5-31-1991

Computer graphic simulation of sweeping of solid objects

Jr-Jyun Jang
New Jersey Institute of Technology

ABSTRACT

Title of Thesis : Computer Graphic Simulation of Sweeping of Solid Objects.

Name of Candidate : Jr-Jyun Jang
Master of Science in Mechanical Engineering, 1991.

Thesis Directed by : Dr. Ming C. Leu,
Professor of Mechanical Engineering,
Sponsored Chair in Manufacturing Productivity.
N.J.I.T.

This thesis is on the computer graphic simulation of the swept volume of a solid object undergoing Euclidean motions (including translation and rotation). The study helps visualize different types of Euclidean motions and supports the previously developed swept volume theories by providing graphic realism.

Included in the thesis presentation are the following :

1) Description of Euclidean motions of polyhedral objects, using ruled surfaces to represent swept volumes.
2) Representation of sweeping of solid objects, using parametric cubic equation and sweep differential equation.
3) Simulation of motions of PUMA and IBM robots.
4) Wire-frame and shaded image displays of swept volumes.

COMPUTER GRAPHIC SIMULATION OF SWEEPING OF SOLID OBJECTS

By

Jr-Jyun Jang

Thesis Submitted to the Faculty of the Graduate School of the New Jersey Institute of Technology in Partial Fulfillment of the Requirements for the Degree of Master of Science in Mechanical Engineering

1991

Blank Page

APPROVAL SHEET

Title of Thesis : Computer Graphic Simulation of Swept Volumes of Solid Objects.

Name of Candidate : Jr-Jyun Jang
Master of Science in Mechanical Engineering , 1991.

Thesis Approved by :

Dr. Ming C. Leu, Thesis Adviser — Date
Professor of Mechanical Engineering,
Sponsored Chair in Manufacturing Productivity,
N.J.I.T.

Dr. Raj Sodhi, Committee Member — Date
Associate Professor & Chairman of Manufacturing Engineering, N.J.I.T.

Dr. Rajesh N. Dave, Committee Member — Date
Assistant Professor of Mechanical Engineering,
N.J.I.T.

VITA

Name : Jr-Jyun Jang

Permanent Address :

Degree & Date to be Conferred : Master of Science in Mechanical Engineering
January, 1991.

Date of Birth :

Place of Birth :

Collegiate Institutions Attended	Date	Degree	Date of Degree
New Jersey Institute of Technology Major : Mechanical Engineering	09/88	MSME	May, 1991
National Chaio Tung University Major : Marine and Navigation Engineering	09/81	BSME	June, 1985

ACKNOWLEDGEMENTS

I thank Professor Ming C. Leu, my advisor, for presenting me the topic in geometric modeling, and for his encouragement and guidance. Also I thank Professor Denis Blackmore in mathematic department of Couran Institute of New York University, for spending so much time giving me suggestion and correction.

I thank my family for their faith and confidence, and Vivian, whose patience, and encouragement have been immeasurable.

To my parents and to my wife, Vivian

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Motion Planning

In robot motion planning there are two main features usually being discussed: first, robot trajectory planning which depends on robot dynamics, and second, the modeling of robot swept volume. The accuracy of off-line robot simulation depends on two factors: the accuracy of robot sweep equations and the resolution of representing swept volumes.

Using swept volumes to perform collision detection and motion planning requires accurate geometry representations. With inaccurate representation, collision may be undetected or a false collision generated. This may lead to inability to determine paths which are not collision-free or false collision-free paths.

A prime idea in [30] is the use of parametric cubic curves to approximate the robot trajectory which can roughly describe the robot motion. In [8][9][18], by using parametric spline one can "fit" a set of points to approximate the trajectory which the robot arm has passed through. It is obvious that by taking more points one can get better approximation of the robot motion, but the trade-off is more computation time.

The main topics in this thesis include the accuracy and the computation efficiency of swept volume. Different methods of swept volume representation are implemented in the computer, with the results shown using computer graphics (both in wire-frame and shaded image representations).

1.2 Parametric Equation and Differential Equation

Parametric equation and differential equation are the two different approaches used in this thesis for representation of swept volume.

Parametric equation is basically a numerical interpolation method (see [1][7][11]) to approximate sweep motion. In [11] parametric cubic equation was used to describe sweep motion. By taking infinitesimal intervals there would always be a fairly good approximate result toward the sweep motion. Parametric cubic equation is a fairly good method of simulating 3-D Euclidean motion. If the 3-D Euclidean motion is a "cubic equation describable" curve implying constant acceleration motion, then the parametric cubic equation will be the most efficient method.

The identification of a smooth sweep can be done with a system of first-order, linear, ordinary differential equations called the sweep differential equation. It follows from the theory of differential equations that the form of the sweep differential equation and the initial position of an object completely determine the swept volume of the object. [37] classified sweeps according to the properties of their sweep differential equations, as certain types of differential equations are likely to produce swept volumes with particularly simple features.

From the sweep equations one can get different types of swept volumes. In order to analyze the various swept volume types, it is beneficial to study the boundary surface of a swept volume.

1.3 Objective of Research

Motion planning and verification have become increasingly important in manufacturing automation. The use of swept volumes has shown great promise in efficient implementation of automation systems. For example, an verification systems requires an efficient implementation of intersection operations between the swept volumes of moving tools and potential obstacles.

The major objective of this thesis is to develop computer codes for graphic representation of swept volumes, which is useful for visualizing complex motions various types of objects such as NC tools, robots, etc. Through graphic displays this study also helps understand the various theories on swept volume geometries recently developed by Blackmore and Leu [37].

We apply two types of sweep equations to simulate motions of robots and other mechanisms. The first one is parametric cubic equation, and the second one is sweep differential equation. We develop a computer graphic simulation package capable of displaying sweeping of objects in wire-frame ad shaded images, for each given object geometry and sweep equation.

Chapter 2

LITERATURE REVIEW

2.1 Past Research

2.1.1 Simulation and Verification of NC Machine Motion

Collision is one of the serious problems in using automatic devices. An NC simulator enables NC programmers and machine operators to detect potential collisions visually and gross programming errors. A machining verifier seeks to determine automatically whether an NC program will produce a specified part without undesirable collisions, or cutter breakage, etc. Both simulator and verifier require solid modelling [3].

Solid modelling can be used for NC verification. In principal the machining operation is the process by which the unwanted portion of the volume is taken away from a given workpiece by " sweeping " the revolving cutter according to the programmed tool path [2].

Verification of part programs for NC machining using swept volumes has been presented and implemented in [2][3][23].

2.1.2 Robot Collision Detection

The most common robot motions are transfer movements for which the ability to plan motions that avoid obstacles is essential to the robot task planner. In [26][27][28] the motion planning schemes which include this swept volume have been presented.

Methods from computational geometry reduce motion planning to a geometric issue. A geometric representation of the volume swept by a moving object is generated. Intersection between this geometric model of swept volumes and geometric models of obstacles in the environment are determined.

[1] states that a solid can be represented by closed bounded surfaces. By analyzing the swept volume, we have two kinds of surfaces which can fully represent the swept volume of a polyhedral object. Dealing with the collision detection problem in this case is the same as doing intersection checking of surfaces. As in [4][5][6][7] there are different kinds of algorithms to implement interference checking. The constraints of "bounded" closed surfaces increase the complexity of interference checking.

Analytically, the swept volume of a polyhedral object can be decomposed into ruled surfaces and developable surfaces (which is basically the line swept surface or plane swept surface). The properties of these two surfaces are discussed in [15][16][18][30][7][8].

2.2 Robot Kinematics

Robot arm kinematics deals with the analytical study of geometry of motion of robot arm with respect to a fixed reference coordinate system without regard to the forces/moments that cause the motion.

There are two fundamental problems in robot arm kinematics. The first problem is usually referred to as the direct (or forward) kinematics problem. The second one is the inverse kinematics (or arm solution) problem.

Forward kinematics : (direct kinematics)

Denavit and Hartinberg [10][34] (here, it is simplified as D-H table) proposed a systematic and generalized approach for utilizing matrix algebra to describe and represent the spatial geometry of the links of robot arm with respect to a fixed reference frame. This method uses a 4X4 homogeneous transformation matrix to describe the spatial relationship between two adjacent rigid mechanical links and reduces the direct kinematics problem to finding an equivalent 4X4 homogeneous transformation matrix that relates the spatial displacement of the hand coordinate frame to the reference coordinate frame.

Inverse kinematics : (arm solution)

In general the inverse kinematics problem can be solved by several techniques. The most commonly used methods are the matrix algebraic iterations and the geometric approach.

Rotation matrix about an arbitrary axis

The rotating coordinate system O_{XYZ} may rotate an angle ø about an arbitrary axis r which is a unit vector having components r_x, r_y, r_z and passing through the origin O. We can first make some rotation about the principal axis of the O_{XYZ} frame to align the axis r with the O_Z axis. Then a rotation about the r axis with ø angle and a rotation about the principal axis of the O_{XYZ} frame return the r axis to its original location.

$$R_{(r,\varphi)} = R_{(x,-\alpha)} R_{(y,\beta)} R_{(z,\varphi)} R_{(y,-\beta)} R_{(x,\alpha)} \qquad (2.3\text{-}2)$$

Rotation matrix with Euler angle representation

Three types of Euler angle systems are :

	Euler 1.	Euler 2.	R.P.Y.
Sequence	OZ axis	OZ axis	OX axis
of	OU axis	OV axis	OY axis
Rotation	OW axis	OW axis	OZ axis

note : XYZ --> UVW

2.3 Parametric Formulation

An intrinsic property is one that depends on only the figure in question, not the figure's relation to a frame of reference. The theory of curves proceeds from the intrinsic equations. It is interesting to make a distinction between intrinsic equations, as just defined, and natural equations,defined in the following way: A natural equation of a curve is any equation connecting the curvature $1/\rho$, the torsion τ, and the arc length s of the curve. We have

$$\boldsymbol{f}\left(\frac{1}{\rho}, \tau, s\right) = 0$$

A natural equation of a curve imposes a condition on the curve.

From a slightly different approach, we can describe a curve parametrically in terms of the arc length, by getting the equations $x = x(s)$ and $y = y(s)$. In fact the functions x,y must be related by the equations

$$\frac{dx}{ds} = \cos\theta \qquad \text{and} \qquad \frac{dy}{ds} = \sin\theta \qquad (2.3\text{-}1)$$

Differentiating these equations with respect to s, we can get a pair of second-order differential equations for any given curvature function k(s).

We cannot express shapes required for geometric modeling with ordinary, single-valued functions. The dominant means of representing shapes in geometric modeling is with parametric equations. If we fit a curve or surface through a set of points, the relationship between the points themselves determines the resulting shape, not the relationship between these points and some arbitrary coordinate system. Besides, the curves and surfaces of geometric modeling are often nonlinear and bounded in some sense and can never be represented by an ordinary nonparametric function. Listed in [1] several advantages of using parametric equation.

2.4 Definition of Swept Volume

In general terms, the swept volume of an object moving in a given space from some initial location at $t = 0$ to some final location at $t= 1$ is defined as the 'volume' through which the object has passed. Let A be an object that is swept and let A_t represent an instance of A during the sweep for some $t \in I=[0,1]$. Then the swept volume of A, SV(A), is the union over I of all instances A_t,

$$SV(A) = \bigcup_{I} A_t$$

The generality of this definition can be removed while including a description of the motion of the sweeping object by redefining the swept volume in terms of trajectories of its point set. The motion of any point or set of points of the object can be determined during the sweep [22,25].

Definition 2.4-1

Let $h : X \dashrightarrow Y$ be a bijection with X and Y two topological spaces. Then the function h is a homeomorphism if both h and the inverse function $h^{-1} : Y \dashrightarrow X$ are continuous.

Definition 2.4-2

An n-ball in IR_n is the set

$$B^n = \{ (x_1,,,,, x_n) \in \mathrm{IR}^n \mid x_1^2 + \cdots\cdots + x_n^2 \leq 1\}$$

An open n-ball is the interior of B^n. A half n-ball is an open n-ball minus the open half-space determined by a hyperplane through its center.

2.4.1 Swept Volumes of Polyhedral Objects

Here we apply the swept volume theorem to the generation of swept volumes for a special class of compact 3-manifolds in IR_3 -- planar polyhedral 3-manifolds.

The polyhedral n-manifolds under consideration in this paper are those with planar faces and will be referred to as polyhedral objects. Here we discuss the geometric representation of the swept volumes for polygons undergoing general motions in IR_3. As we describe the polyhedral objects by using boundary representation, the boundary surfaces of polyhedral objects consist of a finite number of planar polygonal faces which meet along edges and vertices. For any polyhedral object, its boundary can represented as the union of all its polygonal faces.

Swept Volumes of Polyhedral Objects Reduced to Swept Volumes of Polygonal Faces

For any polyhedral object, its boundary can be represented as the union of all its polygonal faces. As shown in Fig. 4.3-1 The swept volume of the union of two objects equals the union of their swept volumes.
Let A be a polyhedral object in IR_3 and let $f_i A$ be the ith face of A. Then the boundary of A is

$$\partial A = \bigcup_{i=0}^{\#faces} f_i A$$

$$SV(A) = A_0 \cup SV(\partial A)$$

$$= A_0 \cup SV\left(\bigcup_{i=1}^{\#faces} f_i A \right)$$

$$= A_0 \cup \left(\bigcup_{i=1}^{\#faces} SV(f_i A) \right)$$

$$SV(A) = \bigcup_{i=1}^{\#faces} SV(f_i A)$$

If it is determined that during the sweep A_0 intersect $A_1$is empty set, Then the swept volume of a polyhedral object is reduced to the swept volume of its planar polygons. See Fig. 4.3-2. The geometric representation of the swept volumes discussed here are for continuous general motions of polygons.

Boundary Surfaces of Swept Volumes of Polygons

The boundary surfaces of the swept volumes of polygons consist of ruled surface segments, segments of developable surfaces, and the surface of the polygon at its initial and final location. For a general motion of a polygon sweeping, there are 6 degrees of freedom.

For the simplest case, where the polygon is undergoing a shifting movement, the envelope can be considered as generated by the sweeping of its edges. By connecting these swept edges — which are ruled surfaces, one forms the ruled surface segments.

In complicated motion, the ruled surface cannot fully describe the swept volume of a polyhedral object, see Fig. 3.2-2(b), and part of the swept volume is formed by the sweeping polygon itself. In the other words, the interior points of the polygon become the boundary points of the swept volume. These surfaces are developable surfaces.

Chapter 3

ROBOT KINEMATICS AND PARAMETRIC FORMULATION

3.1 Parametric Cubic Curves

Parametric cubic curve is a reasonable curve to simulate the solid object motion. The word 'reasonable' implies, that cubic equation is the lowest order continuous equation which can describe accelerated (and decelerated) motion.

The algebraic form of a parametric cubic curve segment is given by the following three polynomials:

$$X(t) = a_{3x}t^3 + a_{2x}t^2 + a_{1x}t^1 + a_{0x}t^0 \qquad (3.1\text{-}1a)$$
$$Y(t) = a_{3y}t^3 + a_{2y}t^2 + a_{1y}t^1 + a_{0y}t^0 \qquad (3.1\text{-}1b)$$
$$Z(t) = a_{3z}t^3 + a_{2z}t^2 + a_{1z}t^1 + a_{0z}t^0 \qquad (3.1\text{-}1c)$$

The coordinates (X(t), Y(t), Z(t)) can be treated as the trajectory of a particle (or 'body-attached coordinate frame') movement in the Cartesian space. The coefficients are the 'record' of this trajectory, which fully describes the position of the particle with respect to time t. The parameter t is restricted from 0 to 1. This restriction makes the curve segment bounded.

The twelve coefficients (in Eq. 3.1) are algebraic constants to be determined. This implies that four points located on the curve have to be known for determining the parametric cubic curve. Described in [1] are other ways of defining the curve.

Figure 3.1-1 gives an example of parametric cubic curve and the associated time histories of x,y, and z coordinates. In 4-point form we get

$$P(t) = A_3t^3 + A_2t^2 + A_1t^1 + A_0t^0 \qquad (3.1\text{-}2)$$

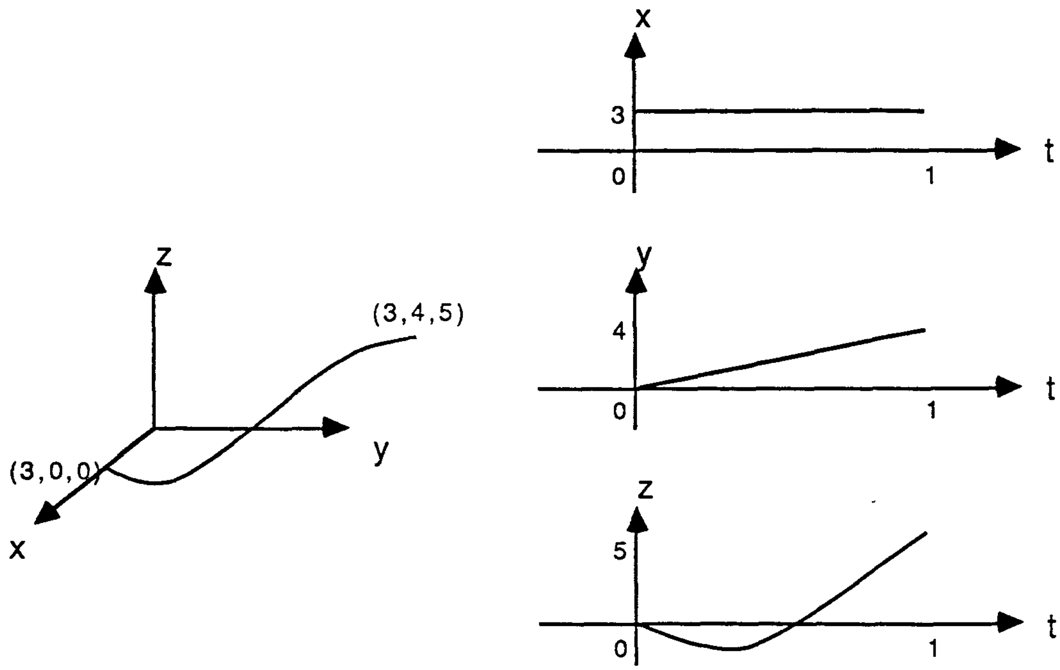


Fig. 3.1-1 The Euclidean value maps onto t domain.

3.2 Surfaces

In section 2.2.1 the swept volume of a polyhedral object can be bounded by the swept surfaces which is generated by the sweeping edges of the polyhedral object. In differential geometry [1][8][9][11], a sweeping line can produce a "ruled surface". Here we also use polyhedral objects to present solid objects. The connection between two vertices is a line segment called generator of the ruled surface.

3.2.1 Sixteen points form cubic surface

Equation 3.1-1 $\sum_{i=0}^{3} a_{ij} t^i$. is a one parametric equation. Which maps the t-domain into Cartesian space. Now take one more parameter into consideration, i.e. map u,w (two independent variable) into Cartesian space. The parameters u and w can define a continuous cubic surface.

16 Points Form Cubic Surface

$$P(u,w) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} u^i w^j \qquad\qquad P = UAW^T$$

$$U = [\, u^3 , u^2 , u^1 , 1 \,], \qquad W = [\, w^3 , w^2 , w^1 , 1 \,]$$

A is a matrix of 4X4X3 --- u,w, x-y-z

The following figure shows a cubic surface, It needs 16 points to define a cubic surface.

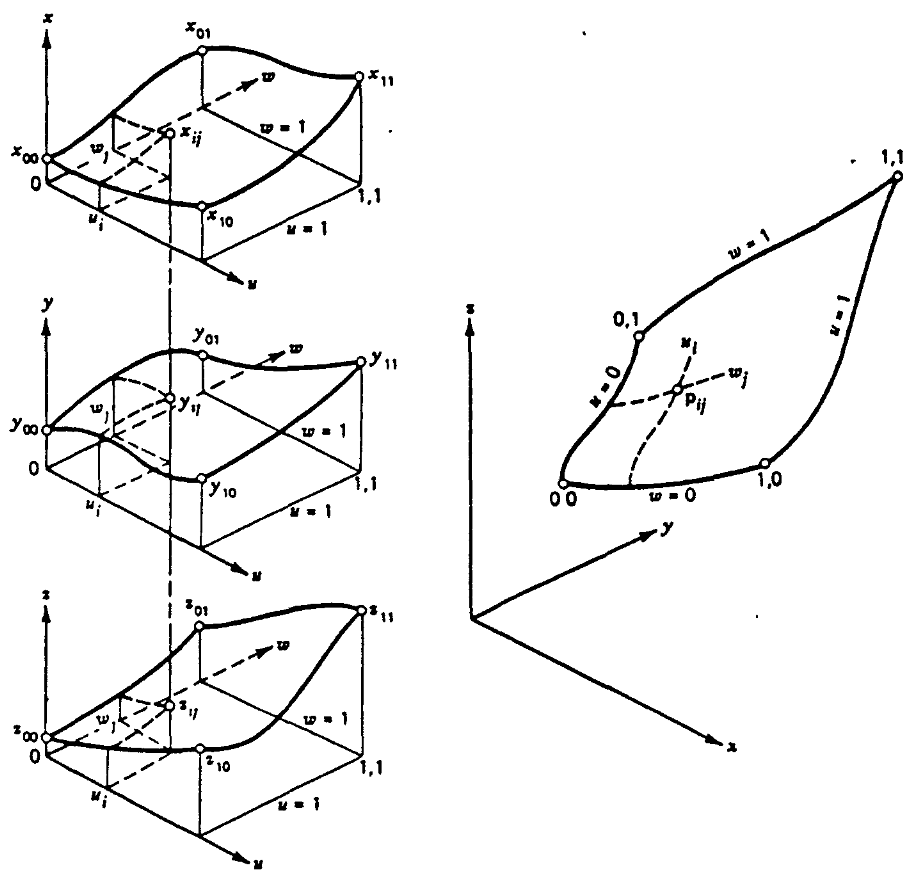


Fig 3.2-1 Sixteen points form of a bicubic surface.

3.2.2 Ruled Surfaces

A ruled surface is generated by a straight line segment undergoing a six-degree-of-freedom motion.

Definition 3.2-1

A surface such that through every point of the surface passes at least one straight line entirely in the surface is known as a ruled surface.

$$P(t,v) = d(t) + vg(t) \tag{3.2-1}$$

d(t) : is a curve in the surface,
g(t) : is a family of unit vectors along d(t) in the direction of sweeping line.
v : determines the distance of the point P(t,v) from d(t) along g(t).

From the above definition, the lengths of solid object edges are bounded by two end vertexes. The same ruled surface can also be defined by two curves $d_1(t)$ and $d_2(t)$ joined by straight line segments. The curves $d_1(t)$ and $d_2(t)$ are part of the boundary of ruled surface.

$$P(t,v) = d_1(t) + v(\, d_2(t) - d_1(t)\,) \tag{3.2-2}$$

Figure 3.2-2 shows two different types of swept volumes. One can see from Fig. 3.2-2(b) that if the planar facet intersects itself, the ruled surface cannot fully describe the boundary of swept volume.

3.2.3 Developable Surfaces

A developable surface is formed by successive planes which intersect each other such that all the intersection lines lie inside the sweeping planes (see Fig 3.2-2(b))

Definition 3.2-2

A surface such that through every point of the surface passes at least one straight line entirely in the surface and the normal to the surface is constant along these straight lines.

General concepts :

A plane is decided by two intersecting lines L1 & L2 on point q

$$P(u,v,t) = q(t) + ur_1(t) + vr_2(t)$$

$$P(t,v) = q - r_1 \frac{q'. r_1 \times r_2}{r'_1. r_1 \times r_2} + v\left(r_2 - r_1 \frac{r'. r_1 \times r_2}{r'_1. r_1 \times r_2}\right)$$

q is a curve and $r_1(t)$, $r_2(t)$ are families of unit vectors passing through the curve q. The tangent direction to the curve q at each value of t is q'. Since $r_1(t)$, $r_2(t)$ are unit vectors, they have perpendicular tangent directions r_1' , r_2' respectively. A developable surface can be regarded as ruled surface in which the normal direction is constant along the straight lines in the surface.

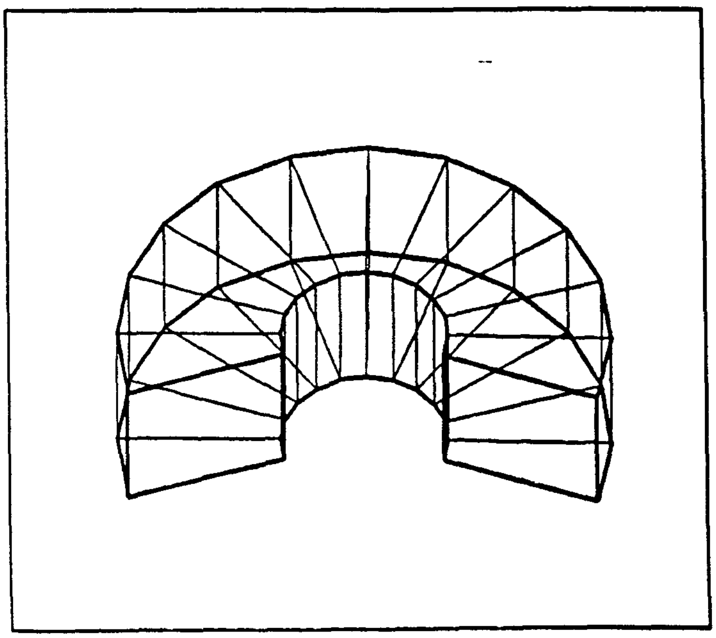


Fig. 3.2-2(a) Swept volume described by ruled surfaces.

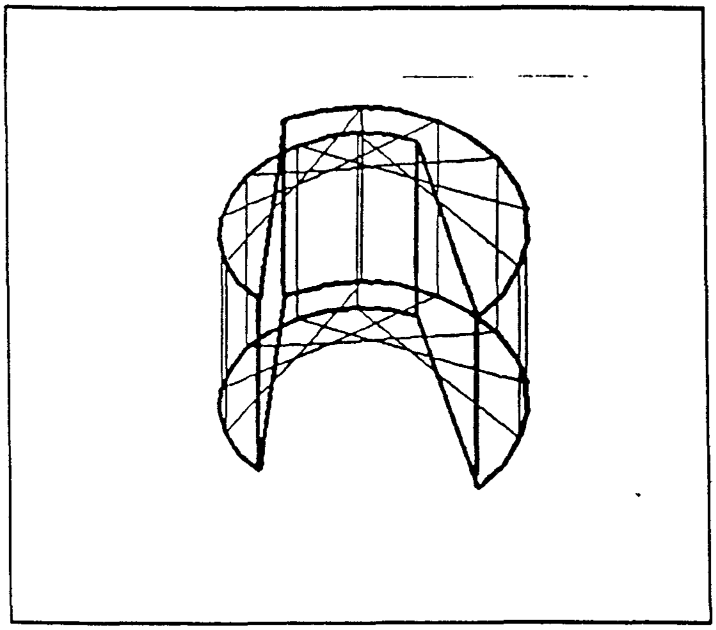


Fig. 3.2-2(b) Swept volume described by ruled surfaces and developable surfaces.

3.3 Robot Kinematics and Cubic Parametric Formulation

Rotational Matrix

A rotation matrix geometrically represents the principal axes of the rotated coordinate system with respect to the reference coordinate system.

Since the inverse of a rotation matrix is equivalent to its transpose, the row vectors of the rotation matrix represent the principal axes of the reference system OXYZ with respect to the rotated coordinate system OUVW. Actually the rotation matrix is orthonormal.

D-H Table

A mechanical manipulator consists of a sequence of rigid bodies, called links, connected by either revolute or prismatic joints. Each joint-link pair constitutes 1 degree of freedom. Fig 3.3-1 shows the relation between links and joints on PUMA robot.

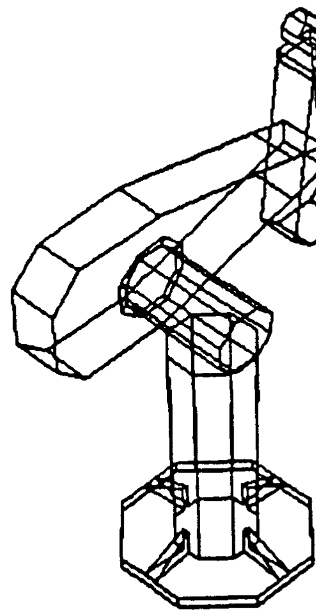


Fig. 3.3.1 Puma Robot and the 'body attached coordinate frame'

To describe the translational and rotation relationships between adjacent links, Denavit and Hartenberg proposed a matrix method of systematically establishing a coordinate system (body attached coordinate frame) to each link of an articulated chain.

Every coordinate frame is determined and established on the basis of three rules:
1) The z_{i-1} axis lies along the axis of motion of the ith joint.
2) The x_i axis is normal to the z_{i-1} axis, and pointing away from it.
3) The y_i axis completes the right-handed coordinate system as required.

The D-H representation of a rigid link depends on four geometric parameters associated with each link.

Robot Kinematics and Parametric Cubic Equation

By using D-H table, we plug in four joint variable values, after the computation we can get four different robot arm configurations. If we plug the values at time t=0, t=1/3, t=2/3, and t=1 (from t=0 to t=1), the four robot configurations become the four boundary condition of the parametric cubic equations.

3.4 Algorithm

The solid object is composed of polyhedral facets (or piecewise flat surfaces). The solid object has the shape of polyhedron. The simple polyhedra are the most important, since they are historically the source of topology's contribution to geometric modeling. The term simple polyhedra refers to all polyhedra that can be continuously deformed into spheres. Regular polyhedra are an example and subset of the simple polyhedra. In other words, regular polyhedra have no reentrance edges; thus they are convex.

The word convex can be applied to every polyhedron that lies entirely on one side of each of its polygonal face. So every convex polyhedron is a simple polyhedron, but a toroidal polyhedron is not.

Among vertices, edges, and faces of a simple polyhedron, called Euler formula for polyhedra: vertices no. - edges no. = 2 - facets no. The above simple formula provides a direct and simple proof that there are only five regular polyhedra.

Take the example of a more general case,i.e. a surface formed by taking a collection of planar surfaces. Any surface formed in this way will obviously be flat everywhere except the edge where flat surfaces are jointed together. The polygon has only straight edges, then the joint surface has curvature only on the edges.

Because of the above properties, the following points have to be characterized in order to describe how the surfaces are jointed together. This kind of representation is named atlas, which is a collection of separate maps of the flat facets of the solid object. As the meaning of atlas, there should be a route and orientation from each location to any other locations. The same thing here, atlas must keep a record of all the relations (including orientation, edges, and parent facets and grandparent solid) between vertices.

