

UCSF

UC San Francisco Electronic Theses and Dissertations

Title

KARMA, a knowledge-based system for receptor mapping

Permalink

<https://escholarship.org/uc/item/3s1097gj>

Author

Klein, Teri Ellen

Publication Date

1987

Peer reviewed|Thesis/dissertation

KARMA: A Knowledge-Based System for Receptor Mapping

by

Teri Ellen Klein

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Medical Information Sciences

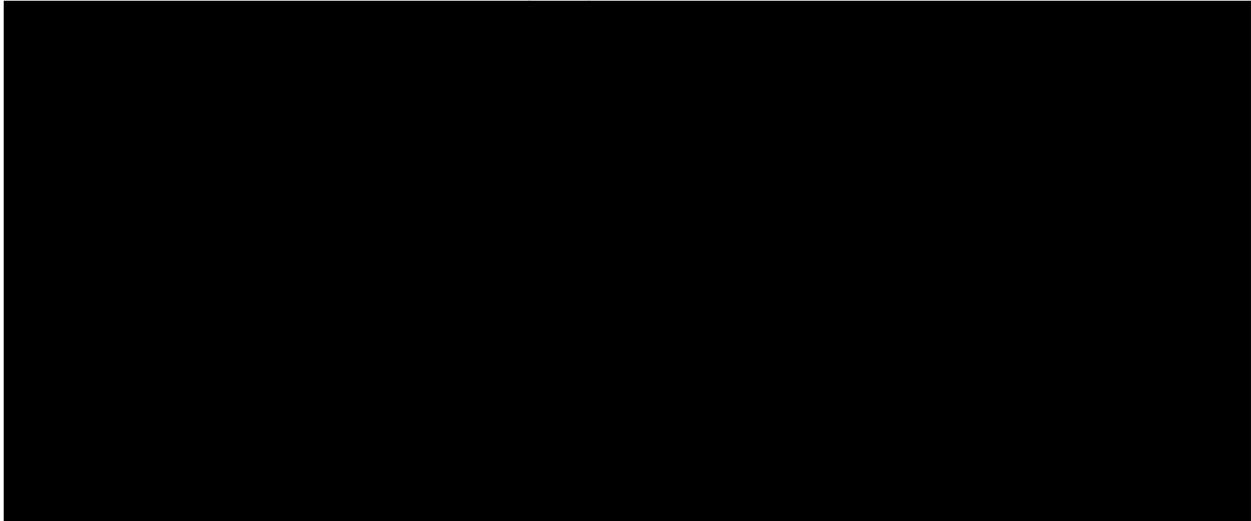
in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

San Francisco



Date

JUN 14 1987

University Librarian

Degree Conferred:

Copyright 1987
by
Teri Ellen Klein

Time flies when you are having fun.

Acknowledgements

I thank my research committee: Professors Robert Langridge, I.D. (Tack) Kuntz and Corwin Hansch for their technical advice and for serving on both my advance to candidacy and thesis committees. Specifically, Professor Langridge for our discussions on receptor mapping and how best to pursue our ideas; Professor Kuntz for sharing his knowledge of science; and, Professor Hansch for our extensive collaborations and sharing his knowledge of quantitative structure-activity relationships.

Additionally, I thank the following members of the Computer Graphics Laboratory for their contributions to this research project: Tom Ferrin for running one of the best computer systems in the field; Conrad Huang for being an invaluable collaborator; and, Conrad Huang, Eric Pettersen, and Don Kneller for their programming expertise as shown in the program listings in Appendices One and Two.

I thank the NIH Division of Research Resources (RR-1081) for providing the facilities and support, and Dennis Miller for the acronym KARMA.

To the following people, I thank you for having enhanced my graduate school days through the giving of your friendship and time:

Robert Langridge for taking a chance on me not once, but twice.

Tack Kuntz for playing devil's advocate to my scientific development.

Corwin Hansch for showing me that one person can make a difference.

Conrad Huang for being able to consume numerous chocolate chip cookies without showing any weight gain and playing basketball with my husband.

To all the past and present members of the CGL especially Don Kneller (NickD), Bruce Cohen and Eric Pettersen for providing a light hearted *working* atmosphere.

Michael Kahn for going along on the MIS ride through the comprehensive and advance to candidacy exams, and for being first in the babyrace.

Willa Crowell whose serenity and skill helped me detangle the paper jungles of the UC administration and for trying to keep me calm throughout graduate school.

Ben Klein for more reasons than just because you are my dad.

Mr. M. for learning to walk like an Egyptian.

And most importantly, Dennis Miller for being able to read my mind.

ABSTRACT

A combination of molecular modeling and quantitative structure activity relationships (QSAR) is a powerful tool in the design of ligands and inhibitors, and provides a means to better understand ligand-receptor interactions. This approach is most successful where the three-dimensional structure of the receptor site is known. KARMA (KEE Assisted Receptor Mapping Analysis) is an interactive computer-assisted rule-based drug design tool that utilizes real-time interactive three-dimensional color computer graphics with numerical computations and symbolic manipulation techniques using the expert system software tool KEE (Knowledge Engineering Environment). KARMA incorporates QSAR, conformational analysis (distance geometry and energy minimization), and graphics to generate a theoretical receptor site surface model.

Three-dimensional structures are input for the conformational analysis programs. Selected structures or distance matrices contain the geometric relations of the input structures and are used to generate the preliminary receptor surface model. An outline of the surface is obtained by the intersection of spheres, while details of the surface are generated using Gregory patches with the outline as the basis set. Gregory patches form a continuous surface which may be reshaped interactively and have local density variations. The multiple visual cues of color, texture, and intensity represent simultaneously a number of receptor properties such as shape, volume, hydrophilicity and hydrophobicity, and effectively summarize large amounts of numerical data.

The deductions made by the KEE inference engine are based on QSAR equations, physicochemical parameters, kinetic data, and structural chemistry. Upon completion of

the deduction phase, the characterized receptor model is displayed. The user may then manipulate and modify the model to test hypotheses and generate new rules. Modified models may be resubmitted to the inference engine to make additional deductions and detect inconsistencies. An integral part of KARMA is user interaction. By combining KARMA's knowledge base with the knowledge and insight of the expert user, a more refined model may be built than permitted by either the user's or KARMA's knowledge alone.

TABLE OF CONTENTS

1	Introduction	1
1.1	Historical Assumptions of Receptor Mapping	2
1.1.1	The Lock and Key Description	3
1.1.2	The Pharmacophore	4
1.2	Computerized Methods in Drug Design	7
1.2.1	General Techniques	7
1.2.1.1	QSAR	8
1.2.1.2	Geometrical Docking	9
1.2.1.3	Energy Calculations	10
1.2.2	Receptor Mapping Techniques	11
1.3	Introduction to KARMA	13
1.4	Summary	16
2	System Configuration	17
2.1	Architecture	17
2.1.1	Lisp and Lisp Machines	19
2.1.2	Three-Dimensional Graphics Machines	20
2.1.3	Workstations	22
2.2	Component Software	23
2.2.1	Introduction to KEE	24
2.2.1.1	Knowledge Bases	25
2.2.1.2	Rule Based Reasoning	25
2.2.2	Pomona MedChem Software	26
2.2.2.1	Unified Driver Program	27
2.2.2.2	Chemical Nomenclature System	27
2.2.2.3	Calculation of Partition Coefficients	28
2.3	Summary	28
3	Generation of the Surface Model	29
3.1	Structure Generation	29
3.2	Surface Generation	32
3.2.1	Initial Approach to Surface Generation	35
3.2.1.1	Generation of Polygonal Net	35
3.2.1.2	Mathematical Basis of Cardinal Patches	43
3.2.1.3	Graphical Generation of Cardinal Patches	45
3.2.1.4	Discontinuity of Cardinal Patch Surface	49
3.2.2	Surface Generation Using Gregory Patches	51
3.2.2.1	Generation of Triangular Net	52
3.2.2.2	Mathematical Basis of Gregory Patches	61
3.2.2.3	Graphical Generation of Gregory Patches	66
3.3	Summary	73

TABLE OF CONTENTS (continued)

4	Future Developments of KARMA	74
4.1	Annotation of the Surface Model	74
4.1.1	KARMA's Knowledge Bases	75
4.1.2	Rules for Characterization	78
4.1.2.1	Generic Rules	79
4.1.2.2	Specific Rules	83
4.2	Testing of KARMA	84
4.3	Current Status of KARMA for the User	87
4.4	Summary	89
	References	91
Appendix 1	Program Listings for Triangulated Surface	96
Appendix 2	Program Listings for Patches Display	125
Appendix 3	Publications Resulting from Ph.D. Research.....	178

LIST OF TABLES

Table 1:	Interactions of a Data Set	77
Table 2:	Phenolic Substrate and QSAR Equation	85
Table 3:	Data for Phenolic Substrates.....	86
Table 4:	Sample Data From User	88

LIST OF FIGURES

Figure 1:	Editing Sample	30
Figure 2:	Molecule Editor	31
Figure 3:	Prototypical Triangulated Sphere	37
Figure 4:	Two Merged Sphere	37
Figure 5:	Triangular Net for Benzylpyrimidine Compound	38
Figure 6:	Quadrilateral Net for Benzylpyrimidine Compound	38
Figure 7:	Pictorial Representation of Merging Algorithm	42
Figure 8:	Cardinal Patch Surface Model	47
Figure 9:	Cardinal Patches Before and After Movement	47
Figure 10:	Rotated Cardinal Patches Before and After Movement	48
Figure 11:	Increased Density for Display of Cardinal Patch Surface	48
Figure 12:	Discontinuity Between Cardinal Patches	50
Figure 13:	Initial Surface Based on Desire Distance	53
Figure 14:	Polygonal Net for Benzylpyrimidine Compound	53
Figure 15:	Uniform Triangular Net for Benzylpyrimidine Compound	54
Figure 16:	Two Polygons with the Same Connectivity	60
Figure 17:	Triangular Gregory Patch	62
Figure 18:	Degree Elevation for Patch Boundary	64
Figure 19:	Boundary Between Two Patches	66
Figure 20:	Gregory Patch Surface Model	68
Figure 21:	Movement of a Control Point	69
Figure 22:	Saved Position Following Movement of the Control Point	69
Figure 23:	Thatched Style Rendering for Gregory Patch Surface	71
Figure 24:	Thatched Style Rendering clipped to show Interior Surface	71
Figure 25:	Cross-Hatched Style Rendering for Gregory Patch Surface	72
Figure 26:	Cross-Hatched Style Rendering clipped to show Interior Surface	72
Figure 27:	General Knowledge Representation Scheme	76
Figure 28:	Alcohol Dehydrogenase and a Substituted Pyrazole	82
Figure 29:	Carbonic Anhydrase C and a Substituted Sulfonamide	82

Introduction

In 1876, R. Buchheim stated that “the mission of pharmacology is to establish the active substances within the [natural] drugs, to find chemical properties responsible for their action, and to prepare synthetically drugs that are more effective” [1]. A little over a century later, medicinal chemists are actively involved in an area of research, the beginnings of which were so eloquently stated by Buchheim, known as *drug design*. Unexpectedly, with the passage of time and the increase of knowledge, the field of drug design has become more complex and amazing than researchers could have imagined.

Drug design is said to “correspond to rational methodologies which give access to a new drug either through the noblest pathway, namely total innovation, or through the more tangible pathway, optimization” [2]. In theory, this implies that the ultimate goal of drug design is a new drug with optimized chemical and physiological properties that are scientifically well understood. In practice, it is extremely difficult for a researcher to *design* a new drug, particularly when there is limited biological and chemical information about a specific drug-receptor system. Even when more knowledge is available, whether it be structural information about the receptor, toxic information of drugs, or regulatory mechanisms, the system is still vastly underparameterized. This lack of precise knowledge results in the inability of the researcher to predict or design *the* novel drug for a particular system.

The data that results from a study of a set of related *drugs* has generally been observed to be insufficient to enable a researcher to design the best drug. Recently, in order to gain more information, researchers have begun to look at the biological receptor

for the drug in question. Current areas of study include structural and chemical characteristics of the receptor and descriptions of the drug-receptor interaction.

As the theory behind drug design has developed and changed over time, so have the tools that are available to the researcher. Most prominent in the modern drug design laboratory is the computer. Computers are playing an increasingly important role in the methodology of drug design, specifically in quantitative and qualitative structure activity relationships, molecular modeling, conformational analysis (including the use of quantum mechanics, molecular mechanics and molecular dynamics) and x-ray crystallography [3]. Researchers may use these methods, alone or in combination, in attempts to design better ligands and inhibitors, and to understand enzyme-ligand interactions. These methods have been found to be most successful when the three-dimensional structure of the receptor site is known [4]. As the knowledge of receptor geometry increases, it becomes easier to use computer graphics and develop a model of the receptor. The use of this model and the theory of three-dimensional complementarity enables the researcher to design more specific and potent ligands or inhibitors [5, 6]. However, research in drug design is often done in the absence of precise molecular detail of the receptor and its related interactions with inhibitors or substrates. This lack of precise information prevents the development of a sophisticated computer receptor model and thus limits the researchers' use of a valuable resource, computer graphics.

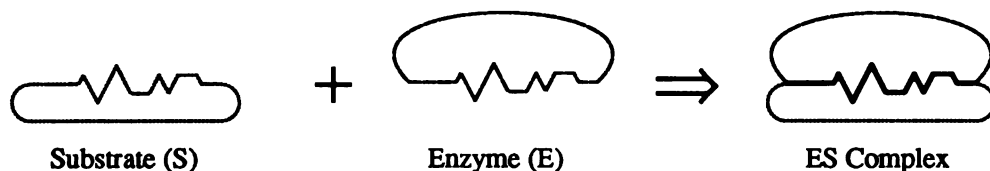
1.1. Historical Assumptions of Receptor Mapping

Around the beginning of the 20th century, E. Fischer set forth the metaphor of a *lock and key* to explain the stereospecificity of catalysis [7] and P. Ehrlich created the term *pharmacophore* based on the term chromophore from his research on dyestuffs [8].

Since their description, these concepts have shaped the approaches of medicinal chemists both in theory and in practice in their attempts to map receptors.

1.1.1. The Lock and Key Description

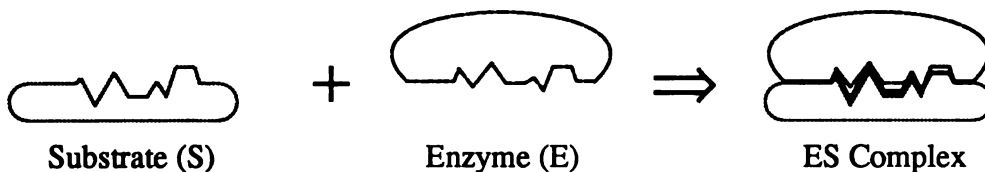
How a substrate binds to an active site of an enzyme is an actively researched field. In the early stages of research on active sites, it was thought that a substrate must have a matching shape (be complementary) to the site prior to binding; hence, the lock (active site) and key (substrate) metaphor.



This implies that both the active site and the substrate are rigid, and that the enzyme's active site becomes complementary in shape to the substrate.

Although the lock and key description emphasizes the structural geometry of both the substrate and the active site, it neglects some important aspects such as hydrophobicity between the enzyme and substrate. Additionally, in representing the substrate as a rigid object like a key, the chemistry of the substrate is not included. No attention is paid to the chemical moieties of the substrate itself.

More recently, the model formulated in the lock and key approach has been extended to include dynamic recognition at the active site. This model is called *induced-fit* [9]. The induced-fit model allows for the active site to have a shape that is complementary to the substrate after the substrate is bound.

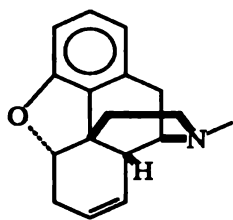


This model accounts for some flexibility in both the substrate and receptor which may be a *more* realistic representation of the drug-receptor complex than the lock and key approach. Unfortunately, rather than simplifying the receptor mapping problem, this new model makes the problem more complex due to an increase in the degrees of freedom.

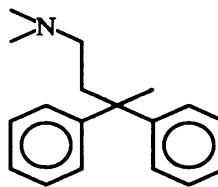
1.1.2. The Pharmacophore

A pharmacophore may be defined, for a set of molecules, as a specific pattern of elements (*e.g.*, atoms) that elicits a biological response. The fewest common structural features of all the drugs recognized by the receptor yield the pharmacophore. The geometrical arrangement of these structural features has been termed the *pharmacophoric pattern* [10]. Pharmacophoric patterns are typically dependent upon the molecular conformation and, therefore, are conducive for describing chemically dissimilar compounds having the same type of bioactivity [11].

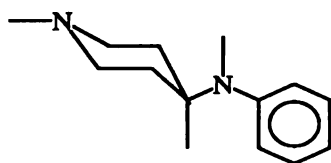
For instance, pharmacophore theory has been used extensively in the attempt to describe the structural requirements of the opiate receptor. Thousands of analgesic drugs are known to bind to these receptors. The problem is that many of these compounds have similar bioactivity, yet dissimilar structures. Among the many structural families which possess opiate activity, are: 4-phenylpiperidines (1), 3,3-diphenylpropylamines (2), and diamines (3, 4).



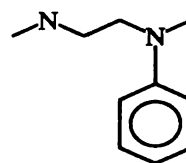
1



2

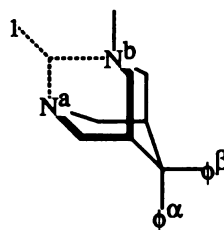


3



4

Several researchers have tried to model the opiate receptors (μ and δ opiate receptors) using a pharmacophore approach. G. Marshall and co-workers have worked many years trying to refine an opiate receptor model based on this approach [12]. They have proposed a pharmacophore model (5) based on 4-phenylpiperidines which include the drugs morphine, heroin, and codeine.



5

Their model incorporates features of both rigid and flexible derivatives of the 4-phenylpiperidine structural class.

It is of interest to note that in early publications this pharmacophore model was for *the* opiate receptor, and in later publications (~1984), the model was for the μ opiate receptor. This example brings up an important point: when you attempt to model a pharmacophore of a structurally unknown receptor, you may not be certain whether or not you are looking at a *unique* receptor. It is thus highly advisable for the researcher to be sure that the class of drugs being analyzed has biological activity at only one specific receptor. If it does not, the researcher will be limited in his ability to find an optimum drug from a set of drugs because he has two sets of drugs. Thus, he cannot model a receptor because his initial assumption involves two receptors. Determination of the pharmacophore is nothing more than a trial and error procedure in the absence (or presence) of knowledge about the receptor.

Many years of research have shown that the original description of the pharmacophore does not fully account for drug-receptor interactions. Otherwise, one might have expected the prediction of a novel analgesic with excellent therapeutic effects and minimal side effects. To account for the many drug-receptor interactions, three different types of additional pharmacophores (recognition, binding and intrinsic) have been described by R. Franke as:

“*Recognition* pharmacophore, pattern of elements which can be recognized by the receptor at the recognition point as being complementary to its own structure; *Binding* pharmacophore, (flexible) pattern of elements the presence of which is necessary and sufficient for the binding of a drug at the active site of the receptor resulting in the formation of the (initial) drug-receptor complex; and *Intrinsic* pharmacophore, pattern of elements necessary and sufficient to induce specific perturbations in the receptor subsequent to the binding step including, if

necessary, its ability to change its arrangement'' [13].

This distinction of *types* of pharmacophores is not in use in everyday research, but serves a useful purpose when one initially thinks about a specific research problem.

The idea of a pharmacophore is very appealing. If a pharmacophore could be correctly defined it might be possible to pursue accurate receptor mapping techniques and predict novel drugs.

1.2. Computerized Methods in Drug Design

Computerized methods in drug design began in the early 1960's, most notably with the correlation analysis approach developed by C. Hansch [4]. Other methods which have been developed and applied to problems in the field of drug design include structure-activity relationships [14, 15], geometrical docking [16, 17] and, molecular mechanics and molecular dynamics [18, 19]. When there is x-ray crystallographic data or other structural data available on the receptor site, computer graphics may be used in conjunction with the above methods to yield a greater understanding of complex molecular interactions. Additionally, computerized methods have been developed to map structurally unknown receptors. Presently, these methods do not take full advantage of the capabilities that interactive three-dimensional color computer graphics has to offer.

1.2.1. General Techniques

Computers have played a major role in the development of methods that investigate the relationship (quantitative or qualitative) between the molecular structures of compounds and their biological activity. Structure-activity relationships (SAR) may be derived from a variety of mathematical methods including: cluster and principal

component analysis, linear and nonlinear multivariable regression analysis, and pattern recognition and discriminant analysis [14, 15].

1.2.1.1. QSAR

The most predominant approach to quantitative structure-activity relationships (QSAR) has been the Hansch approach [20]. There are two major assumptions to any QSAR approach: (1) molecular properties related to biological activities can be separated and quantified for a set of compounds, and (2) a mathematical relationship can be established between the quantified molecular properties and the biological properties of the compounds. The Hansch approach to QSAR, based on a set of congeners, states:

$$\text{Biological Activity} = f(\text{physicochemical parameters})$$

Physicochemical parameters are used to model effects of structural changes on the steric, electronic and hydrophobic effects of molecules [21]. Using multivariable linear regression, a set of equations can be derived from the parameterized data. Statistical analysis yields the *best* equations to fit the empirical data. This mathematical model forms a basis to correlate the biological activity of the compound to its chemical structure.

When x-ray crystallographic data is available for an enzyme-ligand complex, QSAR combined with computer graphics, has been shown to be even more powerful in enabling one to understand enzyme-ligand interactions and design better ligands [4]. A recent example of how the combination of QSAR and molecular graphics has increased the researcher's understanding of enzyme-ligand interactions is the study of amidine inhibition of trypsin [22]. In this study, quantitative structure-activity relationships were formulated for four sets of amidine inhibitors. The quantitative results from these

equations were compared with qualitative molecular models of the inhibitors based on the x-ray crystal complex of benzamidine-trypsin. An important result of this study was the description of three types of solvent accessible surfaces (hydrophobic, semi-hydrophilic and hydrophilic) for the enzyme's active site. The realization and experimental basis for an intermediate type of surface (semi-hydrophilic) proved to be valuable in gaining a better understanding of the interactions between the amidine inhibitors and the solvent accessible surface of trypsin's active site.

One of the most outstanding studies that combined molecular graphics with structure-activity relationships was the molecular modeling study of the binding of thyroxine analogues to prealbumin by Blaney *et al* [23]. In this study, the molecular surface representation of the thyroid hormone-prealbumin complex provided a detailed picture which was used to predict relative binding affinities of thyroid analogues to prealbumin. The relative binding affinities were then verified experimentally by equilibrium dialysis.

1.2.1.2. Geometrical Docking

Surprisingly, a later study by Kuntz *et al* [16], employing their docking programs, found that despite the successful work of Blaney *et al* on the thyroid hormone-prealbumin complex, there were three other possible binding modes of the thyroxine derivatives that would provide good steric fit into the active site. Kuntz *et al* developed an algorithm to geometrically dock rigid ligands into structurally known macromolecular receptors based on complementarity of shape between the receptor and the ligand. This method has recently been extended by DesJarlais *et al* to include some flexibility in the ligand [17]. By including some degree of flexibility of the thyroxine molecule, five

possible binding modes differing from the original Blaney model were found.

The thyroid hormone-prealbumin example illustrates the desire to have a check of some sort to prevent the occurrence of the natural phenomenon of *bias* in the building of a molecular model. It is nearly impossible for a researcher to look at all possible solutions to an open ended problem, and thus, the researcher is apt to stop looking for solutions when he has found a solution that looks *best* at that moment. However, this best solution may only be so from a subjective viewpoint of the researcher. As demonstrated by the above example, it is desirable and profitable to have an objective way to generate initial models.

1.2.1.3. Energy Calculations

A distinct advantage to the geometrical docking programs is their ability to generate many different possible binding modes for ligands. Analysis of these suggested binding modes for chemical sense must be done. This may occur by visual inspection by the researcher and/or comparisons of an energy function for the ligand-enzyme complex. Although there is disagreement amongst researchers as to whether a ligand binds to a receptor in its lowest energy conformation, energy minimization is a useful tool for not only generating geometrically correct starting structures, but also for providing a means of comparing different binding modes. For the researcher who does not have access to a large computing facility that runs a molecular mechanics program such as AMBER [18] for automatic energy minimization of the ligand-enzyme complex, less computationally demanding programs have been developed for the medicinal chemist interested in docking a ligand into a receptor with the incorporation of an energy function [24, 25]. For example, the programs GRID and GRUB developed by P. Goodford for calculating

and subsequent display of a potential energy grid of a structurally known receptor are less demanding of computing facilities because these programs do not energy minimize the entire protein-ligand complex [25]. These programs can aid the researcher in modeling more likely bound conformations of a ligand.

If x-ray crystallographic data on a receptor are not available, energy calculations may still be possible but in a more limited fashion. The energy programs may then be used for refining and comparing ligands for conformational analysis and in conjunction with receptor mapping techniques [26, 13]. If x-ray crystallographic data on the ligand-receptor complex are available, the method using thermodynamic perturbation theory implemented with molecular dynamics to calculate relative changes in the free energy of binding of ligand-enzyme complexes will prove to be valuable [19]. This method was recently applied to the study of the enzyme thermolysin complexed to phosphonamide and phosphonate ester inhibitors [27]. In this study, the inhibitors differed by two atoms, but the binding constants differed by three orders of magnitude. The theoretical calculations were able to reproduce the experimentally measured values within experimental error. This method should prove to be extremely valuable, even in a predictive way, for the medicinal chemist studying a congeneric set of ligands when good x-ray crystallographic data is available for the ligand-enzyme complex.

1.2.2. Receptor Mapping Techniques

Of the techniques mentioned above, only the SAR techniques can be fully utilized in the absence of any x-ray crystallographic data. This limitation is not only because of the inability to utilize computer graphics, but because there is little or no knowledge available of the receptor. Therefore, researchers are limited in the methods that are

available to map receptors.

Several researchers have attempted to map receptors in the absence of structural information, employing some form of static representation of the site. A steric representation of dissimilar ligands having similar polar groups has been developed by Simon *et al* [28]. The calculation of the shape of the receptor is based on binding data. However, Simon's representation contains no information or inferences regarding hydrophobic or hydrophilic nature of the site, but rather only steric accessibility.

Another method, known as the Active Analog Approach (AAA), centers around a static representation of steric properties of small molecules to yield a receptor map. This approach, developed by G. Marshall, is reported to be valuable in determining what volume is available for the small molecule (common volume) versus that volume occupied by the receptor (essential volume) [29]. In this approach, analogues are divided into active and inactive categories. Pseudoelectron-density maps of each active analogue are calculated and the union of these maps yields the total active volume. Pseudoelectron-density maps of each inactive analogue are calculated and individually subtracted from the total active volume. Intersection of the resulting volumes yields the enzyme-essential volume. This volume description defines the limits of allowable modifications to the small molecule for either improving the activity of an analogue or for describing a more specific probe for refinement of the receptor map.

In a differing approach employing distance geometry, G. Crippen has derived a description of a receptor site as an array of points that are representative of various chemical properties that are attractive or repulsive to any ligand molecule [30]. Conformational flexibility of the ligands is considered with this approach which

minimizes the importance of needing a x-ray crystal structure of the ligand since the conformations derived from the distance geometry programs are not based on a specific crystal conformation. Bultsma *et al* have recently refined this method by reducing the number of required energy parameters and incorporating lipophilicity as a hydrophobic bonding parameter [31]. R. Sheridan *et al* have extended the distance geometry programs to treat one or more molecules as a single *ensemble* yielding a pharmacophore model [32]. The pharmacophore model can then be used as a reference “structure” when comparing the common volume of agonists. This approach however, does have one severe drawback: the researcher must be able to select the pattern of elements that determine the pharmacophore for each model (*i.e.*, the limitation for *all* pharmacophore models).

Lastly, Andrews *et al* have developed a receptor model based on potential energy calculations and the activities of semi-rigid analogues [33]. This model is limited to the structural and conformational requirements for the binding to the receptor.

The methods briefly described above have two basic similarities: the representation of the receptor site is static, whether it be a set of points or an enclosed volume, and the representation is derived from steric interactions. It would be most beneficial for researchers in the medicinal chemistry field to have a tool that could create an annotated visualization of a receptor site in the absence of specific three-dimensional information regarding the receptor.

1.3. Introduction to KARMA

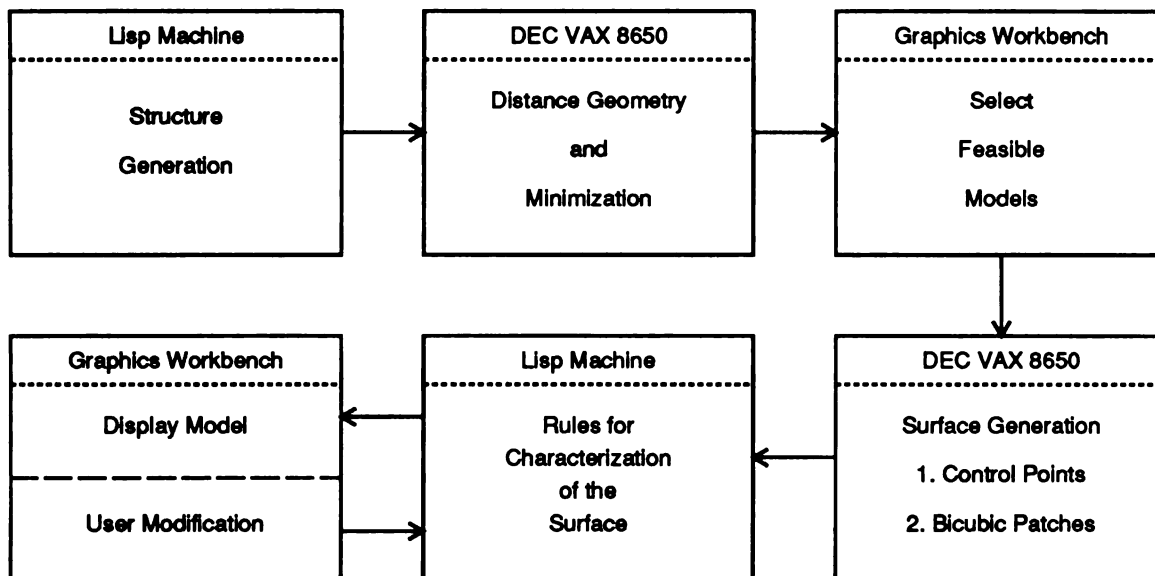
The creation of an annotated surface model of a receptor site would provide the researcher with a dynamic pictorial representation of known characteristics for that

receptor. The combination of real-time computer graphics and chemical knowledge would allow the researcher to manipulate the surface model in the testing of various hypotheses in an interactive setting. KARMA (KEE Assisted Receptor Mapping Analysis) will be a tool to meet this need.

Specifically, KARMA is a rule-based tool that uses real-time interactive three-dimensional color computer graphics, numerical computation and symbolic manipulation techniques to generate a theoretical receptor surface model. Parametric bicubic patches are used to generate the surface model. This method of creating a graphical representation of the surface provides continuity of position, tangent and curvature. In addition, the bicubic patch method allows for local manipulation without recalculating the entire surface. The integrated system uses knowledge sources such as structure-activity relations, conformational analysis and structural chemistry, allowing the researcher to characterize and refine the receptor surface model. The software package KEE¹ (Knowledge Engineering Environment) provides the supporting structure which includes knowledge base storage and access routines, an inference engine to be used by the rule system, and debugging and explanation facilities. KEE has been employed during the prototyping phase of KARMA.

From the user's perspective, the system architecture of KARMA is described on the following page:

¹KEE is a registered trademark of IntelliCorp.



Three-dimensional structures are input data for the conformational analysis programs. Selected structures or distance matrices (from distance geometry) contain the geometric relations of the input structures and are used to generate the preliminary receptor surface model. An outline of the surface is obtained by the intersection of spheres, while details of the surface are generated using parametric bicubic patches with the outline as the basis set. This initial surface model is then characterized iteratively by KARMA's rule system.

The deductions made by the KEE inference engine are based on QSAR equations, physicochemical parameters, and structural chemistry. Upon completion of the deduction phase, the characterized receptor model is displayed. Parametric patches form a continuous surface which may be reshaped interactively and have local density variations. Multiple visual cues of color, texture, and intensity represent simultaneously a number of receptor properties such as shape (i.e., cleft or hole), volume,

hydrophilicity and hydrophobicity, and pictorially summarize large amounts of numerical data. The user may then manipulate and modify the model to test hypotheses and generate new rules. User-altered models may be resubmitted to the inference engine to make additional deductions and detect inconsistencies.

An integral part of KARMA is user interaction. By combining KARMA's knowledge base with the knowledge and intuition of the user, a more refined model may be built by taking advantage of the attributes of both man and machine.

1.4. Summary

Many researchers are attempting to address the need for better methods for rational drug design. The methods currently available for generating receptor models are limited by human bias, static representation of the receptor site and a representation of the receptor derived mainly from steric interactions. KARMA was designed to address these limitations. Specifically, the critical feature of KARMA was the development and implementation of a *dynamic* and *continuous* surface that can be manipulated interactively by the user on a three-dimensional graphics display and in the future, by the rule system currently under development.

System Configuration

KARMA combines symbolic manipulation, numerical computation and interactive three-dimensional color computer graphics, and a project developer interested in these techniques can choose from a diverse set of hardware and software. A high bandwidth network enables different system architectures, such as Lisp machines and high performance three-dimensional graphics machines to be used with the same software package. It is critical to a software project such as KARMA that the design be modular so that space-time performance can be optimized. Modularization allows optimization of both hardware and software (*e.g.*, operating system, development environment, satellite programs, *etc.*).

2.1. Architecture

An *architecture* for a computer system is a specification of an interface [34]. There are many levels and types of architectures in a computer system. D. Moon recently described three general types of architecture (system, instructional, and processor) as:

“*System* architecture, how the system appears to end users and application programmers including characteristics of language, user interface and operating systems; *Instructional* architecture, the instruction set of the machine, the types of data that can be manipulated by those instructions and the environment in which the instructions operate; and, *Processor* architecture, the overall structure of the implementation of the instruction architecture” [34].

Processor architecture may be viewed not only as the interface between firmware and hardware, but also as an interface between the ports of the processor hardware.

Instructional architecture may be viewed as the interface between compilers and hardware. Although these two architectural types are important, the users and developers of a system such as KARMA are mainly interested in system architecture. System architecture provides specification and implementation in software that users and developers employ. The limiting factor in system architecture is the available hardware which sets the bounds on what is possible.

KARMA uses symbolic processing which is a characteristic of Artificial Intelligence (AI) applications. AI software applications are noted for the demands they make on processor power, main and virtual memory size and disk capacity [34]. These demands arise partly from a lack of knowledge and incomplete understanding of a particular problem. AI algorithms are *nondeterministic* since it is difficult to plan what procedures must be executed and terminated based upon a current state of information [35]. In contrast, numerical algorithms, such as the surface calculation in KARMA, are *deterministic*. Bounds on computational performance for deterministic algorithms can be established. The single processor von Neumann machine (*i.e.*, DEC VAX² 8650) excels at sequential and deterministic numerical computations but is not as efficient for nondeterministic symbolic manipulation [36]. AI applications tend to have intensive and irregular memory access patterns which may cause processor/memory bottlenecks where there is centralized control by the hardware architecture [37]. It is difficult to integrate both numerical and symbolic processing into a single computer system because of the conflicts in processing requirements. It is desirable to design a software system in a

²DEC and VAX are registered trademarks of Digital Equipment Corporation.

distributed environment to take advantage of different types of hardware for optimization.

2.1.1. Lisp and Lisp Machines

LISP (LISt Processing), a flexible computer language designed to symbolically represent objects in the world and the relationships that exist among them, is the second oldest structured programming language after Fortran. Developed by J. McCarthy in 1960 [38], Lisp is based on a small number of constructs. Dialects of Lisp include MacLisp, InterLisp, QLisp and Franz Lisp. Common Lisp, a standard for Lisp, has recently been developed [39]. Common Lisp is lexically scoped and is an extension of the MacLisp dialect.

Lisp is an interpretative language so programs written in Lisp do not need to be recompiled for testing purposes following modification. Additionally, Lisp is an untyped language which leaves the identification of data-types to the computer. Data-type checking must be done at compile and run-times. For conventional hardware such as the DEC VAX 8650, data-type checking must be performed in software. This type of checking is not very efficient [36]. Specialized hardware to run Lisp has been developed and manufactured to decrease the total processing and run-time of Lisp as compared to a conventional computer.

Basically, Lisp machines are personal computers programmed in Lisp. System software tools written in Lisp include the operating system, user utility programs, compilers and interpreters. Lisp machines have tagged architecture. The Symbolics

3600³ Series Lisp Machines implement this tagged architecture by parallelism in the processor by concurrently executing the following operations: run-time data-type checking, result tagging, instruction fetch, instruction decode and execution, and garbage collection support [40]. Run time data-type checking invalidates operations before they occur, insuring program reliability and data integrity.

Lisp machines typically have a high resolution bit-map display for graphics and text, virtual memory address space (1 Gbyte for the Symbolics 3600 Series Lisp Machines) and networking capability. The interactive programming environment on the Symbolics Lisp Machines includes a flavor based window system, menus, a mouse, a real-time text editor with interpretation and compilation of source code within the editor, incremental compilers, and dynamic loading and linking [40]. These features provide a programming environment that encourages rapid prototyping and development of large scale AI software projects.

2.1.2. Three-Dimensional Graphics Machines

Computer graphics can be described as the creation and manipulation of pictures with the aid of a computer [41]. The Chinese proverb *a picture is worth ten thousand words* [42] is an understatement when used to describe computer graphics. Interactive real-time computer graphics allows for easy communication between the user and the computer. The use of high performance computer graphics hardware provides the much needed specialized matrix hardware to perform transformations such as rotation, translation, scale, clipping and viewing. This hardware support is desirable for large

³Symbolics and Symbolics 3600 are registered trademarks of Symbolics, Inc.

scale interactive real-time displays since transformations performed in hardware are much faster than those same transformations performed in software. Also, these hardware transformations which provide depth cueing along with perspective yield a sense of three-dimensionality to the representation. With interactive systems, users and developers have a variety of input devices including data tablets, control dials, function switches, alphanumeric keyboard, joy sticks and/or a mouse in which to communicate with the computer.

A graphics display system typically consists of a display processing unit (DPU), a display device such as a cathode ray tube (CRT), and a computer. The DPU can create a drawing either by *raster scan* or *random scan* [43]. A raster scan display system is based on television technology. The graphics primitives are stored in a buffer in terms of their components called *pixels*. The image is formed from a set of horizontal interlaced lines made up of individual pixels. A random scan display system (also known as a vector or a calligraphic display system) draws lines from one point to another as directed by a computer produced display list.

Although raster graphics systems may be capable of displaying a more *realistic* image than that of a vector graphics system, there are clear tradeoffs in cost and speed of interaction. These tradeoffs become increasingly obvious with an interactive real-time system for the display of large objects. Real-time display of and interaction with shaded raster graphics displays are just now becoming available in a cost effective way. However, the manipulation of large scale display objects in real-time motion is still limited. Also, due to the discrete nature of pixel representation, non-solid images (*e.g.*, lines) may be displayed with *jaggies* if additional mathematical calculations (anti-

aliasing) are not performed on a raster scan system. The vectors graphics machine is currently better able to manipulate complex models in real-time since it does not have to waste time filling in the blank space corresponding to the background information.

Historically, it was necessary to have a satellite computer (*i.e.*, DEC VAX 750) drive the graphics display system. Recently, three-dimensional graphics workstations (*i.e.*, Silicon Graphics IRIS) have become available which contain both the CPU and DPU.

2.1.3. Workstations

A software package such as KARMA, which combines a variety of computational techniques, currently cannot run efficiently on one machine. Presently, there are no workstations available that combine numerical computation, symbolic manipulation and high performance interactive three-dimensional color computer graphics. However, workstations that perform two out of three of these techniques are available. It is not unrealistic to expect that in the near future a workstation will effectively perform all these techniques.

Workstations can be as responsive and predictable as a timeshared computer with respect to the processor and main memory [44]. If competition for disk access is the limiting factor for a software application (*i.e.*, AI application), then a workstation which can provide individualized disk caching may alleviate this limitation. Disk caching at each workstation provides both a larger effective disk cache and a higher cache rate.

Workstations connected by a local area network can obtain benefits of both timesharing systems and workstations. The benefits of a timesharing system include shared access to files (data and programs), I/O devices, communication among users and network resources. The benefits of workstations for the single user include fast response,

dedicated processor and memory resources, availability and a personalized environment.

As previously stated, a distributed system is the best design approach for KARMA. This provides for optimization of available hardware for graphical representation, symbolic representation and manipulation, and numerical computation while working in a productive developmental environment.

2.2. Component Software

KARMA is a knowledge based system consisting of two major components: the knowledge base and the inference procedures. The knowledge base contains facts and rules, and the inference procedures consist of processes that search the knowledge base to infer hypotheses and solutions to problems [36]. Representation of the knowledge involves encoding of information about objects, relations, goals, actions and processes into data structures and procedures [37]. The knowledge may be uncertain, fuzzy, or heuristic in nature. Because of the lack of consistent and complete knowledge at the representation level, there is a continual need to modify the existing knowledge base and to maintain its consistency as new knowledge is acquired. Typically, a truth maintenance system is part of a knowledge based system. Truth maintenance consists of recognizing an inconsistency, modifying the state to remove the inconsistency, and verifying that all inconsistencies are detected and corrected in the knowledge base [36].

Two software packages are utilized to develop and maintain the knowledge base and inference procedures for KARMA. KEE (Knowledge Engineering Environment) is a

software development system from IntelliCorp⁴, which provides the necessary framework of support tools for KARMA. Pomona MedChem Software is a chemical information system from Pomona College⁵ which provides the basis for structure input and unique identification, and physiochemical parameters (*e.g.*, calculated log *P*).

2.2.1. Introduction to KEE

KEE (Knowledge Engineering Environment) is a software package that is designed to aid a researcher in building knowledge-based systems about a specific domain [45]. The set of software tools that KEE provides allows the researcher to develop the application instead of implementing known and available support tools. KEE integrates frames for representing static knowledge, object-oriented programming, a rule system for rule-based reasoning and graphics capabilities for the user interface. The frame is a “data” structure that describes an object or a class of objects. Descriptive and procedural attributes of an object are associated with a particular object in a frame.

KEE is the underlying structure for KARMA. Specifically, it provides tools for knowledge representation, storage and manipulation, and reasoning. These tools allow the user of KARMA to test a variety of hypotheses concerning the receptor site. Multiple hypothetical models may be maintained in a *KEEworld*. The *KEEworld* system is constructed on an assumption-based truth maintenance system. A *world* in KEE is a model of a situation and comprises a set of assertions about that situation. Underlying all worlds that are created is the *background*, which is defined as information in a

⁴IntelliCorp is a registered trademark of IntelliCorp. IntelliCorp, 1975 El Camino West, Mountain View, California, 94040-2216.

⁵Medicinal Chemistry Project, Pomona College, Claremont, California, 91711.

knowledge base without a specific context. Rules may act upon either the *background* or *world(s)*.

2.2.1.1. Knowledge Bases

KEE provides support for access, modification and manipulation of the knowledge in the building and storing of a knowledge base. Objects and their attributes (procedural or descriptive) are represented as a frame.

A knowledge base in KEE consists of units, slots, slot values, facets and facet values. There are two types of *units* representing objects in KEE: a *class* unit describes sets of objects and a *member* unit describes a particular instance of a class. *Slots* represent attributes of an object and *slot values* represent the value of an object's attribute. A Lisp function may be situated in the knowledge base as a slot value. The actions that are defined in this Lisp function are part of the knowledge base representational structure, thereby associating actions with units. *Facets* and *facet values* describe slots in the same way slots describe units.

