is constructed to make a primitive and classified in order to know the boolean operator to use (union if classified inside and difference if outside) the information is saved in a stack which is later traversed for the CSG representation. The detailed algorithm is presented in [Vossler85]. Note that, this algorithm fails in some circumstances during the search for pattern operation.

The next conversion algorithm we will talk about is used for the CSG-to-B-reps conversion. These kinds of algorithms are called boundary evaluators and usually tend to be very long unless some heuristic tests are included. The basic idea is to suppose that the surfaces of the subsolids constructing a CSG model are unbounded. First, intersect

each surface of a subsolid A with each surface of another subsolid B. The unbounded edges produced this way are the tentative edges or **t-edges** as explained in [Boyse79]. The next step is to intersect these **t-edges** with the unbounded surfaces in order to produce points lying on the edges. These points seperate a **t-edge** in different segments which are then classified as being outside, inside or on the boundary of the whole solid. The real edges are the segments on the boundary of the solid. The classification of the the segments is a difficult task; without getting into detail, we can mention that a common technique to use to overcome this difficulty is the neighborhood model technique [Mortenson85]. The boundary evaluator algorithms guarantee one attractive feature in that the faces of the overall solid are a subset of the faces of the subsolids it contains. This fact could be used for verification after the conversion.

The last conversion algorithm we introduce is for the octree-B-reps conversion This conversion algorithm is summarized in four different steps as explained in [Tosiyasu85].

1- Converting an octree to an extended octree.

2- Labeling the entities of the extended octree.

3- Generating tables of boundary representation information.

4- Generating a sequence of Euler operations.

In this description of the algorithm, we explain the four different points mentioned above. The first point is basically adding geometric information as an extension to the leaves of an octree. This geometric information is the body, planes, edges and vertices of each octant. The body is a reference to the name of the modelled object, and we initially need one body, six planes, twelve edges and eight vertices for each cube. However, considering the adjacent leaves of the tree, this could be reduced to one body, three planes, three edges and one vertex in order to completely define an octant. The reduction is due to shared information; for example, a cube has eight vertices, seven of which are shared with

adjacent cubes to make their origins. The next step is labeling, which is a hierarchical 3-D application of connected graph labeling, a typical "if-then" algorithm. After labeling there is the boundary table generation. There are usually five tables to generate: the object/plane table, edge/vertex table, plane/edge table and then two data tables, namely, the vertex coordinates table and the plane-normal-vectors table. Once these tables are generated, the next step is to generate a sequence of euler operations that will construct the B-reps of the desired solid. This algorithm is fully detailed in [Tosiyasu85]. Note, however, that the algorithm does not work if the object includes cubes connected by just one edge.

The aim of this section was to demonstrate that the conversion algorithms are useful and show their methods of solving the problem. In the implementation part of this chapter we will see in greater detail a particular conversion technique developed for the simulator's purposes.

3.2.3 Decision

In this section we will make a decision on the approach to be used, and will give the reasons for our choice. The B-reps seemed to be the most appropriate for the application we are seeking, namely, a robotic workcell simulation. There are two major reasons for making this choice: First, looking at our datamodel explained in chapter 2, we notice that the lowest entity in the hierarchy is a surface or face, which is the basic entity for the B-reps approach. Second, in any robotics application, the problem of interference detection or avoidance arises; the B-reps have proved their suitability for those types of problems [Boyse79]. One can argue about the weakness the B-reps present when checking the validity or consistency of solids. This is a great drawback for a solid modeller project, but since our aim is primarily simulation and not necessarily to develop a perfect solid modelier, we left the validity and consistency checking to the user. In the data sublanguage explained earlier we check for the validity of a surface when it is specified so that it is impossible to create dangling edges when modelling a solid, but if the user does not check for the whole solid's validity, it could result in dangling surfaces, open solids or overlaped

surfaces. If we had based our choice just on the suitability of the datamodel, the second alternative would have been the octree representations, but the fact that they are heuristic made them unattractive.

The solid modelling part of this simulator is based on the B-reps approach, however this approach lacks generality. This means that the mathematical tools developed in order to make a particular solid, for example a sphere, can not be used to create another solid such as a cylinder. That is why we considered developing a more general, structured approach, and the obvious choice was that of sweep representations. Since our datamodel suits the B-reps, we developed a sweep to B-reps conversion. The mathematical derivations for the B-reps approach and the conversion sweep to B-reps will be given in the next section

3.3 Implemented Solid Modeller

Our goal in this section is to provide information about the solid modeller and its interaction with the datamodel described in chapter 2. We know that the first 3-D entity in our datamodel is what was called a block, so our aim is then to be able to create the 3-D subsolids at the block level. These subsolids would later be considered as entities to be attached to higher level solids by the attach operation. Every subsolid should be presented as a tree whose branches are transformations and whose leaves are faces stored as surfaces in the datamodel. Of course there is no need to have different surfaces if we are creating regular solids; for example, a cylinder of 32 faces, will need one surface pointed to by 32 different transformations and another surface pointed to twice in order to close the cylinder. Other subsolids such as spheres will require more than one surface in order to be constructed from the same data structure. The next paragraph will present the boundary representation of some common subsolids; however the same idea applies to other subsolids not mentioned here.

3.3.1 Examples of B-reps

In this section we show how to construct some subsolids using boundary repre-

sentation techniques. We choose to give as examples a cylinder, a sphere and an ellipsoid, and we will discuss the representation of these subsolids in the datamodel we have created.

3.3.1.1 A Cylinder

The necessary specifications needed in order to construct a cylinder are five: the name of the cylinder to be stored at the block level, its height h, its radius r, the name of the surface to be used for patching, and the number of patches n. The surface to be used for patching is usually a unit square surface, and if this surface does not exist in the datamodel, it is automatically created. First we determine the existence of any block with the same name as the cylinder in question already loaded in the datamodel. If the search is positive then the action is refused and control returns to the database manager at the key tree matcher. If the search is negative, meaning there is no block of the same name as the cylinder, we then verify the validity of the data being specified to construct the cylinder. The next thing to verify is the existence of the patching surface in the datamodel, and if it does not exist it is created as a unit square and linked to the linked list of surfaces.

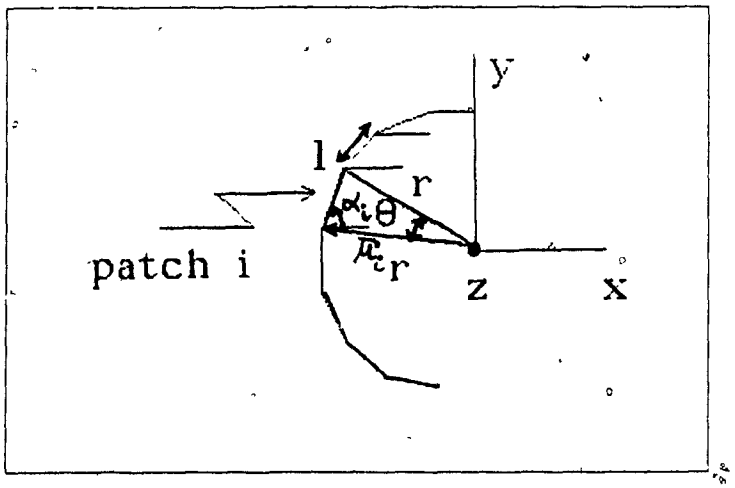


Figure 3.8 Top View of a Cylinder

Figure 3.8 shows part of the top view of the desired cylinder, where we can see that

$$\theta = \frac{2\pi}{n} \tag{3.3a}$$

$$l = r\sqrt{2(1 - cos\theta)} \tag{3.3b}$$

The variables l, from equation 3.3b , and h the height of the cylinder, would be used as scaling factors to the square surface. For the cylinder it is easy to compute the transformation to patch one surface as given by 9 parameters (SRT): scaling (S_x, S_y, S_z), rotating (R_x, R_y, R_z) and translating (T_x, T_y, T_z). We will now show how to calculate all these parameters for the i^{th} patch. From figure 3.8 we can get the deviation of each patch with respect to the horizontal x axis

$$\alpha_1 = cos^{-1}(\frac{r}{l}sin\theta) \tag{3.4a}$$

$$\alpha_2 = cos^{-1}(\frac{r}{l}sin(2\theta) - cos\alpha_1) \tag{3.4b}$$

This formula is recursive and is applicable to the i^{th} patch as follows

$$\alpha_i = cos^{-1}(\frac{r}{l}sin(i\theta) - \sum_{j=1}^{i-1} cos(\alpha_j)) \tag{3.4c}$$

We now have all the information for scaling and rotation, but must still find the translational part of the transformation. Let us call $\mathbf{u}_i$ the translational vector From figure 3 8 we can easily see that:

$$\mathbf{u}_i = \begin{pmatrix} -rsin(i\theta) \\ rcos(i\theta) \\ 0 \end{pmatrix} \tag{3.5}$$

We can now give the results obtained to construct a cylinder out of a 2-D square. The operation involves scaling, rotating, and translating by the following factors:

$$S_x = l, \quad S_y = h, \quad S_z = 1 \tag{3.6a}$$

$$R_x = \frac{\pi}{2}, \quad R_y = 0, \quad R_z = \alpha_i \tag{3.6b}$$

$$(T_x, T_y, T_z) = \mathbf{u}_i^T \tag{3.6c}$$

The above equations are valid for each transform_surface pointing to the unit square surface used in patching. A graphic result is shown in figure 3.9.

The $n + 2$ transform_surfaces would have names whose prefix is specified by the user. Moreover, the transform_surfaces could have been specified as homogeneous

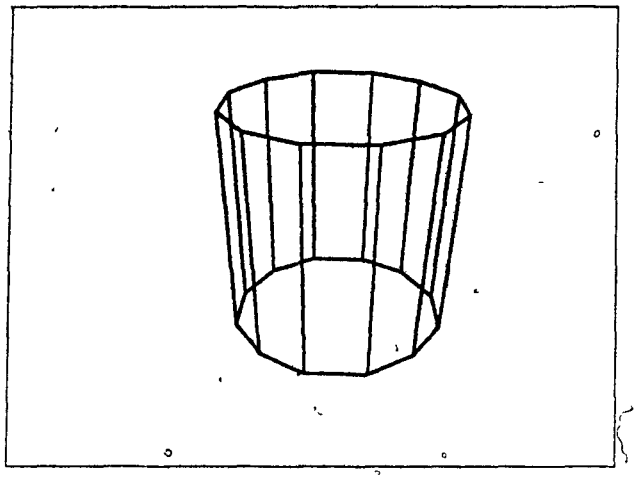


Figure 3.9 B-reps of a Cylinder

transformations instead of seperate SRT factors, however in this example we decided to show the SRT instead, as they are easier to understand. Once all this is done, the cylinder is closed at the top and the bottom by a certain surface. The appropriate surface for closing is created using the information from the vertices of the top or bottom of the cylinder.

3.3.1.2 A Sphere

For a sphere it is not possible to compute the transformations as a set of SRT factors as we did in the case of the cylinder. Instead, homogeneous transformations will be computed since the transform levels in the datamodel support both types of information. . To construct a sphere and link it to the blocks' linked list we need four basic parameters: the name of the sphere to be stored as a block, its radius r, the number of faces per half slice of the sphere n, and the number of slices for the whole sphere m. Some tests are first performed for the validity of the parameters, and then the computation of the homogeneous transformations starts. The number of faces per half slice is the number of surfaces that have to be created and later attached with different homogeneous transformations to make the whole sphere. A general face of a slice is shown in figure 3.10a, a half slice is shown in figure 3.10b, and the position of a slice on a sphere is shown in figures 3.10c and 3.10d. Our purpose is to first construct the n required surfaces and then to find the homogeneous transformations to move them in space in order to patch their appropriate locations. The

construction of the surfaces is easily performed after deriving the following equations based on figure 3.10a:

$$\beta = \frac{2\pi}{m} \tag{3.7a}$$

$$\alpha = \frac{\pi}{2n} \tag{3.7b}$$

The length w is the same for all the surfaces (fig. 3.10a)

$$w = r\sqrt{2(1 - cos\alpha)}. \tag{3.7c}$$

We can also derive the other dimensions of the surfaces, recursively

$$ll_1 = r\sqrt{2(1 - cos\beta)} \tag{3.8a}$$

$$lu_i = 2rcos(i\alpha)\sqrt{2(1 - cos\beta)} \tag{3.8b}$$

$$ll_i = lu_{i-1} \tag{3.8c}$$

where $1 \leq i \leq n$. Notice that since lu_n is equal to zero, the nearest patches to the pole of the sphere are triangles. Once the n_1 surfaces are created, they are stored at the surface level of the datamodel; their names should be specified by providing a prefix. Now that the surfaces are available, we should compute the transformations to manipulate them accordingly. Knowing that each surface is pointed to by m different transform surfaces in order to make a whole circumference and denoting by j the slice's order in the circumference, we compute the orthonormal basis $B'_{ij} = (\mathbf{e}'_{1ij}, \mathbf{e}'_{2ij}, \mathbf{e}'_{3ij})$ attached to the same surface at each desired position, as shown in figures 3.10a and 3.10c. The basis B'_{ij} should be computed with respect to the world basis $B = (\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)$. The three vectors $\mathbf{r}_{1ij}$, $\mathbf{r}_{2ij}$, and $\mathbf{r}_{3ij}$,shown in figure 3.10d, are directly related to the basis B'_{ij}, and we can now give the necessary formulae to compute them:

$$\mathbf{r}_{1ij} = r \begin{pmatrix} cos(i\alpha)cos(j\beta) \\ cos(i\alpha)sin(j\beta) \\ sin(i\alpha) \end{pmatrix} \tag{3.9a}$$

$$\mathbf{r}_{2ij} = r \begin{pmatrix} cos(i\alpha)cos((j+1)\beta) \\ cos(i\alpha)sin((j+1)\beta) \\ sin(i\alpha) \end{pmatrix} = \mathbf{r}_{1i,j+1} \tag{3.9b}$$

$$\mathbf{r}_{3ij} = r \begin{pmatrix} cos((i+1)\alpha)cos(j\beta) \\ cos((i+1)\alpha)sin(j\beta) \\ sin((i+1)\alpha) \end{pmatrix} = \mathbf{r}_{1,i+1,j} \qquad (3.9c)$$

These equations are applicable for $1 \leq j \leq m$ and $1 \leq i \leq n$ and are all expressed with respect to the basis B. From the above equations we can express the basis B'_{ij} as:

$$\mathbf{e}'_{1ij} = \frac{\mathbf{r}_{2ij} - \mathbf{r}_{1ij}}{\|\mathbf{r}_{2ij} - \mathbf{r}_{1ij}\|} \qquad (3.10a)$$

$$\mathbf{e}'_{2ij} = \frac{\frac{\mathbf{r}_{3ij} - \mathbf{r}_{1ij}}{\|\mathbf{r}_{3ij} - \mathbf{r}_{1ij}\|} - (cos\gamma)\mathbf{e}'_{1ij}}{sin\gamma} \qquad (3.10b)$$

and

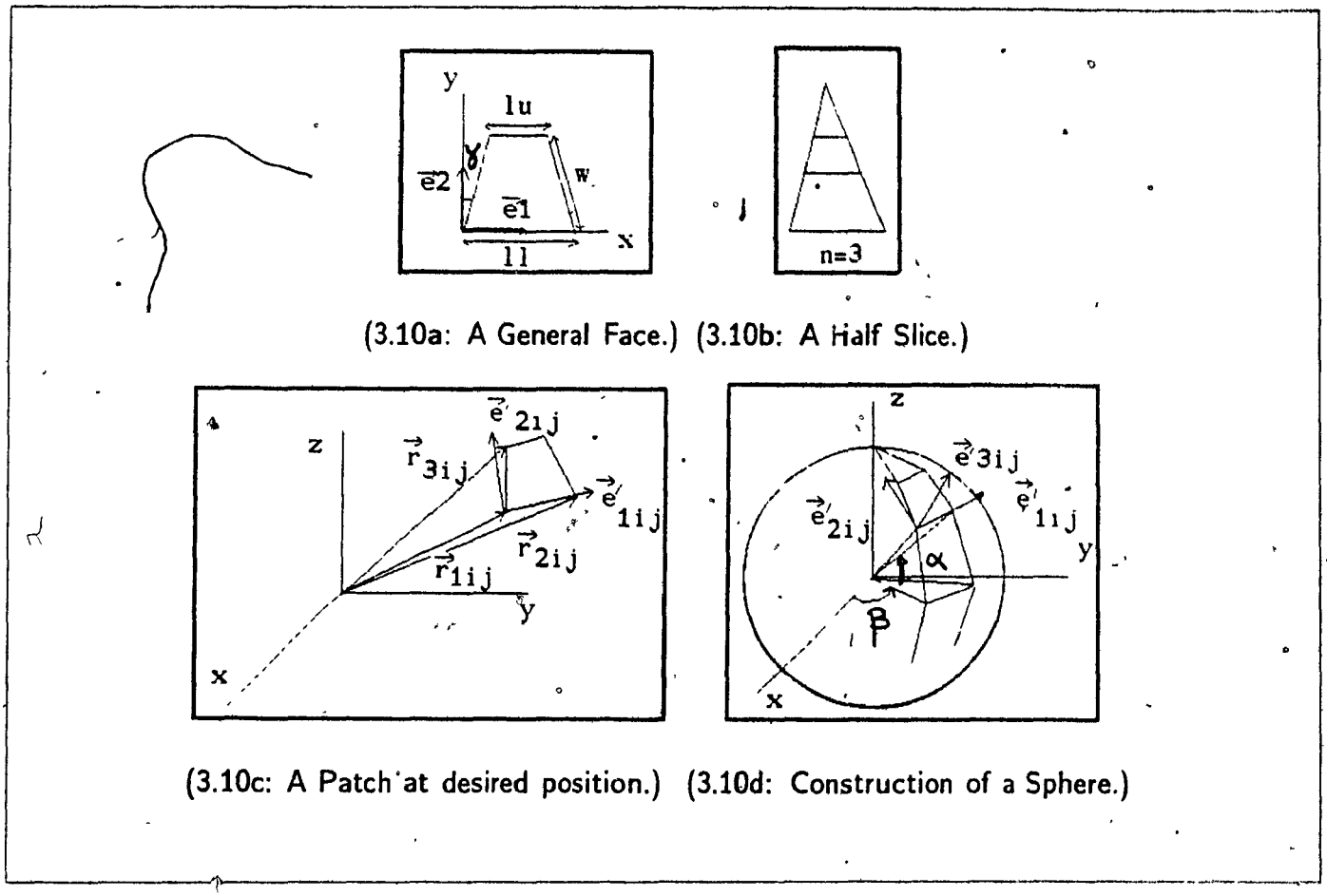
$$\mathbf{e}'_{3ij} = \mathbf{e}'_{1ij} \times \mathbf{e}'_{2ij} \qquad (3.10c)$$

The above equations define the rotational part of the homogeneous transformation. The translational part is only defined by the vector $\mathbf{r}_{1ij}$: thus the whole transformation becomes:

$$\mathbf{T}_{ij} = (\mathbf{e}'_{1ij}, \mathbf{e}'_{2ij}, \mathbf{e}'_{3ij}, \mathbf{r}_{1ij}) \qquad (3.11)$$

Through the above procedure, we create just one hemisphere. In order to create the other half, we multiply all the homogeneous transformations by a reflection with respect to the XY, plane, and link all the transform_surfaces together. The final tree structure of the sphere as stored in the datamodel is shown in figure 3.11.

This idea of attaching a basis to the surface to be patched and then to compute it at the appropriate location of the surface on the solid with respect to the world basis is easy to apply and requires no more than geometrical techniques. Moreover, considering the regularity of the primitives we want to create, it is usually possible to find recursive approaches and hence recursive programming. Examples of spheres that have been created with this approach are shown in figure 3.12, having different (m, n) but keeping the same radius for comparison.



(3.10a: A General Face.) (3.10b: A Half Slice.)

(3.10c: A Patch at desired position.) (3.10d: Construction of a Sphere.)

Figure 3.10 B-reps Construction of Spheres.

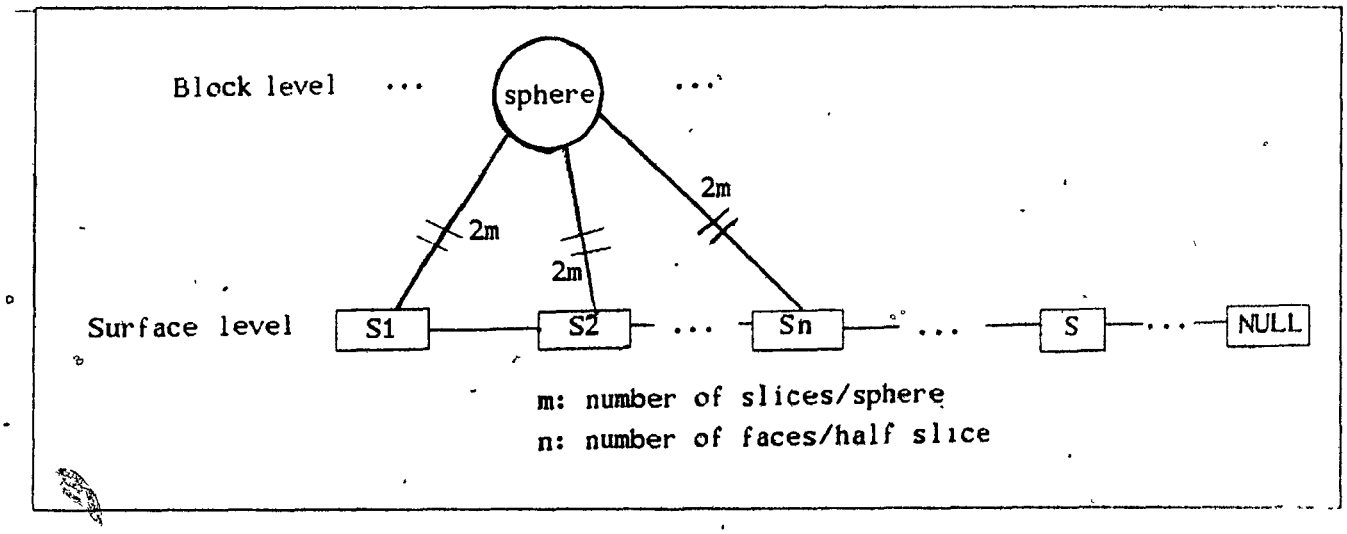


Figure 3.11 Tree Structure of a Sphere.