Now, starts from the construction of a flat facet :

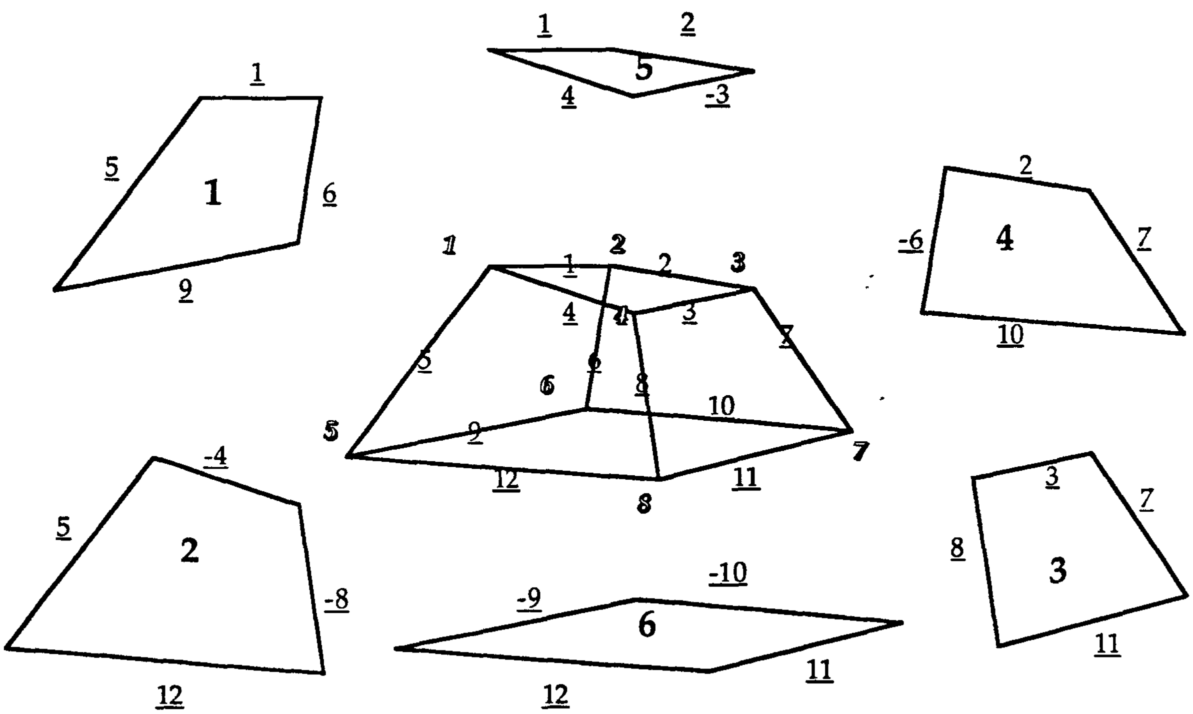


Fig. 3.4-1 Atlas of a solid object.

In order to explain the above solid object better, every vertex, edge and facet should have a name called 1st, 2nd, 3rd,... , 8th vertex, and 1st, 2nd, 3rd,... , 12th edge, 1st, 2nd, 3rd,... , 6th facet. The above figure shows the atlas of a solid object which contains 6 flat facets, each of which composed of 4 oriented edges and every edge has 2 vertexes (a forward vertex — marked by an arrow, and a backward vertex) on both ends.

vertex[1..8][x,y,z] = vertex corrdinate.
Edge[1..12][to=0,fro=1] = 1st .. 8th (vertex ID)
Face[1..6][1..4] = 1st .. 12th (edge ID)
Orientation[1..6][1..4] = 0,1 (edge orientation ID of each face)

0 means the loop of a facet follows the edge direction,

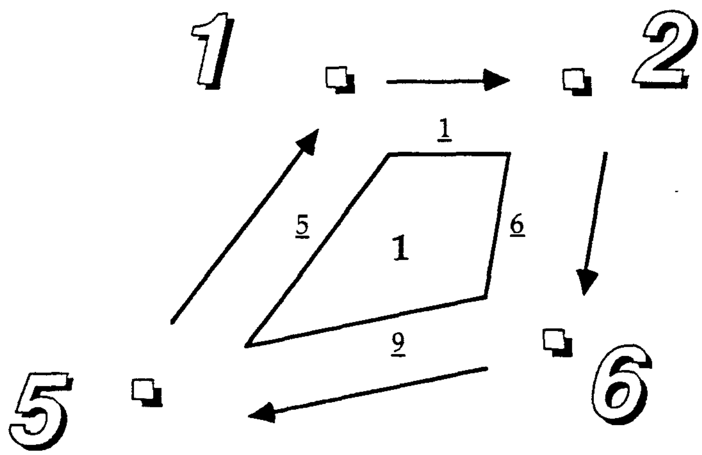
1 means the loop of a facet reverses the edge direction.

Number_of_facets = 6

Number_of_edges = 12

Number_of_edges_of_each_facets[facets] = 4

For example, on facet 1:



Vertex [Edge[Face[1][λ]][Orientation[1][λ]]][xyz] = 1,2,6,5

where λ = 1,2,3,4.

Chapter 4

SWEEP DIFFERENTIAL EQUATION

Parametric formulation is a numerical approach to sweep motion. This chapter is going to describe some simple Euclidean motions by using differential equation (see [37][35]).

The precise definition of swept and swept volume are given in section 2.4. This chapter defines the sweep differential equation, and introduces the mathematical and geometrical meaning of autonomous sweep motion and relatively autonomous sweep motion.

4.1 Sweep Differential Equations

The swept volume of an object in Euclidean n-space R^n is generated by a 1-parameter family of Euclidean motions of the form $\xi + Ax$ (translation + rotation), where x is a generic and ξ a fixed vector in R^n, and A is a matrix in the special orthogonal group

$$SO(n) = \{ A: A \text{ is a real, orthogonal, nXn matrix with } \det A = 1\}$$

SO(n) is a real analytic Lie group of dimension (n/2)(n-1). See [19] for details. Let Euc(n) be the Lie group of Euclidean motions in R^n. It is clear from the form of Euclidean motions that the Euclidean group Euc(n) can be identified with $R^n \times SO(n)$; hence it is a real analytic Lie group of dimension (n/2)(n+1).

Definition 4.1-1

A sweep is a continuous mapping $\sigma : [0;1] \to Euc(n)$ such that $\sigma(0)$ = the identity. We say that the sweep is smooth if it has continuous derivatives of all orders. Every sweep can be written in the form

$$\sigma_t(x) = \xi(t) + A(t)\, x \qquad (4.1\text{-}1)$$

where $\xi(0) = 0$, $A(0) = I$, the identity matrix, $\xi(t) \in R^n$, $A(t) \in SO(n)$, and σ_t is the value of σ at t for every $0 \geq t \geq 1$.

We shall confine our attention, for the most part, to smooth sweeps. This is certainly not unreasonable, since most sweeps encountered in practice are apt to be at least piecewise smooth.

Definition 4.1-2

Let $R^n \supseteq M$ and σ be a sweep in R^n. The swept volume of M under σ is the subset of R^n defined by

$$S_\sigma(M) = U \{ \sigma_t(M) : 0 \geq t \geq 1\} \qquad (4.1\text{-}2)$$

each of the sets $\sigma_t(M) = \{ \sigma_t(x) : x \in M\}$ is a t-section of $S_\sigma(M)$.

Given a smooth sweep σ, let us find a differential equation having solutions $x = x(t)$ which generate the sweep. On setting $x = x(t,x^0) = \sigma_t(x^0) = \xi(t)+A(t)x^0$ and differentiating, we obtain

$$\dot{x} = \dot{\xi}(t) + \dot{A}(t)\, x^0 \qquad (\dot{} = d/dt) \qquad (4.1\text{-}3)$$

Solving $x = \xi + Ax^0$ for x^0 using the fact that $AA^T = A^TA = I$, where denotes the transpose, and substituting the above equation yields

$$\dot{x} = \dot{\xi}(t) + \dot{A}(t)\, A^T(t)(x - \xi(t)) \qquad (4.1\text{-}4)$$

It follows from this derivation that $x(t) = \sigma_t(x^0)$ is the unique solution of this differential equation satisfying the initial condition $x(0) = x^0$. This suggests the following concept.

Definition 4.1-3

Let $\sigma_t(x) = \xi(t)+A(t)x$ be smooth in R^n. The smooth vector field

$$X_\sigma(x,t) = \dot{\xi}(t) + B(t)\,(x - \xi(t)) \qquad (4.1\text{-}5)$$

where $B(t) = \dot{A}(t)A^T(t)$, is called the <u>sweep vector field</u> (SVF) of σ and

$$\dot{x} = X_\sigma (x,t) \tag{4.1-6}$$

is called the sweep differential equation (SDE) of σ.

As (4.1-4) is linear, a solution such that $x(0) = x^0$ exists on the whole interval [0,1] (see [5]). This shows that there is a one-to-one correspondence between smooth sweeps and SDE's. Given this correspondence and the fact that the evolution of an object in a vector field is completely determined by the initial position of the object, it is quite logical to classify sweeps which generate swept volumes exhibiting a variety of particularized geometric and topological features. We shall identify one such class in the next section.

4.2 Autonomous S.D.E.

This section will subdivide the sweep differential equations into different categories. We will starts from the differential equations whose vector field does not explicit depend on t, called autonomous swept differential equation.

Definition.4.2-1

A smooth sweep is said to be autonomous if its SDE is autonomous; i.e.; X_σ in (2) does not depend on t.

We take the partial derivative of X_σ with respect to t and set it equal to zero, whence

$$\partial_t X_\sigma = (\ddot{\xi} - B\dot{\xi} - \dot{B}\xi) + \dot{B}x = 0 \tag{4.2-1}$$

The independence of x and t implies that this equation holds for all x and t if and only if $\dot{B} = 0$ and $\ddot{\xi} - B\dot{\xi} = d/dt\,[\,e^{-tB}\dot{\xi}] = 0$. This, in turn, is equivalent to $\dot{A}=BA$, with B constant, and $e^{-tB}\dot{\xi} = b$, with b constant. Here e^{-tB} is the usual matrix exponential (cf. [3],[5], and [19]). But $\dot{A} = BA$, $A(0) = I$ has unique solution $A(t) = e^{tB}$. Moreover, since $AA^T = I$ we infer that $e^{tB}(e^{tB})^T = e^{t(B+B^T)} = I$ which implies $B + B^T = 0$, so $B \in o(n)$, where

$o(n) = \{ B : B \text{ is a real, nXn skew-symmetric matrix} \}$

We have now essentially proved the following result:

Theorem 4.2-2

Let $\sigma_t(x) = \xi(t) + A(t)x$ be a smooth sweep. Then the following are equivalent :

(1) The sweep is autonomous.

(2) $\dot{A}A^T = B$ is constant and $A^T\dot{\xi} = b$ is constant

(3) $A(t) = e^{tB}$, $B \in o(n)$ and $\dot{\xi} = e^{tB}\, b$ with $b \in R^n$

(4) The SDE of σ is $\dot{x} = Bx + b$ where $B \in o(n)$ and $b \in R^n$

From the definition above, the autonomous sweep differential equation can be put down in the following form.

$$\dot{X}_2 = \Delta X_2 + \dot{X}_1 = \begin{pmatrix} 0 & -a & -b \\ a & 0 & -c \\ b & c & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} \dot{x}_c \\ \dot{y}_c \\ \dot{z}_c \end{pmatrix} \tag{4.2-2}$$

where a,b,c, $(\dot{x}_c, \dot{y}_c, \dot{z}_c)^T$ are independent of t.

The sweep σ is type 1 with respect to Q, if $\sigma_t\, Q \to S_\sigma(Q)$ maps interior points into interior points and boundary points to boundary points, for all $0 \le t \le 1$. (see Fig. 4.2-1).

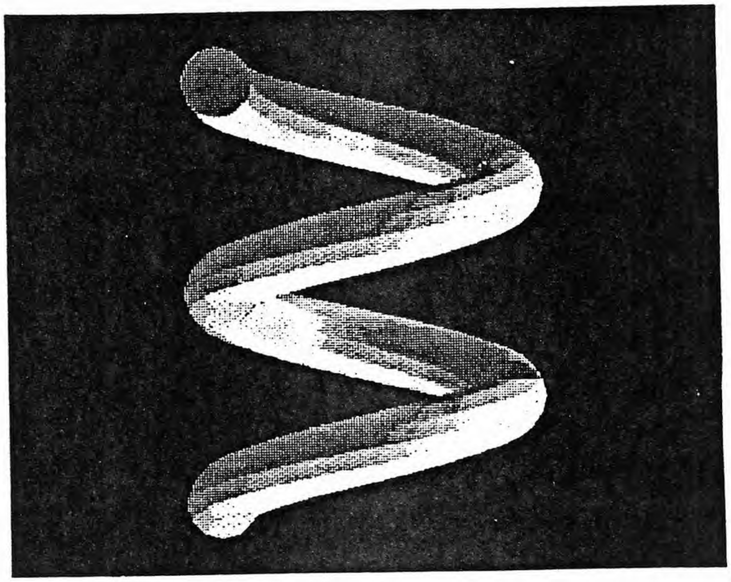


Fig 4.2-1 Type 1; a disk undergoing helical motion.

Type II : All the cases other than type I, is type II. (see Fig. 4.2-2 ,Fig. 3.2-2 (b))

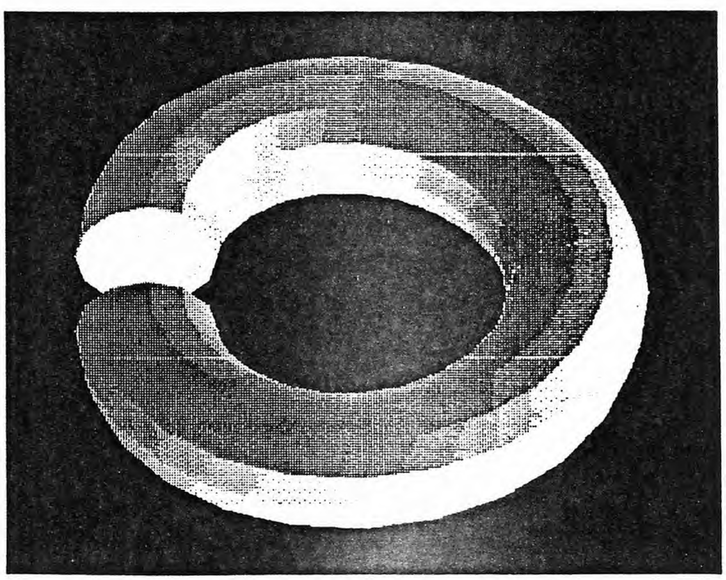


Fig 4.2-2 Type 2, also a disk undergoing helical motion but intersecting itself.

4.3 Relatively Autonomous S.D.E.

Definition 4.3-1

A smooth sweep is said to be relatively autonomous if its relative SDE is autonomous.

Observe that by defining $\zeta = x - \xi$, the differential equation of a sweep can be written in this form

$$\dot{\zeta} = A(\tau)A^T(\tau)\zeta \qquad (4.3\text{-}1)$$

as $\xi(0) = 0$, and $x(0) = x^0$ correspond to solutions of Eq. 4.3-1 subject to $\zeta(0) = x^0$.

Theorem 4.3-1

Let $\sigma_t(x) = \xi(\tau) + A(\tau)x$ be a smooth sweep. Then the following are equivalent :

(1) The sweep is relatively autonomous.

(2) There exist $C \in o(n)$ such that $A(\tau) = e^{\tau C}$ for all $0 \leq \tau \leq 1$.

(3) ${}^{\#}A(\tau) = \dot{A}(\tau)A^T(\tau) = C \in o(n)$ for all $t \in [0,1]$.

(4) The SDE σ has the form $\dot{x} = \dot{\xi}(t) + C(x - \xi(t))$, where $C \in o(n)$ and ξ: $[0,1] \to R^n$ is a smooth function with $\xi(0) = 0$.

From the definition, $\dot{A}A^T = B$ is constant; in the other words, the relatively autonomous sweep motion is autonomous sweep with respect to the relative coordinate frame.

$$\dot{X} = \Delta^{\#}X + \dot{X}_c = \begin{pmatrix} 0 & -a & -b \\ a & 0 & -c \\ b & c & 0 \end{pmatrix} \begin{pmatrix} {}^{\#}x \\ {}^{\#}y \\ {}^{\#}z \end{pmatrix} + \begin{pmatrix} \dot{x}_c(t) \\ \dot{y}_c(t) \\ \dot{z}_c(t) \end{pmatrix} \qquad (4.3\text{-}2)$$

where $(\dot{x}_c(t), \dot{y}_c(t), \dot{z}_c(t))^T$ is a function of t, but $\Delta^{\#}X$ is not.

Chapter 5

IMPLEMENTATION AND ALGORITHM

This chapter discusses implementation of the parametric cubic equation and sweep differential equations. The simulation programs are written in C language and HOOPS computer graphic utilities on the SUN3/60 workstation at NJIT.

The solid objects are defined according to Mobius principle [1][4][30]. All the solid objects are convex polyhedral objects. The computer programs are included in Appendixes I, II, and III.

The first section of this chapter discusses parametric cubic equation implementation on IBM and PUMA robots. The second section discusses the implementation of sweep differential equation on polyhedral objects and IBM robot. With detailed description and computer simulation, one can visualize the difference between autonomous sweep and nonautonomous sweep in Cartesian space. We also include the wireframe representation and shaded image representation of the swept volumes. The third section discusses the methods of showing the shaded images from the swept data by transforming three dimensional coordinates to two dimensional computer screen pixels, and remove the hidden surfaces using z-buffer.

The basic requirements for comprehending these simulation programs are the C programming language, a knowledge of UNIX system, HOOPS 2.02 graphic library, and SUN workstation. The following flow chart (Fig 5-1) illustrates the flow of programs execution. The later sections will discuss these individual blocks in detail.

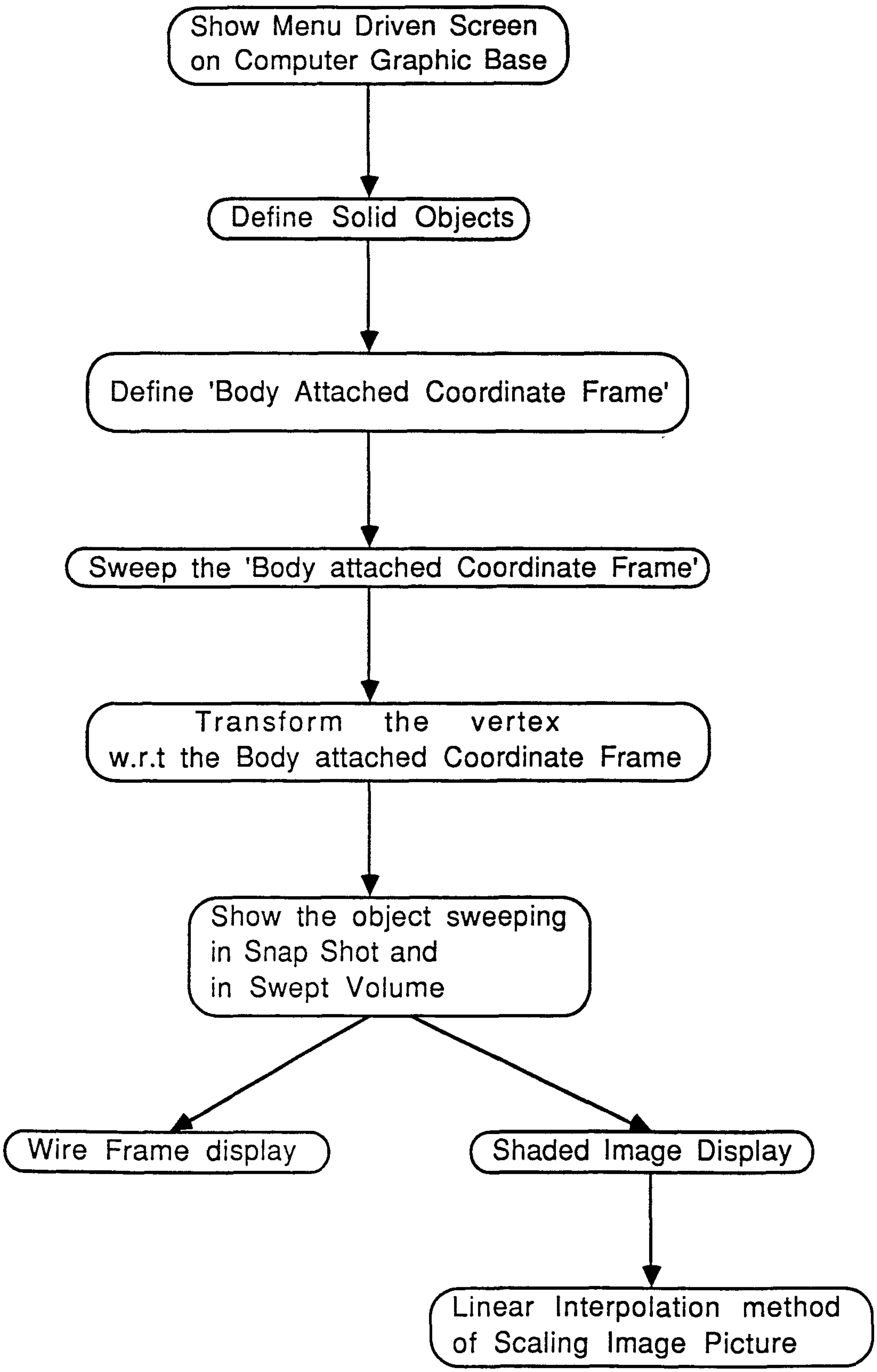


Fig. 5-1 Flow chart of the simulation Programs.

5.1 Parametric Cubic Equation Implementation

As mentioned in section 3.2, parametric cubic equation is used to describe a a point trajectory. Now, take this point as the origin of a 'body attached coordinate frame'. In Euclidean motion there are six degrees of freedom, namely x,y,z,roll,pitch,yaw. When a solid object sweeps, the 'body attached frame' has a smooth motion in Euclidean space. i.e. the x,y,z,roll,pitch,and yaw values change smoothly.

There is a critical issue which needs to be clarified, i.e. the existence of angle $\phi(t)$, $\theta(t)$, $\psi(t)$. Before getting into any explanation, first take a look at Fig 5.1-1 and preview the fixed relationship between the 'body attached coordinate frame' and the vertex of a polygon.

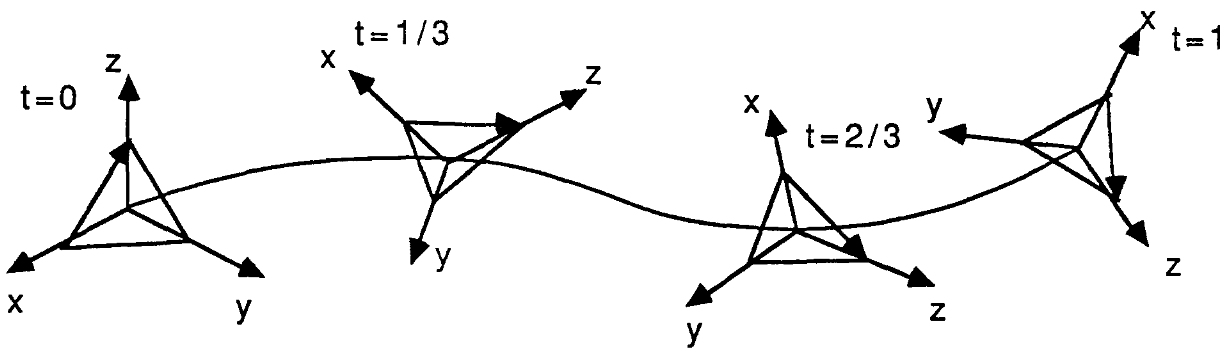


Fig 5.1-1 The body attached frame undergoing Euclidean motion.

One can decide 4 values while at t=0, t=1/3,t=2/3, and t=1, using the Eq. 3.1-1 and [1][11][14].

P(t) is the trajectory of the 'body attached coordinate frame'. The 4 data points P_1 , P_2 , P_3 , P_4 should be known values. In robot application these 4 sets of known values are calculated using the D-H table [see 10]. For example, one has to define the variable at the 'time' t_1 , t_2 , t_3 , t_4 shown in Fig. 5.1-2 which includes a D-H table on the upper portion of the picture.

$$P_n(t) = \begin{pmatrix} x_n(t) \\ y_n(t) \\ z_n(t) \\ \phi_n(t) \\ \theta_n(t) \\ \psi_n(t) \end{pmatrix} \qquad G = \begin{pmatrix} -4.5 & -9.0 & -5.5 & 1 \\ 13.0 & -22.5 & 9 & 0 \\ -13.5 & -18.0 & -4.5 & 0 \\ 4.5 & -4.5 & 1 & 0 \end{pmatrix} \qquad T = \begin{pmatrix} t^3 \\ t^2 \\ t^1 \\ t^0 \end{pmatrix}$$

$$H_p(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \\ \phi(t) \\ \theta(t) \\ \psi(t) \end{pmatrix} = P_nGT = (P_1, P_2, P_3, P_4)\begin{pmatrix} -4.5 & -9.0 & -5.5 & 1 \\ 13.0 & -22.5 & 9 & 0 \\ -13.5 & -18.0 & -4.5 & 0 \\ 4.5 & -4.5 & 1 & 0 \end{pmatrix}\begin{pmatrix} t^3 \\ t^2 \\ t^1 \\ t^0 \end{pmatrix}$$

In [10], there are three types of Euler angle representation toward rotational matrix. In this section, Eq 4.3-4 to Eq 4.3-6 are roll - pitch - yaw representation of rotational matrix.

$$R_{\phi\theta\psi} = R_\phi R_\theta R_\psi$$

$$= \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{pmatrix}$$

$$= \begin{pmatrix} C\phi C\theta & C\phi S\theta S\psi - S\phi C\psi & C\phi S\theta C\psi + S\phi S\psi \\ S\phi C\theta & S\phi S\theta S\psi + C\phi C\psi & S\phi S\theta C\psi - C\phi S\psi \\ -S\theta & C\theta S\psi & C\theta C\psi \end{pmatrix} \qquad (4.3\text{-}7)$$

For roll - pitch - yaw representation, the rotational sequence is roll -> pitch -> yaw which rotate with respect to X, Y, Z axis of world coordinate frame.

Based on the above concepts and the solid modelling method in section 3.4, the parametric cubic equation is used to keep the record of the sweep motion of the solid object. The data set $P_n(t) = (x_n(t),y_n(t),z_n(t),\phi_n(t),\theta_n(t),\psi_n(t))^T$ is required to describe position and orientation of the "body attached coordinate frame" for transfer of the solid object vertices.

Before we start introducing the algorithm used in the simulation programs, first let us show the table of the symbolic arrays (or variables) which is used to represent the arrays (or variables) in the source programs. These symbols will be used in section 5.1, 5.2,

$\#N_f$: The number of the facets of a solid object.
$\#N_e$: The number of edges of a facet.
$\#T_k = 13$: The number of instants when t changes from 0 to 1.

$$p = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad p(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} \qquad o = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \qquad o(t) = \begin{bmatrix} \phi(t) \\ \theta(t) \\ \psi(t) \end{bmatrix}$$

SO[face ; edge ; p] : Polyhedral object vertices, defined by user.

$FR\begin{bmatrix} p(t) \\ o(t) \end{bmatrix}$: Sweeping curve of the 'body attached coordinate frame'.

SV[face ; edge ; p(t)] : Swept volume boundary points.

Algorithm 1

The polyhedral object and 'body-attached coordinate frame' has been defined before starting the simulation. The first step is to find the trajectory of the 'body-attached coordinate frame' by the four sets of given data which can be calculated by an existing preprocessing program.

/* P_1, P_2, P_3, P_4 are known values which should be given. */
for t = 1 to $\#T_k$

$$FR\begin{bmatrix} p(t) \\ o(t) \end{bmatrix} = (P_1,P_2,P_3,P_4)\begin{pmatrix} -4.5 & -9.0 & -5.5 & 1 \\ 13.0 & -22.5 & 9 & 0 \\ -13.5 & -18.0 & -4.5 & 0 \\ 4.5 & -4.5 & 1 & 0 \end{pmatrix}\begin{pmatrix} t^3 \\ t^2 \\ t^1 \\ t^0 \end{pmatrix}$$

end of t.

Algorithm 2

After the FR array is loaded with the information of the 'body-attached coordinate frame', one can transform the body-vertex according to the data in FR array.

/* transform the solid object along the 'body attached coordinate frame'. *./
for f = 1 to $\#N_f$
  for e= 1 to $\#N_e$
    for t = 1 to $\#T_k$

$$SV[f;e;p(t)]=\begin{pmatrix} C\phi C\theta & C\phi S\theta S\psi - S\phi C\psi & C\phi S\theta C\psi + S\phi S\psi \\ S\phi C\theta & S\phi S\theta S\psi + C\phi C\psi & S\phi S\theta C\psi - C\phi S\psi \\ -S\theta & C\theta S\psi & C\theta C\psi \end{pmatrix} \bullet SO[f;e;p]+FR[p(t)]$$

    end of t.
  end of e.
end of f.

var / link	θ_i	α_i	a_i	d_i
1st link	θ_1	0	0	0
2nd link	θ_2	0	-2.4	0
3rd link	0	180	-1.5	d_3

t / var	t	t	t	t
θ_1	0	20	40	60
θ_2	0	30	60	80
d_3	-2.4	-2.2	-2.0	-2.4

Fig. 5.1-2(a) D-H table of IBM robot, and varibles data table of simulation.

Blank Page

Blank Page

var / link	θ_ι	α_ι	a_ι	d_ι
1st link	θ_1	0	0	0
2nd link	θ_2	90	0	0
3rd link	θ_3	0	4.32	0
4th link	θ_4	–90	0	0

t / var	t	t	t	t
θ_1	0	20	40	60
θ_2	0	0	0	0
θ_3	0	0	0	0
θ_4	0	0	0	0

Fig. 5.1-2(a) D-H table of IBM robot, and varibles data table of simulation.