Frame-based representation in the knowledge base is hierarchal, providing a parent-child type inheritance between higher level units and lower level units. This inheritance of slots (attributes) results in an economy of data entry and consistency in the knowledge base.

2.2.1.2. Rule Based Reasoning

KEE uses rules to reason. These rules may be heuristic in nature (*rules of thumb*) or may consist of definite knowledge about a collection of facts. The rules also provide a pathway for explanation to the user either through direct inquiry or graphical display.

KEE's rule system is a production rule system that can make deductions or take actions. The rules are in the form of a condition; (*if*) . . . , and a conclusion (*then*). . .

The rule system uses variables and Lisp forms in rules. Symbolic reasoning may be done with forward or backward chaining. In backward chaining, a conclusion is hypothesized and the rule system looks for conditions of rules to determine which are satisfied and which one should be hypothesized as a new condition for the rule interpreter to prove. In forward chaining, conditions are stated which are assumed to be true and then are tested to see if a conclusion can be reached.

KEE's rule system contains *action* rules and *deductive* rules. There are two types of action rules: same world action rules and new world action rules. The action rules assert or retract facts in either a world or the background, and may run Lisp code. Only new world action rules can create new worlds and make changes in them. Deduction rules deduce new facts based on facts that already exist in the world(s).

2.2.2. Pomona MedChem Software

Pomona MedChem Software is an integrated chemical information and modeling software package [46]. The package contains several modules to perform calculations of physiochemical properties such as $\log P_{\text{octanol/water}}$ (CLOGP3) and molar refractivity (CMR), two-dimensional drawings of molecules (DEPICT), a chemical nomenclature system (SMILES), a thesaurus-oriented chemical database (THOR) and a generalized substructure search program (GENIE). GENIE contains both a substructure search algorithm and an open-ended language for specification of search targets from the user's perspective. The Pomona MedChem Software includes a unified driver program (UDRIVE) to access all capabilities of the software.

2.2.2.1. Unified Driver Program

UDRIVE is a screen-oriented program. KARMA specifically utilizes UDRIVE's functions for interactive structure input via an alphanumeric keyboard, structure input/output from files of various formats, and graphical depiction and verification of structures. It is expected that KARMA's interface to UDRIVE will also have access to the physiochemical property calculations and have access to measured properties via the STARLIST database available from the Medicinal Chemistry Project and Pomona College.

2.2.2.2. Chemical Nomenclature System

SMILES (Simplified Molecular Input Line Entry System) is a chemical nomenclature system that provides the user with an easy way to enter, verify, modify and store chemical structures. SMILES uses standard atomic symbols and a unique SMILES code can be machine generated using the program CANGEN.

The SMILES language provides a simple specification of organic, inorganic, charged, disconnected and unusual valence compounds. Aromaticity is accurately detected for both charged and neutral structures with uniform enforcement of Huckel's rule. The SMILES specification defines the hydrogen attachments and bond types with a unique description of the hydrogen suppressed graph of a structure. The language also provides for further specification of isomerisms, polymerism, disconnections, ionization and complexation.

The nomenclature rules for SMILES are simple. The rules govern atomic specification, attached hydrogens and formal charge specification, bond specification, ring closures and disconnected structures. These rules form a consistent and flexible

language for the specification of chemical structures.

For KARMA, it is essential to uniquely identify chemical structures for structure searches and to minimize duplication of structural information. CANGEN fulfills this need by generating unique SMILES representations of chemical structures and fragments. CANGEN canonicalizes the structure and generates the unique SMILES code for the canonical structure.

2.2.2.3. Calculation of Partition Coefficients

CLOGP3 is a program for estimating octanol/water partition coefficients. The algorithm looks at general hydrophobicity and fragment hydrophilicity as well as shielding, electronic and proximity effects. The Pomona MedChem Software provides an algorithm manager for CLOGP3 for user modification of the fragment database. Users are able to improve their $\log P_{\text{octanol/water}}$ computations for chemical classes by providing their own experimental data.

2.3. Summary

The presence of several different types of computing in KARMA requires the use of a variety of hardware and software. A Lisp machine is used for efficiency for the development and execution of the rule system. A three-dimensional graphics display is used to obtain the necessary performance for display and manipulation of the software. Satellite software such as KEE provides a rich and powerful development environment whereas SMILES provides an efficient method for identifying chemical structures.

Generation of the Surface Model

The surface model for KARMA is generated using a parametric patch representation based on the three-dimensional structures of the molecular compounds of interest. These compounds may either be a *congeneric* series or *noncongeneric* series of compounds. KARMA uses a program based on the Pomona MedChem Software SMILES⁶ (Simplified Molecular Input Line Editor) for structural input. SMILES creates a unique identifying code for each chemical structure which is used for structural searching and minimizing duplication of structural information (see section 2.2.2). These structures are passed to satellite programs including distance geometry⁷ to generate multiple conformations that are then displayed so that the user may select those structures of interest. Those structures selected by the user are used to define the receptor model and to generate a parametric patch surface which is *smooth* and *continuous*.

3.1. Structure Generation

Input of the structure to KARMA is achieved through a series of “pop-up” menus in the Karma Window (see Figure 1a).

⁶SMILES was provided by Dr. David Weininger and Dr. Albert Leo at the Pomona MedChem Project, Department of Chemistry, Pomona College.

⁷The Distance Geometry program was provided by Dr. Gordon Crippen from the University of Michigan and Dr. Jeffrey M. Blaney from E.I. DuPont de Nemours & Company.

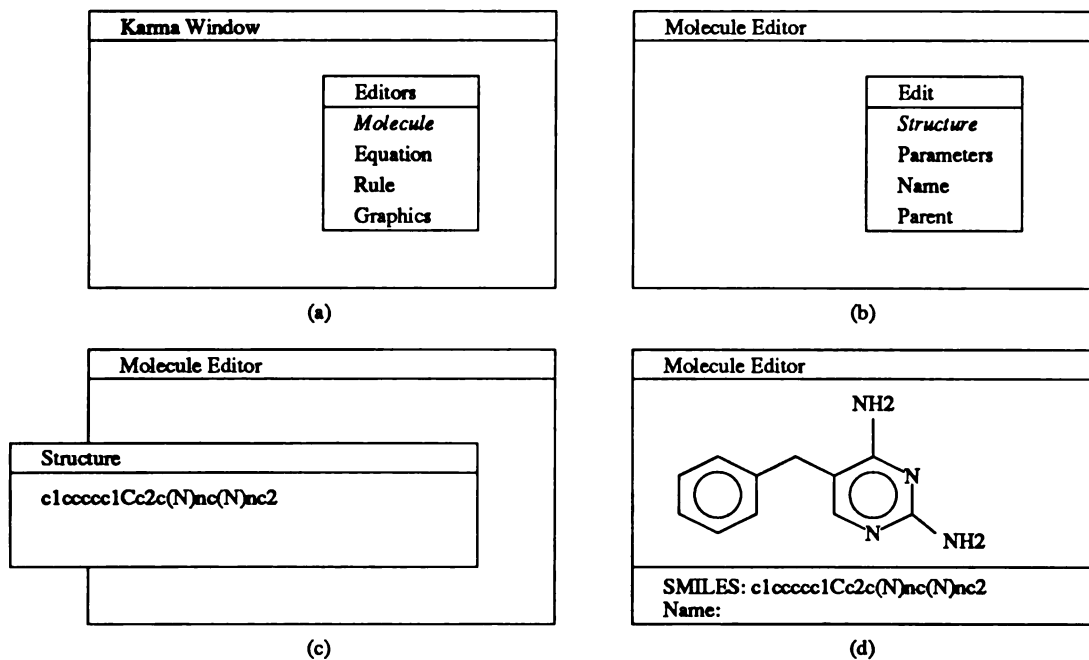


Figure 1. Editing Sample

For example, if the user is interested in entering a set of congeners, the user would select the *molecule* editor. KARMA will then display the molecule editor layout in the current window. Users can then enter the chemical structures selecting *structure* from the molecule editor menu (see Figure 1b). Structures are currently entered using the tree structure of SMILES (see Figure 1c). (The molecule editor will be expanded to allow for graphical input in the future.) KARMA then displays the two-dimensional structure for user verification (see Figure 1d). Coordinates for the three-dimensional structures are saved in a knowledge base in KEE. The three-dimensional structures are based on x-ray crystallographic data, standard bond angles, and bond lengths. Currently, the x-ray crystallographic data is obtained from the Cambridge Crystallographic Data Base using

their software [47]. The Cambridge Crystallographic Data Base resides on the DEC VAX 8650. All congener data, including physiochemical parameters such as $\log P$ or MR (calculated or experimental), can easily be entered and revised in the molecule editor (see Figure 2).

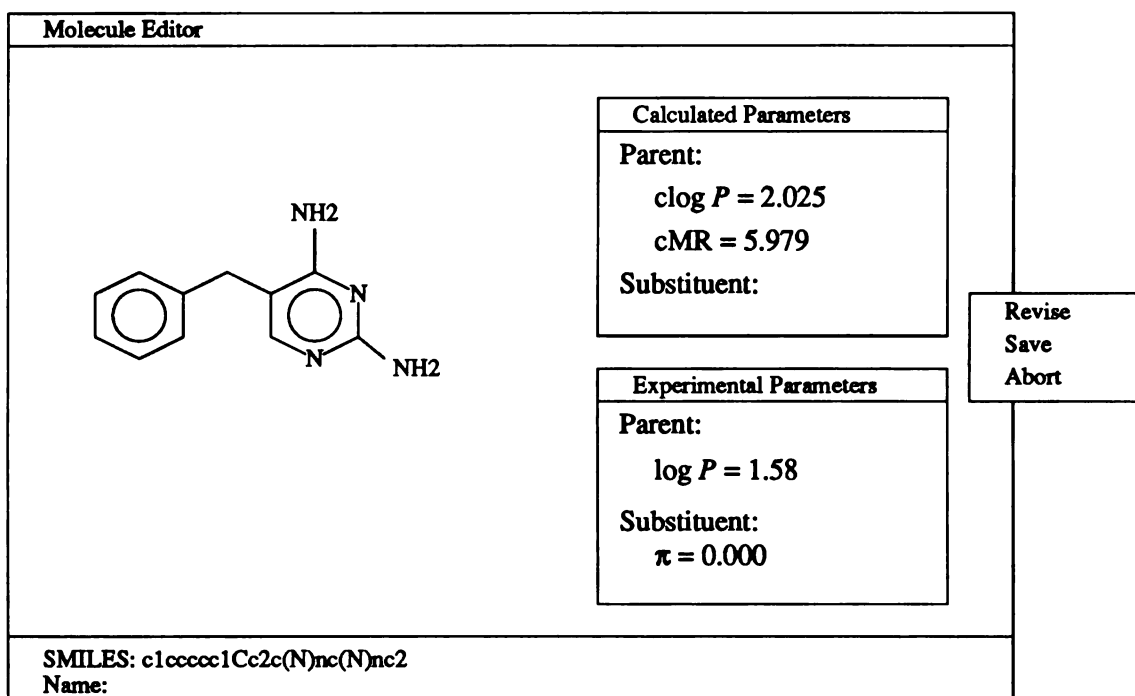


Figure 2. Molecule Editor

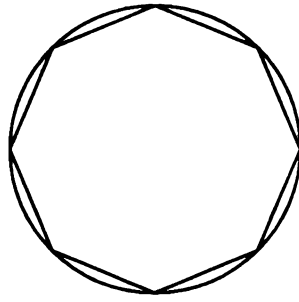
Three-dimensional coordinates for the congener set are passed to the distance geometry program. This program provides an efficient method for searching conformational space. Distance geometry program includes subroutines for controlling ring planarity of aromatic rings and orientation of the molecules based on a common group of atoms.

The output of the distance geometry is passed to the graphics program. The structures are displayed three-dimensionally so users may select structures to represent conformational space. Models are easily selected by pointing at the desired structure. X, Y, and Z rotations and translations, depth cueing, color, and labeling have been incorporated. The display program as well as the distance geometry program provides a RMS matching routine for N arbitrary atoms designated by the user. The selected models are then used for surface generation.

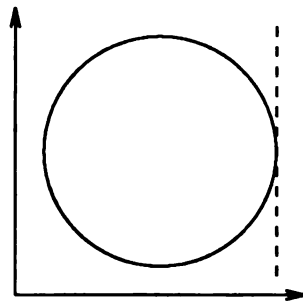
3.2. Surface Generation

The use of surfaces in molecular modeling and drug design applications is well documented [4, 23]. The two most common surfaces for these applications are the solvent accessible [48] and the Van der Waal's [49] representations. Although these surfaces have many fine attributes, arbitrary manipulations of the surface shape are not included in the list. It is necessary for KARMA to generate a surface for arbitrary topology that is both dynamic and interactive in real-time so that feedback is provided to the user. Multiple visual cues such as color and density are also used to provide the user with additional feedback. Because the surface is manipulable, global and local manipulations of the surface must be provided in an efficient manner.

The surface model can be described as a continuous surface for arbitrary topology. The algorithm that generates the surface must be able to generate a closed surface about a convex volume. Required properties include smoothness of the surface and, continuity of position and tangent to the surface. Smooth curves may be obtained via interpolation of points and hence, surfaces in three-dimensional space, as opposed to a polygonal representation as seen on the following page:



Continuity of position and tangent can be obtained by using a parametric representation as opposed to a non-parametric representation [50]. Surfaces enclosing volumes will have vertical tangent lines or planes with respect to any coordinate system. This is illustrated in two-dimensions below:



Surface generation is based on a set of points, P , defined as:

$$P = \bigcup_i P_i - \bigcup_{i,j} I_{ij}$$

where P_i is the distributed set of points over a sphere corresponding to atom i , and, I_{ij} is the set of points in P_i which fall inside atom j . The density of points on spheres can be arbitrarily set by the user. If the density is relatively high, a *large* number of patches with small area are generated; to address each patch at a high density would be time-consuming and difficult at best. If the density of points is low, the patches become too large and do not yield enough detailed information about the surface model.

KARMA presents the results of the surface model from the rule system on a three-dimensional graphics machine. Patches of the surface model are displayed graphically and may be manipulated by the user. The user may also modify the model and return to the control server for another iteration in the rule system if the results are not satisfactory.

The patches are characterized with different colors, intensities and line textures to show attributes such as hydrophobicity and steric properties. Attributes may be displayed, with color and intensity representing the value of an attribute, and line texture representing KARMA's confidence level in the information. For example, when displaying hydrophobicity, red patches are hydrophobic space while blue patches are hydrophilic space. Patches drawn with solid lines represent areas which are well explored while patches with short dashes contain little information. Displaying information using multiple cues allows the user to examine various aspects of the surface model without having to deal with large amounts of numerical data.

The graphics interface is also the appropriate place to alter the model since it lets the user look at an overall picture of the model as it is modified. The graphics interface provides user-friendly tools for this purpose, including a pointing device for selecting the modification site and a hierarchical menu system to guide the user through the actual process of making changes. Thus, the user may select a control point on one of the patches with the pointing device; pop up a menu of permitted modifications; select an operation, *e.g.*, move the control point outwards along the surface normal (see section 3.2.2.3). After the control point data has been modified, the graphics interface will recalculate and redraw the patches of the surface model based on the new data. After modifying the model to the desired state, the user may simply return to the control server

and initiate the rule system for further refinement.

3.2.1. Initial Approach to Surface Generation

Parametric bicubic patches were initially chosen as the mathematical basis for the surface model for KARMA. Advantages of the parametric bicubic surface include continuity of position, slope, and curvature at the points where two patches meet. All points on a bicubic surface are defined by cubic equations of two parameters s and t , where s and t vary from 0 to 1. The equation for $x(s, t)$ is:

$$\begin{aligned}x(s, t) = & a_{11}s^3t^3 + a_{12}s^2t^2 + a_{13}s^3t + a_{14}s^3 \\ & + a_{21}s^2t^3 + a_{22}s^2t^2 + a_{23}s^2t + a_{24}s^2 \\ & + a_{31}st^3 + a_{32}st^2 + a_{33}st + a_{34}s \\ & + a_{41}t^3 + a_{42}t^2 + a_{43}t + a_{44}\end{aligned}$$

Equations for y and z are similar [43]. Overlapping sets of control points allow for the joining of patches. Sixteen points define a bicubic patch. To determine which points define which patches, a polygonal triangular net is initially calculated. The internal edge of two triangles is dropped to form a quadrilateral. Each internal edge is used only once. Nine quadrilaterals define a single patch. Cardinal Patches were chosen for the initial approach to surface generation because these patches are interpolated through the derived control points.

3.2.1.1. Generation of Polygonal Net

In general the first step for computing the surface for a set of molecules is to approximate a surface with a set of adjacent triangles that encloses the volume of the molecules. Initially, each atom in the molecule is replaced by a prototypical triangulated

sphere (see Figure 3). The spheres are then merged together one at a time (see Figure 4); intersections of spheres are deleted, leaving an outline of the volume to be enclosed by the surface. The initial surface is a triangular net (see Figure 5) which is then converted to a quadrilateral net by dropping the internal edge between two triangles (see Figure 6).

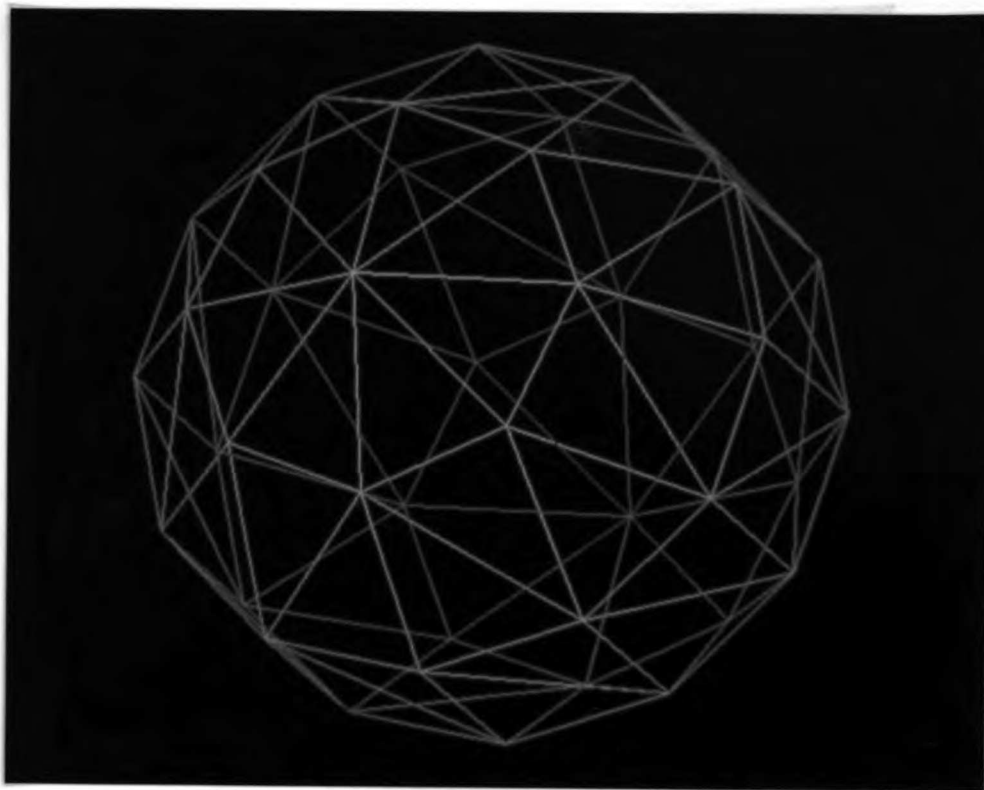


Figure 3. Prototypical Triangulated Sphere

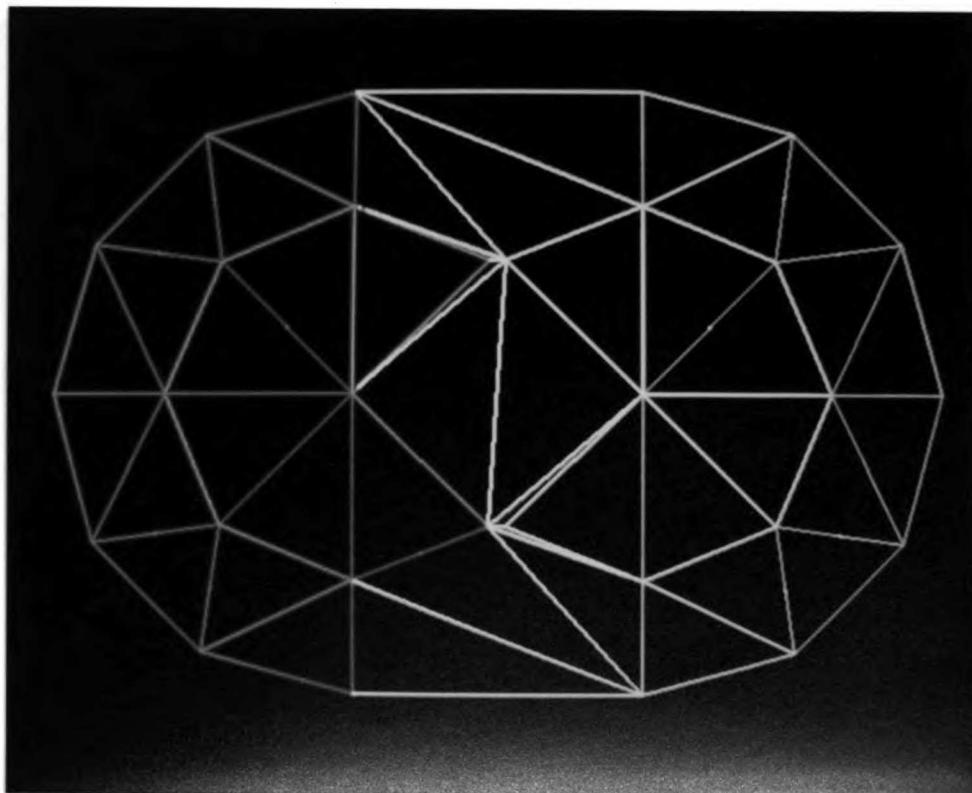


Figure 4. Two Merged Spheres

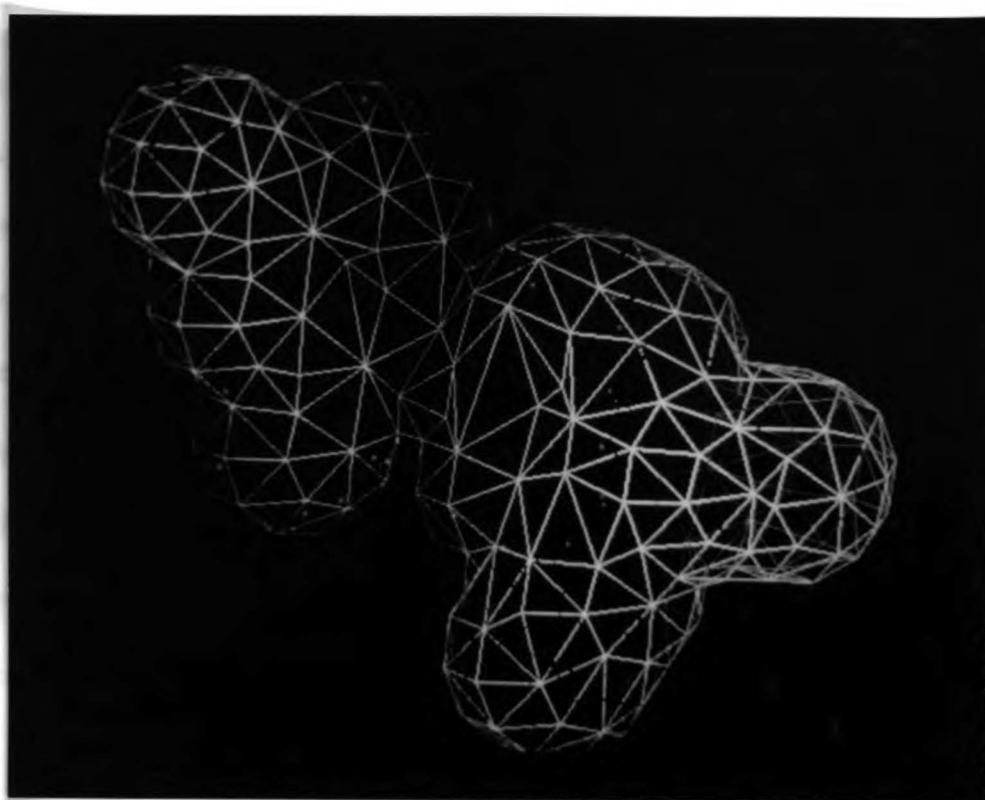


Figure 5. Triangular Net for Benzylpyrimidine Compound

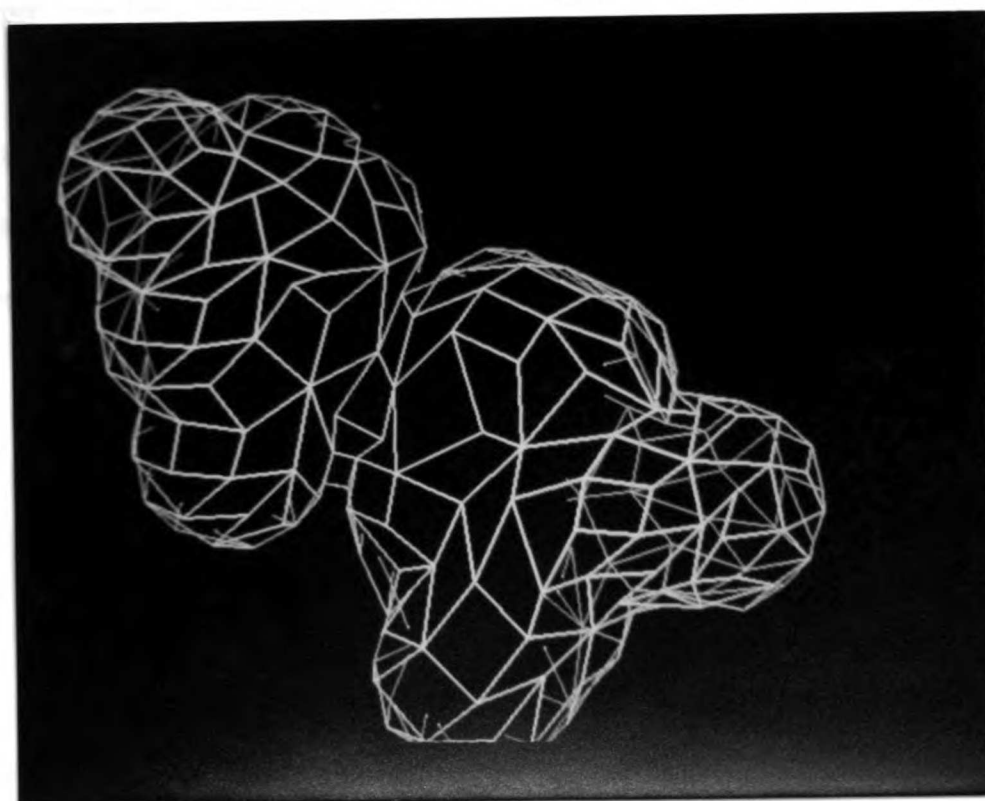
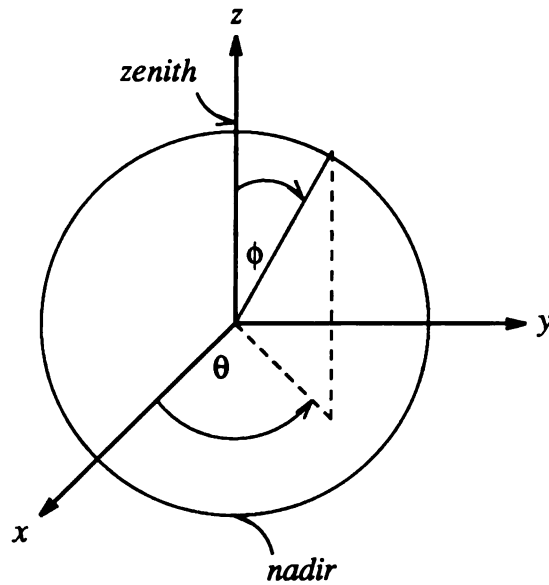


Figure 6. Quadrilateral Net for Benzylpyrimidine Compound

A two-step approach is used in generating the polygonal net. First, the parameters (radius and density) for the prototype sphere are initialized. The user may specify values for the radius and density overriding default values and the control points on the prototype sphere are computed. These control points serve as the vertices for the triangles and quadrilaterals of the polygonal net.

To obtain the desired density of points for a given sphere radius, the sphere is divided into layers each of which contain a number of points uniformly distributed along that layer. The total number of points (obtained by the summation of layers) divided by the area of the sphere ($4\pi r^2$) yields the density of points. To compute the number of layers, points are generated for spheres with between two and ten layers. The layer number that best approximates the desired density is chosen. To compute the number of points per layer for a sphere, the number of points on a great circle are multiplied by the sine of the colatitude angle (ϕ) and to that product, 1 is added. The value of 1 is added so that a point is always obtained at the zenith and nadir of the sphere. The total number of points are computed by summing the points for all layers. The specific angles and points described in this algorithm are diagrammed on the following page:



ϕ = colatitude angle

θ = longitude angle

Points with x , y , and z coordinates are generated by converting from spherical coordinates to Cartesian coordinates.

After generation of the points, triangles are formed on the sphere by initially finding an edge that is the shortest distance between any two points on that sphere. The next closest point to that initial edge and its endpoints are found. The first triangle is then formed and saved in a list for triangles; its edges are also marked and saved as *live* edges. Live edges belong to only one triangle. The live edges are sorted by length and are available to be used in the formation of a new triangle until an edge has become a member of two adjacent triangles. That edge is then marked as a *dead* edge. The longest live edge is always used first to form the next triangle and this procedure is continued until there are no more live edges, yielding a triangulated sphere.

Secondly, the complete surface is calculated by sequentially adding atoms one at a time (see Figures 7a and 7b). The head of the list of atom centers is chosen for the first atom and the center of the prototype sphere is matched to the atom center. This is the initial *blob* which only has dead edges that belong to two triangles. The next atom added is that which is closest in distance to the blob (see Figure 7c). This is done to guarantee that one continuous surface blob is formed as opposed to having two disjoint surfaces (see Figure 7d). Control points that fall within the blob or the new sphere are deleted. The triangles that use these control points are also deleted. Deletion of these triangles leaves edges that are connected to only one triangle (*live* edges). The live edges are subject to reconnections and once an edge has become a member of two triangles, it is marked as a dead edge. The remaining atoms are added sequentially to the blob using the above process. The overlapping subtractions and reconnections are made until all atoms have been incorporated and there are no more live edges.

B
C D
A

Figure 7a

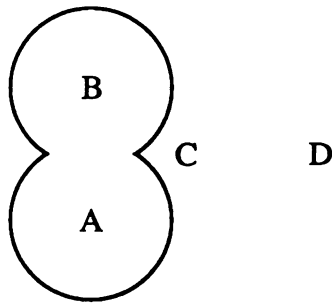


Figure 7b

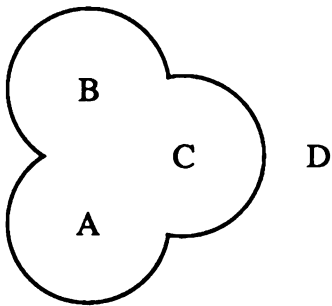
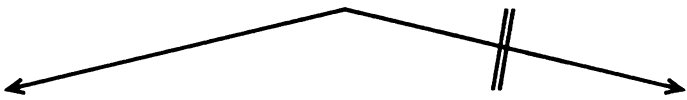


Figure 7c

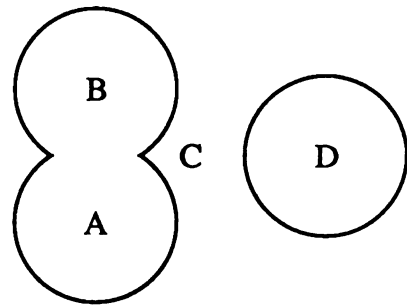


Figure 7d

Figure 7. Pictorial Representation of Merging Algorithm

3.2.1.2. Mathematical Basis of Cardinal Patches

The point described by x , y and z may be defined as a third order polynomial for t by [42, 49, 51]:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + dx$$

$$y(t) = a_y t^3 + b_y t^2 + c_y t + dy$$

$$z(t) = a_z t^3 + b_z t^2 + c_z t + dz$$

A cubic curve segment $C(t)$ can be defined over the range of t where $0 \leq t \leq 1$. $C(t)$ can also be expressed as a vector product as follows:

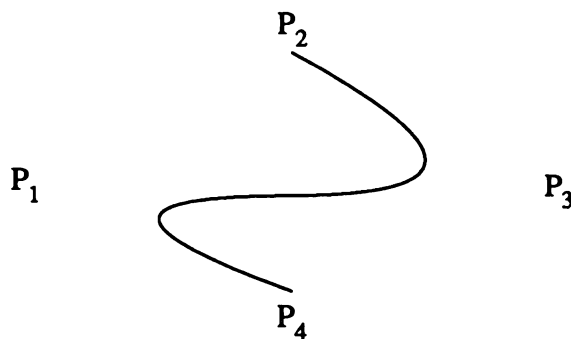
$$C(t) = at^3 + bt^2 + ct + d$$

$$C(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

$$C(t) = TM \quad \text{where} \quad T = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \quad \text{and} \quad M = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

Evaluation of the vector product $C(t)$ for a series of values between 0 and 1 for t yields endpoints for all line segments. The shape of the curve segment which is a function of the control points is determined by the coefficients of the vector product M . M may be defined as the product of the basis matrix (B) and a geometry matrix (G). The geometry matrix contains the control points: four points for spline representation and sixteen

points for surface representation. The values contained in the basis matrix determines the types of spline or patch to be used (e.g., Cardinal or Bézier).⁸ Cardinal splines pass through the two interior control points and are continuous in the first derivatives where the curve segments meet. The rendering of the curve starts at P_2 and ends at P_4 ; the shape of the curve is defined by P_1 and P_3 as seen below:



The basis matrix (B) for Cardinal splines is derived from the following conditions:

$$C(0) = P_2 \quad \text{and} \quad C(0)' = a[P_3 - P_1]$$

$$C(1) = P_3 \quad \text{and} \quad C(1)' = a[P_4 - P_2]$$

where $a > 0$

Therefore, solving for the basis matrix constraints, a Cardinal spline may be defined as:

⁸There are two main differences between the Cardinal splines and Bézier splines. Unlike Bézier splines, Cardinal splines do not have the convex hull property. Additionally, Cardinal splines are interpolating splines whereas Bézier splines are approximating splines.

$$C(t) = TM = T \begin{bmatrix} B \\ G \end{bmatrix}$$

$$C(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -a & 2-a & -2+a & a \\ 2a & -3+a & 3-2a & -a \\ -a & 0 & a & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$

3.2.1.3. Graphical Generation of Cardinal Patches

The graphical generation of Cardinal patches was performed on the Silicon Graphics IRIS 2400T. This graphics workstation was specifically chosen because the IRIS Graphics Library provided the necessary routines for drawing a surface patch.⁹ Patches were displayed as a *wireframe* of curve segments. A single surface patch was generated by specifying the set of sixteen control points, the number of wire segments to be drawn in each direction of the patch and the two parameters which were varied from 0 to 1 for cubic equations (*i.e.*, *s* and *t* in section 3.2.1). The complete surface was generated by overlapping the control points which joined the patches. An enhanced version of BILD¹⁰ was used to display the Cardinal patch surface.

A complete patch surface for a substituted benzylpyrimidine is seen in Figure 8. Figure 8 illustrates the use of multiple colors on a patch surface. These colors may represent different chemical properties. To illustrate the dynamic nature of this surface, a

⁹The routines called for generation of the patches were *defbasis*, *patchbasis*, *patchcurves*, *patchprecision* and *patch*. IRIS User's Guide, Version 2.0

¹⁰BILD is a display program originally written by Oliver Jones at the University of California Computer Graphics Laboratory for the Evans and Sutherland Picture System 2. BILD was ported to the Silicon Graphics IRIS by Conrad Huang.

control point has been moved for one of the surface patches seen in Figure 8. Figure 9 shows a piece of that patch surface before (green) and after (red) the movement of the control point. Where the surface is represented in yellow, there is no difference between the two surfaces. From the view seen in Figure 9, it is difficult to see what effect the movement of the control point had on the surface; however, in Figure 10, the surface has been rotated clearly showing what effect the movement of the control point has had on the local area of the surface. Figures 9 and 10 demonstrate the need for the surface display and manipulation programs of KARMA to be interactive and dynamic. Lastly, Figure 11 shows how the density of patch lines can be changed (as compared to Figures 9 and 10) as a means to convey information to the user.

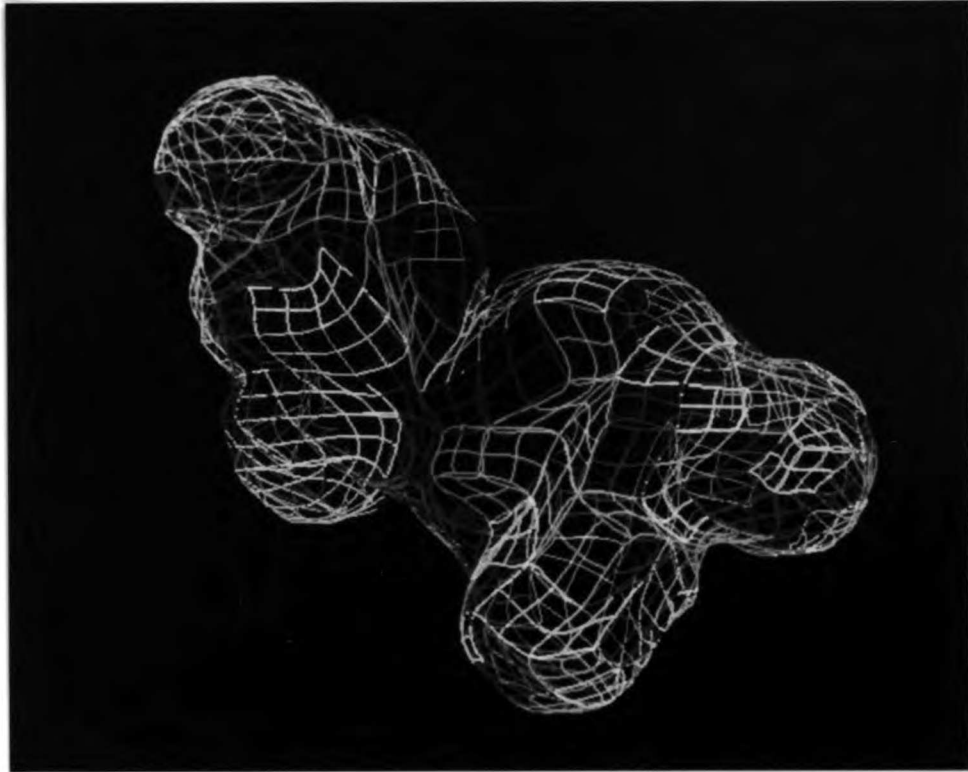


Figure 8. Cardinal Patch Surface Model

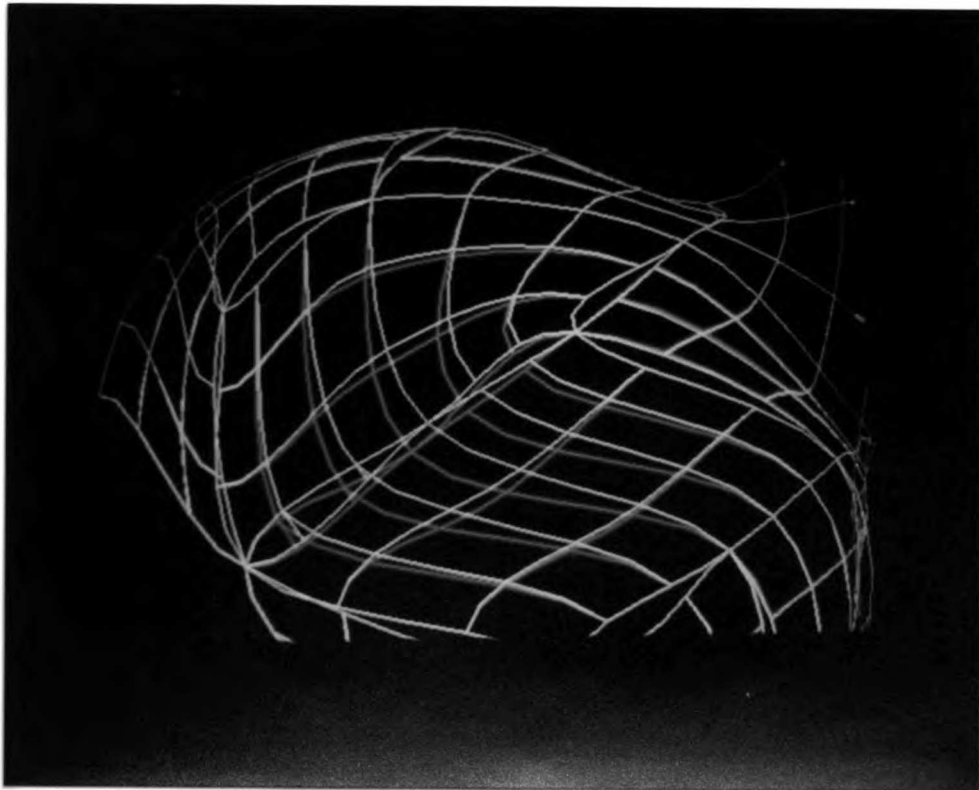


Figure 9. Cardinal Patches Before (*green*) and After (*red*) Movement

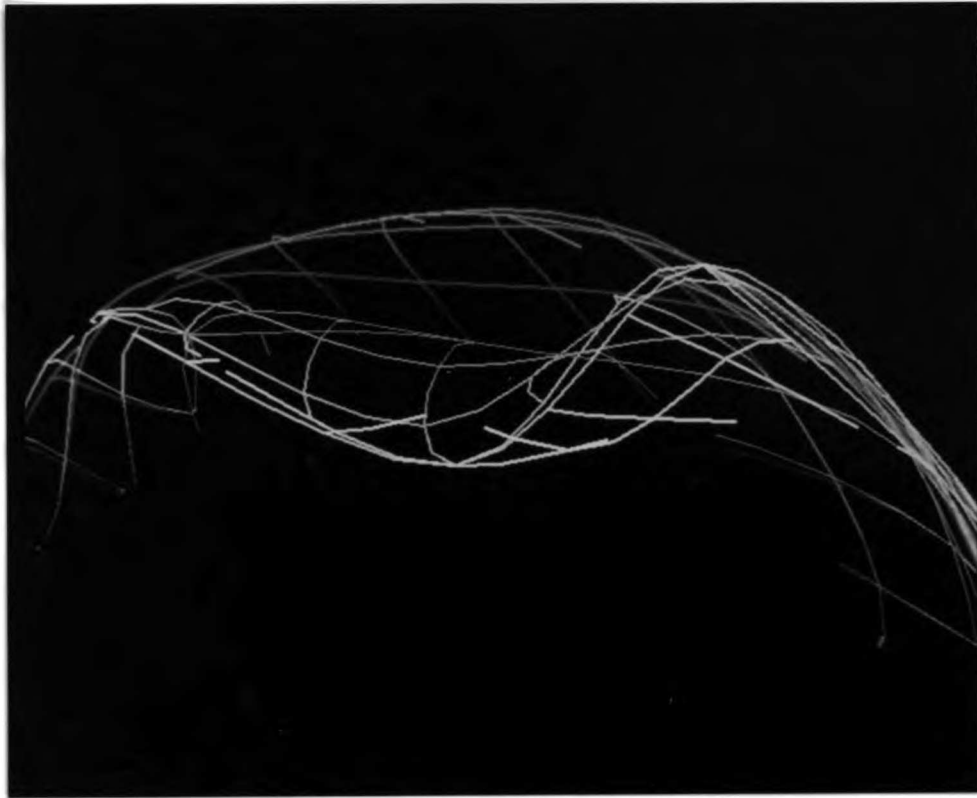


Figure 10. Rotated Cardinal Patches Before (*green*) and After (*red*) Movement

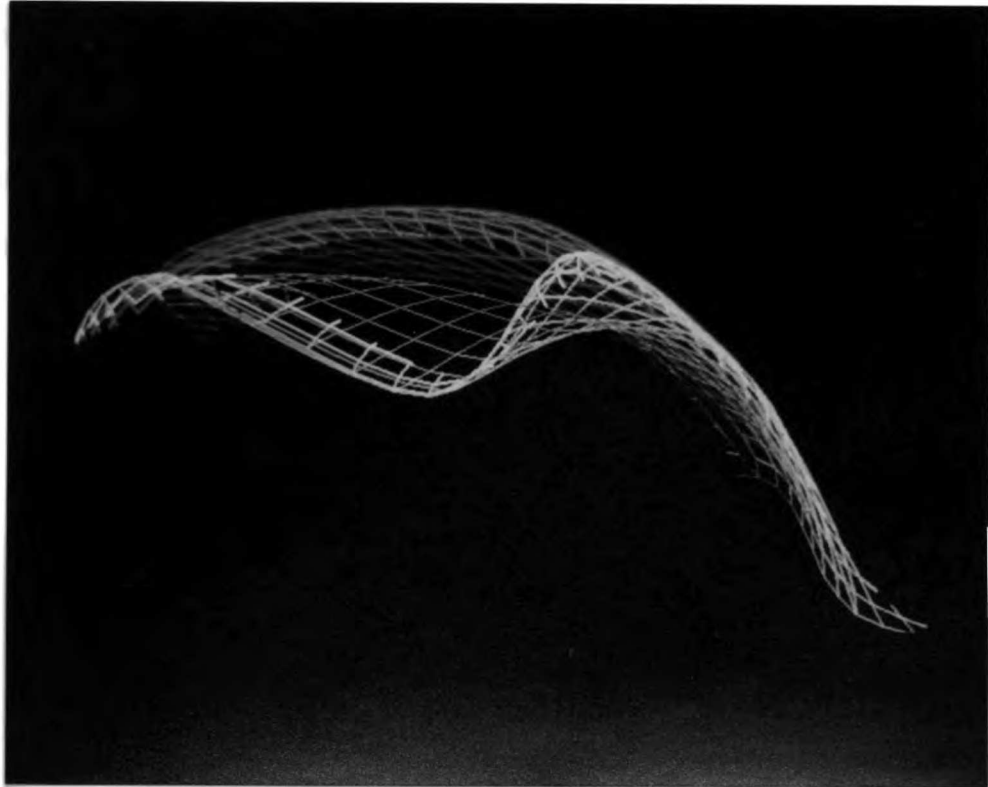
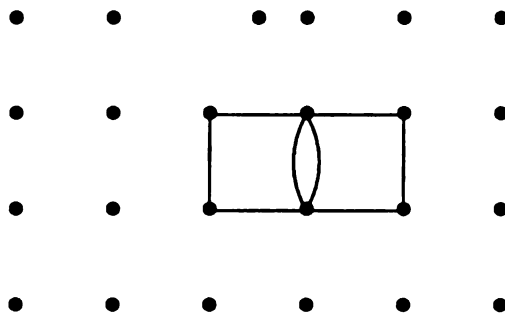


Figure 11. Increased Density for Display of Cardinal Patch Surface

3.2.1.4. Discontinuity of Cardinal Patch Surface

The generation of the bicubic patch surface based on Cardinal patches is not continuous between patches. Graphically, this results in rosette like patterns at the boundaries of adjoining patches (see Figure 12). This problem is attributed to adjacent patches not sharing the same control points; hence, the common spline edges between adjacent patches are not identical as seen below:



Since the control points do not fit a rectangular grid, a control point may belong to three or more patches. Regardless of the basis matrix (*e.g.*, Cardinal or Bézier), this approach to the surface representation will not be successful because there is discontinuity between the patches.