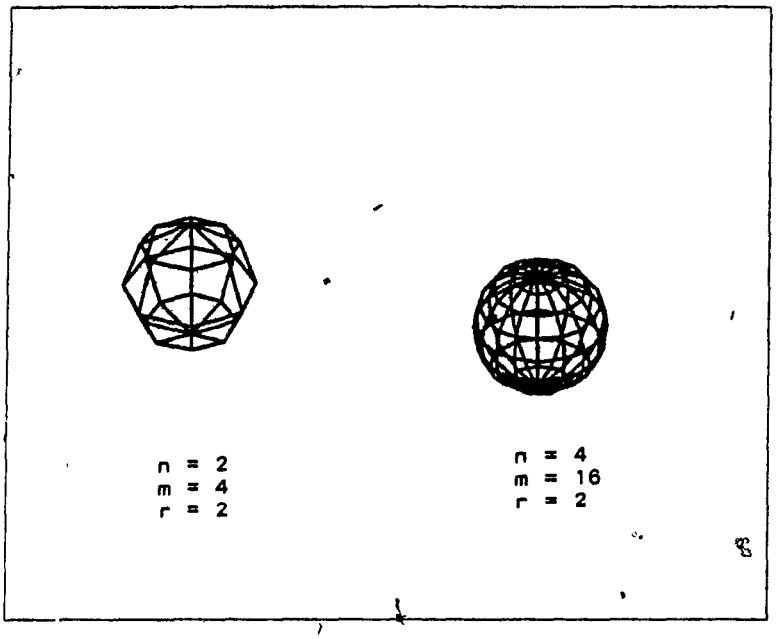


Figure 3.12 B-reps of Spheres.

3.3.1.3 An Ellipsoid

Creating an ellipsoid is a simple extension of a sphere, and there are two possible courses of action. Let us suppose that we have already created a sphere S at the block level. We can attach this sphere S to an object E, using the attach operation, by specifying a transform_block which has different scaling factors in x, y, and z respectively. However, this means that an ellipsoid is stored as an object and the sphere as a block. If we want to store the ellipsoid as a block in the datamodel we can copy the sphere S at the blocks' level to an ellipsoid E at the same level, since the copy query accepts transformation specifications. This means that the initial sphere is used as a template for the ellipsoid. Figure 3.13 shows two ellipsoids which have been created from the two spheres of figure 3.12.

3.3.2 Examples of Sweep Representations

The above examples showed the B-reps manner of representing solids. The most general idea in the B-reps is the one used for constructing the sphere, where a certain basis, at the appropriate patching location, is expressed with respect to a fixed

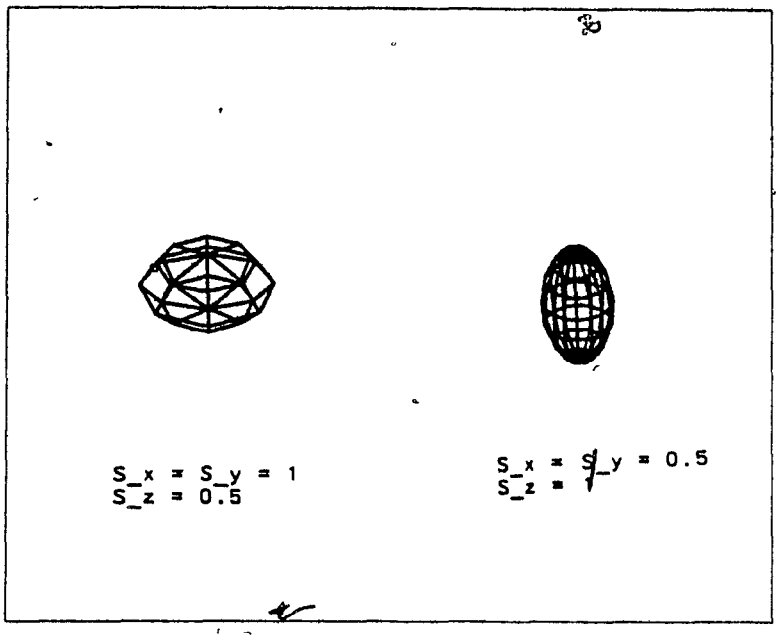


Figure 3.13 B-reps of Ellipsoids.

basis, generally the world coordinate basis. This determines the transformation needed in order to perform the necessary patching. What is attractive about this approach is the possibility of expressing the formulae recursively, because of the uniformity of the B-reps in representing regular solids. However, this approach is very dependent on the subsolid to be created, and the mathematical derivations had to be developed for each subsolid independently. Seeking flexibility, we have developed two sweep types, the translational sweep and the rotational sweep. We will refer to this feature as sweep but it is really a sweep to B-reps conversion since the final result is in terms of patches bounding the solid to be represented, as will be shown later. This sweep feature helps to create non regular solids as well as regular ones, and the advantages of this approach are well known from previous discussions. In this implementation we considered simplified cases of sweep, the first simplification is applicable to both the translational and the rotational sweep, notably the fact that the generators to be used are supposed to be surfaces rather than solids. This limitation could easily be eliminated if needed because a solid itself is represented by surfaces in the datamodel. Some other simplifications have been made and they will be mentioned seperately in the sections describing translational and rotational sweeps.

3.3.2.1 Translational Sweep

The idea is the following: given a surface S, sweep it along a straight director for a distance h. The specifications needed in order to perform this are: the name of the block to be created, the name of the generator to be used, the direction vector, the length of the sweep, and the name of the surface to be used for patching the route swept by each edge of the generator. In this particular example of translational sweep, we considered the case where the director is perpendicular to the generator, which means that the director is parallel to the z axis. As usual the validity of the specifications is verified. In particular, if the unit square surface to be used for the patching is not in the datamodel, it is created and linked to the other surfaces. Figure 3.14 shows an example of a translational sweep.

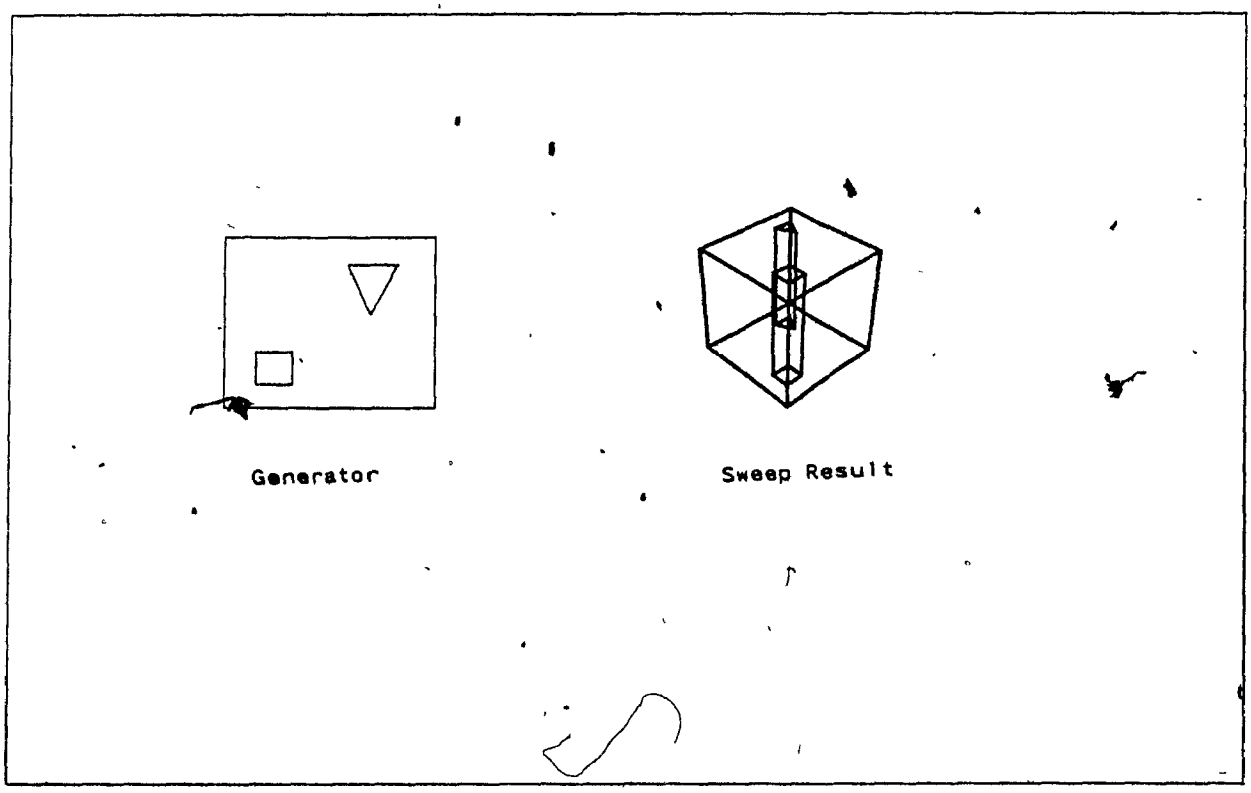


Figure 3.14 Translational Sweep.

The final result of the translational sweep is a block pointing to two surfaces through many transform_surfaces. The first surface is the generator itself which is used

twice for closing the path swept along the director, and the second surface is a unit square which is used to surround the path in question. Let us suppose that the generator has m edges as part of its main contours and n edges as part of its hole contours; then for each block created with this approach we have $m + n + 2$ attached transform_surfaces. In this section we will give the SRT factors necessary to point to the $m + n$ squares. The other two sets of SRT factors that are pointing to the generator are obvious: one of them is the specification for the identity matrix, and the other is just a translation of h along the z axis. The SRT factors pointing to the square are developed next. We note that this algorithm was not stated recursively and depends on the edge to be moved. Let us suppose that the edge is defined by two vertices $P_1 = \begin{pmatrix} x_1 \\ y_1 \\ 0 \end{pmatrix}$ and $P_2 = \begin{pmatrix} x_2 \\ y_2 \\ 0 \end{pmatrix}$, thus the SRT factors are determined as a function of P_1 and P_2:

$$S_x = \|\vec{P_1P_2}\|, \quad S_y = h, \quad S_z = 1 \tag{3.12a}$$

$$R_x = \frac{\pi}{2}, \quad R_y = 0, \quad R_z = \begin{cases} -cos^{-1}(\frac{x_2-x_1}{S_x}), & \text{if } y_2 \leq y_1; \\ cos^{-1}(\frac{x_2-x_1}{S_x}), & \text{otherwise.} \end{cases} \tag{3.12b}$$

$$T_x = x_1, \quad T_y = y_1, \quad T_z = 0 \tag{3.12c}$$

Figure 3.14 was created using the techniques developed above, figure 3.14a shows the generator with holes in it, and figure 3.14b shows the result of the translational sweep of the generator in question along the z axis for a certain height.

3.3.2.2 Rotational Sweep

The mathematical derivations for the rotational sweep will be developed in this section, and we will not develop the SRT factors for the transform_surfaces but rather the transformations themselves. Moreover, if we consider the most general case, the derivations would be rather lengthy and cumbersome. We will therefore develop the case in which only one edge is sweeping and extend it to the general case later. In this particular implementation we considered the case where the revolution is performed with respect to the z axis. We also assumed that the sweep is done uniformly and hence the patching surfaces

created while sweeping an edge are all of same shape. Figure 3 15a shows an edge of length c to be swept with respect to the z axis, and figure 3.15b shows a typical surface that has to be created for the patching. The specifications needed to perform the rotational sweep are the edge's geometric information, the resolution specified as the number of times the patching surface is used (n), and the name of the block to be created. Of course, some tests are performed to ensure the validity of the specifications. The method to be used here is the following: for each edge create a surface or patch, and then express the basis $\mathbf{B}_i = (u_{1_i}, u_{2_i}, u_{3_i})$ which is attached to the i^{th} patch with respect to the world coordinate system.

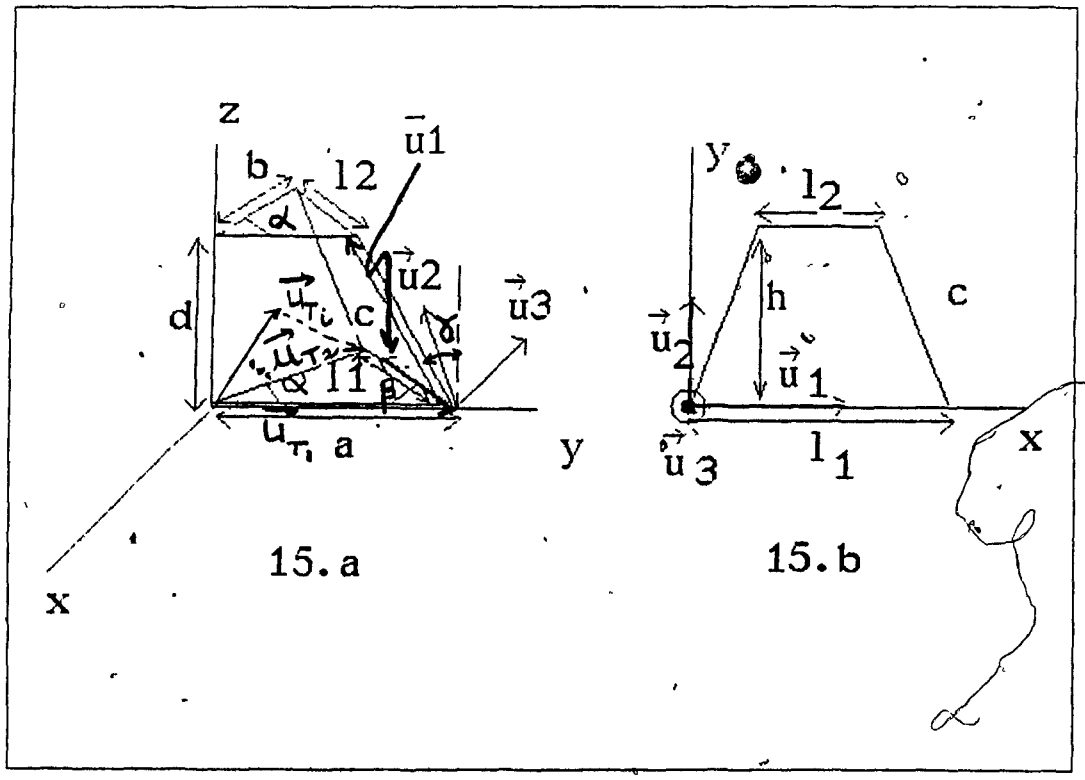


Figure 3.15 Rotational Sweep.

This will determine the rotational part of the transformation required to move the patching surface to its appropriate location. The translational part of the transformation is determined by the vector u_{T_i} of figure 3.15a. Let us now give the necessary formulae to

perform the creation of the patching surface:

$$\alpha = \frac{2\pi}{n}, \quad l_1 = a\sqrt{2(1-cos\alpha)}, \quad l_2 = b\sqrt{2(1-cos\alpha)} \tag{3.13a}$$

$$c = \sqrt{(a-b)^2 + d^2}, \quad h = \sqrt{c^2 - \left(\frac{l_1 - l_2}{2}\right)^2} \tag{3.13b}$$

The above formulae define the 2-D surface that is created for each edge, what is left now is to express the basis $\mathbf{B}_i$ attached to the i^{th} patch with respect to the world coordinate basis $\mathbf{B} = (\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3)$, given that $\beta = \frac{\pi - \alpha}{2}$ and $\gamma = cos^{-1}(d/c)$ then:

$$\mathbf{u}_{1_i} = \begin{pmatrix} l_1 sin((i-1)\alpha - \beta) \\ -l_1 cos((i-1)\alpha - \beta) \\ 0 \end{pmatrix}, \quad \mathbf{u}_{2_i} = \begin{pmatrix} csin\gamma sin((i-1)\alpha) \\ -csin\gamma cos((i-1)\alpha) \\ ccos\gamma \end{pmatrix} \tag{3.14a}$$

and $\mathbf{u}_{3_i} = \mathbf{u}_{1_i} \times \mathbf{u}_{2_i}$, the translational part is given by:

$$\mathbf{u}_{T_i} = \begin{pmatrix} -asin((i-1)\alpha) \\ acos((i-1)\alpha) \\ 0 \end{pmatrix} \tag{3.14b}$$

and the whole transformation pointing to the patch in question is:

$$\mathbf{T}_i = (\mathbf{u}_{1_i}, \mathbf{u}_{2_i}, \mathbf{u}_{3_i}, \mathbf{u}_{T_i}) \tag{3.14c}$$

The above equations were developed for just one edge, however, they are easily extended to a whole surface since a surface is itself constructed by edges defining the main contours and the hole contours. The rotational sweep was developed to be applied with respect to the z axis, but could, of course, be extended easily. Figures 3.16 and 3.17 show examples of solids that have been created using the rotational sweep, we note that the cone of figure 3.16 is closed.

3.3.3 Further Extensions

At this stage, we presented the implemented solid modeller. It is based on the boundary representations. To extend its capability, a sweep to B-reps conversion was added. B-reps representations are very specific to the subsolid being modelled and will

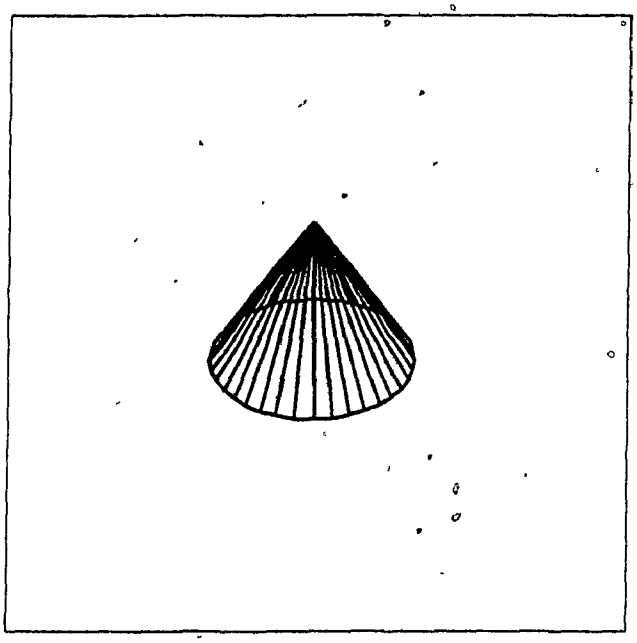


Figure 3.16 A Cone With Rotational Sweep

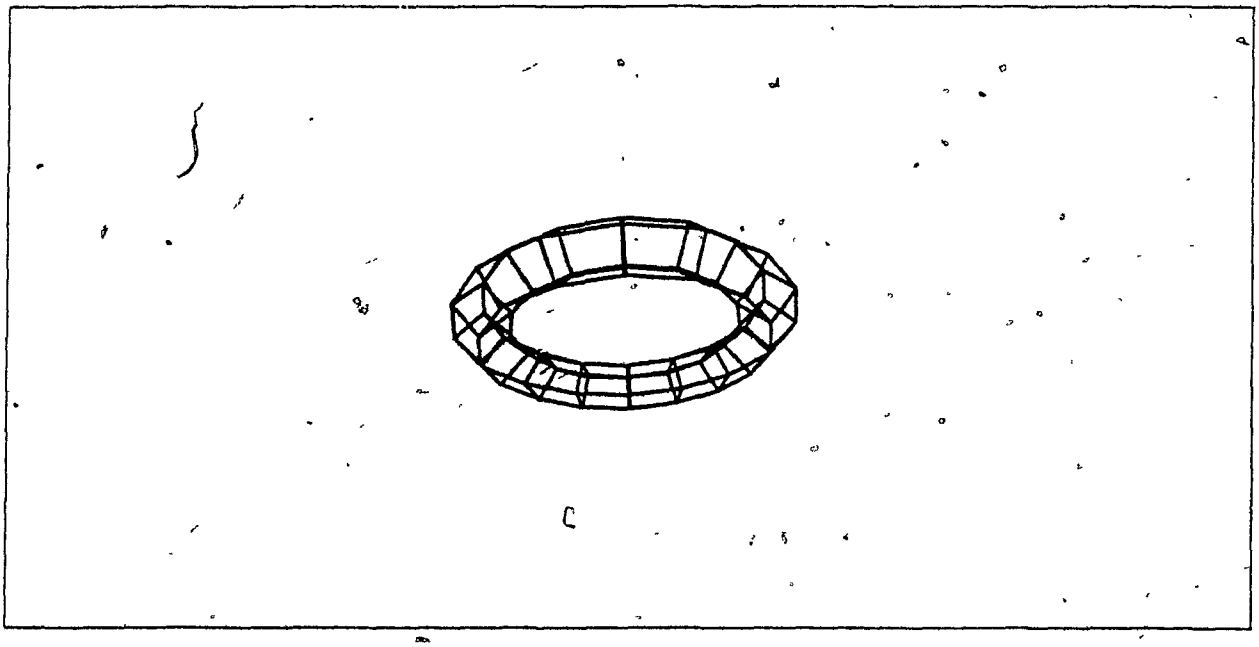


Figure 3.17 A Torus With Rotational Sweep.

always need extensions for other possible subsolids. But, we can suggest one possible extension for the sweep representations, namely, to develop facilities for specifying the director as PD curves instead of just a translational and rotational sweep. This extension could be conducted further to 9 components curves if dealing with flexible solids. The

translational and rotational techniques developed for the sweep representations were made general enough to accomodate the aforementioned extensions.

