C.1

Constraint Satisfaction for Interactive 3-D Model Acquisition

by

Heather M. Cameron

A Thesis Submitted in Partial Fulfillment of
the Requirements for the Degree of
Master of Science

in

The Faculty of Graduate Studies
Department of Computer Science

We accept this thesis as conforming
to the required standard

University of British Columbia
October 1990

51

Department of COMPUTER SCIENCE

The University of British Columbia
Vancouver, Canada

Date 12 OCTOBER 1990

Abstract

More and more computer applications are using three-dimensional models for a variety of uses (e.g. CAD, graphics, recognition). A major bottleneck is the acquisition of these models. The easiest method for designing the models is to build them directly from images of the object being modelled. This paper describes the design of a system, MOLASYS (for MOdeL Acquisition SYStem), that allows the user to build object models interactively from underlying images. This would not only be easier for the user, but also more accurate as the models will be built directly satisfying the dimensions, shape, and other constraints present in the images.

The object models are constructed by constraining model points and edges to match points in the image objects. The constraints are defined by the user and expressed using a Jacobian matrix of partial derivatives of the errors with respect to a set of camera and model parameters. MOLASYS then uses Newton's method to solve for corrections to the parameters that will reduce the errors specified in the constraints to zero. Thus the user describes how the system will change, and the program determines the best way to accomplish the desired changes.

The above techniques, implemented in MOLASYS, have resulted in an intuitive and flexible tool for the interactive creation of three-dimensional models.

Contents

List of Figures

Acknowledgements

I would like to thank my supervisor, David Lowe, for his advice, insight, and knowledge. The uncritical encouragement was much appreciated over the long months. My thanks also go to my second reader, Jim Little, for his valuable comments and suggestions.

I would also like to thank my mother and brother without whose support this would never have happened, and our librarian, Deb Wilson, for tracking down those seemingly impossible to find references and for providing much needed sanity breaks. Thanks also to my friends for making my stay in Vancouver so enjoyable, and in particular to Arlene for constantly nagging me into finishing so I could have more of those good times.

This research was supported by a scholarship from the National Science and Engineering Research Council (NSERC).

Chapter 1

Introduction

It was the best of times,

it was the worst of times

- Charles Dickens

1.1 Purpose

Object modelling systems in recent years have been progressing rapidly, both in terms of the capabilities they possess and the expanding number of purposes to which they can be applied. A variety of solid modelling techniques have been developed that are capable of representing a wide array of shapes and scenes with increasing speed and efficiency. The systems are also allowing more user interaction, rendering them easier to use and giving the user more control over the properties of the model. However, a current problem in the area of object modelling is the creation, or acquisition, of the three-dimensional models. Building the models "by hand" (i.e., plotting coordinates for vertices and edges) is both time-consuming and tedious; it is much simpler to construct the models interactively from images. The models could be overlaid on multiple images from different viewpoints, allowing the user to design the models directly from the objects in the images. This would not only be easier for the user, but also more accurate as the models would be built directly satisfying the dimensions, shape, and other constraints present in the images. Also, instead of using actual images (i.e., digitized photographs), only the image edges could be displayed. This would result in the models reflecting the actual features detectable in the images, an important consideration for computer vision applications.

MOLASYS (for MOdeL Acquisition SYStem) has been designed in an attempt to satisfy the above goals. The approach involves the combining of graphics techniques to computer vision, with the advantages of increased ease-of-use, flexibility, and accuracy in the creation of the models, in addition to reductions in the time required for model-building. MOLASYS is primarily intended for computer vision applications as they frequently use three-dimensional models for matching and recognition tasks. However it could also be a valuable tool for other fields such as CAD and computer graphics, providing a method for easily constructing models from images. CAD systems are designed to facilitate the creation of models, usually drawn to preset specifications. Allowing the user to create the model directly from an image of the desired shape would provide an alternative approach in the design process resulting in a more robust system.