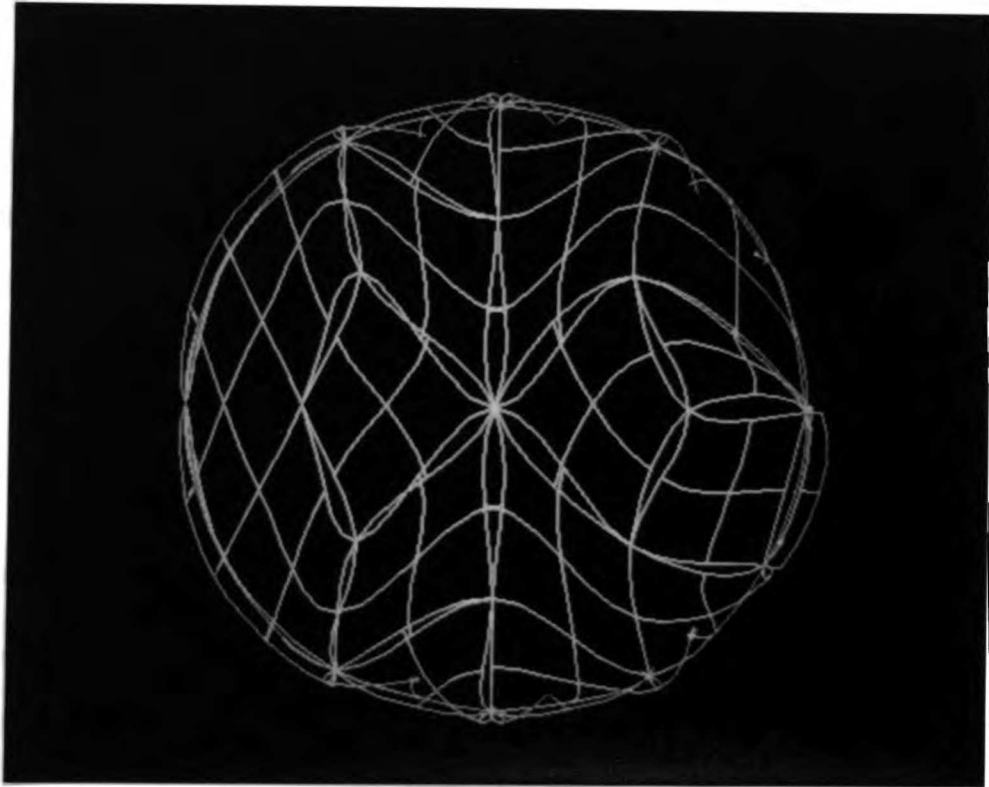


Figure 12. Discontinuity Between Cardinal Patches

3.2.2. Surface Generation Using Gregory Patches

The surface model for KARMA may be considered as a geometrical object with arbitrary shape and topology. The Cardinal patch representation is limited by its conditional need for rectangular regularity of its control points for a continuous fit of surface patches. Triangular patches which are better at describing free form shapes cannot be described mathematically by the Cardinal patch representation. H. Chiyokura has described the mathematics for using non-polynomial Gregory patches for solids modeling with free form surfaces [52, 53]. C. Séquin and his students have recently described a method for the interpolation of Gregory patches between cubic boundaries using Chiyokura's equations [54 - 57]. In their program UNICUBIX which is an extension to the University of California, Berkeley UNIGRAFIX modeling and rendering system, an object is described by vertices positioned in space and by edges and surface patches stretched between these points [54]. The mathematics and implementation techniques are presented in sections 3.2.2.2 and 3.2.2.3.

The smooth and continuous surface generated for KARMA uses cubic Bézier curves for the boundaries between patches. Quartic Gregory patches are joined with tangent-plane continuity to provide a smooth interpolated surface. Gregory patches are used as opposed to Bézier patches because Gregory patches provide twice as many interior control points [57]. These extra control points are needed to obtain tangent-plane continuity between patches. To determine which points define which patches, a polygonal triangular net is calculated. The triangular net is converted to a parametric representation in three steps. The first step calculates the vertex normals based on the vertices of the triangles and the angles associated with these vertices. The second step

involves the determination of Bézier control points for the curved edges. Lastly, the internal control points used for the Gregory patches are determined.

3.2.2.1. Generation of Triangular Net

The first step in computing the surface for a set of molecules is to approximate a surface with a set of adjacent triangles that enclose the volume of the molecules. Unlike the algorithm presented in section 3.2.1.1 where the control points were distributed in approximate uniformity over a sphere and then merged, the control points for the algorithm presented in this section are distributed in approximate uniformity over the entire surface. Uniformity of control points over the entire surface allows the triangles of the polygonal net to better approximate equilateral triangles, yielding a smoother looking surface.

As a general overview, points are generated on the surface based on a desired distance between points (see Figure 13). Edges of the desired distance are created and polygons are formed from these edges (see Figure 14). These polygons consist of triangles and higher order polygons. The non-triangular polygons are then subdivided into triangles by adding new edges, yielding a triangulated surface with approximately uniform triangles (see Figure 15). In terms of complexity, the algorithm is N^2 where N is the number of atoms to be used for determining the surface. An implementation of this triangulated surface is presented in Appendix One.

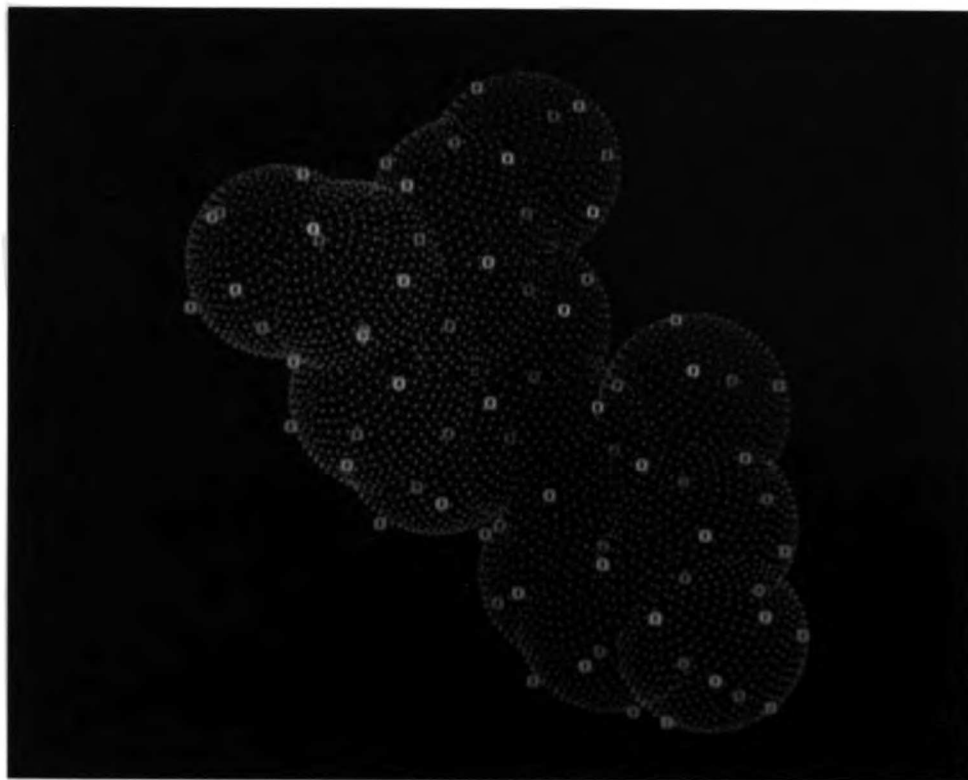


Figure 13. Initial Surface Based on Desired Distance

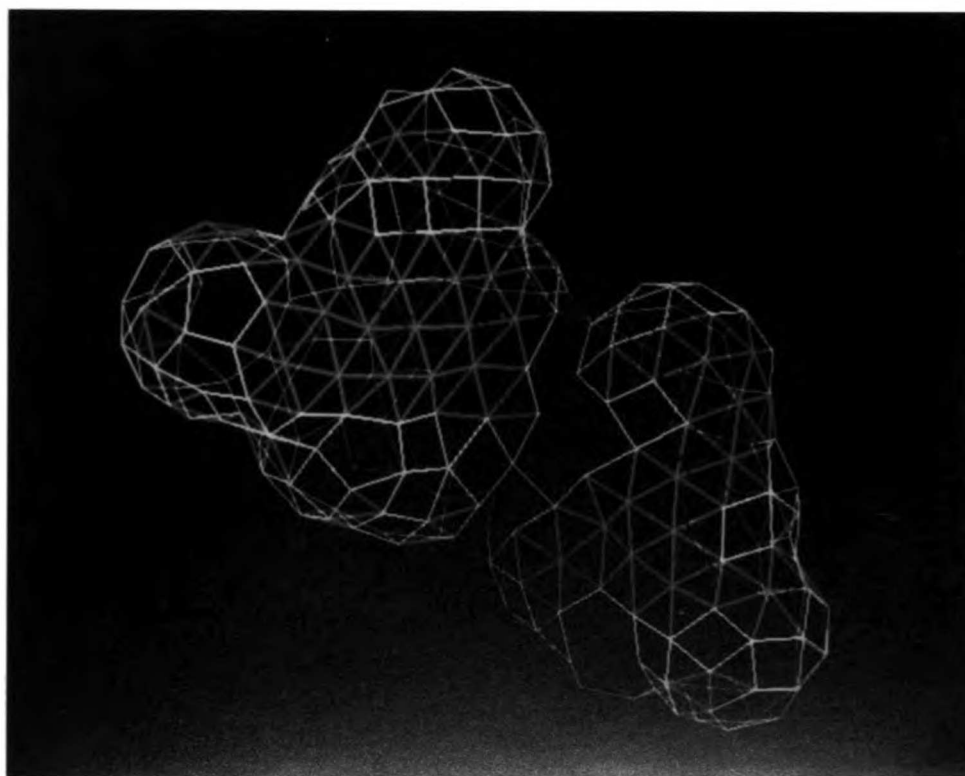


Figure 14. Polygonal Net for Benzylpyrimidine Compound

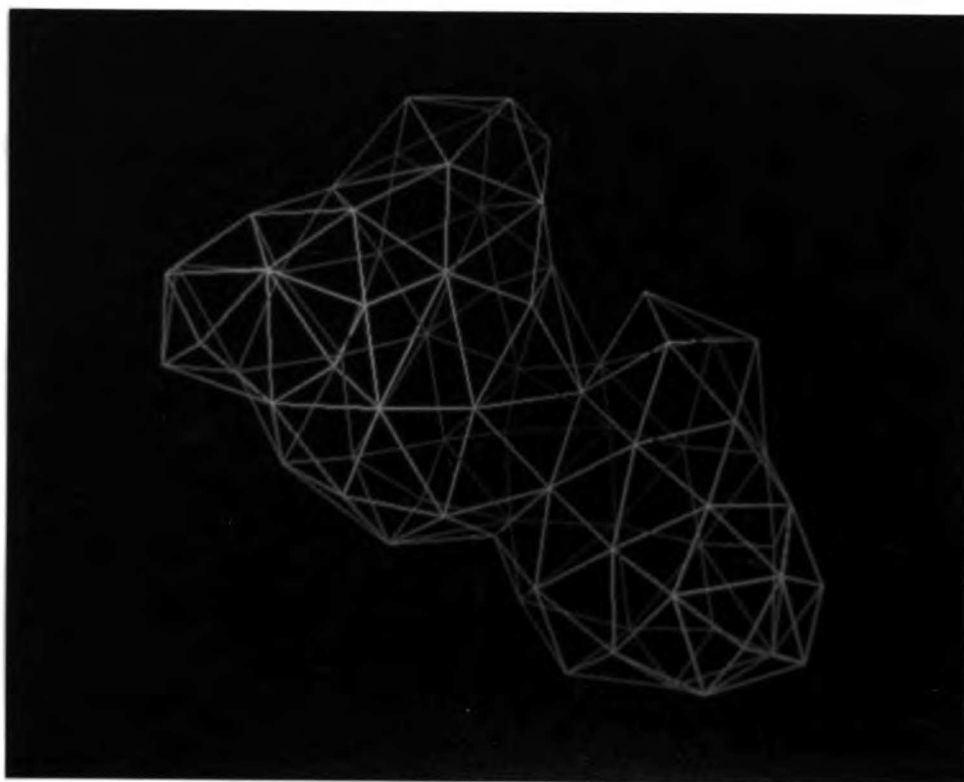


Figure 15. Uniform Triangular Net for Benzylpyrimidine Compound

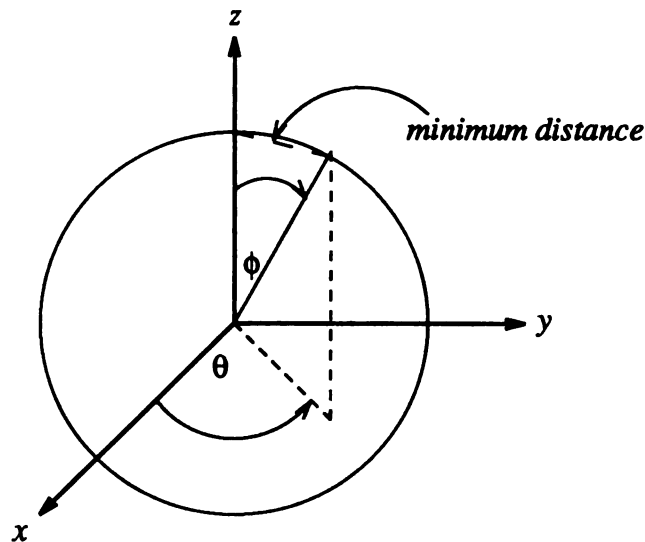
The generation of surface points is based on a desired distance (*target distance*) between the surface points. Initially, points are generated on a sphere that are approximately one-tenth the target distance apart (*minimum distance*). Similar to the algorithm in section 3.2.1.1, the sphere is divided into layers. To compute the number of layers, it is necessary to first compute $\Delta\phi$ which is approximated by:

$$\Delta\phi \approx \arctan \left[\frac{\text{minimum distance}}{\text{radius}} \right]$$

The number of layers can then be obtained by:

$$\text{number of layers} = \frac{\pi}{\Delta\phi}$$

The specific angles and distances used in this algorithm are diagrammed below:



ϕ = colatitude angle

θ = longitude angle

After the number of layers has been calculated, the number of points for each layer is determined. The layers are obtained by stepping through the colatitude angle with a $\Delta\phi$ step size. The number of points at each ϕ is found by first computing the circumference of the circle at that colatitude and then dividing that circumference by the minimum distance. Thus,

$$\text{number of points for layer}_i = \frac{\text{circumference}}{\text{minimum distance}}$$

To obtain all the points on that layer it is necessary to calculate $\Delta\theta$ which is found by:

$$\Delta\theta = \frac{\pi}{\text{number of points for layer}_i}$$

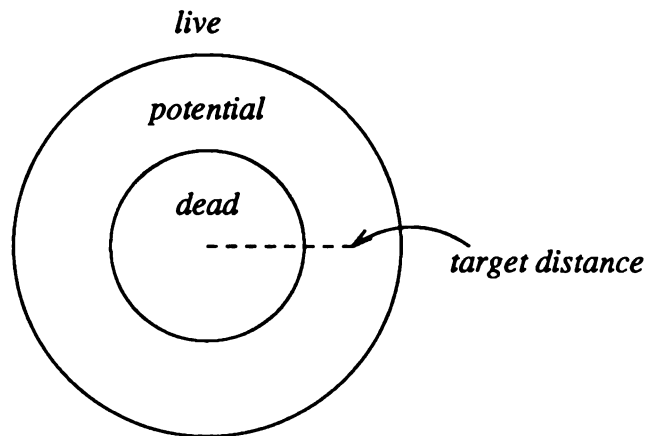
All points on the layer are obtained by stepping through the longitude angle (θ). These points are *not* the control points, but rather potential control points. Most of the points from this calculation are discarded.

Having obtained a sphere described by a set of points, the equation,

$$P = \bigcup_i P_i - \bigcup_{i,j} I_{ij}$$

where P_i is the distributed set of points over a sphere corresponding to atom i , and, I_{ij} is the set of points in P_i which fall inside atom j , is used. The algorithm that described the merging of spheres in section 3.2.1.1 (and pictorially represented in Figure 7) is then used to calculate the complete surface except points are used in the algorithm instead of edges.

To choose the actual control points, a band of acceptance is used to sort out points as seen on the following page:



All points are marked with a label of *live*, *dead*, *potential*, or *saved* and each point has a counter associated with it. Initially, all points are marked as *live*. Then, an arbitrary point is chosen and all points which are less than the target distance minus the minimum distance to that point are marked as *dead*. Any point that is not marked as *dead* and is less than the target distance plus the minimum distance is marked as *potential*. Each time a point is marked *potential*, the counter for that point is incremented. Additionally, associated with each point is how close that point is to the target distance (*i.e.*, minimum Δ). The next point saved is that which is marked as *potential* and has the largest value in its counter. If more than one point exists with the same counter value, the point that has the minimum Δ is chosen and saved. Points can only go from *live* \rightarrow *potential*, *live* \rightarrow *dead*, *potential* \rightarrow *dead*, or *potential* \rightarrow *saved*. This procedure is repeated until there are no more points marked as *live* or *potential*. An approximately uniformly distributed set of points for the surface is obtained. These points are a subset of the control points for the surface model.

After obtaining the set of points that describes the outline of the surface model, the next step in generating the triangular net is to form the obvious edges based on the

distance (d) falling in the interval:

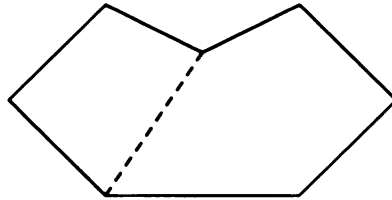
$$(\text{target distance} - \text{minimum distance}) < d < (\text{target distance} + \text{minimum distance})$$

The edges have an associated counter which is incremented every time an edge is used. As with the initial approach, edges that are used twice are no longer available to be used in another polygon. Once the obvious edges have been found, the obvious triangles are formed. For every edge, all points adjacent to one endpoint (E_1) are checked. If an adjacent point (A) is also adjacent to the other endpoint (E_2), then a triangle is formed with vertices A , E_1 , and E_2 . Those points which are neighbors to both endpoints are used to form the obvious triangles.

After the obvious triangles are found, higher order canonical polygons are determined. Canonical polygons are those polygons which cannot be decomposed into smaller polygons. To find the canonical polygons, a depth first search is performed along the live edges. An initial live edge is chosen and a path from one of its endpoints (forming one vertex of the polygon) to the other endpoint is found. The direct connection between the two endpoints is discarded. Paths are only accepted which describe canonical polygons.

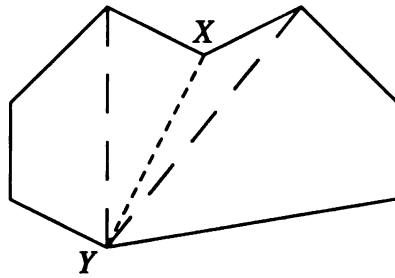
A multistep procedure is used to determine whether a polygon is canonical. Initially, the points and edges for the given polygon are labeled uniquely. The next step is to find a point not on the polygon. This point is marked with the unique label and a depth first search is done from this point. All of its neighboring points and edges which are not already marked with the unique label are visited and marked. This step is continued until all reachable points have been visited. If there are edges which are not marked remain with the unique label, then the polygon is not canonical. This is

illustrated below:

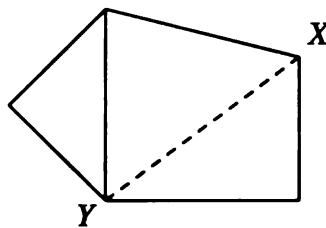


Two procedures improve the search for potential polygons. The first is α pruning. If a canonical polygon is found at search depth N , then the search is limited to depth N from that point on. In addition to α pruning, an aspirative search technique is used. An aspirative search sets a limit to the search depth N from the outset and will not search beyond N . If a solution is found, the search is complete. If a solution is not found, the search depth is increased. For example, if an initial depth value of 7 is chosen and if a polygon cannot be found within that maximum depth, the depth value is incremented by that same amount ($7 + 7 = 14$) and the search is repeated. Termination of these searches occurs when there are no more live edges or the number of live edges left is less than the current aspirative search depth value. The advantage of using the aspirative search is that it finds simple polygons quickly and eliminates those edges from consideration for future searches, thus limiting search space.

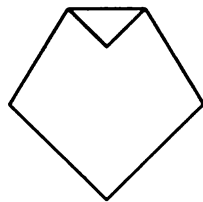
Lastly, it is necessary to subdivide the canonical polygons into triangles. The initial step is to make all polygons convex. This is accomplished by looking at each polygon and its points, and eliminating concavities by forming new edges. A point X is concave if there exists a point Y on a polygon such that the distance between Y and X is less than the distance from the neighbors of Y to X as illustrated on the following page:



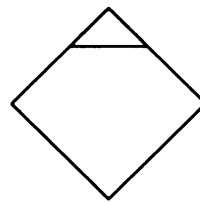
To eliminate the concave point, a new edge is formed between point *Y* and the point closest to point *Y* which satisfies the concavity criterion. This is illustrated below:



Concavity cannot be determined from connectivity and must be determined from numerical values as seen in Figure 16:



concave



convex

Figure 16: Two polygons with the same connectivity.

This process yields two polygons which go through the same checks for the convex property, as breaking a concave polygon does not guarantee two convex polygons.

After all the polygons have been determined to be convex, the target distance constraint is relaxed such that any distance up to four-thirds the target distance is acceptable. The value of relaxation can be set by the user; however, the value of four-thirds the target distance is a good choice because the longer and shorter edges are within equal distance of the target distance. Polygons which are not triangles are split into triangles by adding a new point closest to the center of mass of the vertices of each polygon. The point is added by finding the closest dead point (see section 3.2.2.1) to the center of mass and using it to form new triangles. The addition of this point yields edges which are shorter than the target distance.

3.2.2.2. Mathematical Basis of Gregory Patches

A Gregory patch is defined by its control points,

$$P_{i,j,k} \quad i, j = 0, \dots, n, \quad k = 0, 1.$$

For triangular Gregory patches, barycentric coordinates are used. Quartic boundaries are used for the triangular patches. By using quartic boundaries, more flexibility is gained in the manipulation of the interior control points (six for quartic, three for cubic) when searching for conditions of cross-boundary continuity.

A triangular Gregory patch of degree four is determined by eighteen control points (six interior, twelve boundary) as seen in Figure 17 [56].

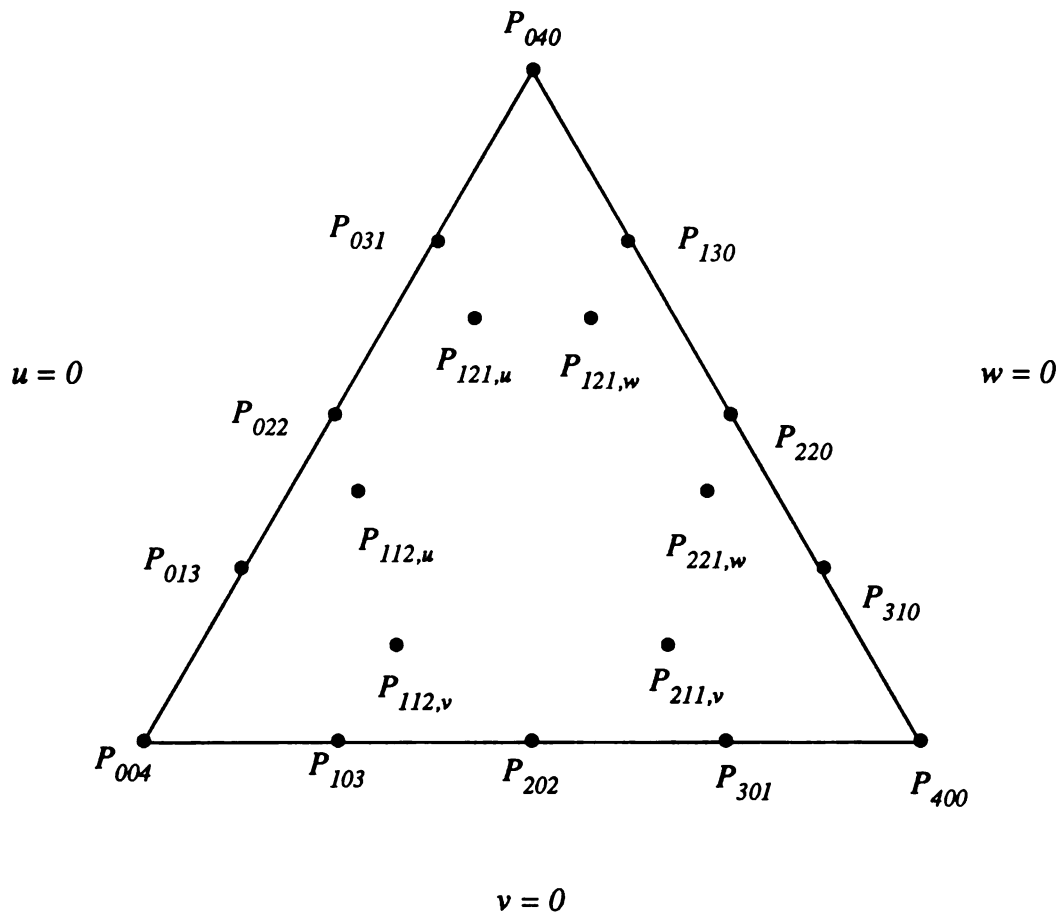


Figure 17. Triangular Gregory Patch

The triangular Gregory patch of degree four can be described mathematically by [55]:

$$g(u, v, w) = (uE + vF + wG)^4 P_{000}(u, v, w), \quad u, v, w \geq 0, \quad u+v+w = 1$$

E, F and G are shift operators where

$$EP_{ijk} = P_{i+1jk}$$

$$FP_{ijk} = P_{ij+1k}$$

$$GP_{ijk} = P_{i j k+1}$$

where $P_{ijk}(u, v, w) = P_{ijk}$ for $i = 0$ or $j = 0$ or $k = 0$

(control points at the bound of the patch)

and

$$P_{121}(u, v, w) = \frac{(1-u)w P_{121, u} + u(1-w) P_{121, w}}{(1-u)w + u(1-w)}$$

$$P_{112}(u, v, w) = \frac{(1-u)v P_{112, u} + u(1-v) P_{112, v}}{(1-u)v + u(1-v)}$$

$$P_{211}(u, v, w) = \frac{(1-v)w P_{211, u} + v(1-w) P_{211, v}}{(1-v)w + v(1-w)}$$

All points on a patch are contained within the convex hull of their control points. Thus, triangular Gregory patches have the convex hull property [58].

Quartic boundaries are obtained in two steps. Bézier cubic boundaries are calculated and then degree elevated to quartic boundaries [55, 56]. Bézier cubic boundaries are constructed by calculating vertex normals and then the Bézier points. To obtain smooth joining cubic boundaries for all vertices of the triangular net constructed in section 3.2.2.1, a vertex normal which defines the tangent plane must be computed. The normal is computed as a weighted average of all normals of the faces for each vertex. The weight factor is the angle between two edges that use a specific vertex. The Bézier points are found by projecting the original edge of the triangular net onto the vertex tangent plane. The vertex tangent is perpendicular to the vertex normal. The distance between the vertex and the Bézier point is equal to one-third the length of the edge.

Quartic boundaries which are used for the Gregory patch surface have one more control vertex than the cubic boundaries. Thus, the cubic boundaries (degree three) need to be degree elevated to degree four for quartic boundaries. The new control points for the quartic boundaries can be found as a linear expression of the control points from the cubic boundary as see in Figure 18 [54].

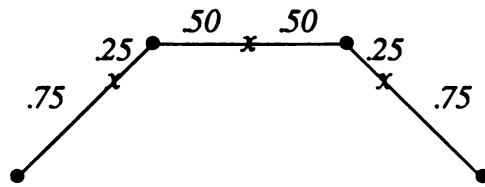


Figure 18. Degree Elevation for Patch Boundaries

To render a patch surface smooth, the patches must meet *smoothly* at the joint boundaries with geometric continuity of degree 1 (G^1). If the surface normal direction at the boundary of two adjoining patches is continuous, the patches have G^1 continuity. Therefore, it is necessary to find the conditions for the interior control points which define the patches on either side of the shared boundary that guarantees G^1 continuity between the two patches.

Two patches may be joined with G^1 continuity¹¹ at the bounds using the method described by H. Chiyokura [52, 53]. This method uses boundary information exclusively to find the two interior control points on either side of the boundary independently

¹¹For a general discussion of geometric continuity across boundaries, please see literature cited by L. Longhi, L. Shirman and C. Séquin and those references cited within.

(requiring two passes for each boundary). A *basis* patch is constructed in the place of the neighboring patch and G^1 conditions are written for the real and basis patch. This results in the following equations for the two interior control points of the real patch based on Figure 19.

$$P_0 = \frac{T_0 - R_0}{|T_0 - R_0|}$$

$$P_3 = \frac{T_3 - R_3}{|T_3 - R_3|}$$

where vectors P_0 and P_3 are unit vectors. Vectors P_1 and P_2 are linearly interpolated between P_0 and P_3 by:

$$P_1 = \frac{2}{3} P_0 + \frac{1}{3} P_3$$

$$P_2 = \frac{1}{3} P_0 + \frac{2}{3} P_3$$

Then,

$$T_1 = \frac{(k_1 - k_0)}{3} P_0 + k_0 P_1 + \frac{2}{3} h_0 S_1 + \frac{h_1}{3} S_0$$

$$T_2 = k_1 P_2 - \frac{(k_1 - k_0)}{3} P_3 + \frac{h_0}{3} S_2 + \frac{2}{3} h_1 S_1$$

where the coefficients k_0 , k_1 , h_0 and h_1 are found from

$$T_0 = k_0 P_0 + h_0 S_0$$

$$T_3 = k_1 P_3 + h_1 S_2$$

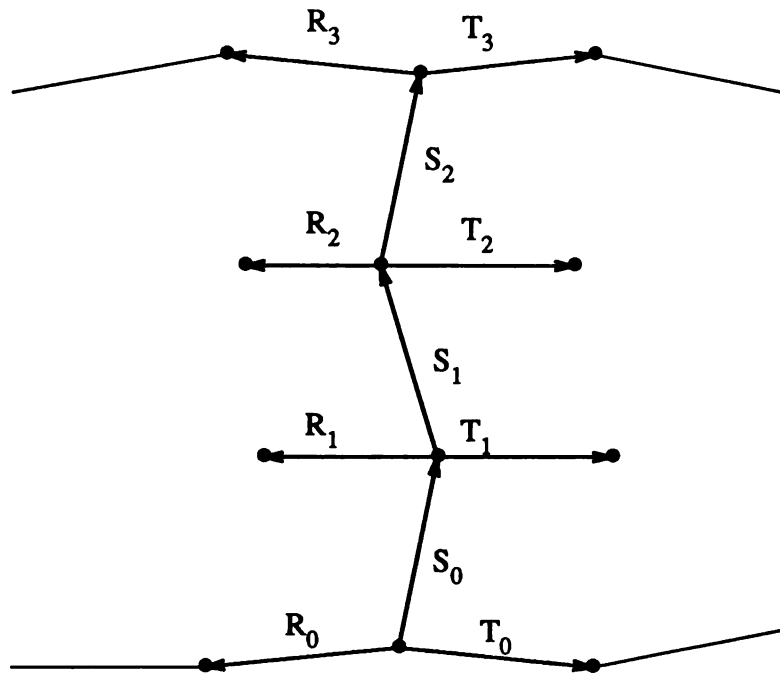


Figure 19. Boundary Between Two Patches

G^1 continuity between the two real patches is guaranteed because the two neighbor basis patches are always G^1 continuous. Using this method, a smooth and continuous surface of Gregory patches is derived.

3.2.2.3. Graphical Generation of Gregory Patches

The graphical generation of Gregory patches was performed on the Silicon Graphics IRIS 2400T. Patches are displayed both as a wireframe of curve segments or a solid polygonal surface. A single wireframe surface patch is generated by specifying the set of eighteen control points, the number of wire segments to be drawn in each direction of the patch and the three parameters (u, v, w) which were varied from 0 to 1. The number of

patches displayed goes up as N^2 where N is the number of atoms used to define the surface. A display program was written based on BILD to display the Gregory patch surface. Currently, Z-buffered Gouraud shading using filled polygons is used for rendering the solid polygonal surface¹². An implementation of the display program for the patches is presented in Appendix Two.

A complete Gregory patch surface for a substituted benzylpyrimidine is seen in Figure 20. Figure 20 illustrates the use of multiple colors on a patch surface. These colors may represent different chemical properties. It is interesting to compare Figure 8 and Figure 20. Clearly, the use of Gregory patches (Figure 20) yields a smoother and more continuous surface than the does the use of Cardinal patches (Figure 14).

To illustrate the dynamic nature of this surface, Figure 21 shows a control point which has been moved but not yet saved. In Figure 21, the control points are represented as crosses (cyan); the original position of the surface is red; the portion of the surface which had to be recalculated as a function of that control point is green; and the orientation axis for movement is yellow. The control point can be moved along the normal or within the tangent plane to the surface. Figure 21 is illustrative of normal translation. A new position can be saved as seen in Figure 22.

¹²The routines on the Silicon Graphics IRIS called for Gouraud shading the patch surface are *setshade*, *pmv*, *pdr*, and *spclos*. IRIS User's Guide, Version 2.0

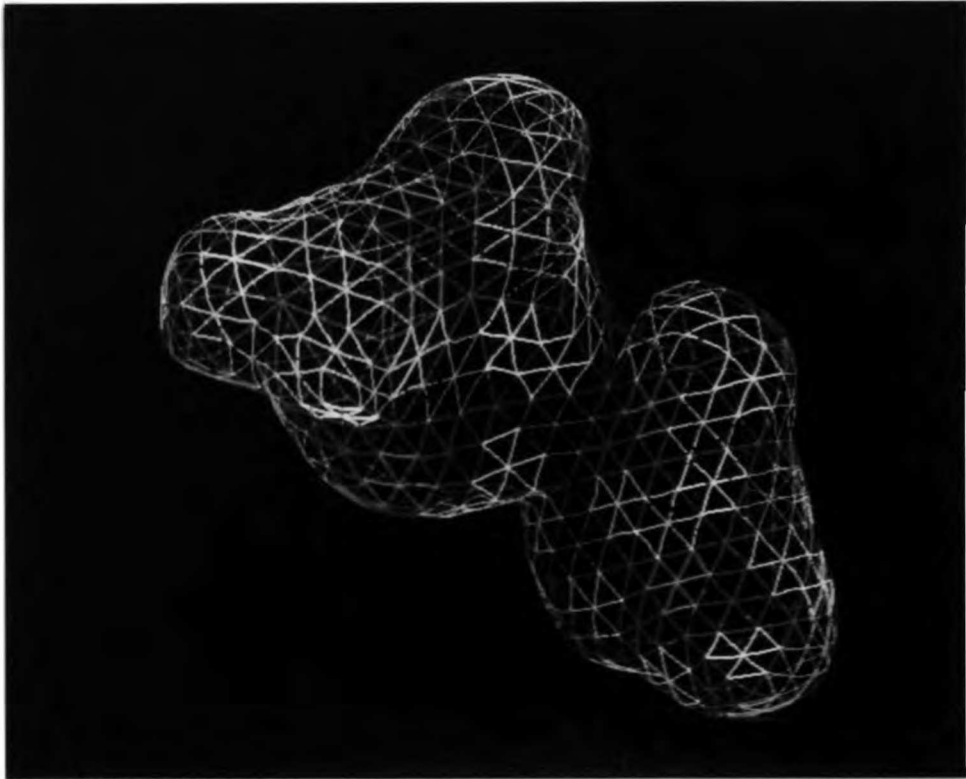


Figure 20. Gregory Patch Surface Model

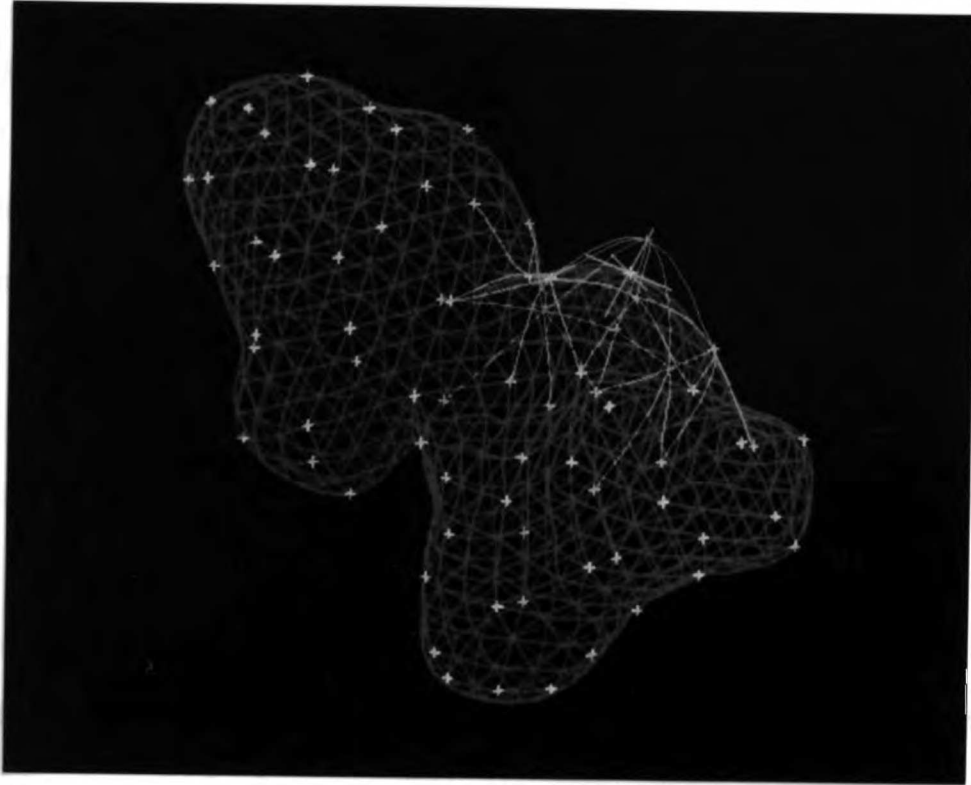


Figure 21. Movement of a Control Point

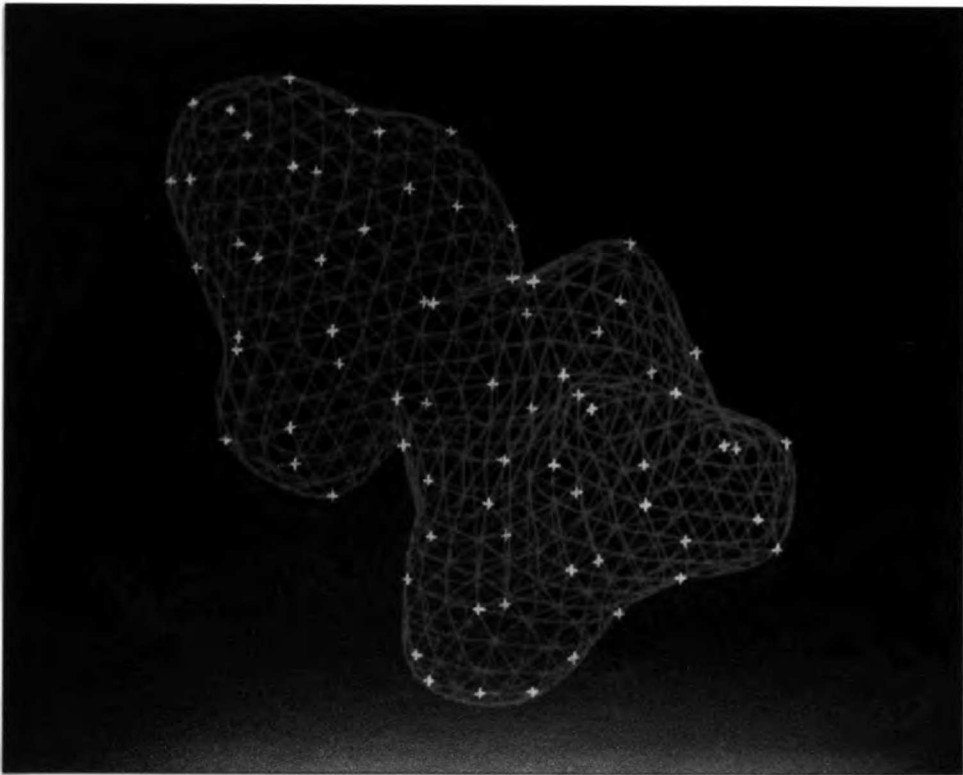


Figure 22. Saved Position Following Movement of the Control Point in Figure 21

Currently, four styles of solid rendering are available. These styles are solid, thatched, half-tone and cross-hatched. For all solid rendering, there is a light source that is infinitely far away which defines a direction from which light rays appear. This light source may be moved around by the user from a menu. Additionally, to distinguish between the inside and the outside of the surface, multiple colors may be used. For example, as seen Figure 23, the outside of the thatched style surface is green and the inside of the surface is magenta. This is more clearly seen in Figure 24 where the clipping plane has been moved forward. The thatched style of surface is useful to give some appearance of transparency¹³ as compared with the cross-hatched style surface as seen in Figure 25. However, multiple colors for the inside (magenta) and outside (green) of the surface is useful for the more *solid looking* representation as seen by movement of the clipping plane in Figure 26.