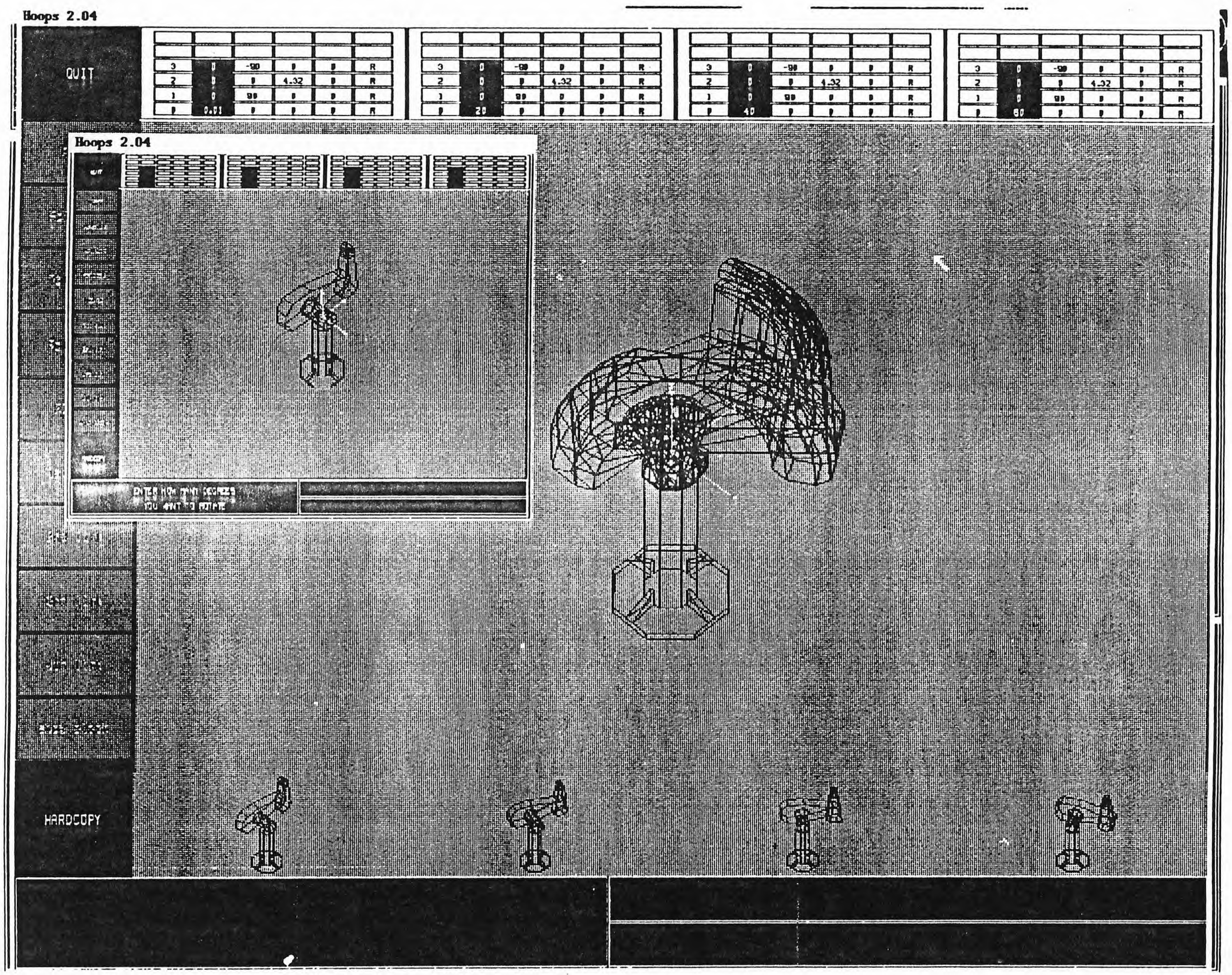


Fig. 5.1-4(b) The simulation of PUMA robot.
The bottom shows four configurations of the robot during the sweep motion.

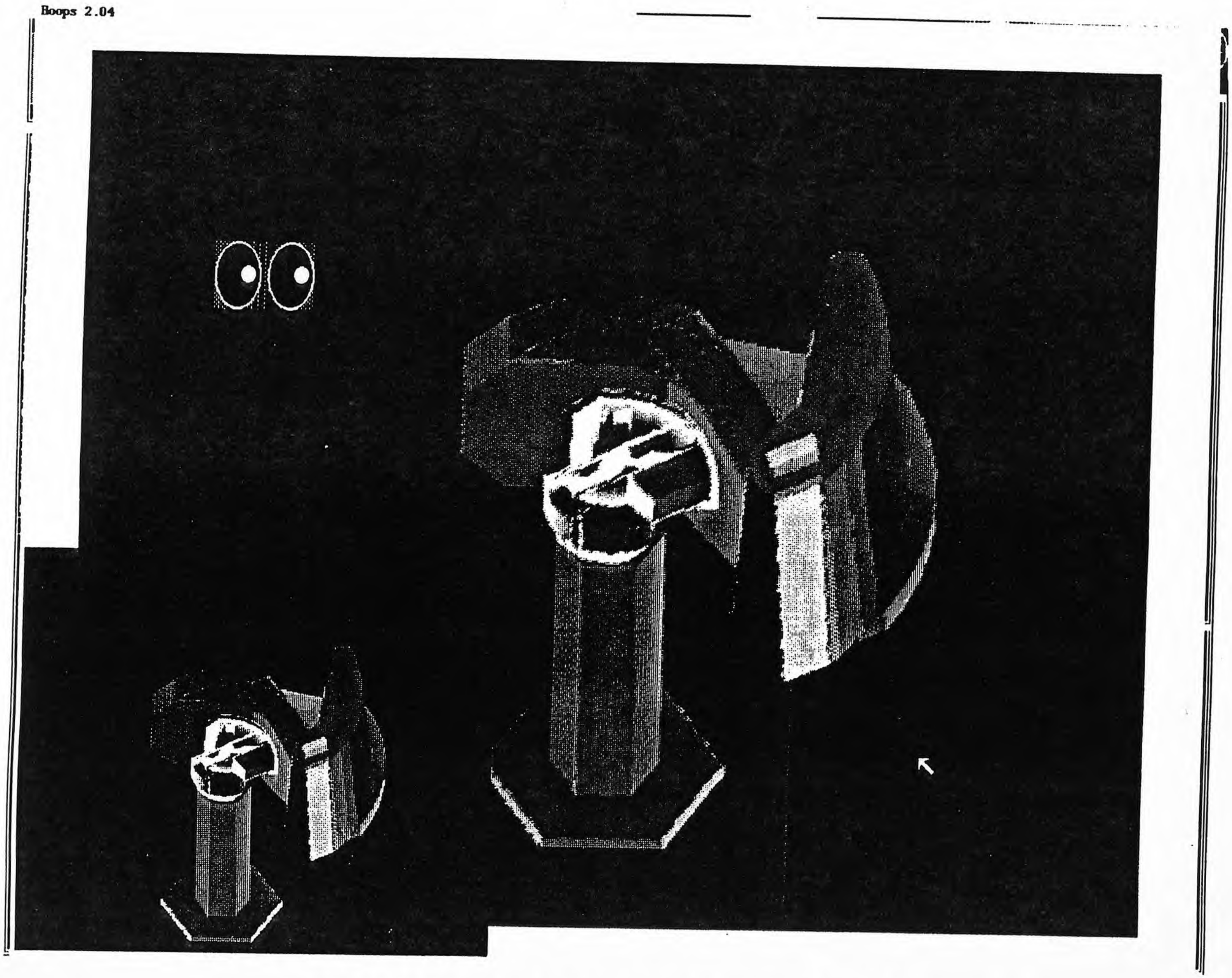


Fig. 5.1-5 Shaded image of the swept volume of Fig. 5.1-4.

5.2 Sweep Differential Equation Implementation

This section describes an implementation of chapter 4. We categorize the pure translational motion and pure rotational motion as autonomous sweep motions, and the autonomous motion in reference translating coordinates as relatively autonomous sweep motion.

The simulation algorithm is similar to the one in section 5.1; however instead of using parametric cubic equation to calculate the motion of the 'body attached coordinate frame', we use sweep differential equations.

Algorithm 1

We first sweep the 'body attached coordinate frame' by using sweep differential equation, then transform the solid vertex according to the differential equation.

for t = 1 to $\#T_k$

$$FR\begin{bmatrix} p(t) \\ o(t) \end{bmatrix} = \text{sweep motion. (see next section).}$$

end of t.

Algorithm 2

The same as the algorithm 2 in section 5.1.

5.2.1. Rolling, Pitching, and Yawing

In section 2.2 and 3.2, Euler angle representation of a rotation matrix was explained in detail. We choose rolling, pitching and yawing angle to represent Euclidean motion. The following provides the equations for rolling, pitching and yawing motion.

$$\begin{pmatrix} x_R(t) \\ y_R(t) \\ z_R(t) \end{pmatrix} = \begin{pmatrix} \cos \omega_R t & -\sin \omega_R t & 0 \\ \sin \omega_R t & \cos \omega_R t & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_R(0) \\ y_R(0) \\ z_R(0) \end{pmatrix} \tag{5.2-1}$$

$$\begin{pmatrix} x_P(t) \\ y_P(t) \\ z_P(t) \end{pmatrix} = \begin{pmatrix} \cos \omega_P t & 0 & \sin \omega_P t \\ 0 & 1 & 0 \\ -\sin \omega_P t & 0 & \cos \omega_P t \end{pmatrix} \begin{pmatrix} x_P(0) \\ y_P(0) \\ z_P(0) \end{pmatrix} \tag{5.2-2}$$

$$\begin{pmatrix} x_Y(t) \\ y_Y(t) \\ z_Y(t) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \omega_Y t & -\sin \omega_Y t \\ 0 & \sin \omega_Y t & \cos \omega_Y t \end{pmatrix} \begin{pmatrix} x_Y(0) \\ y_Y(0) \\ z_Y(0) \end{pmatrix} \tag{5.2-3}$$

Now, there is an interesting question: how can one use the sweep differential equation to explain the RPY motion in the Euclidean space ? From the question above, the above rotation matrix will be back traced to the sweep differential equations model, in order to categorize RPY motion as autonomous sweeping motion. Let's take Eq. 5.2-1 which is the rotation matrices of rolling. By using Laplace transform, one can get the swept differential equation form of Eq.5.2-1.

$$\begin{pmatrix} x_R(S) \\ y_R(S) \\ z_R(S) \end{pmatrix} = \begin{pmatrix} \frac{S}{\omega_R^2 + S_R^2} & \frac{-\omega_R}{\omega_R^2 + S_R^2} & 0 \\ \frac{\omega_R}{\omega_R^2 + S_R^2} & \frac{S}{\omega_R^2 + S_R^2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_R(0) \\ y_R(0) \\ z_R(0) \end{pmatrix} \tag{5.2-4}$$

$$\begin{pmatrix} SX_R - x_R(0) \\ SY_R - y_R(0) \\ SZ_R - z_R(0) \end{pmatrix} = \begin{pmatrix} 0 & -\omega_R & 0 \\ \omega_R & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} X_R \\ Y_R \\ Z_R \end{pmatrix} \tag{5.2-5}$$

$$\dot{X}_R = \Delta X_R + c_o \tag{4.2-2}$$

$$\dot{X}_R = \Delta X_R + c_o = \begin{pmatrix} 0 & -\omega_R & 0 \\ \omega_R & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_R(t) \\ y_R(t) \\ z_R(t) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{5.2-6}$$

Similarly pitching and yawing have the same property, then we can prove that Roll, pitch, and yaw are autonomous sweep motion.

1) $\Delta_R = \begin{pmatrix} 0 & -\omega_R & 0 \\ \omega_R & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ 2) $\Delta_P = \begin{pmatrix} 0 & 0 & -\omega_P \\ 0 & 0 & 0 \\ \omega_P & 0 & 0 \end{pmatrix}$ 3) $\Delta_Y = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -\omega_Y \\ 0 & -\omega_Y & 0 \end{pmatrix}$

5.2.2. IBM 7540 Robot and Two-Link Mechanism

Two-Link Mechanism

This section starts with the equations of two links and analyzes the two links using the sweep differential equation representation. We classify different types of motion in order to choose the proper equations to generate swept volume efficiently.

$$\begin{pmatrix} x_1(t) \\ y_1(t) \\ z_1(t) \end{pmatrix} = \begin{pmatrix} \cos\omega_1 t & -\sin\omega_1 t & 0 \\ \sin\omega_1 t & \cos\omega_1 t & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1(0) \\ y_1(0) \\ z_1(0) \end{pmatrix} \qquad (5.2\text{-}1)$$

$(x_1(0) , y_1(0) , z_1(0)) = (0 , 0 , 0)$ is at the origin of upperarm coordinate frame. By laplace transform, we get the linear system equation:

$$\begin{pmatrix} x_1(S) \\ y_1(S) \\ z_1(S) \end{pmatrix} = \begin{pmatrix} \dfrac{S}{\omega_1^2 + S_1^2} & \dfrac{-\omega_1}{\omega_1^2 + S_1^2} & 0 \\ \dfrac{\omega_1}{\omega_1^2 + S_1^2} & \dfrac{S}{\omega_1^2 + S_1^2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1(0) \\ y_1(0) \\ z_1(0) \end{pmatrix} \qquad (5.2\text{-}7)$$

$$SX_1 - \omega_1 Y_1 = x_1(0) \qquad (5.2\text{-}8a)$$
$$SY_1 - \omega_1 X_1 = y_1(0) \qquad (5.2\text{-}8b)$$
$$SZ_1 = z_1(0) \qquad (5.2\text{-}8c)$$

$$\begin{pmatrix} SX_1 - x_1(0) \\ SY_1 - y_1(0) \\ SZ_1 - z_1(0) \end{pmatrix} = \begin{pmatrix} 0 & -\omega_1 & 0 \\ \omega_1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix} \qquad (5.2\text{-}8)$$

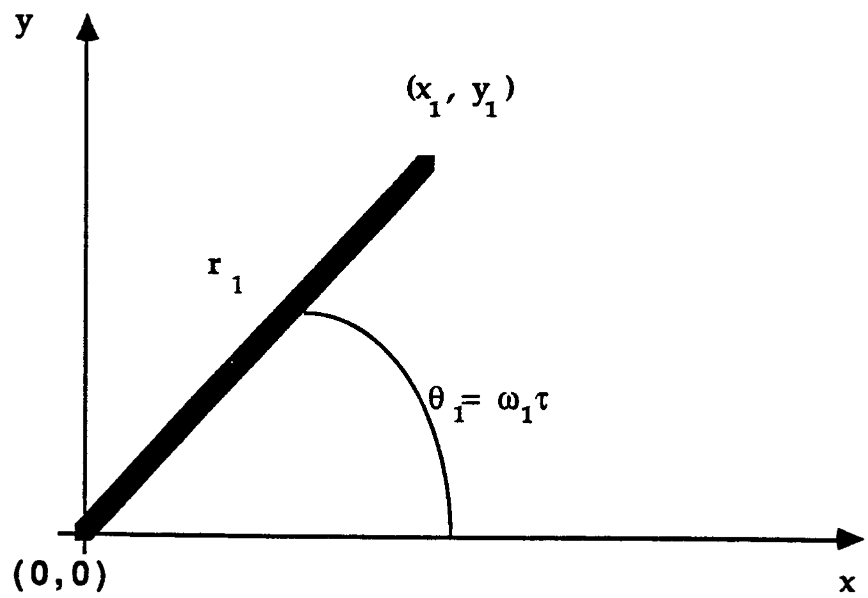


Fig. 5.2-1 Rotation of single link.

The final goal of this derivation is to derive an equation of the format of autonomous equation, which was shown in section 4.2.

$$\dot{X}_1 = \Delta X_1 + c_0 \qquad (4.2\text{-}2)$$

$$\dot{X}_1 = \Delta X_1 + c_0 = \begin{pmatrix} 0 & -\omega_1 & 0 \\ \omega_1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1(t) \\ y_1(t) \\ z_1(t) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \qquad (5.2\text{-}6)$$

This is a typical example of autonomous sweep equation. The homogeneous term of the first-order linear system (Eq 4.2-5) is not a function of t. In autonomous sweeping motion, the sweep vector field does not explicitly depend on t. [38] has similar discussion about this pure rotational motion of a two revolute joint mechanism.

The equation of the second link is:

$$X_2 = \begin{pmatrix} \cos\omega_2 t & -\sin\omega_2 t & 0 \\ \sin\omega_2 t & \cos\omega_2 t & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_2(0) \\ y_2(0) \\ z_2(0) \end{pmatrix} + \begin{pmatrix} \cos\omega_1 t & -\sin\omega_1 t & 0 \\ \sin\omega_1 t & \cos\omega_1 t & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1(0) \\ y_1(0) \\ z_1(0) \end{pmatrix} \qquad (5.2\text{-}9)$$

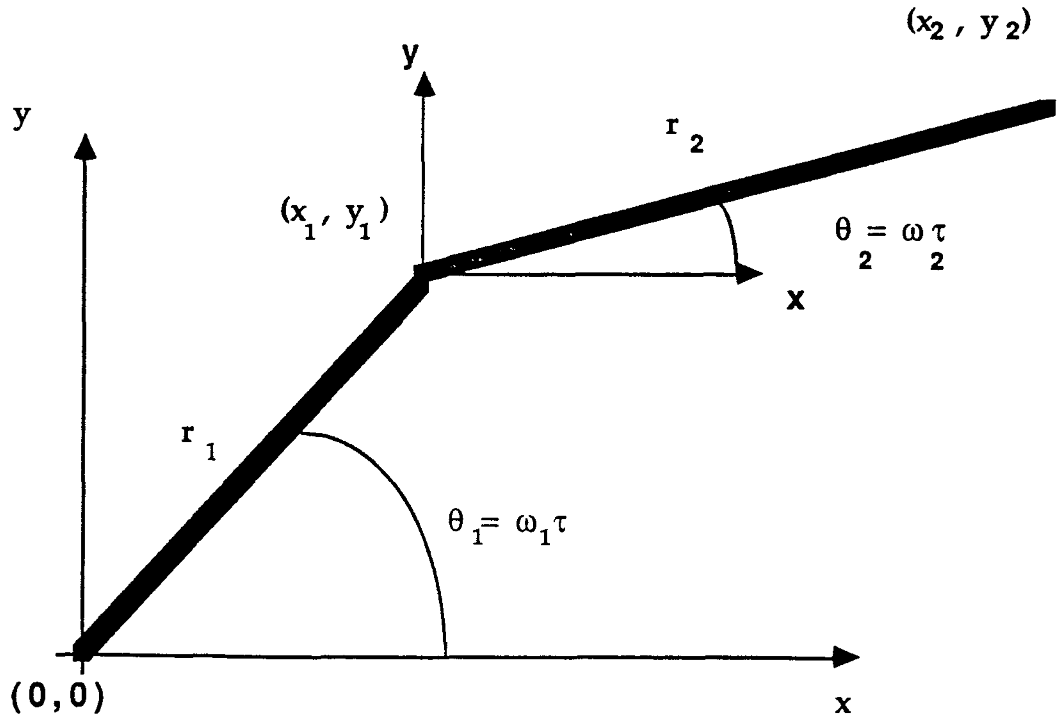


Fig. 5.2-2 Two links motion.

From linear differential equation point of view, the autonomous sweep differential equation consists of two parts, namely ${}^1X_2$, X_1

$$X_2(t) = {}^1X_2(t) + X_1(t) \tag{5.2-10}$$

Now, this format looks similar to the solution of the first-order linear system,

$$X_1(t) = \begin{pmatrix} \cos \omega_1 t & -\sin \omega_1 t & 0 \\ \sin \omega_1 t & \cos \omega_1 t & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1(0) \\ y_1(0) \\ z_1(0) \end{pmatrix} \tag{5.2-11}$$

$${}^1X_2(t) = \begin{pmatrix} {}^1x_2(t) \\ {}^1y_2(t) \\ {}^1z_2(t) \end{pmatrix} = \begin{pmatrix} \cos \omega_2 t & -\sin \omega_2 t & 0 \\ \sin \omega_2 t & \cos \omega_2 t & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} {}^1x_2(0) \\ {}^1y_2(0) \\ {}^1z_2(0) \end{pmatrix} \tag{5.2-12}$$

$x_2(t)$, $y_2(t)$, $z_2(t)$ is the distal point on the second link. $x_1(t)$, $y_1(t)$, $z_1(t)$ is the distal joint point on first link.

Similar to the derivation before, by taking the Laplace transform of Eq. 5.2-8 and Eq. 5.2-9, one can get the following differential equation, with initial condition $(x_1(0), y_1(0), z_1(0)) = (r_1, 0, 0)$, $({}^1x_2(0), {}^1y_2(0), {}^1z_2(0)) = (r_2, 0, 0)$, where r_2 is a point on the second link.

$$\dot{X}_1 = \begin{pmatrix} \dot{x}_1(t) \\ \dot{y}_1(t) \\ \dot{z}_1(t) \end{pmatrix} \tag{5.2-13}$$

$${}^1\dot{X}_2(t) = \begin{pmatrix} 0 & -\omega_2 & 0 \\ \omega_2 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} {}^1x_2(t) \\ {}^1y_2(t) \\ {}^1z_2(t) \end{pmatrix} \tag{5.2-14}$$

$$\dot{X}_2 = \begin{pmatrix} 0 & -\omega_2 & 0 \\ \omega_2 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} {}^1x_2(t) \\ {}^1y_2(t) \\ {}^1z_2(t) \end{pmatrix} + \begin{pmatrix} \dot{x}_1(t) \\ \dot{y}_1(t) \\ \dot{z}_1(t) \end{pmatrix} \tag{5.2-15}$$

From equation 4.1-3 we have the sweep differential equation:

$$\dot{x} = \dot{\xi}(t) + \dot{A}(t)\, x^0 \qquad (\dot{\ } = d/dt) \tag{4.1-3}$$

From the definition of 4.3, we have $\dot{A}A^T = B$ as constant; here we have A Eq. 5.2-12

$$A = \begin{pmatrix} \cos\omega_2 t & -\sin\omega_2 t & 0 \\ \sin\omega_2 t & \cos\omega_2 t & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

so

$$\dot{A}A^T = \omega_2 \begin{pmatrix} -\sin\omega_2 t & -\cos\omega_2 t & 0 \\ \cos\omega_2 t & -\sin\omega_2 t & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\omega_2 t & \sin\omega_2 t & 0 \\ -\sin\omega_2 t & \cos\omega_2 t & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & -\omega_2 & 0 \\ \omega_2 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \text{Constant} \tag{5.2-16}$$

The above equation shows that the second link motion is relatively autonomous motion.

Now let's refer to Fig. 5.2-2, If ω_1 equals to ω_2, from the definition in section 4.2, the two links should undergo autonomous motion. Because $\dot{A}A^T = B$ is constant and $A^T\dot{\xi} = b$ is constant.

$$\xi = \begin{pmatrix} \cos\omega_1 t & -\sin\omega_1 t & 0 \\ \sin\omega_1 t & \cos\omega_1 t & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.2-11}$$

$$A^T\dot{\xi} = \begin{pmatrix} \cos\omega_2 t & \sin\omega_2 t & 0 \\ -\sin\omega_2 t & \cos\omega_2 t & 0 \\ 0 & 0 & 1 \end{pmatrix} \omega_1 \begin{pmatrix} -\sin\omega_1 t & -\cos\omega_1 t & 0 \\ \cos\omega_1 t & -\sin\omega_1 t & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \omega_1 \begin{pmatrix} -\sin(\omega_1-\omega_2)t & -\cos(\omega_1-\omega_2)t & 0 \\ \cos(\omega_1-\omega_2)t & -\sin(\omega_1-\omega_2)t & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.2-17}$$

if $\omega_1 = \omega_2$ then

$$A^T\dot{\xi} = \begin{pmatrix} 0 & -\omega_2 & 0 \\ \omega_2 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \text{Constant} \tag{5.2-18}$$

The above derivation shows that if ω_1 equals to ω_2 the relatively autonomous sweep motion become autonomous motion; in the other words, Eq. 5.2-17 and Eq. 5.2-18 show that autonomous sweep motion is only a special case of relatively autonomous motion. Eq. 5.2-17 also shows that the relatively autonomous motion has a translational term which is a function of t.

A similar engineering application to the two-link case is IBM 7540 robot, which is a three-degree-of-freedom robot (two rotational joints and one prismatic joint). The first two degrees of freedom are manipulated by the upperarm and forearm, and the third degree of freedom is manipulated by an prismatic joint which is the end-effector.

IBM 7540 Robot

The following is the model of IBM 7540 robot links. This model includes three parts :

I) The upper-arm.

From the kinematics analysis, the upper arm motion can be categorized as autonomous motion. The whole upper robot arm undergoes rotational motion.

II) The fore-arm.

The fore-arm motion can be categorized as partial autonomous motion. The forearm undergoes rotational and translational motion.

Now take a look at two different forearm movement simulations from Fig. 5.2-3 to Fig. 5.2-6. By comparing the different sweep simulation results, from Fig.5.2-5 and Fig.5.2-6 one can see exactly the sweep vector field (the sweep field lines) which intersect itself. Fig. 5.2-6 shows that the swept surfaces are not generated only by its polygon edges, but also by polygon facets in the form of developable surfaces.

III) The end-effector.

For the end-effector, the model of forearm still holds but is under different initial conditions ($x_1(0)$, $y_1(0)$, $z_1(0)$) = (r_1 , 0 , 0), (${}^1x_2(t)$, ${}^1y_2(t)$, ${}^1z_2(t)$) = (r_2 , 0 , 0). For this prismatic joint there exists a velocity in the Z axis direction.

$$\dot{X}_3 = \begin{pmatrix} 0 & -\omega_2 & 0 \\ \omega_2 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} {}^1x_2(t) \\ {}^1y_2(t) \\ {}^1z_2(t) \end{pmatrix} + \begin{pmatrix} \dot{x}_1(t) \\ \dot{y}_1(t) \\ \dot{z}_3(t) \end{pmatrix} \qquad (5.2\text{-}19)$$

The translational term $\dot{z}_3(t)$, describes the up and down translational motion of the end-effector.

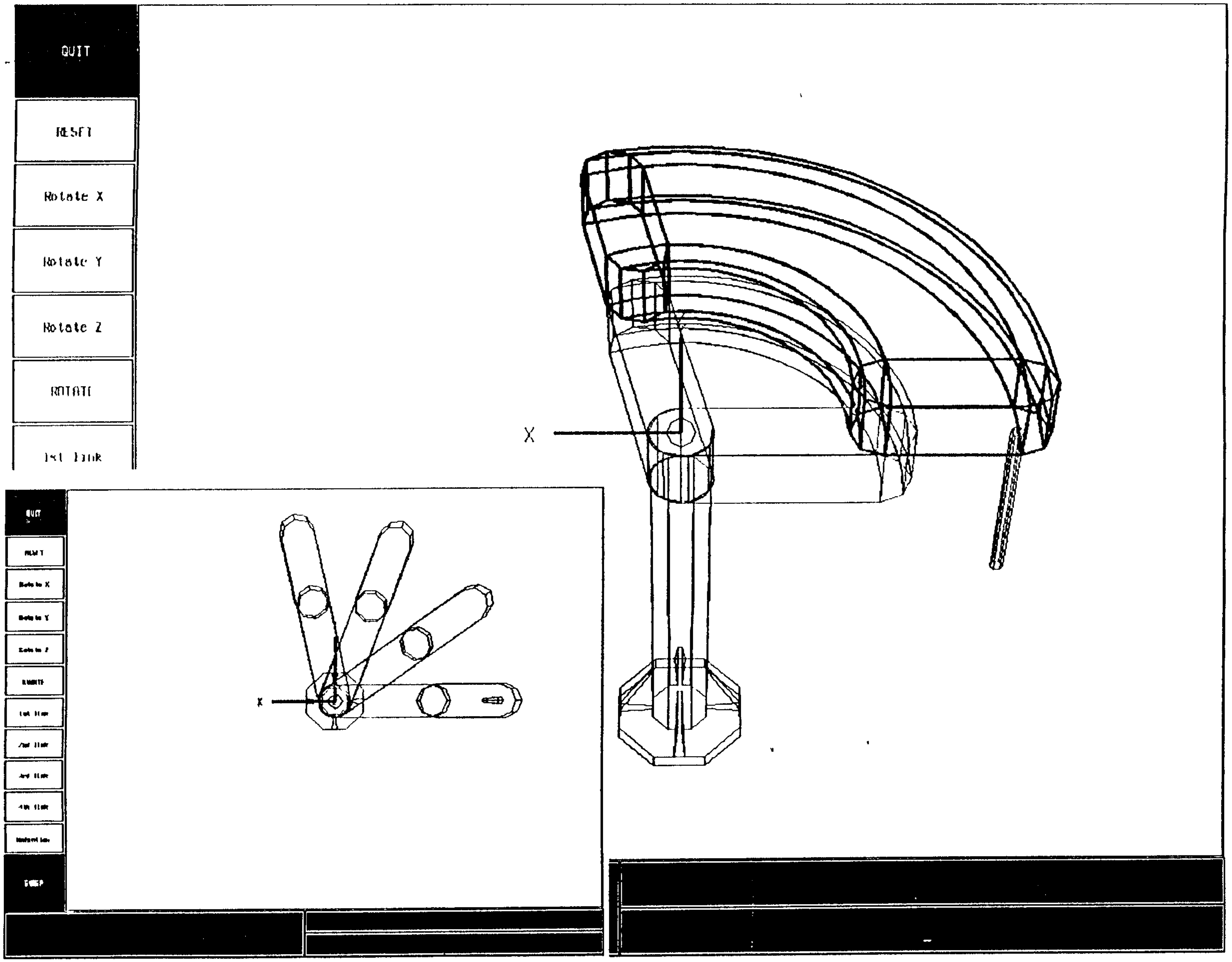


Fig. 5.2-3 The simulation of IBM robot using SDE.
This is a special case of relatively autonomous sweep motion, for $\omega_1 = \omega_2$.

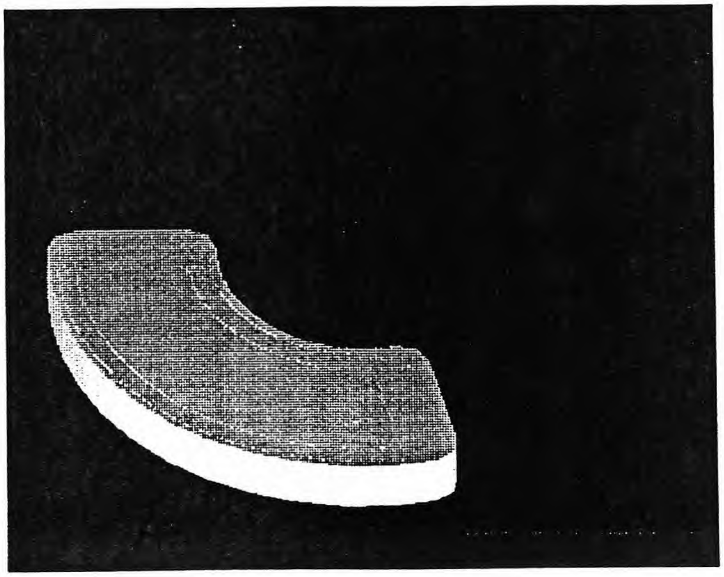


Fig. 5.2-4 (a) Shaded image of the swept volume of the second link.