3.4 Graphics

The solids which have been created using the solid modeller need to be presented graphically for the simulation. In this section we present the datamodel modifications for graphics purposes and the graphics facilities that have been developed. If we look at our datamodel, we notice that the solids at any 3-D entity level are presented by means of transformations pointing to lower level entities, this representation scheme is propagated until the 2-D surfaces are reached. We have to modify this form of storage to something easier to manipulate graphically namely edges and vertices. This process is called the world coordinates generator, and the entities obtained are the world_solids. Once the world_solids are computed the information that was stored in the form of figure 2.5 is no longer needed. Therefore, the previous datamodel can be cleared using some facilities in the data sublanguage. The datamodel of figure 2.5 is then replaced by another datamodel, which is more appropriate for graphics manipulations, but is less user friendly. In the next paragraph we present the world coordinates generator and the queries that could be applied on the world_solids, after this, we present some graphics facilities that have been developed such as the view point transformation, the clipping, the perspective view, the back surface removal, and the multiple windows facilities.

3.4.1 World Coordinates Generator

The world coordinates of the world_solids are stored the following way: each world_solid has a name, a type (block, object, or scene), the name of the entity that it was created from, and a field of geometric properties; this information is stored in a world_solid structure. Moreover, each world_solid points to three other structures, the first one is the beginning of a linked list of 3-D main surface contours, the second is for the 3-D hole

surface contours, and the third structure is another world_solid in order to form a linked list of world_solids. Each 3-D surface contour (main and hole) structure points to another 3-D surface contour in order to form linked lists, and points to the beginning of a linked list of 3-D vertices. The representation is better shown in figure 3.18. Hole contours were omitted to simplify the figure. All the linked lists are bounded by a begin and an end mark and garbage collections is used as in the previous datamodel.

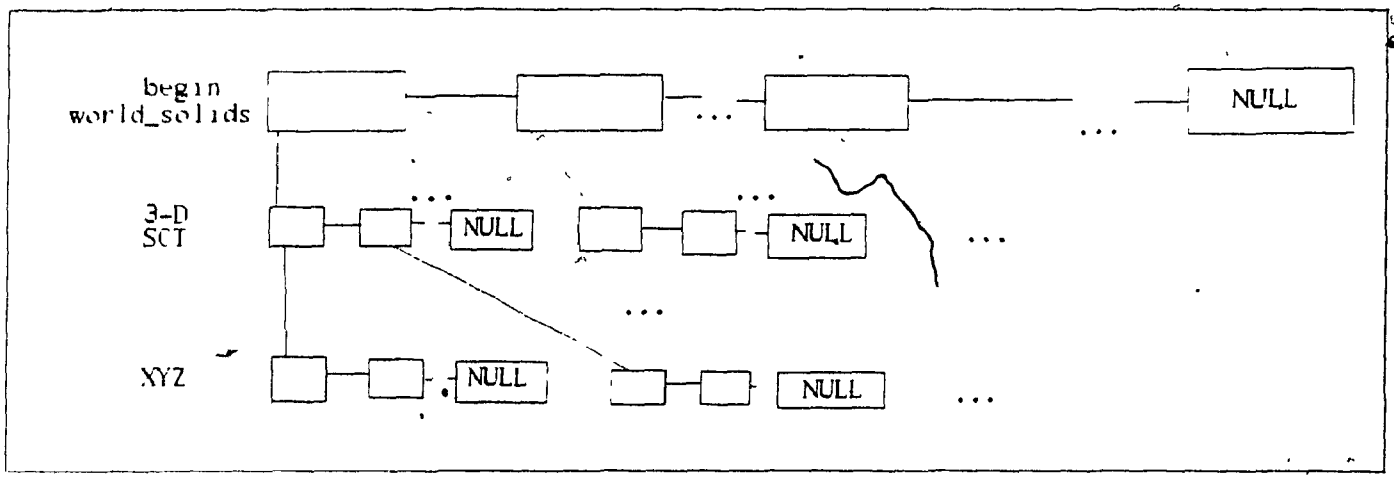


Figure 3.18 World_solid Structure

Some facilities have been developed in order to interact with the world_solids, the ones which are visible to the user are the creation, the removal, the showing, and the saving and loading facilities. Some other facilities have been implemented in order to accomodate the queries, they are, the searching and the memory allocation if needed. The modification query has not been implemented since it is difficult for the user to modify 3-D world coordinates. However, if the user needs to modify these coordinates it is necessary to go back to the other form of storage (i.e the one of figure 2.5). A world_solid could be created for any 3-D entity without distinction between a scene, an object or a block. We should mention here that a world_solid is not necessarily just one solid , it could be made of many solids as long as they are supposed to be constructing the same entity or different entities which are manipulated together; for example a table could be constructed as an object in the datamodel, an oscilloscope as another object placed on the table, if those two

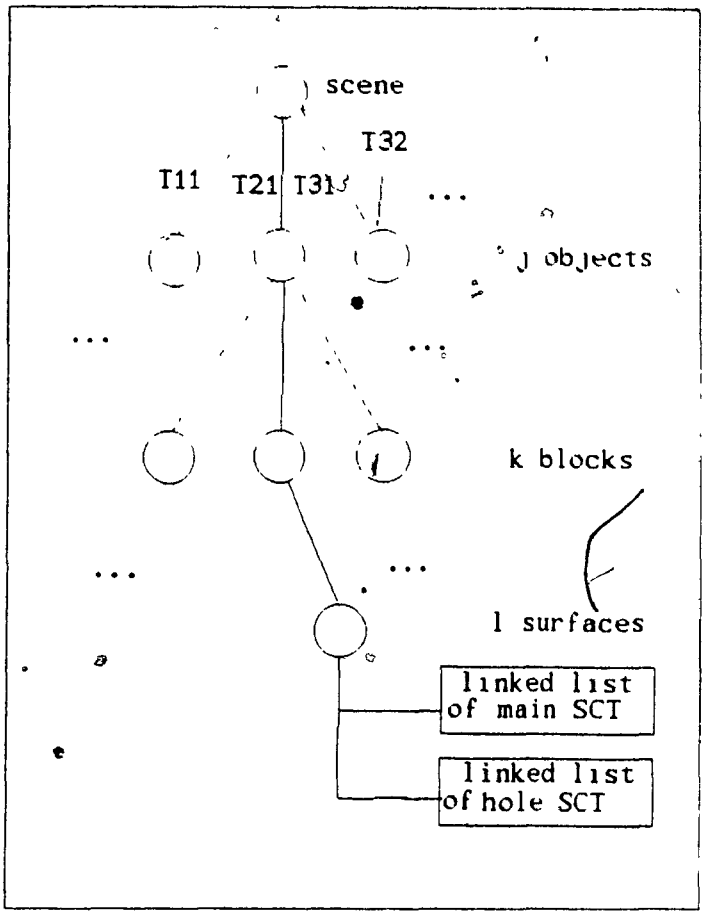


Figure 3.19 Structure of a Solid in the Datamodel

entities are stable with respect to each other, they could be grouped together as a scene and then a world_solid would contain both of them.

Figure 3.19 shows the structure of a solid or a group of solids that need to be processed through the world coordinates generator, which transforms the representation of figure 3.19 to the one of figure 3.18, where each entity is expressed as:

$$\text{entity} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m_i} \mathbf{T}_{ij}(\text{lower level entity})_i \qquad (3.15)$$

where n is the number of lower level entities, and m_i is the number of transformations pointing to the i^{th} lower level entity. The above equation is applied for all the 3-D solids, at the surfaces level. Both main and hole contours are considered. Once the world coordinates of a world_solid are created by invoking the world coordinates generator, they should be

transformed to the view coordinate system and then processed graphically for the display, the graphics facilities will be explained in the next section.

3.4.2 Graphics Facilities

Once a view point is specified by the user, we need to perform the view point transformation. The coordinates of a world_solid are transformed and stored in a view structure This structure is the one that will be modified for the perspective view and for the motion. A view_solid structure could combine many world_solids in a linked list. A view_solid structure contains some descriptive fields as well as geometric fields, the most important descriptive fields are: the name, the color, and a field which describes the solid as movable or stable. The geometric fields are many. First the whole geometry of the view_solid stored as linked lists as described for the world_solid shown in figure 3 18. Second a bounding volume is given as the dimensions of a parallelepiped. Finally a basis is attached to the parallelepiped expressed with respect to the world coordinate system which will be used for the grasping operation. As usual the view_solids are stored as bounded linked lists with garbage collections at each level i.e the view_solids level, the 3-D surface countours level, and the vertices level. All the facilities that were stated above for the world_solids are applicable to the view_solids. Moreover, a modification facility has been implemented for modifying the descriptive fields of the view_solids. The view point transformation is easy to perform once we express the view coordinate basis with respect to the world coordinate basis. Some simplifications have been made in order to compute the view coordinate basis. We suppose that the z axis of the view coordinate system is pointing to the origin of the world coordinate system. We also suppose that the x axis for the view coordinate system is parallel to the xy plane of the world coordinate system with a predefined direction. In our discussion from now on, matrices are 3×3 matrices and their subscripts determine the basis with respect to which they are expressed. The same notation applies to vectors and points. If the subscripts are omitted then the default reference basis is with respect to the world coordinate system. Let us now suppose that $\mathbf{Q}$ is the matrix that expresses the view coordinate system with respect to the world coordinate system,

and V is the view point known in the world coordinate system, then for each point P of the world_solid we have:

$$[P]_v = \begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} = [\mathbf{Q}]^T [\bar{V}\bar{P}]_w \tag{3.16}$$

Next the clipping should be performed against a viewing pyramid. The user defines the ratio $\frac{D}{S}$ where D is the distance from the view point to the plane of the display screen and S is the screen size; a point is visible on the screen if :

$$-z_v \leq \frac{D}{S} x_v \leq z_v, \quad and \quad -z_v \leq \frac{D}{S} y_v \leq z_v \tag{3 17}$$

The details of the clipping algorithm can be found in references such as [Newman79] or [Foley82]. We should state however, that the clipping can cause changes in the vertices and can of course change the number of vertices or even surface countours. These results are managed and stored in the same linked lists. If more space is needed it is allocated, if less space is needed the surplus is released for later use. After clipping the perspective transformation is performed, for each clipped point $P = \begin{pmatrix} x_c \\ y_c \\ z_v \end{pmatrix}$ of the view solid, we compute $P_s = \begin{pmatrix} x_s \\ y_s \\ z_v \end{pmatrix}$ where x_s and y_s are the screen coordinates, and are defined as follows:

$$x_s = (\frac{x_c}{z_v}) V_{sx} + V_{cx}, \quad and \quad y_s = (\frac{y_c}{z_v}) V_{sy} + V_{cy} \tag{3 18}$$

Where $V_{cx}, V_{sx}, V_{cy}, V_{sy}$ are the commonly used parameters to describe a window on the display screen. After the perspective view transformation a view_solid is represented at the lowest level-by linked lists of vertices, each vertex has the screen coordinates $(x_s, y_s)^T$ and a view coordinate z_v which has stayed unchanged after the graphics manipulations. z_v is not useful for displaying but will later be used for motion.

A back surface removal algorithm could easily be integrated. After performing the view point transformation we compute a normal vector to each surface contour. If the z coordinate of the normal vector with repect to the view coordionate system is negative, that means that the viewer is facing the surface, therefore the surface should be processed for

the clipping and the perspective transformation, if the z coordinate is positive the surface contour should not be visible, and hence no further graphic manipulations are needed.

One attractive graphic facility is the use of multiple windows available in the SUN workstation. For simulation purposes different views would be helpful to the user. The different windows are stored as linked lists in the datamodel, they are identified by names, each window structure has all the information needed for the view point transformation, clipping, and perspective view Each window is managed and served by a different process running as a "child" of the main program in order to respond to changes in window's properties such as closing, opening, and changes in size. In order to display a particular solid, the user invokes the display command with the specification of the window. All the queries that could be applied to the world_solids could also be applied to the windows. Every window's structure has additional information which will be used later for motion. Only the basic facilities such as clearing a particular window or clearing a particular view_solid in a particular window have been developed to ease the use of the graphics interface for the user. As future work, we can suggest more graphic options that would be helpful, such as "zoom" to help the user to closely observe the motion in the workcell.

3.5 Geometric Properties

Usually, after representing a solid using the solid modeller's facilities, we are interested in computing some of its geometric properties such as the area of its closed boundary, its centroid, and its volume or 0^{th} moment. In this section, we introduce the facilities that were developed in order to accomplish this. These facilities are developed to fulfil the fourth property given by Requicha mentioned in section 1.

The area of a planar region could be considered as the 0^{th} moment in a two dimensional space. The k^{th} moment of a closed region Ω in a ν dimensional euclidean space is defined as:

$$I_k = \int_\Omega f_k(\vec{r}) d\Omega \qquad (3.19)$$

where $f_k(\mathbf{r})$ is a homogeneous function of k^{th} degree of the position vector $\mathbf{r}$ and is at the same time a tensor of the k^{th} rank. Let $g_m(\mathbf{r})$ be a m^{th}-rank tensor such that:

$$div[g_m(\mathbf{r})] = f_k(\mathbf{r}) \tag{3.20}$$

where $m = k + 1$. Using the Gauss Divergence Theorem, we can state that.

$$\mathbf{I}_k = \int_{\partial\Omega} g_m(\mathbf{r}) \cdot \mathbf{n} d\partial\Omega \tag{3.21}$$

Where $\partial\Omega$ denotes the boundary of the closed region Ω and $\mathbf{n}$ is the unit normal vector to $\partial\Omega$, pointing out of Ω region. The computation of the 0^{th} moment can be reduced to

$$\mathbf{I}_0 = \frac{1}{\nu} \int_{\partial\Omega} \mathbf{r} \cdot \mathbf{n} d\partial\Omega \tag{3.22}$$

as shown in [Angeles83] Using this above formula, the area and the volume of a world-solid represented in the datamodel shown in figure 3.18 can be computed and these will be presented in the following sections.

3.5.1 Area

Let us assume that a solid is represented by n patches, the area of the solid is then the sum of the areas of each individual patch: $A = \sum_{i=1}^{n} A_i$, wher A_i is the area of a patch. The problem is now reduced to the computation of the area of a patch. Moreover, the area of a patch could be computed as if the patch were in a 2-D space, thus $\partial\Omega$ represents, in this case, a closed m-sided polygon:

$$\partial\Omega = \bigcup_{i=1}^{m} \partial\Omega_i \tag{3.23}$$

Where $\partial\Omega_i$ denotes the i^{th} side of the polygon. And the area a of the polygon is then expressed as:

$$a = \frac{1}{2} \sum_{i=1}^{m} \int_{\partial\Omega_i} \mathbf{r} \cdot \mathbf{n}_i d\partial\Omega_i \tag{3.24}$$

Where $\mathbf{n}_i$ is the unit normal vector to the side or edge in question in a 2-D space. In order to decompose this further, let us consider separately the contribution of one edge to the area of the whole polygon, this could be written as:

$$\frac{1}{2}\mathbf{n}_i \cdot \int_{\partial\Omega_i} \mathbf{r} d\partial\Omega_i = \frac{1}{2}\mathbf{n}_i \cdot \bar{\mathbf{r}}_i s_i \tag{3.25}$$

Where s_i is the length of the considered edge and $\bar{\mathbf{r}}_i$ is the position vector of its centroid, as shown in figure 3.20; all those parameters are easily computed knowing the two vertices of the edge.

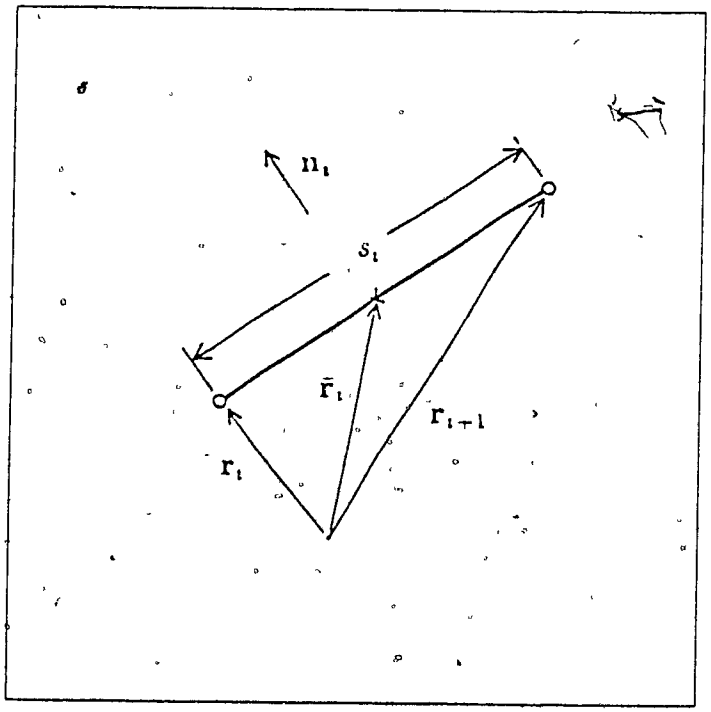


Figure 3.20 A Line Segement

Then, the area of the bounded region Ω is finally stated as:

$$A = \frac{1}{2} \sum_{patches} \sum_{i=1}^{m} \mathbf{n}_i \cdot \bar{\mathbf{r}}_i s_i \tag{3.26}$$

The latter formula is suitable for computer implementation.

3.5.2 Volume

The 0^{th} moment in a 3-D space is the volume V of a solid:

$$V = \frac{1}{3}\int_{\partial\Omega} \mathbf{r}\cdot\mathbf{n}d\partial\Omega \tag{3.27}$$

Where $\partial\Omega$ represents the whole boundary of the world_solid, let us now suppose that we have m patches in the solid in question, then $\partial\Omega$ could be expressed as $\partial\Omega = \bigcup_{i=1}^{m}\partial\Omega_i$ where $\partial\Omega_i$ is a patch, the volume is then:

$$V = \frac{1}{3}\sum_{i=1}^{m}\int_{\partial\Omega_i} \mathbf{r}\cdot\mathbf{n}_i d\partial\Omega_i \tag{3.28}$$

where $\mathbf{n}_i$ is the unit normal vector to the patch in question. Let us now consider the contribution of the i^{th} patch to the whole volume:

$$V_i = \frac{1}{3}\mathbf{n}_i\cdot\int_{\partial\Omega_i}\mathbf{r}d\partial\Omega_i = \frac{1}{3}\mathbf{n}_i\cdot\bar{\mathbf{r}}_i\Delta_i \tag{3.29}$$

where $\bar{\mathbf{r}}_i$ is the position vector of the centroid of the patch, and Δ_i its area that could be computed using formula 3.26 developed in the previous section.

3.5.3 Centroid

Both computations of the area and the volume of a solid need the computation of a centroid, this is accomplished easily, the position vector $\mathbf{c}$ of the centroid of an edge or a patch is readily computed knowing the position vectors of the n vertices:

$$\mathbf{c} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{p}_i \tag{3.30}$$

The above formula assumes that the mass is concentrated at the vertices of the different patches. In the present work, we have implemented the facilities for computing the three previously mentioned properties and stored them in the geometric fields of the world_solids upon user request. As an extension, we can foresee the need of computing the 1^{st} moment being the first-rank tensor and the 2^{nd} moment of inertia, this extension could be easily added using the Gauss Divergence theorem as for the 0^{th} moment. A more accurate computation of the centroid would then be $\mathbf{c} = \frac{\mathbf{I}_1}{I_0}$. The reader is referred to the work of Angeles [Angeles86b] if the extension is considered.

3.6 Summary

In this chapter, we discussed different solid modelling approaches. We chose the boundary representations method for the simulator and the reasons for this choice were given. We also gave examples of the implemented B-reps and examples of the sweep to B-reps conversion. We, then, introduced some graphics facilities and the computational aspect of the geometric properties of solids. In the next chapter, we consider the problem of motion of solids in 3-D space. We shall investigate the articulated motion of general manipulators, and shall also solve the forward and inverse kinematics of manipulators, keeping the generality as an important goal. Our main concern will be speed and storage optimization.

Chapter 4 Motion and Programming

4.1 Moving a Solid

Let us start by solving the problem of moving one point in 3-D along a certain trajectory. The trajectory should be decomposed into segments of position and orientation to facilitate the simulation of motion. Thus the motion problem is reduced to the motion of a point for a certain PD segment which could be fully defined by a matrix **A** for the orientation and a vector **u** for the displacement, as shown in figure 4.1.

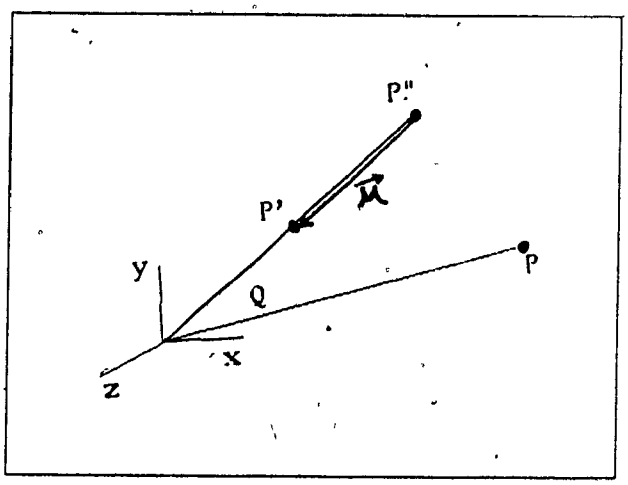


Figure 4.1 Point Motion.