¹³Implementation of this surface program on newer hardware will have *true* transparency.

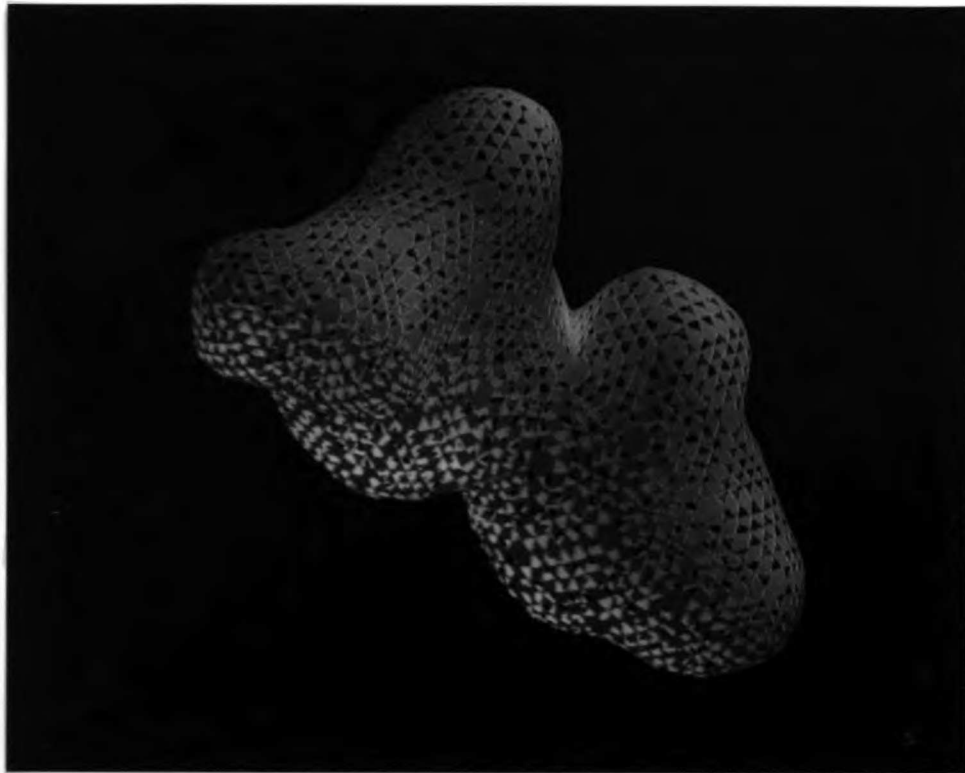


Figure 23. Thatched Style Rendering for Gregory Patch Surface

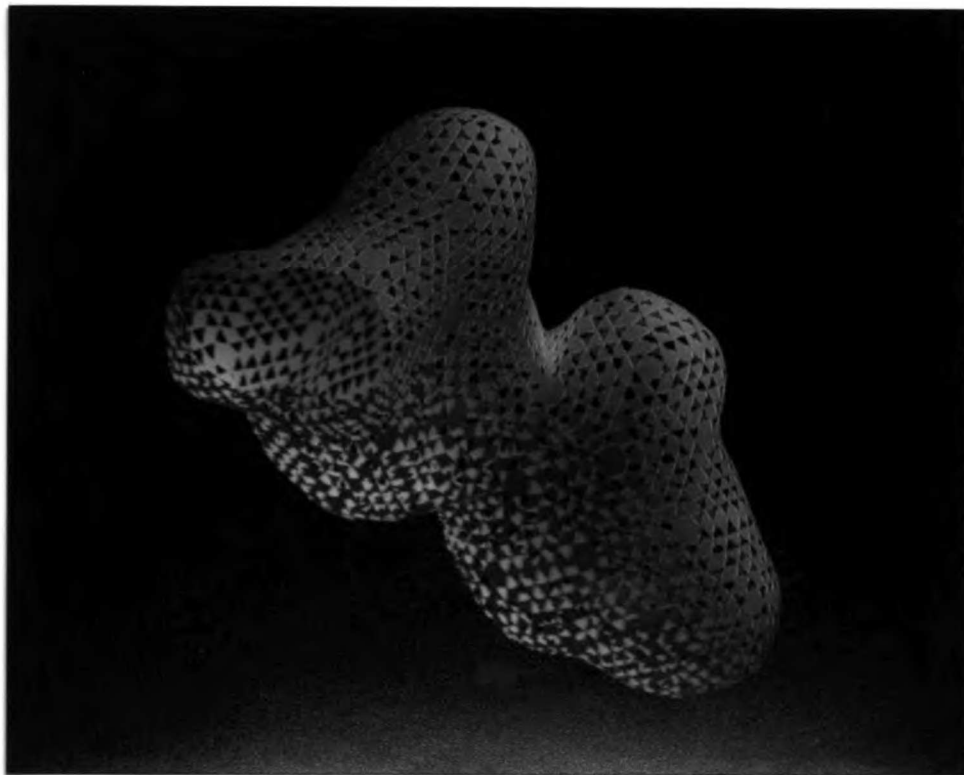


Figure 24. Thatched Style Rendering clipped to show Interior Surface



Figure 25. Cross-Hatched Style Rendering for Gregory Patch Surface



Figure 26. Cross-Hatched Style Rendering clipped to show Interior Surface

3.3. Summary

The receptor model for KARMA is defined by a set of structures chosen by the user. These structures are used to generate a surface for representing the receptor model. The design criteria for the surface included: (1) the surface be closed and generated about a convex volume; (2) the surface be smooth and continuous; and (3) the surface be manipulable in real-time. Initial attempts including the Cardinal patch representation did not result in a surface meeting all of the above criteria. Specifically, the Cardinal patch surface was not a continuous surface. The Gregory patch surface successfully and efficiently met the above design criteria for the KARMA surface.

Future Developments of KARMA

Currently, KARMA is in the prototyping phase. The hardware is connected via the high bandwidth network and the servers for data communications will be implemented in the near future. The graphics package in development for KARMA is being rewritten for a new three-dimensional graphics workstation based on the shadowfax technology of the Evans and Sutherland PS390¹⁴.

As a result of the desire to create an efficient system for surface development and generation¹⁵, the knowledge bases and rule system are just now being implemented in KEE. The structure of knowledge bases will undergo many refinements as the system becomes more developed. Since the rule system has the flexibility for user additions, as the scientific field develops and becomes more refined so will the rule system of KARMA. Issues such as collinearity and compensation (between enthalpy and entropy) of the data will have to be dealt with as both the system and medicinal chemistry gets more sophisticated.

4.1. Annotation of the Surface Model

The surface model is characterized by a set of rules which act upon information stored in KARMA's knowledge bases. The knowledge bases contain information obtained from quantitative structure-activity relationships, kinetic data and structural chemistry.

¹⁴PS390 is a registered trademark of Evans and Sutherland.

¹⁵The surface generation program of KARMA is being considered for implementation for other molecular modeling applications such as an enhancement to the solvent accessible surface [48].

Presently, two sets of rules govern the gross morphology and detailed shape and characteristics of the theoretical surface receptor model.

4.1.1. KARMA's Knowledge Bases

Information used by KARMA's rule system is stored in knowledge bases. KEE provides the storage, consistency and manipulation functions for the knowledge bases (see section 2.2.1.1). The overall knowledge representation is schematically shown in Figure 27. Using KEE's nomenclature for Figure 27, a class (denoted in *italics*) describes a set of objects, a member (denoted in **bold**) describes a particular instance of a class and slots (denoted in Times-Roman) describe attributes of an object. Methods (denoted in *Helvetica-oblique*) act on an object. Slots may contain a single value (*i.e.*, a SMILES's identifying code) or they may contain multiple values (*i.e.*, the three-dimensional coordinates for a molecule).

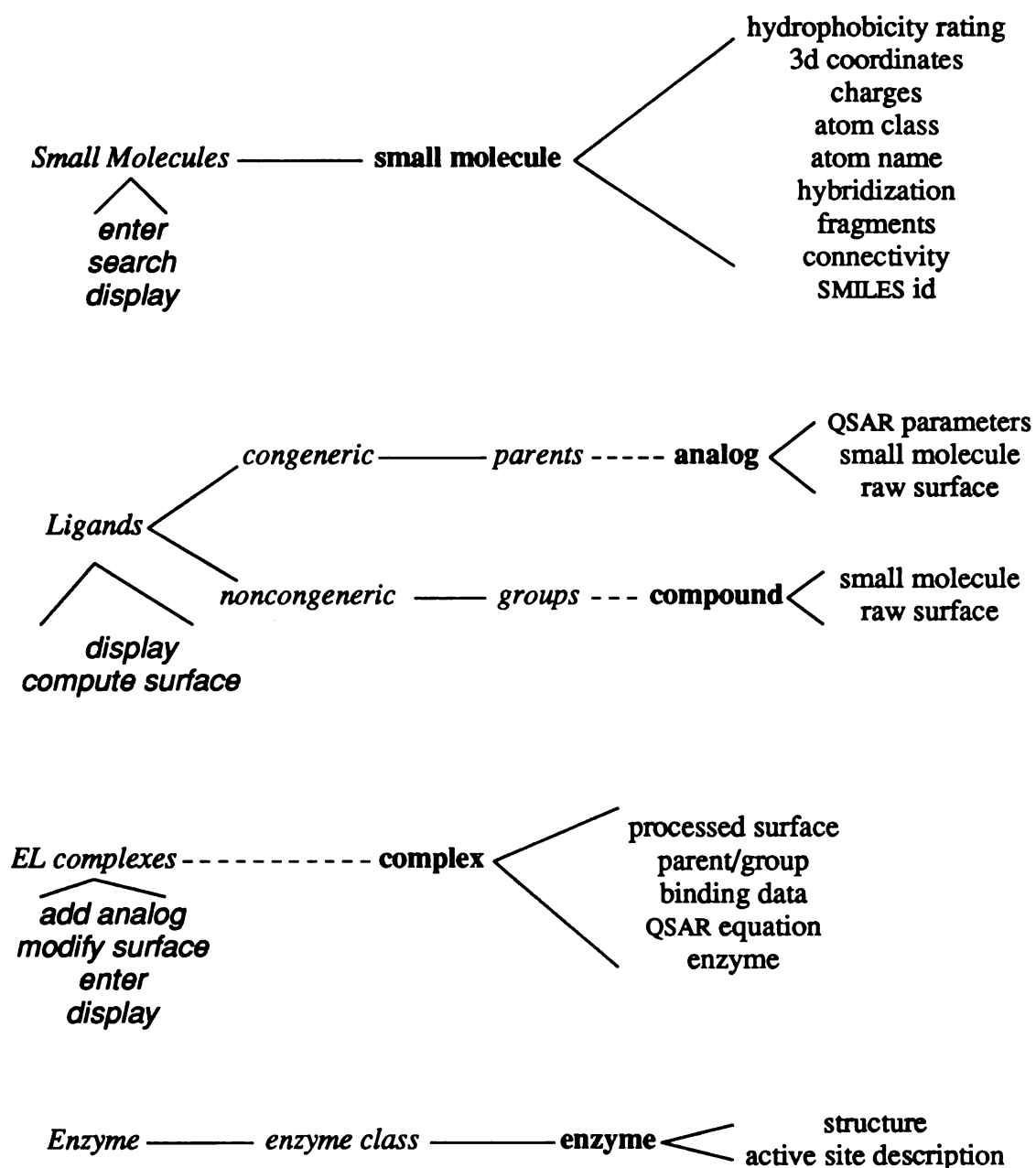


Figure 27. General Knowledge Representation Scheme

The information currently contained in KARMA's knowledge bases is obtained from QSAR, kinetic data and structural chemistry. The combination of QSAR and kinetic data allows for the study of enzyme-ligand interactions. Physicochemical parameters are used to model the effects of structural changes on the electronic, hydrophobic and steric

effects for organic molecules [21]. Examples of physiochemical parameters include, among others: σ , one of several electronic constants based on the Hammett equation for the ionization of substituted benzoic acids; π , the hydrophobic parameter for a chemical substituent based on the octanol-water partition coefficient $\log P$; MR, the molar refractivity, which parameterizes polarizability and steric effects; and Verloop's parameters, which are steric substituent values calculated from bond angles and distances.

The knowledge bases are being developed such that the data will describe interactions for enzyme-ligand binding using QSAR equations and parameters, and structural information of the congener data. If a noncongeneric data set is being used, there is no knowledge available with respect to QSAR equations. The absence of a QSAR equation will limit the use of KARMA's rule system with respect to the overall shape of the receptor model. (see section 4.1.2.1). The interactions for a congener set, with illustrative examples, are shown in Table 1.

Interaction	Example
enzyme → specific enzyme	(DHFR* → chicken DHFR)
congener → specific enzyme	(benzylpyrimidines → chicken DHFR)
congener → specific congener	(inhibitors** → benzylpyrimidines)
substituents → specific congener	(3,4,5 OMe → benzylpyrimidines)
equations → congeners	(equation → benzylpyrimidines)
variable → substituents	(4-Cl → 4-Br)
specific enzyme → specific enzyme	(chicken DHFR → <i>L. casei</i> DHFR)

*DHFR - Dihydrofolate Reductase

**inhibitors - triazines, benzylpyrimidines, etc.

Table 1. Interactions of a Data Set

The two class units *Small Molecules* and *Ligands* pictured in Figure 27 contain chemical information pertaining to chemical elements and molecular substituents. Elemental data includes atom type, atomic radii, hybridization, etc. Substituent data consists of unique identifying codes, physiochemical parameter data and x-ray crystallographic data. For each substituent, where known, there will be values for QSAR parameters such as π , σ or MR. The associated x-ray crystallographic data is used for building the small molecules and is currently obtained either from the Cambridge Crystallographic Data Base [47] or from standard bond lengths and angles. This data is also useful for specifying constraints used in the distance geometry calculations.

The two class units *EL complexes* and *Enzyme* contain information which the user enters, e.g., QSAR equations and congener set, as well as information about previously studied enzyme-ligand binding complexes.

KEE provides the mechanisms including a truth maintenance system to maintain consistency within the knowledge bases. There are additional rules (other than those used for surface characterization) that help maintain accurate knowledge bases. This is important as the number of users of the system increases. Although a user may not be able to alter the *core* data base of KARMA, the user can alter the user-defined data. With a number of different users, it is important that the accuracy of the knowledge base be maintained as the characterization rules operate on all available information.

4.1.2. Rules for Characterization

KARMA's rules are formulated in an (*if*) . . . , (*then*) . . . format (see section 2.2.1.2). For characterization of the surface model, two types of rules (*generic* and *specific*) are being developed. These rules are empirically derived from the results of QSAR as well

as from molecular structure.

4.1.2.1. Generic Rules

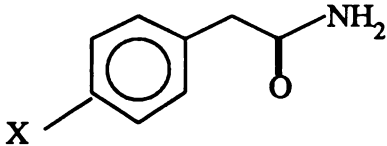
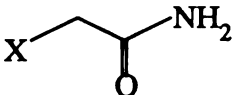
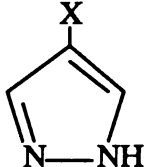
The generic rules govern the the gross morphology of the theoretical receptor surface. These rules are based on QSAR equations and their coefficients. These rules do not define the uneven and non-uniform details of the surface model, but rather, the overall shape. Currently, rules have been empirically described for five categories of receptor shapes [59]: (1) A flat hydrophobic surface where there is a coefficient of approximately 0.5π ; (2) Two parallel surfaces (*i.e.*, two parallel phenyl rings) where there is no π or MR effect seen (*e.g.*, 3, 4 and 5 substituents project into the solvent or there is rotation about a substituted phenyl ring such that the substituents can project into the solvent); (3) A cleft (two parallel surfaces with a bottom) where *meta* substituents that are polar show no effect and *meta* substituents that are apolar show an effect; (4) A cleft with a bottom where steric effects are seen if *para* substituents become large enough; and, (5) a hole (four sides and a bottom) where π affects are seen for 3, 4 and 5 substituents.

Using the above categories, a receptor shape can be crudely looked at as five sides of a box. The shape of the receptor will be refined as the amount of data increases. With limited substituent data, it would be difficult to probe the depth for category (3) or define the enclosed hole for category (5). It is expected that the user will use KARMA iteratively as more data for their system becomes available.

An example of an abstracted rule for category (5) may take the form:

If the coefficient of the hydrophobic parameter is approximately equal to one, then expect complete desolvation about substituent *X* of the ligand.

This rule was derived empirically from work on several species of alcohol dehydrogenase [60] where the following equations were found¹⁶:

Compounds	Enzyme	Equations
	Horse ADH	$\log 1/K_i = 0.89 \log P + 3.56$ $n = 11, r = 0.960, s = 0.197$
	Horse ADH	$\log 1/K_i = 0.98 \log P - 0.83 \sigma^* + 3.69$ $n = 14, r = 0.937, s = 0.280$
	Human ADH	$\log 1/K_i = 0.87 \log P - 2.06\sigma_{meta} + 4.60$ $n = 13, r = 0.977, s = 0.303$
	Rat ADH	$\log 1/K_i = 1.22 \log P - 1.80 \sigma_{meta} + 4.87$
	Horse ADH	$\log 1/K_i = 0.96 \log P + 5.70$ $n = 5, r = 0.990, s = 0.207$

where *X* is the substituent and $\log P$ is based on the octanol-water partition.

The average of the coefficients of the hydrophobic term is approximately equal to one (average = 0.97) suggesting complete desolvation about substituent *X* if it is assumed that the hydrophobic interaction mimics the partitioning of an organic compound between

¹⁶For actual substituent data, please see reference [60].

ocatanol and water. Figure 28 shows complete desolvation by the enzyme alcohol dehydrogenase (hydrophobic space - red; polar space - blue) around substituent X of the pyrazole (green). If the shape of the receptor model was based only on this one rule, the lack of solvent molecules about substituent X would imply that the receptor hole was relatively close in size to the substituent X. However, the site would become more defined in terms of size based on rules that looked at the other compounds (*i.e.*, *specific* rules) in the data set.

Another example of a rule dealing with hydrophobicity may take the form:

If the coefficient of the hydrophobic parameter is greater than 0.5 and less than 1.0, then expect a concave surface about substituent X of the ligand.

This type of rule is empirically based on the enzyme-ligand binding such as that of carbonic anhydrase c and sulfonamides [61] where the following equation was found:

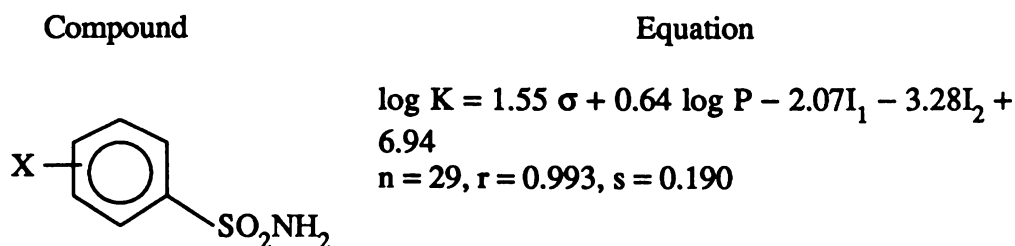


Figure 29 shows how the solvent accessible surface of the enzyme carbonic anhydrase c (hydrophobic space - red; polar space - blue) is slightly concave about the substituent X of the sulfonamide (green).

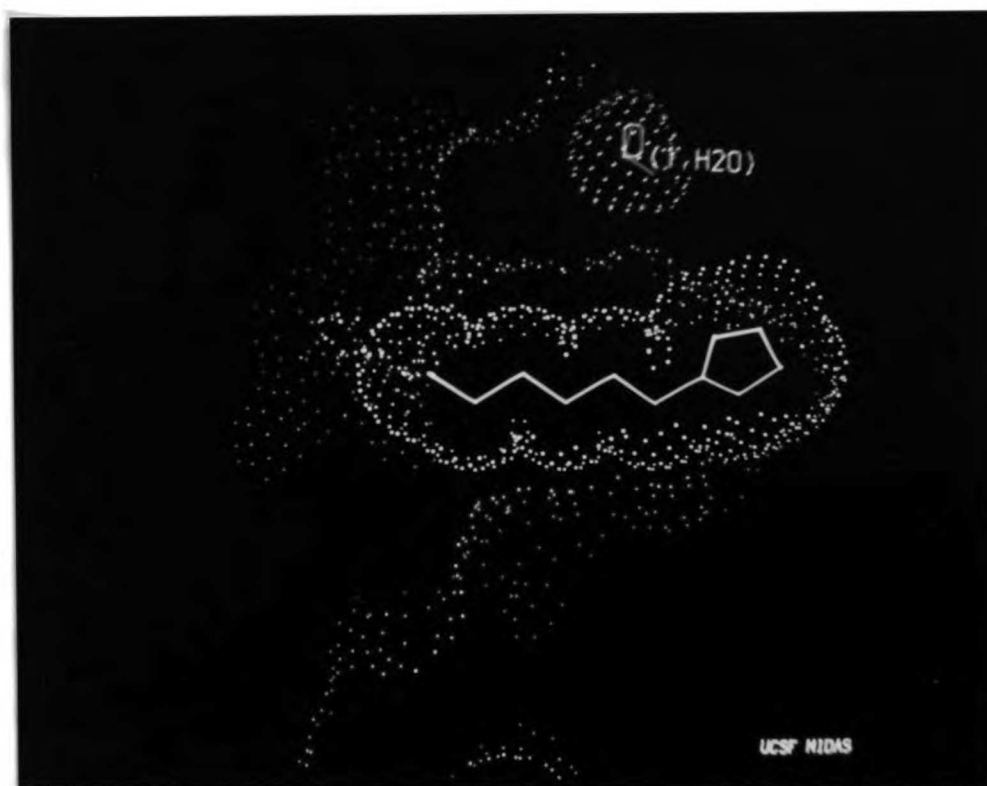


Figure 28. Alcohol Dehydrogenase and a Substituted Pyrazole.

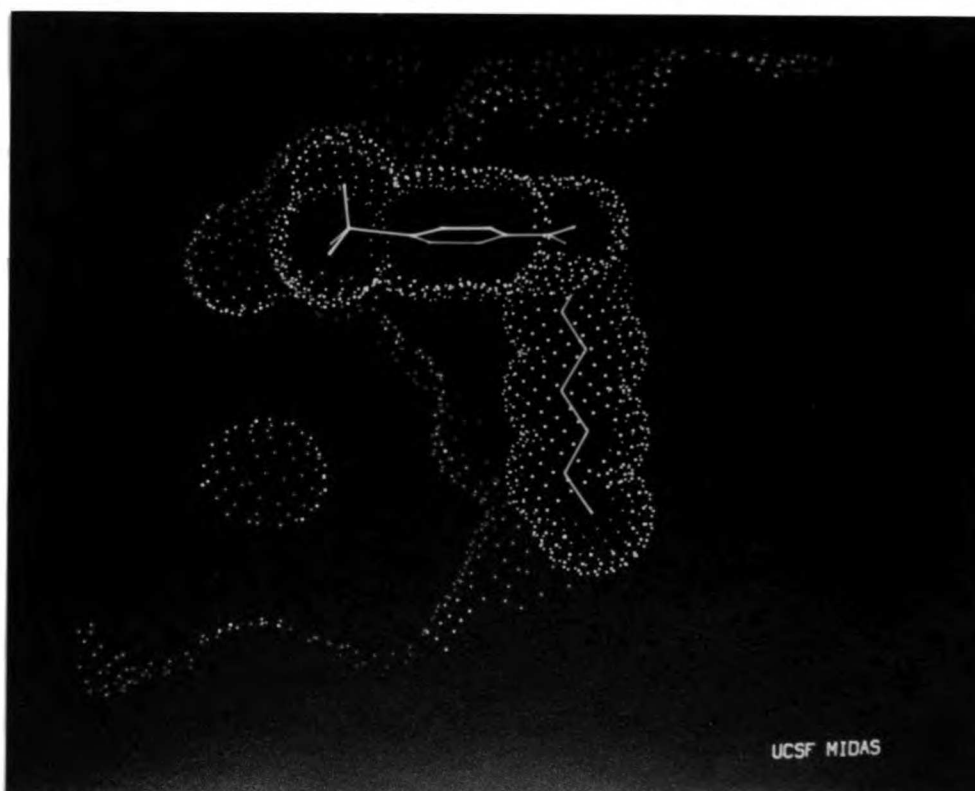


Figure 29. Carbonic Anhydrase C and a Substituted Sulfonamide.

An example of an abstracted rule for category (3) may take the following form if **applied** to a compound with a substituted phenyl ring:

If the substituents of a *meta* disubstituted compound are similar in volume, and the biological activities differ between hydrophobic and polar substituents, then expect possible ring rotation to maximize hydrophobic and polar interactions between the ring substituents and the hydrophobic and polar surface.

4.1.2.2. Specific Rules

Specific rules will yield the detailed shape and character for the receptor model. **These** rules are based on the characteristics of small molecules including the biological **activity** and the molecular structure and not QSAR coefficients. Comparisons based on **biological** activities and, bond lengths and van der Waal's radii may be seen by the **following** abstracted rule:

If the biological activity of substituent X_2 is less than the biological activity of substituent X_1 , and, X_2 is atomically larger than X_1 , then expect possible steric hindrance with the receptor wall about X_2 , provided that other factors are equal.

This example can be exemplified by two compounds that differ by the type of the **substituent**, *i.e.*, a chlorine and a bromine atom. If the binding affinity for the bromine **compound** was lower (and possibly even lower for the iodine compound), it would **suggest** that the wall of the receptor model is contacted by the ligand at the bond distance **of the** chlorine atom and its related van der Waals radius. Therefore, one could assume **that the** larger bromine atom represents an intrusion into the receptor wall.

Both the generic and specific rules are under development. KARMA is designed such **that** the user can create his own rules and add them to KARMA's basic complement of **rules**. It is expected that the user will ultimately add their own rules to KARMA. The **user's** rules will be used in conjunction with those rules provided by KARMA. During **evaluation** of the rules, if a branch point is reached in which the rule system cannot **decide** which path to follow, it is up to the user to decide which path the rule system **should** follow. It is possible to generate multiple receptor models using Keeworlds (see **section 2.2.1**) for the various branch points of the rule system. The decision tree may be **graphically** displayed by KEE as well as explanation of the rule path followed.

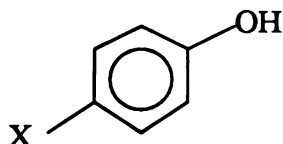
4.2. Testing of KARMA

For a novel system, it is critical that known experimental cases be available. This **allows** for verification of the accuracy of the system by comparing the results of the **system** with known data. Fortunately, numerous enzyme-ligand systems whose three-**dimensional** structures are known are available for testing. The knowledge bases and **rules** are being developed from several classes of enzymes including enzymes from the **serine** proteases (*i.e.*, carbonic anhydrase c) and the dehydrogenases (*i.e.*, alcohol **dehydrogenase**). The known test case for KARMA will be dihydrofolate reductase since **there** is a wealth of ligand data for a variety of enzyme species. Prediction of a receptor **surface** model that approximates the known receptor site for a species of dihydrofolate **reductase** would prove to the scientific community that KARMA is a valuable tool for **medicinal** chemistry research.

In terms of predicting a receptor site whose structure has yet to be solved from x-ray **crystallography**, the enzyme phenol sulfotransferase will be used. This human liver

enzyme is known to catalyze the sulfate conjugation of many phenol and catechol drugs, **xenobiotic** compounds and neurotransmitters [62]. A QSAR equation (Table 2) for **thirty-five** phenolic compounds whose biological activities range over five orders of **magnitude** (see Table 3) has recently been developed for this human liver phenol **sulfotransferase** [62]. Additional data is available for the rat liver phenol **sulfotransferase**. This enzyme-substrate complex is a good test case because of the **reliable** data and QSAR. Comparison of the KARMA model to the x-ray derived structure (**when** available) will prove the effectiveness of KARMA.

Compound:



Equation:

$$\log 1/K_m = 0.99 \log P - 1.48 MR'_4 - 0.64 MR_3 + 1.04 MR_2 + 0.67 \sigma + 4.03$$

$$n = 35, r = 0.950, s = 0.477$$

Table 2. Phenolic Substrate and QSAR Equation

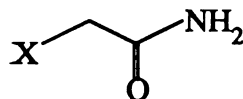
X	observed log 1/K _m	calculated log 1/K _m	log P	σ*	MR ₄	MR ₃	MR ₂
2-Cl	7.41	6.75	2.15	0.27	0.00	0.10	0.60
3,4-Cl ₂	7.05	7.23	3.33	0.64	0.00	0.60	0.10
3-I	6.92	6.18	2.93	0.35	0.00	1.39	0.10
4-CF ₃	6.89	7.21	2.95	0.65	0.00	0.10	0.10
2-CF ₃	6.88	7.49	2.80	0.65	0.00	0.10	0.50
4-Cl	6.64	6.45	2.39	0.27	0.00	0.10	0.10
3-Cl	6.62	6.30	2.50	0.37	0.00	0.60	0.10
3-NO ₂	6.55	5.98	2.00	0.71	0.00	0.74	0.10
3,5-Cl ₂	6.44	7.19	3.62	0.74	0.00	1.20	0.10
2-F	6.24	5.69	1.71	0.05	0.00	0.10	0.10
3-C(CH ₃) ₃	6.16	5.86	3.31	-0.10	0.00	1.96	0.10
4-CN	6.12	6.21	1.60	1.00	0.00	0.10	0.10
4-NO ₂	6.08	5.56	1.91	1.24	0.74	0.10	0.10
2-CH ₃	5.90	6.24	1.95	-0.15	0.00	0.10	0.56
3-CN	5.90	5.67	1.70	0.56	0.00	0.63	0.10
4-CH ₃	5.66	5.75	1.94	-0.15	0.00	0.10	0.10
2-OH	5.40	4.97	0.88	-0.16	0.00	0.10	0.29
3,5-(CH ₃) ₂	5.14	5.49	2.35	-0.14	0.00	1.12	0.10
4-C ₂ H ₅	5.08	4.81	2.58	-0.15	1.03	0.10	0.10
H	4.78	5.41	1.46	0.00	0.00	0.10	0.10
4-OCH ₃	4.44	4.03	1.34	-0.16	0.79	0.10	0.10
4-C ₃ H ₇	4.39	4.69	3.20	-0.15	1.50	0.10	0.10
4-NH ₂	3.86	4.01	0.04	-0.15	0.0	0.10	0.10
4-C(CH ₃) ₃	3.75	4.12	3.31	-0.13	1.96	0.10	0.10
4-CONH ₂	3.51	3.35	0.33	0.63	0.98	0.10	0.10
3-NH ₂	3.51	3.84	0.17	-0.16	0.00	0.54	0.10
4-NHCOCH ₃	2.35	2.33	0.51	0.00	1.49	0.10	0.10
2-OCH ₃	6.77	5.89	1.32	-0.16	0.00	0.10	0.79
2-CN	6.40	6.77	1.61	1.00	0.00	0.10	0.63
2-Br	7.62	7.24	2.35	0.28	0.00	0.10	0.89
3,5-OH ₂	3.48	4.01	0.16	0.12	0.00	0.58	0.10
2-I	7.76	8.04	2.65	0.30	0.00	0.10	1.39
2-CH ₂ OH	4.81	5.39	0.58	0.73	0.00	0.10	0.72
3-CH ₂ OH	4.11	4.13	0.49	0.00	0.00	0.72	0.10
3-NHCOCH ₃	3.65	4.00	0.73	0.21	0.00	1.49	0.10

Table 3. Data for Phenol Substrates

4.3. Current Status of KARMA for the User

KARMA is not currently available for the general user. However, a user could find **KARMA** scientifically helpful even at this early stage of development. Presently, the user **can** obtain and interact with a surface model that is representative of the volume of the **molecules** of interest similar to the Active Analog Approach (see section 1.2.2). The **user** is able to manipulate the surface model to test their hypotheses about a specific **system**.

Initially, the user will bring a data set to be entered into KARMA. An example of a **data** set that user might enter into the system is seen in Table 4.

Compound	Enzyme	Equation
	Horse ADH	$\log 1/K_i = 0.98 \log P - 0.83 \sigma^* + 3.69$ $n = 14, r = 0.937, s = 0.280$

substituent	observed log 1/K _i	calculated log 1/K _i	log P	σ*
F	1.41	1.77	-1.03	1.10
F ₂	1.12	1.24	-0.76	2.05
F ₃	1.33	1.09	-0.11	3.06
Cl	2.12	2.24	-0.59	1.05
Cl ₂	2.19	2.05	-0.03	1.94
Cl ₃	2.07	2.26	0.79	2.65
Br	2.24	2.35	-0.52	1.00
Br ₂	2.49	2.20	0.18	2.00
I	2.72	4.39	0.53	0.85
CH ₂ =	2.11	2.16	-1.01	0.65
CH ₂ =, CH ₃	3.02	2.54	-0.71	0.55
CH ₃ CH=	2.89	2.89	-0.51	0.36
CH ₃ CH=, CH ₃	3.15	3.27	-0.21	0.26
(CH ₃) ₂	3.77	3.46	-0.39	-0.19

Table 4. Sample Data From User [60]

The chemical structures can be entered using the language SMILES which can be mastered by a new user in less than twenty minutes. The two-dimensional display of the chemical structure following keyboard entry takes only a few seconds on the DEC VAX 8650 for small organic molecules. Compounds similar in structure are easily entered by modifying the previously entered structure. Currently, the three-dimensional coordinates for the structure must be obtained separately by the user from either a search of the Cambridge Crystallographic Data Base, the literature, or model building using a program

such as MIDAS [63]. Eventually, KARMA will have an automated system for retrieving **the** three-dimensional coordinates for the user based on x-ray crystallographic data and **standard** geometries. After obtaining the three-dimensional coordinates, the user may **choose** to run the structure through the distance geometry program which only takes a **few** minutes to generate hundreds of conformations on the DEC VAX 8650. The user may **graphically** select the conformations to be used for deriving the surface model on the **Silicon** Graphics IRIS 2400T.

The surface is generated based on the user selected molecules. The generation of **the** triangular net for a molecule of twenty atoms takes approximately three minutes on a **DEC** VAX 8650 with a minimum distance between control points set at 2 angstroms. The **minimum** distance is a user-defined parameter and the time for generating the surface **will** change with this parameter. As the minimum distance is decreased (higher density **surface**), the time for surface generation increases as the square of the distance. The **display** of the patch surface for a molecule of twenty atoms takes approximately one **minute** on the Silicon Graphics IRIS 2400T. The manipulation of the surface is a real-time **process**.

Following display of the surface model, the user can analyze their data and ascribe **characteristics** to the surface by manipulating the shape of the surface or assigning colors **to the** patches which are indicative of certain properties such as hydrophobicity.

4.4. Summary

Current methods in computer-assisted drug design are most successful if the **structure** of the receptor is known. The goal of this research project was to create a new **tool** to aid the investigator in those situations where the structure of the receptor may or

may not be known. KARMA emphasizes two critical factors: First, three-dimensional **graphics** will present the results from the rule-based system in a manageable form. **Second**, KARMA provides a means for the user to inject knowledge about the model. **KARMA** is designed as a tool to aid the chemist and has the ability to incorporate ideas **from** the user which is a very important aspect. Computer-assisted drug design will be **looked** at from a new perspective after KARMA.

Literature Cited

- (1) Buchheim, R., *Arch. Exp. Pathol. Pharmacol.* **1876**, *5*, 261.
- (2) Jolles, G., in "Drug Design: Fact or Fantasy?", Jolles, G. and Wooldridge, K.R.H., eds., Academic Press, Great Britain, **1984**, xii.
- (3) Hopfinger, A., *J. Med. Chem.*, **1985**, *28*, 1133.
- (4) Hansch, C.; Klein, T.E., *Acc. Chem. Res.*, **1986**, *19*, 392.
- (5) Olson, G.L.; Cheung, H.C.; Morgan, K.D.; Bleunt, J.F.; Todaro, L.; Berger, L.; Davidson, A.B.; Boff, E., *J. Med. Chem.*, **1981**, *24*, 1026.
- (6) Boger, J.S., Proceedings from 3rd SCI-RSC Medicinal Chemistry Symposium, September **1985**.
- (7) Fischer, E., *Proc. R. Soc.*, **1894**, *B187*, 397.
- (8) Ehrlich, P., *Chem. Ber.*, **1909**, *42*, 17.
- (9) Koshland, D.E., Jr., *Sci. Amer.*, **1973**, *229*, 52.
- (10) Gund, P., in "Progress in Molecular and Subcellular Biology", Hahn, F.E., ed., Springer-Verlag, New York, **1977**, *5*, 117.
- (11) Gund, P., *Ann. Repts. Med. Chem.*, **1979**, *14*, 299.
- (12) Humblet, C.; Marshall, G.R., *Drug Development Res.*, **1981**, *1*, 409.
- (13) Franke, R., "Theoretical Drug Design Methods", Elsevier, German Democratic Republic, **1984**, *7*, 319.
- (14) For a survey of methods, see: "Drug Design: Fact or Fantasy?", Jolles, G. and Woolridge, K.R.H., eds., Academic Press, Great Britain, **1984**.
- (15) Jurs, P.C.; Stouch, T.R.; Czerwinski, M.; Narvaez, J.N., *J. Chem. Inf. Comput. Sci.*, **1985**, *25*, 296.

- (16) Kuntz, I.D.; Blaney, J.M.; Oatley, S.J.; Langridge, R.; Ferrin, T.E., *J. Mol. Biol.*, **1982**, *161*, 269.
- (17) DesJarlais, R.L.; Sheridan, R.P.; Dixon, J.S.; Kuntz, I.D.; Venkataraghavan, R., *J. Med. Chem.*, **1986**, *29*, 2149.
- (18) Weiner, P.; Kollman, P., *J. Comput. Chem.*, **1981**, *2*, 287.
- (19) Lybrand, T.; McCammon, J.A.; Wipff, G., *Proc. Nat. Acad. Sci. (US)*, **1986**, *83*, 833.
- (20) Martin, Y.C., *J. Med. Chem.*, **1981**, *24*, 229.
- (21) Hansch, C.; Leo, A., "Substituent Constants for Correlation Analysis in Chemistry and Biology", Wiley-Interscience, USA, **1979**.
- (22) Recanatini, M.; Klein, T.; Yang, C.-Z.; McClarin, J.; Langridge, R.; Hansch, C., *Molec. Pharm.*, **1986**, *29*, 436.
- (23) Blaney, J.M.; Jorgensen, E.C.; Connolly, M.L.; Ferrin, T.E.; Langridge, R.; Oatley, S.J.; Burrige, J.M.; Blake, C.C.F., *J. Med. Chem.*, **1982**, *25*, 785.
- (24) Pattabiraman, N.; Levitt, M.; Ferrin, T.; Langridge, R., *J. Comput. Chem.*, **1985**, *6*, 432.
- (25) Goodford, P.J., *J. Med. Chem.*, **1985**, *28*, 849.
- (26) Kollman, P., *Acc. Chem. Res.*, **1985**, *18*, 105.
- (27) Bash, P.A.; Singh, U.C.; Brown, F.K.; Bartlett, P.A.; Langridge, R.; Kollman, P.A., *Science*, **1987**, *235*, 574.
- (28) Simon, Z.; Badileseu, I.; Racovitan, T., *J. Theo. Biol.*, **1977**, *66*, 485.
- (29) Marshall, G.R., in "Medicinal Chemistry VI", Simkin, A., ed., Proceedings from the 6th International Symposium in Medicinal Chemistry, **1978**.
- (30) Crippen, G.M., *J. Med. Chem.*, **1979**, *22*, 988.
- (31) Linschoten, M.R.; Bultsma, T.; IJzerman, A.P.; Timmerman, H., *J. Med. Chem.*, **1986**, *29*, 278.

- (32) Sheridan, R.P., Nilakantan, R., Dixon, J.S.; Venkataraghavan, R., *J. Med. Chem.*, **1986**, *29*, 899.
- (33) Andrews, P.R.; Carson, J.M.; Caselli, A.; Spark, M.J.; Woods, R., *J. Med. Chem.*, **1985**, *28*, 393.
- (34) Moon, D.A., *Computer*, **1987**, *20*, 43.
- (35) Wah, B.J., *Computer*, **1987**, *20*, 10.
- (36) Wah, B.J.; Li, G.J., *ACM SIGART Newsletter*, **1986**, Number 96, 28.
- (37) Hwang, K.; Ghosh, J.; Chowkwanyun, R., *Computer*, **1987**, *20*, 19.
- (38) McCarthy, J., *Communications of the ACM*, **1960**, *7*, 184.
- (39) Steele, G., "Common LISP: The Language", Digital Press, USA, **1984**.
- (40) Symbolics Technical Summary #990098, Cambridge, Massachusetts, **1985**.
- (41) Newman, W.M.; Sproull, R.F., "Principles of Interactive Computer Graphics. Second Edition", McGraw-Hill, Inc., USA, **1979**.
- (42) Bartlett, J., "Familiar Quotations. Fifteenth Edition", Little, Brown & Company, USA, **1980**, 132.
- (43) Foley, J.D.; Van Dam, A., "Fundamentals of Interactive Computer Graphics", Addison-Wesley Publishing Company, USA, **1982**.
- (44) Sun Microsystems, Inc., Promotional Literature for the Sun-3 Family, **1986**.
- (45) Section 2.2.1 **Introduction to KEE** contains information obtained from the KEE Software Development System Core Reference Manual (July 21, **1986**) and User's Manual (July 25, **1986**).
- (46) Section 2.2.2 **Pomona MedChem Software** contains information obtained from the MedChem Software Manual. Release 3.32 (December **1984**), Medicinal Chemistry Project, Pomona College, Claremont, California, 91711.
- (47) Allen, F.H.; Kennard, O.; Motherwell, W.D.S.; Town, W.G.; Watson, D.G., *J. Chem. Doc.*, **1973**, *13*, 119.