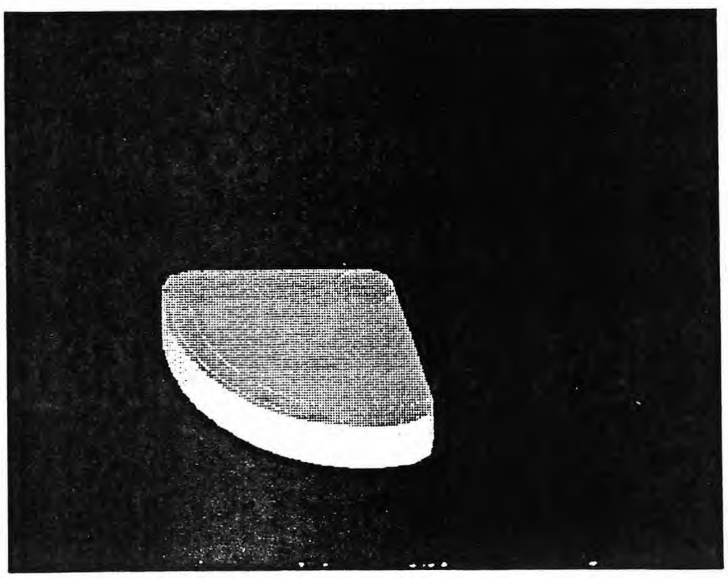


Fig. 5.2-4(b) Shaded image of the swept volume of the first link.

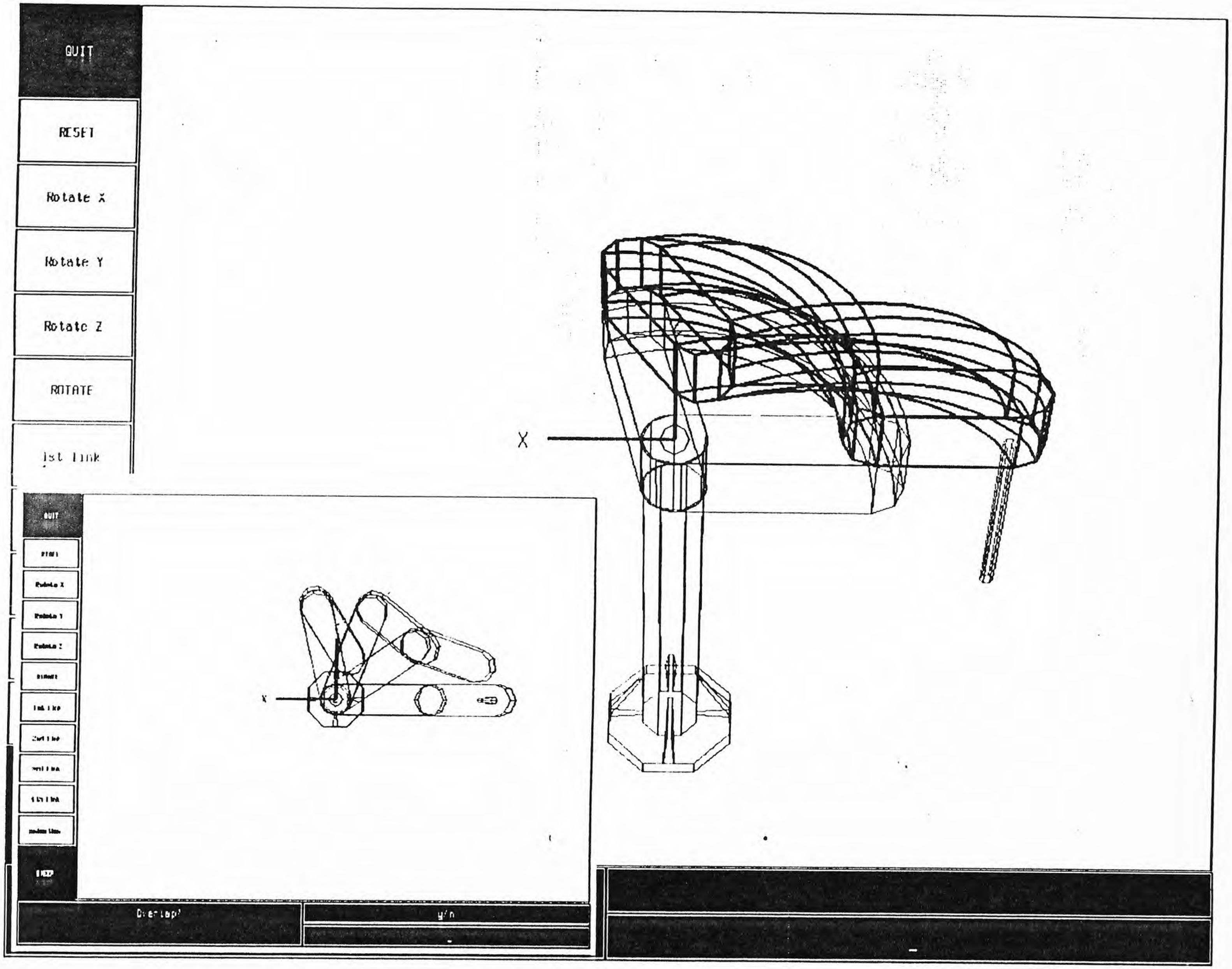


Fig. 5.2-5 The simulation of IBM robot using SDE. This is relatively autonomous sweep motion. The fore-arm swept volume should be partly bounded by developable surfaces (not shown).

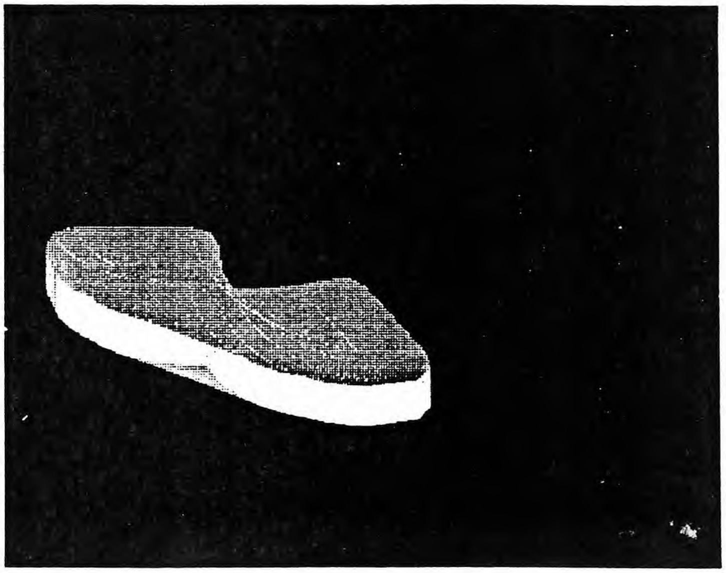


Fig. 5.2-6 (a) Shaded image of the swept volume of the second link.

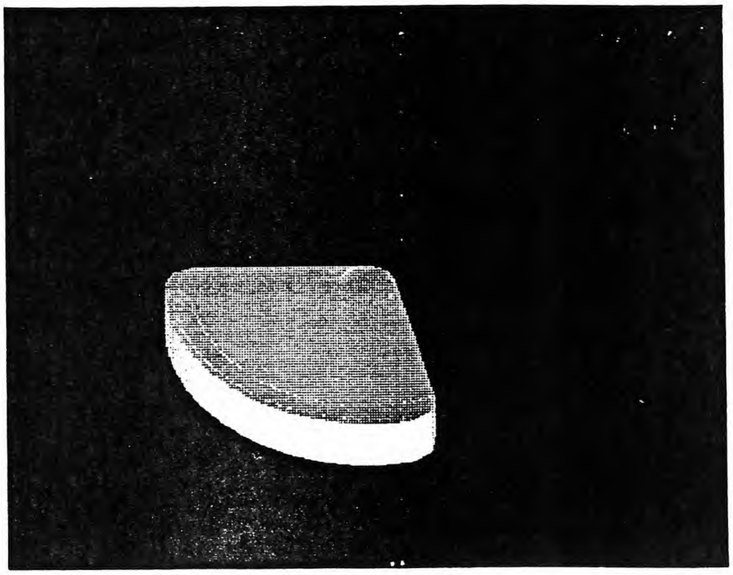


Fig. 5.2-6 (b) Shaded image of the swept volume of the first link.

5.3 Shaded Image Representation

Shaded image representation needs complicated programming algorithms, including reflectance calculation, hidden surfaces removal, and color rendering techniques. This section discusses reflectance calculation and hidden surface removal of ruled surface. There is no discussion on color rendering, which is one of HOOPS bulid-in functions.

We use scan-line algorithm to detect the intersections between polyhedral facet edges and scan lines, and use z-buffer to remove hidden surfaces. For calculating reflectance intensity we use linear interpolation method to get the smooth surface normal change. There are other algorithms; see [1][4][8][9][10].

The interfacing data between shaded image representation and wire-frame representation in our simulation software is the array SV[facet; edge; xyz(t)]. In other words, shading programs transfer the wire-frame data (3-D Cartesian space) to shaded pictures (2-D computer screen). Some initial conditions have to be determined before stating the shading calculations. They are 1) eyesight direction (the default eye position is the origin, looking in the negative Z direction), 2) light source direction, 3) surface reflection constant (experience value).

As the eye location has been determined, only the "top facets" can be seen. Shading programs calculate the normals of all facets. Z-buffer algorithm compares the z depth of each facet, and save the data which are close to the eye position. An illumination subroutine is used to calculate the reflectance.

Fig 5.3-1 shows the flow of program execution. All blocks are named in computer program files in a SUN 3/60 workstation system at NJIT.

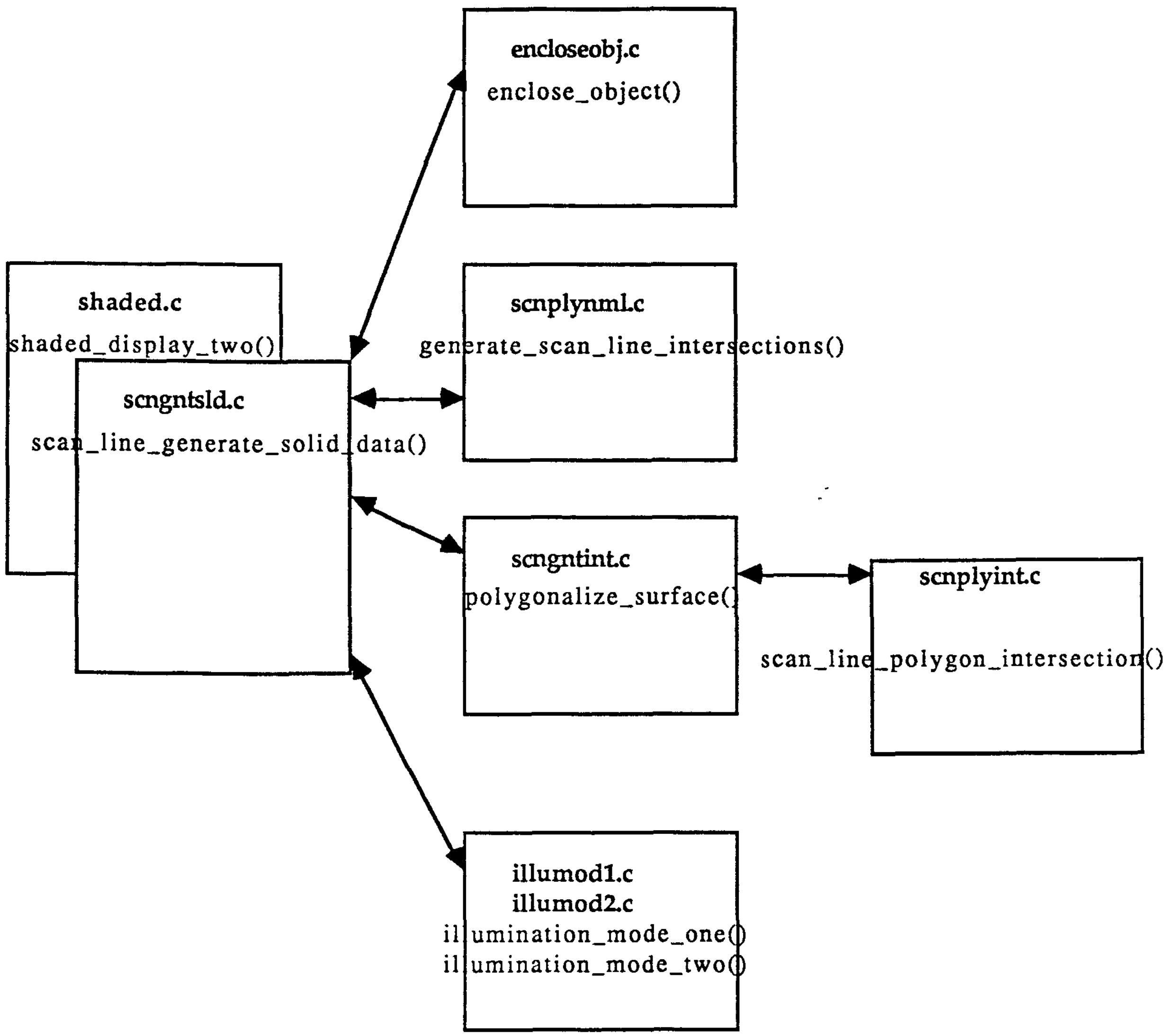


Fig 5.3-1 The block diagram of shading programs.

The following symbols are used in the program subroutines:

$\#N_f$: The number of the facets of a solid object.
$\#N_e$: The number of edges of a facet.
$\#T_k$ = 13 : The number of instants when t changes from 0 to 1.

$$SV\begin{bmatrix} & x \\ \text{face; edge;} & y \\ & z \end{bmatrix} = SV[\ \text{face; edge; } p(t)] : \text{Swept volume boundary points.}$$

$$IM\begin{bmatrix} x=512 \\ y=410 \end{bmatrix} : \text{Shaded image reflectance values.}$$

$\mathrm{Max}\left[\mathrm{face}\,;\mathrm{edge}\,;\begin{matrix}x\\y\end{matrix}\right]$: Maximum coordinate of each ruled surface.

$\mathrm{Min}\left[\mathrm{face}\,;\mathrm{edge}\,;\begin{matrix}x\\y\end{matrix}\right]$: Minimum coordinate of each ruled surface.

$\mathrm{PL\_Vtx}\left[\#T_k{*}\#N_f\,;\mathrm{edge}\,;\begin{matrix}x\\y\\z\end{matrix}\right]$: Polygonalized $\mathrm{SV}\left[\mathrm{face}\,;\mathrm{edge}\,;\begin{matrix}x(t)\\y(t)\\z(t)\end{matrix}\right]$.

$\mathrm{PL\_Nml}\left[\#T_k{*}\#N_f\,;\begin{matrix}x\\y\\z\end{matrix}\right]$: Normal of $\mathrm{PL\_Vtx}\left[\#T_k{*}\#N_f\,;\mathrm{edge}\,;\begin{matrix}x\\y\\z\end{matrix}\right]$.

$\mathrm{Vtx\_Nml}\left[20\,;\begin{matrix}x\\y\\z\end{matrix}\right]$: Vertex normal of each polygonal facet.

shaded.c

Shaded.c gets the sweep data (SV[face; edge; xyz; t]) from the simulation programs. Then it calls SCAN_LINE_GENERATE_SOLID_DATA() to process the wire-frame sweep data, After the processing the results are stored in array IM[x;y]. Here the shaded picture size is 512 by 410 pixels, each value in IM[x;y] represents a pixel intensity value (form 0 to 20). ('pixel' is the size of a computer screen spot unit). In the end, SHADED_DISPLAY_TWO() calls the HOOPS computer graphic routines to render the intensity values in IM[x; y] on computer screen.

In summary, this subroutine does the following:

SHADED_DISPLAY_TWO()
{ call SCAN_LINE_GENERATE_SOLID_DATA()
call HOOP subroutines to display the image data. }

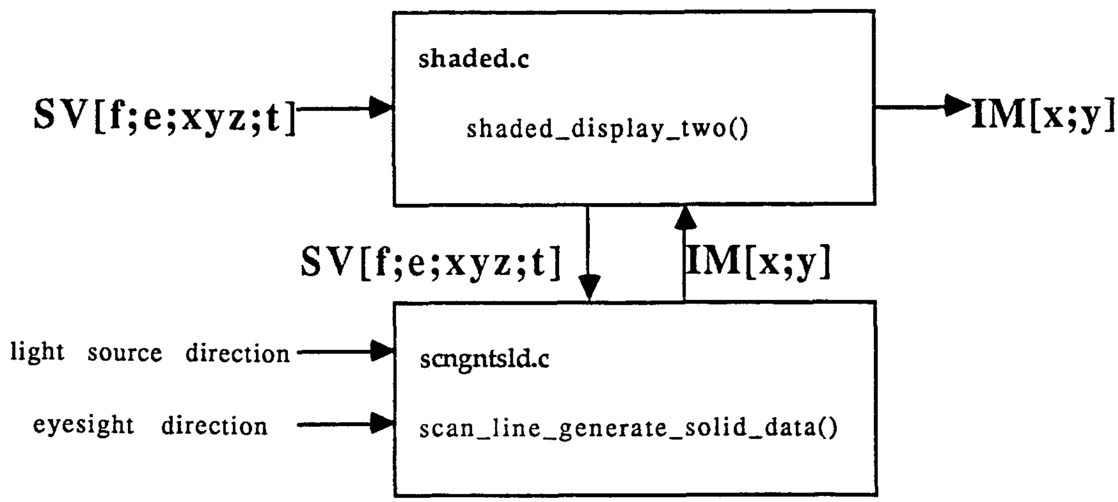


Fig. 5.3-2 Block diagram of shaded.c and scngntsld.c.

scngntsld.c

This subroutine defines light source direction and eyesight direction, then calls ENCLOSE_OBJECT() to find the enclosing rectangle boundaries for each ruled surface.

The second step is to call POLYGONALIZE_SURFACES() to polygonalize the ruled surface segments and calculate their unit normals (include the initial and final polygons).

The third step is to call GENERATE_SCANLINE_INTERSECTION(), which renders the normal value to the ruled surfaces and the initial and final polygons. The subroutine uses scan line/Z-buffer algorithm to record the visible surface/ray intersections and maps the surface reflectance to proper pixels.

The last step is to call either ILLUMINATION_MODE_ONE() or ILLUMINATION_MODE_TWO() to calculate the reflectance of the ruled surfaces.

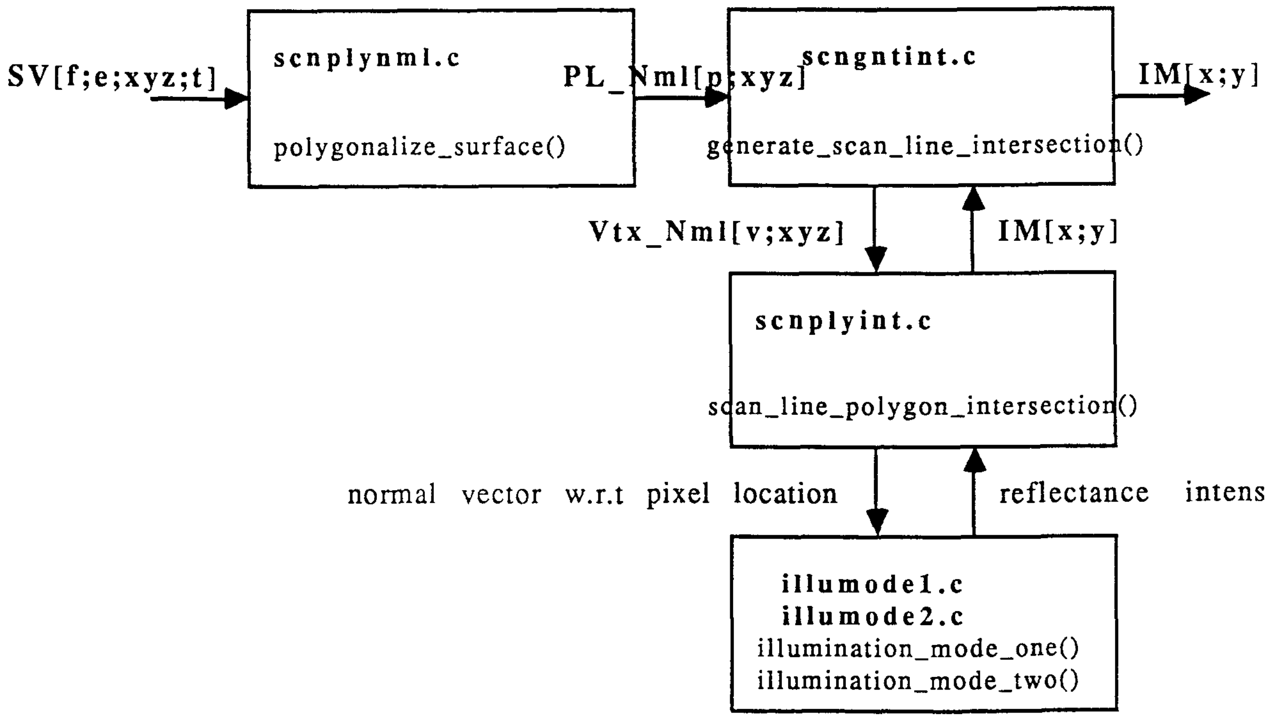


Fig. 5.3-3 Block diagram of scnplynml.c, scngntint.c, scnplyint.c & illumode.c

enclosebj.c

This subroutine finds the maximum and minimum value of every ruled surface. The pixels which do not map inside to the bounding rectangle are treated as background (the reflectance intensity is 0).

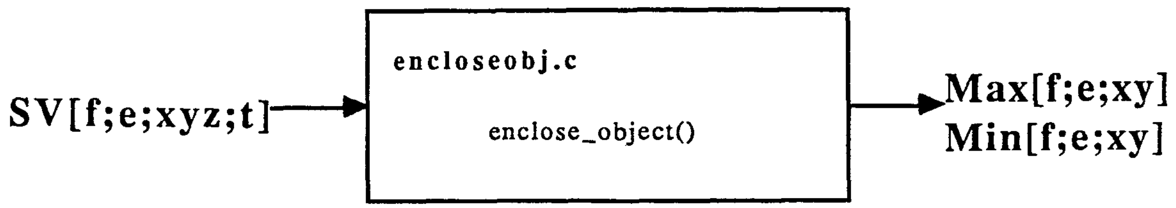


Fig. 5.3-4 Block diagram of encloseobj.c.

scnplynml.c

This subroutine is to polygonalize the SV[face; edge; xyz ;t] to PL_Vtx[face*#T_k+t; xyz; edge] and to generate the normal of each polygon, in similar structure PL_Nml[face*#N_f+t; xyz; edge].

for face = 1 to #N_f
  for edge = 1 to #N_e
    for t = 1 to #T_k

$$PL\_Vtx\left[face{*}\#T_k{+}t\,;edge\,;\begin{matrix}x\\y\\z\end{matrix}\right] = SV\left[face\,;edge\,;\begin{matrix}x(t)\\y(t)\\z(t)\end{matrix}\right].$$

number_polys = number_polys + 1

    end of t.
  end of edge.
end of face.

for i= 1 to number_polys

$$A\begin{pmatrix}x\\y\\z\end{pmatrix} = PL\_Vtx\left[face{*}\#T_k{+}t\,;1\,;\begin{matrix}x\\y\\z\end{matrix}\right] - PL\_Vtx\left[face{*}\#T_k{+}t\,;0\,;\begin{matrix}x\\y\\z\end{matrix}\right]$$

j = 0

while (lines_not_parallel)

$$\{ \quad B\begin{pmatrix}x\\y\\z\end{pmatrix} = PL\_Vtx\left[i\,;j\,;\begin{matrix}x\\y\\z\end{matrix}\right] - PL\_Vtx\left[i\,;0\,;\begin{matrix}x\\y\\z\end{matrix}\right]$$

j = j+1

$$PL\_Nml\left[i\,;\begin{matrix}x\\y\\z\end{matrix}\right] = \frac{A\begin{pmatrix}x\\y\\z\end{pmatrix} \times B\begin{pmatrix}x\\y\\z\end{pmatrix}}{\sqrt{A\begin{pmatrix}x\\y\\z\end{pmatrix}\bullet B\begin{pmatrix}x\\y\\z\end{pmatrix}}} \quad \}$$

end of i.

scngntint.c

This subroutine uses geometrical simplification to subdivide the total number of scan lines needed to process the object movement into 10 increments. The scan-line algorithm decides what polygons are visible in a scan-line window, and these decisions are made by comparing line segments in the x-z plane.

If one uses only the polygon normal of the ruled surface to calculate the reflectance, the shaded picture can show a sculpture like surface. So we need to get the normal of every vertex, then use linear interpolation algorithm to smooth out the ruled surface. The following figure show the concepts of getting the vertex normal form the surrounding polygonal facets, the vertex 1 is surrounded by polygon A,B, and C.

Fig. 5.3-5 shows the ideas of getting the vertex normal. But in the program each swept surface generated by polyhedral edge sweeping will be in rectangular shape, but three points will decide a plane. We have to sub-divide the polygon into triangular facets. Because the default number of sweeping instant is thirteen, there should be twelve segments for each swept curve. In the other word, there should be twelve rectangular polygon which can determine twenty-four triangular facets on each ruled surface.

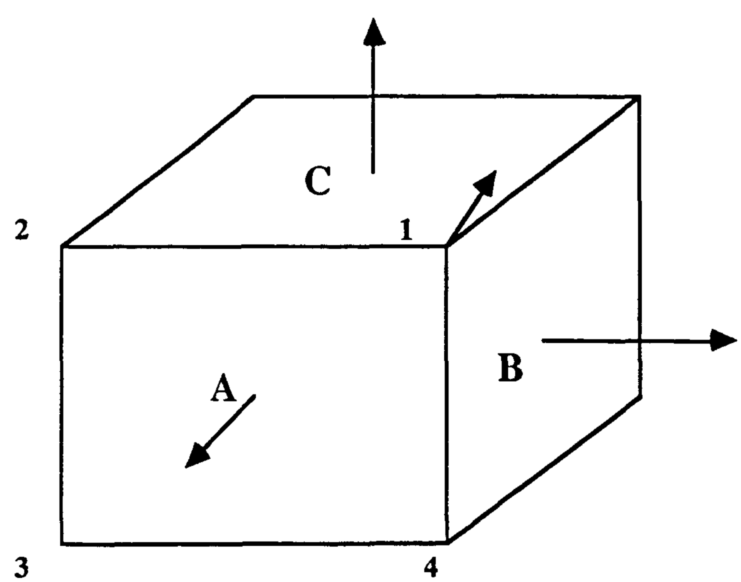


Fig. 5.3-5 Three polygons determine a vertex and its vertex normal.

Refer to Fig. 5.3-3 which shows the data flow between scnplynml.c, scngntint.c, scnplyint.c & illumode.c.

for p = 1 to number_polys

for v = 1 to number_of_each_polygonal_facet[p]

$$\text{Vtx\_Nml}\begin{bmatrix} & x \\ v\,; & y \\ & z \end{bmatrix} = \text{PL\_Nml}\begin{bmatrix} & x \\ A\,; & y \\ & z \end{bmatrix} + \text{PL\_Nml}\begin{bmatrix} & x \\ B\,; & y \\ & z \end{bmatrix} + \text{PL\_Nml}\begin{bmatrix} & x \\ C\,; & y \\ & z \end{bmatrix}$$

$$\text{Vtx\_Nml}\begin{bmatrix} & x \\ v\,; & y \\ & z \end{bmatrix} = \frac{\text{Vtx\_Nml}\begin{bmatrix} & x \\ v\,; & y \\ & z \end{bmatrix}}{\sqrt{\text{Vtx\_Nml}\begin{bmatrix} & x \\ v\,; & y \\ & z \end{bmatrix} \bullet \text{Vtx\_Nml}\begin{bmatrix} & x \\ v\,; & y \\ & z \end{bmatrix}}}$$

end of v.

call SCAN_LINE_POLYGON_INTERSECTION()

end of p.

scnplyint.c

A polygon is input to this subroutine, i.e. the coordinates of the vertices of the polygon, the polygon normal, and the vertex normals determined as the average of the polygon normals for all polygons sharing the vertices of the input polygon. These vertex normals are used to determine shading using the linear interpolated normals across the polygon. The polygon is displayed using scan-line z-buffer algorithm. For all the edges in a polygon determine the x-coordinate of intersection with the scan line.

Polygon is a convex object. Each pair of the x coordinate intersection are interior to the polygon.