We can now state that the motion of a point P to P' is defined as:

$$[\vec{OP'}]_w = [(\vec{OP''}) + \mathbf{u}]_w = [\mathbf{A}[\vec{OP}] + \mathbf{u}]_w \tag{4.1}$$

The motion is supposed to be defined in the world coordinate system, and O is its origin. For compatibility with our view_solids coordinate system, it is better to express the above equation in the view coordinate system. To this end, we assume that the rotational part of the view point transformation is defined by the matrix $\mathbf{V}$ and that the view point is O_e. Then we can state that:

$$\mathbf{A}_v = \mathbf{V}_v \mathbf{A}_w \mathbf{V}_v^{-1} = \mathbf{V}_v \mathbf{A}_w \mathbf{V}_v^T \tag{4.2a}$$

and

$$\mathbf{u}_v = \mathbf{V}_v \mathbf{u}_w \tag{4.2b}$$

Thus the new point P' is derived in the view coordinates:

$$[\vec{O_e P'}]_v \equiv P'_v = \mathbf{V}_v \mathbf{A}_w \mathbf{V}_v^T P_v + (\mathbf{V}_v \mathbf{A}_w \mathbf{V}_v^T - \mathbf{I})[\vec{OO_e}]_v + \mathbf{V}_v \mathbf{u}_w \tag{4.3}$$

where $\mathbf{I}$ is the identity matrix. Assuming that $\mathbf{M} = \mathbf{V}_v \mathbf{A}_w \mathbf{V}_v^T$ and $\mathbf{t} = (\mathbf{M} - \mathbf{I})[\vec{OO_e}]_v + \mathbf{V}_v \mathbf{u}_w$ then the motion could be expressed similarly as in the world coordinate system:

$$P'_v = \mathbf{M} P_v + \mathbf{t} \tag{4.4}$$

with $P'_v \equiv [\vec{O_e P'}]_v = (x', y', z')^T$ and $P_v = (x, y, z)^T$. The common way of producing the motion graphically is to apply equation 4.4 to the view_solid, apply clipping to the result and then prespective view transformation. All of the above operations should be applied at each sample of motion or each set $(\mathbf{A}, \mathbf{u})$ in the trajectory. A less time consuming alternative would be to apply the motion transformations in screen coordinates, this can be accomplished using the z coordinate of the points previously kept in the view coordinate system. We will now adapt equation (4.4) to the form of storage we have been using. Every point P of a solid is stored as $P_s = (a, b, z)^T$ where the z coordinates of P and P_s are the same. Under a set $(\mathbf{A}, \mathbf{u})$ describing a step of motion in the world coordinate system, the point P_s is transformed to $P'_s = (a', b', z')^T$; in order to simulate this motion we need to compute P'_s as a funtion of $\mathbf{A}$, $\mathbf{u}$ and P_s. Let us suppose that $\mathbf{M} = (\mathbf{u}_1^T, \mathbf{u}_2^T, \mathbf{u}_3^T)^T$ and $\mathbf{t} = (t_x, t_y, t_z)^T$, then:

$$P'_v = \begin{pmatrix} \mathbf{u}_1^T(\vec{O_e P}) + t_x \\ \mathbf{u}_2^T(\vec{O_e P}) + t_y \\ \mathbf{u}_3^T(\vec{O_e P}) + t_z \end{pmatrix} \tag{4.5}$$

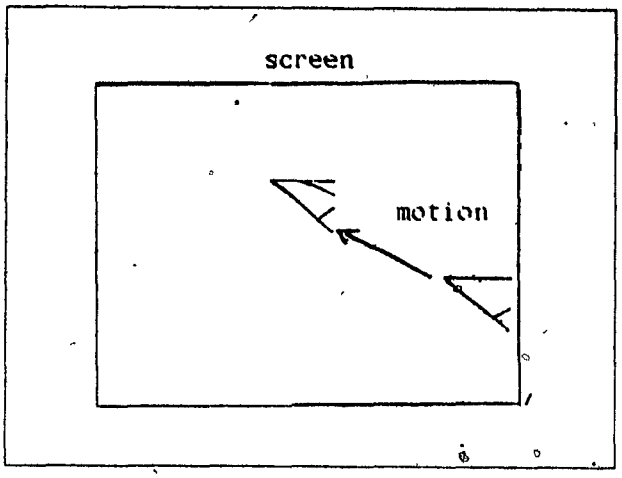


Figure 4.2 Cliping Effect on Motion.

From the previous graphics transformations we have

$$P_s = \begin{pmatrix} a \\ b \\ z \end{pmatrix} = \begin{pmatrix} \frac{D}{S}(\frac{x}{z})V_{sx} + V_{cx} \\ \frac{D}{S}(\frac{y}{z})V_{sy} + V_{cy} \\ z_v \end{pmatrix} \tag{4.6a}$$

and

$$P'_s = \begin{pmatrix} a' \\ b' \\ z' \end{pmatrix} = \begin{pmatrix} \frac{D}{S}(\frac{x'}{z'})V_{sx} + V_{cx} \\ \frac{D}{S}(\frac{y'}{z'})V_{sy} + V_{cy} \\ z'_v \end{pmatrix} \tag{4.6b}$$

Let us now define the vector:

$$\mathbf{w}^T = (\alpha, \beta, 1) \quad \text{where} \begin{cases} \alpha = \frac{S}{D}(\frac{a - V_{cx}}{V_{sx}}) \\ \beta = \frac{S}{D}(\frac{b - V_{cy}}{V_{sy}}) \end{cases}. \tag{4.7}$$

Then, we can derive that:

$$\begin{cases} a' = \frac{D}{S}V_{sx}(\frac{z(\mathbf{u}_1^T\mathbf{w}) + t_x}{z'}) + V_{cx} \\ b' = \frac{D}{S}V_{sy}(\frac{z(\mathbf{u}_2^T\mathbf{w}) + t_y}{z'}) + V_{cy} \\ z' = z(\mathbf{u}_3^T\mathbf{w}) + t_z \end{cases} \tag{4.8}$$

The above three equations determine the motion in screen coordinates and update the z coordinate of the point P under motion. The method we have used for the motion simulation accelerates the process but poses one disadvantage: The clipping and perspective view are done just once at the very beginning in order to produce P_s given P; thus, if a solid is clipped at its initial position it will always stay so, and if the clipped solid should appear on the screen under the effect of motion, it will appear as if it were still clipped, the effect is shown in figure 4.2.

This could be avoided by specifying graphics data which will keep the moving solids and the possibly grasped solids in the middle of the screen. Since time requirements are important in the simulator, the motion was implemented the way shown above accepting the clipping disadvantage.

A trajectory is sampled at certain intervals in order to produce motion, however, in practice the motion of a link of a manipulator is either rotational or translational, and hence equation (4.8) could be further simplified by taking care of the two types of motion seperately. Moreover, the motion or the trajectory of the motion is divided in such a way as to be mostly repetitive. As an example, assume that a revolute link of a robot is supposed to move by 65° from its initial position. The motion could then be considered as $6 \times 10° + 5°$ in order to make 7 frames of the whole trajectory. Thus, for the 6 first frames the matrix **A** of rotation is the same and describes 10° of rotation. The motion equations (4.8) could be written recursively for each step of motion by replacing (a', b', z') by $(a_{i+1}, b_{i+1}, z_{i+1})$ and (a, b, z) by (a_i, b_i, z_i). Since the repetitive aspect of the sampled motion, **M** and **t** remain unchanged $(n-1)$ frames on n frames.

In order to make the above derivations more general, we should include the case where the motion is given with respect to a coordinate system (i) different from the world coordinate system. Assuming that the motion is given by a set $(\mathbf{A}_i, \mathbf{u}_i)$, then the matrix **M** and the vector **t** of equation 4.4 become:

$$\mathbf{M} = \mathbf{V}_v(\mathbf{T}_i)_w \mathbf{A}_i (\mathbf{V}_v(\mathbf{T}_i)_w)^T \tag{4.9a}$$

$$\mathbf{t} = \mathbf{V}_v[((\mathbf{T}_i)_w \mathbf{A}_i (\mathbf{T}_i)_w^T - \mathbf{I})(O_i \vec{O}_e)_w + (\mathbf{T}_i)_w \mathbf{u}_i] \tag{4.9b}$$

Where $(\mathbf{T}_i)_w$ is the 3×3 matrix which transforms the world coordinate frame to the frame i of motion, and O_i the origin of the latter. The above formula is useful whenever the motion to be simulated is given in the joint space of a manipulator.

4.2 Manipulator Identification and Storage Structure

In the previous section we have discussed the implementation of motion and

the means adapted to speed it up as applied to one view_solid. In this section, we will introduce ways of constructing a whole manipulator and identifying it; we will also introduce the structure necessary for the storage. A manipulator is constructed geometrically with links which are represented in the datamodel as groups of view_solids identified by their names. Each link i has a name and points to a set of Hartenberg and Denavit parameters [Hartenberg64] associating the preceeding joint to the next one. A matrix $\mathbf{Q}_i$ and a vector $\mathbf{a}_i$ are hence formed and stored in each link's structure. Moreover, each link has a descriptive field which defines the type of its preceeding joint to be revolute, prismatic or others. This is done so that we can speed up the simulation by taking care only of the rotational or the translational aspect of the motion. For the time being, only prismatic or revolute joints can be simulated. The links are stored as linked lists, each of which defines a robot and is pointed to by a robot structure. Each robot structure has a name, some descriptive fields for information such as isotropic, serial and others. The robot structure points to a linked list of grasped soilds, generally just one and contains information about the end effector's position and orientation. Finally the robot structure includes information such as the nominal speed in the joint space. All the robot structures are themselves stored as linked lists, and as for the previous linked lists in the datamodel, some queries could be applied. The most important queries are creation, modification of the descriptive fields, removal and showing. The new data structure for articulated motion could then be defined as follows: The lowest entities are view_solids, then links and then robots. However, because of the multiple windows' use in the simulator, a higher level entity is needed to which robots and view_solids could be assigned, this entity was called a tool. Each tool is basically a window divided into a graphics subwindow for motion simulation and a text subwindow for time simulation, as will be explained later. Every tool is represented in the datamodel by a structure which has a name and points to robots and view_solids identified by their respective names. Each tool has a different process managing its windows in order to respond to queries such as close, expand, or quit. In each tool structure we also find the graphics data proper to its graphics subwindow. The last field in a tool structure points to the beginning of a linked list of structures called off_line motion structures which will

be introduced in the section concerning the types of simulation. The tool structures are themselves organized in a linked list manner and respond to the standard queries.

The above data structure was implemented keeping in mind the possibility of further extensions. There is, however, one problem remaining concerning the fact that the view_solids are stored using their screen coordinates and therefore the same view_solid cannot be used in different windows; if a view_solid is to be used twice or more it should have a different name in each tool. This confirms the fact that by gaining time we lose memory, having gained time by applying motion to the screen coordinates.

In order to clarify the process of constructing and identifying a robot, we will give an example for the Puma 260. We first create the base of the robot which can be approximated by a cylinder, as is the first link of the Puma. Once created, the two cylinders would be linked to the level of blocks in the datamodel. The second and third links of the manipulator could be modelled using an approximative surface and sweep it along the z axis from the x-y plane. At the end of the third link we have a cylindrical solid, which must be constructed. The last three links together could be modelled by a sphere connected to a smaller cylinder. Finally, the gripper is modelled using an approximative surface and applying a translational sweep on it. The sizes of the surfaces, cylinders, and spheres are found in the mechanical manual of the robot, which should be used precisely. All the constructed pieces are stored as blocks in the datamodel and should be moved to their appropriate place in space in order to construct the robot as an entity. The displacements to the appropriate place could be performed using the different pieces as instances and copying them to the appropriate places by giving the exact transformations needed to this end. At this stage, the world_solids could be built of the different blocks and the view_solids could be constructed with the option of specifying color. There is now no need to keep the blocks and the world_solids, so they can be cleared from the datamodel. The view_solids are then associated to a particular tool with the needed graphics data, and a result is shown in figure 4.3.

At this point, the robot is not yet identified, and neither are the links. For

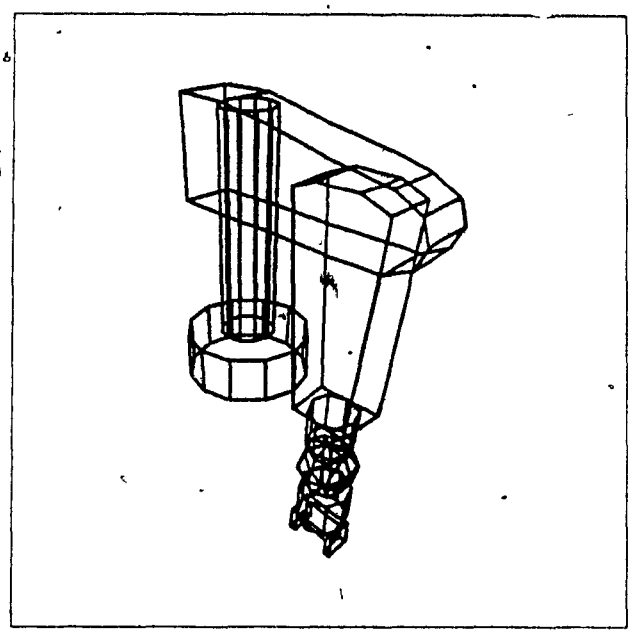


Figure 4.3 Puma 260

the purpose of identification, we create six links and assign a view_solid or a group of view_solids to each link. After doing so, we create a robot in the same tool and assign to it the different links. The descriptive fields for the robot and the links shoulu be specified. After the idendification process, the robot is ready to be moved around under specific commands. Since we have the total liberty in bringing the different blocks along in order to construct the robot, we should specify the initial values for the variables of each robot. The next section is devoted to articulated motion.

4.3 Articulated Motion

Generally, the motion could be described in two different spaces, namely, the joint space and the cartesian space. This section will introduce the two spaces and will also discuss ways of changing from one space to the other. If the motion is described in joint space, the displacements or rotations are known for each joint and therefore there are no complications for motion simulation. If the motion is described in the cartesian space, the final position and orientation of the end effector are usually known with respect to the world coordinate system; then the inverse kinematics would have to be solved in order to

find the motion in joint space. As an introduction, we shall provide some background on robots kinematics without getting into detail since this material could be found in a variety of books [Paul81a, Craig86]. We shall also give means for solving the inverse kinematics problem keeping in mind the generality of the simulator. This section will also present the different types of simulation that have been implemented in order to perform the animation. At the end of this section we will consider the problem of grasping a solid and the method developed to solve it.

4.3.1 Forward Kinematics

Let us consider a general robot architecture as shown in figure 4.4.

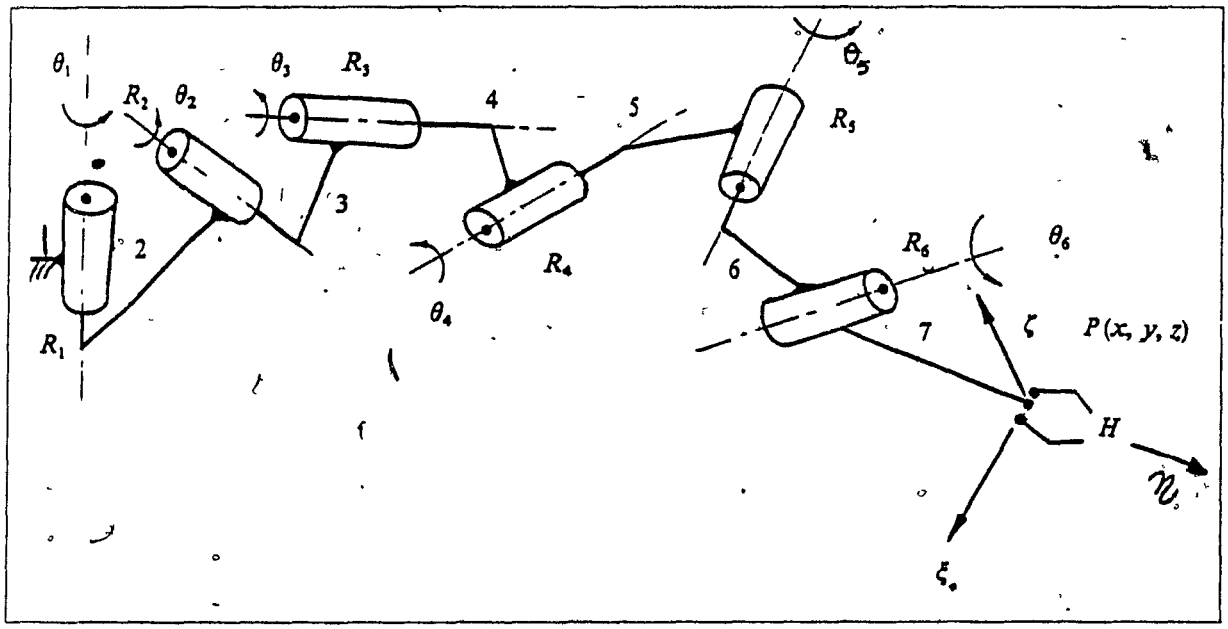


Figure 4.4 General Robot Architecture.

The manipulator is supposed to be constructed of n links, the first one being fixed. The types of manipulators that we are considering are serial and form an open chain, and therefore the numbering of the links is unambiguous. The closure equations to be used are those of Hartenberg and Denavit [Hartenberg64], where the architecture of the chain is determined by the set (a_i, b_i, α_i) with $i = 1 \cdots n$; each link i has a basis B_i attached to

it. $B_i = (O_i, X_i, Y_i, Z_i)$ which respects the conditions stated by Hartenberg and Denavit. The relative position of the basis B_{i+1} with respect to the basis B_i is given by a rotation matrix $\mathbf{Q}_i$ and a displacement vector $\mathbf{a}_i$.

$$\mathbf{Q}_i = \begin{pmatrix} c\theta_i & -s\theta_i c\alpha_i & s\theta_i s\alpha_i \\ s\theta_i & c\theta_i c\alpha_i & -c\theta_i s\alpha_i \\ 0 & s\alpha_i & c\alpha_i \end{pmatrix} \tag{4.10a}$$

$$\mathbf{a}_i = \begin{pmatrix} a_i c\theta_i \\ a_i s\theta_i \\ b_i \end{pmatrix} \tag{4.10b}$$

Where c stands for cos and s for sin. The orientation and position of the end effector are respectively determined by $\mathbf{Q}$ the orientation matrix with respect to the world coordinate system and $\mathbf{r}$ the vector relating the origin of B_1 to the end effector, in basis B_1. So we can write

$$\mathbf{Q}_1\mathbf{Q}_2\cdots\mathbf{Q}_n = \mathbf{Q} \tag{4.11a}$$

$$\sum_{i=1}^{n}[\mathbf{a}_i]_1 = \mathbf{r} \tag{4.11b}$$

In this treatment, we assume that the basis B_1 is the one of the world coordinate system. In forward kinematics, $\mathbf{Q}_i$ and $\mathbf{a}_i$ are known and the computation of $\mathbf{Q}$ and $\mathbf{r}$ is easily accomplished. If the motion is described in joint space, the simulation could be performed by sampling a curve which connects the initial and final positions of each joint. The forward kinematics computations would be used for updating the position and orientation of the end effector in the robot structure. However, the trajectory for a motion is usually supplied in the cartesian space, and therefore inverse kinematics would have to be solved.

4.3.2 Inverse Kinematics

Before solving the inverse kinematics problem for the whole trajectory, we should solve it for one particular position and orientation of the end effector. For a general purpose robot simulation we certainly need a general purpose inverse kinematics package, which will deliver the joint angle θ_i for revolute joints or the displacement b_i for the prismatic ones, given the position and orientation of the end effector in the world coordinate space. Closed

form solutions were developed and reported [Paul81a, Paul81b] for a particular simple class of manipulators. Another more general solution appeared later [Featherstone83], but it still deals with a sub-class of manipulators namely the wrist partitioned ones. Some other techniques appeared which are a mixture of closed form and numerical iterative solutions [Tsai84, Takano85]; these techniques apply to the class of six revolute (6R) manipulators. The inverse kinematics procedure deals with what is probably the most popular architecture, that of the six degree of freedom manipulators (revolute or prismatic joints). For simplicity, we will first develop the basic equations for a 6R manipulator and deal with prismatic joints subsequently. From equation 4.11a and 4.11b with $n = 6$, we have a system of twelve equations with six unknowns. These equations are dependent because of the fact that a rotation has the two properties

$$\mathbf{Q}\mathbf{Q}^T = \mathbf{I} = \mathbf{Q}^T\mathbf{Q} \tag{4.12a}$$

$$det(\mathbf{Q}) = 1 \tag{4.12b}$$

By taking the vector invariants and the trace of both sides in equation (4.11a) we can reduce the number of equations to seven as follows:

$$vect(\mathbf{Q}_1\mathbf{Q}_2\cdots\mathbf{Q}_6) - vect(\mathbf{Q}) = 0 \tag{4.13a}$$

$$tr(\mathbf{Q}_1\mathbf{Q}_2\cdots\mathbf{Q}_6) - tr(\mathbf{Q}) = 0 \tag{4.13b}$$

$$\sum_{i=1}^{6}[\mathbf{a}_i]_1 - \mathbf{r} = 0 \tag{4.13c}$$

Equations (4.13a) and (4.13b) are nonlinearly dependent because

$$\|vect(\mathbf{Q})\|^2 + [tr(\mathbf{Q}) - 1]^2/4 = 1 \tag{4.13d}$$

We cannot, however, keep just the vector invariant information without the trace, as that would generate confusion on the angles of rotation [Angeles85]. Let us consider the vector $\mathbf{f}$ in seven dimensional space. The components of this vector are the seven left sides of equations (4.13a), (4.13b) and (4.13c), so $\mathbf{f}$ is a function of $\theta_1 \cdots \theta_6$; for simplicity, $\mathbf{f}(\theta_1 \cdots \theta_6)$ is denoted $\mathbf{f}(\vec{\theta})$, where $\vec{\theta}$ is a six dimensional vector. The inverse kinematics problem can then be stated as follows: solve the equation

$$\mathbf{f}(\vec{\theta}) = 0 \tag{4.14}$$

The Jacobian of f is a 7x6 matrix known to be

$$\mathbf{J} = \frac{\partial \mathbf{f}}{\partial \vec{\theta}} \tag{4.15}$$

After differentiations, we find that:

$$\mathbf{J} = \begin{pmatrix} (tr(\mathbf{Q})\mathbf{I} - \mathbf{Q})\mathbf{A} \\ -2vect(\mathbf{Q})^T\mathbf{A} \\ \mathbf{B} \end{pmatrix} \tag{4.16a}$$

where **A** and **B** are 3x6 matrices defined as follows

$$\mathbf{A} = [\mathbf{e}, \mathbf{Q}_1\mathbf{e}, \cdots, \mathbf{Q}_1 \cdots \mathbf{Q}_5\mathbf{e}] \tag{4.16b}$$

$$\mathbf{B} = [\mathbf{e} \times \mathbf{r}_1, \mathbf{Q}_1\mathbf{e} \times \mathbf{r}_2, \cdots, \mathbf{Q}_1 \cdots \mathbf{Q}_5\mathbf{e} \times \mathbf{r}_6] \tag{4.16c}$$

with $\mathbf{e}^T = (0, 0, 1)$, and

$$\mathbf{r}_6 = \mathbf{a}_6, \quad \mathbf{r}_k = \mathbf{a}_k + \mathbf{Q}_k\mathbf{r}_{k+1} \tag{4.17}$$

Since we have more equations than variables, Newton-Gauss method seems to be the most appropriate for solving equation (4.14). Let us state the procedure : given an initial guess of the six dimensional vector $\vec{\theta}^0$, generate a sequence $\vec{\theta}^1, \vec{\theta}^2, \cdots, \vec{\theta}^k, \vec{\theta}^{k+1}, \cdots, \vec{\theta}^n$, such as to decrease the objective function z at each iteration, z^k is the objective function at iteration k. It is defined as

$$z^k = [\mathbf{f}^T(\vec{\theta}^k)\mathbf{f}(\vec{\theta}^k)]^{1/2} \tag{4.18}$$

at each iteration a correction vector $\Delta\vec{\theta}^k$ is applied to the vector $\vec{\theta}^k$, so that

$$\vec{\theta}^{k+1} = \vec{\theta}^k + \Delta\vec{\theta}^k \tag{4.19}$$

Using the Newton-Gauss method, this correction factor must satisfy the following equation :

$$\mathbf{J}(\vec{\theta}^k)\Delta\vec{\theta}^k = -\mathbf{f}(\vec{\theta}^k). \tag{4.20}$$

Thus

$$\Delta\vec{\theta}^k = -\mathbf{J}^I(\vec{\theta}^k)\mathbf{f}(\vec{\theta}^k) \quad \text{with} \quad \mathbf{J}^I = (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T \tag{4.21}$$

More detailed explanation about this could be found in [Angeles85]. We keep iterating until we reach a small prespecified objective function. At the end, the vector $\vec{\theta}$ holds the solution we are looking for. For prismatic joints the unknown is not θ_i anymore but b_i. This does not significantly change the procedure except that the Jacobian is modified so as to differentiate with respect to the variable b_i. Later for the resolution of the equation (4.14), the unknowns are modified according to the joint's type. The method we just described, namely, the Newton-Guass approach, has two points that we should discuss further. The first is the fact that the method garantees convergence to the solution only when the initial guess is already close enough to the solution. The second point involves the multiplicity of solutions and methods of finding them all. This second point is beyond the scope of the work we developed, and research is being conducted as a different project in order to solve the problem. The solution for the second point would be used especially for path planning and collision avoidance as future development for the simulator. The first point is of more interest for us now, and will be discussed next. Two problems could cause the non convergence of the algorithm

i : $\vec{\theta}^0$ is far away from the solution we are seeking.

ii : $\vec{\theta}^0$ is in a region where the jacobian J is ill conditioned.