- (48) Connolly, M.L., *Science*, **1983**, *221*, 709.
- (49) Bash, P.A.; Pattabiraman, N.; Huang, C.; Ferrin, T.E.; Langridge, R., *Science*, **1983**, *222*, 1325.
- (50) Mortenson, M.E., "Geometric Modeling", John Wiley & Sons, USA, **1985**.
- (51) Clark, J.; "Parametric Curves, Surfaces, and Volumes in Computer Graphics and Computer Aided Geometric Design", Technical Report No. 221, Computer Systems Laboratory, Stanford University, **1981**.
- (52) Chiyokura, H.; Kimura, F., *Computer Graphics (Siggraph '83 Conf. Proc.)*, **1983**, *17*, 289.
- (53) Chiyokura, H., *Proc. Computer Graphics Conf.*, Tokoyo, **1986**.
- (54) Séquin, C., *Proc. Third USENIX Computer Graphics Workshop*, **1986**, 63.
- (55) Longhi, L., "Interpolating Patches Between Cubic Boundaries", Master's Project Report, Computer Science Division, University of California, Berkeley, California, **1985**.
- (56) Shirman, L., "Symmetric Interpolation of Triangular and Quadrilateral Patches Between Cubic Boundaries", Master's Project Report, Computer Science Division, University of California, Berkeley, California, **1986**.
- (57) Gregory, J.A., "Computer Aided Geometric Design", Barnhill, R.E. and Riesenfeld, R.F., eds., Academic Press, New York, **1974**, 71.
- (58) Faux, I.D.; Pratt, M.J., "Computational Geometry for Design and Manufacture", Ellis Horwood, Ltd., Great Britain, **1979**, 132.
- (59) Hansch, C., private communications.
- (60) Hansch, C.; Klein, T.; McClarin, J.; Langridge, R.; Cornell, N., *J. Med. Chem.*, **1986**, *29*, 615.
- (61) Hansch, C.; McClarin, J.; Klein, T.; Langridge, R., *Molec. Pharm*, **1985**, *27*, 493.
- (62) Campbell, N.R.C.; Van Loon, J.A.; Sundaram, R.S.; Ames, M.M.; Hansch, C.; Weinshilboum, R., *Molec. Pharm.* (submitted).

- (63) Huang, C.; Jarvis, L.; Ferrin, T.E.; Langridge, R., UCSF MIDAS: Molecular Interactive and Display, 1983.

Appendix One: Program Listings for Triangulated Surface

PLEASE NOTE:

Copyrighted materials in this document have not been filmed at the request of the author. They are available for consultation, however, in the author's university library.

These consist of pages:

P. 97-177

University
Microfilms
International

300 N. ZEEB RD., ANN ARBOR, MI 48106 (313) 761-4700

center.c

```

/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
#include <stdio.h>
#include <math.h>
#include <pdb.h>
#include "center.h"

/*
 * read_pdb:
 * Read the given pdb file and put all the atoms into the
 * unused list.
 */
CENTER *
read_pdb(fp)
FILE *fp;
{
    register CENTER *cp;
    CENTER *center;
    pdb_record r;
    extern char *emalloc();

    center = NULL;
    do {
        r = read_pdb_record(fp);
        switch(r.record_type) {
            case R_END:
            case R_UNKNOWN:
            case R_TER:
                break;
            case R_ATOM:
                cp = (CENTER *) emalloc(sizeof (CENTER));
                cp->coord[0] = r.pdb.atom.x;
                cp->coord[1] = r.pdb.atom.y;
                cp->coord[2] = r.pdb.atom.z;
                cp->next = center;
                center = cp;
                break;
            default:
                fputs("Ignored bad atom record\n", stderr);
                break;
        }
    } while (r.record_type != R_END);
    return center;
}

```

```

/*
 * compute_inter:
 * Compute the interaction parameters of centers
 */
INTER **
compute_inter(center, radius, distance)
CENTER *center;
double radius, distance;
{
    register CENTER *cp;
    register int ncenter, i;
    double dsq, dist, rprime, diamsq;
    INTER **inter;
    extern char *emalloc();
    double compute_rprime();

    diamsq = radius * radius * 4;
    ncenter = 0;
    for (cp = center; cp != NULL; cp = cp->next)
        cp->ident = ncenter++;
    inter = (INTER **) emalloc(sizeof (INTER *) * ncenter);
    for (i = 0; i < ncenter; i++)
        inter[i] = (INTER *) emalloc(sizeof (INTER) * ncenter);
    while (center != NULL) {
        inter[center->ident][center->ident].dsq = -1;
        for (cp = center->next; cp != NULL; cp = cp->next) {
            dsq = 0;
            for (i = 0; i < 3; i++) {
                delta = center->coord[i] - cp->coord[i];
                dsq += delta * delta;
            }
            if (dsq > diamsq) {
                dsq = -1;
                rprime = 0;
            } else {
                dist = sqrt(dsq);
                rprime = compute_rprime(radius, dist, distance);
            }
            inter[center->ident][cp->ident].dsq = dsq;
            inter[center->ident][cp->ident].rdsq = rprime;
            inter[cp->ident][center->ident].dsq = dsq;
            inter[cp->ident][center->ident].rdsq = rprime;
        }
        center = center->next;
    }
}

```

```

center.c
}
return inter;
}

/* compute rprime:
 * Compute the effective radius for clipping for two connected
 * spheres
 */
double
compute_rprime(radius, dist, length)
double radius, dist, length;
{
    double radsq;
    double x, y, xflen, ysq;
    double xp, yp;
    double t;

    radsq = radius * radius;
    t = radius + length;
    x = (radsq + dist * dist - t * t) / (2 * dist);
    ysq = radsq - x * x;
    if (ysq < 0)
        return radsq;
    y = sqrt(ysq);
    xflen = hypot(x - dist, y);
    xp = (x - dist) / xflen * radius + dist;
    yp = y / xflen * radius;
    t = xp * xp + yp * yp;
    return radsq > t ? radsq : t;
}

```

center.h

```
/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
typedef struct center_def
struct center_def
double
int
} CENTER;

typedef struct inter_def
double disq;
double radsq;
} INTER;

extern CENTER *read_pdb();
extern INTER **compute_inter();
```

```

point.c
/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
#include <stdio.h>
#include <math.h>
#include "center.h"
#include "point.h"

#define PI 3.1415926

/*
 * alloc_point:
 * Allocate a point structure
 */
POINT *
alloc_point(coord, owner)
double coord[3];
CENTER *owner;
{
    register POINT *pp;
    register int i;
    extern char *emalloc();

    pp = (POINT *) emalloc(sizeof (POINT));
    pp->mode = M_UNKNOWN;
    pp->nconn = 0;
    pp->mark = 0;
    pp->diabel = 0;
    pp->maxdelta = 0;
    pp->owner = owner;
    pp->conn = NULL;
    for (i = 0; i < 3; i++)
        pp->coord[i] = coord[i];
    return pp;
}

/*
 * add_conn:
 * Add a connection to this point
 */
POINT *pp;
EDGE *edge;
POINT *point;

register CONN *cp;

cp = (CONN *) emalloc(sizeof (CONN));
cp->edge = edge;
cp->point = point;
cp->status = S_CHILD;
cp->next = pp->conn;
pp->conn = cp;
pp->nconn++;
}

/*
 * sphere:
 * Generate a list of points on a sphere
 */
POINT *
sphere(radius, min_dist)
double radius;
double min_dist;
{
    register POINT *pp;
    register int layer, i;
    register int nlayer, npoint;
    POINT *point;
    double theta, phi;
    double rad;
    double circumference;
    double coord[3];

    point = NULL;
    theta = min_dist / radius;
    nlayer = (PI / theta) + 1;
    for (layer = 0; layer < nlayer; layer++) {
        theta = layer * (PI / (nlayer - 1));
        coord[2] = radius * cos(theta);
        rad = radius * sin(theta);
        circumference = 2 * PI * rad;
        npoint = circumference / min_dist;
        for (i = 0; i < npoint; i++) {
            phi = i * (2 * PI / npoint);
            coord[0] = rad * sin(phi);
            coord[1] = rad * cos(phi);
            pp = alloc_point(coord, (CENTER *) NULL);
            pp->next = point;
            point = pp;
        }
    }
}

```

```
point.c  
}  
return point;  
}
```

point.h

```
/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
typedef struct point_def
struct point_def
char
char
char
int
double
double
CENTER
struct conn_def
POINT;
}

typedef struct edge_def
struct edge_def
POINT
short
short
int
EDGE;
}

typedef struct conn_def
struct conn_def
POINT
EDGE
int
CONN;
}

#define M_UNKNOWN 0
#define M_DEAD 1
#define M_SAVE 2
#define M_CHECK 3

#define S_CHILD 1
#define S_PARENT 2

extern POINT *alloc_point();
extern POINT *sphere();
```

surface.c

```

/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
#include <stdio.h>
#include "center.h"
#include "point.h"

#define KEEP 1
#define DELETE 2

/*
 * surface:
 *   Generate a set of points that lie on the intersection
 *   of spheres at the given centers
 */
POINT*
surface(radius, distance, atom, proto, inter)
double radius;
double distance;
CENTER *atom;
POINT *proto;
INTER **inter;
{
    register POINT *pp;
    register CENTER *cp, *center;
    register int i;
    register POINT *surf, *newpp;
    double delta, dsq, radsq;

    surf = NULL;
    radius = radius + distance / 8;
    radsq = radius * radius;
    for (center = atom; center != NULL; center = center->next) {
        /*
         * Mark all points on the proto sphere as good
         */
        for (pp = proto; pp != NULL; pp = pp->next) {
            pp->mode = M_SAVE;
            for (i = 0; i < 3; i++)
                pp->coord[i] += center->coord[i];
        }

        /*
         * Now mark all the bad ones
         */
        for (cp = atom; cp != NULL; cp = cp->next) {
            /*
             * If the spheres do not intersect, then
             * we don't need to check points individually
             */
            if (inter[center->ident][cp->ident].dsq < 0)
                continue;

            /*
             * Figure out whether we drop this point or not
             */
            radsq = inter[center->ident][cp->ident].radsq;
            for (pp = proto; pp != NULL; pp = pp->next) {
                if (pp->mode != M_SAVE)
                    continue;
                delta = pp->coord[0] - cp->coord[0];
                dsq = delta * delta;
                for (i = 1; i < 3 && dsq < radsq; i++) {
                    delta = pp->coord[i] - cp->coord[i];
                    dsq += delta * delta;
                }
                if (dsq < radsq)
                    pp->mode = M_DEAD;
            }
        }

        /*
         * Now we can save the good ones
         */
        for (pp = proto; pp != NULL; pp = pp->next) {
            if (pp->mode == M_SAVE) {
                newpp = alloc_point(pp->coord, center);
                newpp->next = surf;
                surf = newpp;
            }
            for (i = 0; i < 3; i++)
                pp->coord[i] -= center->coord[i];
        }
        return surf;
    }

    /*
     * chooses:
     *   Choose the set of points that will become the vertices of
     *   triangles
     */
}

```


surface.c

POINT *

~~choose(point, distance, min_dist)~~

```
POINT **point;
double distance, min_dist;
{
    register POINT *pp;
    register int i;
    register POINT *save, *saveprev, *prev, *next;
    register POINT *live, *dead, *good;
    double coord;
    double delta, dsq;
    double maxdsq, mindsq;

    live = *point;
    dead = NULL;
    good = NULL;
    saveprev = NULL;
    save = live;
    maxdsq = (distance + min_dist) * (distance + min_dist);
    mindsq = (distance - min_dist) * (distance - min_dist);
    while (save != NULL) {
        /*
         * Move the point we're saving from the live list to
         * the good list
         */
        If (saveprev == NULL)
            live = save->next;
        else
            saveprev->next = save->next;
        save->next = good;
        good = save;
        save->mode = M_SAVE;
        coord = save->coord;

        /*
         * Now eliminate points and mark potential ones
         */
        save = NULL;
        prev = NULL;
        for (pp = live; pp != NULL; pp = next) {
            next = pp->next;

            /*
             * Compute the distance
             */
            delta = coord[0] - pp->coord[0];
            dsq = delta * delta;

```

```

        for (i = 1; i < 3 && dsq < maxdsq; i++) {
            delta = coord[i] - pp->coord[i];
            dsq += delta * delta;
        }

        /*
         * If the point is too close, move it to the
         * dead list
         */
        If (dsq < mindsq) {
            if (prev == NULL)
                live = pp->next;
            else
                prev->next = pp->next;
            pp->next = dead;
            dead = pp;
            continue;
        }

        /*
         * If the point falls within the band of
         * acceptance, mark it
         */
        If (dsq < maxdsq) {
            pp->mode = M_CHECK;
            if (pp->nconn++ == 0 || pp->maxdelta < dsq)
                pp->maxdelta = dsq;
        }

        /*
         * Check whether this is the optimal point
         * to work on for the next iteration
         */
        If (pp->mode != M_CHECK) {
            prev = pp;
            continue;
        }
        If (save == NULL || save->nconn < pp->nconn) {
            save = pp;
            saveprev = prev;
            prev = pp;
            continue;
        }
        If (save->nconn > pp->nconn) {
            prev = pp;
            continue;
        }
    }
}

```

surface.c

```
if (save->maxdelta > pp->maxdelta) {
    save = pp;
    saveprev = prev;
}
prev = pp;
}
}
*point = dead;
return good;
}
}
```

surface.h

```
/*  
 * Copyright (c) 1987 by the Regents of the University of California  
 * All rights reserved.  
 */  
extern POINT *surface();  
extern POINT *choose();
```

triangle.c

```

/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
#include <stdio.h>
#include <math.h>
#include "center.h"
#include "point.h"
#include "triangle.h"

#define TRUE 1
#define FALSE 0
#endif

#define MAXVALUE 7

typedef struct path_def
{
    struct path_def *next;
    POINT point;
    EDGE edge;
    PATH;
}

typedef struct polygon_def
{
    struct polygon_def *next;
    *path;
    int inside;
} POLYGON;

/*
 * form_triangle:
 * Form triangles from the chosen points
 */
TRIANGLE *
form_triangle(live, dead, distance, min_dist, max_ratio, inter, elist)
POINT **live, **dead;
double distance, min_dist;
double max_ratio;
INTER **inter;
EDGE **elist;
{
    EDGE *edge;
    TRIANGLE *triangle;
    POLYGON *polygon;
    EDGE *form_edges();
    *find_triangle();
}

/*
 * find_polygon():
 * convex_polygon():
 * small_polygon():
 * triangulate():
 * merge_polygon():
 */
/*
 * Find the edges first
 */
edge = form_edges(*live, distance, min_dist, inter);

/*
 * Now find the triangles. Multiple sided polygons are
 * ignored.
 */
triangle = find_triangle(*live);

/*
 * find other polygons, split them, and add them to the
 * list of triangles
 */
polygon = find_polygon(*live, edge);
print_edges(*live, triangle, polygon);

/*
 * Make sure polygons are convex. If any is not, break it
 * down until the parts are
 */
polygon = convex_polygon(polygon, &edge);
polygon = small_polygon(polygon, distance * max_ratio, &edge);
polygon = triangulate(polygon, live, dead, &edge);

/*
 * Now convert all polygons into triangles and merge the lists
 */
triangle = merge_polygon(polygon, triangle);

*elist = edge;
return triangle;
}

/*
 * form_edges:
 * Form obvious edges
 */
EDGE *
form_edges(point, distance, min_dist, inter)

```

triangle.c

```

POINT *point;
double distance, min_dist;
INTER **inter;
{
    register POINT *pp;
    register int i;
    register EDGE *edge;
    register CENTER *cp1, *cp2;
    double delta, dsq;
    double maxdsq;
    extern char *emalloc();
    EDGE *add_edge();

    for (pp = point; pp != NULL; pp = pp->next)
        pp->nconn = 0;
    edge = NULL;
    maxdsq = (distance + min_dist) * (distance + min_dist);
    while (point != NULL) {
        cp1 = point->owner;
        for (pp = point->next; pp != NULL; pp = pp->next) {
            /* If the spheres don't interact,
             * don't connect them
             */
            cp2 = pp->owner;
            if (cp1 != cp2 && inter[cp1->ident][cp2->ident].dsq < 0)
                continue;

            /* If the points are too far apart,
             * don't connect them
             */
            dsq = 0;
            for (i = 0; i < 3; i++) {
                delta = pp->coord[i] - point->coord[i];
                dsq += delta * delta;
            }
            if (dsq > maxdsq)
                continue;

            /* This is it. Connect them up
             */
            edge = add_edge(point, pp, edge);
        }
        point = point->next;
    }

    return edge;
}

/* *add_edge:
 * Add an edge between two points
 */
EDGE *
add_edge(point, pp, edge)
POINT *point, *pp;
EDGE *edge;
{
    register EDGE *ep;

    ep = (EDGE *) emalloc(sizeof(EDGE));
    ep->p1 = point;
    ep->p2 = pp;
    ep->count = 0;
    ep->skip = 0;
    ep->clabel = 0;
    ep->next = edge;
    add_conn(point, ep, pp);
    add_conn(pp, ep, point);
    return ep;
}

/* *find_conn:
 * Find the connection pointer between two points
 */
CONN *
find_conn(p1, p2)
register POINT *p1, *p2;
{
    register CONN *cp;

    for (cp = p1->conn; cp != NULL; cp = cp->next)
        if (cp->point == p2)
            return cp;

    return NULL;
}

/* *find_triangle:
 * Find all the triangles that are in the surface already
 */
TRIANGLE *

```

triangle.c

```

find_triangle(point)
register POINT *point;
{
    register CONN
    register POINT
    TRIANGLE
    extern char

    triangle = NULL;
    while (point != NULL) {
        for (cp = point->conn; cp != NULL; cp = cp->next) {
            if (cp->status == S_PARENT)
                continue;
            pp = cp->point;
            for (ocp = pp->conn; ocp != NULL; ocp = ocp->next) {
                if (ocp->point == point) {
                    ocp->status = S_PARENT;
                    break;
                }
            }
            for (ocp = point->conn; ocp != NULL; ocp = ocp->next) {
                if (ocp->status == S_PARENT)
                    continue;
                if ((tcp = find_conn(pp, ocp->point)) == NULL)
                    continue;
                if (add_triangle(&triangle, point, pp,
                    ocp->point) == 0) {
                    cp->edge->count++;
                    ocp->edge->count++;
                    tcp->edge->count++;
                }
            }
            point = point->next;
        }
        return triangle;
    }
}

/*
 * add_triangle:
 *   Add a triangle if it does not already exist
 */
add_triangle(tp, p1, p2, p3)
TRIANGLE **tpp;
POINT *p1, *p2, *p3;
{
    register TRIANGLE *tp;
    register POINT *tmp;

    #define swap(a, b) tmp = a, a = b, b = tmp

    /* Sort the points by address
    */
    if (p1 > p2)
        swap(p1, p2);
    if (p1 > p3)
        swap(p1, p3);
    if (p2 > p3)
        swap(p2, p3);

    for (tp = *tpp; tp != NULL; tp = tp->next)
        if (tp->vertex[0] == p1 && tp->vertex[1] == p2
            && tp->vertex[2] == p3)
            return -1;

    tp = (TRIANGLE *) malloc(sizeof (TRIANGLE));
    tp->vertex[0] = p1;
    tp->vertex[1] = p2;
    tp->vertex[2] = p3;
    tp->next = *tpp;
    *tpp = tp;
    return 0;
}

static int max_value;
static PATH *best_path;
static PATH *current_path;

/* find_polygon:
 *   Find polygons using alpha-pruning
 */
POLYGON *
find_polygon(point, edge)
POINT *point;
EDGE *edge;
{
    register EDGE *ep;
    register int curmax;
    register int again;
    POLYGON *polygon;
    extern char *malloc();
    PATH *find_path();

    polygon = NULL;
}

```

triangle.c

```

again = 1;
for (curmax = MAXVALUE; again; curmax += MAXVALUE) {
    get_polygons(point, edge, &polygon, curmax);
    again = 0;
    for (ep = edge; ep != NULL; ep = ep->next)
        if (ep->count < 2) {
            ep->skip = 0;
            again++;
        }
    if (again < curmax)
        break;
}
return polygon;
}

get_polygons(point, edge, ppp, max)
POINT *point;
EDGE *edge;
POLYGON* ppp;
int
max;
{
    register EDGE *ep, *uep;
    register PATH *pathp;
    POLYGON *polyp;
    extern char *emalloc();
    PATH *find_path();

    for (;;) {
        /* Find the edge to start work on
        */
        uep = NULL;
        for (ep = edge; ep != NULL; ep = ep->next) {
            if (ep->count >= 2 || ep->skip)
                continue;
            if (uep == NULL || ep->count > uep->count)
                uep = ep;
        }
        if (uep == NULL)
            break;

        /* Set things up and get back a path
        */
        max_value = max;
        current_path = (PATH *) emalloc(sizeof (PATH));
        current_path->point = uep->p1;

        current_path->edge = uep;
        current_path->next = NULL;
        find_path(uep->p1, uep->p2, 0, current_path, point);
        free_path(current_path);
        current_path = NULL;
        if (best_path == NULL) {
            uep->skip = 1;
            continue;
        }
        /* We found something!
        */
        polyp = (POLYGON *) emalloc(sizeof (POLYGON));
        polyp->path = best_path;
        polyp->nside = max_value + 1;
        polyp->next = *ppp;
        *ppp = polyp;
        for (pathp = best_path; pathp != NULL; pathp = pathp->next)
            pathp->edge->count++;
        best_path = NULL;
    }
}

/* find_path:
 * Find a path to the target after appending the
 * next node
 */
PATH *
find_path(start, target, curval, path_end, point)
POINT *start, *target;
int curval;
PATH *path_end;
POINT *point;
{
    register CONN *cp;
    extern char *emalloc();

    if (curval >= max_value)
        return;

    /* If this is a path to the target, it must be better than
    * the current best path
    */
    if (start == target) {

```

triangle.c

```

/*
 * If we found a polygon with more than 3 edges
 * and it is canonical, save it
 */
if (curval > 2 && canonical(current_path, point)) {
    save_path();
    max_value = curval;
}
return;
}
path_end->next = (PATH *) emalloc(sizeof (PATH));
path_end->next->next = NULL;
start->mark = 1;
for (cp = start->conn; cp != NULL; cp = cp->next) {
    if (cp->edge->count >= 2)
        continue;
    if (cp->point->mark)
        continue;
    path_end->next->point = cp->point;
    path_end->next->edge = cp->edge;
    find_path(cp->point, target, curval + 1, path_end->next, point);
}
free((char *) path_end->next);
path_end->next = NULL;
start->mark = 0;
}

/* free_path:
 * Free path structure
 */
free_path(pp)
register PATH *pp;
{
    register PATH *nextpp;

    while (pp != NULL) {
        nextpp = pp->next;
        free((char *) pp);
        pp = nextpp;
    }
}

/* label_surface:
 * Label this point and all edges with the given label and
 * all connected points
 */
label_surface(pp, label)
POINT *pp;
int label;
{
    register CONN *cp;

    if (pp->clabel == label)
        return;
    pp->clabel = label;
    for (cp = pp->conn; cp != NULL; cp = cp->next)
        cp->edge->clabel = label;
    for (cp = pp->conn; cp != NULL; cp = cp->next)
        label_surface(cp->point, label);
}

/* canonical:
 * Determine whether the current path describes
 * a canonical polygon (one which cannot be decomposed into
 * smaller polygons)
 */
canonical(path, point)
PATH *path;
POINT *point;
{
    static int clabel = 0;
    register PATH *pathp;
    register POINT *pp;
    register CONN *cp;

    /* Generate a unique label (let's hope there are fewer than
     * MAXINT verification requests
     */
    clabel++;

    /* Label the nodes and edges of the proposed polygon.
     * This cuts the surface into two pieces.
     */
    for (pathp = path; pathp != NULL; pathp = pathp->next) {
        pathp->point->clabel = clabel;
        pathp->edge->clabel = clabel;
    }
}

```


triangle.c

```

* Find a point which is not part of the polygon
*/
for (pp = point; pp != NULL; pp = pp->next)
    if (pp->clabel != clabel)
        break;
if (pp == NULL)
    /*
    * Wow, a polygon that covers the whole surface!
    */
    return FALSE;

/*
* Label the surface which is connected to this point.
*/
label_surface(pp, clabel);

/*
* Now we check if there are unlabeled edges next to the
* proposed polygon. If there are, then the polygon is
* not canonical because there must be smaller polygons
* that may be formed using the unlabeled edges.
*/
for (pathp = path; pathp != NULL; pathp = pathp->next)
    for (cp = pathp->point->conn; cp != NULL; cp = cp->next)
        if (cp->edge->clabel != clabel)
            return FALSE;

return TRUE;

}

/*
* save_path:
* Save the current path as the best path
*/
save_path()
{
    register PATH *pp, *newpp;

    if (best_path != NULL)
        free_path(best_path);
    newpp = NULL;
    for (pp = current_path; pp != NULL; pp = pp->next) {
        if (newpp == NULL) {
            newpp = (PATH *) malloc(sizeof (PATH));
            best_path = newpp;
        }
        else {
            newpp->next = (PATH *) malloc(sizeof (PATH));

```

```

        newpp = newpp->next;
    }
    newpp->point = pp->point;
    newpp->edge = pp->edge;
}
newpp->next = NULL;
}

```

```

/*
* convex_polygon:
* Break concave polygons into smaller convex ones
*/

```

```

POLYGON *
convex_polygon(polygon, edge)
POLYGON *polygon;
EDGE **edge;
{

```

```

    register POLYGON
    register PATH
    PATH
    double
    PATH
    double
    PATH
    POLYGON
    double
    *polyp;
    *pp, *tp;
    *p1, *p2, *p3;
    *dsq1, *dsq2, *dsq3;
    *savep1, *savep2;
    mindsq;
    *lastp;
    *convex;
    distsq();

```

```

    convex = NULL;
    while (polygon != NULL) {

```

```

        /*
        * We know that "our" quadrilaterals cannot be concave so we
        * move it to the convex list and go to the next polygon
        */

```

```

        if (polygon->nside <= 4) {
            polyp = polygon;
            polygon = polyp->next;
            polyp->next = convex;
            convex = polyp;
            continue;
        }
    }
}

```

```

/*
* Try to find a concave point
* A concave point is defined as a point which is close to
* some other point on the polygon than both its neighbors
* First we make the polygon a circular list to make life
* easier when checking for concave points, then we check,

```

triangle.c

```

* and finally we put the polygon back to original form.
*/
for (pp = polygon->path; pp->next != NULL; pp = pp->next)
    continue;
lastp = pp;
pp->next = polygon->path;
savep1 = NULL;
mindseq = 0;
savep2 = NULL;
do {
    /* Identify the two neighbors (p1, p3) and the
    * actual point (p2). Then loop over all other
    * points on the polygon to find the one point that
    * is closest to p2 while being closer to p2
    * than to both p1 *and* p3.
    */
    p1 = pp;
    p2 = p1->next;
    p3 = p2->next;
    for (tp = p3->next; tp != p1; tp = tp->next) {
        dsq1 = distsq(p1->point, tp->point);
        dsq2 = distsq(p2->point, tp->point);
        dsq3 = distsq(p3->point, tp->point);
        if (dsq2 > dsq1 || dsq2 > dsq3)
            continue;
        if (savep1 == NULL || dsq2 < mindseq) {
            savep1 = p2;
            savep2 = tp;
            mindseq = dsq2;
        }
    }
    pp = pp->next;
} while (pp != polygon->path);
lastp->next = NULL;
/*
* If there is no concave point,
* then the polygon must be convex
*/
if (savep1 == NULL) {
    polyg = polygon;
    polygon = polyg->next;
    polyg->next = convex;
}
convex = polyg;
continue;
}
/*
* Now we have to break the polygon into two smaller polygons
* by forming an edge between savep1 and savep2
* savep1 and savep2 will both be the head of polygon paths
*/
break_polygon(&polygon, savep1, savep2, edge);
}
return convex;
}
/*
* distsq: Compute the square of the distance between two points
*/
double
distsq(p1, p2)
POINT *p1, *p2;
{
    double dx, dy, dz;
    dx = p1->coord[0] - p2->coord[0];
    dy = p1->coord[1] - p2->coord[1];
    dz = p1->coord[2] - p2->coord[2];
    return dx * dx + dy * dy + dz * dz;
}
/*
* break_polygon:
* Break the given polygon by adding an edge between
* the two path components. We do this by first closing
* the polygon. Then we find the two path components which
* meet at the two given components. We then add two new
* path components as the polygon path terminators. Finally,
* we create a new polygon and adjust the old one.
*/
break_polygon(ppp, p1, p2, edge)
POLYGON* ppp;
PATH *p1, *p2;
EDGE **edge;
{
    register PATH *pp, *tp;
    register PATH *lastp;
    POLYGON *polyg;
}

```

triangle.c

```

for (pp = (*ppp)->path; pp->next != NULL; pp = pp->next)
    continue;
lastp = pp;
lastp->next = (*ppp)->path;

for (pp = lastp; pp->next != p1; pp = pp->next)
    continue;
tp = (PATH *) malloc(sizeof (PATH));
tp->point = p1->point;
tp->edge = p1->edge;
tp->next = NULL;
pp->next = tp;

for (pp = p1; pp->next != p2; pp = pp->next)
    continue;
tp = (PATH *) malloc(sizeof (PATH));
tp->point = p2->point;
tp->edge = p2->edge;
tp->next = NULL;
pp->next = tp;

*edge = add_edge(p1->point, p2->point, *edge);
p1->edge = *edge;
p2->edge = *edge;

(*ppp)->path = p2;
(*ppp)->nside = 0;
for (pp = p2; pp != NULL; pp = pp->next)
    (*ppp)->nside++;

polyp = (POLYGON *) malloc(sizeof (POLYGON));
polyp->path = p1;
polyp->nside = 0;
for (pp = p1; pp != NULL; pp = pp->next)
    polyp->nside++;
*ppp = polyp;
}

/*
 * small_polygon:
 * Break polygons into small polygons if reasonable
 */
POLYGON *
small_polygon(polygon, distance, edge)
POLYGON *polygon;
distance;
**edge;
{
    register PATH
    POLYGON
    PATH
    double
    double
    double
    double
    small = NULL;
    dsq_limit = distance * distance;
    while (polygon != NULL) {
        if (polygon->nside < 4) {
            polyp = polygon;
            polygon = polygon->next;
            polyp->next = small;
            small = polyp;
            continue;
        }
        minpp = NULL;
        mindeq = 0;
        for (pp = polygon->path; pp != NULL; pp = pp->next) {
            for (cp = pp->point->conn; cp != NULL; cp = cp->next)
                cp->point->mark = 1;
            for (tp = pp->next; tp != NULL; tp = tp->next) {
                if (tp->point->mark)
                    continue;
                dsq = distsq(pp->point, tp->point);
                if (dsq > dsq_limit)
                    continue;
                if (minpp == NULL || dsq < mindsq) {
                    minpp = tp;
                    mindsq = dsq;
                }
            }
            for (cp = pp->point->conn; cp != NULL; cp = cp->next)
                cp->point->mark = 0;
            if (minpp != NULL)
                break;
        }
        if (minpp == NULL) {
            /*
             * This polygon cannot be broken further

```

triangle.c

```

    /*
    polyp = polygon;
    polygon = polygon->next;
    polyp->next = small;
    small = polyp;
    continue;
}
break_polygon(&polygon, pp, minpp, edge);
}
return small;
}
/*
 * triangulate:
 * Break polygons into triangles
 */
POLYGON *
triangulate(polygon, live, dead, edge)
POLYGON *polygon;
POINT **live, **dead;
EDGE **edge;
{
    register POINT
    register POINT *prevpp, *saveprev;
    register PATH
    double
    double cx, cy, cz;
    double dx, dy, dz;
    double dsq, mindsq;
    POLYGON
    POLYGON *triangle, *polyp;
    *make_triangle();

    triangle = NULL;
    while (polygon != NULL) {
        /* Skip over triangles
        */
        if (polygon->inside < 4) {
            polyp = polygon;
            polygon = polyp->next;
            polyp->next = triangle;
            triangle = polyp;
            continue;
        }
        /* Now we need to break this polygon into as many
        * triangles as it has sides. The way we do this is
        * by computing the center of mass and finding the
        * closest dead point to it and promoting it to a
        * live point.
        */
        cx = cy = cz = 0;
        for (path = polygon->path; path != NULL; path = path->next) {
            cx += path->point->coord[0];
            cy += path->point->coord[1];
            cz += path->point->coord[2];
        }
        cx = cx / polygon->inside;
        cy = cy / polygon->inside;
        cz = cz / polygon->inside;
        savepp = NULL;
        mindsq = 0;
        prevpp = NULL;
        for (pp = *dead; pp != NULL; pp = pp->next) {
            dx = pp->coord[0] - cx;
            dy = pp->coord[1] - cy;
            dz = pp->coord[2] - cz;
            dsq = dx * dx + dy * dy + dz * dz;
            if (savepp == NULL || dsq < mindsq) {
                savepp = pp;
                saveprev = prevpp;
                mindsq = dsq;
            }
            prevpp = pp;
        }
        if (saveprev == NULL)
            *dead = savepp->next;
        else
            saveprev->next = savepp->next;
        savepp->next = *live;
        savepp->nconn = 0;
        *live = savepp;
    }
    /* Now we add the edges between the points on the
    * polygon and the new point
    */
    for (path = polygon->path; path != NULL; path = path->next)
        *edge = add_edge(path->point, savepp, *edge);
}
}

```

triangle.c

```

    * Now we create "inside" triangles from the new point
    * and the old polygon.
    */
    for (path = polygon->path; path != NULL; path = path->next)
        triangle = make_triangle(path->edge, savepp, triangle);

    /*
    * Go on to the next polygon
    */
    polyp = polygon;
    polygon = polyp->next;
    free_path(polyp->path);
    free((char *) polyp);
    }
    return triangle;
}

/*
 * make_triangle:
 * Create a triangle from an edge and a point
 */
POLYGON *
make_triangle(edge, point, polygon)
EDGE *edge;
POINT *point;
POLYGON *polygon;
{
    register POLYGON *polyp;
    register PATH *p1, *p2, *p3;
    register EDGE *edge;

    /* Create the polygon
    */
    polyp = (POLYGON *) malloc(sizeof (POLYGON));
    polyp->next = polygon;
    polyp->inside = 3;

    /* Create the path data structure
    */
    p1 = (PATH *) malloc(sizeof (PATH));
    p2 = (PATH *) malloc(sizeof (PATH));
    p3 = (PATH *) malloc(sizeof (PATH));
    p1->next = p2;
    p2->next = p3;
    p3->next = NULL;

    polyp->path = p1;

    /* Now fill in each one
    */
    p1->point = point;
    p2->point = edge->p1;
    p3->point = edge->p2;
    p1->edge = find_point_edge(point, edge->p2);
    p2->edge = find_point_edge(point, edge->p1);
    p3->edge = edge;

    return polyp;
}

/*
 * find_point_edge:
 * Find an edge between two points
 */
EDGE *
find_point_edge(from, to)
POINT *from;
register POINT *to;
{
    register CONN *cp;

    for (cp = from->conn; cp != NULL; cp = cp->next)
        if (cp->point == to)
            return cp->edge;

    return NULL;
}

/*
 * merge_polygon:
 * Convert the polygons into triangles and
 * return a list of them
 */
TRIANGLE *
merge_polygon(polygon, triangle)
POLYGON *polygon;
TRIANGLE *triangle;
{
    register POLYGON *p;
    register TRIANGLE *tp;
    register PATH *pathp;
    register int i;
}

```

triangle.c

```

while (polygon != NULL) {
    /*
    * Copy this polygon as a triangle
    */
    tp = (TRIANGLE *) malloc(sizeof (TRIANGLE));
    pathp = polygon->path;
    for (i = 0; i < 3; i++) {
        tp->vertex[i] = pathp->point;
        pathp = pathp->next;
    }
    tp->next = triangle;
    triangle = tp;
}

/*
 * Release this polygon and go to the next one
 */
polyp = polygon->next;
free_path(polygon->path);
free((char *) polygon);
polygon = polyp;
}

return triangle;
}

/*
 * print_triangle:
 * Print the triangles and points in some format
 */
FILE
TRIANGLE
POINT
{
    register int
    register POINT
    register TRIANGLE
    register CONN
    n;
    *pp;
    *tp;
    *cp;

    n = 0;
    for (pp = point; pp != NULL; pp = pp->next) {
        pp->clabel = n++;
        sort_neighbor(pp);
    }

#ifdef UNIFORM_NORMALS
    reorder_normals(point);
#endif
    printf(tp, "%d vertices\n", n);
}

while (polygon != NULL; pp = pp->next) {
    printf(tp, "%d\n(%f, %f), (%f, %f)\n", pp->clabel,
        pp->coord[0], pp->coord[1], pp->coord[2], pp->nconn);
    for (cp = pp->conn; cp != NULL; cp = cp->next)
        printf(tp, "%5d", cp->point->clabel);
    (void) puts("\n", fp);
}

n = 0;
for (tp = triangle; tp != NULL; tp = tp->next)
    n++;
printf(tp, "%d triangles\n", n);
for (tp = triangle; tp != NULL; tp = tp->next)
    order_triangle(tp, tp);
}

/*
 * sort_neighbor:
 * Sort the neighbors of the given point so that
 * they are in either clockwise or counterclockwise order
 */
sort_neighbor(point)
POINT *point;
{
    register CONN **cp;
    register CONN **bp;
    CONN *conn;
    CONN *neighbor[2];

    /*
    * We start by finding the neighbors to the initial connection
    */
    conn = point->conn;
    find_neighbor(conn, conn->point, neighbor);

    /*
    * Now we move one neighbor from the old list to the new list
    */
    for (bp = &conn; *bp != neighbor[0]; bp = &(*bp)->next)
        continue;
    *bp = (*bp)->next;
    neighbor[0]->next = NULL;
    point->conn = neighbor[0];
}

/*
 * Now we move the initial connection onto the new list
 */
}

```

triangle.c

```

cp = conn;
conn = conn->next;
cp->next = point->conn;
point->conn = cp;
}
/*
 * Now we move the other neighbor onto the new list
 */
for (bp = &conn; *bp != neighbor[1]; bp = &(*bp)->next)
    continue;
*bp = (*bp)->next;
neighbor[1]->next = point->conn;
point->conn = neighbor[1];
}
/*
 * Now we can proceed to find the succeeding neighbors
 * and keep at it until the old list is empty. In each
 * iteration, we should find exactly one neighbor
 */
while (conn != NULL) {
    find_neighbor(conn, point->conn->point, neighbor);
    for (bp = &conn; *bp != neighbor[0]; bp = &(*bp)->next)
        continue;
    *bp = (*bp)->next;
    neighbor[0]->next = point->conn;
    point->conn = neighbor[0];
}
}

/*
 * find_neighbor:
 * When given a list of connections, find the two possible
 * connections which are also connected to a given point
 */
find_neighbor(conn, point, neighbor)
CONN *conn;
POINT *point;
CONN *neighbor[2];
{
    register CONN *cp;
    register int i;

    i = 0;
    while (conn != NULL) {
        for (cp = conn->point->conn; cp != NULL; cp = cp->next)
            if (cp->point == point)
                break;

```

```

        if (cp != NULL)
            neighbor[i++] = conn;
        conn = conn->next;
    }
    while (i < 2)
        neighbor[i++] = NULL;
}

```

```

#endif
UNIFORM_NORMALS

```

```

/*
 * reorder_normals:
 * Reorder the normals such that they all either point
 * outwards or inwards
 */
reorder_normals(point)
POINT *point;
{
    register CONN *cp;
    register POINT *pp, *savepp;
    register int i;
    double normal[3], a[3], b[3];
}
/*
 * Find the point with the greatest z coordinate and
 * compute a normal for one of its triangles. If the
 * normal points in the positive z direction, then it
 * is pointing outwards, otherwise it is pointing inwards
 */
savepp = point;
for (pp = point->next; pp != NULL; pp = pp->next)
    if (pp->coord[2] > savepp->coord[2])
        savepp = pp;
cp = savepp->conn;
for (i = 0; i < 3; i++) {
    a[i] = cp->point->coord[i] - savepp->coord[i];
    b[i] = cp->next->point->coord[i] - savepp->coord[i];
}
vxx3(normal, a, b);
}
/*
 * We want all our normals to point outwards, so we reverse
 * the arguments if the normal is negative
 */
if (normal[2] > 0)
    reorder(savepp, cp->point, cp->next->point);
else
    reorder(savepp, cp->next->point, cp->point);
}

```

triangle.c

```

/*
 * Now we clean up after ourselves
 */
for (pp = point; pp != NULL; pp = pp->next)
    pp->mark = 0;
}

/* reorder:
 * Reverse the connectivity if "to" points to "from" in
 * center's connectivity list
 */
static
reorder(center, from, to)
POINT *center, *from, *to;
{
    register CONN *cp, *ocp;

/*
 * If this point is already fixed, then ignore it.
 * Otherwise, mark it and do the work
 */
if (center->mark)
    return;
center->mark = 1;

/* Make sure the connectivity is going the right direction
 */
for (cp = center->conn; cp != NULL; cp = cp->next)
    if (cp->point == from)
        break;
ocp = (cp->next == NULL) ? center->conn : cp->next;
if (ocp->point != to) {
/*
 * The point following the "from" is not "to" so the
 * direction is reversed. Just reverse the list.
 */
    ocp = center->conn;
    center->conn = NULL;
    while (ocp != NULL) {
        cp = ocp->next;
        ocp->next = center->conn;
        center->conn = ocp;
        ocp = cp;
    }
}

}

/* Now we process all children connectivity
 */
for (cp = center->conn; cp != NULL; cp = cp->next) {
    ocp = (cp->next == NULL) ? center->conn : cp->next;
    reorder(cp->point, ocp->point, center);
}

}

/* order_triangle:
 * Order the triangle vertices and output it to the file
 */
order_triangle(tp, to)
FILE *fp;
TRIANGLE *tp;
{
    register CONN *cp, *ncp;

for (cp = tp->vertex[0]->conn; cp != NULL; cp = cp->next)
    if (cp->point == tp->vertex[1])
        break;
ncp = (cp->next == NULL) ? tp->vertex[0]->conn : cp->next;
if (ncp->point == tp->vertex[2])
    fprintf(fp, "%5d %5d %5d\n", tp->vertex[0]->dlabel,
        tp->vertex[1]->dlabel, tp->vertex[2]->dlabel);
else
    fprintf(fp, "%5d %5d %5d\n", tp->vertex[2]->dlabel,
        tp->vertex[1]->dlabel, tp->vertex[0]->dlabel);
}

#define m(pp)    fprintf(fp, "m %.3f %.3f %.3f\n", (pp)->coord[0],
                (pp)->coord[1], (pp)->coord[2])
#define d(pp)    fprintf(fp, "d %.3f %.3f %.3f\n", (pp)->coord[0],
                (pp)->coord[1], (pp)->coord[2])
#define dot(pp)  fprintf(fp, ".dot %.3f %.3f %.3f\n", (pp)->coord[0],
                (pp)->coord[1], (pp)->coord[2])
#define polord(pp)
                fprintf(fp, ".polord %.3f %.3f %.3f\n", (pp)->coord[0],
                (pp)->coord[1], (pp)->coord[2])
#define color(c)
                fprintf(fp, ".color %d\n", c)
#define trans(x, y, z)
                fprintf(fp, ".tran %.3f %.3f %.3f\n", x, y, z)
#define scale(s)
                fprintf(fp, ".scale %.3f %.3f %.3f\n", s, s, s)
#define cmov(pp)
                fprintf(fp, ".cmov %.3f %.3f %.3f\n", (pp)->coord[0],
                (pp)->coord[1], (pp)->coord[2])