This is the scan line parametric form :

$$x(t) = a_x + b_x t$$
$$y(t) = a_y + b_y t$$

This is the current polyhedral edges :

$$x(s) = c_x + d_x s$$
$$y(s) = c_y + d_y s$$

$$s = \frac{b_x*(c_y-a_y) - b_y*(c_x-a_x)}{(d_x*b_y - b_x*d_y)} \qquad t = \frac{c_x-a_x+s*d_x}{b_x}$$

After the scan line intersects with the edges, (in point E,F), the next step is to linearly interpolate the scan points normal value between points E,F (see Fig. 5.3-6), for example, the point D. The following equations show how we get the linear interpolation value for the point D of the plane α.

$$u = \frac{y_{min\_point} - y_E}{y_{min\_point} - y_{next\_min\_point}} \qquad w = \frac{y_{max\_point} - y_F}{y_{max\_point} - y_{next\_max\_point}}$$

$$Nml_E = u*Vtx\_Nml\left[min\_point \; ; \begin{matrix} x \\ y \\ z \end{matrix} \right] + (1-u)*Vtx\_Nml\left[next\_min\_point \; ; \begin{matrix} x \\ y \\ z \end{matrix} \right]$$

$$Nml_F = w*Vtx\_Nml\left[max\_point\,;\begin{matrix}x\\y\\z\end{matrix}\right] + (1-w)*Vtx\_Nml\left[next\_max\_point\,;\begin{matrix}x\\y\\z\end{matrix}\right]$$

$$\Delta Nml = \frac{Nml_E - Nml_F}{Number\_of\_Pixels}$$

$$Nml_D = Nml_E + j^* \Delta Nml$$

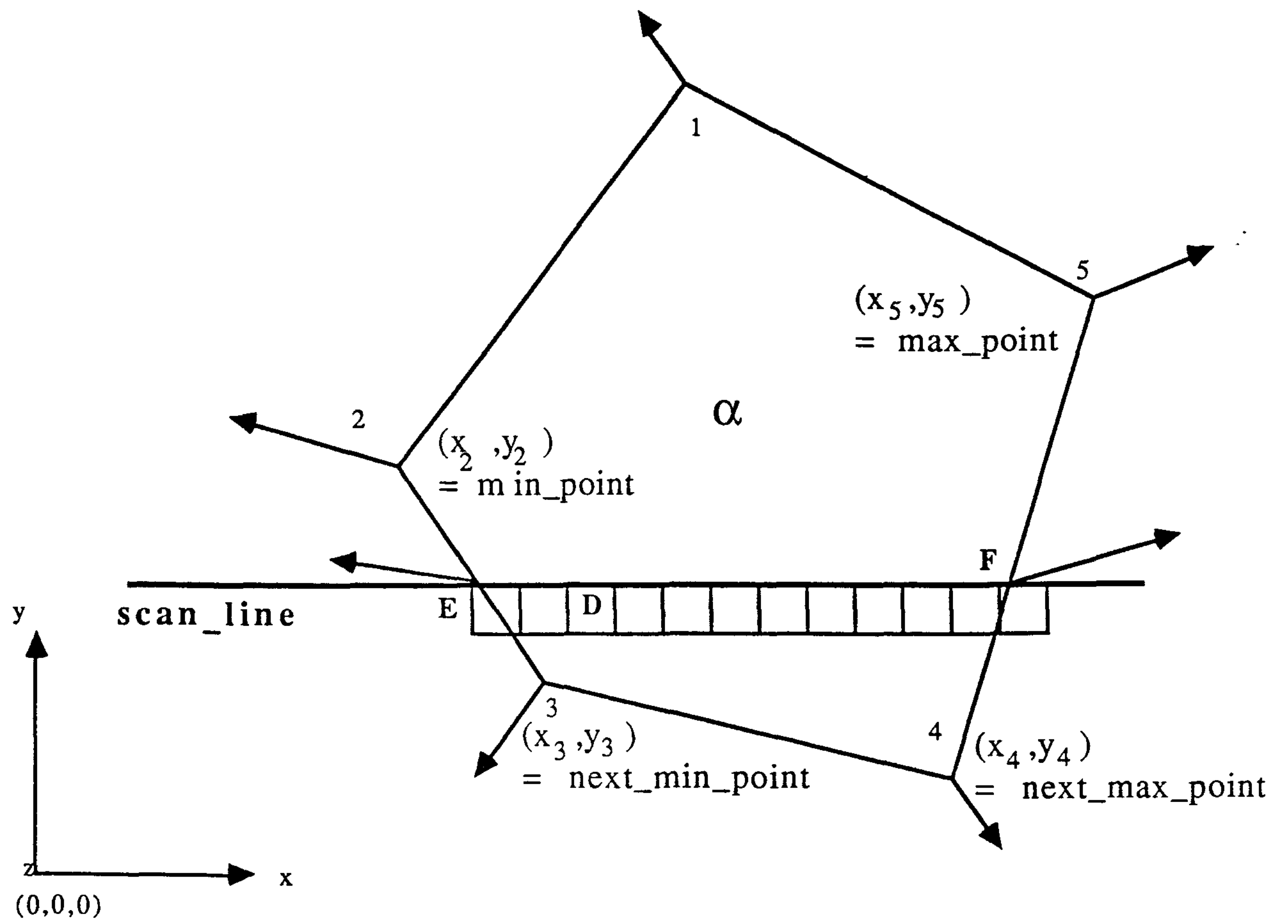


Fig.5.3-6 Linear interpolation of scan line normal values.

In the end of this subroutine compare the Z-buffer value with the previous value, and keep the top most set of normal value for illumination calculation.

illumod1.c or illumode2.c

The following reflectance formula was used as the illumination model, where all the parameters are constant except n^ which represents the surface normal at a pixel.

Reflectance Formula:

$$I = I_a * K_a + \frac{I_l}{d+K} [K_d*(n\wedge \bullet L\wedge) + K_s * (R\wedge \bullet S\wedge)^n]$$

I : reflected intensity.
I_a = incident ambient light intensity.
I_l = incident point source light intensity.
K_a = ambient diffuse reflection constant usually $0 \le K_a \le 1$.
K_d = diffuse reflection constant $0 \le K_d \le 1$.
K_s = experimental constant representing reflectance curve $w(i,\lambda)$.
d = distance from the closest object to the viewpoint.
K = arbitrary constant.
n = approximates spatial distribution of reflected light.
n^ = unit surface normal vector at current pixel.
L^ = unit light source direction vector.
R^ = unit reflected ray direction vector.
S^ = unit line-of-sight direction vector.

Chapter 6

CONCLUSION

This thesis presents computer graphic implementation of methods for generating geometric representations of swept volumes for polyhedral objects. The geometrical applications of our methods include autonomous swept volumes, relatively autonomous swept volumes, and parametric cubic equation.

Parametric cubic equation is used to approximate general Euclidean motions. Autonomous sweep differential equation is applied to simple translational and rotational motions in Euclidean space. Relatively autonomous sweep differential equation is used to describe the autonomous motion in relative coordinate system. These equations can be used to describe various kinds of machines or robots motions. The computer graphic simulation has been implemented on polyhedral objects, IBM robot, and PUMA robot. The results of computer simulation are helpful in visualizing motions of objects by showing their swept volumes in wire-frame and shaded images and helpful in visualizing some of the theories previously developed on swept volumes.

Although there are several types of motions being presented in the representation of swept volumes, these equations cover only very limited types of Euclidean motions. Moreover, the surface characteristics of swept volumes have not been much studied from the above formulations. Further research is needed to achieve these goals.

BIBLIOGRAPHY

[1] M.E. Mortenson, Geometric Modeling, John Wiley & Son, New York, 1985.

[2] Wang, W.P., Wang, K.K., "Real-Time Verification of Multiaxis NC Programs with Raster Graphics," Proceedings of 1986 IEEE International Conference on Robotics and Automation, April 1986.

[3] Sungurtekin, U.A., Voelcker, H.B., "Graphic Simulation & Automatic Verification of NC machine Programs," Proceedings of 1986 IEEE International Conference on Robotics and Automation, April 1986.

[4] Hanna, S.L., Abel, j.F., and Greenberg, D.P., "Intersection of Parametric Surfaces by means of Look-up Table," Journal of IEEE in Computer Graphics and Applications, pp39-48, Oct. 1983.

[5] Houghton, E.G., Emnett, R.F., Factor, J.D., Sabharwal, R.F., "Implementation of a Divide-and-Conquer methods for Intersection of Parametric Surfaces," Computer Aided Geometric Design, pp173-183, 1985.

[6] Boyse, W.J., "Interference Detection Among Solids and Surfaces," Communications of the ACM, vol. 22, No. 1, Jan. 1979.

[7] Lee, B.R., "Intersection of Parametric Surfaces and a Plane," Journal of IEEE in Computer Graphics and Applications, pp48-51, Aug. 84.

[8] I.D. Faux, M.J. Pratt, ., Computational Geometry for Design and Manufacture,Ellis Horwood Limited, England, 1979.

[9] Manfredio. Do Carmo, Differential Geometry of Curves and Surfaces, Prentice Hall,1976.

[10] K.S. Fu, R.C. Gonzalez, C.S.G. Lee, Robotics — Control, Sensing, Vision, and Intelligence, McGraw-Hill,1987.

[11] Weld, J.D. and Leu, M.C., "Geometric Representation of Swept Volumes with Application to Polyhedral Objects," Sibley School of Mechanical and Aerospace Engineering Technical Report No. MSD-87-03, Cornell University, Ithaca, New York, 1987.

[12] Melvin, J. Maron, Numerical Analysis — A Practical Approach, Macmillian Publishing Co., Inc. 1982.

[13] Coquillart, Sabine, "A Control-point-based Sweeping Technique," Journal of IEEE Computer Graphics and Applications, pp36-45, Nov. 87.

[14] Wang, W.P., Wang, K.K., "Geometric Modeling for Swept Volume of Moving Solids," Journal of IEEE Computer Graphic and Applications, pp8-17, Dec. 1986.

[15] Stocker, J.J., "Developable Surfaces in the Large," Communications on Pure and Applied Mathematics, Vol. 14, pp627-635, 1961.

[16] Stoker, J.J., Differential Geometry, Wiley-Interescience, 1969.

[17] Van Wijk, J.J., "Ray tracing subjects Defined by Sweeping a Sphere," Computers & Graphics, Vol. 9, No. 3 , pp283-290, 1985.

[18] O'Neill, B., Elementary Differential Geometry, Academic Press, New York, 1966.

[19] Paul, R.P., Robot Manipuklators: Mathematics, Programming, and Control, MIT Press, 1981.

[20] Brooks, R.A., "Planning Collision-Free Motions for Pick-and-Place Operations," The International Journal of Robotics Research.

[21] Baer, A., Eastman, C., and Henrion, M., "Geometric Modelling: A Survey," Computer-Aided Design, Vol. 11, No. 5, pp253-272, Sept. 1979.

[22] Blackett, D.W., Elementary Topology, A Combinational and Algebraic Approach, Academic Press, New York, 1982.

[23] Fridshal, R., Duncan, D., Cheng, K.D., and Zucher, W., "Numerical Control Part Program Verification System," Proceedings of MIT Conference on CAD/CAM Technology for Mechanical Engineering, Cambridge, MA, March 1982.

[24] Meagher, D.J., "Geometric Modeling Using Octree Encoding," Computer Graphics and Image Processing, pp129-147, Vol. 19, 1962.

[25] Munkres, J.R., Topology : A First Course, Prentices-Hall, Inc., Englewood Cilffs, New Jersy, 1975.

[26] Lozano-Perez, T., and Wesley, M.A., "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles," Communications of the ACM, Vol. 22, No. 10, Oct. 1979.

[27] Lozano-Perez, T., "Automatic Planning of Manipulator Transfer Movements," IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-11, No. 10,pp681-698, Oct. 1981.

[28] Lozano-Perez, T., "Spatial Planning: A Configuration Space Approach ," IEEE Transactions on Computers, Vol. C-32, No. 2, pp108-120, Feb. 1983.

[29] Rogers, D.F., Procedural Elements for Computer Graphics, McGraw-Hill Book Company, New York, 1985.

[30] Weld, J.D., "Geometric Representation of Swept Volumes with Application to Polyhedral Objects," Ph. D. dissertation, Cornell University, Ithaca, New York, 1987.

[31] Schwartz, J.T., Sharir, M., Hopcroft, J., Planning, Geometry & Complexity of Robot Motion, Ablex,1987.

[32] Korein, J.U., Ph. D. disseration, "A Geometric Investigation of Reach", U. of Pennsylvania, Pennsylvania, 1984.

[33] Ganter, M.A., Ph. D. disseration, "Dynamic Collision Dection Using Kinematics & Solid Modeling Technique", The U. of Wisconsin-Madison, Wisconsin,1985.

[34] Groover, M.P., Weiss, M., Nagel, R.N., Odrey, N.G., "Industrial Robot --- Technology,Programming,and Application", McGraw Hill, 1986.

[35] Strang, G. 1988. Linear Algebra and its Application, 3rd ed. New York; Harcourt-Brace Jovanovish.

[36] Blackmore, D., Leu, M.C., "A Differential Equation Approach to Swept Volumes", Proceedings of Second International Conference on Computer Intergrated Manufacturing, Rensselaer Polytechnic Institute, Troy, New York, May 21-23, 1990.

APPENDIX I

Programs for Graphic Representation of Swept Volumes

::::::::::::::
Draw_wire.cx
::::::::::::::

```

#include "sweepparm2.inc"

show_point_object_in_swept(links,segm1,segm2,color1,color2)
int links;
char *segm1,*segm2,*color1,*color2;
{
int i,j,k,jojo,face,pc_curve,joke,increments;
FILE *getin;
if(*segm1 != 'n')
{
HC_Open_Segment(segm1);
HC_Set_Color(color1);

  for (face = 0;face<number_polygons[links];face++)
    for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
     {
       for(i=0,jojo=1;i<Division+1,jojo<Division+1;i++,jojo++)
     {
        HC_Insert_Line(
        point_object[i][x][pc_curve][face],
        point_object[i][y][pc_curve][face],
        point_object[i][z][pc_curve][face],
        point_object[jojo][x][pc_curve][face],
        point_object[jojo][y][pc_curve][face],
        point_object[jojo][z][pc_curve][face]);
     }
    }

  for (face = 0;face<number_polygons[links];face++)
    for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
      {
       if(pc_curve == number_edges[links][face] -1) j=0;
         else  j = pc_curve+1;
        HC_Insert_Line(
        point_object[0][x][pc_curve][face],
        point_object[0][y][pc_curve][face],
        point_object[0][z][pc_curve][face],
        point_object[0][x][j][face],
        point_object[0][y][j][face],
        point_object[0][z][j][face]);
      }


  for (face = 0;face<number_polygons[links];face++)
    for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
      {
       if(pc_curve == number_edges[links][face] -1) j=0;
         else j = pc_curve+1;
        HC_Insert_Line(
        point_object[Division][x][pc_curve][face],
        point_object[Division][y][pc_curve][face],
        point_object[Division][z][pc_curve][face],
        point_object[Division][x][j][face],
        point_object[Division][y][j][face],
        point_object[Division][z][j][face]);
     }
   HC_Close_Segment();
  }

if(*segm2 != 'n')
```

```
 {
   HC_Open_Segment(segm2);
   HC_Set_Color(color2);

  for (face = 0;face<number_polygons[links];face++)
    for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
     {
       for(i=0,jojo=1;i<Division+1,jojo<Division+1;i++,jojo++)
       {
        HC_Insert_Line(
        point_object[i][x][pc_curve][face],
        point_object[i][y][pc_curve][face],
        point_object[i][z][pc_curve][face],
        point_object[jojo][x][pc_curve][face],
        point_object[jojo][y][pc_curve][face],
        point_object[jojo][z][pc_curve][face]);
       }
     }

  for (face = 0;face<number_polygons[links];face++)
    for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
      {
       if(pc_curve == number_edges[links][face] -1) j=0;
         else  j = pc_curve+1;
        HC_Insert_Line(
        point_object[0][x][pc_curve][face],
        point_object[0][y][pc_curve][face],
        point_object[0][z][pc_curve][face],
        point_object[0][x][j][face],
        point_object[0][y][j][face],
        point_object[0][z][j][face]);
     }


  for (face = 0;face<number_polygons[links];face++)
    for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
      {
       if(pc_curve == number_edges[links][face] -1) j=0;
         else j = pc_curve+1;
        HC_Insert_Line(
        point_object[Division][x][pc_curve][face],
        point_object[Division][y][pc_curve][face],
        point_object[Division][z][pc_curve][face],
        point_object[Division][x][j][face],
        point_object[Division][y][j][face],
        point_object[Division][z][j][face]);
      }
    HC_Close_Segment();
 }
}



show_point_object_in_snap(links,segm3,segm4,color3,color4)
int links; char *segm3,*segm4,*color3,*color4;
{
int i,j,k,jojo,face,pc_curve,joke,increments;
FILE *getin;

 if(*segm3 != 'n')
  {
    HC_Open_Segment(segm3);
    HC_Set_Color(color3);
    getin = fopen("infoc","r");
    fscanf(getin,"%d,",&increments);
```

```
    fclose(getin);

  for(joke=0;joke<Division+1;joke+=increments)
   {
   for (face = 0;face<number_polygons[links];face++)
    for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
     {
      if(pc_curve == number_edges[links][face] -1) j=0;
        else j = pc_curve+1;
        HC_Insert_Line(
        point_object[joke][x][pc_curve][face],
        point_object[joke][y][pc_curve][face],
        point_object[joke][z][pc_curve][face],
        point_object[joke][x][j][face],
        point_object[joke][y][j][face],
        point_object[joke][z][j][face]);
     }
   }
  HC_Close_Segment();
 }


if(*segm4 != 'n')
  {
   HC_Open_Segment(segm4);
   HC_Set_Color(color4);
   for(joke=0;joke<Division+1;joke+=increments)
     {
   for (face = 0;face<number_polygons[links];face++)
    for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
      {
       if(pc_curve == number_edges[links][face] -1) j=0;
         else j = pc_curve+1;
        HC_Insert_Line(
        point_object[joke][x][pc_curve][face],
        point_object[joke][y][pc_curve][face],
        point_object[joke][z][pc_curve][face],
        point_object[joke][x][j][face],
        point_object[joke][y][j][face],
        point_object[joke][z][j][face]);
      }
     }
   HC_Close_Segment();
  }
}
```

::::::::::::::
IBM_Formula.cx
::::::::::::::

```
/*
          | 0     -a      -b |
     A =  | a      0      -c |
          | b      c       0 |
*/

#include "sweepparm2.inc"


IBM_First_link(links)
int links;
{
FILE *getin,*gin;
float XX,YY,ZZ;
float at,bt,ct,t,difference,Xt,Yt,Zt,a,b,c,Cx,Cy,Cz,Xo=0,Yo=0,Zo=0;
char tempo[20],*tmp;
int i,las,j,k,jojo,pc_curve,face;
char wierd[3];

getin = fopen("IBM_1c","r");
fscanf(getin,"%f,%f,%f,%f,%f,%f,%f,%f,%f,",&a,&b,&c,&Cx,&Cy,&Cz,&Xo,&Yo,&Zo);
fclose(getin);

        difference = 1.0/Division;
  for (face = 0;face<number_polygons[links];face++)
    for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
     {
        Xo = vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][x];
        Yo = vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][y];
        Zo = vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][z];

       las = 0;
      for(t=0;t<1.01;t=t+difference)
       {
        if(t>1.0) t = 1.0;
        at = a*t;
        point_object[las][x][pc_curve][face] = Xt = cos(at)*Xo+sin(at)*Yo;
        point_object[las][y][pc_curve][face] = Yt = -sin(at)*Xo+cos(at)*Yo;
        point_object[las][z][pc_curve][face] = Zo;
        las++;
        }
    }
}

IBM_Second_link(links)
int links;
{
char *file_name;
FILE *getin,*gin;
float XX,YY,ZZ;
float R,a1t,a2t,bt,ct,t,difference,Xt,Yt,Zt,a1,a2,b,c,Cx,Cy,Cz,Xo=0,Yo=0,Zo=0;
char tempo[20],*tmp;
int i,las,j,k,jojo,pc_curve,face;
char wierd[3];


getin = fopen("IBM_1c","r");
fscanf(getin,"%f,%f,%f,%f,%f,%f,%f,%f,%f,",&a1,&b,&c,&Cx,&Cy,&Cz,&Xo,&Yo,&Zo);
fclose(getin);

getin = fopen("IBM_2c","r");
```

```
fscanf(getin,"%f,%f,%f,%f,%f,%f,%f,%f,%f,",&a2,&b,&c,&Cx,&Cy,&Cz,&Xo,&Yo,&Zo);
fclose(getin);

  gin = fopen("shift_IBM2c","r");
  fscanf(gin,"%f,%f,%f,",&XX,&YY,&ZZ);
  fclose(gin);

        difference = 1.0/Division;
  las = 0;
  for(t=0;t<1.01;t=t+difference)
    {
      if(t>1.0) t = 1.0;
      a2t = a2*t; a1t = a1*t; R = 0.0;
      trajectory_point[links][las][x] = Xt = (YY)*sin(a1t)+(XX)*cos(a1t);
      trajectory_point[links][las][y] = Yt = -(XX)*sin(a1t)+(YY)*cos(a1t);
      trajectory_point[links][las][z] = 0.0;
      trajectory_point[links][las][roll] = -a2t;
      trajectory_point[links][las][pitch] = 0.0;
      trajectory_point[links][las][yaw] = 0.0;

        las++;
    }
     transform_vertices(links);                          /* trans1.c */
}
```

::::::::::::::
IBMlinks.cx
::::::::::::::

```
#include "sweepparm2.inc"
#include "transform.c"


short IBM_links()
{
  float length, radius, theta, theta1, tan, x_angle,y_angle,z_angle;
  int   i, j, k, face, polyhedron, item;
  char exit,string,mouse[10];
  double atof();
  float scale = 2.0;


  IBM_base();                               /* "IBM_base_link.c" */
  IBM_Arms(1,"IBM_upper_data.c");           /* "IBM_ARMS.c" */
  IBM_Arms(2,"IBM_fore_data.c");            /* "IBM_ARMS.c" */
  IBM_hand();                               /* "IBM_end_effector.c" */

      { int r,s,t;
        float dummy_points[6][100][3];
        for(r=0;r<6;r++)
         for(s=0;s<100;s++)
          for(t=0;t<3;t++)
           dummy_points[r][s][t]=vertex[r][s][t];
             transfer(-2.4*scale,0.0,0.0,0.0,0.0,0.0,2,dummy_points);
             transfer(-4.0*scale,0.0,-2.4*scale,0.0,0.0,-pi,3,dummy_points);

          insert_dummy_lines(0,3,dummy_points,"?picture/geometry/links/A/dummy");
          insert_dummy_lines(0,3,dummy_points,"?picture/geometry/links/B/dummy");
       }
 return polyhedron;
 }




 /* The same as the subroutine transform() */

 transfer(XX,YY,ZZ,rollZ,pitchY,yawX,link,dy_points)
 float XX,YY,ZZ,rollZ,pitchY,yawX;
 int link;
 float dy_points[6][100][3];
   {
     float t[3][3];
     float pseudoX,pseudoY,pseudoZ;
     int i,j,pc_curve,face;
     t[0][0] = cos(rollZ)*cos(pitchY);
     t[1][0] = sin(rollZ)*cos(pitchY);
     t[2][0] = -sin(pitchY);
     t[0][1] = cos(rollZ)*sin(pitchY)*sin(yawX)-sin(rollZ)*cos(yawX);
     t[1][1] = sin(rollZ)*sin(pitchY)*sin(yawX)+cos(rollZ)*cos(yawX);
     t[2][1] = cos(pitchY)*sin(yawX);
     t[0][2] = cos(rollZ)*sin(pitchY)*cos(yawX)+sin(rollZ)*sin(yawX);
     t[1][2] = sin(rollZ)*sin(pitchY)*cos(yawX)-cos(rollZ)*sin(yawX);
     t[2][2] = cos(pitchY)*cos(yawX);

     for(i=0;i<=2;i++)
       for(j=0;j<=2;j++)
         if(fabs(t[i][j])<0.00001) t[i][j]=0.0;

         for (pc_curve=0;pc_curve<no_of_vertix[link];pc_curve++)
```

```
          {
            pseudoX =  dy_points[link][pc_curve][x] ;
            pseudoY =  dy_points[link][pc_curve][y] ;
            pseudoZ =  dy_points[link][pc_curve][z] ;

            dy_points[link][pc_curve][x]=t[0][0]*pseudoX+t[0][1]*pseudoY+t[0][2]*pseudoZ+XX;
            dy_points[link][pc_curve][y]=t[1][0]*pseudoX+t[1][1]*pseudoY+t[1][2]*pseudoZ+YY;
            dy_points[link][pc_curve][z]=t[2][0]*pseudoX+t[2][1]*pseudoY+t[2][2]*pseudoZ+ZZ;
          }
  }





insert_dummy_lines(header,limiter,dum_points,seg_name)
   int header,limiter;
   float dum_points[6][100][3];
   char *seg_name;
    {
      int ucla,face,i,j;
      HC_Open_Segment(seg_name); /*"?picture/geometry/links/dummy");*/
      HC_Set_Color("line=red");

       for(ucla= header;ucla<=limiter;ucla++)
         {
           printf(" \n\n link no. is %d",ucla);

            for(face=0;face<number_polygons[ucla];face++)
                for(i=0;i<number_edges[ucla][face];i++)
                  {
                     if(i==number_edges[ucla][face]-1)
                      j = 0;
                     else
                      j = i + 1;
                HC_Insert_Line(
                 dum_points[ucla][edge[ucla][polygon[ucla][face][i]][cycle[ucla][face][i]]][x],
                 dum_points[ucla][edge[ucla][polygon[ucla][face][i]][cycle[ucla][face][i]]][y],
                 dum_points[ucla][edge[ucla][polygon[ucla][face][i]][cycle[ucla][face][i]]][z],
                 dum_points[ucla][edge[ucla][polygon[ucla][face][j]][cycle[ucla][face][j]]][x],
                 dum_points[ucla][edge[ucla][polygon[ucla][face][j]][cycle[ucla][face][j]]][y],
                 dum_points[ucla][edge[ucla][polygon[ucla][face][j]][cycle[ucla][face][j]]][z]);

  edge_length[ucla][face][i] = sqrt(
        (dum_points[ucla][edge[ucla][polygon[ucla][face][i]][1]][x]
        - dum_points[ucla][edge[ucla][polygon[ucla][face][i]][0]][x])*
        (dum_points[ucla][edge[ucla][polygon[ucla][face][i]][1]][x]
        - dum_points[ucla][edge[ucla][polygon[ucla][face][i]][0]][x])
        + (dum_points[ucla][edge[ucla][polygon[ucla][face][i]][1]][y]
        - dum_points[ucla][edge[ucla][polygon[ucla][face][i]][0]][y])*
        (dum_points[ucla][edge[ucla][polygon[ucla][face][i]][1]][y]
        - dum_points[ucla][edge[ucla][polygon[ucla][face][i]][0]][y])
        + (dum_points[ucla][edge[ucla][polygon[ucla][face][i]][1]][z]
        - dum_points[ucla][edge[ucla][polygon[ucla][face][i]][0]][z])*
        ( dum_points[ucla][edge[ucla][polygon[ucla][face][i]][1]][z]
        - dum_points[ucla][edge[ucla][polygon[ucla][face][i]][0]][z]));
                 }
        }
     HC_Close_Segment();
    }
```

::::::::::::::
Message.cx
::::::::::::::

```
message_window(upper,lower,third)
char *upper,*lower,*third;
{
      HC_Open_Segment("?picture/message/mes1");
      HC_Flush_Segment(".");
      HC_Insert_Text(0.0,0.5,0.0,upper);
      HC_Insert_Text(0.0,-0.5,0.0,lower);
      HC_Close_Segment();
      HC_Open_Segment("?picture/message/mes2/ma");
      HC_Flush_Segment(".");
      HC_Insert_Text(0.0,0.0,0.0,third);
      HC_Close_Segment();
}

char *answer_window()
{
int answers;
  char gogo[20];
      HC_Open_Segment("?picture/message/mes2/mb");
      HC_Get_String("junk", gogo);
      HC_Flush_Segment(".");
      HC_Close_Segment();
      HC_Flush_Segment("?picture/message");
    return (&gogo[0]);

}


char *Hardcopy_window()
{
HC_Open_Segment("?picture/copy");
        HC_QSet_Visibility("?hardcopy","on");
        HC_Update_Display();
        HC_QSet_Visibility("?hardcopy","off");
        HC_Flush_Segment("?picture/copy");
        HC_Insert_Text(0.0,0.0,0.0,"HARDCOPY");
HC_Close_Segment();
}

char *Quit_window()
{
char mouse[10];
HC_Open_Segment("?picture/quit");
HC_Insert_Text(0.0,0.0,0.0,"QUIT");
HC_Get_Selection(mouse);
          if (strcmp(mouse,"quit")==0) exit();
HC_Flush_Segment(".");
HC_Close_Segment();
}
```

::::::::::::::
New_stuff.cx
::::::::::::::

```
#include "sweepparm2.inc"
#include "Message.c"

Formulee(links)
int links;
{
int polygon_increment, i, j;
short sweep_defined,sweep_display,vertices_distance_error;
short IBM_links(), sweep_polyhedron();
char gogo[5],teeem;


message_window("What kind of curve you like to get/","R_P_Y /AUTO/1/2/3/f/s","r/a/1/2/3/f/s");
 teeem = *answer_window();

  printf("%c\n",teeem);
  switch(teeem)
      {
        case 'r' :  initial_formula(links,"rpudatac");  break;          /* auto_formula.c */
        case 'a' :  Fast_transform_vertices(links); break;              /* fast_trans1.c */
        case '1' :  links = 1; number_segments[links] = 1;
                                PUMA_first_link(links); break;          /* PUMA_Formula.c */
        case '2' :  links = 2; number_segments[links] = 1;
                                PUMA_second_link(links); break;         /* PUMA_Formula.c */
        case '3' :  links = 3; number_segments[links] = 1;
                                PUMA_third_link(links); break;          /* PUMA_Formula.c */

        case 'f' :  links = 1; number_segments[links] = 1;
                                IBM_First_link(links); break;           /* IBM_new_form.c */
        case 's' :  links = 2; number_segments[links] = 1;
                                IBM_Second_link(links); break;          /* IBM_new_form.c */
      }

   message_window("Sweep ?","","");
  if(*answer_window() == 'y')
   show_point_object_in_swept(links,"?picture/geometry/links/A","no","line=black","no");

  message_window("animation ?","","");
  if(*answer_window() == 'y')
    show_point_object_in_snap(links,"?picture/geometry/links/B","no","line = red","no");

  message_window("Overlap?","","y/n");
  if(*answer_window() == 'y')
   {
    show_point_object_in_swept(links,"?picture/geometry/links/C", "no", "line = blue","no");
     HC_QSet_Line_Weight("?picture/geometry/links/C",3.0);
   }

  message_window("shaded it ?","","");
   if(*answer_window() == 'y')
     {
      /* Before appling shading technique, first user can rotate the swept volume to
         a better view, and then shade it. Conceptially, the viewer never change location,
         but swept volume do. */

        transform_object(NQ_image,links); /*transobjt1.c*/
        shaded_display_two(NQ_image,links);
     }
 }
```