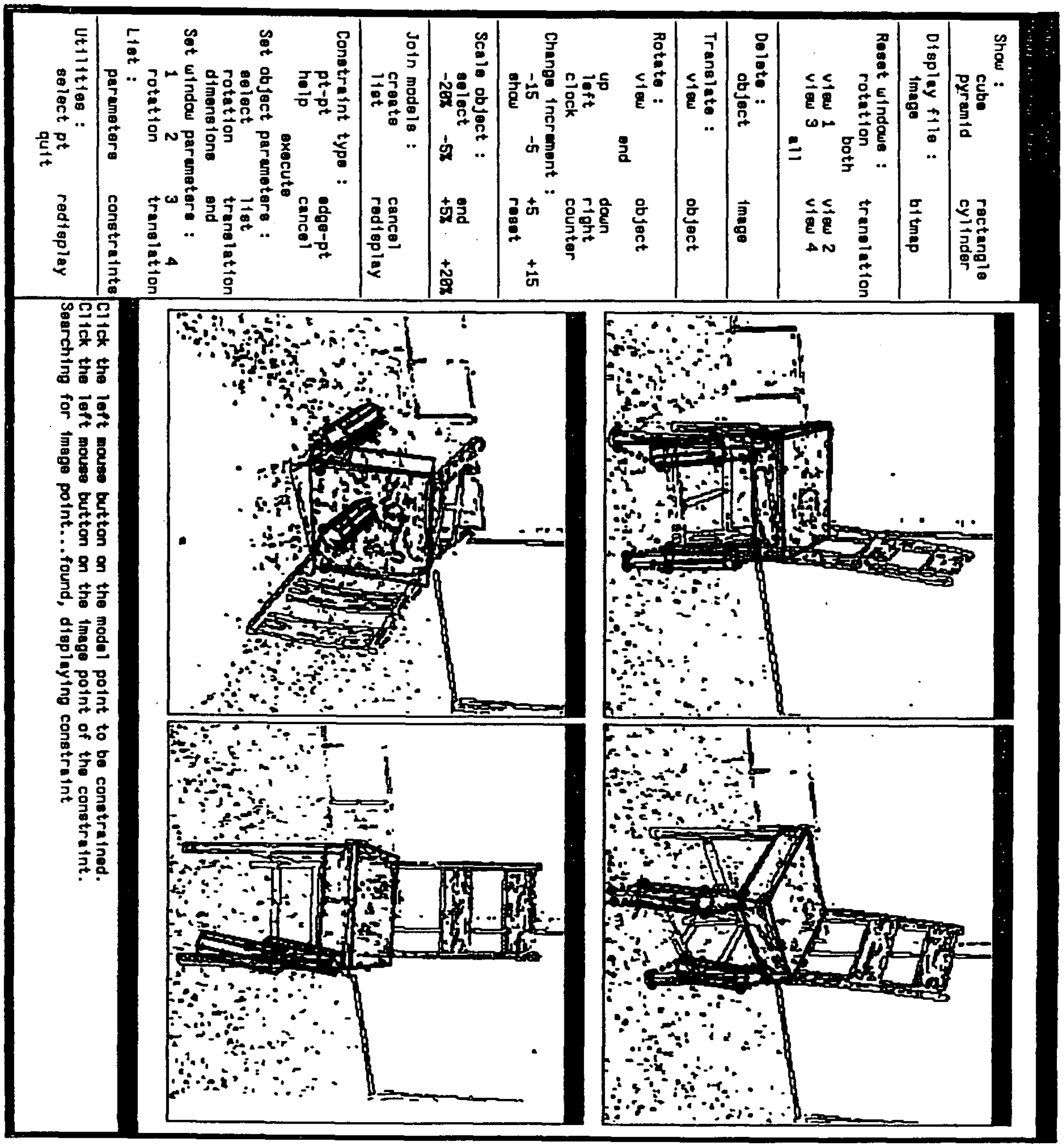
1.2 System Overview

MOLASYS is an interactive three-dimensional modelling system. The user is provided with a set of 3-D primitives (cubes, rectangular solids, pyramids and cylinders) which can be manipulated (translated, rotated, and scaled) through mouse and menu control. The primitives can also be joined together to create more complex objects. The primitives are displayed in four windows representing different views of the models. The unique aspect of this system is that the object models are built directly from images displayed in the background. Photographs of an object/scene are taken from any four different viewpoints, digitized, passed through an edge-detector, and shown in the background of each of the four views. The models are then constructed on top of the images, using constraints to alter the shapes of the primitives to correspond with the shapes of the image objects being modelled. Individual facilities for translation, rotation, and scaling are also provided for initial placement of the models and minor adjustments. Figure 1.1 is an example of the MOLASYS interface, showing the menu, the background images from multiple viewpoints, and an intermediate step of the model construction in which two cylinder primitives are being constrained to model a chair's legs. The constraints for the rectangular primitive modelling the chair seat in figure 1.1 were specified and solved for in a previous step.

Constraints, specified interactively by the user, allow vertices and edges in the models to be fixed to certain points in the image. The user specifies how the models should be changed and it is left to the program to decide how best to alter the parameters to reflect those changes. A constraint will be of two possible forms: *point-to-point*, where the user specifies a point in the model and a point in the background image to which the model should be moved, or *edge-to-point*, in which a model edge is moved a perpendicular distance to an image point. Many of these constraints can be defined (in any or all of the windows) and simultaneously evaluated, solving for a combination of camera parameters (translation and rotation of the scene) and object parameters (rotation and translation with respect to the scene, length, width, height) specified by the user. Altering the camera rotation and/or translation (about any of x, y, and z) affects all the models in that window, while changing the object rotation and/or translation affects only that object in all views with respect to the rest of the object models.