In the latter case, the computation of $\Delta\vec{\theta}^k$ from equation 4.21 is very likely to be erroneous since $\mathbf{J}^T\mathbf{J}$ has a condition number which is the square of the condition number of J. The condition number is a measure of ill conditioning and is defined as follows:

$$\kappa(\mathbf{J}) = \|\mathbf{J}\|\|\mathbf{J}^{-1}\|, \quad \|\mathbf{J}\| = \sqrt{tr(\mathbf{J}\mathbf{W}\mathbf{J}^T)} \tag{4.22}$$

where W is a positive definite and usually diagonal matrix and should be chosen so that $\|\mathbf{I}\| = 1$. One way of making sure that the initial guess does not cause an ill conditioned jacobian is to first minimize $\kappa(\mathbf{J})$ and use as $\Delta\vec{\theta}^0$ the vector that satisfies the minimization. This minimization problem could usually be solved in a closed form manner, but otherwise it is an optimization problem. From the end effector's position and orientation hence caused

by $\Delta\vec{\theta}^0$, we use a continuation method towards the desired position and orientation. At each step in between, we use as an initial guess for the point P_i and orientation given by Q_i the vector $\Delta\vec{\theta}^{i-1}$ which is the solution for the inverse kinematic problem at step $i-1$. This way we assure the vicinity of the initial guess to the desired solution, and hence the two problems mentioned above for the Newton-Guass approach are solved using the minimization of $\kappa(J)$ as a start for the initial guess, and a continuation method to the desired end effector's position and orientation. The mathematical detail of the approach are not presented here, but are well explored in [Angeles87].

The numerical solution of the inverse kinematics problem was investigated in order to preserve the generality of the simulator in handling any type of robot. We should, however, notice that the approach developed above is applicable just to 6 degrees of freedom robots. Moreover, in practice most robots are designed so that they have a closed form solution for the inverse kinematics. Therefore, we kept the option of linking a closed form inverse kinematics package to the simulator; this could be useful for time saving because a closed form solution for a particular robot is usually faster than a general numerical solution. Also, the closed form methods provide the multiple solutions, so if the user is interested in comparing the multiple solutions or paths, the closed form solution is needed, at least for the moment. As an example, a closed form solution for the Puma 6-R robot was developed and resides in the simulator, so if a Puma is involved in the simultion, the closed form is interrogated for the inverse kinematics instead of the numerical solution. The closed form solution for the Puma is not presented here since it is well known and could be found in many references [Paul81a, Lloyd85]. The Puma is a manipulator of 3 singularities and hence $2^3 = 8$ real solutions exist for an inverse kinematic problem. The user has the option of specifying the solution he is looking for; the eight solutions are selected using three of the following possibilities: right-left, up-down, and flip-(no-flip); the eight possibilities are shown in [Lloyd85].

4.4 Motion Simulation

In the previous section, we discussed the problem of forward and inverse kinematics At this stage, a motion described in cartesian space could be transformed to a motion in joint space; we now develop means of simulating joint space motion. Interactive and real time simulation is not easy to accomplish and may even be impossible for a general purpose simulator depending on the complexity of the workcell in which we are interested. However, time scaling can be used and the simulation can occur in "slow motion". Here a clock is shown in the text subwindow of the tool to indicate the time it would take if the program we are simulating were being run on the real robot instead of the simulator. The simulator was designed to run in three different modes as selected by the user; there are two interactive modes and one off-line mode which will be explained later in this section. In this section, we are not showing the result of simulating a whole robot program, which will be the subject of a subsequent section. Instead, we suppose that the user wants to move relatively the n joints of a selected robot by a vector $\Delta\vec{\theta}$. If a joint is prismatic then its corresponding $\Delta\theta_i$ is replaced by Δb_i.

4.4.1 Interactive Simulation

In this context, interactive means that the simulation is done at the time when the command is issued, so the computations are performed at the same time as the simulation. This type of simulation will be reffered to as on-line also. Depending on the application, the trajectory followed by the end effector may or may not be of interest. For example, in pick and place operations, only the final position and orientation of the end effector are of importance. However, in path planning and many other applications, the trajectory followed by the end effector and the different links is of primary interest. In the interactive mode, we developed two types of simulation, the first is called joint-by-joint simulation and could be used for pick and place and other operations where the trajectory is not important. The second type is called the path simulation and is used for applications where the path followed is important. The second type of simulation is more time

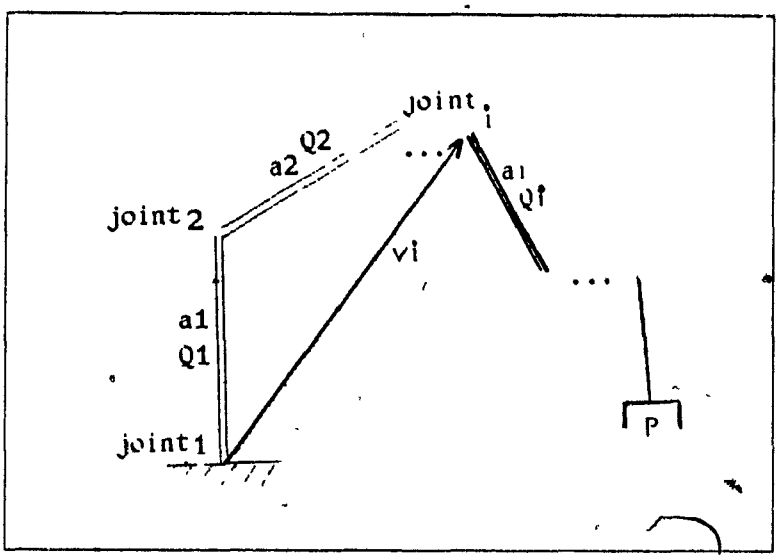


Figure 4.5 Joint's Motion

consuming because of computational aspects as will be shown later, however, it is closer to reality.

4.4.1.1 Joint-by-joint Simulation

Joint-by-joint simulation is a type of simulation where the joints are moved one by one respectively. In the joint-by-joint mode, the required parameters are the tool's name, the robot's name, the vector of the total relative motion $\Delta\vec{\theta}$ and the step size of motion sampling. First a verification of the existence of the tool and the robot in the tool in question is performed, then the validity of the other given data is checked. Let us denote by $\delta\theta$ the step of motion, and by $\vec{\theta}$ the vector of variables for the robot to be moved. In this treatment of joint motion, we assume that the joint variables increase linearly from the start position to the end position. This is an arbitrary choice, and in practice, the joint motion could be following a spline[Späth78], a 3-4-5 polynomial[Derby83], or a linear motion as in RCCL [Hayward84]. Now we shall state the problem in a manner which will allow the material of section 1 of this chapter to apply towards its solution. First, we make a list of moving view_solids for each joint to be moved. The list is made of the links following the joint in question plus the grasped view_solids if any. Figure 4.5 shows an example where joint i is the one to be moved.

The motion is defined for each step by the set $(\mathbf{A}_i(\delta\theta), \mathbf{u}_i(\delta\theta))$: in the present implementation, we have either $\mathbf{A}_i(\delta\theta) = \mathbf{I}$ or $\mathbf{u}_i(\delta\theta) = \mathbf{0}$ depending on whether the joint is prismatic or revolute, and hence the computation is reduced. The case where the motion is described in a reference frame other than the world coordinate frame was discussed earlier and equations 4.9*a* and 4.9*b* should be used in this case. For the motion of joint i, we need to perform the computations for $(\mathbf{T}_i)_w$ and O_i of equations 4.9*a* and 4.9*b*, which are computed the following way:

$$(\mathbf{T}_1)_w = \mathbf{I}, \quad (\mathbf{T}_i)_w = (\mathbf{T}_{i-1})_w \mathbf{Q}_{i-1} \tag{4.23a}$$

$$O_1 \equiv \vec{\mathbf{0}}, \quad O_i \equiv \mathbf{v}_i = \mathbf{v}_{i-1} + \mathbf{a}_{i-1} \tag{4.23b}$$

Now we shall state the motion set $(\mathbf{A}_i, \mathbf{u}_i)$ in the i^{th} frame. Respecting the Hartenberg and Denavit choice, $\mathbf{A}_i(\delta\theta)$ is a step of a rotational motion with respect to the z axis of the i^{th} frame and hence is defined as:

$$\mathbf{A}_i(\delta\theta) = \begin{pmatrix} c(\delta\theta) & -s(\delta\theta) & 0 \\ s(\delta\theta) & c(\delta\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{4.24}$$

The vector $\mathbf{u}_i(\delta\theta)$ is a step of a translational motion with respect to the z axis of the i^{th} frame and is defined as:

$$\mathbf{u}_i^T = (0, 0, \delta\theta) \tag{4.25}$$

For translational motion δb replaces $\delta\theta$, which is a step for the variable of a prismatic joint. Before the motion of the i^{th} joint, the previous joints have been moved respectively and hence the matrices $\mathbf{Q}_j$ and the vectors $\mathbf{a}_j$, with $1 \leq j \leq i-1$, remain unchanged. Thus the computation of 4.23*a* and 4.23*b* is needed just once for each joint to be moved. After the computations for 4.23*a* and 4.23*b*, M and t are computed from equations 4.9*a* and 4.9*b* with the simplification introduced by the fact that the joint is either translational or rotational. At this stage, the motion equations of section 1 are applied on each view_solid which is in the linked list of moving view_solids of joint i. The procedure is repeated $(m_i + 1)$ times for each joint i where:

$$m_i = \lfloor \frac{\Delta\theta_i}{\delta\theta} \rfloor \tag{4.26}$$

The last frame of motion is produced by $(\Delta\theta_i - m_i\delta\theta)$, completing the total relative motion.

The whole procedure mentioned above is repeated for each joint of the robot starting from the first joint to the last one. After applying the motion to joint i, the variable θ_i is updated so that it could be used correctly for computing $(\mathrm{T}_{i+1})_w$ and O_{i+1}. At the end of the motion, the end effector's position and orientation fields in the robot's structure are updated. In order to display the motion, the screen is updated once every step of motion for each joint. The computations of motion are applied to the moving view solids only, so for each frame the total scene is made of the union of the moving view solids and the static view_solids.

4.4.1.2 Path Simulation

Path simulation is used when we are interested in the path followed by the manipulator during a certain task. In this particular implementation, we assume that the joint variables attain their final values linearly from their start positions. If the changes in joint space are not linear, for example following a spline,this assumption produces an error. However, if the control algorithms for the joints controllers are known, they could be incorporated into the simulator system. Figure 4.6 shows an example of motion in joint space.

The information needed in order to perform this type of motion is the tool's name, the robot's name, and the total relative motion $\Delta\theta$ and $\delta\theta$. This type of motion is closer to reality than the previous one, and the motion sampling is done based on time intervals δT. As was mentioned earlier, the joint speed is specified by the user and is stored in the robot's structure; it is used as being the speed of the joint which has to move the most in order to accomplish the whole motion command.

$$\delta T = \frac{\delta\theta}{\dot{\theta}}, \quad \text{and} \quad \dot{\theta} = \frac{(\theta_F - \theta_I)_{max}}{T} \tag{4.27}$$

In this type of simulation, the step size of the motion is different from one joint to another, let us denote by $\delta\theta^T = (\delta\theta_1, \cdots, \delta\theta_i, \cdots, \delta\theta_n)$ the vector of steps of motion for the different

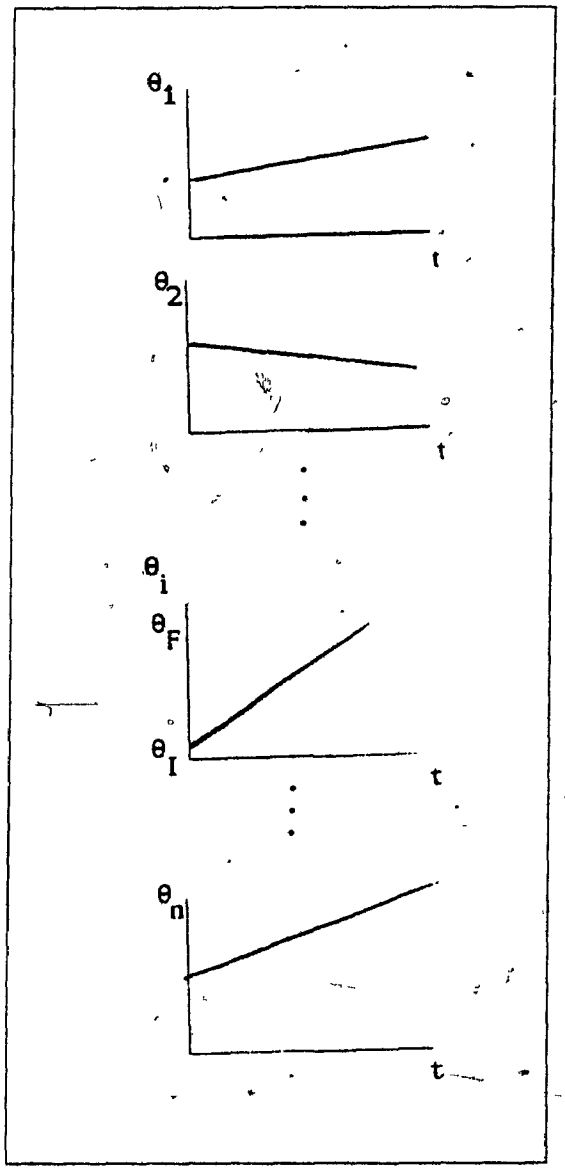


Figure 4.6 Joint-time Relations.

joints, where:

$$\delta\theta_i = \frac{\Delta\theta_i}{T}\delta T \tag{4.28}$$

This motion is not applied joint by joint, but instead is applied in terms of time. Assuming that $T = m\delta T + \Delta T$, we first simulate m times the motion caused by the joints altogether, then we add the motion caused during ΔT. For the total relative motion we then have $(m+1)$ frames shown on the screen. For each step of motion, a list of moving view_solids is built for each joint and the computations of equations 4.23*a* and 4.23*b* are performed. However the display is not updated until the step motion is computed for all n joints.

Moreover, the re-computation of equations 4.23a and 4.23b is required for each step of motion, because if joint i moves by $\delta\theta_i$ then, in order to apply the motion caused by the $(i+1)^{st}$ joint, we should update $\mathbf{Q}_i$ or $\mathbf{a}_i$ by incrementing θ_i by $\delta\theta_i$. Thus, for each joint i the matrix $(\mathbf{T}_i)_w$ or the vector $(\mathbf{v}_i)_w$ is updated $(m+1)$ times. There is, however, no point of updating the end effector's position and orientation at every step, so this is done at the very end of the total relative motion. This approach of motion simulation is computationally lengthy because of the frequent updating, but is closer to reality and is the approach to be used in approaximating the trajectory followed by the end effector.

4.4.2 Play-back Simulation

Play-back simulation is meant to be used when the solution time of the simulation becomes excessive and the moiton display becomes degraded. Here the sequence of image displays are stored in memory, permitting then to be viwed or "played back" in a faster sequence. The word off-line will sometimes be used for play-back. In this section, we explain the data structure for off-line or play-back simulation and its computational aspects. Each tool structure, as mentioned before, points to the beginning of a linked list of off-line motion structures. In its turn, an off-line motion structure points to two linked lists of view_solids, namely, moving view_solids and static view_solids. An off-line motion structure has two fields. The first is a descriptive field for determining when there is a change in the linked lists of moving and stable view_solids, such as when the robot grasps an object during a certain task. The second field contains the value of the time needed to perform the motion if the motion command were given to a real robot instead of a simulated robot, this time is referred to as simulated time. The simulated time is displayed in the text subwindow of the tool of motion.

When an off-line relative motion command is issued, the same computations as in the case of on-line or interactive path simulation approach are performed, and the result is stored in the view_solids fields of the off-line motion structures. The split of view_solids into moving and static is done so that if there is no change in the change field, the static

solids of the $(i+1)^{st}$ motion frame are just the same as those in the i^{th} frame, and hence no further memory is needed. The change field could be set in many circumstances such as when robot A stops moving after a certain motion command, or when robot B starts moving in the same tool. The change field was made available for saving time and memory in not duplicating the same stable solids between subsequent frames. All of the off-line motion structures are stored as linked lists. For some long relative motion commands, long programs or even for complicated scenes, the memory needed for storing the different frames of motion may exceed the amount of memory that could be allocated for one process In this case, the process opens a disc file and saves the needed information. After computing and saving the frames of motion, the user can request a simulation command in one specific tool, and then the graphic display is updated and the motion is simulated. For each frame of motion, the vertices of the moving view_solids are the ones previously computed and stored, and thus the time needed for this type of simulation is basically the time needed for the displaying plus the time needed for loading the precomputed information from external storage. Using this particular approach, a real time simulation or animation is possible

4.4.3 Discussion

The above three types of simulation have been developed for different purposes. The last type, namely the off-line simulation, is interesting when an exact duplication of a motion command is wanted. However, an interactive simulation is sometimes needed for more interactive programming , and we therefore developed the interactive joint-by-joint and path simulation approaches. In order to assess the interactive approaches, we developed a simulated clock which shows the time that the action would have taken if the execution was on a real robot. For comparison, a real clock is also shown on the screen We will now discuss the simulated time implementation. Let us start with the case of path simulation aproach, where the time needed for each display update or step motion is known from equation 4.27. Thus the time display in the text subwindow is just incremented by δT for each step motion. In the joint-by-joint simulation approach, the display is updated $(\sum_{j=1}^{n} m_i + 1)$ times for each motion command, and the time is incremented during the

motion of one joint which we chose to be the joint that has to move the fastest for the whole motion. For the off-line simulation approach, the time is updated in the same manner as in the path simulation approach, but is stored in the frames of motion instead. In order to make the off-line approach closer to reality in terms of time, a wait statement is included in the program at each frame. The wait interval should be shorter than δT due to the time taken to update the display, however this is negligible.

4.5 Grasping Solids

When the view_solids were introduced, we mentioned that their structures contain three fields that were used for the grasping facilities. The first field is descriptive and is entered by the user in order to tell if the view_solid in question is movable or not. For example, a cube on a table can be movable whereas the table itself or one link of a particular robot cannot be. If the view_solid is movable, then as soon as it is created a certain bounding parallelepiped is computed for it and its dimensions are stored in the view_solid's structure. Moreover, the view_solid's centroid is computed and a coordinate frame is attached to it, as shown in figure 4.7.