```


triangle.c

```

#define charstr(str)    fprintf(fp, "%s\n", str)
#define SCALEPROD     140.0

/*
 * print_bld:
 * Print the triangles in some format or other
 */
print_bld(fp, points, edge, dead)
FILE *fp;
POINT *points, *dead;
EDGE *edge;
{
    register POINT *pp;
    register EDGE *ep;
    register int i;
    double min[3], max[3], coord[3];
    double sc;

    for (i = 0; i < 3; i++) {
        min[i] = points->coord[i];
        max[i] = points->coord[i];
    }
    for (pp = points; pp != NULL; pp = pp->next) {
        for (i = 0; i < 3; i++) {
            if (pp->coord[i] < min[i])
                min[i] = pp->coord[i];
            else if (pp->coord[i] > max[i])
                max[i] = pp->coord[i];
        }
        sc = max[0] - min[0];
        if (max[1] - min[1] > sc)
            sc = max[1] - min[1];
        scale(sc);
        for (i = 0; i < 3; i++)
            coord[i] = (min[i] + max[i]) / 2;
        trans(-coord[0], -coord[1], -coord[2]);
        color(1);
        for (ep = edge; ep != NULL; ep = ep->next) {
            m(ep->p1);
            d(ep->p2);
        }
        for (pp = points; pp != NULL; pp = pp->next) {
            cmov(pp);
        }
    }
}

charstr("o");
}
for (pp = dead; pp != NULL; pp = pp->next)
    dot(pp);

/*
 * print_edges:
 * Print the triangles and polygons in some format or other
 */
print_edges(points, triangles, polygons)
POINT *points;
TRIANGLE *triangles;
POLYGON *polygons;
{
    register FILE *fp;
    register POINT *pp;
    register TRIANGLE *tp;
    register POLYGON *polyp;
    register PATH *pathp, *nextp;
    register int i;
    double min[3], max[3], coord[3];
    double sc;

    fp = fopen("debug", "w");
    for (i = 0; i < 3; i++) {
        min[i] = points->coord[i];
        max[i] = points->coord[i];
    }
    for (pp = points; pp != NULL; pp = pp->next) {
        for (i = 0; i < 3; i++) {
            if (pp->coord[i] < min[i])
                min[i] = pp->coord[i];
            else if (pp->coord[i] > max[i])
                max[i] = pp->coord[i];
        }
        sc = max[0] - min[0];
        if (max[1] - min[1] > sc)
            sc = max[1] - min[1];
        scale(sc);
        for (i = 0; i < 3; i++)
            coord[i] = (min[i] + max[i]) / 2;
        trans(-coord[0], -coord[1], -coord[2]);
        color(1);
        for (tp = triangles; tp != NULL; tp = tp->next) {
            cmov(tp);
        }
    }
}
}

```

triangle.c

```
m(tp->vertex(0));
d(tp->vertex(1));
d(tp->vertex(2));
d(tp->vertex(0));
}
color(2);
for (polyp = polygons; polyp != NULL; polyp = polyp->next) {
    for (pathp = polyp->path; pathp != NULL; pathp = pathp->next) {
        nextp = (pathp->next == NULL) ? polyp->path
        : pathp->next;
        m(pathp->point);
        d(nextp->point);
    }
}
(void) fclose(fp);
}
```

triangle.h

```
/*  
 * Copyright (c) 1987 by the Regents of the University of California  
 * All rights reserved.  
 */  
typedef struct triangle_def  
struct triangle_def  
POINT  
TRIANGLE;  
}  
extern TRIANGLE *form_triangle();
```

uniform.c

```

/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
#include <stdio.h>
#include <math.h>
#include "center.h"
#include "point.h"
#include "surface.h"
#include "triangle.h"

#define TRUE
#define TRUE 1
#define FALSE 0
#define FALSE 0

#define RATIO 10.0
/* #define MAXRATIO sqrt(2.0) */
/* #define MAXRATIO 4.0 / 3.0 */
/* Ratio of delta to distance */
/* Ratio of max edge to distance */

usage(progname)
char *progname;
{
    fprintf(stderr, "Usage: %s [-i pdb_file] [-o output] [-b bild_file]\n",
        progname);
    fprintf(stderr, "\t[-d distance] [-r radius] [-m max_ratio] [-v]\n\n");
    fprintf(stderr, "\t-i pdb_file\tInput file in PDB format\n");
    fprintf(stderr, "\t-i-o output\tOutput file of points and triangles\n");
    fprintf(stderr, "\t-i-b bild_file\tOutput file of triangles in bild format\n");
    fprintf(stderr, "\t-i-d distance\tDesired distance between points\n");
    fprintf(stderr, "\t-r radius\tRadius of atoms\n");
    fprintf(stderr, "\t-i-m max_ratio\tMaximum ratio of longest edge to
        desired distance\n");
    fprintf(stderr, "\t-v\tVerbose mode\n");
}

main(ac, av)
int ac;
char **av;
{
    register CENTER *atom;
    register POINT *proto;
    POINT *point, *chosen;
    EDGE *edge;
    TRIANGLE *triangle;
    INTER **inter;
    verbose;
    c;
    "infile, *outfile, *bildfile;
    *infd, *outfd, *bildfd;
    distance, radius;
    min_dist;
    max_ratio;
    optind;
    *optarg;

    /*
     * Initialize default values
     */
    radius = 1.7;
    distance = 1.0;
    infile = outfile = bildfile = NULL;
    max_ratio = MAXRATIO;
    verbose = 0;

    /*
     * Process command line arguments
     */
    while ((c = getopt(ac, av, "i:o:d:r:m:b:vh")) != EOF)
        switch (c) {
            case 'i':
                infile = optarg;
                break;
            case 'o':
                outfile = optarg;
                break;
            case 'd':
                distance = atof(optarg);
                break;
            case 'r':
                radius = atof(optarg);
                break;
            case 'm':
                max_ratio = atof(optarg);
                break;
            case 'b':
                bildfile = optarg;
                break;
            case 'v':
                verbose = TRUE;
                break;
            case 'h':
                usage(av(0));
        }
}

```

uniform.c

```

        }
        exit(0);
    }

    /* Make sure input and output files are accessible
    */
    if (optind < ac) {
        usage(av[0]);
        exit(1);
    }
    if (infile == NULL) {
        infd = stdin;
        infile = "standard input";
    }
    else if ((infd = fopen(infile, "r")) == NULL) {
        perror(infile);
        exit(1);
    }
    if (outfile == NULL) {
        outfd = stdout;
        outfile = "standard output";
    }
    else if ((outfd = fopen(outfile, "w")) == NULL) {
        perror(outfile);
        exit(1);
    }
    if (bildfile != NULL) {
        if ((bildfd = fopen(bildfile, "w")) == NULL) {
            perror(bildfile);
            exit(1);
        }
    }
    else
        bildfd = NULL;
    min_dist = distance / RATIO;

    /* Get list of atoms
    * and inter-atom properties
    */
    if (verbose || (infd == stdin && isatty(0)))
        printf(stderr, "Read pdb file \"%s\n\"", infile);
    atom = read_pdb(infd);
    inter = compute_inter(atom, radius, distance + min_dist);

    /* Get prototype sphere
    */
    }

    /* (verbose)
    printf(stderr, "Generate prototype sphere\n");
    proto = sphere(radius, min_dist);

    /* Generate the list of points on surface
    */
    if (verbose)
        printf(stderr, "Generate surface points\n");
    point = surface(radius, distance, atom, proto, inter);

    /* Choose the list of points we want to retain
    */
    if (verbose)
        printf(stderr, "Choosing surface points\n");
    chosen = choose(&point, distance, min_dist);

    /* Make up triangles from the chosen points
    * (and maybe the eliminated points as well)
    */
    if (verbose)
        printf(stderr, "Making triangles\n");
    triangle = form_triangle(&chosen, &point, distance, min_dist,
        max_ratio, inter, &edge);

    /* Output triangles
    */
    if (verbose)
        printf(stderr, "Printing surface file \"%s\n\"", outfile);
    print_triangle(outfd, triangle, chosen);

    /* Output in bild format if desired
    */
    if (bildfd != NULL) {
        if (verbose)
            printf(stderr, "Printing bild file \"%s\n\"", bildfile);
        print_bild(bildfd, chosen, edge, point);
    }
    exit(0);
}

```

Appendix Two: Program Listings for Patches Display

colors.c

```

/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
#include <stdio.h>
#include <math.h>
#include <gl.h>
#include <device.h>
#include "colors.h"
#define BLUEFUDGE 1.38 /* factor to increase blue to increase
 * separation of green and red */

#define max(a,b) ((a) > (b) ? (a) : (b))

double color_pow = 1.0;
typedef struct {
    short r, g, b;
} RGB;

RGB *Orig_map;

/* Map the color range as:
 * first 1/2 of wheel is red -> green
 * next 1/4 of wheel is green -> blue
 * last 1/4 of wheel is blue -> red */
#define WHEELSTART 8
#define WHEELEND ((WHEELSTART - WHEELSTART) / 16)
#define WHEELSLICE WHEELSTART
#define REDSTART (REDSTART + 5 * WHEELSLICE)
#define GREENSTART (GREENSTART + 4 * WHEELSLICE)
#define BLUESTART 255
#define MAXINTENSITY

/* Convert r,g,b value so largest value is MAXINTENSITY and others
 * are scaled accordingly.
 */
static void
rgb_normalize(r, g, b)
double *r, *g, *b;
{
    double mx;

    mx = max(*r, 0.0);
    mx = max(*g, mx);
    mx = max(*b, mx);
    if (mx == 0.0 || mx <= MAXINTENSITY)
        return;
    *r = (*r * MAXINTENSITY) / mx;
    *g = (*g * MAXINTENSITY) / mx;
    *b = (*b * MAXINTENSITY) / mx;
}

#define FROMPOW(c, from, to) \
    MAXINTENSITY * pow((double) (c - from) / (to - from), 1.0 / color_pow)
#define TOPOW(c, from, to) \
    MAXINTENSITY * pow((double) (to - c) / (to - from), 1.0 / color_pow)

/* Calculate the rgb value according to the input index. Default
 * value is white. Map colors 0 - 7 like the IRIS default.
 */
static void
rgb_getcolor(c, r, g, b)
int c;
double *r, *g, *b;
{
    if (c >= 0 && c < WHEELSTART) {
        *r = c & 0x01 ? MAXINTENSITY : 0;
        *g = c & 0x02 ? MAXINTENSITY : 0;
        *b = c & 0x04 ? MAXINTENSITY : 0;
    } else if (c >= REDSTART && c < GREENSTART) {
        *r = TOPOW(c, REDSTART, GREENSTART);
        *g = FROMPOW(c, REDSTART, GREENSTART);
        *b = 0;
    } else if (c >= GREENSTART && c < BLUESTART) {
        *r = 0;
        *g = TOPOW(c, GREENSTART, BLUESTART);
        *b = FROMPOW(c, GREENSTART, BLUESTART);
    } else if (c >= BLUESTART && c < NUM_COLORS) {
        *r = FROMPOW(c, BLUESTART, NUM_COLORS);
        *g = 0;
        *b = BLUEFUDGE * TOPOW(c, BLUESTART, NUM_COLORS);
    } else
        *r = *g = *b = MAXINTENSITY;
    rgb_normalize(r, g, b);
}

#define RND(x) ((int) (x + 0.5))
#define colormap()
register ColorIndex c;
auto double r, g, b;

```

colors.c

```

for (c = 0; c < NUM_COLORS; c++) {
    rgb_getcolor(c, &r, &g, &b);
    map_range(c, RND(r), RND(g), RND(b));
}

#define M_RGB 0
#define M_SINGLE 1
#define M_DOUBLE 2

int num_shades = 1;

map_colors(
{
    register Colorindex c;
    register RGB *rgb;
    auto short or, og, ob;
    int cur_mode;

    cur_mode = getisplaymode();
    if (cur_mode != M_DOUBLE) {
        doublebuffer();
        gconfig();
    }
    num_shades = 1 << (getplanes() - LOGN_COLORS);
    Orig_map = (RGB *) emalloc(sizeof (RGB) * NUM_COLORS * num_shades);
    for (rgb = Orig_map, c = 0; c < NUM_COLORS * num_shades; c++, rgb++)
        getimcolor(c, &or, &og, &ob);
    rgb->r = or;
    rgb->g = og;
    rgb->b = ob;
}

# ifdef DISKCOLORMAP
readcolormap();
setcolormap();

switch (cur_mode) {
case M_RGB:
    RGBmode();
    gconfig();
    break;
case M_SINGLE:
    singlebuffer();
    gconfig();
}

for (c = 0; c < NUM_COLORS; c++) {
    rgb_getcolor(c, &r, &g, &b);
    map_range(c, RND(r), RND(g), RND(b));
}

#define GAMMA 0.75

int vh = 255; /* intensity at hither plane (0-255) */
int vy = 128; /* intensity at yon plane (0-255) */

map_range(c, r, g, b)
Colorindex c;
short r, g, b;
{
    register double d;
    register double maxinten, mininten;
    register int i;

    maxinten = vh;
    mininten = vy;
    c *= num_shades;
    for (i = 0; i < num_shades; i++) {
        d = pow((maxinten - i * (maxinten - mininten) / num_shades) /
            maxinten, 1 / GAMMA);
        mapcolor(c + num_shades - 1 - i, (short) (d * r),
            (short) (d * g), (short) (d * b));
    }
}

restore_map(
{
    register Colorindex c;
    register RGB *rgb;

    rgb = Orig_map;
    for (rgb = Orig_map, c = 0; c < num_shades * NUM_COLORS; c++, rgb++)
        mapcolor(c, rgb->r, rgb->g, rgb->b);
    fflush();
}

# ifndef DISKCOLORMAP
readcolormap(
{
    FILE *fp, *fopen();
    char buf[BUFSIZ];
    int c, r, g, b;
}

```


colors.c

```

    if ((fp = fopen(DISKCOLORMAP, "r")) == NULL)
        return;
    while (fgets(buf, BUFSIZ, fp)) {
        if (sscanf(buf, "%d%d%d", &c, &r, &g, &b) != 4)
            warn("Bad scan from readcolormap\n");
        else
            map_range(c, r, g, b);
    }
    fclose(fp);
}
# endif

# define IHEIGHT 30
# define IWIDHT 120
# define IMHEIGHT (IHEIGHT * 3 / 2)
# define IMWIDHT (IWIDHT * 3 / 2)

# define IDECHITHER 0
# define IINCHITHER 1
# define IDECYON 2
# define IINCYON 3
/*
 * adjust_intensity:
 * / Adjust the min and max intensities and remap the color map
Object background;
{
    register int x, y;
    register int dx, dy;
    register int ox, oy;
    register int isup;
    register int which, rate;
    int ovh, ovy;
    int quit;
    short data;

    ox = getvaluator(MOUSEX);
    oy = getvaluator(MOUSEY);
    ovh = vh;
    ovy = vy;

    isup = TRUE;
    for (quit = FALSE; !quit;) {
        x = getvaluator(MOUSEX);
        y = getvaluator(MOUSEY);
        dx = x - ox;
        dy = y - oy;
        if (dx < -IMWIDHT || dx > IMWIDHT)
            break;
        if (dy < -IMHEIGHT || dy > IMHEIGHT)
            break;
        if (dx < 0)
            which = (dy < 0) ? IDECYON : IDECHITHER;
        else
            which = (dy < 0) ? IINCYON : IINCHITHER;
        rate = (dx / (double) IWIDHT) * 5;
        if (rate < 0)
            rate = -rate;
        rate++;
        draw_intensity(background, ox, oy, vh, vy, which, isup);
        if (qtest())
            switch (qread(&data)) {
                case LEFTMOUSE:
                case MIDDLEMOUSE:
                case RIGHTMOUSE:
                    isup = ! (Boolean) data;
                    break;
                default:
                    quit = TRUE;
                    break;
            }
        switch (which) {
            case IINCHITHER:
                vh += rate;
                if (vh > 255)
                    vh = 255;
                break;
            case IINCYON:
                vy += rate;
                if (vy > vh)
                    vy = vh;
                break;
            case IDECHITHER:
                vh -= rate;
                if (vh < vy)
                    vh = vy;
                break;
            case IDECYON:
                vy -= rate;
                if (vy < 0)
                    vy = 0;
                break;
        }
    }
}

```

colors.c

```

        vy = 0;
        break;
    }
}

/*
 * If we need to remap the color map, do it
 */
if (vh != ovh || vy != ovy) {
    callobj(background);
    swapbuffers();
    setcolormap();
}

/*
 * draw_intensity:
 * Draw the intensity control box over the given background
 */
static
draw_intensity(background, x, y, hither, yon, which, isup)
Object
background;
int
x, y;
int
hither, yon;
int
which, isup;
{
    char
    buff[20];
    Coord
    tx, ty;
    Colorindex
    other;

    other = isup ? YELLOW : GREEN;
    callobj(background);
    pushmatrix();
    translate((Coord) x, (Coord) y, (Coord) 0);
    color(COLOR(BLACK));
    rectf((Coord) -IWIDTH, (Coord) -IHEIGHT,
          (Coord) IWIDTH, (Coord) IHEIGHT);

    color(COLOR(MAGENTA));
    (void) sprintf(buff, "Max intensity: %3d", hither);
    ty = getheight(t);
    tx = stwidth(buff);
    cmov(-tx / 2, (Coord) ((IHEIGHT - ty) / 2), (Coord) 0);
    charstr(buff);
    (void) sprintf(buff, "Min intensity: %3d", yon);
    tx = stwidth(buff);
    cmov(-tx / 2, (Coord) (-IHEIGHT - ty) / 2 - ty, (Coord) 0);
    charstr(buff);

    cmov((Coord) -IWIDTH, (Coord) (-IHEIGHT - ty) / 2 - ty,
          (Coord) 0);
    color(which == IDECYON ? COLOR(other) : COLOR(RED));
    charstr("-");

    cmov((Coord) -IWIDTH, (Coord) ((IHEIGHT - ty) / 2),
          (Coord) 0);
    color(which == IDECHITHER ? COLOR(other) : COLOR(RED));
    charstr("-");

    tx = stwidth("+");
    cmov((Coord) IWIDTH - tx, (Coord) (-IHEIGHT - ty) / 2 - ty,
          (Coord) 0);
    color(which == IINCYNON ? COLOR(other) : COLOR(RED));
    charstr("+");

    cmov((Coord) IWIDTH - tx, (Coord) ((IHEIGHT - ty) / 2),
          (Coord) 0);
    color(which == IINCHITHER ? COLOR(other) : COLOR(RED));
    charstr("+");

    color(COLOR(BLUE));
    rectf((Coord) -IWIDTH, (Coord) -IHEIGHT,
          (Coord) IWIDTH, (Coord) IHEIGHT);

    popmatrix();
    swapbuffers();
}
}

```

colors.h

```
/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
/* $Header: /usr/kneller/src/bld/RCS/colors.h,v 1.1 86/10/27 15:38:42 kneller Exp $ */

/* The first 8 colors are reserved for the IRIS definitions and for
 * compatibility with PS2 bitd. The next NUM_COLORS - 8 will be mapped
 * to a color wheel.
 */
#define LOGNCOLORS 3
#define NUM_COLORS (1 << LOGNCOLORS)
#define ZNEAR 0xC000
#define ZFAR 0x3FFF
/* Define this if you don't like the default color map.
 * The colors will be read out of the file.
 */
#define DISKCOLORMAP "colormap"

#define COLOR(c) ((c) * num_shades + num_shades - 1)
#define RANGE(c) (c) * num_shades, COLOR(c), ZNEAR, ZFAR

extern int vh, vy;
extern int num_shades;
```

depict.c

```
/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
#include <stdio.h>
#include <gl.h>
#include <device.h>
#include "colors.h"
#include "menu.h"

#define HALFLEN 100.
#define PER_FACTOR 3
#define EYE (PER_FACTOR * HALFLEN)

#define MOD_THRESHOLD200

#define ORIGIN 0.
#define JOYSTICK_WIDTH 200.
#define JOYSTICK_HEIGHT 130.
#define POTENTIOMETER_WIDTH 200.
#define POTENTIOMETER_HEIGHT 10.
#define POT_HEIGHT (POTENTIOMETER_HEIGHT + 24.)
#define JOY_HEIGHT (JOYSTICK_HEIGHT + 25.)
#define VBARSTART 740. /* Top position onscreen */

#define MIN_X 795.
#define MAX_X 995.

#define NONE 0
#define JOYSTICK 0x1
#define POTENTIOMETER 0x2

#define SCALE 1
#define SCALE_X MIN_X
#define SCALE_Y VBARSTART
#define SCALE_LABEL "scale"

#define YON 2
#define YON_X MIN_X
#define YON_Y (SCALE_Y - POT_HEIGHT)
#define YON_LABEL "yon"

#define HITHER 3
#define HITHER_X MIN_X
#define HITHER_Y (YON_Y - POT_HEIGHT)
#define HITHER_LABEL "hither"

#define SECTION 4
#define SECTION_X MIN_X
#define SECTION_Y (HITHER_Y - POT_HEIGHT)
#define SECTION_LABEL "section"

#define ZTRAN 5
#define ZTRAN_X MIN_X
#define ZTRAN_Y (SECTION_Y - POT_HEIGHT)
#define ZTRAN_LABEL "z translation"

#define ZROT 6
#define ZROT_X MIN_X
#define ZROT_Y (ZTRAN_Y - POT_HEIGHT)
#define ZROT_LABEL "z rotation"

#define XYROT 7
#define XYROT_X MIN_X
#define XYROT_Y (ZROT_Y - JOY_HEIGHT)
#define XYROT_LABEL "x-y rotation"

#define XYTRAN 8
#define XYTRAN_X MIN_X
#define XYTRAN_Y (XYROT_Y - JOY_HEIGHT)
#define XYTRAN_LABEL "x-y translation"

#define NORMAL 9
#define NORMAL_X MIN_X
#define NORMAL_Y (XYTRAN_Y - 2 * POT_HEIGHT)
#define NORMAL_LABEL "normal translation"

#define TANGENT 10
#define TANGENT_X MIN_X
#define TANGENT_Y (NORMAL_Y - JOY_HEIGHT)
#define TANGENT_LABEL "tangent translation"

#define DEFAULTSCALE 1.0

typedef struct view_def View;
struct view_def {
    View *v_next;
    float v_scale;
    Matrix v_matrix;
    Boolean v_ortho;
    Coord v_hither, v_yon;
};
```

depict.c

extern Object

Object

everything;
Object view_obj, transformation_obj, scale_obj;
Tag obj_tag;
Matrix ident;

bid_obj, point_obj, move_obj;

static Object frame = -1;
static Object joystick, potentiometer;
static Tag xytran_tag, xyrot_tag, scale_tag, zrot_tag, ztran_tag;
static Tag normal_tag, tangent_tag;
static Tag hither_tag, yon_tag, section_tag;
static Tag window_tag, end_window_tag, size_tag;
static Tag transform_tag, end_transform_tag;
static Tag scale_label_tag;
static Menu *main_menu;
static Menu *move_menu;
static Menu *render_menu;
static Menu *color_menu;
static View *view_stack = NULL;
static int nmods = 0;
static Boolean ortho_view;

static Cursor hour_glass = {
0x1FF0, 0x1FF0, 0x0820, 0x0820,
0x0820, 0x0C60, 0x06C0, 0x0100,
0x0100, 0x06C0, 0x0C60, 0x0820,
0x0820, 0x0820, 0x1FF0, 0x1FF0
};

char view_mode[] = "perspective0"; /* Long enough for "orthographic" */
char which_eye[] = "left eye0"; /* Long enough for "right eye" */

MENU_ITEM main_menu_item[] = {
"roll", M_ROLL,
"rock", M_ROCK,
"reset", M_RESET,
"pop", M_POP,
"push", M_PUSH,
view_mode, M_OP,
which_eye, M_OTHEREYE,
"intensity", M_INTENSITY,
"stop", M_STOP
};

MENU_ITEM move_menu_item[] = {
"move point", M_PICK,
};

"display marker", M_ADD_MARKER,
"remove marker", M_REMOVE_MARKER,
"update", M_REDRAW,
"abort move", M_ABORT_MOVE,
"accept move", M_ACCEPT_MOVE,
"set color", M_SET_COLOR,
"render", M_RENDER,
"position light", M_LIGHT,
"stop", M_STOP,

};
MENU_ITEM color_menu_item[] = {
"red", C_RED,
"green", C_GREEN,
"blue", C_BLUE,
"yellow", C_YELLOW,
"cyan", C_CYAN,
"magenta", C_MAGENTA,
"white", C_WHITE,
};

MENU_ITEM render_menu_item[] = {
"none", T_NONE,
"hatched", T_THATCHED,
"halftone", T_HALFTONE,
"crosshatch", T_CROSSHATCH,
};

define remake_main_menu() free_menu(main_menu),\nmain_menu = make_menu(main_menu_item),\nsizeof main_menu_item);

/* *init_depict - creates all objects called by depict() */
init_depict()

{ /* * initialize object/tags that are used later */
everything = genobj();
view_obj = genobj();
transformation_obj = genobj();
scale_obj = genobj();
xytran_tag = genobj();
xyrot_tag = genobj();
scale_tag = genobj();
};

depict.c

```

scale_label_tag = gentag();
zrot_tag = gentag();
ztran_tag = gentag();
normal_tag = gentag();
tangent_tag = gentag();
hither_tag = gentag();
yon_tag = gentag();
section_tag = gentag();
end_window_tag = gentag();
size_tag = gentag();
transform_tag = gentag();
end_transform_tag = gentag();
obj_tag = gentag();
font(0);

/*
 * The frame around the screen
 */
frame = genobj();
makeobj(frame);
    move2(0, 0);
    draw2(0, YMAXSCREEN);
    draw2(XMAXSCREEN, YMAXSCREEN);
    draw2(XMAXSCREEN, 1);
    draw2(0, 0);
    move2(YMAXSCREEN, 0);
    draw2(YMAXSCREEN, YMAXSCREEN);
closeobj();

/*
 * The horizontal potentiometer
 */
potentiometer = genobj();
makeobj(potentiometer);
    move2(ORIGIN, ORIGIN);
    draw2(POTENTIOMETER_WIDTH, ORIGIN);
    move2(POTENTIOMETER_WIDTH, POTENTIOMETER_HEIGHT);
    move2(ORIGIN, POTENTIOMETER_HEIGHT/2);
    draw2(POTENTIOMETER_WIDTH, POTENTIOMETER_HEIGHT/
    move2(POTENTIOMETER_WIDTH/2, ORIGIN);
    draw2(POTENTIOMETER_WIDTH/2, POTENTIOMETER_HEIGHT);
closeobj();

/*
 * The square joystick
 */
joystick = genobj();
makeobj(joystick);
    move2(ORIGIN, ORIGIN);
    draw2(JOYSTICK_WIDTH, JOYSTICK_HEIGHT);
    move2(ORIGIN, JOYSTICK_HEIGHT/2);
    draw2(JOYSTICK_WIDTH, JOYSTICK_HEIGHT/2);
    move2(ORIGIN, JOYSTICK_HEIGHT);
    move2(JOYSTICK_WIDTH/2, JOYSTICK_HEIGHT);
    draw2(JOYSTICK_WIDTH/2, ORIGIN);
closeobj();

/*
 * define the "wait-a-second" cursor
 */
defcursor(1, hour_glass);

/*
 * create the menus
 */
main_menu = make_menu(main_menu_item, sizeof(main_menu_item));
move_menu = make_menu(move_menu_item, sizeof(move_menu_item));
render_menu = make_menu(render_menu_item, sizeof(render_menu_item));
color_menu = make_menu(color_menu_item, sizeof(color_menu_item));
identity(ident);

}

/*
 * depict - draw up the user-defined object and manipulate it
 */
depict()
{
    register Boolean
    Boolean
    View
    Coord
    Matrix
    float
    short
    char
    lint
    Boolean
    quit, pen_down, changed;
    rocking, rolling;
    *v;
    hither, yon;
    m;
    sc;
    data;
    buff[BUFSZ];
    char_height, str_length, showing_right, mv;
    *malloc();
    check_devices(), rock(), roll();
}

```

depict.c

```
* make sure objects have all been created
*/
if (frame == -1)
    init_depict();

/*
 * double buffering always looks better
 */
doublebuffer();
gconfig();
char_height = getheight();

/*
 * initialize the variables used in the display loop
 */
ortho = FALSE;
ortho_view = TRUE;
showing_right = TRUE;
rocking = rolling = FALSE;
pen_down = getbutton(LEFTMOUSE);
sc = DEFAULTSCALE;
hither = -HALFLEN;
yon = HALFLEN;
identity(m);
setcursor(1, COLOR(RED), (1 <= getplanes() - 1); /* wait cursor */
attachcursor(MOUSEX, MOUSEY);
cursor();

/*
 * construct the object we'll be displaying
 */
makeobj(view_obj);
    maketag(window_tag);
    ortho(-HALFLEN, HALFLEN, -HALFLEN, HALFLEN, hither, yon);
    maketag(end_window_tag);
closeobj();
makeobj(transformation_obj);
    maketag(transform_tag);
    multmatrix(m);
    maketag(end_transform_tag);
closeobj();
makeobj(scale_obj);
    maketag(size_tag);
    scale(sc, sc, sc);
closeobj();
makeobj(everything);
*/
```

```
 * blue background
*/
color(COLOR(BLACK));
clear();
color(COLOR(BLUE));
callobj(frame);

/*
 * display the user-defined object
 */
pushattribs();
pushviewport();
pushmatrix();
    depthcue(TRUE);
    loadmatrix(ident);
    viewport(0, YMAXSCREEN, 0, YMAXSCREEN);
    callobj(view_obj);
    callobj(transformation_obj);
    callobj(scale_obj);
    maketag(obj_tag);
    callobj(bld_obj);
    callobj(point_obj);
    callobj(move_obj);
    depthcue(FALSE);
popmatrix();
popviewport();
popattribs();

/*
 * draw xy-translation joystick
 */
pushmatrix();
translate(XYTRAN_X, XYTRAN_Y, 0.);
maketag(xytran_tag);
color(COLOR(RED));
callobj(joystick);
popmatrix();
str_length = strwidth(XYTRAN_LABEL);
cmov(XYTRAN_X + (JOYSTICK_WIDTH - str_length) / 2,
     XYTRAN_Y - char_height - 4, 0.);
charstr(XYTRAN_LABEL);

/*
 * draw xy-rotation joystick
 */
pushmatrix();
translate(XYROT_X, XYROT_Y, 0.);
```

deplot.c

```

maketag(xyrot_tag);
color(COLOR(RED));
callobj(joystick);
popmatrix();
str_length = stwidth(XYROT_LABEL);
cmov(XYROT_X + (JOYSTICK_WIDTH - str_length) / 2,
      XYROT_Y - char_height - 4, 0);
charstr(XYROT_LABEL);

/* * draw tangent joystick */
pushmatrix();
translate(TANGENT_X, TANGENT_Y, 0);
maketag(tangent_tag);
color(COLOR(RED));
callobj(joystick);
popmatrix();
str_length = stwidth(TANGENT_LABEL);
cmov(TANGENT_X + (JOYSTICK_WIDTH - str_length) / 2,
      TANGENT_Y - char_height - 4, 0);
charstr(TANGENT_LABEL);

/* * draw z-rotation potentiometer */
pushmatrix();
translate(ZROT_X, ZROT_Y, 0);
maketag(zrot_tag);
color(COLOR(RED));
callobj(potentiometer);
popmatrix();
str_length = stwidth(ZROT_LABEL);
cmov(ZROT_X + (POTENTIOMETER_WIDTH - str_length) / 2,
      ZROT_Y - char_height - 4, 0);
charstr(ZROT_LABEL);

/* * draw hither potentiometer */
pushmatrix();
translate(HITHER_X, HITHER_Y, 0);
maketag(hither_tag);
color(COLOR(RED));
callobj(potentiometer);
popmatrix();
str_length = stwidth(HITHER_LABEL);

cmov(HITHER_X + (POTENTIOMETER_WIDTH - str_length) / 2,
      HITHER_Y - char_height - 4, 0);
charstr(HITHER_LABEL);

/* * draw yon potentiometer */
pushmatrix();
translate(YON_X, YON_Y, 0);
maketag(yon_tag);
color(COLOR(RED));
callobj(potentiometer);
popmatrix();
str_length = stwidth(YON_LABEL);
cmov(YON_X + (POTENTIOMETER_WIDTH - str_length) / 2,
      YON_Y - char_height - 4, 0);
charstr(YON_LABEL);

/* * draw section potentiometer */
pushmatrix();
translate(SECTION_X, SECTION_Y, 0);
maketag(section_tag);
color(COLOR(RED));
callobj(potentiometer);
popmatrix();
str_length = stwidth(SECTION_LABEL);
cmov(SECTION_X + (POTENTIOMETER_WIDTH - str_length) / 2,
      SECTION_Y - char_height - 4, 0);
charstr(SECTION_LABEL);

/* * draw scale potentiometer */
pushmatrix();
translate(SCALE_X, SCALE_Y, 0);
maketag(scale_tag);
color(COLOR(RED));
callobj(potentiometer);
popmatrix();
(void) sprintf(buf, "%s: %5.2f", SCALE_LABEL, sc);
str_length = stwidth(buf);
cmov(SCALE_X + (POTENTIOMETER_WIDTH - str_length) / 2,
      SCALE_Y - char_height - 4, 0);
charstr(SCALE_LABEL);
charstr(": ");

```


depict.c

```

makeitag(scale_label_tag);
(void) sprintf(buf, "%5.2f", sc);
charstr(buf);

/*
 * draw normal potentiometer
 */
pushmatrix();
translate(NORMAL_X, NORMAL_Y, 0.);
makeitag(normal_tag);
color(COLOR(RED));
callobj(potentiometer);
popmatrix();
str_length = strwidth(NORMAL_LABEL);
cmov(NORMAL_X + (POTENTIOMETER_WIDTH - str_length) / 2,
      NORMAL_Y - char_height - 4, 0.);
charstr(NORMAL_LABEL);

/*
 * draw z-translation potentiometer
 */
pushmatrix();
translate(ZTRAN_X, ZTRAN_Y, 0.);
makeitag(ztran_tag);
color(COLOR(RED));
callobj(potentiometer);
popmatrix();
str_length = strwidth(ZTRAN_LABEL);
cmov(ZTRAN_X + (POTENTIOMETER_WIDTH - str_length) / 2,
      ZTRAN_Y - char_height - 4, 0.);
charstr(ZTRAN_LABEL);

closeobj(/* everything */);

/*
 * initialize the cursor position and set up the event queue
 */
qreset();
qdevice(RIGHTMOUSE);
qdevice(MIDDLEMOUSE);
qdevice	LEFTMOUSE);

/*
 * get a nice simple viewport/window set up for drawing the
 * frame and tracking the cursor
 */
viewport(0, XMAXSCREEN, 0, YMAXSCREEN);
setdepth(ZNEAR, ZFAR);
ortho2(0, XMAXSCREEN, 0, YMAXSCREEN);

/*
 * display the object and manipulate it.
 * loop until user hits right mouse button.
 */
setcursor(0, COLOR(RED), (1 << getplanes()) - 1); /* default cursor */
callobj(everything);
swapbuffers();
while (!quit) {
    /*
     * check for mouse buttons activity
     */
    if (changed = qtest())
        switch (qread(&data)) {
            case MIDDLEMOUSE:
                if (!(Boolean) data)
                    break;
                switch (mv = check_menu(move_menu,
                                        move_menu_item, everything)) {
                    case M_NONE:
                        break;
                    case M_ADD_MARKER:
                        make_point();
                        break;
                    case M_REMOVE_MARKER:
                        delobj(point_obj);
                        break;
                    case M_REDRAW:
                        redraw_move();
                        break;
                    case M_ABORT_MOVE:
                        abort_move();
                        break;
                    case M_ACCEPT_MOVE:
                        end_move();
                        break;
                    case M_PICK:
                        start_move();
                        break;
                    case M_RENDER:
                        mv = check_menu(render_menu,
                                        render_menu_item, everything);
                        if (mv != M_NONE)

```

```

        break;
        render(mv);
    case M_LIGHT:
        position_light(everything);
        break;
    case M_SET_COLOR:
        mv = check_menu(color_menu,
            color_menu_item, everything);
        if (mv == M_NONE)
            break;
        set_point_color((Colorindex) mv);
        break;
    case M_STOP:
        quit = TRUE;
        break;
    }
    case RIIGHTMOUSE:
        if (! (Boolean) data)
            break;
        switch (mv = check_menu(main_menu,
            main_menu_item, everything)) {
        case M_NONE:
            break;
        case M_STOP:
            quit = TRUE;
            break;
        case M_OP:
            change_view(hither, yon, lortho_vie,
                lortho_vie, lortho_vie,
                "orthographic": "perspect");
            remake_main_menu();
            break;
        case M_PUSH:
            compact(m);
            v = (View *) malloc(sizeof (View));
            if (v == NULL) {
                flash(YELLOW);
                break;
            }
            v->v_scale = sc;
            bcopy(m, v->v_matrix, sizeof (Matrix));
            v->v_ortho = ortho_view;
            v->v_hither = hither;
            v->v_yon = yon;
            v->v_next = view_stack;
            view_stack = v;
        }
    case M_POP:
        if (view_stack == NULL) {
            flash(RED);
            break;
        }
        set_scale(view_stack->v_scale);
        set_transform(view_stack->v_matrix);
        hither = v->v_hither;
        yon = v->v_yon;
        change_view(hither, yon,
            view_stack->v_ortho);
        v = view_stack;
        view_stack = view_stack->v_next;
        (void) free((char *) v);
        break;
    case M_RESET:
        sc = DEFAULTSCALE;
        set_scale(sc);
        set_transform(ident);
        hither = HALFLEN;
        yon = HALFLEN;
        change_view(hither, yon, TRUE);
        ortho_view = TRUE;
        break;
    case M_ROCK:
        rocking = !rocking;
        break;
    case M_ROLL:
        rolling = !rolling;
        break;
    case M_OTHEREYE:
        if ((Boolean) data) {
            editobj(transformation_obj);
            objinsert(transformation_tag);
            if (showing_right)
                rotate(-60, 'y');
            else
                rotate(60, 'y');
            closeobj();
            showing_right = !showing_right;
            (void) strcpy(which_eye,
                !showing_right ?
                "right eye":
                "left eye");
            remake_main_menu();
        }
    }
}

```

deplct.c

```

break;
case M_INTENSITY:
    adjust_intensity(everything);
    break;
case M_READ_MAP:
    readcolormap();
    break;
default:
    warn("Unsupported menu value %d\n", menu_value);
    break;
}
break;
case LEFTMOUSE:
    pen_down = (Boolean) data;
    break;
}
}

/* check for mouse position/state for manipulating object
changed |= check_devices(pen_down, &sc,
                        &hither, &yon, ortho_view);
If (rocking)
    changed |= rock();
If (rolling)
    changed |= roll();
If (changed) {
    /* start working on the next frame
    */
    callobj(everything);
    swapbuffers();
    /* compact transformation matrix if necessary
    */
    if (nmods > MOD_THRESHOLD)
        compact(m);
}

/* reset the event queue
*/
}

undevice(RIGHTMOUSE);
undevice(MIDDLEMOUSE);
undevice(LEFTMOUSE);
qreset();
cursoff();
}

/* identity - make "m" an identity matrix
*/
identity(m)
Matrix m;
{
    register int i, j;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            m[i][j] = (i == j);
}

/* check_devices - check whether the mouse is over the devices
* The color of items (joystick or potentiometer) indicates its state:
* red = unselected
* yellow = mouse over item but button not depressed
* green = item selected
*/
Boolean
check_devices(selected, scale_factor, hither, yon, ortho_view)
Boolean selected;
float *scale_factor;
Coord *hither, *yon;
Boolean ortho_view;
{
    register Coord x, y;
    register int new_object;
    register Colorindex new_color;
    register Boolean changed;
    float xfactor, yfactor, diff;
    static int selected_object = NONE;
    static Boolean last_select = FALSE;
}

/* determine if the mouse is in a potentially active region
*/
x = getvaluator(MOUSEX);
y = getvaluator(MOUSEY);

```

depict.c

```
new_object = find_object(x, y);
/*
 * reset the color of the previously active item
 */
changed = FALSE;
if (selected_object != new_object || last_select != selected) {
    edtojb(everything);
    switch (selected_object) {
        case XYTRAN:
            objreplace(xytran_tag);
            color(COLOR(RED));
            break;
        case XYROT:
            objreplace(xyrot_tag);
            color(COLOR(RED));
            break;
        case ZTRAN:
            objreplace(ztran_tag);
            color(COLOR(RED));
            break;
        case ZROT:
            objreplace(zrot_tag);
            color(COLOR(RED));
            break;
        case HITHER:
            objreplace(hither_tag);
            color(COLOR(RED));
            break;
        case YON:
            objreplace(yon_tag);
            color(COLOR(RED));
            break;
        case SECTION:
            objreplace(section_tag);
            color(COLOR(RED));
            break;
        case SCALE:
            objreplace(scale_tag);
            color(COLOR(RED));
            break;
        case NORMAL:
            objreplace(normal_tag);
            color(COLOR(RED));
            break;
        case TANGENT:
            objreplace(tangent_tag);
            color(COLOR(RED));
            break;
    }
}

new_color = selected ? GREEN : YELLOW;
switch (new_object) {
    case XYTRAN:
        objreplace(xytran_tag);
        color(COLOR(new_color));
        break;
    case XYROT:
        objreplace(xyrot_tag);
        color(COLOR(new_color));
        break;
    case ZTRAN:
        objreplace(ztran_tag);
        color(COLOR(new_color));
        break;
    case ZROT:
        objreplace(zrot_tag);
        color(COLOR(new_color));
        break;
    case HITHER:
        objreplace(hither_tag);
        color(COLOR(new_color));
        break;
    case YON:
        objreplace(yon_tag);
        color(COLOR(new_color));
        break;
    case SECTION:
        objreplace(section_tag);
        color(COLOR(new_color));
        break;
    case SCALE:
        objreplace(scale_tag);
        color(COLOR(new_color));
        break;
    case NORMAL:
        objreplace(normal_tag);
        color(COLOR(new_color));
        break;
    case TANGENT:
        objreplace(tangent_tag);
        color(COLOR(new_color));
        break;
}
```

deplot.c

```

        color(COLOR(new_color));
        break;
    case NONE:
        break;
    }
    closeobj();

    selected_object = new_object;
    last_select = selected;

}

/*
 * operate on the new item
 */
if (selected && selected_object != NONE) {
    switch (selected_object) {
    case XYTRAN:
        ja_value(x, y, XYTRAN_X, XYTRAN_Y, &xfactor, &yfactor
        if (xfactor != 0 || yfactor != 0) {
            editobj(transformation_obj);
            objinsert(transformation_tag);
            translate(xfactor * 4, yfactor * 4, 0.);
            closeobj();
            changed = TRUE;
            nmods++;
        }
        break;
    case XYROT:
        ja_value(x, y, XYROT_X, XYROT_Y, &xfactor, &yfactor)
        if (xfactor != 0 || yfactor != 0) {
            editobj(transformation_obj);
            objinsert(transformation_tag);
            if (xfactor != 0) {
                rotate((limit) (xfactor * 50), 'Y');
                nmods++;
            }
            if (yfactor != 0) {
                rotate((limit) (-yfactor * 50), 'X');
                nmods++;
            }
            closeobj();
            changed = TRUE;
        }
        break;
    case ZTRAN:
        break;
    case ZROT:
        pot_value(x, ZTRAN_X, &xfactor);
        if (xfactor != 0) {
            break;
        }
        pot_value(x, ZROT_X, &xfactor);
        if (xfactor != 0) {
            editobj(transformation_obj);
            objinsert(transformation_tag);
            rotate((limit) (xfactor * 50), 'Z');
            closeobj();
            changed = TRUE;
            nmods++;
        }
        break;
    case SCALE:
        pot_value(x, SCALE_X, &xfactor);
        if (xfactor != 0) {
            *scale_factor = (1 + xfactor / 10);
            set_scale(*scale_factor);
            changed = TRUE;
        }
        break;
    case HITHER:
        break;
        pot_value(x, HITHER_X, &xfactor);
        if (xfactor != 0) {
            xfactor *= 5.;
            if ("hither - xfactor" > *yon) {
                flash(GREEN);
                break;
            }
            *hither -= xfactor;
            if ("hither + EYE <= 0" ) {
                *hither = 1 - EYE;
                if ("hither > *yon"
                    *hither = *yon;
            }
            change_view("hither, *yon, ortho_view");
            changed = TRUE;
        }
        break;
    case YON:
        break;
        pot_value(x, YON_X, &xfactor);