One of the key features of MOLASYS is the constraint-satisfaction process. The constraints are implemented as a system of linear equations, $\mathbf{Jx} = \mathbf{e}$, where $\mathbf{e}$ is a vector of error measurements specified by the constraints (i.e., the distance a vertex or edge is to be moved), $\mathbf{x}$ represents the corrections to the parameters be solved for in order to eliminate the errors (i.e., satisfy the constraints), and $\mathbf{J}$ is the Jacobian matrix of partial derivatives of the errors with respect to the parameters (i.e., $J_{ij} = \partial e_i / \partial x_j$). As some of the image transformations involved are not linear (e.g. rotation), iteration frequently will be required for convergence to a solution for $\mathbf{x}$. Newton's method will be used for this iteration as follows: $\mathbf{p}^{(i+1)} = \mathbf{p}^{(i)} - \mathbf{x}$ where $\mathbf{p}^{(i)}$ is the vector of parameter estimates for iteration i, and $\mathbf{x}$ is the vector of corrections discussed above. The iteration continues until the error measurements $\mathbf{e}$ between components of the image and the model have been reduced to within a predetermined threshold, or until the errors are no longer being reduced (i.e., the system has converged to a "best-fit" solution).

The primitives are defined by a boundary representation using polygonal approximations for the surfaces. The models are described in a relatively simple format useful for many applications, i.e., the description is higher-level, consisting of vertices, edges, poly-



Click the left mouse button on the model point to be constrained.
Click the left mouse button on the image point of the constraint.
Searching for image point...found, displaying constraint

Figure 1.1: The MOLASYS interface and model construction

gons, variables, etc. Promising results have thus far been obtained using this approach. It is hoped that this work will yield an effective tool for the design of 3-D models, using the constraints in the images to guide and simplify the design process.

1.3 Organization of the Thesis

The thesis is divided up into five chapters. Chapter two presents a review of work previously conducted in this and related areas. Object modelling is discussed first. As MOLASYS is presenting a new method for creating models, it is useful to first examine existing methods of model acquisition. Included in this section are brief explanations of current modelling techniques to provide a basis for later discussions of modelling systems. The next section of chapter two examines several constraint-based modelling systems, studying the approaches used to represent and solve systems of constraints and the applications for which they were intended. Also in this chapter is a discussion of interface design issues, briefly presenting factors to be considered in the design of an interface from a human-computer interaction point of view.