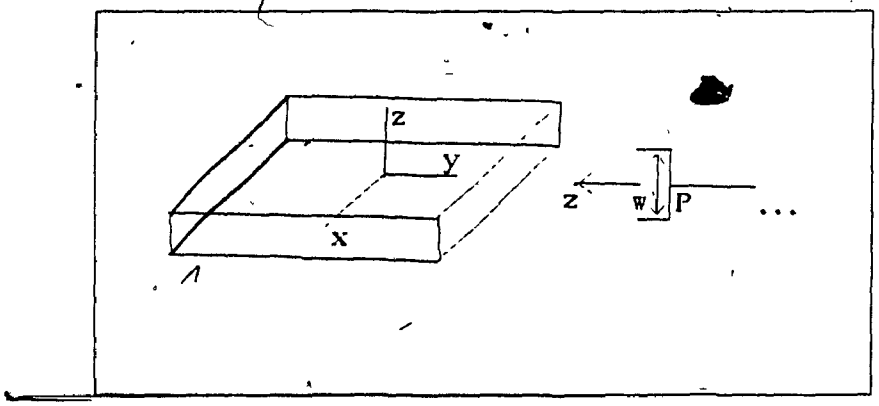


Figure 4.7 Grasping a Solid

The frame in question is stored in a field of the structure of the view_solid, and we should note that the frame is expressed with respect to the world coordinate system. The grasping routines are called when a command to close the gripper of certain

robot is entered. The data needed in order to perform the 'close_gripper' command are: Tool's name, robot's name, maximum distance allowed in order to consider the gripper close enough to the solid to be grasped, the maximum angle of deviation between the end effector's z axis and the x, y or z axis of the bounding parallelepiped, and finally a specification of the type of simulation being used: Joint-by-joint, path, or off-line simulation. As for all commands, a verification of the validity of the given data is performed first, then some tests are conducted in order to know if a solid is indeed being grasped. The tests are performed on the movable view_solids only. The first test is to know if the z axis of the end effector is parallel to any of the axes of the frame attached to the view_solid being tested. If the test succeeds within a certain margin fixed by the user, namely the angle of deviation, then the possibility of grasping the view_solid by the gripper along a certain direction is verified. This test is a verification to determine if $\Delta y \leq w$, as shown in figure 4.7. If the two tests are positive then a distance test is performed to know if the point P of the end effector is close enough to be capable of grasping the solid being tested. The distance test is approximative within a certain tolerated distance specified by the user. If one movable view_solid among all the movable view_solids in the workcell satisfies the tests of grasping, then it is declared grasped and is linked to the linked list of grasped solids of the robot which received the close_gripper command. If a robot that is grasping a view_solid is ordered to move, then the view_solid in question is also moved accordingly, and moreover, the frame attached to the grasped view_solid at its centroid is updated at the end of each motion command that the robot in question receives. In the case of off-line motion we should note that the view_solids that we test for grasping could be in the lists of both stable or moving view_solids. The view_solid to grasp could have previously been moving, such as in the case where one robot hands a solid to another one; this particular example will be discussed in the next chapter, under results.

4.6 Program Development

We have by now developed means for handling commands such as 'move_joints_relative' and 'close_gripper', and the input data required for both types of commands was mentioned

earlier. In this section, we introduce some more commands based on the ones mentioned above, then we introduce means of interaction with the simulator, and we also provide an example of a program to be simulated. We should mention here that all commands can be applied in three different modes of simulation as previously mentioned.

4.6.1 Simulation Commands

4.6.1.1 Motion Commands

The only motion command that we have developed at this stage is 'move_joints_relative'; the execution of this command has been explained previously. In this section we present some other commands that are available in the simulator. The absolute motion in joints space is easily transformed to relative motion by taking the difference $(\vec{\theta}_{final} - \vec{\theta}_{initial})$, where $\vec{\theta}_{final}$ is the desired vector of absolute motion in joint space and $\vec{\theta}_{initial}$ is the vector of the robot's variables found in the robot's structure. When the motion is described in cartesian space, the inverse kinematics package is used to give the absolute motion in joint space which, in its turn, is transformed to relative motion in joint space and executed as explained before depending on the type of simulation being used. A useful command in various robotics applications is 'move_straight_line', which guarantees a straight motion of the end effector. In the present implementation of the straight line motion, we suppose that the orientation of the end effector stays unchanged from the initial to the final configuration; the position of the end effector is sampled with some prefixed step. At each step, the inverse kinematics is solved and the motion is simulated using the previous development of relative motion in joint space. If a Puma robot is under simulation, the closed form inverse kinematics is solved and thus it is the user's responsibility to specify the desired configuration among the eight possibilities. If no configuration is chosen, then a default configuration is used.

At this stage of development of the simulator, we have four motion commands, two in joint space, 'move_joints_relative' and 'move_joints_absolute', and two motion commands described in cartesian space, 'move_end_effector' and 'move_straight_line'. For the

'move_end_effector' command, the joints move linearly from their initial to their final positions.

For the 'move_staright_line' command, however, a sampling of a straight line is done in cartesian space in order to assure an approximative linear motion in cartesian space. Between the samples, the motion is linear in joint space and thus not necessarily in cartesian space, depending on the type of robot. This error is negligible since the sampling frequency can be increased to more closely approximate a straight line motion in cartesian space.

4.6.1.2 Other Commands

The only command other than motion commands that we have discussed this far is 'close_gripper', which is explained in the section on "Grasping Solids". Two other commands must be added here. The first is called 'open gripper' and it is easily served by clearing the field of grasped solids in the robot structure of the robot which receives this command. If a motion command is sent to the same robot after the 'open_gripper' command, the solid which was grasped before would no longer move with the robot. The second command is 'robot_speed', which changes the speed field in a robot structure. This affects the simulated time, the step motion $\delta\theta$ in the path simulation, and the wait duration in the off-line simulation. In chapter 5, we will see through examples the use of all the commands that have been introduced. We should note, however, that the commands syntax is not shown in detail in this section, and that each command requires data such as tool's name, robot's name, and others.

4.6.2 Programming the Simulator

Once the workcell is created and the manipulators identified and initialized, the time comes to program the workcell. The available simulator's commands are seven and have been introduced in a previous section. All of the commands could be entered

interactively directly using the key tree matcher which simplifies the debugging of the tasks. For example, assume that we want to order a certain robot to grasp a certain solid without knowing the exact position and direction needed for the end effector in order to do so. This could be be tried directly on the simulator until the action succeeds and the solid is successfully grasped. Then we can read the end effector's position and orientation or even the value of the joint variables so that they can be used later in the robot program. This method is equivalent to using a teaching pendant when we program a real robot. Moreover, some parameters such as speed, step motion or grasping parameters could be modified interactively for comparison and time considerations. This method of trying commands directly on the simulator is useful and comes usually at the first stage of developing a program. The second stage involves the ability to write a robot program using a certain syntax or predefined language, and be able to observe the behaviour of the whole robotic workcell using the simulator. This process is explained in the next paragraph, omitting however, the programming detail.

4.6.2.1 Interaction with the Simulator

Our purpose is to develop a program and then apply it to the simulator. A communication package is therefore needed to assure the interaction between the application program and the simulator. From the simulator's end, just one command is needed, namely the command to interpret a program in a given tool using one specified simulation mode. When the simulator's process receives the interpret command it goes to a state of a 'receiver' from a certain communication channel, and the simulator is thus ready to execute any transmitted command from the communication package through the channel. At the sending end, once a program is written with simulator commands in it, every time that one of these commands is encountered, this process transmits the necessary information to the simualtor through the the same channel as the simulator, as shown in figure 4.8.

The child process created as a transmitter terminates upon reception of a signal from the simulator. This signal could be an acknowledge signal or an error signal, in which

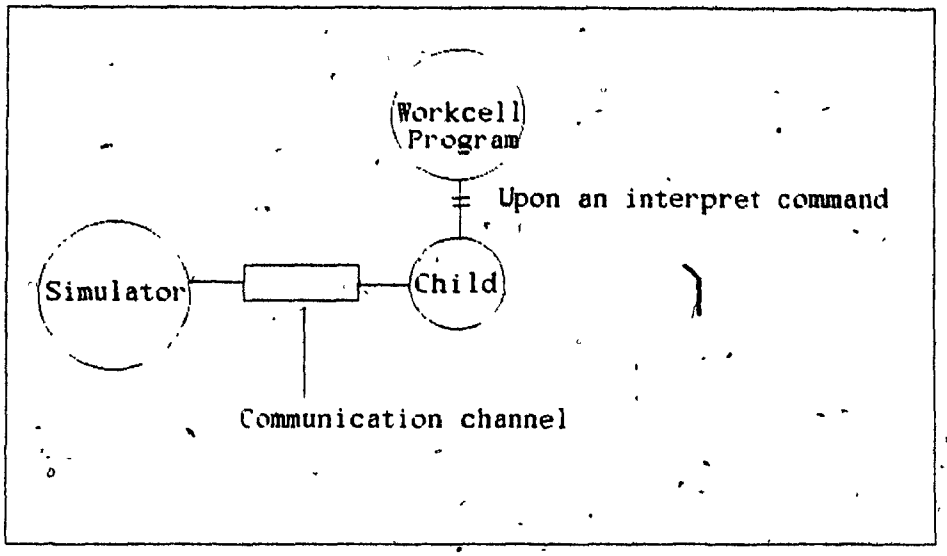


Figure 4.8 Interaction Between the Application Program and the Simulator

case the program being simulated is also terminated, and the type of error is displayed at the simulator. The workcell application program including the simulator's commands is written in the 'C' programming language. However it may be written in any language and compiled using the proper compiler since the program itself is not interpreted line by line. For the present implementation 'C' was chosen because the process which establishes the communication protocol and the simulator itself are written in 'C'. If the program is written in another language, communication modules should be adapted according to differences in syntax. The simulator's commands have been assigned a particular syntax. If the user wishes to use a different syntax, it is easy to make the required translation, since the simulator's commands in the application program are function names which create the communication processes internally. At this stage of the simulator, the application programs reside on the same machine as the simulator itself, however this is not necessary. The communication processes were developed keeping in mind the future extention of having the simulator and the application program running on different machines.

4.7 Summary

In this chapter, we presented the motion commands and the way they are.

treated for the simulation. The inverse kinematics problem was also introduced and solved to a certain extent for a general robot. Moreover, a closed form solution for the inverse kinematics of the Puma robot was implemented. We also discussed some other commands which were judged necessary for a robotic work simulation. Finally, we tackled the problem of interaction with the simulator and exposed both ways of interaction namely directly through the simulator or through an independent program which is the result we are seeking. At this stage, we are capable of writing a robot program and simulating it as long as we follow a certain predefined syntax. The next chapter presents results, where we will simulate and discuss different programs in various workcells.

Chapter 5 Results

This chapter presents simulation results. In order to obtain these results, we had to go through three stages. The first stage was the creation of 2-D surfaces necessary to model the workcell, as was explained in chapter 2. After the creation of the correct 2-D surfaces, the techniques developed for solid modelling were used to represent the components of the workcell, as per chapter 3. Then the manipulators' identification and initialization took place. The simulation commands can, at this stage, be applied in two ways, from the simulator directly or through an external program. During the development phase of a robot program, the commands are usually entered interactively using the key tree matcher. Once the debugging is complete, the programmer can write the robot program successfully. The robot program sends simulation commands with the required data to the simulator through a communication channel, and the simulation can be performed in three different modes as explained in chapter 4. The simulated time appears on the screen upon request from the user, which helps assess the workcell performance. This chapter is devoted to three examples of simulation. The first example involves only one manipulator in a workcell and will deal with joint space motion. The second example involves one manipulator and one movable object and will deal with motion in cartesian space and the problem of grasping. The third and last example involves two manipulators and will deal with most of the problems that can be solved using the simulator. All three examples can run in the three modes of simulation. On the display screen, the motion step $\delta\theta$ is specified by the user; the smaller $\delta\theta$ is made, the more continuous is the motion. For illustrations, some frames are presented but obviously not all frames are given in figures.

5.1 Experiments

5.1.1 Example 1:One Manipulator

Let us assume that a workcell involves a Puma 260 manipulator, which is 6 degrees of freedom and six revolute. Let us assume also that we want to order the manipulator to move, and that the commands of motion are given in joint space. Before trying these commands on the real manipulator, it is prefered to try them on a simulation of the manipulator. First, we should construct graphically the robot in question and supply all the required data for simulation such as the robot's identification, speed, grasping data, initial values of the Hartenberg and Denavit variables of the robot, and others. We can then write a small program, as shown in figure 5.1, to move the joints of the Puma.

The program is basically constructed of three motion commands: The first command moves the robot relatively from its initial position by a vector of Hartenberg and Denavit variables $(-70, -10, 10, 30, -20, -10)$. The second motion command is also in joint space but is absolute and brings the robot to the state $(-60, 0, -70, 0, 90, 30)$. The last motion command brings back the robot to its initial position which is $(0, 0, -90, 0, 90, 0)$ in joint space. In practice, the program is interpreted and the result is shown smoothly on the display screen. For the sake of clarity, we show three states of the Puma robot. The first is shown in figure 5.2, and presents the initial position of the robot. Figure 5.3 presents the state of the robot at the end of the execution of the relative motion command, and figure 5.4 presents the state of the robot after completion of the absolute motion command. The final state of the robot is the same as in figure 5.2.

5.1.2 Example 2: One manipulator and one solid to move

The second example involves the same robot as in example 1, one small movable cube, and two non movable stages on the table. This example is a particular case of a pick and place task, where the solid to be moved is the cube. Let us suppose that we

```
include "system_comm.h" /*definition and declaration files for the communication link*/
include "types.h"
define NUM 6 /*number of joints*/
/*-------------------------------------------------------*/
main(argc, argv)
int argc;
char *argv[];
[
float v[NUM];
init_comm(argv); /* establish the communication link */
speed("Puma", 0.3); /* 0.3 rd/s for the joint which moves the fastest*/
affect_var(v, -70, -10, 10, 30, -20, -10);
if ( !move_joints_rel("Puma", v) ) /* relative motion in joint space to a robot called Puma in the simulator */
[
clean(); /* if any error, close the communication link and exit the program */
exit(1);/*the error type is shown at the simulator's end */
]
affect_var(v, -60, 0, -70, 0, 90, 30);
if ( !move_joints_abs("Puma", v) ) /* absolute motion for Puma */
[
clean();
exit(2);
]
affect_var(v, 0, 0, -90, 0, 90, 0); if ( !move_joints_abs("Puma", v) ) /*back to "start" position and orientation
*/
[
clean();
exit(3);
]
clean(); /* terminate successfully the transmission of simulator's commands */
]
```

Figure 5.1 Program 1.

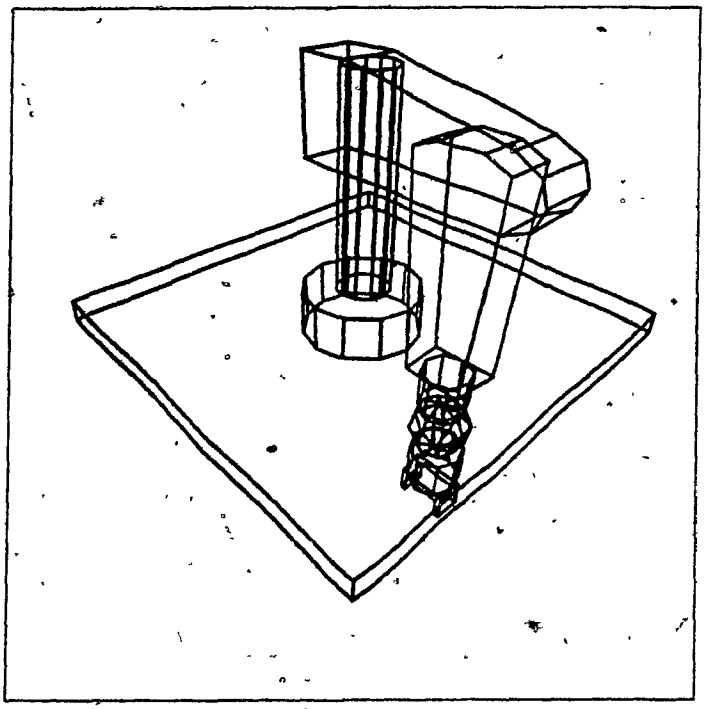


Figure 5.2 Initial Position of the Puma. $v = (0, 0, -90, 0, 90, 0)$.

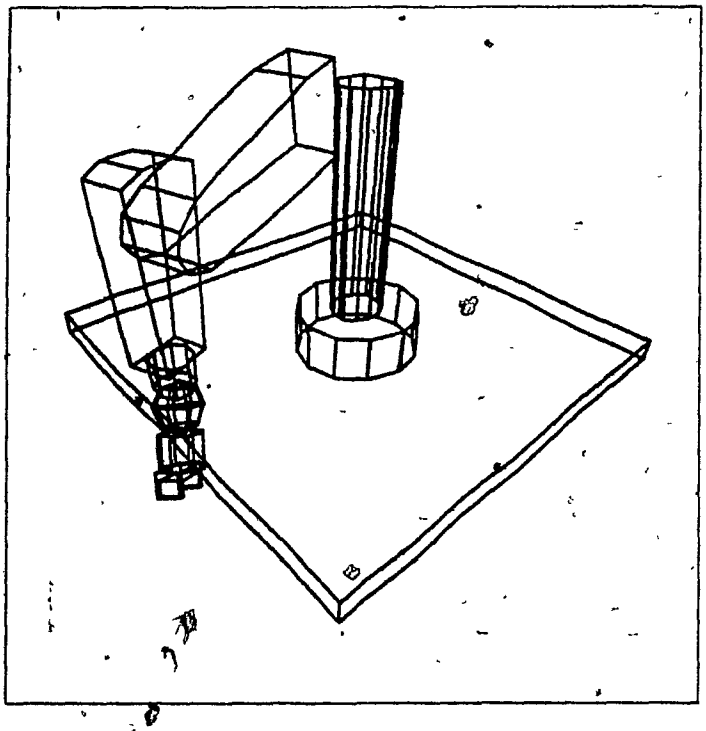


Figure 5.3 Relative Motion by $\Delta v = (-70, -10, 10, 30, -20, -10)$.

know the position and orientation of the cube, the first task would then be to order the robot to move to a configuration from which the end effector can grasp the cube. The

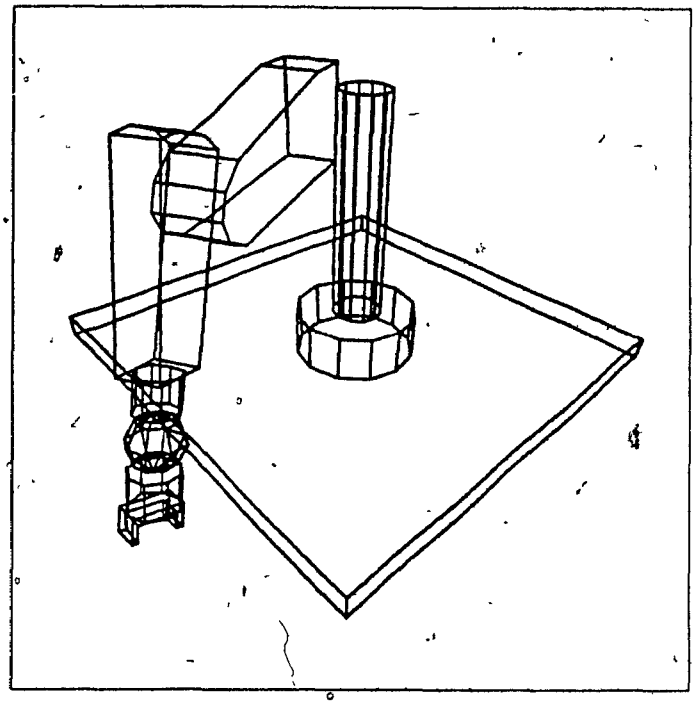


Figure 5.4 Absolute Motion to $v = (-60, 0, -70, 0, 90, 30)$.

orientation is such that the orientation constraints of grasping are satisfied, as explained earlier. Once the robot is at the desired configuration, which could be tested directly from the key tree matcher as a replacement of a teaching pendant, the robot is required to close its gripper and hence grasp the cube. Let us now suppose that we want the cube to be placed on one stage. Knowing the position and orientation of the parallelepiped bounding the stage, a motion command is issued to the robot to place its end effector over the stage, and the gripper is opened. In order to terminate the whole program an absolute motion in joint space is ordered to bring the robot back to its initial position. The first two motion commands are in cartesian space and hence the inverse kinematics problem is solved using the closed form solution of the Puma robot, if another robot was under simulation, the Newton-Guass or general inverse kinematics solution would have been used as explained in the previous chapter. A pseudocode of the program to perform the whole task is shown in figure 5.5. The results are shown in figure 5.6,5.7,5.8 and 5.9. The reader may notice some discrepancy in the robot model for some configurations such as joint 2 in figure 5.7. This is due to the approximate measures used for modelling the Puma 260 since an accurate

```
/* error recovery is not shown, it should exist after each simulator command*/
initialization();
move_end_effector (Puma, C1);/* C1: Configuration suitable fog grasping the cube*/
close_gripper (Puma),
move_end_effector (Puma, C2);/* C2: Configuration above stage_1 */.
open_gripper (Puma), /*release the cube on stage_1*/
move_joints_abs (Puma, V1); /*V1  The initial configuration of Puma */
end;
```

Figure 5.5 Pseudocode of Program 2

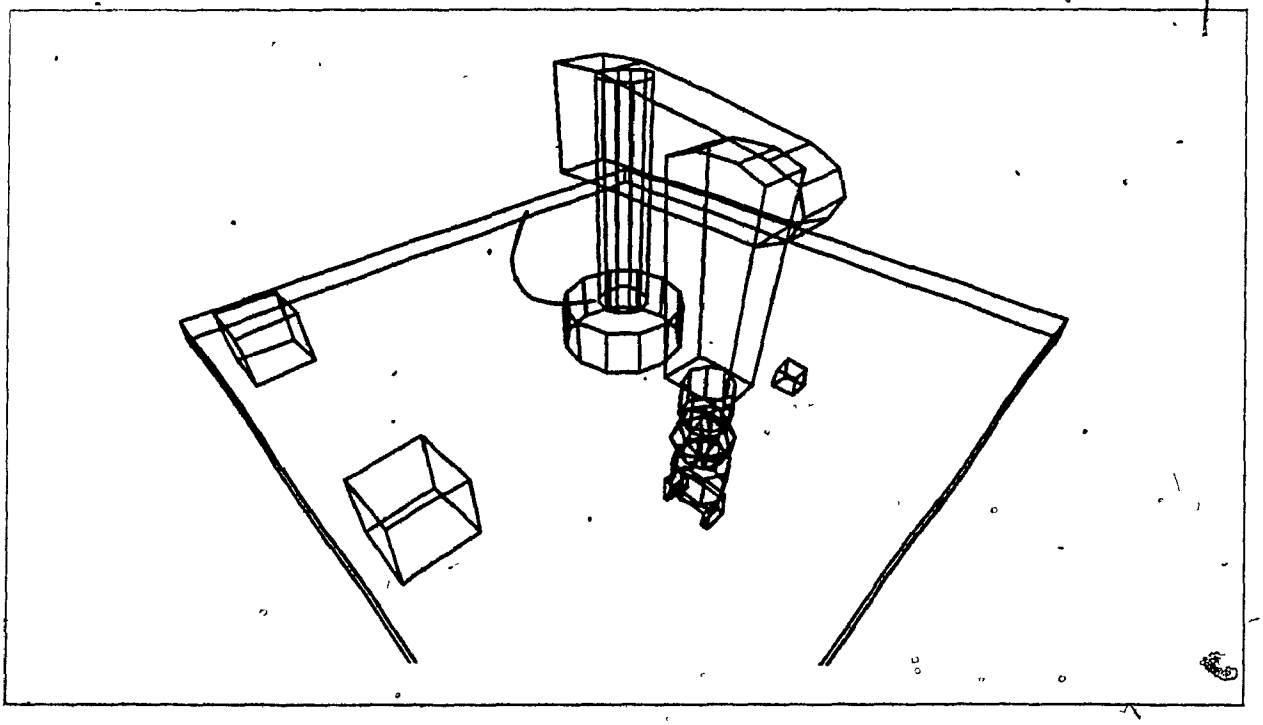


Figure 5.6 Initial Configuration of Workcell 2.

definition was not available.