```

```

If (xfactor != 0) {
    xfactor *= 5.;
    if ((*yon - xfactor) < *hither) {
        flash(GREEN);
        break;
    }
    *yon -= xfactor;
    if (*hither > *yon)
        *yon = *hither;
    change_view(*hither, *yon, ortho_view);
    changed = TRUE;
}
break;
case SECTION:
    pot_value(x, SECTION_X, &xfactor);
    if (xfactor != 0) {
        xfactor *= 5.;
        diff = *yon - *hither;
        *hither -= xfactor;
        if (*hither + EYE <= 0)
            *hither = 1 - EYE;
        *yon = *hither + diff;
        change_view(*hither, *yon, ortho_view);
        changed = TRUE;
    }
    break;
case NORMAL:
    pot_value(x, NORMAL_X, &xfactor);
    move_move(0.0, 0.0, xfactor);
    changed = TRUE;
    break;
case TANGENT:
    js_value(x, y, TANGENT_X, TANGENT_Y,
            &xfactor, &yfactor);
    if (xfactor != 0 || yfactor != 0)
        move_move(xfactor, yfactor, 0.0);
    changed = TRUE;
    break;
}
return changed;
}

/*
 * find_object - find the item the mouse is resting on.
 * This routine used to depend on the layout
 * of the joysticks and potentiometers.
 */
find_object(x, y)
register Coord x, y;
{
    if (x < MIN_X || x > MAX_X)
        return NONE;
    if (y < XYTRAN_Y + JOYSTICK_HEIGHT && y > XYTRAN_Y)
        return XYTRAN;
    if (y < XYROT_Y + JOYSTICK_HEIGHT && y > XYROT_Y)
        return XYROT;
    if (y < TANGENT_Y + JOYSTICK_HEIGHT && y > TANGENT_Y)
        return TANGENT;
    if (y < NORMAL_Y + POTENTIOMETER_HEIGHT && y > NORMAL_Y)
        return NORMAL;
    if (y < YON_Y + POTENTIOMETER_HEIGHT && y > YON_Y)
        return YON;
    if (y < SECTION_Y + POTENTIOMETER_HEIGHT && y > SECTION_Y)
        return SECTION;
    if (y < SCALE_Y + POTENTIOMETER_HEIGHT && y > SCALE_Y)
        return SCALE;
    if (y < ZTRAN_Y + POTENTIOMETER_HEIGHT && y > ZTRAN_Y)
        return ZTRAN;
    if (y < HITHER_Y + POTENTIOMETER_HEIGHT && y > HITHER_Y)
        return HITHER;
    if (y < ZROT_Y + POTENTIOMETER_HEIGHT && y > ZROT_Y)
        return ZROT;
    return NONE;
}

/*
 * pot_value - returns a number between -0.5 and 0.5 for a potentiometer
 */
pot_value(x, minx, value)
float x, minx;
float *value;
{
    *value = (x - (minx + POTENTIOMETER_WIDTH / 2)) / POTENTIOMETER_WIDTH;
}

/*
 * js_value - returns numbers between -0.5 and 0.5 for a joystick
 */
js_value(x, y, minx, miny, xvalue, yvalue)
float x, y, minx, miny;
float *xvalue, *yvalue;
{
    *xvalue = (x - (minx + JOYSTICK_WIDTH / 2)) / JOYSTICK_WIDTH;
}

```

deplct.c

```

}
    *yvalue = (y - (miny + JOYSTICK_HEIGHT / 2)) / JOYSTICK_HEIGHT;
}

/*
 * compact - compacts the transformation object back down to a
 * single multmatrix()
 */
compact(matrix)
Matrix matrix;
{
    /*
     * show the wait-a-second cursor to let user know we're working
     */
    setcursor(1, COLOR(RED), (1 << getplanes()) - 1);

    /*
     * figure out what the cumulative transformation is
     */
    pushmatrix();
        loadmatrix(ident);
        callobj(transformation_obj);
        getmatrix(matrix);
    popmatrix();

    /*
     * delete the inserted incremental transformations and put in
     * the cumulative one
     */
    set_transform(matrix);

    /*
     * let user know we're ready to go again
     */
    setcursor(0, COLOR(RED), (1 << getplanes()) - 1); /* default cursor */
}

/*
 * set_transform - changes the current transformation to the given
 * matrix. All current mods are lost.
 */
set_transform(matrix)
Matrix matrix;
{
    editobj(transformation_obj);
    objdelete(transformation_tag, end_transform_tag);
    objinsert(transformation_tag);
    multmatrix(matrix);
}
}
    closeobj();
    nmods = 0;
}

/*
 * set_scale - changes the current scaling factor to the given value.
 */
set_scale(sc)
float sc;
{
    char buf[BUFSIZ];

    editobj(scale_obj);
    objreplace(size_tag);
    scale(sc, sc, sc);
    closeobj();
    (void) sprintf(buf, "%.5.2f", sc);
    editobj(everything);
    objreplace(scale_label_tag);
    charstr(buf);
    closeobj();
}

/*
 * change_view - changes the viewing transformation to be orthogonal or
 * perspective depending on the value of ortho_view
 */
change_view(hither, yon, new_view)
Coord hither, yon;
Boolean new_view;
{
    editobj(view_obj);
    if (new_view) {
        if (lortho_view) {
            objdelete(window_tag, end_window_tag);
            objinsert(window_tag);
            ortho(-HALFLEN, HALFLEN, -HALFLEN, -HALFLEN, HALFLEN,
                hither, yon);
            ortho_view = TRUE;
        }
        else {
            objreplace(window_tag);
            ortho(-HALFLEN, HALFLEN, -HALFLEN, -HALFLEN, HALFLEN,
                hither, yon);
        }
    }
    else {
}
}
}
    else {
}
}
}

```

depict.c

```

    }
    else {
        objreplaces(window_tag);
        perspective(300, 1., EYE + hither, EYE + yon);
        lookat(0., 0., EYE, 0., 0., 0);
        ortho_view = FALSE;
    }
}
else {
    objreplaces(window_tag);
    perspective(300, 1., EYE + hither, EYE + yon);
}
closeobj();
}
/*
 * flash - flashes the screen in the given color.
 * flash() assumes that the current buffer contains nothing
 */
flash(hue)
Colorindex hue;
{
    Colorindex cur_color;

    cur_color = getcolor();
    color(COLOR(hue));
    clear();
    swapbuffers();
    color(COLOR(cur_color));
    swapbuffers();
}

/*
 * roll - roll the object about the y axis one degree
 */
Boolean
roll()
{
    editobj(transformation_obj);
    objinsert(transform_tag);
    rotate(10, 'y');
    closeobj();
    nmods++;
    return TRUE;
}
/*
 * rock - rock the object back and forth about the y axis
 */
Boolean
rock()
{
    static double rock_angle = 0.;
    double sin();

    editobj(transformation_obj);
    objinsert(transform_tag);
    rotate((int) (sin(rock_angle) * 20), 'y');
    closeobj();
    nmods++;
    rock_angle += 0.1;
    if (rock_angle >= 6.2832)
        rock_angle = 0.;
    return TRUE;
}
}

```



```

display.c
/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
#include <stdio.h>
#ifdef IRIS
#include <signal.h>
#include <gl.h>
#endif

#ifdef TRUE
#define TRUE 1
#define FALSE 0
#endif

usage(progname)
char *progname;
{
    printf(stderr, "Usage: %s [-i input_file] [-v] [-n nstripes]\n",
        progname);
    printf(stderr, "\t-i input_file\tOutput file from \"uniform\"\n");
    printf(stderr, "\t-v\t\t\tVerbose mode\n");
    printf(stderr, "\t-n nstripes\tNumber of stripes per triangle\n");
}

#ifdef IRIS
Object bild_obj;
Object point_obj;
Object marker_obj;
Object move_obj, move_surface_obj, move_axis_obj;
int stored_map = 0;
#endif

main(ac, av)
int ac;
char **av;
{
    register int c;
    char *infile;
    FILE *infp;
    int verbose;
    int nstripe;
    extern int optind;
    extern char *optarg;
    int onintr();

    nstripe = 1;
    verbose = FALSE;
    infile = NULL;
    while ((c = getopt(ac, av, "i:v:n:h")) != EOF)
        switch (c) {
            case 'i':
                infile = optarg;
                break;
            case 'v':
                verbose = TRUE;
                break;
            case 'n':
                nstripe = atoi(optarg);
                break;
            case 'h':
                usage(av[0]);
                exit(0);
        }
    if (optind < ac) {
        usage(av[0]);
        exit(1);
    }
    if (nstripe <= 0) {
        printf(stderr, "Number of stripes reset from %d to 1\n",
            nstripe);
        nstripe = 1;
    }
    if (infile == NULL) {
        infile = "standard input";
        infp = stdin;
    }
    else if ((infp = fopen(infile, "r")) == NULL) {
        perror(infile);
        exit(1);
    }
    if (verbose)
        printf(stderr, "Read input file \"%s\"\n", infile);
    read_input(infp);
    (void) fclose(infp);
    if (verbose)
        printf(stderr, "Initialize update routine\n");
    init_update(nstripe);
}
#endif
}

```

display.c

```
#ifndef IRIS
ginit();
(void) signal(SIGINT, onintr);
If (verbose)
    fprintf(stderr, "Setup color map\n");
map_colors();
stored_map = 1;
setpats();
#endif

If (verbose)
    fprintf(stderr, "Create graphics objects\n");
make_global();

#ifndef IRIS
If (verbose)
    fprintf(stderr, "Depict object\n");
depict();
#endif

#ifndef IRIS
color(BLACK);
clear();
swapbuffers();
restore_map();
gexit();
exit(0);
}

#endif
char *
emalloc(n)
int n;
{
    register char *cp;
    extern char *malloc();

    If ((cp = malloc(n)) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    return cp;
}

onintr()
```

```

display.h
/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
#ifndef IRIS
#include <gl.h>
#endif

typedef struct point_def
double
int
struct point_def
struct edge_def
double
int
#define IRIS Colorindex
#endif
POINT;
}

typedef struct edge_def {
struct point_def
struct edge_def
struct triangle_def
double
double
double
double
int
IRIS
Object
EDGE;
}

typedef struct side_def {
struct edge_def
double
int
SIDE;
}

typedef struct triangle_def
struct side_def
struct point_def
int
IRIS
Object;
}

{
coord[3];
npoint;
**point;
**edge;
normal[3];
status;
color;

*point[2];
*cbt_edge[2][2];
*triangle[2];
*cbt_point[2][2];
ipoint[2][3];
epoint[3][3];
cbt[2][3];
status;
object;

*edge;
ctrl[2][3];
status;

{
side[3];
*point[3];
status;
object;
}

#endif TRIANGLE;
extern POINT *point;
extern EDGE *edge;
extern TRIANGLE *triangle;

extern int npoint;
extern int nedge;
extern int ntriangle;

#define S_NORMAL 0x1
#define S_INTERMEDIATE 0x2
#define S_CBT 0x4
#define S_EDGE (S_CBT | S_INTERMEDIATE)
#define S_SIDE 0x8
#define S_PATCH 0x10

#define P_P004 0
#define P_P103 1
#define P_P202 2
#define P_P301 3
#define P_P400 4
#define P_P112V 5
#define P_P211V 6
#define P_P013 7
#define P_P112U 8
#define P_P211W 9
#define P_P310 10
#define P_P022 11
#define P_P121U 12
#define P_P121W 13
#define P_P220 14
#define P_P031 15
#define P_P130 16
#define P_P040 17
#define P_SIZE 18

```

Input.c

```

/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 *
 * #include <stdio.h>
 * #include <ctype.h>
 * #include "display.h"
 */
#define DEBUG /* */

POINT *point;
EDGE *edge;
TRIANGLE *triangle;

#ifdef IRIS
#define FORMAT "%d\t%f,%f,%f)\t%d\n"
#else
#define FORMAT "%d\t%f,%f,%f)\t%d\n"
#endif

int npoint;
int nedge;
int ntriangle;

/* read_input:
 * Read the input and construct the data structure
 */
read_input(fp)
FILE *fp;
{
    read_points(fp);
    read_triangles(fp);
    generate_edges();
    generate_sides();
    generate_cbx();
    DEBUG
    dump_data();
}

#ifdef
#endif

/* read_points:
 * Read the points (coordinates and neighbors)
 */
read_points(fp)

```

```

FILE
{
    *fp;
    register int i;
    register POINT *pp;
    int n;
    char buf[BUFSIZ];
    extern char *emalloc();

    edges(buf, sizeof buf, fp);
    if (scanf(buf, "%d vertices", &npoint) != 1)
        error("No vertex count\n");
    point = (POINT *) emalloc(sizeof (POINT) * npoint);
    for (i = 0; i < npoint; i++) {
        pp = &point[i];
        pp->status = 0;
        edges(buf, sizeof buf, fp);
        if (scanf(buf, FORMAT, &n, &pp->coord[0],
                &pp->coord[1], &pp->coord[2], &pp->npoint) != 5)
            error("Bad vertex input line\n");
        if (n != i)
            error("Vertex input out of sync\n");
        edges(buf, sizeof buf, fp);
        generate_neighbor(buf, pp, pp->npoint);
    }
}

/* generate_neighbor:
 * Generate the neighbor array for the given point
 */
generate_neighbor(s, pp, npt)
char *s;
POINT *pp;
int npt;
{
    register int i;
    register char *cp;
    int n;

    pp->point = (POINT **) emalloc(sizeof (POINT *) * npt);
    cp = s;
    for (i = 0; i < npt; i++) {
        if (scanf(cp, "%d", &n) != 1)
            error("Bad neighbor list\n");
        pp->point[i] = &point[n];
        while (isspace(*cp))
            cp++;
    }
}

```

input.c

```

    while (!isspace(*cp))
        cp++;
}

/*
 * read_triangles:
 *   Read in triangles (vertices of the triangle)
 */
read_triangles(fp)
FILE *fp;
{
    register int
    register TRIANGLE
    int
    char
    extern char
    fgets(buf, sizeof buf, fp);
    if (sscanf(buf, "%d triangles", &ntriangle) != 1)
        error("No triangle count\n");
    triangle = (TRIANGLE *) malloc(sizeof (TRIANGLE) * ntriangle);
    for (i = 0; i < ntriangle; i++) {
        tp = &triangle[i];
        tp->status = 0;
        fgets(buf, sizeof buf, fp);
        if (sscanf(buf, "%d %d %d", &p1, &p2, &p3) != 3)
            error("Bad triangle data\n");
        tp->point[0] = &point[p1];
        tp->point[1] = &point[p2];
        tp->point[2] = &point[p3];
    }
}

/*
 * generate_edges:
 *   Compute the edges and sides
 */
generate_edges()
{
    register int
    register POINT
    register int
    EDGE
    EDGE
    extern char
    i;
    *pp, *endpp;
    max_edge;
    *add_edge();
    *find_edge();
    *malloc();

    max_edge = 0;
    endpp = &point[npoint];
    for (pp = point; pp < endpp; pp++)
        max_edge += pp->npoint;
#ifdef DEBUG
    if (max_edge & 1)
        error("Odd number of sides\n");
#endif
    max_edge = max_edge / 2;
    edge = (EDGE *) malloc(sizeof (EDGE) * max_edge);
    nedge = 0;

    /*
     * Add the edges sorted by first point
     */
    for (pp = point; pp < endpp; pp++) {
        pp->edge = (EDGE **) malloc(sizeof (EDGE *) * pp->npoint);
        for (i = 0; i < pp->npoint; i++)
            if (pp->point[i] > pp)
                else
                    pp->edge[i] = add_edge(pp, pp->point[i]);
            else
                pp->edge[i] = find_edge(pp->point[i], pp);
    }
#ifdef DEBUG
    if (nedge != max_edge)
        fprintf(stderr, "Nedge = %d, max_edge = %d\n", nedge, max_edge);
#endif
}

/*
 * add_edge:
 *   Add an edge to edge list
 */
EDGE *
add_edge(from, to)
POINT *from, *to;
{
    register EDGE *ep;
    register int i;
    double
    delta[3];

    ep = &edge[nedge++];
    ep->point[0] = from;
    ep->point[1] = to;
    ep->triangle[0] = ep->triangle[1] = NULL;
    ep->status = 0;
    for (i = 0; i < 3; i++) {

```

Input.c

```

    delta[j] = to->coord[j] - from->coord[j];
    ep->point[0][j] = from->coord[j] + delta[j] / 4;
    ep->point[1][j] = from->coord[j] + delta[j] * 3 / 4;
}
return ep;
}

/* find_edge:
 * Find an edge in the existing edge list
 */
EDGE*
find_edge(from, to)
POINT *from, *to;
{
    register EDGE *hi, *lo, *mid;
    register POINT *tmp;

    if (from > to) {
        tmp = from;
        from = to;
        to = tmp;
    }

    lo = edge;
    hi = &edge[nedge];
    while (hi > lo) {
        mid = lo + (hi - lo) / 2;
        if (mid->point[0] == from)
            break;
        else if (mid->point[0] > from)
            hi = mid;
        else
            lo = mid + 1;
    }
    if (mid->point[0] != from) {
        fprintf(stderr, "A: Cannot find edge (%x,%x)\n", from, to);
        return NULL;
    }
    for (lo = mid; lo >= edge; lo--) {
        if (lo->point[0] != from)
            break;
        if (lo->point[1] == to)
            return lo;
    }
    hi = &edge[nedge];
    for (lo = mid + 1; lo < hi; lo++) {
        if (lo->point[0] != from)
            break;
        if (lo->point[1] == to)
            return lo;
    }
}

}
    if (lo->point[0] != from)
        break;
    if (lo->point[1] == to)
        return lo;
}
fprintf(stderr, "B: Cannot find edge (%x,%x)\n", from, to);
abort();
/* NOTREACHED */
}

/* generate_sides:
 * Generate the triangle sides
 */
generate_sides()
{
    register TRIANGLE *tp, *endtp;
    register int i, j;
    register EDGE *ep, *endep;

    endtp = &triangle[ntriangle];
    for (tp = triangle; tp < endtp; tp++)
        for (i = 0; i < 3; i++) {
            j = (i + 1) % 3;
            add_side(tp, &tp->sides[i], tp->point[i], tp->point[j]);
        }
}

/* add_side:
 * Add a side and update the edge data structure
 */
add_side(tp, ep, from, to)
TRIANGLE *tp;
SIDE *sp;
POINT *from, *to;
{
    register EDGE *ep;
    register int i;
}

```

Input.c

```

ep = find_edge(from, to);
for (i = 0; i < 2; i++)
    if (ep->triangle[i] == NULL)
        break;
#ifdef DEBUG
    if (i >= 2)
        fprintf(stderr, "Edge with more than 2 triangles %x, %x\n", ep, tp);
#endif
ep->triangle[i] = tp;
sp->edge = ep;
}
/*
 * generate_cbt:
 * Generate the cbt pointers for each edge
 */
generate_cbt()
{
    register EDGE *ep, *endep;

    endep = &edge[nedge];
    for (ep = edge; ep < endep; ep++) {
        add_cbt(ep->triangle[0], ep, ep->cbt_point[0], ep->cbt_edge[0]);
        add_cbt(ep->triangle[1], ep, ep->cbt_point[1], ep->cbt_edge[1]);
    }
}
/*
 * add_cbt:
 */
add_cbt(tp, ep, cb, cbt_edge)
    TRIANGLE *tp;
    EDGE **ep;
    double **dp;
    EDGE **cbt_edge;
{
    register int i, j;
    register SIDE *sp;

    for (i = 0; i < 3; i++) {
        sp = &tp->side[i];
        if (sp->edge == ep)
            continue;
        if (sp->edge->point[0] == ep->point[0]
            || sp->edge->point[1] == ep->point[0])
            j = 0;
        else
            j = 1;
        cbt_edge[j] = sp->edge;
        if (sp->edge->point[0] == ep->point[j])
            dp[j] = sp->edge->ipoint[0];
        else
            dp[j] = sp->edge->ipoint[1];
    }
}
/*
 * efgets:
 * Same as fgets but exits on failure
 */
efgets(buf, size, fp)
    char *buf;
    int size;
    FILE *fp;
{
    if (fgets(buf, size, fp) == NULL)
        error("efgets() failed\n");
}
/*
 * error:
 */
error(s)
    char *s;
{
    fputs(s, stderr);
    exit(1);
}
#ifdef DEBUG
/*
 * dump_data:
 * Dump the data back out in exactly the same format
 * to make sure we read everything in correctly
 * Also dump a bid format file from the edge list
 */
dump_data()
{
    register int i;
    register POINT *pp;
    register TRIANGLE *tp;
}

```

input.c

```

register EDGE
register FILE
register POINT
register TRIANGLE
register EDGE
double
double
double

*ep;
*fp;
*endpp;
*endtp;
min[3], max[3], coord[3];
sc;
*dp;

if ((fp = fopen("display.debug", "w")) == NULL) {
    perror("display.debug");
    exit(1);
}
fprintf(fp, "%d vertices\n", npoint);
endpp = &point[npoint];
for (pp = point; pp < endpp; pp++) {
    fprintf(fp, "%d\t(%f %f %f)\n", pp->coord[0], pp->coord[1], pp->coord[2]);
    for (i = 0; i < pp->npoint; i++)
        fprintf(fp, "%5d", pp->point[i] - point);
    (void) puts("\n", fp);
}

fprintf(fp, "%d triangles\n", ntriangle);
endtp = &triangle[ntriangle];
for (tp = triangle; tp < endtp; tp++) {
    fprintf(fp, "%5d %5d %5d\n", tp->point[0] - point,
        tp->point[1] - point, tp->point[2] - point);
    (void) fclose(fp);
}

/* Now for the bid format file
*/
if ((fp = fopen("display.bid", "w")) == NULL) {
    perror("display.bid");
    exit(1);
}
for (i = 0; i < 3; i++) {
    min[i] = point[0].coord[i];
    max[i] = point[0].coord[i];
}
endpp = &point[npoint];
for (pp = &point[1]; pp < endpp; pp++) {
    for (i = 0; i < 3; i++) {
        if (pp->coord[i] < min[i])
            min[i] = pp->coord[i];
        else if (pp->coord[i] > max[i])
            max[i] = pp->coord[i];
    }
    sc = max[0] - min[0];
    if (max[1] - min[1] > sc)
        sc = max[1] - min[1];
    sc = 140.0 / sc;
    fprintf(fp, "scale %.3f %.3f %.3f\n", sc, sc, sc);
    for (i = 0; i < 3; i++)
        coord[i] = (min[i] + max[i]) / 2;
    fprintf(fp, "tran %.3f %.3f %.3f\n", -coord[0], -coord[1], -coord[2]);
    fprintf(fp, "color 1\n");
    endep = &edge[nedge];
    for (ep = edge; ep < endep; ep++) {
        pp = ep->point[0];
        fprintf(fp, ".m %.3f %.3f %.3f\n", pp->coord[0], pp->coord[1],
            pp->coord[2]);
        pp = ep->point[1];
        fprintf(fp, ".d %.3f %.3f %.3f\n", pp->coord[0], pp->coord[1],
            pp->coord[2]);
    }
    fprintf(fp, "color 2\n");
    endep = &edge[nedge];
    for (ep = edge; ep < endep; ep++) {
        dp = ep->cbt_point[0][0];
        fprintf(fp, ".m %.3f %.3f %.3f\n", dp[0], dp[1], dp[2]);
        dp = ep->cbt_point[1][0];
        fprintf(fp, ".d %.3f %.3f %.3f\n", dp[0], dp[1], dp[2]);
        dp = ep->cbt_point[0][1];
        fprintf(fp, ".m %.3f %.3f %.3f\n", dp[0], dp[1], dp[2]);
        dp = ep->cbt_point[1][1];
        fprintf(fp, ".d %.3f %.3f %.3f\n", dp[0], dp[1], dp[2]);
    }
    (void) fclose(fp);
}
}
#endif

```


mark.c

```
/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
#include <stdio.h>
#include "display.h"

/*
 * mark_point:
 * Mark a point as needing normal recomputed
 */
mark_point(target, remake)
POINT *target;
int remake;
{
    register int i;

    target->status &= ~S_NORMAL;
    mark_edge(target, remake);
    for (i = 0; i < target->npoint; i++) {
        target->point[i]->status &= ~S_NORMAL;
        mark_edge(target->point[i], remake);
    }
}

/*
 * mark_edge:
 * Mark an edge as needing intermediate recomputed
 */
mark_edge(target, remake)
POINT *target;
int remake;
{
    register int i;
    register EDGE *ep;

    for (i = 0; i < target->npoint; i++) {
        ep = target->edge[i];
        ep->status &= ~S_EDGE;
        ep->cbt_edge[0]->status &= ~S_CBT;
        ep->cbt_edge[1]->status &= ~S_CBT;
        ep->cbt_edge[1][0]->status &= ~S_CBT;
        ep->cbt_edge[1][1]->status &= ~S_CBT;
        if (!remake)
            continue;
        mark_triangle(ep);
    }
}

/*
 * mark_triangle:
 * Mark the triangles attached to this edge
 */
mark_triangle(target)
EDGE *target;
{
    register int i;
    register SIDE *sp;

    target->triangle[0]->status &= ~S_PATCH;
    target->triangle[1]->status &= ~S_PATCH;
    for (i = 0; i < 3; i++) {
        sp = &target->triangle[0]->side[i];
        if ((sp->edge->status & S_EDGE) != S_EDGE)
            sp->status &= ~S_SIDE;
        sp = &target->triangle[0]->side[i];
        if ((sp->edge->status & S_EDGE) != S_EDGE)
            sp->status &= ~S_SIDE;
    }
}

/*
 * mark_patch:
 * Mark all triangles that have the given point as a vertex
 */
mark_patch(pp)
register POINT *pp;
{
    register int i;

    for (i = 0; i < pp->npoint; i++) {
        pp->edge[i]->status &= ~S_EDGE;
        pp->edge[i]->triangle[0]->status &= ~S_PATCH;
        pp->edge[i]->triangle[1]->status &= ~S_PATCH;
    }
}

```

menu.c

```

/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
#include <stdio.h>
#include <gl.h>
#include <device.h>
#include "colors.h"
#include "menu.h"

/*
 * make_menu - make an object which contains the menu items
 */
Menu *
make_menu(item, m_size)
MENU_ITEM *item;
int m_size;
{
    register int i;
    register int menu_width, menu_height;
    register int height, width;
    register int menu_size;
    Menu *menu;
    char *malloc();

    menu_size = m_size / sizeof item[0];
    if (menu_size <= 0)
        return NULL;

    /*
     * allocate space for this menu
     */
    if ((menu = (Menu *) malloc(sizeof (Menu))) == NULL) {
        fprintf(stderr, "out of memory\n");
        return NULL;
    }

    /*
     * Figure out the dimension of the menu
     */
    height = getheight() + 8;
    menu_height = height * menu_size;
    menu_width = -1;
    for (i = 0; i < menu_size; i++) {
        width = strwidth(item[i].name);
        if (width > menu_width)
            menu_width = width;
    }

    menu_width = width;
    menu_width += 8;

    /*
     * create the menu object
     */
    menu->menu_object = genobj();
    menu->menu_tag = (Tag *) malloc(menu_size * sizeof (Tag));
    if (menu->menu_tag == NULL) {
        (void) free((char *) menu);
        fprintf(stderr, "Out of memory\n");
        return NULL;
    }
    for (i = 0; i < menu_size; i++)
        menu->menu_tag[i] = gentag();
    menu->menu_width = menu_width;
    menu->menu_height = menu_height;
    menu->menu_size = menu_size;
    menu->menu_last_item = menu_size - 1;
    makeobj(menu->menu_object);
    color(COLOR(BLACK));
    rectf(0, 0, (Coord) menu_width, (Coord) menu_height);
    color(COLOR(BLUE));
    rect(0, 0, (Coord) menu_width, (Coord) menu_height);
    for (i = 0; i < menu_size; i++) {
        width = strwidth(item[i].name);
        cmov((Coord) (menu_width - width) / 2,
            (Coord) (height * (menu_size - i - 1) + 4, 0.);
        maketag(menu->menu_tag[i]);
        color(COLOR(RED));
        charstr(item[i].name);
    }
    closeobj();
    return menu;
}

/*
 * free_menu - release menu data structure
 */
free_menu(menu)
Menu *menu;
{
    (void) free((char *) menu->menu_tag);
    (void) free((char *) menu);
}

```

menu.c

```

# define INMENU(i)      (-1 < i) && (i) < menu->menu_size)
/*
 * check_menu - display the menu and return the selected item number
 */
check_menu(menu, item, display_obj)
Menu *menu;
MENU_ITEM *item;
Object display_obj;
{
    register int      x, y;
    register int      item_number;
    register int      height;
    register int      menu_x, menu_y;
    Boolean            quit;
    short             data;

    /*
     * Get the menu location from the mouse
     */
    height = menu->menu_height / menu->menu_size;
    menu_x = getvaluator(MOUSEX) - menu->menu_width / 2;
    menu_y = getvaluator(MOUSEY) - menu->menu_height
              + (menu->menu_last_item + 0.5) * height;
    menu->menu_last_item = -1;

    /*
     * Loop until user selects an object or moves the cursor off
     * the menu
     */
    qreset();
    quit = FALSE;
    while (!quit) {
        x = getvaluator(MOUSEX) - menu_x;
        y = getvaluator(MOUSEY) - menu_y;
        if (x < 0 || x > menu->menu_width)
            return M_NONE;
        if (y < 0 || y > menu->menu_height)
            return M_NONE;
        item_number = (menu->menu_height - y - 4) / height;
        if (!INMENU(item_number))
            return M_NONE;
        if (item_number != menu->menu_last_item) {
            edfobj(menu->menu_object);
            if (INMENU(menu->menu_last_item)) {
                objreplace(menu->menu_tag[menu->menu_last_item]);
                color(COLOR(RED));
            }
            objreplace(menu->menu_tag[item_number]);
            color(COLOR(YELLOW));
            closeobj();
            menu->menu_last_item = item_number;
            callobj(display_obj);
            pushmatrix();
            translate(menu_x, menu_y, 0.);
            callobj((menu->menu_object);
            popmatrix();
            swapbuffers();
        }
    }
    if (qtest()) /*
     * If the mouse button clicked, return the item;
     * otherwise, just act as if nothing happened
     */
        switch (qread(&data)) {
            case LEFTMOUSE:
            case MIDDLEMOUSE:
            case RIGHTMOUSE:
                if ((Boolean) data)
                    return item[item_number].value;
                break;
            default:
                quit = TRUE;
                break;
        }
    }
    return M_NONE;
}

```

```

menu.h
/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
struct menu_def {
    Object menu_object; /* Display object */
    Tag *menu_tag; /* Tags for colors */
    int menu_width, menu_height; /* Dimensions */
    int menu_size; /* Number of items */
    int menu_last_item; /* Last item picked */
};

typedef struct menu_def Menu;

extern Menu *make_menu();

typedef struct {
    char *name;
    int value;
} MENU_ITEM;

#define M_NONE -1
#define M_STOP 0

#define M_OP 1
#define M_PUSH 2
#define M_POP 3
#define M_RESET 4
#define M_ROCK 5
#define M_ROLL 6
#define M_OTHEREYE 7
#define M_ZBUFFER 8
#define M_INTENSITY 9

#define M_ADD_MARKER 1
#define M_REMOVE_MARKER 2
#define M_REDRAW 3
#define M_ABORT_MOVE 4
#define M_ACCEPT_MOVE 5
#define M_PICK 6
#define M_RENDER 7
#define M_LIGHT 8
#define M_SET_COLOR 9

#define C_RED RED
#define C_GREEN GREEN

```

```

#define C_BLUE BLUE
#define C_MAGENTA MAGENTA
#define C_YELLOW YELLOW
#define C_CYAN CYAN
#define C_WHITE WHITE

#define T_THATCHED 1
#define T_HALFTONE 2
#define T_CROSSHATCH 3
#define T_NONE 4

#define HALFTONE 1
#define CROSSHATCH 2

```

misc.c

```
/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
#include <stdio.h>
#include <gl.h>
#include "menu.h"

warn(fmt, a1, a2, a3, a4, a5, a6, a7, a8)
char
{
    fprintf(stderr, fmt, a1, a2, a3, a4, a5, a6, a7, a8);
}

static Pattern16 halftone[] = {
    0x5555, 0xAAAA, 0x5555, 0xAAAA, 0x5555, 0xAAAA, 0x5555, 0xAAAA,
    0x5555, 0xAAAA, 0x5555, 0xAAAA, 0x5555, 0xAAAA, 0x5555, 0xAAAA
};

static Pattern16 crosshatch[] = {
    0x5555, 0x2222, 0x5555, 0x8888, 0x5555, 0x2222, 0x5555, 0x8888,
    0x5555, 0x2222, 0x5555, 0x8888, 0x5555, 0x2222, 0x5555, 0x8888
};

setpats()
{
    defpattern(HALFTONE, 16, halftone);
    defpattern(CROSSHATCH, 16, crosshatch);
}
```

move.c

```

/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 *
 */
#include <stdio.h>
#include <gl.h>
#include <device.h>
#include "colors.h"
#include "menu.h"
#include "display.h"

#define PICK 2
#define FRAME_MAX 765

static Cursor cpick = {
    0x84C9, 0x852A, 0xE50C, 0x950C, 0x950C,
    0x952A, 0xE4C9, 0x0000, 0x0180,
    0x0380, 0x0700, 0x8E00, 0xDC00,
    0xF800, 0xF000, 0xF800, 0xFC00,
};

static POINT *move_point;
static double save_coord[3];
static double new_pos[3], new_coord[3];
static Matrix move_mat, inv_move_mat;
}

/* pick_control: Pick a control point
 */
POINT *
pick_control()
{
    short quit, which_cursor;
    short hits, type, val, namebuffer[50];
    short old_index;
    Colorindex old_color, old_wtm;
    Boolean old_b;
    Coord x, y;
    POINT *picked;
    int had_point;
    static int pick_flag = FALSE;
    extern Matrix ident;
    extern Object everything, point_obj;
    extern Object view_obj, transformation_obj, scale_obj;

    if (!pick_flag) {
        pick_flag = TRUE;
        picksize(4, 4);
        defcursor(PICK, cpick);
        curorigin(PICK, 0, 15);
    }

    getcursor(&old_index, &old_color, &old_wtm, &old_b);
    setcursor(PICK, COLOR(WHITE), old_wtm);
    had_point = isobj(point_obj);
    if (!had_point)
        make_point();
    callobj(everything);
    swapbuffers();

    qreset();
    quit = FALSE;
    picked = NULL;
    while (!quit) {
        type = qread(&val);
        if (val == 0)
            continue;
        switch (type) {
            case LEFTMOUSE:
            case MIDDLEMOUSE:
                pushviewport();
                pushattributes();
                pushmatrix();
                loadmatrix(ident);
                viewport(0, YMAXSCREEN, 0, YMAXSCREEN);
                pick(namebuffer, 50);
                callobj(view_obj);
                callobj(transformation_obj);
                callobj(scale_obj);
                callobj(point_obj);
                hits = endpick(namebuffer);
                popmatrix();
                popattributes();
                popviewport();
                if (hits == 1) {
                    picked = &point(namebuffer[1]);
                    quit = TRUE;
                }
                break;
            case RIGHTMOUSE:
                quit = TRUE;
                break;
        }
    }
}

```

MOVE.C

```

    }
    (void) fread(&val);

}

setcursor(old_index, old_color, old_wtm);
if (!head_point)
    delobj(point_obj);
callobj(everything);
swapbuffers();
return picked;
}

/*
 * start_move:
 */
Set up for moving the given point
start_move()
{
    register POINT *pp;
    register int i;
    double at[3], from[3], size;
    extern Object move_obj, move_axis_obj;
    extern Object move_surface_obj, marker_obj;
    double build_translation();

/*
 * Terminate any previous moving point
 */
if (move_point != NULL)
    end_move();
pp = pick_control();

/*
 * Save the original values
 */
move_point = pp;
save_coord[0] = pp->coord[0];
save_coord[1] = pp->coord[1];
save_coord[2] = pp->coord[2];

/*
 * Set up the matrices for converting from changed values
 * to Cartesian coordinates
 */
at[0] = pp->coord[0] + pp->normal[0];
at[1] = pp->coord[1] + pp->normal[1];
at[2] = pp->coord[2] + pp->normal[2];

viewat(move_mat, inv_move_mat, pp->coord, at, pp->point[0]->coord);

/*
 * Set up the reference axis and initial position
 */
new_pos[0] = new_pos[1] = new_pos[2] = 0;
makeobj(move_surface_obj);
size = build_translation();
color(COLOR(GREEN));
shadrange(RANGE(GREEN));
move((Coord) move_point->coord[0], (Coord) move_point->coord[1],
      (Coord) move_point->coord[2]);
callobj(marker_obj);
closeobj();
makeobj(move_axis_obj);
from[0] = from[1] = from[2] = 0;
color(COLOR(YELLOW));
shadrange(RANGE(YELLOW));
for (i = 0; i < 3; i++) {
    from[i] = size * 10;
    mult_vector(at, from, inv_move_mat);
    move((Coord) at[0], (Coord) at[1], (Coord) at[2]);
    from[i] = -size * 10;
    mult_vector(at, from, inv_move_mat);
    draw((Coord) at[0], (Coord) at[1], (Coord) at[2]);
    from[i] = 0;
}
closeobj();
makeobj(move_obj);
pushmatrix();
callobj(move_surface_obj);
callobj(move_axis_obj);
popmatrix();
closeobj();

/*
 * move_move:
 * Move the moving point
 */
move_move(dx, dy, dz)
double dx, dy, dz;
{
    extern Object move_surface_obj, marker_obj;

/*
 * ignore nops
 */
}

```

move.c

```

*/
if (move_point == NULL)
    return;
if (dx == 0 && dy == 0 && dz == 0)
    return;

/* * Compute new coordinate
*/
new_pos[0] += dx;
new_pos[1] += dy;
new_pos[2] += dz;
mult_vector(move_point->coord, new_pos, inv_move_mat);

/* * Remake edges so user can see the surface change
*/
makeobj(move_surface_obj);
build_translation();
color(COLOR(GREEN));
shadrange(RANGE(GREEN));
mark_point(move_point, FALSE);
update(FALSE);
move(move_point->coord[0], move_point->coord[1],
      move_point->coord[2]);
closeobj();

}

/* * redraw_move:
* Redraw the surface including triangle interiors
*/
redraw_move()
{
    extern Object move_surface_obj, marker_obj;

    if (move_point == NULL)
        return;
    mark_point(move_point, TRUE);
    update(TRUE);
    makeobj(move_surface_obj);
    (void) build_translation();
    color(COLOR(GREEN));
    shadrange(RANGE(GREEN));
    move((Coord) move_point->coord[0], (Coord) move_point->coord[1],
         (Coord) move_point->coord[2]);

    closeobj(marker_obj);

    /* * end_move:
    * Stop moving this point
    */
    end_move()
    {
        extern Object move_obj, point_obj;

        if (move_point == NULL)
            return;
        redraw_move();
        move_point = NULL;
        delobj(move_obj);
        if (isobj(point_obj))
            make_point();
    }

    /* * abort_move:
    * Abort the move and restore original coordinates
    */
    abort_move()
    {
        if (move_point == NULL)
            return;
        move_point->coord[0] = save_coord[0];
        move_point->coord[1] = save_coord[1];
        move_point->coord[2] = save_coord[2];
        end_move();
    }

    /* * set_point_color:
    * Set the color of a point to be picked
    */
    set_point_color(new_color)
    ColorIndex new_color;
    {
        register POINT *pp;

        pp = pick_control();
        pp->color = new_color;
        mark_patch(pp);
    }
}

```



```
move.c  
    update(TRUE);  
}
```

render.c

```

/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 *
 */
#include <gl.h>
#include <device.h>
#include "menu.h"
#include "colors.h"

#define HOURGLASS 3

static Cursor hourglass = {
    0x1FF0, 0x1FF0, 0x0820, 0x0820,
    0x0820, 0x0C60, 0x06C0, 0x0100,
    0x0100, 0x06C0, 0x0C60, 0x0820,
    0x0820, 0x0820, 0x1FF0, 0x1FF0,
};

/* Put the IRIS into single buffer mode and start up zbuffering. Display
 * the object, then wait for a single mouse button push before restoring
 * double buffer mode w/o zbuffering
 */
render(type)
    Int type;
{
    register Int i;
    short old_index;
    Colorindex old_color, old_wtm;
    Boolean old_b;
    Int deltax;
    static Int first = 1;
    extern Object view_obj, transformation_obj, scale_obj;
    extern Matrix ident;

    If (first) {
        defcursor(HOURGLASS, hourglass);
        curorigin(HOURGLASS, 0, 15);
        first = 0;
    }

    cursoff();
    singlebuffer();
    gconfig();
    zbuffer(TRUE);
    zclear();
    color(BLACK);

    /* Initialize the Z-buffer */
    /* and black it out */
}

clear();
pushattributes();
pushviewport();
pushmatrix();

deltax = (XMAXSCREEN - YMAXSCREEN) / 2;
color(COLOR(BLUE));
move2i(deltax, 0);
draw2i(deltax, YMAXSCREEN);
draw2i(YMAXSCREEN + deltax, YMAXSCREEN);
draw2i(YMAXSCREEN + deltax, 0);

/* Setup the transformations
 */
viewport(deltax, YMAXSCREEN + deltax, 0, YMAXSCREEN);
callobj(view_obj);
callobj(transformation_obj);
callobj(scale_obj);
backface(TRUE);

/* Setup the drawing mode
 */
switch (type) {
case T_NONE:
case T_THATCHED:
    break;
case T_HALFTONE:
    setpattern(HALFTONE);
    break;
case T_CROSSHATCH:
    setpattern(CROSSHATCH);
    break;
}

/* Render the surface
 */
update_render(type);

/* Reset the drawing mode
 */
switch (type) {