Chapter three describes the result of this work, MOLASYS. This discussion includes an explanation of the approach and design issues, the modelling representation, and the main algorithms and their implementation.

Chapter four provides several examples of MOLASYS in action, as well as an analysis of its performance. Chapter five then concludes the thesis with a discussion of any problems found using this approach, and any extensions and directions for further work discovered during the design and implementation phases of MOLASYS.

Chapter 2

Previous Work

2.1 Object Modelling

Object modelling, or solid modelling, is the creation and manipulation of representations of objects. A model represents the structure and behaviour of an object and allows well-defined properties of that object to be calculated automatically, allowing for much greater flexibility. Many solid modelling techniques exist (e.g., boundary representation, constructive solid geometry, sweep representation) for a large array of possible applications (CAD/CAM, recognition, graphical display, scientific visualization, games, etc.). An important consideration in modelling is to design/implement the representations to best suit the properties of the objects and the requirements of the system. The next several subsections present a variety of these modelling techniques, discussing examples of systems in which they have been implemented and examining their applicability to interactive modelling, especially to the process of creating models from images.

2.1.1 Boundary Representation

One of the earliest approaches to solid modelling was the wireframe representation. The first systems were interactive and two-dimensional, using simple lists of lines and arcs to represent an object's edges. These lines and arcs were generalized to three-dimensional space curve segments, with projections used to create perspective views. While these systems have been useful, they have serious limitations. For example, one wireframe structure can represent several different objects as only the edges are modelled and other information (e.g., concerning faces) is lost.

To model more complex objects, boundary/surface methods (B-rep) can be used. A more advanced version of the earlier wireframe representations, B-rep algorithms model objects by their enclosing surfaces (i.e., as a set of faces or patches, with each face as a set of edges plus surface data). The faces can be represented by either polygonal approximations or parametric polynomial patches (Bézier and B-spline forms), both of which will be defined in the next two subsections.

Polygon Mesh

A polygon mesh is a set of connected planar polygon surfaces. Any three-dimensional object can be represented by polygons although the precision of this representation varies with the object. Polyhedrons (e.g., buildings, desks, cabinets) can be modelled exactly using polygon meshes, whereas for curved surfaces the polygonal approximations may not be nearly as good. The approximation can be improved by dividing the polygon surface into smaller polygon faces although this has the corresponding increase in space and execution costs.

A polygon mesh representation consists of vertices, edges and polygons. The edges are sets of connected vertices and the polygons can be viewed as sets of connected vertices or edges. This results in rapid calculation and display of polygonal surfaces that are easily used and modified, and hence also results in good performance for interactive systems requiring fast response time. Polygonal approximations are the most commonly used method.

Parametric Surfaces

Parametric (usually bicubic) patches give a much more precise description of curved surfaces, resulting in less "jagged" visual displays. The coordinates of points on the surfaces are represented with parametric equations (in u and v) for x, y and z. This representation is more complex than that of polygon meshes, but fewer bicubic than polygonal patches are required to represent a surface to a given accuracy. An additional advantage is that parametric surfaces are independent of the coordinate system.

A parametric curve equation is one in which x, y, and z are represented as polynomials of a parameter u. For an arbitrary curve, it may or may not be possible to find a single set of parametric functions $x(u), y(u)$, and $z(u)$ that completely defines the shape of the curve. However, by using different sets of parametric equations for different pieces of the curve, any curve can be represented. In piecing together this representation, the sets of equations are derived carefully to ensure smooth transitions, or continuity, between adjacent segments of the curve. If the curves meet, they are said to be zero-order continuous. If the tangent lines of the two adjoining curve sections are the same at the joining point, then the representation possesses first-order continuity. Finally, second-order continuity means that curvatures (second derivatives) of the two curve sections are the same at the intersection. Cubic curves are commonly used to approximate curves as no lower order representation can provide continuity of position and slope while ensuring that the ends of the curve segment pass through specified points. Higher order curves could be used but these frequently have undesirable oscillations.