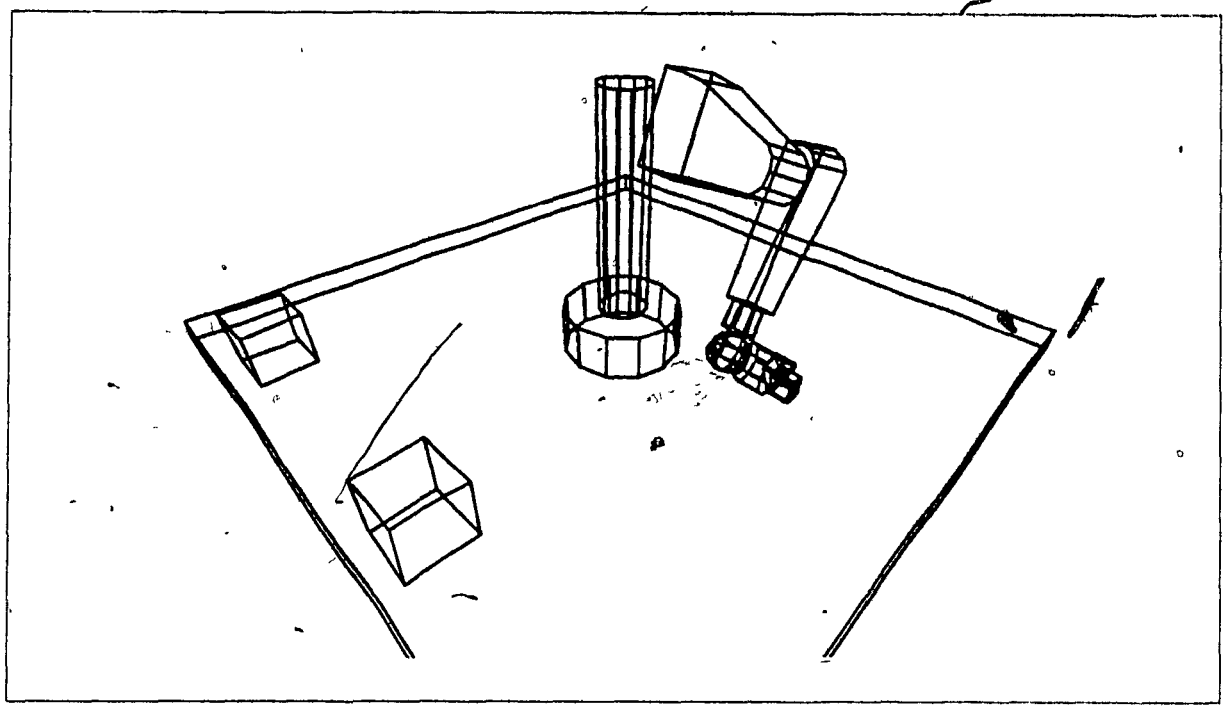


Figure 5.7 Pick Configuration

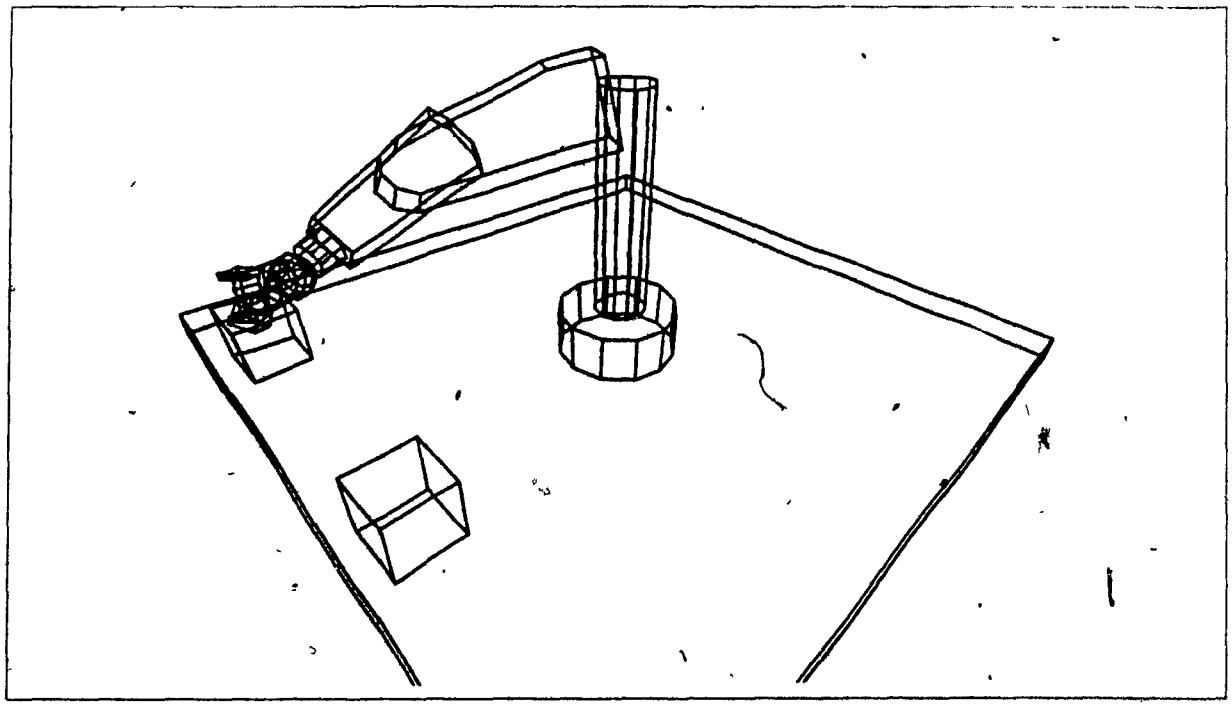


Figure 5.8 Place Configuration.

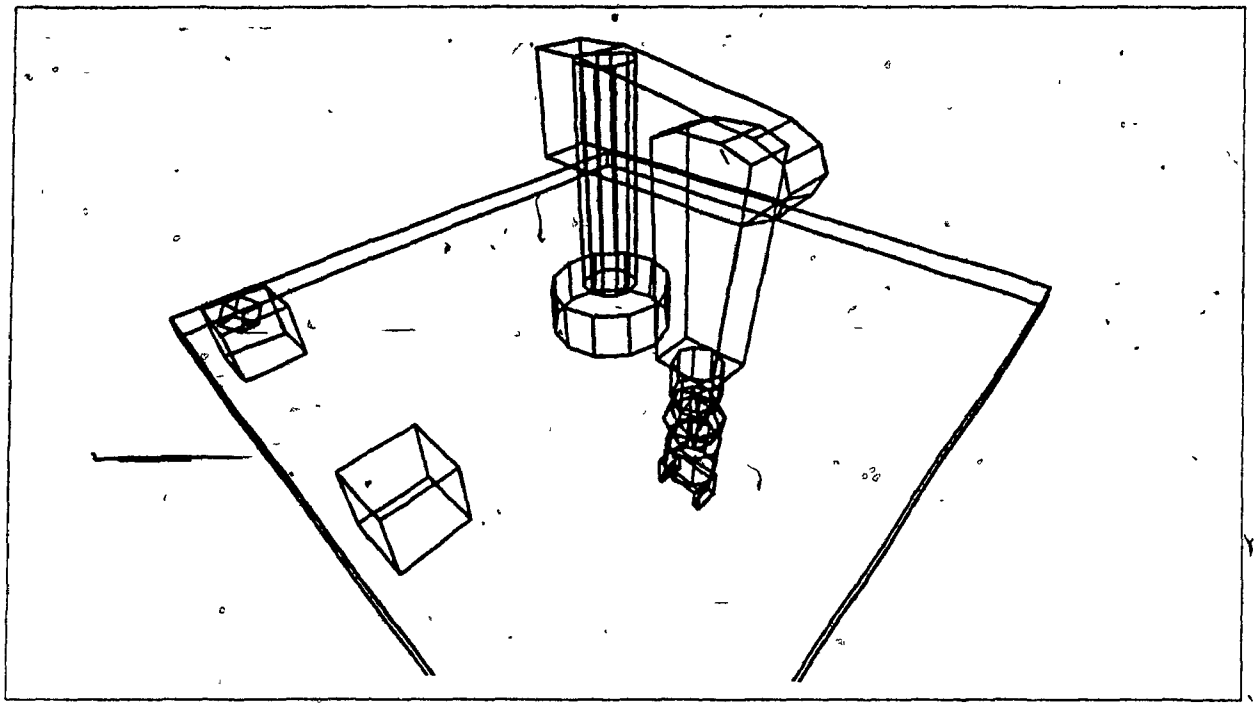


Figure 5.9 Final Configuration of Workcell 2

5.1.3 Example 3: Two manipulators and one solid to move

This last example involves the coordination of two manipulators. Both were chosen to be Puma manipulators, however, there is no obligation to do so. The workcell is composed of a spherical solid and two robots, each on its own table. The aim of this program is to show the capability of handling multiple robots, and the possible interaction between them. First of all, the workcell is created, the manipulators are identified, and their respective parameters initialized. The motion and grasping tasks are then tested directly through the key tree matcher to facilitate debugging. Then the program is written in order to manipulate the workcell in a convenient manner. This example consists of ordering one robot to go and pick up an object knowing its position and orientation. Then the same robot, "Puma_1", moves to a particular point which belongs to the intersection of the work environments of both robots. The second robot, "Puma_2", is then ordered to move to almost the same point with an orientation of its end effector which enables grasping the object. The robot "Puma_2" is then ordered to close its gripper, and "Puma_1" is ordered to open it, thus constituting a transfer of the solid. "Puma_1" goes back to its initial

configuration, and "Puma_2" brings the solid to within its work environment, releases it on the table in a chosen position and orientation and goes back to its initial configuration. A pseudocode of the program is shown in figure 5.10. Certain tests are omitted, such as the test needed to verify the success of any type of motion command. We also present some of the sequences during the execution of this program; the titles of the figures explain the actions being simulated.

```
initialization();
move_end_effector Puma_1, C1),/* C1: Configuration suitable to grasp the sphere*/
close_gripper(Puma_1);
move_end_effector (Puma_1, C2); /*C2: Configuration in the intersection of the workspaces of Puma 1 and
Puma_2*/
move_end_effector (Puma_2, C3), /*C3: Configuration suitable for grasping the sphere handed by Puma 1*/
close_gripper(Puma_2);
open_gripper(Puma_1);
move_joints_absolute (Puma_1, V1), /*V1: Initial values for the variables of Puma 1*/
move_end_effector (Puma_2, C4); /*C4: A certain configuration above the table*/
open_gripper(Puma_2);
move_joints_absolute (Puma_2, V2); /*V2: Initial values for the variables of Puma_2*/
end;
```

Figure 5.10 Pseudocode of Program 3.

5.2 Discussion

5.2.1 Time Considerations

As mentioned earlier, the play-back simulation or animation approach is the fastest, once the computations are complete. This approach is used when user interaction with the simulator is unnecessary. Some timing experiments have been conducted, we give the results for the first example in this chapter. The speed was set to 0.3 rd/s, and the

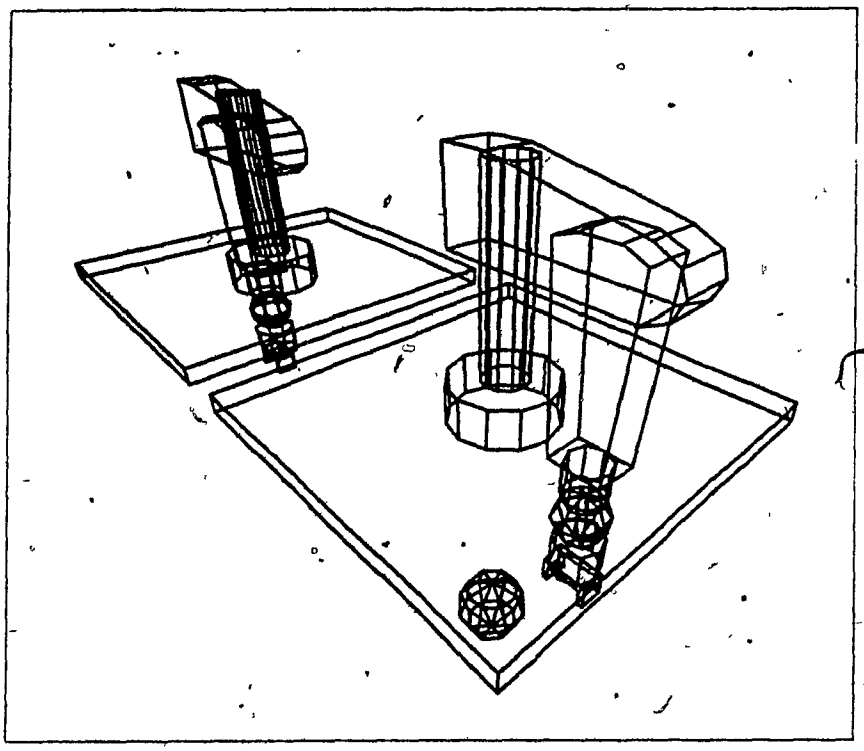


Figure 5.11 Initial Configuration of Workcell 3.

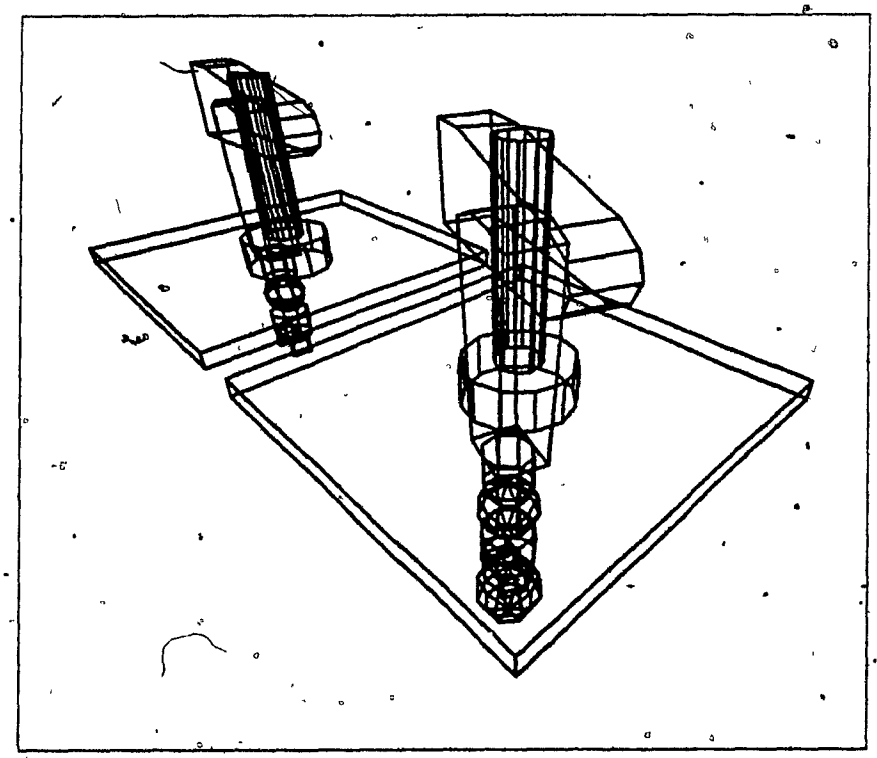


Figure 5.12 Pick Configuration of Puma_1.

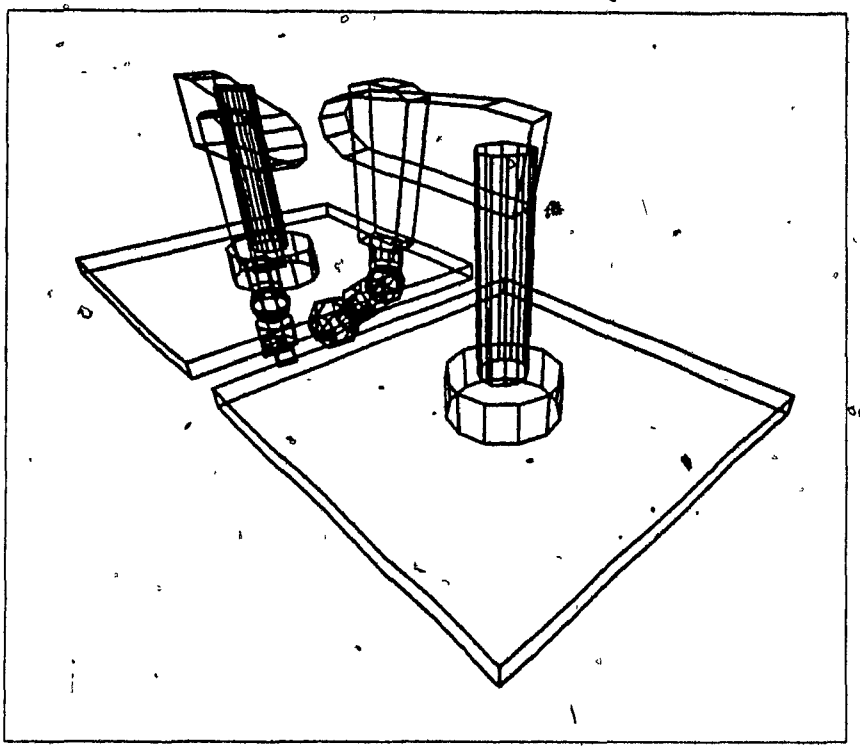


Figure 5.13 Motion of Puma 1 to the Transfer Configuration.

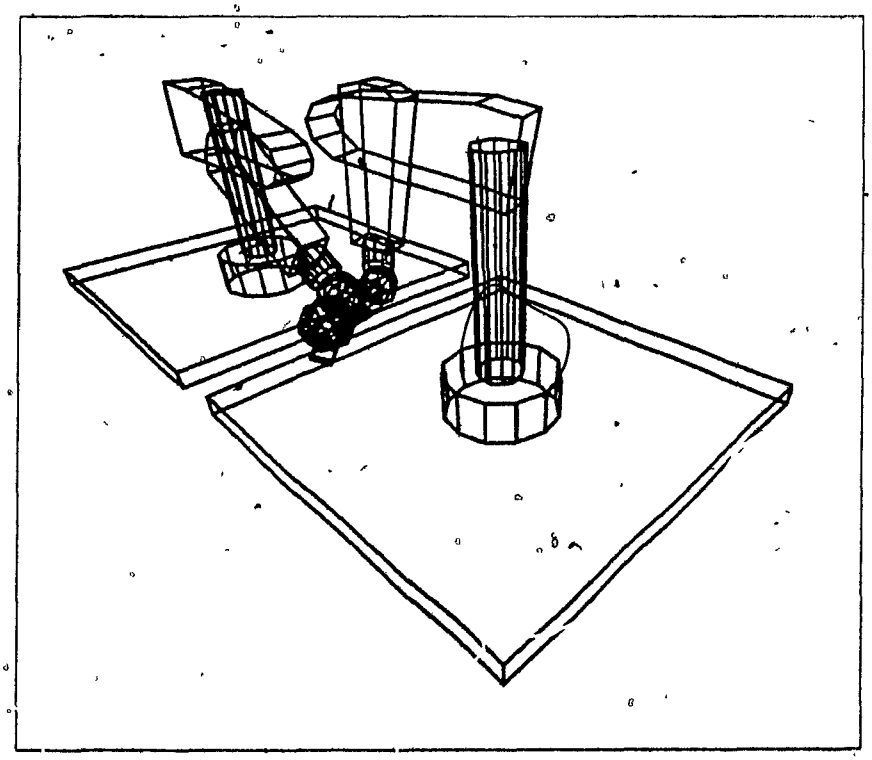


Figure 5.14 Transfer Configuration.

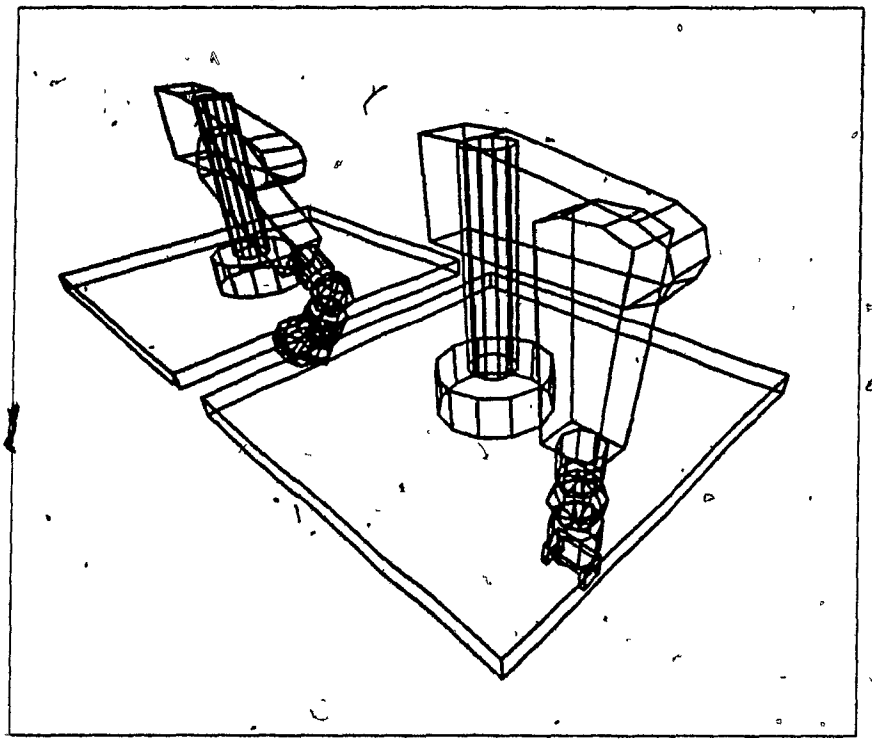


Figure 5.15 Puma_1 Back to Initial Configuration.