::::::::::::::
PUMA_Formula.cx
::::::::::::::

```
/*
          | 0     -a      -b |
      A = | a      0      -c |
          | b      c       0 |
*/

#include "sweepparm2.inc"

PUMA_first_link(links)
int links;
{
FILE *getin,*gin;
float XX,YY,ZZ;
float at,bt,ct,t,difference,Xt,Yt,Zt,a,b,c,Cx,Cy,Cz,Xo=0,Yo=0,Zo=0;
char tempo[20],*tmp;
int i,las,j,k,jojo,pc_curve,face;
char wierd[3];

printf("this is link %d\n",links);
getin = fopen("PUMA_1c","r");
fscanf(getin,"%f,%f,%f,%f,%f,%f,%f,%f,%f,",&a,&b,&c,&Cx,&Cy,&Cz,&Xo,&Yo,&Zo);
fclose(getin);

        difference = 1.0/Division;
for (face = 0;face<number_polygons[links];face++)
  for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
   {
    Xo = vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][x];
    Yo = vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][y];
    Zo = vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][z];

      las = 0;
      for(t=0;t<1.01;t=t+difference)
        {
         if(t>1.0) t = 1.0;
         at = a*t;
         point_object[las][x][pc_curve][face] = Xt = cos(-at)*Xo+sin(-at)*Yo;
         point_object[las][y][pc_curve][face] = Yt = -sin(-at)*Xo+cos(-at)*Yo;
         point_object[las][z][pc_curve][face] = Zo;

        las++;
        }
    }
}



/* This part of the porgram is implemented but the sweep differential has not been found yet. */

PUMA_second_link(links)
int links;
{
char *file_name;
FILE *getin,*gin;
float XX,YY,ZZ;
float R,a1t,a2t,bt,ct,t,difference,Xt,Yt,Zt,a1,a2,b,c,Cx,Cy,Cz,Xo=0,Yo=0,Zo=0;
char tempo[20],*tmp;
int i,las,j,k,jojo,pc_curve,face;
char wierd[3];
```

```
getin = fopen("PUMA_1c","r");
fscanf(getin,"%f,%f,%f,%f,%f,%f,%f,%f,%f,",&a1,&b,&c,&Cx,&Cy,&Cz,&Xo,&Yo,&Zo);
fclose(getin);

getin = fopen("PUMA_2c","r");
fscanf(getin,"%f,%f,%f,%f,%f,%f,%f,%f,%f,",&a2,&b,&c,&Cx,&Cy,&Cz,&Xo,&Yo,&Zo);
fclose(getin);

  gin = fopen("shift_PUMA2c","r");
  fscanf(gin,"%f,%f,%f,",&XX,&YY,&ZZ);
  fclose(gin);
  R = sqrt(XX*XX+YY*YY*ZZ*ZZ);

        difference = 1.0/Division;
las = 0;
for(t=0;t<1.01;t=t+difference)
    {
     if(t>1.0) t = 1.0;
     a2t = a2*t; a1t = a1*t;
      trajectory_point[links][las][x] = Xt = (YY)*sin(a1t)+(XX)*cos(a1t);
      trajectory_point[links][las][y] =Yt = -(XX)*sin(a1t)+(YY)*cos(a1t);
      trajectory_point[links][las][z] = 0.0;

/*
        | 0              o               -a2tCos(a1t) |
    A = | o              0               -a2tSin(a1t) |
        | 2tCos(a1t)     2tSin(a1t)       0           |
*/
      trajectory_point[links][las][roll] = a1t;
      trajectory_point[links][las][pitch] = a2t;
      trajectory_point[links][las][yaw] = 0.0;

        las++;
    }
     transform_vertices(links);                          /* trans1.c */
}




/* This part of the porgram is implemented but the sweep differential has not been found yet. */

PUMA_third_link(links)
int links;
{
char *file_name;
FILE *getin,*gin;
float XX,YY,ZZ;
float R,a1t,a2t,a3t,bt,ct,t,difference,Xt,Yt,Zt,a1,a2,a3,b,c,Cx,Cy,Cz,Xo=0,Yo=0,Zo=0;
float b1,b2,c1,c2,b3,c3;
char tempo[20],*tmp;
int i,las,j,k,jojo,pc_curve,face;
char wierd[3];


getin = fopen("PUMA_1c","r");
fscanf(getin,"%f,%f,%f,%f,%f,%f,%f,%f,%f,",&a1,&b1,&c1,&Cx,&Cy,&Cz,&Xo,&Yo,&Zo);
fclose(getin);

getin = fopen("PUMA_2c","r");
fscanf(getin,"%f,%f,%f,%f,%f,%f,%f,%f,%f,",&a2,&b2,&c2,&Cx,&Cy,&Cz,&Xo,&Yo,&Zo);
fclose(getin);

getin = fopen("PUMA_3c","r");
fscanf(getin,"%f,%f,%f,%f,%f,%f,%f,%f,%f,",&a3,&b3,&c3,&Cx,&Cy,&Cz,&Xo,&Yo,&Zo);
```

```
fclose(getin);

  gin = fopen("shift_PUMA3c","r");
fscanf(gin,"%f,%f,%f,",&XX,&YY,&ZZ);
fclose(gin);

        difference = 1.0/Division;
las = 0;
for(t=0;t<1.01;t=t+difference)
   {
    if(t>1.0) t = 1.0;
    a3t = a3*t; a2t = a2*t; a1t = a1*t;
    R = 4.32*cos(a2t);
     trajectory_point[links][las][x] = Xt = R*cos(a1t);
     trajectory_point[links][las][y] = Yt = R*sin(a1t);
     trajectory_point[links][las][z] =  (4.32)*sin(-a2t);

      trajectory_point[links][las][roll] = a1t;
      trajectory_point[links][las][pitch] = a3t;
      trajectory_point[links][las][yaw] = 0.0;

        las++;
   }
     transform_vertices(links);                          /* trans1.c */
}
```

```
::::::::::::::
PUMAlinks.cx
::::::::::::::
#include "sweepparm2.inc"

short PUMA_links()
{
  float length, radius, theta, theta1, tan, x_angle,y_angle,z_angle;
  int   i, j, k, face, polyhedron, item;
  char exit,string,mouse[10];
  double atof();

   /* The following subroutine define the solid model of PUMA robot. */
          PUMA_base();          /* "PUMA_base_link.c" */
          PUMA_motor();         /* "PUMA_motor_link.c"*/
          PUMA_forearm();       /* "PUMA_fore_arm.c" */
          PUMA_upperarm();      /* "PUMA_upper_arm.c" */
          PUMA_hand();          /* "PUMA_end_effector.c" */

   /* This subroutine shows the PUMA robot initial condition, in red line. */
        dummy_PUMA();

   /* The subroutine transfrom" is to transform the solid vertex to proper initial
      position. While defining the PUMA links, it is easier to define it corelated
      to the world coordinate then transfer the object vertex to proper location.
       The format of this subroutine is transform(x,y,z,roll,pitch,yaw,link_number); */

    transform(0.0,0.0,0.0,0.0,0.0,-1.57080,1);  /* link 1 --- motor */
    transform(0.0,-0.75,0.0,0.0,0.0,0.0,1);

    transform(0.0,0.0,0.0,0.0,0.0,1.57080,2);   /* link 2 --- upper-arm */

    transform(0.0,0.0,0.0,0.0,0.0,1.57080,3);   /* link 3 --- fore-arm */


    transform(0.19,0.0,0.0,0.0,0.0,0.0,4);      /* link 4 --- end-effector */
    transform(0.0,-1.25,0.0,0.0,0.0,0.0,4);

 return polyhedron;
 }



   dummy_PUMA()
    {
     int r,s,t;
     float dummy_points[6][100][3];
         for(r=0;r<6;r++)
          for(s=0;s<100;s++)
           for(t=0;t<3;t++)
            dummy_points[r][s][t]=vertex[r][s][t];
            transfer(0.0,0.0,0.2,0.0,0.0,0.0,0,dummy_points);
            transfer(0.0,0.0,0.0,0.0,0.0,-1.57080,1,dummy_points);
            transfer(0.0,-0.75,0.0,0.0,0.0,0.0,1,dummy_points);
            transfer(0.0,0.0,0.0,0.0,0.0,1.57080,2,dummy_points);
            transfer(0.0,0.0,0.0,0.0,0.0,1.57080,3,dummy_points);
            transfer(4.32,0.0,0.0,0.0,0.0,0.0,3,dummy_points);
            transfer(4.51,0.0,0.0,0.0,0.0,0.0,4,dummy_points);
            transfer(0.0,-1.25,0.0,0.0,0.0,0.0,4,dummy_points);
            insert_dummy_lines(0,4,dummy_points,"?picture/geometry/links/dummy");
    }
```

```
:::::::::::::
Rotat_Segment.cx
:::::::::::::

static float  x_angle,y_angle,z_angle;
rotate_segments(seg_nam)
char seg_nam[25];
{
  float length, radius, theta, theta1, tan;
  int   i, j, k, face, polyhedron, item;
  char exit,string[10],mouse[10];
  FILE *getin;
  char takein[90];
  double atof();

     for(;;)
       {

HC_Open_Segment("?picture/menus");
   HC_Flush_Segment(".");


   HC_Open_Segment("?picture/menus/reset");
   HC_Insert_Text(0.0,0.0,0.0,"RESET");
   HC_Close_Segment();

   HC_Open_Segment("?picture/menus/rotate_x");
   HC_Insert_Text(0.0,0.0,0.0,"Rotate X");
   HC_Close_Segment();

   HC_Open_Segment("?picture/menus/rotate_y");
   HC_Insert_Text(0.0,0.0,0.0,"Rotate Y" );
   HC_Close_Segment();

   HC_Open_Segment("?picture/menus/rotate_z");
   HC_Insert_Text(0.0,0.0,0.0,"Rotate Z");
   HC_Close_Segment();


HC_Open_Segment("?picture/menus/m5");
   HC_Insert_Text(0.0,0.0,0.0,"ROTATE");
   HC_Close_Segment();

   HC_Open_Segment("?picture/menus/m6");
   HC_Insert_Text(0.0,0.0,0.0,"1st link");
   HC_Close_Segment();

   HC_Open_Segment("?picture/menus/m7");
   HC_Insert_Text(0.0,0.0,0.0,"2nd link");
   HC_Close_Segment();

   HC_Open_Segment("?picture/menus/m8");
   HC_Insert_Text(0.0,0.0,0.0,"3rd link");
   HC_Close_Segment();

   HC_Open_Segment("?picture/menus/m9");
   HC_Insert_Text(0.0,0.0,0.0,"4th link");
   HC_Close_Segment();

   HC_Open_Segment("?picture/menus/m10");
   HC_Insert_Text(0.0,0.0,0.0,"Animation");
   HC_Close_Segment();
HC_Close_Segment();

```

```
HC_Get_Selection(mouse);
      if (strcmp(mouse,"m6")==0) {
                   /*if(buffer =='i')IBM_First_link(link_no_buffer = 1);*/
                                  break;}
      else if(strcmp(mouse,"m7")==0){
                   /*if(buffer =='i')IBM_Second_link(link_no_buffer=2);*/
                                   break;}
      else if (strcmp(mouse,"m8")==0) {Formulee(link_no_buffer = 3);break;}
      else if (strcmp(mouse,"m9")==0) {Formulee(link_no_buffer = 4);break;}
      else if (!strcmp(mouse,"m10") && Formulee(link_no_buffer)) break;
      else if (strcmp(mouse,"okokokokok")==0) { HC_UnSet_Modelling_Matrix();
                                        x_angle=0.0;
                                        y_angle=0.0;
                                        z_angle=0.0;
                                          }
      else if (strcmp(mouse,"m5")==0) HC_QRotate_Object(seg_nam,x_angle,y_angle,z_angle);
      else if (strcmp(mouse,"reset")==0)   /** Quit_window()*/
             {getin = fopen("autodatac","r");
              fscanf(getin,"%s\n",takein);
              fclose(getin);
              message_window(takein,"  a,  b,  c,  Cx,  Cy,  Cz","");}
      else if (strcmp(mouse,"sweep")==0) Formulee(link_no_buffer);
      else if (strcmp(mouse,"rotate_x")==0 || strcmp(mouse,"rotate_y")==0
                                          || strcmp(mouse,"rotate_z")==0)
        {
 char temmp;
 message_window("ENTER HOW MANY DEGREES","YOU WANT TO ROTATE","");
     HC_Open_Segment("?picture/message/mes2/mb");
     HC_Flush_Segment(".");
          HC_Get_String("angle",string);
     HC_Close_Segment();
     if(temmp == 'u') seg_nam = "?picture/geometry/links/upper";
        else if(temmp == 'l') seg_nam = "?picture/geometry/links/lower";
               else seg_nam = "?picture/geometry/links";

     }

          if (strcmp(mouse,"rotate_x")==0)
            {
              x_angle=atof(string);
              y_angle=0.0;
              z_angle=0.0;
            }
          else if (strcmp(mouse,"rotate_y")==0)
            {
              y_angle=atof(string);
              x_angle=0.0;
              z_angle=0.0;
             }
          else if (strcmp(mouse,"rotate_z")==0)
            {
              z_angle=atof(string);
              x_angle=0.0;
              y_angle=0.0;
            }
        }
   HC_Update_Display();
 }
```

```
::::::::::::::
auto_formula.cx
::::::::::::::
#include "sweepparm2.inc"

initial_formula(links,file_name)
int links;
char *file_name;
{
FILE *getin,*gin;
float XX,YY,ZZ;
float at,bt,ct,t,difference,Xt,Yt,Zt,a,b,c,Cx,Cy,Cz,Xo=0,Yo=0,Zo=0;
char tempo[20],*tmp;
int i,las,j,k,jojo,pc_curve,face;
char wierd[3];

getin = fopen(file_name,"r");
fscanf(getin,"%f,%f,%f,%f,%f,%f,%f,%f,%f,",&a,&b,&c,&Cx,&Cy,&Cz,&Xo,&Yo,&Zo);
fclose(getin);

/* Check the sweeping solid object belongs to PUMA or IBM robot. */
switch(buffer)
  {
    /* If it is PUMA link 2 then difine the coordinate at (-2.4,0,0).
       If it is PUMA link 3 then difine the coordinate at (-4,0,-2.4). */

    case 'p': if(links==2) {Xo=-2.4;Yo=0.0;Zo=0.0;}
                else if(links==3){Xo=-4.0;Yo=0.0;Zo=-2.4;}
                break;

    /* If it is PUMA link 1 then difine the coordinate at (0,-.75,0).
       If it is PUMA link 2 then difine the coordinate at (0,-1.25,0).
       If it is PUMA link 3 then difine the coordinate at (0.19,-1.25,0). */

    case 'i':if(links==1){Xo=-0.0;Yo=-0.75;Zo=0.0;}
                else if(links==2);
                        else if(links==3);
                                else if(links==4){Xo=0.19;Yo=-1.25;}
                break;
  }


 /* This is to redifine the autonomous sweep initial location, The new location should
    be put in the file "shiftdatac" before answer the following question. */

  message_window("Would you like to redifine the initial point?","","y/n");
  if(*answer_window() == 'y')
   {
     gin = fopen("shiftdatac","r");
     fscanf(gin,"%f,%f,%f,",&XX,&YY,&ZZ);
     fclose(gin);
     shiftt(XX,YY,ZZ,links);
   }

  difference = 1.0/Division;
 for (face = 0;face<number_polygons[links];face++)
 for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
 {
 Xo = vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][x];
 Yo = vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][y];
 Zo = vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][z];

        las = 0;
        for(t=0;t<1.01;t=t+difference)
```

```
        {
          if(t>1.0) t = 1.0;
          at = a*t;
          point_object[las][x][pc_curve][face] = Xt
                      = 1/a*(sin(at)*Cx + (1-cos(at))*Cy)+cos(at)*Xo+sin(at)*Yo;
          point_object[las][y][pc_curve][face] = Yt
                      = 1/a*(-(1-cos(at))*Cx + sin(at)*Cy)-sin(at)*Xo+cos(at)*Yo;
          point_object[las][z][pc_curve][face] = Zo;
           las++;
        }
      }
}
```

```
::::::::::::::
e.cx
::::::::::::::

/* This is the subroutine, which set up the screen manu,
    by using HOOPS buildin functions. */

#include <math.h>
#include <stdio.h>
#include <string.h>
#define streq(x,y) strcmp(x,y)==0



environment()
{
 char  name[10],*message_up="\0",*message_down="\0",*menu1="\0",
        *menu2="\0",*menu3="\0",*menu4="\0",*menu5="\0";
 char  *menu6="\0",*menu7="\0",*menu8="\0",*menu9="\0",*menu10="\0";

float x_cord = 10.,y_cord = 10., z_cord = 10.;
HC_Open_Segment("?picture");
HC_Set_Camera_Position(0.0,0.0,5.0);
HC_Open_Segment("?picture/message");
HC_Set_Window(-1.0,1.0,-1.0,-0.8);
HC_Set_Text_Size(0.5);
HC_Set_Color("window=black, window contrast = white, text=white");
HC_Set_Window_Frame("on");

   HC_Open_Segment("?picture/message/mes1");
   HC_Set_Window(-1.0,0.0,-1.0,1.0);
   HC_Set_Window_Frame("on");
   HC_Insert_Text(0.0,0.5,0.0,message_up);
   HC_Insert_Text(0.0,-0.5,0.0,message_down);
   HC_Close_Segment();

   HC_Open_Segment("?picture/message/mes2");
   HC_Set_Window(0.0,1.0,-1.0,1.0);
   HC_Set_Window_Frame("on");

   HC_Open_Segment("?picture/message/mes2/ma");
   HC_Set_Window(-1.0,1.0,0.0,1.0);
   HC_Set_Window_Frame("on");
   HC_Close_Segment();

   HC_Open_Segment("?picture/message/mes2/mb");
   HC_Set_Window(-1.0,1.0,-1.0,0.0);
   HC_Set_Window_Frame("on");
   HC_Close_Segment();

   HC_Close_Segment();

   HC_Close_Segment();

HC_Open_Segment("?picture/Sweep");
HC_Set_Window(-1.0,-0.8,-0.8,-0.55);
HC_Set_Text_Size(0.3);
HC_Set_Color("window=blue,text=yellow");
HC_Set_Window_Frame("on");
HC_Insert_Text(0.0,0.0,0.0,"SWEEP");
HC_Close_Segment();

HC_Open_Segment("?picture/quit");
HC_Set_Window(-1.0,-0.8,0.8,1.0);
```

```
HC_Set_Text_Size(0.3);
HC_Set_Color("window=red,text=yellow");
HC_Set_Window_Frame("on");
HC_Insert_Text(0.0,0.0,0.0,"QUIT");
HC_Close_Segment();


HC_Open_Segment("?picture/menus");
HC_Set_Window(-1.0,-0.8,-0.55,0.8);
HC_Set_Text_Size(0.3);
HC_Set_Color("window=yellow,text=blue");
HC_Set_Window_Frame("on");


    HC_Open_Segment("?picture/menus/reset");
    HC_Set_Window(-1.0,1.0,0.8,1.0);
    HC_Set_Window_Frame("on");
    HC_Insert_Text(0.0,0.0,0.0,menu1);
    HC_Close_Segment();

    HC_Open_Segment("?picture/menus/rotate_x");
    HC_Set_Window(-1.0,1.0,0.6,0.8);
    HC_Insert_Text(0.0,0.0,0.0,menu2);
    HC_Set_Window_Frame("on");
    HC_Close_Segment();

    HC_Open_Segment("?picture/menus/rotate_y");
    HC_Set_Window(-1.0,1.0,0.4,0.6);
    HC_Set_Window_Frame("on");
    HC_Insert_Text(0.0,0.0,0.0,menu3);
    HC_Close_Segment();

    HC_Open_Segment("?picture/menus/rotate_z");
    HC_Set_Window(-1.0,1.0,0.2,0.4);
    HC_Set_Window_Frame("on");
    HC_Insert_Text(0.0,0.0,0.0,menu4);
    HC_Close_Segment();

    HC_Open_Segment("?picture/menus/m5");
    HC_Set_Window(-1.0,1.0,0.,0.2);
    HC_Insert_Text(0.0,0.0,0.0,menu5);
    HC_Set_Window_Frame("on");
    HC_Close_Segment();

    HC_Open_Segment("?picture/menus/m6");
    HC_Set_Window(-1.0,1.0,-0.2,0.0);
    HC_Set_Window_Frame("on");
    HC_Insert_Text(0.0,0.0,0.0,menu6);
    HC_Close_Segment();

    HC_Open_Segment("?picture/menus/m7");
    HC_Set_Window(-1.0,1.0,-0.4,-0.2);
    HC_Insert_Text(0.0,0.0,0.0,menu7);
    HC_Set_Window_Frame("on");
    HC_Close_Segment();

    HC_Open_Segment("?picture/menus/m8");
    HC_Set_Window(-1.0,1.0,-0.6,-0.4);
    HC_Set_Window_Frame("on");
    HC_Insert_Text(0.0,0.0,0.0,menu8);
    HC_Close_Segment();

    HC_Open_Segment("?picture/menus/m9");
    HC_Set_Window(-1.0,1.0,-0.8,-0.6);
    HC_Set_Window_Frame("on");
```

```
    HC_Insert_Text(0.0,0.0,0.0,menu9);
    HC_Close_Segment();

    HC_Open_Segment("?picture/menus/m10");
    HC_Set_Window(-1.0,1.0,-1.0,-0.8);
    HC_Insert_Text(0.0,0.0,0.0,menu10);
    HC_Set_Window_Frame("on");
    HC_Close_Segment();


HC_Close_Segment();


HC_Open_Segment("?picture");
HC_Open_Segment("?picture/geometry");
HC_Set_Window(-0.8,1.0,-0.8,1.0);
HC_Set_Color("window=green yellow,line=red");
HC_Set_Window_Frame("on");


HC_Open_Segment("?picture/geometry/links");
HC_Set_Window(-1.,1.0,-1.,1.0);
              HC_Open_Segment("AXES");
                HC_Set_Text_Size(0.3);
                HC_Set_Line_Weight(1.0);
                HC_Set_Color("LINES = red,text = gold");
                HC_Insert_Line(0.0,0.0,0.0,3.0,0.0,0.0);
                HC_Insert_Line(0.0,0.0,0.0,0.0,3.0,0.0);
                HC_Insert_Line(0.0,0.0,0.0,0.0,0.0,3.0);
                HC_Insert_Text(3.10,0.0,0.0,"X");
                HC_Insert_Text(0.0,3.10,0.0,"Y");
                HC_Insert_Text(0.0,0.0,3.10,"Z");
                HC_Insert_Marker(0.0,0.0,0.0);
               HC_Close_Segment();
HC_Close_Segment();
HC_Close_Segment();
HC_Close_Segment();

HC_Open_Segment("?picture/geometry/links/A");
HC_Close_Segment();
HC_Open_Segment("?picture/geometry/links/B");
HC_Close_Segment();
HC_Open_Segment("?picture/geometry/links/C");
HC_Close_Segment();
}
```

```
:::::::::::::
fast_trans1.cx
:::::::::::::
/* This fast_transform_vertices is using the property of the autonomous sweep vector
   field, that the fild lines should parallel to each orher. For the reason, one can
   sweep the solid vertex directly, without sweeping the "body_attatched_coordinate_frame".
   In this way, one can save the computer computation time of transform the solid vertex
   with respect to the "body_attatched_ coordinate_frame. */

#include "sweepparm2.inc"
#include "auto_head.inc"


Fast_transform_vertices(links)
int links;
{

int i,j,k,pc_curve,face,las;
float XX,YY,ZZ;
float at,bt,ct,t,difference,Xt,Yt,Zt,a,b,c,Cx,Cy,Cz,Xo=0,Yo=0,Zo=0;
FILE *getin,*gin;
getin = fopen("autodatac","r");
fscanf(getin,"%f,%f,%f,%f,%f,%f,",&a,&b,&c,&Cx,&Cy,&Cz);
fclose(getin);

Pre_Processing(a,b,c);
message_window("Would you like to redifine the initial point?","","y/n");
if(*answer_window() == 'y')
{
  gin = fopen("shiftdatac","r");
  fscanf(gin,"%f,%f,%f,",&XX,&YY,&ZZ);
  fclose(gin);
  shiftt(XX,YY,ZZ,links);
}

        difference = 1.0/Division;

 for (face = 0;face<number_polygons[links];face++)
  for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
  {
  Xo = vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][x];
  Yo = vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][y];
  Zo = vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][z];

 las = 0;
 for(t=0;t<1.01;t=t+difference)
        {
        if(t>1.0) t = 1.0;
        at = a*t; bt = b*t; ct = c*t; Kt = K*t;
        COSS = cos(Kt); SINN = sin(Kt);
        ISIN = (1-cos(Kt))/K; ICOS = sin(Kt)/K;

      point_object[las][x][pc_curve][face] = autox(a,b,c,Cx,Cy,Cz,Xo,Yo,Zo,t);
      point_object[las][y][pc_curve][face] = autoy(a,b,c,Cx,Cy,Cz,Xo,Yo,Zo,t);
      point_object[las][z][pc_curve][face] = autoz(a,b,c,Cx,Cy,Cz,Xo,Yo,Zo,t);

        las++;

        }
      }
 }
```

```
float autox(a,b,c,Cx,Cy,Cz,Xo,Yo,Zo,t)
  float a,b,c,Cx,Cy,Cz,Xo,Yo,Zo,t;
{
float Res,Rst,Rso;
    Rso = (CC+(AA+BB)*COSS)*Xo+(a*c*(COSS-1)+b*K*SINN)*Yo+(-b*c*(COSS-1)+K*a*SINN)*Zo;
    Rst = (CC*t+(AA+BB)*ICOS)*Cx+(a*c*(ICOS-t)+b*K*ISIN)*Cy+(-b*c*(ICOS-t)+K*a*ISIN)*Cz;
    Res = Rso+Rst;
 return Res/KK;
}

float autoy(a,b,c,Cx,Cy,Cz,Xo,Yo,Zo,t)
  float a,b,c,Cx,Cy,Cz,Xo,Yo,Zo,t;
{
 float Res,Rst,Rso;

    Rso = (a*c*(COSS-1)-b*K*SINN)*Xo+(AA+(CC+BB)*COSS)*Yo+(a*b*(COSS-1)+K*c*SINN)*Zo;
    Rst = (a*c*(ICOS-t)-b*K*ISIN)*Cx+(AA*t+(CC+BB)*ICOS)*Cy+(a*b*(ICOS-t)+K*c*ISIN)*Cz;
    Res = Rso+Rst;

 return Res/KK;
}

float autoz(a,b,c,Cx,Cy,Cz,Xo,Yo,Zo,t)
  float a,b,c,Cx,Cy,Cz,Xo,Yo,Zo,t;
{
 float Res,Rst,Rso;

    Rso = (-b*c*(COSS-1)-K*a*SINN)*Xo+(a*b*(COSS-1)-K*c*SINN)*Yo+(BB+JJ*COSS)*Zo;
    Rst = (-b*c*(ICOS-t)-K*a*ISIN)*Cx+(a*b*(ICOS-t)-K*c*ISIN)*Cy+(BB*t+JJ*ICOS)*Cz;
    Res = Rso+Rst;
 return Res/KK;
}



Pre_Processing(a,b,c)
float a,b,c;
{
AA = a*a;
BB = b*b;
CC = c*c;
JJ = a*a+c*c;
J = sqrt( JJ );
KK = a*a+b*b+c*c;
K = sqrt( KK );
}


shiftt(XX,YY,ZZ,links)
float XX,YY,ZZ;
int links;
{
int i,j;
for(i=0;i<no_of_vertix[links]; i++)
 {
  vertex[links][i][x] = vertex[links][i][x] + XX;
  vertex[links][i][y] = vertex[links][i][y] + YY;
  vertex[links][i][z] = vertex[links][i][z] + ZZ;
  }
}
```

```
::::::::::::::
head.incx
::::::::::::::

#define x 0
#define y 1
#define z 2
#define pi 3.1415926

char *Quit_window();
char *Hardcopy_window();
char *answer_window();
#include <math.h>
#include <stdio.h>
```