Parametric surface equations are expressed with two parameters, u and v, and a coordinate on a surface is represented by $P(u,v) = (x(u,v), y(u,v), z(u,v))$. Varying both parameters u and v between 0 and 1 defines all points on a surface patch. Varying only one parameter with the other constant describes a curve. A curve or surface can be defined by a set of control points which indicate the shape of the curve. There are several methods for constructing curves from these control points and the next few paragraphs will contain brief descriptions of two of the more commonly used methods: Bézier and B-spline. For details on other parametric representations such as Hermite and Coons surfaces, see [Bar84] or [Fol82].

An approximating *Bézier curve* is calculated for a set of control points by adding a sequence of polynomial functions (blending functions) formed from the coordinates of the control points. The Bézier function for $n+1$ control points is expressed as $P(u) = \sum_{k=0}^{n} p_k B_{k,n}(u)$ where p_k specifies the locations of the control points. $B_{k,n}$ are the blending functions, and blend the control points to form a composite function of degree n describing the curve. The form of blending functions determines how the control points influence the shape of the curve. $B_{0,3} = 1$ at $u = 0$ and $B_{2,3} = 1$ at $u = 1$ are the only non-zero blending functions, ensuring that the Bézier curve will pass through the endpoints p_0 and p_3. $B_{1,3}$

and $B_{2,3}$ are maximum at 1/3 and 2/3 respectively, influencing the curve towards p_1 and p_2 for intermediate values of u. Any Bézier curve lies within its convex hull (polygon boundary) of the control points, quaranteeing a curve that smoothly follows the control points without oscillations. As Bézier curves have the property that the tangent to the curve at an endpoint is along the line joining that endpoint to its adjacent control point, first-order continuity is achieved by selecting control points so that the last two control points of one curve section are along the same straight line as the first two control points of the next curve section. However, Bézier curves do not usually exhibit second-order continuity. For more details, see [Hea86], [Fol82].

To represent *Bézier surfaces*, two sets of Bézier curves are used. The resulting function for $(m+1)$ by $(n+1)$ control points is $P(u,v) = \sum_{j=0}^{m} \sum_{k=0}^{n} p_{j,k} B_{j,m}(u) B_{k,n}(v)$. To assure a smooth transition between adjacent surface patches, the four control points on the edges of the patches should be equal. First-order continuity is obtained when the control points across the edges are collinear and there is a constant ratio of the lengths of these collinear line segments. Bézier surfaces have the same properties as Bézier curves and are useful in interactive design as the control points can be easily manipulated to change the shape of the surface.

Spline curves are piecewise approximations of cubic polynomial functions that possess zero, first, and second-order continuity. *B-splines* are a class of splines useful for graphics and have the following equation: $P(u) = \sum_{k=0}^{n} p_k N_{k,t}(u)$, where $N_{k,t}$ are blending functions defined recursively over various subintervals of the parameter u. The positions u_j defining these subintervals are referred to as *breakpoints*, and *knots* are the points on the B-spline curve corresponding to the breakpoints. The blending functions are each zero over part of the range of u, i.e., each function and its corresponding control point only influence the shape of part of the curve. Thus localized changes are made easily, without greatly affecting the shape of the rest of the curve. Any number of control points can be specified for B-spline curves without increasing the degree of the curve, allowing one cubic curve to represent many different curve shapes without being required to piece together smaller curve segments. Modifying the shape of the curve is accomplished easily by adding any number of control points. B-splines lie within the convex hull of the control points, and as mentioned above possess second-order continuity resulting in "smoother" curves than for the Bézier method. B-spline sufaces are formed similarly to Bézier surfaces, using a Cartesian product of its blending functions over two parameters, u and v.