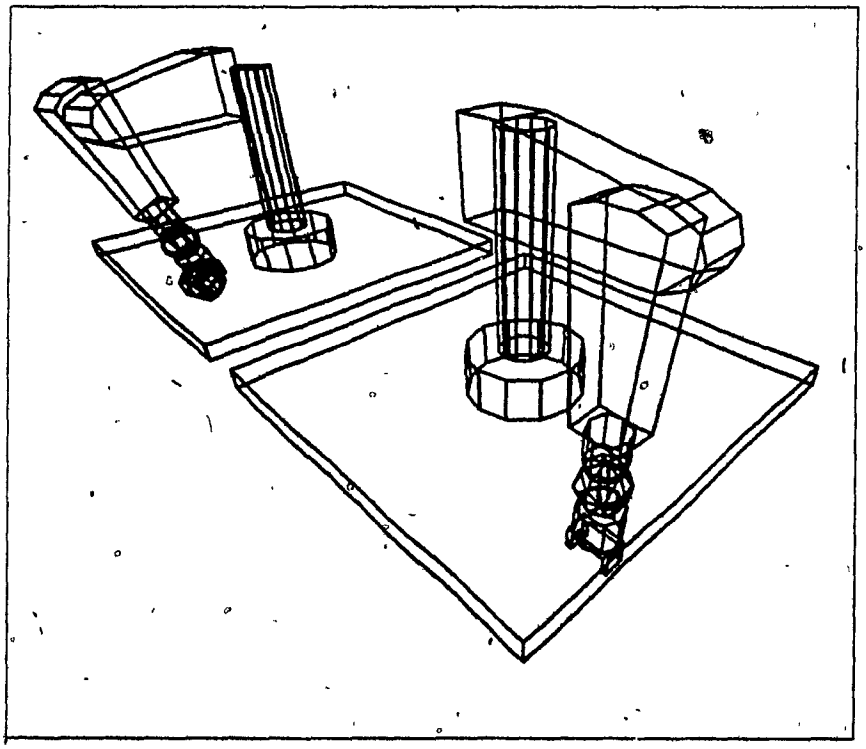


Figure 5.16 Puma_2 at Place Cofiguration.

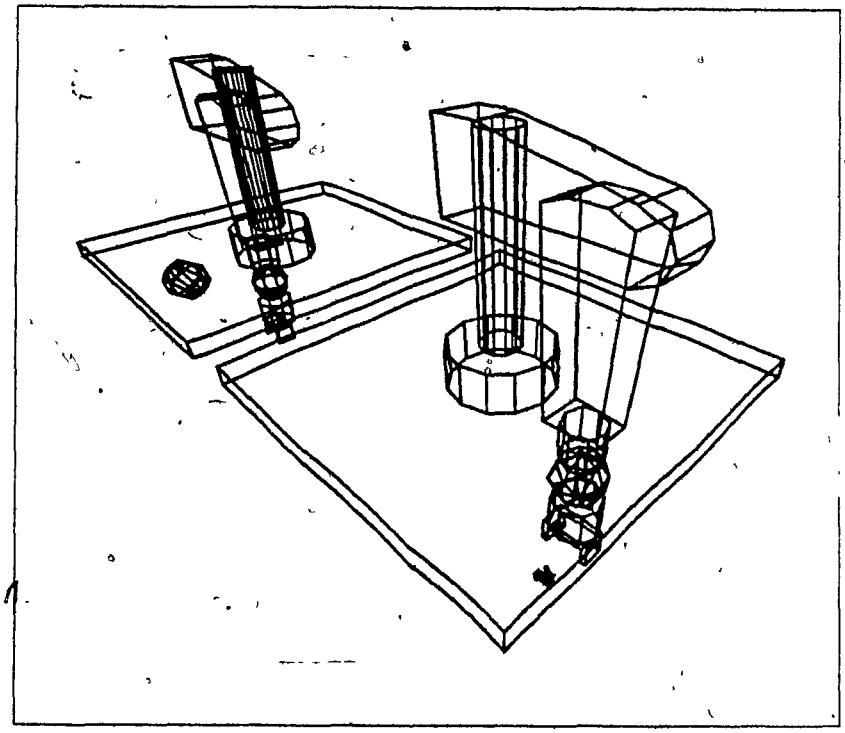


Figure 5.17 Final Configuration of Workcell 3.

program took 9.06 *sec* to animate after the computations, without the insertion of any wait statement. Considering the speed and the total motion, the time that the whole program would have taken if it were running on the real robot is shown as the simulated time in the time subwindow and is about 14 *sec*. Therefore, real time simulation is sometimes possible. However, this is not garanteed since it depends on the complexity of the workcell and the speed of the manipulators. For the on-line joint-by-joint simulation, the program takes 54.74 *sec*, which is about four times the real time. In the case of the on-line path simulation which is the closest to reality but the slowest for reasons of frequent updating as mentioned in the previous chapter, the program took 124 *sec* to simulate. These results show the tradeoffs to consider while simulating. We should note that the motion simulation in this particular example involves the manipulation of almost 800 vectors, as shown in figure 5.2.

5.2.2 Implementation Aspects

The simulation program was implemented in a modular manner which includes fifty "C" modules. There are 5 main blocks in the program: Database and data sublanguage (5900 lines), solid modelling (2400), motion and kinematics (7500), graphics and database suitable for graphics (1800), and the communication manager (400). The number of lines given above for each block is approximate. For the moment all the blocks reside in the same SUN 3 workstation. This is not necessary however, since some of the blocks do not need to be on the SUN. The design can be easily extended to place some of the blocks on the VAX 750 using the concept of a session layer [Freedman85] over the local area network.

5.3 Summary

From the examples shown above, we can see the facilities that the simulator presents. The procedure to follow in order to run the simulator is as follows: First construct the workcell to be programmed: This is done using the database, data sublanguage and solid modelling facilities. After the construction of the workcell, an identification process should take place to identify the manipulators and describe them; the solids other than manipulators should also be described as movable or not. The third stage involves is programming the workcell, under certain parameters such as simulation mode and the speed of the manipulators. In the examples given above, the manipulators in use were six degrees of freedom and six revolute manipulators. However, this was not necessary, since the manipulators storage structure can accept different architectures.

Chapter 6 Conclusion

In this thesis, an actual implementation of a graphic simulator for robotic workcells was developed. The problem of robotic workcell simulation was decomposed into smaller problems of database, data sublanguage, solid modelling, motion of solids in 3-D, robot kinematics, robot programming, and graphics. A database was developed in order to store and allow the manipulation of 3-D solids. The database is hierarchical in structure, but also has the property of explicit relations, as typical of a relational database. A data sublanguage was implemented to enable interaction with the database. It has the basic three queries allowing addition, deletion, and update of the entities and relations of the database. To further ease the interaction with the database, other facilities were developed. Chapter 2 of this thesis presented in detail the database and data sublanguage which are included in this simulator.

In order to model solids, certain solid modelling techniques were developed. The boundary representations were preferred for reasons mentioned in chapter 3, and the common primitives in solid modelling were created using those techniques. Seeking generality however, a sweep to boundary representations conversion was implemented. The rotational and translational sweep are therefore among the solid modelling facilities in this simulator. Complicated solids can be modelled using B-reps and sweep facilities. At the end of chapter 3, we presented the graphics facilities, as well as ways of extracting geometric properties of solids.

Once solids have been modelled, techniques were developed to move them according to simulation commands. Those techniques were implemented using mostly 2-D information in order to speed up motion processing. Other simulation commands such as speed, and grasp, were also implemented. All the simulation commands may be entered using the keyboard or through an independant program. The first approach is used at the debugging stage, and the second approach is used when the user is interested in simulating an entire program. The simulation commands and programming aspects of the simulator were presented in chapter 4. In chapter 5, we presented simulation results of some robot programs.

The simulation of robotic workcells can be extended to appear closer to reality. For this simulator, we suggest four areas of development that can be added in the near future. The first extension concerns graphics, where helpful graphic modules can be added, such as zooming, polygon filling, hidden surface removal, and others. There is, however, a tradeoff of speed to consider. The second extension concerns solid modelling, where by using the rotational and translational sweep facilities, we can extend to sweep along any PD curve. The third possible extension concerns collision detection for which boundary representations are very suitable. The fourth possible extension concerns the programming aspect of this simulator. At this stage, the programming is done at the manipulator level, which means that the commands are given in terms of joint or end effector values. However, since a database is available to us, and objects are already defined in it, object level programming or even task level programming would be possible to implement. Dynamics may also be considered as a desirable addition to the simulator.

References

[Angeles83] Angeles J., *Cálculo de Cantidades físicas Globales Asociadas a Volúmenes Acotados por Superficies Cerradas Mediante Integración en la Frontera*, Ingeniería, Vol. LIII, No. 1, pp. 95-102, 1983.

[Angeles85] Angeles J., *On the Numerical Solution of the Inverse Kinematic Problem*, *int. J. Robotics Res.* pp. 21 - 37 , 1985.

[Angeles86a] Angeles J., *Iterative Kinematic Inversion of General Five-Axis Robot Manipulators*, Int. J. Robotics Res., Vol. 4, No. 4, pp. 59-70, Winter 1986.

[Angeles86b] Angeles J., *The Evaluation of Moments of Bounded Regions Reduced to Line Integration*, Tech. Rep. Mech. Eng. McRCIM, McGill U., 1986.

[Angeles87] Angeles J., Rojas A., *Manipulator Inverse Kinematics Via Condition-Number Minimization and Continuation*, to appear in the Int. J. of Robotics and Automation, May 1987.

[Astrahan76] Astrahan M. M., et al., *System R: A Relational Approach to Data Management*, ACM Transactions on Database Systems, Vol. 1, No. 2, pp. 4 - 10, 1976.

[Baer79] Baer A., Eastman C., Henrion M., *Geometric modelling: a Survey*, Computer Aided Design, Vol. 11, No. 5, pp. 253-272, Sept. 79.

[Barnhill74] Barnhill R. E., Riesenfeld R. F., *Computer Aided Geometric Design*, Academic Press, N.Y., 1974.

[Baumgart74] Baumgart B. G., *Geometric Modelling for Computer Vision*, Rep. STAN-CS-74-463, Stanford Artificial Intelligence Lab., Stanford Univ, Stanford., Calif., 1974.

[Borrel83] Borrel P., Bernard F., Liegeois A., Bourcier D., Dombre E., *The Robotics Facilities in the CAM-CAM CATIA System*, Developments in Robotics, edited by B. Rooks, IFS Pub., 1983.

[Boyse79] Boyse J. W., *Interference Detection Among Solids and Surfaces*, Commun. ACM, Vol. 22, No. 1, pp. 3-9, January 1979.

[Boyse82] Boyse J. W., Gilchrist J. E., *GMSolid: Interactive Modelling for Design and Analysis of Solids*, IEEE Comp. Graph. Appl., Vol 2, No 2, pp. 27-42, March 1982.

[Cardenas79] Cardenas A. F., *Data Base Management Systems*, Allyn and Bacon, Boston, Mass, 1979.

[Claybrook85] Claybrook B. G., Claybrook A., Williams J., *Defining Database Views as Data Abstraction*, IEEE Trans. Software Engineering, Vol. 11, No. 1, pp. 3 - 14, January 1985.

[Codd72] Codd E. F., *Relational Completeness of Data Base Sublanguages*, ibid, pp. 65 - 98, 1972.

[Craig86] Craig J.J., *Introduction to Robotics: Mechanics and Control*, Addison-Wesley, 1986.

[Date81] Date C. J., *An Introduction to Database Systems*, Addison-Wesley, Reading, Mass., 1975.

[Derby82a] Derby S. J., *General Robot Arm Simulation Program (GRASP): Parts 1 and 2*, ASME Comp. Eng. Conf., San Diego, pp. 139-154, 1982.

[Derby82b] Derby S. J., *Computer Graphics Robot Simulation Programs: a Comparison*, Robotics Research and Advanced Applications, ASME Pub., Edited by W. J., Book:203-211, 1982.

[Derby83] Derby S., *Simulating Motion Elements of General-Purpose Robot Arms*, Int. J. Robotics Res., Vol. 2, No. 1, pp. 3-12, 1983.

[Dittrich85] Dittrich K., Lorie R., *Object-Oriented Database Concepts for Engineering Applications*, COMPINT, Montréal, pp. 321-325, September 1985.

[Faverjon86] Faverjon B., *Object Level Programming of Industrial Robots*, Proc. IEEE Int. Conf. Robotics and Automation, San Fransisco, pp. 1406-1411, 1986.

[Featherstone83] Featherstone R. *Position And Velocity Transformations Between Robot End-Effector Coordinates and Joint Angles*, *Int. J. Robotics Res*, Vol. 2, No. 2, pp. 35 - 45, 1983.

[Fenves85] Fenves S. J., *Representation and Processing of Engineering Design Constraints in a Relational Database*, COMPINT, Montréal, pp. 343 - 347, September 1985.

[Foley82] Foley J. D., Van Dam A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Raeding, MA, 1982.

[Freedman85] Freedman P., Carayannis G., Gauthier D., Malowany A., *A Session Layer for a Distributed Robotics Environment*, IEEE Proc. COMPINT, Montréal, Québec, Canada, 1985.

[Freedman86] Freedman P., Michaud C., Malowany A., *The Design of a Robotics Database for a High Level Programming Environment*, Technical Report TR-86-7R,

Computer Vision and Robotics Laboratory, Dept. of EE, McGill University, Montréal, June 1986.

[GMP81] Staff of GMP., *NONAME User Manual*, Dept. Mech. Eng., Univ. of Leeds., U. K., Oct. 1981.

[Hartenberg64] Hartenberg, R. S. and Denavit, J., *Kinematics synthesis of linkages* NEW YORK: McGraw-Hill, 1964.

[Haskin82] Haskin R., Lorie R., *On Extending the Functions of a Relational Database System*, Proc. Int. Conf. Mnagement of Data (ACM), June 1982.

[Hayward84] Hayward V., R. P. Paul, *Introduction to RCCL: A Robot Control "C" Library*, IEEE first Int. Conf. on Robotics, Atlanta, June 1984.

[Hayward86] Hayward V., *Fast Collision Detection Scheme by Recursive Decomposition of Manipulator Workspace*, Proc. IEEE Int. Con. Robotics and Automation, Vol. 2, pp. 1044-1049, April 1986.

[Heginbotham73] Heginbotham W. B., Dooner M., Kennedy D. N., *Computer Graphics Simulation of Industrial Robot Interactions*, SME, Proc. of 3rd CIRT, Seventh ISIR, 1973.

[Holland83] Holland J.M., *Basic Robotics Concepts*, Howard W. Sams and Co., Inc., 1983.

[Hunter79] Hunter G. M., Steiglitz, *Operations on Images Using Quadtrees*, IEEE Trans. Patern Anal. Machine Intell., PAMI-1, No. 2, April 79.

[Lloyd85] Lloyd J., *Implementation of a Robot Control Development Environment*, M. Eng. Thesis, McGill U., Montréal, Québec, Canada, Dec. 1985.

[Kretch82] Kretch S. J., *Robotics Animation*, Mechanical Engineering, pp. 32-35, Aug. 1982.

[Kunwoo85] Lee K., and Grossard D. C., *A hierarchical data structure for representing assemblies: part 1*, Computer-aided Design, Vol. 17, No. 1, pp. 15 - 24, January/February 1985.

[Lee82a] Lee Y. T., Requicha A. G., *Algorithms for Computing the Volume and Other Integral Properties of Solids. I. Known Methods and Open Issues*, Commun. ACM, Vol. 25, No. 9, pp. 635-641, 1982.

[Lee82b] Lee, Y. T., Requicha A. G., *Algorithms for Computing the Volume and Other Integral Properties of Solids. II. A Family of Algorithms Based on Representation Conversion and Cellular Approximation*, Commun. ACM, Vol. 25, No. 9, pp. 642-650, 1982.

[Liegeois80] Liegeois A., Fournier A., Aldan M. J., Borrel P., *A System for Computer-aided Design of Robots and Manipulators*, SME, Proc. of 10th ISIR, 1980.

[Light82] Light R., Gossard D., *Modification of Geometric models through Variational Geometry*, Computer Aided Design, Vol. 14, No. 4, pp. 209-214, July 82.

[Lossing74] Lossing D. L., Eshleman A. L., *Planning a Common Data Base for Engineering and Manufacturing*, SHARE XLIII, Chicago, Aug. 1974.

[Meghaer80] Meghaer D. J., *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary Three Dimensional Objects by Computer*, Tech. Rep. IPL-Tr-80; 111, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, N.Y., Oct. 80.

[Meghaer82] Meghaer D. J., *Octree Generation, Analysis, and Manipulation*, Tech. Rep. IPL-Tr-027, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, N.Y., April 82.

[Mortenson85] Mortenson M. E., *Geometric Modeling*, Wiley, N. Y., 1985.

[Newman79] Newman W. M., Sproull R. F., *Principles of Interactive Computer Graphics*, McGraw-Hill, N.Y., 1979.

[O'rourke79] O'rourke J., Badler N., *Decomposition of Three-Dimensional Objects into Spheres*, IEEE PAMI, Vol. PAMI-1, No. 3, July 1979.

[Paul81a] Paul R. P., *Robotic Manipulators: Mathematics Programming, and control*, Cambridge: MIT Press, 1981.

[Paul81b] Paul R. P., Shimano B., and Mayer G. E., *Kinematic Control Equations For simple Manipulators*, *IEEE Trans. Sys. Man Cybernetics* SCM - 11(16), pp. 449 - 455, 1981.

[Redd78] Reddy D. R., Rubin S., *Representation of Three Dimensional Objects*, Rep. CMU-CS-78-113, Dep. Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa., April 78.

[Requicha77] Requicha A. A. G., Voelcker H. B., *Constructive Solid Geometry*, Tech. Memo. 25, Production Automation Project, Univ. Rochester, Rochester, N.Y., Nov. 77.

[Requicha80] Requicha A. A. G., *Representations for Rigid Solids: Theory, Methods, and Systems*, Computing Surveys, Vol. 12, No. 4, pp. 437-464, December 1980.

[Sata81] Sata T., et al., *Robot Simulation System as a Task Programming Tool*, Proc. 11th. ISIR, Tokyo, Oct. 1981.

[Shumaker80] Shumaker G G.. *Robotics-Air Force Project.* Computer World. March 1980.

[Soroka80] Soroka B. I.. *Debugging Manipulator Programs With a Simulator.* Proc Autofact West Conf.. pp. 659-671. 1980.

[Späth78] Späth H., Munich-Vienna:*Spline-Algorithmen Zur KonstruKtion Glatter Kurven und Flächen.* 2nd. edition. R.-Oldenburg Verlag: pp. 27-4. 1978.

[Takano85] Takano. M. A.. *A new Effective Solution for Inverse Kinematics Problem (Synthesis) of a Robot with Any Type of Configuration* Journal of the Faculty of Engineering. The University of Tokyo. pp. 107 -135. 1985.

[Tosiyasu85] Tosjyasu L. K., Toshiaki S., Kazunori Y.. *Generation of Topological Boundary Representations from Octree Encoding.* IEEE Comp. Grap. and Appl., Vol. 5. No. 3. pp. 29-38. March 85.

[Tsai84] Tsai L. W., and Morgan A. P.. *Solving the Kinematics of the Most General six- and five-degree-of-freedom Manipulators by Continuation Methods.* ASME paper 84-DET-20. Cambridge. Mass.. ASME Design Engineering Technical Conference. 1984.

[Ullman82] Ullman J. D.. *Principles of Database Systems.* Computer Science Press. Inc.. 1982.

[Voelcker74] Voelcker H. B.. Middleditch A. E.. Zuckerman P. R., Fisher W. B.. Nelson T. S.. Requicha A. A. G.. Shopiro J. E.. *Discrete Part Manufacturing: Theory and Practice.* Part 1. Tech. Rep. 1. Production Automation Project. Univ. Rochester. Rochester. N.Y.. December 1974.

[Voelcker77] Voelcker H. B.. Requicha A. A. G.. *Geometric Modelling of Mechanical Parts and Processes.* IEEE Comput.. Vol. 10. No. 12. pp. 48-57. December 1977.

[Voelcker78] Voelcker H. B.. Requicha A. A. G.. Hartquist E.. Fisher W.. Metzger J.. Tilove R.. Birrell N.. Hunt W.. Armstrong G.. Check T.. Moote R.. Mcsweeney J.. *The PADL-1.0/2 System for Defining and Displaying Solid Objects.* ACM Comput. Gr.. Vol 12. No. 3. pp. 257-263. Aug. 78.

[Vossler85] Vossler L. D.. *Sweep-to-CSG Conversion Using Pattern Recognition Techniques.* IEEE Comp. Grap. and Appl.. Vol. 5. No. 8. pp. 61-68. August 85.

[Wang86] Wang W.P.. Wang K.K.. *Geometric Modeling for Swept Volume of Moving Solids.* IEEE Comp. Graph. App.. pp. 8-17. December 1986.

[Woodwark82] Woodwark J. R.. Quinlan K. M.. *Reducing the Effect of Complexity on Volume Model Evaluation.* Computer Aided Design. Vol. 14. pp. 89-95. 1982.