::::::::::::::
nofun.cx
::::::::::::::

```
#include "sweepparm2.inc"
#include "Table_var.inc"
#include "e.c"
#include "Rotat_Segment.c"

main()
{
int polyhedron_defined,link_no;
     environment();                      /* e.c */
     message_window(" IBM ?","PUMA ?","choose p/i");
     buffer = *answer_window();
 switch(buffer)
   {
    case 'p':
                PUMA_links(); /* PUMAlinks.c */
                link_no = 5;
                HC_QSet_Camera_Position("?picture/geometry/links",0.0,0.0,70.0);
                break;
    case 'i':
                IBM_links(); /* "IBMlinks.c" */
                link_no = 4;
                HC_QSet_Camera_Position("?picture/geometry/links",0.0,0.0,50.0);

                message_window("orthographic","","y/n");
                if(*answer_window() == 'y')
                HC_QSet_Camera_Projection("?picture/geometry/links","orthographic");
                break;

   }


        for(;;) rotate_segments("?picture/geometry/links");     /* "Rotat_Segment.c" */
  }
```

```
::::::::::::::
sweepparm2.incx
::::::::::::::
/* THE FIRST [6] MEANS THERE ARE AT MOST 6 LINK FOR A ROBOT */


#define x 0
#define y 1
#define z 2
#define roll 3
#define pitch 4
#define yaw 5
#define pi 3.1415926
#define Division 12

char *Quit_window();
char *Hardcopy_window();
char *answer_window();
float autox(),autoy(),autoz();
float  I_Sin_Sin(),I_Cos_Cos(),I_Sin_Cos();

#include <math.h>
#include <stdio.h>
#include <string.h>

float trajectory_configuration[6][4][6]; /* the last 6 is 6 dof */
float trajectory_point[6][Division+1][6];  /* the last 6 is 6 dof */
float phi_image_object, theta_image_object;
int polygon[6][40][20], cycle[6][40][20], edge[6][100][2];
float  vertex[6][100][3];
float edge_length[6][40][20];
/*** float N[4][4], Nu[3][4];  */
float ray_position_image[3], ray_direction_image[3];
float scan_line_position_image[3];
float L[3][2], S[3];
short color[7][40];
int number_segments[6];
int number_polygons[6], number_edges[6][40], total_number_edges[6];
int number_polys[6], number_poly_vertices[6][3000];
int number_links, curve_increment;
int x_minimum_pixel, x_maximum_pixel;
int y_minimum_pixel, y_maximum_pixel;
int no_of_vertix[7];
long keynumber;
char buffer;
int link_no_buffer;


float point_object[Division+1][3][20][30], point_image[Division+1][3][20][30];
float NQ[4][4][3][20][30],NQ_image[4][4][3][20][30];
```

```
::::::::::::::
trans1.cx
::::::::::::::
#include "sweepparm2.inc"
#define jokee 1.0

transform_vertices(links)
  int links;
   {
        float t[3][3];
        int i,j,k,pc_curve,face;

     /* Transform the vertices of each polygon[links] of the polyhedron
        according to the position and orientation of the four points
        that define the trajectory.  For some swept objects of a polygon[links],
        the size of the polygon[links] may change considerably, i.e., +-2%,
        so the four point form of the swept object is subdivided into
        either 2 or 4 segments.  A record of this subdivision is maintained
        to be used for correct display. */

          curve_increment = 1;


        for( i = 0; i<Division+1; i=i+curve_increment)
         {
          t[0][0] = cos(trajectory_point[links][i][roll]/jokee)
                *cos(trajectory_point[links][i][pitch]/jokee);
          t[1][0] = sin(trajectory_point[links][i][roll]/jokee)
                *cos(trajectory_point[links][i][pitch]/jokee);
          t[2][0] = -sin(trajectory_point[links][i][pitch]/jokee);
          t[0][1] = cos(trajectory_point[links][i][roll]/jokee)
                *sin(trajectory_point[links][i][pitch]/jokee)
                *sin(trajectory_point[links][i][yaw]/jokee)
                - sin(trajectory_point[links][i][roll]/jokee)
                *cos(trajectory_point[links][i][yaw]/jokee);
          t[1][1] = sin(trajectory_point[links][i][roll]/jokee)
                *sin(trajectory_point[links][i][pitch]/jokee)
                *sin(trajectory_point[links][i][yaw]/jokee)
                + cos(trajectory_point[links][i][roll]/jokee)
                *cos(trajectory_point[links][i][yaw]/jokee);
          t[2][1] = cos(trajectory_point[links][i][pitch]/jokee)
                *sin(trajectory_point[links][i][yaw]/jokee);
          t[0][2] = cos(trajectory_point[links][i][roll]/jokee)
                *sin(trajectory_point[links][i][pitch]/jokee)
                *cos(trajectory_point[links][i][yaw]/jokee)
                + sin(trajectory_point[links][i][roll]/jokee)
                *sin(trajectory_point[links][i][yaw]/jokee);
          t[1][2] = sin(trajectory_point[links][i][roll]/jokee)
                *sin(trajectory_point[links][i][pitch]/jokee)
                *cos(trajectory_point[links][i][yaw]/jokee)
                - cos(trajectory_point[links][i][roll]/jokee)
                *sin(trajectory_point[links][i][yaw]/jokee);
          t[2][2] = cos(trajectory_point[links][i][pitch]/jokee)
                *cos(trajectory_point[links][i][yaw]/jokee);



          for (j = 0;j<3;j++)
            for (k = 0;k<3;k++)
              if (t[j][k]<0.0001 && t[j][k]>-0.0001)
                t[j][k] = 0.0;

    for (face = 0;face<number_polygons[links];face++)
      for (pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
       {
```

```
        point_object[i][x][pc_curve][face] =
          t[0][x]*vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][x]
          + t[0][y]*vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][
          + t[0][z]*vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][
          + trajectory_point[links][i][x];
        point_object[i][y][pc_curve][face] =
          t[1][x]*vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][x]
          + t[1][y]*vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][
          + t[1][z]*vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][
          + trajectory_point[links][i][y];
        point_object[i][z][pc_curve][face] =
          t[2][x]*vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][x]
          + t[2][y]*vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][
          + t[2][z]*vertex[links][edge[links][polygon[links][face][pc_curve]][cycle[links][face][pc_curve]]][
                  + trajectory_point[links][i][z];
      }
    }
  }
```

```
::::::::::::::
transform.cx
::::::::::::::
#include "sweepparm2.inc"

/* This function is using the roll, pitch, and yaw angle to transform points
    with respect to the world coordinate frame. */

transform(XX,YY,ZZ,rollZ,pitchY,yawX,link)
float XX,YY,ZZ,rollZ,pitchY,yawX;
int link;
  {
    float t[3][3];
    float pseudoX,pseudoY,pseudoZ;
    int i,j,pc_curve,face;
    t[0][0] = cos(rollZ)*cos(pitchY);
    t[1][0] = sin(rollZ)*cos(pitchY);
    t[2][0] = -sin(pitchY);
    t[0][1] = cos(rollZ)*sin(pitchY)*sin(yawX)-sin(rollZ)*cos(yawX);
    t[1][1] = sin(rollZ)*sin(pitchY)*sin(yawX)+cos(rollZ)*cos(yawX);
    t[2][1] = cos(pitchY)*sin(yawX);
    t[0][2] = cos(rollZ)*sin(pitchY)*cos(yawX)+sin(rollZ)*sin(yawX);
    t[1][2] = sin(rollZ)*sin(pitchY)*cos(yawX)-cos(rollZ)*sin(yawX);
    t[2][2] = cos(pitchY)*cos(yawX);

    for(i=0;i<=2;i++)
      for(j=0;j<=2;j++)
        if(fabs(t[i][j])<0.00001) t[i][j]=0.0;

        for (pc_curve=0;pc_curve<no_of_vertix[link];pc_curve++)
          {
            pseudoX =  vertex[link][pc_curve][x] ;
            pseudoY =  vertex[link][pc_curve][y] ;
            pseudoZ =  vertex[link][pc_curve][z] ;

            vertex[link][pc_curve][x]=t[0][0]*pseudoX+t[0][1]*pseudoY+t[0][2]*pseudoZ+XX;
            vertex[link][pc_curve][y]=t[1][0]*pseudoX+t[1][1]*pseudoY+t[1][2]*pseudoZ+YY;
            vertex[link][pc_curve][z]=t[2][0]*pseudoX+t[2][1]*pseudoY+t[2][2]*pseudoZ+ZZ;
          }
  }
Y
"NORTON","NORTON UTILITIES","",1
2
"SPEED DISK",0,1,""
SD
""
"NORTON UTILITIES",0,1,""
NI
""
```

APPENDIX II

Shading programs

```
::::::::::::::
enclosobj.cx
::::::::::::::
#include "sweepparm2.inc"

enclose_object(NQ_image,surface_minimum_point,surface_maximum_point,links)
  float NQ_image[4][4][3][20][30],surface_minimum_point[4][2][20][30],
        surface_maximum_point[4][2][20][30];
  int links;
  {
        float swept_object_minimum_point[2],swept_object_maximum_point[2];
        float pc_point_image[2];
        int accumulator, ADDs;
        float u,increment;
        int pc_curve,previous_pc_curve,segment,face;

    /*  Initialize the minimum and maximum points of the swept object itself and the
        minimum and maximum points of all the ruled surfaces in the swept object.
        These points are in the image coordinates.   */

        swept_object_minimum_point[x] = -10.0;
        swept_object_maximum_point[x] = 10.0;
        swept_object_minimum_point[y] = -10.0;
        swept_object_maximum_point[y] = 10.0;

        for(face = 0;face < number_polygons[links]; face++)
         {
            for(pc_curve = 0; pc_curve < number_edges[links][face]; pc_curve++)
              for(segment = 0; segment < number_segments[links]; segment++)
               {
                surface_minimum_point[segment][x][pc_curve][face] = -10.0;
                surface_maximum_point[segment][x][pc_curve][face] = 10.0;
                surface_minimum_point[segment][y][pc_curve][face] = -10.0;
                surface_maximum_point[segment][y][pc_curve][face] = 10.0;
               }

    /*  Calculate the points along both of the parametric cubic curves for each ruled
        surface given in image coordinates.  Determine the minimum and maximum points of
        each ruled surface and of the swept object.  */


            increment = 1.0*number_segments[links]/12.0;
                ADDs=0;accumulator = 0;
                ADDs = number_segments[links];

            for(pc_curve = 0;pc_curve < number_edges[links][face]; pc_curve++)
             {
              if (pc_curve == 0)
                previous_pc_curve = number_edges[links][face] - 1;

              else
                previous_pc_curve = pc_curve - 1;

              for(segment = 0;segment < number_segments[links]; segment++)
               {
                for(u = 0.0;u <= 1.01;u += increment)
                 {
                   accumulator += ADDs;
                   pc_point_image[x] = point_image[accumulator][x][pc_curve][face];
                   pc_point_image[y] = point_image[accumulator][y][pc_curve][face];
     /* Calculate the bounding rectangle for each of the ruled surfaces. */

                   if (pc_point_image[x] <
                        surface_minimum_point[segment][x][pc_curve][face])
```

```
                    surface_minimum_point[segment][x][pc_curve][face]
                        = pc_point_image[x];

                  if (pc_point_image[x] <
                  surface_minimum_point[segment][x][previous_pc_curve][face])
                    surface_minimum_point[segment][x][previous_pc_curve]
                        [face]= pc_point_image[x];

                  if (pc_point_image[x] >
                        surface_maximum_point[segment][x][pc_curve][face])
                    surface_maximum_point[segment][x][pc_curve][face]
                        = pc_point_image[x];


                  if (pc_point_image[x] >
                  surface_maximum_point[segment][x][previous_pc_curve][face])
                    surface_maximum_point[segment][x][previous_pc_curve]
                        [face]= pc_point_image[x];

                  if (pc_point_image[y] <
                        surface_minimum_point[segment][y][pc_curve][face])
                    surface_minimum_point[segment][y][pc_curve][face]
                        = pc_point_image[y];

                  if (pc_point_image[y] <
                  surface_minimum_point[segment][y][previous_pc_curve][face])
                    surface_minimum_point[segment][y][previous_pc_curve]
                        [face]=pc_point_image[y];

                  if (pc_point_image[y] >
                        surface_maximum_point[segment][y][pc_curve][face])
                    surface_maximum_point[segment][y][pc_curve][face]
                        = pc_point_image[y];

                  if (pc_point_image[y] >
                  surface_maximum_point[segment][y][previous_pc_curve][face])
                    surface_maximum_point[segment][y][previous_pc_curve]
                        [face]=pc_point_image[y];

   /*  Calculate the bounding rectangle for the swept object.   */

                  if (pc_point_image[x] <
                        swept_object_minimum_point[x])
                    swept_object_minimum_point[x]
                        = pc_point_image[x];

                  if (pc_point_image[x] >
                        swept_object_maximum_point[x])
                    swept_object_maximum_point[x]
                        = pc_point_image[x];

                  if (pc_point_image[y] <
                        swept_object_minimum_point[y])
                    swept_object_minimum_point[y]
                        = pc_point_image[y];

                  if (pc_point_image[y] >
                        swept_object_maximum_point[y])
                    swept_object_maximum_point[y]
                        = pc_point_image[y];

                }
              }
            }
        }
```

```
    /* Transform the minimum and maximum points of the swept object given in image
       coordinates into the pixel location of the raster display.  Increase the bounding
       rectangle around the object by 2 pixels in each direction.  */

       x_minimum_pixel = (swept_object_minimum_point[x] + 10.0) *(512.0/20.0) - 2;
       x_maximum_pixel = (swept_object_maximum_point[x] + 10.0) *(512.0/20.0) + 2;
       y_minimum_pixel = (swept_object_minimum_point[y] + 8.0) *(410.0/16.0) - 2;
       y_maximum_pixel = (swept_object_maximum_point[y] + 8.0) *(410.0/16.0) + 2;
}
```

::::::::::::::
illumodl1.cx
::::::::::::::

```
illumination_model_one(unit_normal,L,S,intensity)

  float unit_normal[3],L[3][5],S[3],*intensity;
  {
        float B[4],R[3];
        float dot1,dot2;
        float d,K,ka,kd,ks,n;
        float Ia,Il;
        FILE *constant_data;
    /*  The illumination model for a single light source for color
        display is
                       Il
          I = Ia*ka + -----[kd*(n^.L^) + ks*(R^.S^)**n]
                      d + K

        where

        I  = reflected intensity
        Ia = incident ambient light intensity
        Il = incident point source light intensity
        ka = ambient diffuse reflection constant (0 <= ka <= 1)
        kd = diffuse reflection constant (0 <= kd <= 1)
        ks = experimental constant representing reflectance curve w(i,lambda)
        d  = distance from the closest object to the viewpoint
        K  = arbitrary constant
        n  = approximates spatial distribution of specularly reflected light
        n^ = unit surface normal vector at current pixel
        L^ = unit light source direction vector
        R^ = unit reflected ray direction vector
        S^ = unit line-of-sight direction vector
    */
        Ia = 0.3;
        ka = 1.0;
        K = 1.0;
        Il = 0.7;
        kd = 0.45;
        ks = 0.55;
        d = 0.0;
        n = 2.0;

     /* The unit reflected ray vector is found by the two equations: n^xL^ = R^xn^ to ensure
        planarity and n^.L^ = n^.R^ to ensure equal angles between vectors.  */

        B[0] = unit_normal[z]*L[y][0] - unit_normal[y]*L[z][0];
        B[1] = unit_normal[x]*L[z][0] - unit_normal[z]*L[x][0];
        B[2] = unit_normal[y]*L[x][0] - unit_normal[x]*L[y][0];
        B[3] = unit_normal[x]*L[x][0] + unit_normal[y]*L[y][0] + unit_normal[z]*L[z][0];

        R[x] = unit_normal[x]*B[3] - unit_normal[y]*B[2] + unit_normal[z]*B[1];
        R[y] = unit_normal[x]*B[2] + unit_normal[y]*B[3] - unit_normal[z]*B[0];
        R[z] = -unit_normal[x]*B[1] + unit_normal[y]*B[0] + unit_normal[z]*B[3];

    /*  If the angle between the unit surface normal vector and the unit light source direction
        vector is greater than 90.0 (or cos(angle) < 0.0) then the light source is behind
        the object.  */

        dot1 = unit_normal[x]*L[x][0] + unit_normal[y]*L[y][0] + unit_normal[z]*L[z][0];
        dot1 = dot1/(sqrt(unit_normal[x]*unit_normal[x]+
                          unit_normal[y]*unit_normal[y]+
                          unit_normal[z]*unit_normal[z])
```

```
                *sqrt(L[x][0]*L[x][0]+L[y][0]*L[y][0]+L[z][0]*L[z][0]));
    if (dot1 < 0.0) dot1 = 0.0;

/*  If the angle between the unit reflected ray direction vector and the unit
    line-of-sight direction vector is greater than 90.0 (or cos(angle) < 0.0)
    then the reflected ray cannot be seen.  */

    dot2 = R[x]*S[x] + R[y]*S[y] + R[z]*S[z];
    dot2 = dot2/(sqrt(R[x]*R[x] + R[y]*R[y] + R[z]*R[z])
                *sqrt(S[x]*S[x] + S[y]*S[y] + S[z]*S[z]));


    if (dot2 < 0.0) dot2 = 0.0;

    *intensity = Ia*ka  + (Il/(d + K))*(kd*dot1 +ks* pow(dot2,n));

    if (*intensity > 1.0) { printf("%f\n",*intensity); *intensity = 1.0;}

}
```

```
::::::::::::::
illumodl2.cx
::::::::::::::

illumination_model_two(unit_normal,L,S,intensity)

  float unit_normal[3],L[3][5],S[3],*intensity;
  {
        float B[4][2],R[3][2];
        float dot1[2],dot2[2];
        float d[2],K,ka,kd[2],ks[2],n[2];
        float Ia,Il[2];
        FILE *constant_data;

   /*   The illumination model for a single light source for color display is

                       Il
          I = Ia*ka + -----[kd*(n^.L^) + ks*(R^.S^)**n]
                      d + K

        where

        I  = reflected intensity
        Ia = incident ambient light intensity
        Il = incident point source light intensity
        ka = ambient diffuse reflection constant (0 <= ka <= 1)
        kd = diffuse reflection constant (0 <= kd <= 1)
        ks = experimental constant representing reflectance curve w(i,lambda)
        d  = distance from the closest object to the viewpoint
        K  = arbitrary constant
        n  = approximates spatial distribution of specularly reflected light
        n^ = unit surface normal vector at current pixel
        L^ = unit light source direction vector
        R^ = unit reflected ray direction vector
        S^ = unit line-of-sight direction vector
    */
        Ia = 0.33;
        ka = 0.90;
        K = 1.0;
        Il[0] = 0.72;
        Il[1] = 0.72;
        kd[0] = 0.23;
        kd[1] = 0.23;
        ks[0] = 0.27;
        ks[1] = 0.27;
        d[0] = 0.0;
        d[1] = 0.0;
        n[0] = 2.0;
        n[1] = 2.0;

    /*  The unit reflected ray vector is found by the two equations: n^xL^ = R^xn^ to ensure
        planarity and n^.L^ = n^.R^ to ensure equal angles between vectors.  */

        B[0][0] = unit_normal[z]*L[y][0] - unit_normal[y]*L[z][0];
        B[1][0] = unit_normal[x]*L[z][0] - unit_normal[z]*L[x][0];
        B[2][0] = unit_normal[y]*L[x][0] - unit_normal[x]*L[y][0];
        B[3][0] = unit_normal[x]*L[x][0] + unit_normal[y]*L[y][0] + unit_normal[z]*L[z][0];

        R[x][0] = unit_normal[x]*B[3][0] - unit_normal[y]*B[2][0] + unit_normal[z]*B[1][0];
        R[y][0] = unit_normal[x]*B[2][0] + unit_normal[y]*B[3][0] - unit_normal[z]*B[0][0];
        R[z][0] = -unit_normal[x]*B[1][0] + unit_normal[y]*B[0][0] + unit_normal[z]*B[3][0];

    /*  If the angle between the unit surface normal vector and the unit light source direction
        vector is greater than 90.0 (or cos(angle) < 0.0) then the light source is behind
```

```
      the object.  */

      dot1[0] = unit_normal[x]*L[x][0] + unit_normal[y]*L[y][0] + unit_normal[z]*L[z][0];

     dot1[0] = dot1[0]/(sqrt(unit_normal[x]*unit_normal[x]+
                        unit_normal[y]*unit_normal[y]+
                        unit_normal[z]*unit_normal[z])
                  *sqrt(L[x][0]*L[x][0]+L[y][0]*L[y][0]+L[z][0]*L[z][0]));

      if (dot1[0] < 0.0) dot1[0] = 0.0;

   /*  If the angle between the unit reflected ray direction vector and the unit line-of-sight
      direction vector is greater than 90.0 (or cos(angle) < 0.0) then the reflected ray
      cannot be seen.  */

      dot2[0] = R[x][0]*S[x] + R[y][0]*S[y] + R[z][0]*S[z];
      dot2[0] = dot2[0]/(sqrt(R[x][0]*R[x][0] + R[y][0]*R[y][0] + R[z][0]*R[z][0])
                  *sqrt(S[x]*S[x] + S[y]*S[y] + S[z]*S[z]));


      if (dot2[0] < 0.0) dot2[0] = 0.0;

      B[0][1] = unit_normal[z]*L[y][1] - unit_normal[y]*L[z][1];
      B[1][1] = unit_normal[x]*L[z][1] - unit_normal[z]*L[x][1];
      B[2][1] = unit_normal[y]*L[x][1] - unit_normal[x]*L[y][1];
      B[3][1] = unit_normal[x]*L[x][1] + unit_normal[y]*L[y][1] + unit_normal[z]*L[z][1];

      R[x][1] = unit_normal[x]*B[3][1] - unit_normal[y]*B[2][1] + unit_normal[z]*B[1][1];
      R[y][1] = unit_normal[x]*B[2][1] + unit_normal[y]*B[3][1] - unit_normal[z]*B[0][1];
      R[z][1] = -unit_normal[x]*B[1][1] + unit_normal[y]*B[0][1] + unit_normal[z]*B[3][1];

   /*  If the angle between the unit surface normal vector and the unit light source direction
      vector is greater than 90.0 (or cos(angle) < 0.0) then the light source is behind the object.  */

      dot1[1] = unit_normal[x]*L[x][1] + unit_normal[y]*L[y][1]
             + unit_normal[z]*L[z][1];

      dot1[1] = dot1[1]/(sqrt(unit_normal[x]*unit_normal[x]+
                        unit_normal[y]*unit_normal[y]+
                        unit_normal[z]*unit_normal[z])
                  *sqrt(L[x][1]*L[x][1]+L[y][1]*L[y][1]+L[z][1]*L[z][1]));

      if (dot1[1] < 0.0) dot1[1] = 0.0;

   /*  If the angle between the unit reflected ray direction vector and the unit
      line-of-sight direction vector is greater than 90.0 (or cos(angle) < 0.0) then
      the reflected ray cannot be seen.  */

      dot2[1] = R[x][1]*S[x] + R[y][1]*S[y] + R[z][1]*S[z];

    dot2[1] = dot2[1]/(sqrt(R[x][1]*R[x][1] + R[y][1]*R[y][1] + R[z][1]*R[z][1])
                  *sqrt(S[x]*S[x] + S[y]*S[y] + S[z]*S[z]));

      if (dot2[1] < 0.0) dot2[1] = 0.0;

      *intensity = Ia*ka  + (Il[0]/(d[0] + K))*(kd[0]*dot1[0]
             + ks[0]*pow(dot2[0],n[0]))
                   + (Il[1]/(d[1] + K))*(kd[1]*dot1[1]
             + ks[1]*pow(dot2[1],n[1]));

      if (*intensity > 1.0) *intensity = 1.0;


}
```