As an example, Ostby [Ost86] used bicubic patches in creating a system for the interactive manipulation of free-form surfaces. Free-form surfaces include human and animal forms, some landforms, and many hand-made objects. These surfaces are not easy to model using current methods as they can not be readily described mathematically or "grown" through the use of fractal techniques. Bicubic patches however, can approximate any continuous surface accurately, while allowing relative ease of manipulation and having reasonable storage requirements. The patches are flexible and can be molded into a wide variety of shapes. Ostby has achieved initial success in implementing a set of tools for creating and modifying bicubic patches. The surfaces of an object are represented as grids of bicubic patches, and the interactive tools implemented for manipulating the surfaces include the dragging of control points, manipulating (rotating, translating, scaling, and skewing) of groups of control patches, subdividing patches to increase the resolution, and the reorienting of individual tangents through rotation, translation, and scaling. The Polhemus (a 3-D sensor) was used to allow for easy input of the three-dimensional data for the surface models being created.

Thingvold and Cohen's [Thi90] work also focussed on developing a representation for elastic surfaces to be used for modelling and animation. Their implementation used B-splines combined with refinement techniques to model objects. The user starts the modelling process with a completely defined model possessing relatively low resolution and adds more information to the model as it becomes necessary. The refinement process combines the new information with the existing model by searching for another representation of the same model which has more degrees of freedom and greater flexibility. That is, new values are added to the knot vectors resulting in additional rows and/or columns being added to the set of coefficients for the blending functions. The rows/columns correspond to the location(s) of the inserted knot(s). In this way, the B-spline curve is successfully refined and more closely approximates the surface. The above method is a more continuous representation than many of the more discretized models and supports interactive modelling and animation but only of elastic surfaces.

While these representations show good flexibility and efficiency, they are limited in the range of shapes that can be modelled, i.e., only curved edges can be adequately represented. As the domain of MOLASYS will include polyhedral objects, this is a serious drawback for our purposes.

Other examples of boundary representations include Boyse and Gilchrist [Boy82] who created **GMSolid**, one of the earlier interactive 3-D graphics systems, and Bhanu and Ho [Bha87]. In [Bha87], the object models are first created using a spline-based CAD front-end, while the second step then develops representations (e.g., surface curvature, generalized sweep, extended Gaussian image) from the models to construct features needed for 3-D object recognition. This is similar in a way to MOLASYS, as MOLASYS combines a simple boundary representation with constraints to achieve the desired modelling capabilities.

2.1.2 Constructive Solid Geometry

Another major approach in solid modelling is constructive solid geometry, or CSG, which defines an object by the space it occupies. Primitives (blocks, cylinders, spheres) are manipulated (translated, rotated, scaled) and combined (union, intersection, difference), defining the operations in terms of the primitives instead of using the actual point data, resulting in more easily constructed models. In CSG, an object is represented as a binary tree with each leaf as a primitive and each node as an operation to be applied to its leaves. While CSG systems are unambiguous and easy to use, the disadvantages are that their modelling power is limited (i.e., difficulties in modelling freeform shapes) and that they are computationally expensive as the surfaces must be recomputed for each display.

A simple example of a CSG system is provided by Kunii [Kun85], in which the nodes are set operations performed on the primitives and the tree is represented as a directed acyclic graph to avoid the problem offf the leaves representing different copies of the same primitives. The system is effective but simple as it was designed to test new ideas such as using an object-oriented approach in describing the primitives as data plus procedures.

an octree structure for rendering.

As computing the surfaces from a CSG-tree is costly, creating a secondary structure for display purposes is used in a number of CSG systems. One such system is GMSolid by Boyse and Gilchrist [Boy82] introduced in the previous section. In GMSolid, two representations of each solid are generated: a boundary representation and a constructive representation. The boundary representation contains the points, edges, faces and their connections, and is generated by procedures for the primitives or by a boundary evaluator for combined primitives. The constructive representation is a binary tree generated when two primitives are combined. The tree then contains the primitives and the set operation combining them.