```

render.c

```
case T_NONE:
case T_THATCHED:
    break;
case T_HALFTONE:
case T_CROSSHATCH:
    setpattern(0);
    break;
}

/* Reset the cursor
*/
getcursor(&old_index, &old_color, &old_wtm, &old_b);
setcursor(HOURGLASS, COLOR(WHITE), old_wtm);
qreset();
cursor();
waitformousebutton();

backface(FALSE);
popmatrix();
popviewport();
popattribues();

zbuffer(FALSE);
doublebuffer();
setcursor(old_index, old_color, old_wtm);
gconfig();
}

/* Wait for a down click of any mouse button
*/
waitformousebutton()
{
    short data;
    for (;;)
        switch (qread(&data)) {
            case LEFTMOUSE:
            case MIDDLEMOUSE:
            case RIGHTMOUSE:
                return;
            default:
                warn("notamouse\n");
                break;
        }
}
```

update.c

```

/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 *
 */
#include <stdio.h>
#include <math.h>
#include "display.h"
#include "colors.h"
#include "menu.h"
#include <device.h>

/* #define DEBUG */

#define FRACTION (1.0/3.0)
#define PRECISION(n) (2 * ((n) + 1))

/* * Precision for rendering surfaces */
#define RP 7
#define PRETTY_PICTURE
#define OUTSIDE_COLOR GREEN
#define INSIDE_COLOR MAGENTA
#define PATCH_COLOR(i,j,k) ((i > j && i > k) ? 2 : ((j > k) ? 1 : 0))

#define DEBUG
#define debug;
#define d_move(x, y, z) fprintf(debug, "m %.3f %.3f %.3f\n", x, y, z)
#define d_draw(x, y, z) fprintf(debug, "d %.3f %.3f %.3f\n", x, y, z)

static double z[3] = { 0, 0, 0 };
static int bijk_num[5][5] = {
    1, 4, 6, 4, 1,
    4, 12, 12, 6, 0,
    6, 12, 6, 0, 0,
    4, 4, 0, 0, 0,
    1, 0, 0, 0, 0,
};

static double light_vector[3] = { 0, 0, 1 };
static int nstripes = 0;
static double **ustripe = NULL;
static double **vstripe = NULL;
static double **wstripe = NULL;

```

```

#endif
static Colorindex IRIS;
static Colorindex *ucolor = NULL;
static Colorindex *vcolor = NULL;
static Colorindex *wcolor = NULL;
#endif

/*
 * init_update:
 * Initialize the update routines
 */
init_update(n)
int n;
{
    register int i, j, size;
    extern char *emalloc();

    if (nstripes > 0) {
        size = PRECISION(nstripes) + 1;
        for (i = 0; i < size; i++) {
            free((char *) ustripe[i]);
            free((char *) vstripe[i]);
            free((char *) wstripe[i]);
        }
        free((char *) ustripe);
        free((char *) vstripe);
        free((char *) wstripe);
        free((char *) ucolor);
        free((char *) vcolor);
        free((char *) wcolor);
    }

    nstripes = n;
    size = PRECISION(n) + 1;
    ustripe = (double **) emalloc(sizeof (double *) * size);
    vstripe = (double **) emalloc(sizeof (double *) * size);
    wstripe = (double **) emalloc(sizeof (double *) * size);
    for (i = 0; i < size; i++) {
        ustripe[i] = (double *) emalloc(sizeof (double) * 3);
        vstripe[i] = (double *) emalloc(sizeof (double) * 3);
        wstripe[i] = (double *) emalloc(sizeof (double) * 3);
    }
    IRIS
    ucolor = (Colorindex *) emalloc(sizeof (Colorindex) * size);
    vcolor = (Colorindex *) emalloc(sizeof (Colorindex) * size);
    wcolor = (Colorindex *) emalloc(sizeof (Colorindex) * size);
}
#endif

```

```

update.c
}

/*
 * update: Update the objects associated with each patch
 */
update(remake)
int remake;
{
    update_normals();
    update_intermediate(remake);
    if (remake) {
        update_cbt();
        update_triangles();
    }
}

/*
 * update_normals: Update the point normals
 */
update_normals()
{
    register int i, j, k;
    register POINT *pp, *endpp;
    double a[3], b[3], normal[3];
    double theta, total;
    extern double dist3(), vv3();

    endpp = &point[npoint];
    for (pp = point; pp < endpp; pp++) {
        if (pp->status & S_NORMAL)
            continue;
        pp->status |= S_NORMAL;

        /*
         * Compute the normal for this point by taking a weighted
         * average of the angles associated with each triangle
         * that uses this point
         */
        total = 0;
        for (k = 0; k < 3; k++)
            pp->normal[k] = 0;
        for (i = 0; i < npoint; i++) {
            j = (i + 1) % npoint;
            for (k = 0; k < 3; k++) {
                a[k] = pp->point[i]->coord[k] - pp->coord[k];
                b[k] = pp->point[j]->coord[k] - pp->coord[k];
            }
            vv3(normal, a, b);
            theta = acos(vv3(a, b) / dist3(a, z) / dist3(b, z));
            for (k = 0; k < 3; k++)
                pp->normal[k] += theta * normal[k];
            total += theta;
        }
        for (k = 0; k < 3; k++)
            pp->normal[k] /= total;
    }

    /*
     * update_intermediate:
     *   - Update the intermediate points on an edge
     *   - Assumes that point normals are up to date
     */
    update_intermediate(remake)
    int remake;
    {
        register EDGE *ep, *endep;

        endep = &edge[nedge];
        for (ep = edge; ep < endep; ep++) {
            if (ep->status & S_INTERMEDIATE)
                continue;
            ep->status |= S_INTERMEDIATE;

            /*
             * Compute the intermediate point by taking some fraction
             * along the edges and projecting that point onto the
             * plane defined by the point normal
             */
            update_ip(ep->point[0], ep->point[1], ep->ipoint[0]);
            update_ip(ep->point[1], ep->point[0], ep->ipoint[1]);
            elevate_edge(ep);
            render_edge(ep, remake);
        }

        /*
         * update_ip:
         *   - Compute a single intermediate point
         *   - This is done by computing a vector "proj" from "from" to "to", which
         *     is some FRACTION of the distance between the two points
         *   - We then compute the distance from ("from" + "proj") to the tangent
         */
    }
}

```

update.c

```

*
* plane by taking the dot product of "proj" and the plane normal
* We can finally compute the intermediate point location by
* computing "from" + "proj" - plane normal of above-mentioned distance
*/
update_ip(from, to, ip)
POINT *from, *to;
double ip[3];
{
    register int i;
    double vdist, idist, ndist;
    double proj[3];
    extern double dist3(), vv3();

    ndist = dist3(from->normal, z);
    vdist = dist3(from->coord, to->coord);
    idist = FRACTION * vdist;
    for (i = 0; i < 3; i++)
        proj[i] = (to->coord[i] - from->coord[i]) / vdist * idist;
    vdist = vv3(proj, from->normal) / ndist;
    for (i = 0; i < 3; i++)
        ip[i] = from->coord[i] + proj[i]
            - from->normal[i] / ndist * vdist;
}

/*
* elevate_edge:
* Elevate the two intermediate points to three for each edge
* Assumes that intermediate points and cbt are up to date
*/
elevate_edge(ep)
register EDGE *ep;
{
    register int i;

/*
* Elevate edge intermediate points to side control points
* Compute and store the control points
* This is done by degree elevation (from cubic to quartic).
*/
    ndx1
    ip1 +-----+ ip2
    / | | |
    ndx0 * | | * ndx2
    / | | |
    / | | |
    * v1 + | | | + v2

```

update.c

```

w2 = w1 * w1;
w3 = w2 * w1;
w4 = w3 * w1;
for (i = 0; i < 3; i++)
    coord[i] = w4 * ep->point[0]->coord[i]
              + 4.0 * v1 * w3 * ep->epoint[0][i]
              + 6.0 * v2 * w2 * ep->epoint[1][i]
              + 4.0 * v3 * w1 * ep->epoint[2][i]
              + v4 * ep->point[1]->coord[i];

register EDGE *ep, *endep;
endep = &edge[nedge];
for (ep = edge; ep < endep; ep++) {
    if (ep->status & S_CBT)
        continue;
    ep->status |= S_CBT;
}
/*
 * Compute the cross boundary derivative by taking
 * the difference between the two end points and normalizing
 */
for (i = 0; i < 3; i++) {
    ep->cbt[0][i] =
        ep->cbt_point[1][0][i] - ep->cbt_point[0][0][i];
    ep->cbt[1][i] =
        ep->cbt_point[1][1][i] - ep->cbt_point[0][1][i];
    normalize(ep->cbt[0]);
    normalize(ep->cbt[1]);
}

}

/* normalize:
 * Normalize a vector
 */
normalize(v)
double v[3];
{
    register int i;
    double dist;
    extern double dist3();
    dist = dist3(v, z);
    for (i = 0; i < 3; i++)
        v[i] /= dist;
}

/* update_triangles:
 * Redraw the triangles
 * Assumes that intermediate points and cbt are up to date
 */
update_triangles()
{
    register TRIANGLE *tp, *endtp;

```

update.c

```

register SIDE *sp, *endsp;
double ctrl[P_SIZE][3];

endtp = &triangle[ntriangle];
for (tp = triangle; tp < endtp; tp++) {
    if (tp->status & S_PATCH)
        continue;
    tp->status |= S_PATCH;
}

/* Make sure the sides are up to date */
for (sp = tp->side; sp < endsp; sp++)
    update_control(tp, sp);
construct_array(tp, ctrl);
render_patch(tp, ctrl);
}

/* update_control:
 * Fill in the Gregory patch control points derived from
 * the given edge. This is the method described by
 * Chiyokura (H. Chiyokura, Localized Surface Interpolation
 * Method for Irregular Meshes, Computer Graphics, 1986).
 *
 * The variables are as follows:
 * p? - cross boundary tangents
 * r? - vectors from edge to triangle[1]
 * t? - vectors from edge to triangle[0]
 * s? - vectors along edge
 */
update_control(tp, sp)
TRIANGLE *tp;
SIDE *sp;
{
    register EDGE *ep;
    register int i;
    double h0, k0, h1, k1;
    double p0[3], p1[3], p2[3], p3[3];
    double t0[3], t3[3];
    double r0[3], r3[3];
    double s0[3], s1[3], s2[3];

    /* Compute all the easily known vectors
    */
    for (i = 0; i < 3; i++) {
        ep = sp->edge;
        if (ep->triangle[0] == tp)
            for (i = 0; i < 3; i++) {
                p0[i] = ep->cbt[0][i];
                p3[i] = ep->cbt[1][i];
                t0[i] = ep->cbt_point[0][i] - ep->point[0]->coord[i];
                t3[i] = ep->cbt_point[1][i] - ep->point[1]->coord[i];
                r0[i] = ep->cbt_point[1][0][i] - ep->point[0]->coord[i];
                r3[i] = ep->cbt_point[1][1][i] - ep->point[1]->coord[i];
                s0[i] = ep->ipoint[0][i] - ep->point[0]->coord[i];
                s1[i] = ep->ipoint[1][i] - ep->ipoint[0][i];
                s2[i] = ep->point[1]->coord[i] - ep->ipoint[1][i];
            }
        else
            for (i = 0; i < 3; i++) {
                p0[i] = -ep->cbt[0][i];
                p3[i] = -ep->cbt[1][i];
                r0[i] = ep->cbt_point[0][0][i] - ep->point[0]->coord[i];
                r3[i] = ep->cbt_point[0][1][i] - ep->point[1]->coord[i];
                t0[i] = ep->cbt_point[1][0][i] - ep->point[0]->coord[i];
                t3[i] = ep->cbt_point[1][1][i] - ep->point[1]->coord[i];
                s0[i] = ep->ipoint[0][i] - ep->point[0]->coord[i];
                s1[i] = ep->ipoint[1][i] - ep->ipoint[0][i];
                s2[i] = ep->point[1]->coord[i] - ep->ipoint[1][i];
            }
    }

    /* Interpolate the cbt's
    */
    for (i = 0; i < 3; i++) {
        p1[i] = p0[i] * 2 / 3 + p3[i] / 3;
        p2[i] = p0[i] / 3 + p3[i] * 2 / 3;
    }

    /* Compute the h and k coefficients
    */
    h0 = (t0[0] * p0[1] - t0[1] * p0[0]) / (s0[0] * p0[1] - p0[0] * s0[1]);
    k0 = (s0[1] * t0[0] - s0[0] * t0[1]) / (p0[0] * s0[1] - p0[1] * s0[0]);
    h1 = (t3[0] * p3[1] - t3[1] * p3[0]) / (s2[0] * p3[1] - p3[0] * s2[1]);
    k1 = (s2[1] * t3[0] - s2[0] * t3[1]) / (p3[0] * s2[1] - p3[1] * s2[0]);

    /* Compute the actual control points
    */
    for (i = 0; i < 3; i++) {

```


update.c

```

    sp->ctrl[0][i] = sp->edge->ipoint[0][i]
        + (k1 - k0) / 3.0 * p0[i] + k0 * p1[i]
        + 2.0 / 3.0 * h0 * s1[i] + h1 / 3.0 * s0[i];
    sp->ctrl[1][i] = sp->edge->ipoint[1][i]
        + k1 * p2[i] - (k1 - k0) / 3.0 * p3[i]
        + h0 / 3.0 * s2[i] + 2.0 / 3.0 * h1 * s1[i];
}

/*
 * construct_array:
 *   Create the Gregory patch control point array
 */
construct_array(tp, ctrl)
    TRIANGLE *tp;
    double ctrl[P_SIZE][3];
{
    register int i;
    register SIDE *sp;

    for (i = 0; i < 3; i++) {
        ctrl[P_P004][i] = tp->point[0]->coord[i];
        ctrl[P_P040][i] = tp->point[1]->coord[i];
        ctrl[P_P400][i] = tp->point[2]->coord[i];
    }

    sp = &tp->side[0];
    if (sp->edge->point[0] == tp->point[0])
        for (i = 0; i < 3; i++) {
            ctrl[P_P013][i] = sp->edge->epoint[0][i];
            ctrl[P_P022][i] = sp->edge->epoint[1][i];
            ctrl[P_P031][i] = sp->edge->epoint[2][i];
            ctrl[P_P121U][i] = sp->ctrl[0][i];
            ctrl[P_P121U][i] = sp->ctrl[1][i];
        }
    else
        for (i = 0; i < 3; i++) {
            ctrl[P_P031][i] = sp->edge->epoint[0][i];
            ctrl[P_P022][i] = sp->edge->epoint[1][i];
            ctrl[P_P013][i] = sp->edge->epoint[2][i];
            ctrl[P_P121U][i] = sp->ctrl[0][i];
            ctrl[P_P112V][i] = sp->ctrl[1][i];
        }

    sp = &tp->side[1];
    if (sp->edge->point[0] == tp->point[1])
        for (i = 0; i < 3; i++) {
            ctrl[P_P130][i] = sp->edge->epoint[0][i];
            ctrl[P_P220][i] = sp->edge->epoint[1][i];
            ctrl[P_P310][i] = sp->edge->epoint[2][i];
            ctrl[P_P121W][i] = sp->ctrl[0][i];
            ctrl[P_P211W][i] = sp->ctrl[1][i];
        }
    else
        for (i = 0; i < 3; i++) {
            ctrl[P_P310][i] = sp->edge->epoint[0][i];
            ctrl[P_P220][i] = sp->edge->epoint[1][i];
            ctrl[P_P130][i] = sp->edge->epoint[2][i];
            ctrl[P_P211W][i] = sp->ctrl[0][i];
            ctrl[P_P121W][i] = sp->ctrl[1][i];
        }

    sp = &tp->side[2];
    if (sp->edge->point[0] == tp->point[2])
        for (i = 0; i < 3; i++) {
            ctrl[P_P301][i] = sp->edge->epoint[0][i];
            ctrl[P_P202][i] = sp->edge->epoint[1][i];
            ctrl[P_P103][i] = sp->edge->epoint[2][i];
            ctrl[P_P112V][i] = sp->ctrl[0][i];
            ctrl[P_P211V][i] = sp->ctrl[1][i];
        }
    else
        for (i = 0; i < 3; i++) {
            ctrl[P_P103][i] = sp->edge->epoint[0][i];
            ctrl[P_P202][i] = sp->edge->epoint[1][i];
            ctrl[P_P301][i] = sp->edge->epoint[2][i];
            ctrl[P_P211V][i] = sp->ctrl[0][i];
            ctrl[P_P112V][i] = sp->ctrl[1][i];
        }
}

/*
 * render_patch:
 *   Render the interior of a triangular patch
 */
render_patch(tp, ctrl)
    TRIANGLE *tp;
    double ctrl[P_SIZE][3];
{
    register int i, j;
    register int precision;
    register int u, v, w;
    double *c[5];
    double p112[3], p121[3], p211[3];
}

```

```

update.c
#fdef IRIS
double lu, lv, lw;
double mu, mv, mw;
Colorindex compute_color();
#endf

init_ctrl(ctrl, q, p112, p121, p211);

IRIS
makeobj(tp->object);
lu = 1 / (double) (nstrips + 1);
for (i = 1; i <= nstrips; i++) {
    u = i / (double) (nstrips + 1);
    precision = PRECISION(nstrips + 1 - i);
    mu = (lu + u) / 2;
    lv = 0;
    lw = 1.0 - lu - lv;
    for (j = 0; j <= precision; j++) {
        v = (j / (double) precision) * (1.0 - u);
        w = 1.0 - u - v;
        compute_point(q, ctrl, u, v, w, ustripe[j]);
        compute_point(q, ctrl, v, u, w, vstripe[j]);
        compute_point(q, ctrl, w, v, u, wstripe[j]);
        mv = (lv + v) / 2;
        mw = (lw + w) / 2;
        ucolor[j] = compute_color(mu, mv, mw, tp);
        vcolor[j] = compute_color(mv, mu, mw, tp);
        wcolor[j] = compute_color(mw, mu, mv, tp);
        lv = v;
        lw = w;
    }
    draw_stripe(ustripe, ucolor, precision);
    draw_stripe(vstripe, vcolor, precision);
    draw_stripe(wstripe, wcolor, precision);
    lu = u;
#fdef IRIS
}
IRIS
closeobj();
#endf
}

/*
 * init_patch:
 * Initialize the path parameters
 */
init_ctrl(ctrl, q, p112, p121, p211)
double ctrl[P_SIZE][3];
double *q[5][5];
double *p112, *p121, *p211;
{
    q[0][0] = ctrl[P_P004];
    q[0][1] = ctrl[P_P013];
    q[0][2] = ctrl[P_P022];
    q[0][3] = ctrl[P_P031];
    q[0][4] = ctrl[P_P040];
    q[1][0] = ctrl[P_P103];
    q[1][3] = ctrl[P_P130];
    q[2][0] = ctrl[P_P202];
    q[2][2] = ctrl[P_P220];
    q[3][0] = ctrl[P_P301];
    q[3][1] = ctrl[P_P310];
    q[4][0] = ctrl[P_P400];
    q[1][1] = p112;
    q[1][2] = p121;
    q[2][1] = p211;
}

/*
 * compute_point:
 * Compute a point on a Gregory patch when given the
 * control points and the barycentric coordinates
 */
compute_point(q, ctrl, u, v, w, coord)
double *q[5][5];
double ctrl[P_SIZE][3];
double u, v, w;
double coord[3];
{
    register int i, j, k;
    double bijk;
    double usup[5], vsup[5], wsup[5];

/*
 * Compute the effective interior control points
 */
    if (u != 0 && v != 0 && w != 0)
        for (i = 0; i < 3; i++) {

```

update.c

```

q[1][1][0] = ((1-u)*v*ctr[P_P112U][0]
+u*(1-v)*ctr[P_P112V][0])
/((1-u)*v+u*(1-v));
q[1][2][0] = ((1-u)*w*ctr[P_P121U][0]
+u*(1-w)*ctr[P_P121W][0])
/((1-u)*w+u*(1-w));
q[2][1][0] = ((1-v)*w*ctr[P_P211V][0]
+v*(1-w)*ctr[P_P211W][0])
/((1-v)*w+v*(1-w));
}
else
for (i = 0; i < 3; i++) {
q[1][1][i] = 0;
q[1][2][i] = 0;
q[2][1][i] = 0;
}
}
/*
* Now actually compute the point itself
* The constants are hardwired for quantic triangles
*/
coord[0] = coord[1] = coord[2] = 0;
usup[0] = vsup[0] = wsup[0] = 1;
for (i = 1; i < 5; i++) {
usup[i] = usup[i-1]*u;
vsup[i] = vsup[i-1]*v;
wsup[i] = wsup[i-1]*w;
}
for (i = 0; i < 5; i++)
for (j = 0; j < (5-i); j++) {
k = 4-i-j;
bijk = bijk_num[i][j]*usup[i]*vsup[j]*wsup[k];
coord[0] += bijk*q[1][0];
coord[1] += bijk*q[1][1];
coord[2] += bijk*q[1][2];
}
}
#endif IRIS
/*
* compute_color:
* Compute the color of a point by its barycentric coordinate
*/
ColorIndex
compute_color(u, v, w, tp)
double
u, v, w;
*tp;
TRIANGLE
{
if (u > v && u > w)
return tp->point[2]->color;
else if (v > w)
return tp->point[1]->color;
else
return tp->point[0]->color;
}
/*
* draw_stripe:
* Draw the stripe in BILD format
*/
draw_stripe(stripe, cindex, stripe_size)
register double
**stripe;
register ColorIndex
*cindex;
int
stripe_size;
{
register int
i;
register int
cur_color;
#endif
DEBUG
d_move(stripe[0][0], stripe[0][1], stripe[0][2]);
for (i = 1; i <= stripe_size; i++)
d_draw(stripe[i][0], stripe[i][1], stripe[i][2]);
IRIS
cur_color = -1;
move((Coord) stripe[0][0], (Coord) stripe[0][1], (Coord) stripe[0][2]);
for (i = 1; i <= stripe_size; i++) {
if (cindex[i] != cur_color) {
cur_color = cindex[i];
color(COLOR(cur_color));
shadrange(RANGE(cur_color));
}
draw((Coord) stripe[i][0], (Coord) stripe[i][1],
(Coord) stripe[i][2]);
}
#endif
}
/*
* update_render:
* Render the object with Gouraud shading
*/
update_render(type)
int
type;

```

update.c

```
{
    register TRIANGLE
    register int
    double *tp, *endtp;
    double i, j;
    double ctrl[P_SIZE][3];
    double build_translation();
    double light[3];
    Matrix m;
    extern Matrix ident;
    extern Object transformation_obj;

    /* * Get the transformation matrix so we can compute intensity */
    /* */
    pushmatrix();
    loadmatrix(ident);
    callobj(transformation_obj);
    getmatrix(m);
    popmatrix();
    for (i = 0; i < 3; i++) {
        light[i] = 0;
        /*
        * This multiplication is correct. We are multiplying
        * by the transpose, which just happens to be the
        * inverse as well
        */
        for (j = 0; j < 3; j++)
            light[i] += light_vector[j] * m[i][j];
        }
    normalize(light);

    (void) build_translation();
    endtp = &triangle[ntriangle];
    for (tp = triangle; tp < endtp; tp++) {
        construct_array(tp, ctrl);
        render_triangle(tp, ctrl, type, light);
    }
}

/* * render_triangle:
 *   Render the particular triangle using Gouraud shading
 */
TRIANGLE *tp;
double ctrl[P_SIZE][3];
int type;
double light[3];
    register int
    double i, j, k;
    coord[RP][RP][3];
    colors[RP][RP][2], cur_color;
    normal[3];
    unormal[3], vnormal[3], wnormal[3];
    u, v, w;
    *q[5][5];
    p112[3], p121[3], p211[3];
    intensity;
    PRETTY_PICTURE
    register int si, sj, sk;
    Colorindex new_color;

    #ifndef
    #endif

    #endif

    init_ctrl(ctrl, q, p112, p121, p211);

    /* * Compute the normalized normals
    */
    for (i = 0; i < 3; i++) {
        wnormal[i] = tp->point[0]->normal[i];
        vnormal[i] = tp->point[1]->normal[i];
        unormal[i] = tp->point[2]->normal[i];
    }

    /* * Now compute the coordinates and intensity at each
    * u, v, w
    */
    for (i = 0; i < RP; i++) {
        u = i / (double) (RP - 1);
        for (j = 0; j < (RP - 1); j++) {
            v = j / (double) (RP - 1);
            w = 1 - u - v;
            intensity = 0;
            for (k = 0; k < 3; k++)
                normal[k] = u * unormal[k] + v * vnormal[k]
                    + w * wnormal[k];
            normalize(normal);
            for (k = 0; k < 3; k++)
                intensity += light[k] * normal[k];
            k = (intensity < 0) ? 1 : 0;
            intensity = intensity * intensity;
            colors[i][j][k] = (1 - intensity) * (num_shades - 1);
            colors[i][j][1 - k] = (num_shades - 1);
            compute_point(q, ctrl, u, v, w, coord[i][j]);
        }
    }
}
```

update.c

```
    }
}

/*
 * Now actually create the polygons
 */
#ifdef PRETTY_PICTURE
cur_color = OUTSIDE_COLOR;
for (i = 0; i < RP - 1; i++) {
    PRETTY_PICTURE
        si = (i) + (i + 1);
#ifdef PRETTY_PICTURE
        for (j = 0; j < (RP - 1 - i); j++) {
            PRETTY_PICTURE
                sj = (j) + (j + 1);
                sk = 3 * RP - si - sj;
                cur_color = tp->point[PATCH_COLOR(si, sj, sk)]->color;

                setshade(COLOR(cur_color) - colors[i][j][0]);
                pmv((Coord) coord[i][j][0],
                    (Coord) coord[i][j][1],
                    (Coord) coord[i][j][2]);
                setshade(COLOR(cur_color) - colors[i + 1][j][0]);
                pdr((Coord) coord[i + 1][j][0],
                    (Coord) coord[i + 1][j][1],
                    (Coord) coord[i + 1][j][2]);
                setshade(COLOR(cur_color) - colors[i][j + 1][0]);
                pdr((Coord) coord[i][j + 1][0],
                    (Coord) coord[i][j + 1][1],
                    (Coord) coord[i][j + 1][2]);
                setshade(COLOR(cur_color) - colors[i][j + 1][1]);
                pdr((Coord) coord[i][j + 1][1],
                    (Coord) coord[i][j + 1][2]);
                setshade(COLOR(cur_color) - colors[i + 1][j + 1][0]);
                pdr((Coord) coord[i + 1][j + 1][0],
                    (Coord) coord[i + 1][j + 1][1],
                    (Coord) coord[i + 1][j + 1][2]);
                spclos();
            }
        }
}
/*
 * Now the other half of the polygons
 */
if (type == T_THATCHED)
    return;
PRETTY_PICTURE
cur_color = OUTSIDE_COLOR;
for (i = 1; i < RP - 1; i++) {
    PRETTY_PICTURE
        si = (i) + (i - 1) + (i);
#ifdef PRETTY_PICTURE
        for (j = 0; j < (RP - 1 - i); j++) {
            sj = (j) + (j + 1) + (j + 1);
            sk = 3 * RP - si - sj;
            cur_color = tp->point[PATCH_COLOR(si, sj, sk)]->color;

            setshade(COLOR(cur_color) - colors[i][j][0]);
            pmv((Coord) coord[i][j][0],
                (Coord) coord[i][j][1],
                (Coord) coord[i][j][2]);
            setshade(COLOR(cur_color) - colors[i][j + 1][0]);
            pdr((Coord) coord[i][j + 1][0],
                (Coord) coord[i][j + 1][1],
                (Coord) coord[i][j + 1][2]);
            setshade(COLOR(cur_color) - colors[i + 1][j][0]);
            pdr((Coord) coord[i + 1][j][0],
                (Coord) coord[i + 1][j][1],
                (Coord) coord[i + 1][j][2]);
            spclos();
        }
    }
}
PRETTY_PICTURE
cur_color = INSIDE_COLOR;
for (i = 0; i < RP - 1; i++) {
    PRETTY_PICTURE
        si = (i) + (i + 1) + (i);
#ifdef PRETTY_PICTURE
        for (j = 0; j < (RP - 1 - i); j++) {
            sj = (j) + (j) + (j + 1);
            sk = 3 * RP - si - sj;
            cur_color = tp->point[PATCH_COLOR(si, sj, sk)]->color;

            setshade(COLOR(cur_color) - colors[i][j][0]);
            pmv((Coord) coord[i][j][0],
                (Coord) coord[i][j][1],
                (Coord) coord[i][j][2]);
            setshade(COLOR(cur_color) - colors[i + 1][j][0]);
            pdr((Coord) coord[i + 1][j][0],
                (Coord) coord[i + 1][j][1],
                (Coord) coord[i + 1][j][2]);
            setshade(COLOR(cur_color) - colors[i][j + 1][0]);
            pdr((Coord) coord[i][j + 1][0],
                (Coord) coord[i][j + 1][1],
                (Coord) coord[i][j + 1][2]);
            setshade(COLOR(cur_color) - colors[i + 1][j + 1][0]);
            pdr((Coord) coord[i + 1][j + 1][0],
                (Coord) coord[i + 1][j + 1][1],
                (Coord) coord[i + 1][j + 1][2]);
            spclos();
        }
    }
}
#endif
#endif
}
```

update.c

```

    }
} PRETTY_PICTURE
cur_color = INSIDE_COLOR;
#endif
#endif
for (i = 1; i < RP - 1; i++) {
    PRETTY_PICTURE
    si = (i) + (i) + (i - 1);
    for (j = 0; j < (RP - 1 - i); j++) {
        sj = (j) + (j + 1) + (j + 1);
        sk = 3 * RP - si - sj;
        cur_color = tp->point[PATCH_COLOR(si, sj, sk)]->color;
        setshade(COLOR(cur_color) - colors[i][j][1]);
        pmv((Coord) coord[i][j][0],
            (Coord) coord[i][j][1],
            (Coord) coord[i][j][2]);
        setshade(COLOR(cur_color) - colors[i][j] + 1[1]);
        pdr((Coord) coord[i][j] + 1[0],
            (Coord) coord[i][j] + 1[1],
            (Coord) coord[i][j] + 1[2]);
        setshade(COLOR(cur_color) - colors[i - 1][j] + 1[1]);
        pdr((Coord) coord[i - 1][j] + 1[0],
            (Coord) coord[i - 1][j] + 1[1],
            (Coord) coord[i - 1][j] + 1[2]);
        spclos();
    }
}
}
/*
 * build_translation:
 * Construct translation and scaling commands to make
 * surface appear in the center of the screen
 */
double
build_translation()
{
    register Int
    register POINT
    double
    static double
    static Int
    If (first)
        i;
        *pp, *endpp;
        min[3], max[3];
        coord[3], sc, range;
        first = 1;
}
goto done;
for (i = 0; i < 3; i++) {
    min[i] = point[0].coord[i];
    max[i] = point[0].coord[i];
}
endpp = &point[0].coord[i];
for (pp = &point[1]; pp < endpp; pp++) {
    for (i = 0; i < 3; i++) {
        if (pp->coord[i] < min[i])
            min[i] = pp->coord[i];
        else if (pp->coord[i] > max[i])
            max[i] = pp->coord[i];
    }
}
sc = max[0] - min[0];
if (max[1] - min[1] > sc)
    sc = max[1] - min[1];
sc = 140.0 / sc;
for (i = 0; i < 3; i++)
    coord[i] = (min[i] + max[i]) / 2;
range = (max[0] - min[0]) / 100;
IRIS
scale((Coord) sc, (Coord) sc, (Coord) sc);
translate((Coord) -coord[0], (Coord) -coord[1], (Coord) -coord[2]);
DEBUG
if (debug != NULL) {
    fprintf(debug, ".scale %.3f %.3f %.3f\n", sc, sc, sc);
    fprintf(debug, ".tran %.3f %.3f %.3f\n", -coord[0], -coord[1],
        -coord[2]);
}
}
first = 0;
return range;
}
/*
 * make_global:
 * Create the global data structure
 */
make_global()
{
    #ifdef IRIS
    register POINT
        *pp, *endpp;
    register EDGE
        *ep, *endep;
    #endif
}

```

update.c

```

register TRIANGLE      *tp, *endtp;
double                size;
extern Object         bild_obj, marker_obj, point_obj;
extern Object         move_obj, move_surface_obj, move_axis_obj;

#ifdef DEBUG
if ((debug = fopen("display.update", "w")) == NULL) {
    perror("display.update");
    exit(1);
}
#endif
#ifdef IRIS
bild_obj = genobj();
makeobj(bild_obj);
pushmatrix();
#endif

/*
 * Some mundane setup code
 */
#ifdef DEBUG
fprintf("color %2n", debug);
size = build_translation();
#endif
#ifdef IRIS
/*
 * Make all points red for now
 */
endpp = &point[npoint];
for (pp = point; pp < endpp; pp++)
    pp->color = RED;
#endif

/*
 * Create unique object labels for edges
 */
endep = &edge[nedge];
for (ep = edge; ep < endep; ep++) {
    ep->object = genobj();
}
#ifdef IRIS
    callobj(ep->object);
#endif

/*
 * Create unique object labels for triangles
 */
register TRIANGLE      *tp, *endtp;
double                size;
extern Object         bild_obj, marker_obj, point_obj;
extern Object         move_obj, move_surface_obj, move_axis_obj;

#ifdef DEBUG
if ((debug = fopen("display.update", "w")) == NULL) {
    perror("display.update");
    exit(1);
}
#endif
#ifdef IRIS
bild_obj = genobj();
makeobj(bild_obj);
pushmatrix();
#endif

/*
 * Make the marker object
 */
#ifdef IRIS
marker_obj = genobj();
makeobj(marker_obj);
rmv(size, (Coord) 0, (Coord) 0);
rdr((Coord)-size * 2, (Coord) 0, (Coord) 0);
rmv(size, size, (Coord) 0);
rdr((Coord) 0, (Coord)-size * 2, (Coord) 0);
rmv((Coord) 0, size, size);
rdr((Coord) 0, (Coord) 0, (Coord)-size * 2);
closeobj();
point_obj = genobj();
move_obj = genobj();
move_surface_obj = genobj();
move_axis_obj = genobj();
#endif

/*
 * Make sure that the objects have been created
 */
update(TRUE);
#ifdef DEBUG
(void) fclose(debug);
debug = NULL;
#endif
#ifdef IRIS
/*
 * make_point:
 */

```

update.c

```

*      Creates the point object
*/
make_point()
{
    register POINT *pp, *endpp;
    extern Object point_obj;

    makeobj(point_obj);
    pushmatrix();
    (void) build_translation();
    color(COLOR(CYAN));
    shaderrange(RANGE(CYAN));
    endpp = &point[npoint];
    for (pp = point; pp < endpp; pp++) {
        loadname((short) (pp - point));
        move((Coord) pp->coord[0], (Coord) pp->coord[1],
            (Coord) pp->coord[2]);
        callobj(marker_obj);
    }
    popmatrix();
    closeobj();
}

#define LIGHT_RADIUS 50
#define MOVE_RADIUS 75
#define SOURCE_RADIUS 1

/* position_light:
 *      Position the light source
 */
position_light(background)
Object background;
{
    register int x, y;
    register int dx, dy, rsq;
    register int ox, oy;
    register int sign;
    double r;
    short data;

    ox = getvaluator(MOUSEX);
    oy = getvaluator(MOUSEY);

    sign = (light_vector[2] < 0) ? -1 : 1;
    draw_light(background, ox, oy, sign);
    for (;;) {
        x = getvaluator(MOUSEX);
        y = getvaluator(MOUSEY);
        dx = x - ox;
        dy = y - oy;
        rsq = dx * dx + dy * dy;
        if (rsq > MOVE_RADIUS * MOVE_RADIUS)
            return;
        if (rsq > LIGHT_RADIUS * LIGHT_RADIUS) {
            r = sqrt((double) rsq);
            dx = dx / r * LIGHT_RADIUS;
            dy = dy / r * LIGHT_RADIUS;
        }
        if (qtest())
            switch (qtest(&data)) {
                case LEFTMOUSE:
                    if (!(Boolean) data)
                        break;
                    light_vector[0] = dx / (double) LIGHT_RADIUS;
                    light_vector[1] = dy / (double) LIGHT_RADIUS;
                    light_vector[2] = sign * sqrt(1.0
                        - light_vector[0] * light_vector[0]
                        - light_vector[1] * light_vector[1]);
                    draw_light(background, ox, oy, sign);
                    break;
                case RIGHTMOUSE:
                    if (!(Boolean) data)
                        break;
                    sign = -sign;
                    light_vector[2] = -light_vector[2];
                    draw_light(background, ox, oy, sign);
                    break;
                default:
                    return;
            }
    }
}

/* draw_light:
 *      Draw the light position object over the global object
 */
static
draw_light(background, x, y, sign)
Object background;
int x, y;
int sign;

```


update.c

```
{
    callobj(background);
    pushmatrix();
    translate((Coord) x, (Coord) y, (Coord) 0);
    color(COLOR(BLACK));
    circf((Coord) 0, (Coord) 0, (Coord) LIGHT_RADIUS);
    color(COLOR(BLUE));
    circ((Coord) 0, (Coord) 0, (Coord) LIGHT_RADIUS);
    move((Coord) -LIGHT_RADIUS, (Coord) 0, (Coord) 0);
    draw((Coord) LIGHT_RADIUS, (Coord) 0, (Coord) 0);
    move((Coord) 0, (Coord) -LIGHT_RADIUS, (Coord) 0);
    draw((Coord) 0, (Coord) LIGHT_RADIUS, (Coord) 0);
    color(COLOR(MAGENTA));
    circf((Coord) (light_vector[0] * LIGHT_RADIUS),
          (Coord) (light_vector[1] * LIGHT_RADIUS),
          (Coord) SOURCE_RADIUS);
    if (sign < 0) {
        cmov((Coord) -10, (Coord) (-LIGHT_RADIUS - 10),
            (Coord) 0);
        charstr("Backlit");
    }
    popmatrix();
    swapbuffers();
} #endif
```

viewat.c

```

/*
 * Copyright (c) 1987 by the Regents of the University of California
 * All rights reserved.
 */
/*
 * This routine is "borrowed" from Paul Bash. The math involved in
 * putting an arbitrary vector onto the z axis may be found in
 * Newman & Sproul, "Principles of Interactive Computer Graphics".
 */
#include <math.h>
#include <gl.h>
viewat(M, invM, P1, P2, P3)
Matrix M, invM;
double P1[3], P2[3], P3[3]; /* lookfrom, lookat, lookup */
{
    double d12;
    double P120, P121, P122, P130, P131, P132;
    void nmcrosprod();

    P120 = P2[0] - P1[0];
    P121 = P2[1] - P1[1];
    P122 = P2[2] - P1[2];
    P130 = P3[0] - P1[0];
    P131 = P3[1] - P1[1];
    P132 = P3[2] - P1[2];
    d12 = sqrt(P120*P120 + P121*P121 + P122*P122);
    invM[2][0] = M[0][2] = P120/d12;
    invM[2][1] = M[1][2] = P121/d12;
    invM[2][2] = M[2][2] = P122/d12;

    nmcrosprod(P130, P131, P132, P120, P121, P122,
               &M[0][0], &M[1][0], &M[2][0]);
    invM[0][0] = M[0][0];
    invM[0][1] = M[1][0];
    invM[0][2] = M[2][0];

    nmcrosprod(M[0][2], M[1][2], M[2][2], M[0][0], M[1][0], M[2][0],
               &M[0][1], &M[1][1], &M[2][1]);
    invM[1][0] = M[0][1];
    invM[1][1] = M[1][1];
    invM[1][2] = M[2][1];

    invM[3][0] = P1[0];
    invM[3][1] = P1[1];

```

```

    invM[3][2] = P1[2];
    invM[3][3] = 1.0;

    M[3][0] = -P1[0]*M[0][0] - P1[1]*M[1][0] - P1[2]*M[2][0];
    M[3][1] = -P1[0]*M[0][1] - P1[1]*M[1][1] - P1[2]*M[2][1];
    M[3][2] = -P1[0]*M[0][2] - P1[1]*M[1][2] - P1[2]*M[2][2];
    M[3][3] = 1.0;

    M[0][3] = M[1][3] = M[2][3] = 0.0;
    invM[0][3] = invM[1][3] = invM[2][3] = 0.0;
}

static void
nmcrosprod(x1, y1, z1, x2, y2, z2, x3, y3, z3)
double x1, y1, z1, x2, y2, z2; /* r1 cross r2 */
double *x3, *y3, *z3; /* Normalized crossproduct */
{
    double dis; /* length of crossproduct vector r1 x r2 */
    double x, y, z;
    x = y1*z2 - y2*z1;
    y = z1*x2 - z2*x1;
    z = x1*y2 - x2*y1;
    dis = sqrt(x*x + y*y + z*z);
    *x3 = x / dis;
    *y3 = y / dis;
    *z3 = z / dis;
}

mult_vector(r, v, m)
double r[3], v[3];
Matrix m;
{
    r[0] = v[0]*m[0][0] + v[1]*m[1][0] + v[2]*m[2][0] + m[3][0];
    r[1] = v[0]*m[0][1] + v[1]*m[1][1] + v[2]*m[2][1] + m[3][1];
    r[2] = v[0]*m[0][2] + v[1]*m[1][2] + v[2]*m[2][2] + m[3][2];
}

```

Appendix Three: Publications Resulting From Ph.D. Research

Publications Resulting From Ph.D. Research

C. Hansch, J. McClarin, T. Klein and R. Langridge,
“A Quantitative Structure-Activity
Relationship and Molecular Graphics Study
of Carbonic Anhydrase Inhibitors,”
Molec. Pharm. **27**, 498-498, 1985.

T.E. Klein, D. Kneller, C. Huang, T.E. Ferrin, and R. Langridge,
“Computer Graphics and Artificial Intelligence in Drug Design and
Protein Engineering,”
J. Molec. Graph. **3**, 111-113, 1985.

M. Recanatini, T. Klein, C.Z. Yang, J. McClarin, R. Langridge, and C. Hansch,
“Quantitative Structure-Activity Relationships and Molecular Graphics in
Ligand Receptor Interactions: Amidine Inhibition of Trypsin,”
Molec. Pharm. **29**, 436-446, 1986.

T.E. Klein, C. Huang, T.E. Ferrin, R. Langridge and C. Hansch,
“Computer-Assisted Drug Receptor Mapping Analysis,” in
Artificial Intelligence in Chemistry. T.H. Pierce and B.A. Hohne, eds.
ACS Symposium Series 306, 147-158, 1986.

C. Hansch, T. Klein, J. McClarin, R. Langridge and N. Cornell,
“A Quantitative Structure-Activity Relationship and
Molecular Graphics Analysis of
Hydrophobic Effects in the Interactions of Inhibitors
of Alcohol Dehydrogenase,”
J. Med. Chem. **29**, 615-620, 1986.

C.D. Selassie, Z-X Fang, R-l Li, C. Hansch, T. Klein,
R. Langridge and B.T. Kaufman, “Inhibition of
Chicken Liver Dihydrofolate Reductase
by 5-(Substituted benzyl)-2,4-diaminopyrimidines. A Quantitative
Structure-Activity Relationship and Graphics Analysis,”
J. Med. Chem. **29**, 621-626, 1986.

C. Hansch and T. Klein, “Molecular Graphics and QSAR in the Study of
Enzyme-Ligand Interactions. On the Definition of Bioreceptors,”
Accts. Chem. Res. **19**, 392-400, 1986.

L. Morgenstern, M. Recanatini, T. Klein, W. Steinmertz, C.Z. Yang, R. Langridge and C. Hansch, "Chymotrypsin Hydrolysis of X-Phenyl Hippurates. A QSAR and Molecular Graphics Analysis," *J. Bio. Chem.* (in press).

E.M. Price, P.L. Smith, T.E. Klein, J.H. Freisheim, "Photoaffinity Analogues of Methotrexate as Folate Antagonist Binding Probes. I. Photoaffinity Labeling of Murine L1210 Dihydrofolate Reductase and Amino Acid Sequence of the Binding Region," *Biochemistry* (in press).

... the last brick in the wall ...



FOR REFERENCE

NOT TO BE TAKEN FROM THE ROOM



CAT. NO. 23 012

PRINTED
IN
U.S.A.