::::::::::::::
scngntint.cx
::::::::::::::

```
/* There are two types of increase the efficiency of scan-line algorithms 1) scan-line coherence
   2) geomeytrical simplification Here I am using type 2) geomeytrical simplification :
   the particular choice of "windows" examined by the algorithm, the windows are one scan-line high
   and span the width of the screen. */

generate_scanline_intersections(face_color,ruled_polys,polygon_vertices,polygon_normal,
                                polygon_minimum_point,polygon_maximum_point,depth_array,
                                normal_array,color_array,plane_normal_distance,junkee)

  int   face_color[90][2],ruled_polys;
  float polygon_vertices[3000][3],polygon_normal[3000][3],
        polygon_minimum_point[2][3000],polygon_maximum_point[2][3000];
  float depth_array[512][410],normal_array[512][410][3];
  short color_array[512][410];
  float plane_normal_distance[3000];
  int junkee;
  {
        float vertex_normal[20][3],normal_magnitude;
        int scan_line_increment,scan_line_signal;
        int scan_lines_processed,amount_processed;
        int i,j,k,poly,face;

        scan_line_position_image[x] = -10.0;

   /*   Subdivide the total number of scan lines needed to process the object into 10 increments.
        During rendering write to the terminal the amount of the scene that has been processed
        in 1/10ths of the total number of scan lines.   */


        scan_line_increment=(y_maximum_pixel-y_minimum_pixel+1)/10.0+0.5;
        scan_line_signal = scan_line_increment;
        amount_processed = 1;
        scan_lines_processed = 0;

        for(j=y_minimum_pixel;j<=y_maximum_pixel;j++)
         {
          scan_lines_processed = scan_lines_processed + 1;
          if ((scan_lines_processed==scan_line_signal) ||
              (scan_lines_processed==y_maximum_pixel-y_minimum_pixel+1 &&
               amount_processed <= 10))
             {
               printf("%d",amount_processed);
               printf("/10 processed\n");
               scan_line_signal = scan_line_signal + scan_line_increment;
               amount_processed = amount_processed + 1;
             }


  /*  Calculate the new scan line position in the image coordinate system. This position
      must be converted from pixels to the object size of the vector screen.
      Here -10.0 < x < 10.0 corresponds to 0 < x_pixel < 512 and -8.0 < y < 8.0
      corresponds to 0 < y_pixel < 410.  The position vector is different for each screen pixel.*/

          scan_line_position_image[y] = j/(410.0/16.0)-8.0;

  /* Processing each scan-line */
  /* The scan-line algorithm must decide what polygons are visible in a scan-line window,
     and these decisions are all made by comparing line segments in the X-Z plane. */

  /* For all the polygons in the swept object calculate the intersections with each scan line.
     Only search for polygon/scan_line intersections with those polygons in which the scan line
```

```
    is within the bounding rectangle.   */

        i = 0;

        for(poly = 0; poly <= ruled_polys; poly++)
         {
          if (poly < face_color[i][0])
            face = face_color[i][1];

          else if (poly == face_color[i][0])
                {
                 face = face_color[i][1];
                 i = i + 1;
                }

          if (polygon_normal[poly][z] != 0.0)
            if (scan_line_position_image[y] >=
                       polygon_minimum_pcint[y][poly]
                       && scan_line_position_image[y] <=
                       polygon_maximum_point[y][poly])
             {
   /*  "even" polygons numbers   */

               if (poly % 2 == 0)
                 if (poly % 24 == 0)
                  {
                   vertex_normal[0][x] = polygon_normal[poly][x]
                       + polygon_normal[poly+1][x];
                   vertex_normal[0][y] = polygon_normal[poly][y]
                       + polygon_normal[poly+1][y];
                   vertex_normal[0][z] = polygon_normal[poly][z]
                       + polygon_normal[poly+1][z];

                   vertex_normal[1][x] = polygon_normal[poly][x];
                   vertex_normal[1][y] = polygon_normal[poly][y];
                   vertex_normal[1][z] = polygon_normal[poly][z];

                   vertex_normal[2][x] = polygon_normal[poly][x]
                       + polygon_normal[poly+1][x]
                       + polygon_normal[poly+2][x];
                   vertex_normal[2][y] = polygon_normal[poly][y]
                       + polygon_normal[poly+1][y]
                       + polygon_normal[poly+2][y];
                   vertex_normal[2][z] = polygon_normal[poly][z]
                       + polygon_normal[poly+1][z]
                       + polygon_normal[poly+2][z];
                  }

                 else if (poly % 24 == 22)
                  {
                   vertex_normal[0][x] = polygon_normal[poly-1][x]
                       + polygon_normal[poly][x]
                       + polygon_normal[poly+1][x];
                   vertex_normal[0][y] = polygon_normal[poly-1][y]
                       + polygon_normal[poly][y]
                       + polygon_normal[poly+1][y];
                   vertex_normal[0][z] = polygon_normal[poly-1][z]
                       + polygon_normal[poly][z]
                       + polygon_normal[poly+1][z];

                   vertex_normal[1][x] = polygon_normal[poly-2][x]
                       + polygon_normal[poly-1][x]
                       + polygon_normal[poly][x];
                   vertex_normal[1][y] = polygon_normal[poly-2][y]
                       + polygon_normal[poly-1][y]
```

```
                        + polygon_normal[poly][y];
                    vertex_normal[1][z] = polygon_normal[poly-2][z]
                        + polygon_normal[poly-1][z]
                        + polygon_normal[poly][z];

                    vertex_normal[2][x] = polygon_normal[poly][x]
                        + polygon_normal[poly+1][x];
                    vertex_normal[2][y] = polygon_normal[poly][y]
                        + polygon_normal[poly+1][y];
                    vertex_normal[2][z] = polygon_normal[poly][z]
                        + polygon_normal[poly+1][z];
                  }
                else
                  {
                    vertex_normal[0][x] = polygon_normal[poly-1][x]
                        + polygon_normal[poly][x]
                        + polygon_normal[poly+1][x];
                    vertex_normal[0][y] = polygon_normal[poly-1][y]
                        + polygon_normal[poly][y]
                        + polygon_normal[poly+1][y];
                    vertex_normal[0][z] = polygon_normal[poly-1][z]
                        + polygon_normal[poly][z]
                        + polygon_normal[poly+1][z];

                    vertex_normal[1][x] = polygon_normal[poly-2][x]
                        + polygon_normal[poly-1][x]
                        + polygon_normal[poly][x];
                    vertex_normal[1][y] = polygon_normal[poly-2][y]
                        + polygon_normal[poly-1][y]
                        + polygon_normal[poly][y];
                    vertex_normal[1][z] = polygon_normal[poly-2][z]
                        + polygon_normal[poly-1][z]
                        + polygon_normal[poly][z];

                    vertex_normal[2][x] = polygon_normal[poly][x]
                        + polygon_normal[poly+1][x]
                        + polygon_normal[poly+2][x];
                    vertex_normal[2][y] = polygon_normal[poly][y]
                        + polygon_normal[poly+1][y]
                        + polygon_normal[poly+2][y];
                    vertex_normal[2][z] = polygon_normal[poly][z]
                        + polygon_normal[poly+1][z]
                        + polygon_normal[poly+2][z];

                  }

/*   "odd" polygon numbers   */

              else
                if (poly % 24 == 23)
                  {
                    vertex_normal[0][x] = polygon_normal[poly][x]
                        + polygon_normal[poly-1][x];
                    vertex_normal[0][y] = polygon_normal[poly][y]
                        + polygon_normal[poly-1][y];
                    vertex_normal[0][z] = polygon_normal[poly][z]
                        + polygon_normal[poly-1][z];

                    vertex_normal[1][x] = polygon_normal[poly][x];
                    vertex_normal[1][y] = polygon_normal[poly][y];
                    vertex_normal[1][z] = polygon_normal[poly][z];

                    vertex_normal[2][x] = polygon_normal[poly-2][x]
                        + polygon_normal[poly-1][x]
                        + polygon_normal[poly][x];
```

```
        vertex_normal[2][y] = polygon_normal[poly-2][y]
            + polygon_normal[poly-1][y]
            + polygon_normal[poly][y];
        vertex_normal[2][z] = polygon_normal[poly-2][z]
            + polygon_normal[poly-1][z]
            + polygon_normal[poly][z];
      }
    else if (poly % 24 == 1)
      {
        vertex_normal[0][x] = polygon_normal[poly-1][x]
            + polygon_normal[poly][x]
            + polygon_normal[poly+1][x];
        vertex_normal[0][y] = polygon_normal[poly-1][y]
            + polygon_normal[poly][y]
            + polygon_normal[poly+1][y];
        vertex_normal[0][z] = polygon_normal[poly-1][z]
            + polygon_normal[poly][z]
            + polygon_normal[poly+1][z];

        vertex_normal[1][x] = polygon_normal[poly][x]
            + polygon_normal[poly+1][x]
            + polygon_normal[poly+2][x];
        vertex_normal[1][y] = polygon_normal[poly][y]
            + polygon_normal[poly+1][y]
            + polygon_normal[poly+2][y];
        vertex_normal[1][z] = polygon_normal[poly][z]
            + polygon_normal[poly+1][z]
            + polygon_normal[poly+2][z];

        vertex_normal[2][x] = polygon_normal[poly-1][x]
            + polygon_normal[poly][x];
        vertex_normal[2][y] = polygon_normal[poly-1][y]
            + polygon_normal[poly][y];
        vertex_normal[2][z] = polygon_normal[poly-1][z]
            + polygon_normal[poly][z];
      }

    else
      {
        vertex_normal[0][x] = polygon_normal[poly-1][x]
            + polygon_normal[poly][x]
            + polygon_normal[poly+1][x];
        vertex_normal[0][y] = polygon_normal[poly-1][y]
            + polygon_normal[poly][y]
            + polygon_normal[poly+1][y];
        vertex_normal[0][z] = polygon_normal[poly-1][z]
            + polygon_normal[poly][z]
            + polygon_normal[poly+1][z];

        vertex_normal[1][x] = polygon_normal[poly][x]
            + polygon_normal[poly+1][x]
            + polygon_normal[poly+2][x];
        vertex_normal[1][y] = polygon_normal[poly][y]
            + polygon_normal[poly+1][y]
            + polygon_normal[poly+2][y];
        vertex_normal[1][z] = polygon_normal[poly][z]
            + polygon_normal[poly+1][z]
            + polygon_normal[poly+2][z];

        vertex_normal[2][x] = polygon_normal[poly-2][x]
            + polygon_normal[poly-1][x]
            + polygon_normal[poly][x];
        vertex_normal[2][y] = polygon_normal[poly-2][y]
            + polygon_normal[poly-1][y]
            + polygon_normal[poly][y];
```

```
                        vertex_normal[2][z] = polygon_normal[poly-2][z]
                            + polygon_normal[poly-1][z]
                            + polygon_normal[poly][z];

                      }

    /*  Return the unit normal vector of each vertex.   */
                  for(k=0;k<3;k++)
                   {
                    normal_magnitude = sqrt(vertex_normal[k][x]*
                        vertex_normal[k][x]+vertex_normal[k][y]*
                        vertex_normal[k][y]+vertex_normal[k][z]*
                        vertex_normal[k][z]);
                    vertex_normal[k][x]=vertex_normal[k][x]/normal_magnitude;
                    vertex_normal[k][y]=vertex_normal[k][y]/normal_magnitude;
                    vertex_normal[k][z]=vertex_normal[k][z]/normal_magnitude;

                   }

                  scan_line_polygon_intersection(poly,j,face,vertex_normal,
                            polygon_vertices,polygon_normal,depth_array,
                            normal_array,color_array,plane_normal_distance,junkee);
                                                  /* scnplyint.c */
               }
             }

            for( poly = ruled_polys+1; poly <= number_polys[junkee]; poly++)
             {
              if (poly < face_color[i][0])
                face = face_color[i][1];

              else if (poly == face_color[i][0])
                   {
                    face = face_color[i][1];
                    i = i + 1;
                   }

              if (polygon_normal[poly][z] != 0.0)
                if (scan_line_position_image[y] >=
                          polygon_minimum_point[y][poly]
                          && scan_line_position_image[y] <=
                          polygon_maximum_point[y][poly])
                 {
                  for(k = 0;k<number_poly_vertices[junkee][poly]; k++)
                   {
                    vertex_normal[k][x]  = polygon_normal[poly][x];
                    vertex_normal[k][y]  = polygon_normal[poly][y];
                    vertex_normal[k][z]  = polygon_normal[poly][z];
                   }

                  scan_line_polygon_intersection(poly,j,face,vertex_normal,
                            polygon_vertices,polygon_normal,depth_array,
                            normal_array,color_array,plane_normal_distance,junkee);
                                                  /* scnplyint.c */
                 }
             }
           }
}
```

```
::::::::::::::
scngntsld.cx
::::::::::::::
#include "sweepparm2.inc"

      float polygon_vertices[3000][3][20],polygon_normal[3000][3];
      float polygon_vertices[3000][3][20],polygon_normal[3000][3],
            plane_normal_distance[3000],depth_array[512][410], normal_array[512][410][3];
      float polygon_minimum_point[2][3000],polygon_maximum_point[2][3000],
            surface_minimum_point[4][2][20][30], surface_maximum_point[4][2][20][30];

scan_line_generate_solid_data(NQ_image,color_array,image_array,junkee)
  float NQ_image[4][4][3][20][30];
  float image_array[512][410];
  short color_array[512][410];
  int junkee;
{

      double root2,root225,sight;
      FILE *direction;
      float cosine_angle,normal_array_element[3];
      int i,j,face_color[90][2],ruled_polys,illumodl;

{
char *string;
l100:
  message_window("ONE LIGHT SOURCE 1 ?","TWO LIGHT SOURCES 2 ?","CHOOSE (1/2)");
  string=answer_window();
     if (*string == '1') illumodl = 1;
        else if(*string == '2') illumodl = 2;
                else  goto l100;
}


    /*  Find the enclosing rectangle around each ruled surface and around the entire swept object.*/

  enclose_object(NQ_image,surface_minimum_point,surface_maximum_point,junkee);  /*enclosobj.c */

  /*    Initialze the raster elements for the z values and normals of the polygon/scan_line
        intersections, the image array and the color array for raster display.  */

        for(j=y_minimum_pixel-1;j<y_maximum_pixel;j++)
          for(i=x_minimum_pixel-1;i<x_maximum_pixel;i++)
           {
            depth_array[i][j] =  -30.0;
            normal_array[i][j][x] = 0.0;
            normal_array[i][j][y] = 0.0;
            normal_array[i][j][z] = 0.0;
           }

   /*   Initialize the unit light source direction vector L and the unit line-of-sight direction
        vector S in the image coordinate system. */


 message_window("Do you want to define light and view direction ?","","");
   {
      root2 = 1.414214;         /*sqrt(2.0);*/
      root225 = 1.5;            /*sqrt(2.25);*/

        L[x][0] = 0.0;
        L[y][0] = -1.0 /root2;
        L[z][0] = 1.0/root2;
        L[x][1] = 1.0/root225;
        L[y][1] = -1.0/root225;
```

```
        L[z][1] = 0.5/root225;

        S[x] = 0.0;
        S[y] = 0.0;
        S[z] = 1.0;
  }


   /* Polygonalize the ruled surface segments, and calculate their unit normals. Also calculate
      unit normals for the initial and final locations of the polygons in the sweep.*/

      polygonalize_surfaces(face_color,&ruled_polys,NQ_image,polygon_vertices,polygon_normal,
      plane_normal_distance,polygon_minimum_point,polygon_maximum_point,junkee);  /*"scnplynml.c" */


  /*  Begin rendering of all the ruled surfaces and the initial and final polygons using a scan
      line z-buffer algorithm to record the visible surface/ray intersections.  Calculate the
      display attributes of each pixel and store these in an image array.  */

      generate_scanline_intersections(face_color,ruled_polys,polygon_vertices,polygon_normal,
      polygon_minimum_point,polygon_maximum_point,depth_array,normal_array,color_array,
      plane_normal_distance,junkee);          /* scngntint.c */

      for(j = y_minimum_pixel-1;j<y_maximum_pixel;j++)
        for(i = x_minimum_pixel-1;i<x_maximum_pixel;i++)
         {
          if (normal_array[i][j][x] != 0.0 || normal_array[i][j][y] != 0.0
              || normal_array[i][j][z] != 0.0)
           {

  /* Determine the shading of each pixel in the scene by calculating the cosine of the
     angle between the unit line_of_sight direction vector and the unit normal previously
     determined for this pixel. The cosine of the angle is found by taking their inner product. */

     cosine_angle=normal_array[i][j][x]*S[x]+normal_array[i][j][y]*S[y]+normal_array[i][j][z]*S[z];

  /*  Determine if this location of the surface is visible or non-visible by testing the value
      of the cosine of the angle between the unit line-of-sight direction vector and the unit
      normal to the surface (visible when -90.0 < angle < 90.0).  If the surface is visible at
      this location then find the pixel intensity and store it in the image array.  */

            if (cosine_angle < 0.0)
             {
              normal_array[i][j][x] = -normal_array[i][j][x];
              normal_array[i][j][y] = -normal_array[i][j][y];
              normal_array[i][j][z] = -normal_array[i][j][z];
             }

            normal_array_element[x] = normal_array[i][j][x];
            normal_array_element[y] = normal_array[i][j][y];
            normal_array_element[z] = normal_array[i][j][z];
     if(illumodl==1)
          illumination_model_one(normal_array_element,L,S,&image_array[i][j]); /*"illumodl1.c"*/
     else illumination_model_two(normal_array_element,L,S,&image_array[i][j]);/*"illumodl2.c"*/
           }
         }
 }
```

```
::.::::::::::
scnplyint.cx
::::::::::::::
scan_line_polygon_intersection(poly,j,face,vertex_normal,polygon_vertices,
          polygon_normal,depth_array,normal_array,color_array, plane_normal_distance,junkee)

  int   poly,j,face;
  float vertex_normal[20][3],polygon_vertices[3000][3][20],
        polygon_normal[3000][3],depth_array[512][410],
        normal_array[512][410][3],plane_normal_distance[3000];
  short color_array[512][410];
  int junkee;
  {
      float s_parameter,t_parameter,tempr;
      float scan_line_edge_intersection[20];
      float z_depth[512],pixel_normal[512][3],delta_depth,delta_normal[3];
      float u,w;
      int scan_segment_minimum_pixel,scan_segment_maximum_pixel;
      int edge_number[20];
      int minimum_vertex,minimum_next_vertex;
      int maximum_vertex,maximum_next_vertex;
      int i,k,m,vert,next_vert,tempi;
      int number_edge_intersections;

   /*  A polygon is input to this subroutine, i.e., the coordinates of the vertices of the polygon,
       the polygon normal, and the vertex normals determined as the average of the polygon normals
       for all polygons sharing the vertices of the input polygon. These vertex normals are used to
       determine shading using the Phong shading technique which linearly interpolates normals
       across the polygon.

        The polygon is displayed using a scan line z-buffer algorithm. For all the edges in a
        polygon determine the x-coordinate of intersection with the scan line.  */

        number_edge_intersections = -1;

        for(vert = 0; vert < number_poly_vertices[junkee][poly]; vert++)
         {
          if (vert==number_poly_vertices[junkee][poly]-1)
            next_vert = 0;

          else
            next_vert = vert + 1;

    /*  From the parametric representations of the scan line (x(t) = ax + t*bx and
        y(t) = ay + t*by) and the current polygon edge (x(s) = cx + s*dx and y(s) = cy + s*dy)
        solve for t and s by equating x(t) = x(s) and y(t) = y(s) to get
        s = (bx*(cy - ay) - by*(cx - ax))/(dx*by - bx*dy) and t = (cx - ax + s*dx)/bx.
        Note that since he scan line is horizontal, bx = 1 and by = 0 so the equations for
        s and t are simplified.  */

          if (polygon_vertices[poly][y][vert] !=
              polygon_vertices[poly][y][next_vert])

    /*  Only test for intersections with those edges of the polygon that have one endpoint
        above the scan line and one endpoint below the scan line.  */

            if (polygon_vertices[poly][y][vert] <= scan_line_position_image[y] &&
                polygon_vertices[poly][y][next_vert] >= scan_line_position_image[y] ||
                polygon_vertices[poly][y][vert] >= scan_line_position_image[y] &&
                polygon_vertices[poly][y][next_vert] <= scan_line_position_image[y])
             {
              s_parameter = (polygon_vertices[poly][y][vert] - scan_line_position_image[y])
                       /(polygon_vertices[poly][y][vert] - polygon_vertices[poly][y][next_vert]);

```

```
                t_parameter = polygon_vertices[poly][x][vert] - scan_line_position_image[x]
                  - s_parameter*(polygon_vertices[poly][x][vert] - polygon_vertices[poly][x][next_vert]);

                number_edge_intersections = number_edge_intersections + 1;

                scan_line_edge_intersection[number_edge_intersections] =
                  scan_line_position_image[x] + t_parameter;

                edge_number[number_edge_intersections] = vert;
               }
           }

       /* Sort all these scan_line/edge intersections in increasing x_coordinate values. Pairs of
          these x_coordinate intersections form scan line segments that are interior to the polygon.  */

          if (number_edge_intersections > 0)
            for(k = 0;k<=number_edge_intersections-1; k++)
              for( m = number_edge_intersections;m >= k+1; m--)
                if (scan_line_edge_intersection[m-1] >
                          scan_line_edge_intersection[m])
                 {
                  tempr = scan_line_edge_intersection[m-1];
                  scan_line_edge_intersection[m-1] = scan_line_edge_intersection[m];
                  scan_line_edge_intersection[m] = tempr;
                  tempi = edge_number[m-1];
                  edge_number[m-1] = edge_number[m];
                  edge_number[m] = tempi;
                 }

          for(k = 0; k < number_edge_intersections; k += 2)
           {
            minimum_vertex = edge_number[k];

            if (minimum_vertex == number_poly_vertices[junkee][poly]-1)
              minimum_next_vertex = 0;

            else
              minimum_next_vertex = minimum_vertex + 1;

            maximum_vertex = edge_number[k+1];

            if (maximum_vertex == number_poly_vertices[junkee][poly]-1)
              maximum_next_vertex = 0;

            else
              maximum_next_vertex = maximum_vertex + 1;

       /* For both endpoints of a scan line segment determine their pixel values.  */

            scan_segment_minimum_pixel =
                (scan_line_edge_intersection[k]+10.0)*(512.0/20.0)+0.5;
            scan_segment_maximum_pixel =
                (scan_line_edge_intersection[k+1]+10.0)*(512.0/20.0)+0.5;

            i = scan_segment_minimum_pixel-1;

       /* Find the z depth and normal of the pixel at the left endpoint of each scan line segment.  */

            u = (polygon_vertices[poly][y][minimum_vertex] - scan_line_position_image[y])
                  /(polygon_vertices[poly][y][minimum_vertex]
                   - polygon_vertices[poly][y][minimum_next_vertex]);

            w = (polygon_vertices[poly][y][maximum_vertex] - scan_line_position_image[y])
                  /(polygon_vertices[poly][y][maximum_vertex]
                  - polygon_vertices[poly][y][maximum_next_vertex]);
```

```
        z_depth[i] = -(polygon_normal[poly][x] *scan_line_edge_intersection[k]
              + polygon_normal[poly][y] *scan_line_position_image[y]
              + plane_normal_distance[poly]) /polygon_normal[poly][z];

        pixel_normal[i][x] = u*vertex_normal[minimum_next_vertex][x]
                             + (1-u)*vertex_normal[minimum_vertex][x];
        pixel_normal[i][y] = u*vertex_normal[minimum_next_vertex][y]
                             + (1-u)*vertex_normal[minimum_vertex][y];
        pixel_normal[i][z] = u*vertex_normal[minimum_next_vertex][z]
                             + (1-u)*vertex_normal[minimum_vertex][z];

        if (z_depth[i] > depth_array[i][j-1])
         {
          depth_array[i][j-1] = z_depth[i];
          normal_array[i][j-1][x] = pixel_normal[i][x];
          normal_array[i][j-1][y] = pixel_normal[i][y];
          normal_array[i][j-1][z] = pixel_normal[i][z];
          color_array[i][j-1]=color[junkee][face];
         }

        delta_depth = polygon_normal[poly][x] /(polygon_normal[poly][z]*(512.0/20.0));

        if (scan_segment_maximum_pixel == scan_segment_minimum_pixel)
         {
          delta_normal[x] = 0.0;
          delta_normal[y] = 0.0;
          delta_normal[z] = 0.0;
         }
        else
         {
          delta_normal[x] = (w*(vertex_normal[maximum_next_vertex][x]
               - vertex_normal[maximum_vertex][x])
               + vertex_normal[maximum_vertex][x]
               - u*(vertex_normal[minimum_next_vertex][x]
               - vertex_normal[minimum_vertex][x])
               - vertex_normal[minimum_vertex][x])
               /(scan_segment_maximum_pixel-scan_segment_minimum_pixel);

          delta_normal[y] = (w*(vertex_normal[maximum_next_vertex][y]
               - vertex_normal[maximum_vertex][y])
               + vertex_normal[maximum_vertex][y]
               - u*(vertex_normal[minimum_next_vertex][y]
               - vertex_normal[minimum_vertex][y])
               - vertex_normal[minimum_vertex][y])
               /(scan_segment_maximum_pixel-scan_segment_minimum_pixel);

          delta_normal[z] = (w*(vertex_normal[maximum_next_vertex][z]
               - vertex_normal[maximum_vertex][z])
               + vertex_normal[maximum_vertex][z]
               - u*(vertex_normal[minimum_next_vertex][z]
               - vertex_normal[minimum_vertex][z])
               - vertex_normal[minimum_vertex][z])
               /(scan_segment_maximum_pixel-scan_segment_minimum_pixel);

         }

   /*  Interpolate the remaining z depths and normals for all other pixels of each
       scan line segment.  */

         for(i=scan_segment_minimum_pixel;i<scan_segment_maximum_pixel;
             i++)
          {
           z_depth[i] = z_depth[i-1] - delta_depth;

```

```
            pixel_normal[i][x] = pixel_normal[i-1][x] + delta_normal[x];
            pixel_normal[i][y] = pixel_normal[i-1][y] + delta_normal[y];
            pixel_normal[i][z] = pixel_normal[i-1][z] + delta_normal[z];

            if (z_depth[i] > depth_array[i][j-1])
             {
              depth_array[i][j-1] = z_depth[i];

              normal_array[i][j-1][x] = pixel_normal[i][x];
              normal_array[i][j-1][y] = pixel_normal[i][y];
              normal_array[i][j-1][z] = pixel_normal[i][z];

              color_array[i][j-1]=color[junkee][face];
             }
           }
         }
 }
```

```
::::::::::::::
scnplynml.cx
::::::::::::::
#include "sweepparm2.inc"

polygonalize_surfaces(face_color,ruled_polys,NQ_image,polygon_vertices,
                          polygon_normal,plane_normal_distance,
                          polygon_minimum_point,polygon_maximum_point,links)

  int face_color[90][2],*ruled_polys;
  float NQ_image[4][4][3][20][30],polygon_vertices[3000][3][20],
        polygon_normal[3000][3],plane_normal_distance[3000],
        polygon_minimum_point[2][3000],polygon_maximum_point[2][3000];
  int links;

  {
        float A[3],B[3],u;
        float normal_magnitude,increment;
        int pc_curve,next_pc_curve,segment,face;
        int i,j,vert;
        short lines_are_parallel,segment_start;
        int ADDs,accumulator;

        number_polys[links] = -1;

        increment = 1.0*number_segments[links]/Division;
                ADDs=0;
                ADDs = number_segments[links];
                accumulator =  -ADDs;
        i = -1;

     /* Calculate the vertices of the polygons representing the ruled surface segments, and
        the initial and final polygons in the sweep in the image coordinate system. */

        for(face=0;face<number_polygons[links];face++)
          {
            i = i + 1;

            for(pc_curve=0;pc_curve<number_edges[links][face];pc_curve++)
             {
              if (pc_curve==number_edges[links][face]-1)
                next_pc_curve = 0;

              else
                next_pc_curve = pc_curve + 1;

              for(segment = 0; segment<number_segments[links]; segment++)
               {
                segment_start=1;
                accumulator =  -ADDs;
                for(u = 0.0; u<1.01; u+=increment)
                 {
                  if (u<0.98)
                   {
                     accumulator += ADDs;

                polygon_vertices[number_polys[links]+1][x][0] =
                        point_image[accumulator][x][pc_curve][face];
                polygon_vertices[number_polys[links]+1][y][0] =
                        point_image[accumulator][y][pc_curve][face];
                polygon_vertices[number_polys[links]+1][z][0] =
                        point_image[accumulator][z][pc_curve][face];
                polygon_vertices[number_polys[links]+1][x][1] =
                        point_image[accumulator][x][next_pc_curve][face];
```

```
                polygon_vertices[number_polys[links]+1][y][1] =
                        point_image[accumulator][y][next_pc_curve][face];
                polygon_vertices[number_polys[links]+1][z][1] =
                        point_image[accumulator][z][next_pc_curve][face];


                    if (segment_start!=1)
                     {
                      polygon_vertices[number_polys[links]][x][1] =
                        polygon_vertices[number_polys[links]+1][x][0];

                      polygon_vertices[number_polys[links]][y][1] =
                        polygon_vertices[number_polys[links]+1][y][0];

                      polygon_vertices[number_polys[links]][z][1] =
                        polygon_vertices[number_polys[links]+1][z][0];

                      polygon_vertices[number_polys[links]][x][0] =
                        polygon_vertices[number_polys[links]+1][x][1];

                      polygon_vertices[number_polys[links]][y][0] =
                        polygon_vertices[number_polys[links]+1][y][1];

                      polygon_vertices[number_polys[links]][z][0] =
                        polygon_vertices[number_polys[links]+1][z][1];

                      polygon_vertices[number_polys[links]-1][x][2] =
                        polygon_vertices[number_polys[links]+1][x][1];

                      polygon_vertices[number_polys[links]-1][y][2] =
                        polygon_vertices[number_polys[links]+1][y][1];

                      polygon_vertices[number_polys[links]-1][z][2] =
                        polygon_vertices[number_polys[links]+1][z][1];

                     }

                    polygon_vertices[number_polys[links]+2][x][2] =
                        polygon_vertices[number_polys[links]+1][x][0];

                    polygon_vertices[number_polys[links]+2][y][2] =
                        polygon_vertices[number_polys[links]+1][y][0];

                    polygon_vertices[number_polys[links]+2][z][2] =
                        polygon_vertices[number_polys[links]+1][z][0];

                    number_poly_vertices[links][number_polys[links]+1] = 3;
                    number_poly_vertices[links][number_polys[links]+2] = 3;

                    number_polys[links] = number_polys[links] + 2;
                    segment_start = 0;
                   }
                  else
                   {

                        accumulator += ADDs;
                polygon_vertices[number_polys[links]][x][1] =
                        point_image[accumulator][x][pc_curve][face];
                polygon_vertices[number_polys[links]][y][1] =
                        point_image[accumulator][y][pc_curve][face];
                polygon_vertices[number_polys[links]][z][1] =
                         point_image[accumulator][z][pc_curve][face];
                polygon_vertices[number_polys[links]][x][0] =
                         point_image[accumulator][x][next_pc_curve][face];
                polygon_vertices[number_polys[links]][y][0] =
```

```
                      point_image[accumulator][y][next_pc_curve][face];
             polygon_vertices[number_polys[links]][z][0] =
                      point_image[accumulator][z][next_pc_curve][face];


                 polygon_vertices[number_polys[links]-1][x][2] =
                     polygon_vertices[number_polys[links]][x][0];

                 polygon_vertices[number_polys[links]-1][y][2] =
                     polygon_vertices[number_polys[links]][y][0];

                 polygon_vertices[number_polys[links]-1][z][2] =
                     polygon_vertices[number_polys[links]][z][0];

                }
              }
            }
          }

         face_color[i][0] = number_polys[links];
         face_color[i][1] = face;

       }

     *ruled_polys = number_polys[links];

     for(face=0;face<number_polygons[links];face++)
       {
         i = i + 1;

         for(pc_curve = 0; pc_curve < number_edges[links][face]; pc_curve++)
          {
           segment = 0;
           u = 0.0;

           polygon_vertices[number_polys[links]+1][x][pc_curve] =
             point_image[0][x][pc_curve][face];
           polygon_vertices[number_polys[links]+1][y][pc_curve] =
             point_image[0][y][pc_curve][face];
           polygon_vertices[number_polys[links]+1][z][pc_curve] =
             point_image[0][z][pc_curve][face];


          }

         number_poly_vertices[links][number_polys[links]+1]=number_edges[links][face];

         number_polys[links] = number_polys[links] + 1;

         face_color[i][0] = number_polys[links];
         face_color[i][1] = face;

      }

     for(face = 0;face<number_polygons[links];face++)
      {
         i = i + 1;

         for(pc_curve = 0;pc_curve<number_edges[links][face];pc_curve++)
          {
           segment = number_segments[links]-1;
           u = 1.0;
             polygon_vertices[number_polys[links]+1][x][pc_curve] =
                      point_image[Division][x][pc_curve][face];
             polygon_vertices[number_polys[links]+1][y][pc_curve] =
```

```
                        point_image[Division][y][pc_curve][face];
                polygon_vertices[number_polys[links]+1][z][pc_curve] =
                        point_image[Division][z][pc_curve][face];
             }

            number_poly_vertices[links][number_polys[links]+1] = number_edges[links][face];

            number_polys[links] = number_polys[links] + 1;

            face_color[i][0] = number_polys[links];
            face_color[i][1] = face;

         }

      /* Calculate the bounding rectangle for each polygon.  */

         for( i = 0; i <= number_polys[links]; i++)
          {
           polygon_minimum_point[x][i] = 100.0;
           polygon_maximum_point[x][i] = -100.0;
           polygon_minimum_point[y][i] = 100.0;
           polygon_maximum_point[y][i] = -100.0;
          }

         for(i = 0; i<=number_polys[links]; i++)
           for(vert = 0; vert < number_poly_vertices[links][i]; vert++)
            {
             if (polygon_vertices[i][x][vert]<polygon_minimum_point[x][i])
               polygon_minimum_point[x][i]=polygon_vertices[i][x][vert];

             if (polygon_vertices[i][x][vert]>polygon_maximum_point[x][i])
               polygon_maximum_point[x][i]=polygon_vertices[i][x][vert];

             if (polygon_vertices[i][y][vert]<polygon_minimum_point[y][i])
                polygon_minimum_point[y][i]=polygon_vertices[i][y][vert];

             if (polygon_vertices[i][y][vert]>polygon_maximum_point[y][i])
               polygon_maximum_point[y][i]=polygon_vertices[i][y][vert];

            }

     /*  Calculate the unit normal for each of the polygons. This is done by taking the cross
         product between two vectors A and B formed by the vertices of a polygon.  A check is
         included to see if the vectors are parallel.  */

         for(i = 0; i<=number_polys[links]; i++)
          {
           A[x] = polygon_vertices[i][x][1] - polygon_vertices[i][x][0];
           A[y] = polygon_vertices[i][y][1] - polygon_vertices[i][y][0];
           A[z] = polygon_vertices[i][z][1] - polygon_vertices[i][z][0];

           lines_are_parallel = 1;
           j = 1;

           while(lines_are_parallel)
            {
             j = j + 1;

             B[x] = polygon_vertices[i][x][j] - polygon_vertices[i][x][0];
             B[y] = polygon_vertices[i][y][j] - polygon_vertices[i][y][0];
             B[z] = polygon_vertices[i][z][j] - polygon_vertices[i][z][0];

             polygon_normal[i][x] = A[y]*B[z] - A[z]*B[y];
             polygon_normal[i][y] = A[z]*B[x] - A[x]*B[z];
             polygon_normal[i][z] = A[x]*B[y] - A[y]*B[x];
```

```
            if ((polygon_normal[i][x] != 0.0) ||
                (polygon_normal[i][y] != 0.0) ||
                (polygon_normal[i][z] != 0.0) ||
                (j == number_poly_vertices[links][i]-1))
              lines_are_parallel = 0;

           }

         if ((polygon_normal[i][x] != 0.0) ||
                (polygon_normal[i][y] != 0.0) ||
                (polygon_normal[i][z] != 0.0))
          {
    /* Return the unit normal vector of each polygon. */

          normal_magnitude=sqrt(polygon_normal[i][x]*polygon_normal[i][x]
                              + polygon_normal[i][y]*polygon_normal[i][y]
                              + polygon_normal[i][z]*polygon_normal[i][z]);

          polygon_normal[i][x] = polygon_normal[i][x]/normal_magnitude;
          polygon_normal[i][y] = polygon_normal[i][y]/normal_magnitude;
          polygon_normal[i][z] = polygon_normal[i][z]/normal_magnitude;

    /* Calculate the normal distance between the plane of the polygon
       and the origin (calculate D from Ax + By + Cz + D = 0). */

          plane_normal_distance[i] =
                -(polygon_normal[i][x]*polygon_vertices[i][x][0]
                + polygon_normal[i][y]*polygon_vertices[i][y][0]
                + polygon_normal[i][z]*polygon_vertices[i][z][0]);
          }

         }
}
```

::::::::::::::
shaded.cx
::::::::::::::

```c
/* This shading is using the raster-scan display by generating these picture requires
   techniques for removing hidden surfaces and for shading visible surface. The principle
   technique is the scab-line algorithm for hidden surfaces elimination. */

#include "sweepparm2.inc"
#include "scngntint.c"
#include "illumodl1.c"
#include "illumodl2.c"
#include "scnplyint.c"


    float image_array[512][410];
    short color_array[512][410];
    float values[100][3];
    static short pixels[245760];
    float hue[] = {120.0,240.0,0.0,60.0,15.0}; /* green,blue,red,,yellow,brown*/
    float chromaticity[] = {1.0,1.0,1.0,1.0,1.0};

shaded_display_two(NQ_image,links)
  float NQ_image[4][4][3][20][30];
  int links;
  {
    int i,j,k,face,width,height,count;
    int polyhedron,Ilike;
    extern float hue[],chromaticity[];
    extern char *links_name[];
    FILE *outfile,*fopen();
    FILE *lala;
    short pig; int xoffset,yoffset;
    long offset=0;
    short stin;

    for(i=0;i<512;i++)
      for(j=0;j<410;j++)
        { color_array[i][j]=0;
          image_array[i][j]=0.0; }

    for(face=0;face<number_polygons[links];face++) color[links][face]=polyhedron;

 scan_line_generate_solid_data(NQ_image,color_array,image_array,links);  /* "scngntsld.c" */

 k=0;
  {int me;
  for(j=0;j<5;j++)
    for(me=0;me<20;me++)
    {
      values[k][0]=hue[j];
      values[k][2]=chromaticity[j];
         values[me][1]=me/20.0;
            k++;
    }
  }

    x_maximum_pixel=(x_maximum_pixel+100<512) ? x_maximum_pixel+100 : 512;
    x_minimum_pixel=(x_minimum_pixel-100>1) ? x_minimum_pixel-100 : 1;
    y_maximum_pixel=(y_maximum_pixel+100<410) ? y_maximum_pixel+100 : 410;
    y_minimum_pixel=(y_minimum_pixel-100>1) ? y_minimum_pixel-100 : 1;
    k=0;
    width=x_maximum_pixel-x_minimum_pixel+1;
    height=y_maximum_pixel-y_minimum_pixel+1;

```

```
  if (links == 0) lala=fopen("I0","w");
  if (links == 1) lala=fopen("I1","w");
  if (links == 2) lala=fopen("I2","w");
  if (links == 3) lala=fopen("I3","w");
  if (links == 4) lala=fopen("I4","w");
  if (links == 5) lala=fopen("I5","w");

   for(j=y_maximum_pixel-1;j>=y_minimum_pixel-1;j--)
     for(i=x_minimum_pixel-1;i<=x_maximum_pixel-1;i++)
      {
       if (image_array[i][j]<=1.0 && image_array[i][j]> 0.001)
        {
       pixels[k] = image_array[i][j]*19;/* from 0 to 19 is the criteria */
       pixels[k] = pixels[k] + stin;
       fprintf (lala,"%d,%d=%d,",i,j,pixels[k]);
         }
       else pixels[k] = 0;
        k++;
      }
  fclose(lala);

   count=100;
  message_window("shaded ?","","y/n");
   if(*answer_window() == 'y')
    {
      HC_Set_Color_Map_By_Value("HIC",count,values);
      keynumber=HC_KInsert_Pixel_Array(0.0,0.0,0.0,width,height,pixels);
    }
   HC_Pause();
 }
```