Combining two (or more) representations can be quite useful as no one representation can efficiently store all the information needed for model creation and display for objects of varying shapes. CSG is compact and can be generated quickly when the primitives are combined. B-rep contains surface data valuable for calculating intersections of objects and for display purposes. Mirolo and Pagello [Mir89] use both of these representations in **World Modeler**, a solid modeller for complex polyhedrons. CSG, using generalized cylinders as primitives, is combined with a polygonal boundary representation to create geometric models consisting of volumes and the surfaces bounding the volumes. Two internal representations are maintained: the polygons are defined by their vertices and edges, and the objects are represented as a graph of generalized cylinders. Laidlaw and Hughes [Lai86] also combine CSG and polygonal B-rep by performing the CSG operations directly on polygons. The objects are defined as surfaces bounding a volume; the surfaces are approximated by polygons and the operations applied directly to these boundaries.

While good results have been achieved with these dual representations, they are more appropriate for the industrial applications for which they were intended as they are more complicated than is necessary for MOLASYS. With this added complexity some flexibility is lost, i.e., the flexibility to use constraints to aid the modelling process. With a dual representation, it would be much more difficult to extend both representations to include the constraints and constraint-satisfaction process.

2.1.3 Sweep Representation

Sweep representation uses generalized sweeps, or cylinders - the volume swept out by moving an area (cross-section) along an axis (trajectory). Transformation rules specify the changes in the cross-section as it moves along the axis. In other words, the set of points of the object model is represented as the cartesian product of an area and a trajectory. The sweep representations are concise, easy to use and unambiguous. However, objects that can be modelled with this technique are limited to those possessing rotational or translational symmetry.

Siska et al. [Sis88] use a sweep representation in **ADROS**, a 3D interactive solid modelling system developed for the design of industrial parts. In ADROS, the user draws in 2D a cross-section of an object and a plot along which the shape runs. The user can then select the interaction between the cross-section and the path to be either parallel, radial or normal. For parallel paths, the cross-section is copied and moved parallel to itself along the path. The radial approach creates models that revolve by moving the cross-section points angularly along the path. The normal interaction specifies that the cross-section is placed normal to the path. It is also possible to draw two cross-sections and one path, or two paths and one cross-section to allow for the creation of a variety of irregular shapes. The primitives can be combined (added, subtraction, intersection) to construct more complex solids. While ADROS works well for its intended task, modelling systems in general require a greater variety of shapes than can be produced by the sweep representation in ADROS. To provide a more varied domain, Klok [Klo86] swept cross-sections along 3D trajectories represented by parametric polynomials. It is also possible to replace the cross-section with a sphere of varying radius, or a closed 2D contour represented by piecewise polynomials (for further details, see [Klo86]). In addition, the sweep representation can be combined with CSG ([Bro81], [Mir89]) to result in systems where the primitives are generalized cylinders connected in CSG-like trees, as discussed in the previous section. Once again though, the increased complexity of this modelling approach limits its potential usefulness for the purposes of this paper as it will be difficult to modify the models interactively and through the use of constraints.

2.1.4 The Gaussian Sphere

Another type of solid modelling involves the use of Gaussian spheres. In these algorithms the normal vectors of the surface points of an object are translated to a common origin, retaining their original directions. The vectors then define a Gaussian sphere, and the locus of the endpoints is the Gaussian image of the object. These images can be simplified for some objects, e.g., polyhedrons need only one normal vector for each face. One of the advantages to this representation is that the Gaussian sphere is independent of the object's position in representing the object; only the object's orientation is used. Also, the mapping is invariant with respect to rotation. However, the degree of freedom of this representation is reduced from six to three as the position data isn't used. Unfortunately, this means that much surface information (such as position, area, curvature) is lost. Two objects of the same shape but differing sizes will have the same Gaussian image. A nonconvex object will usually produce a Gaussian image equivalent to that for some convex object.

To remedy these problems, more information can be added to this representation to derive an extended Gaussian image (EGI). To create an EGI, the points of the Gaussian image are weighted by a weighting function to become point masses. The weighting function can be, for example, surface area or the inverse of Gaussian curvature for smooth objects. However, while this produces a unique image for convex objects, for concave objects different faces can have the same orientation and will be mapped onto the same location on the Gaussian sphere resulting in an ambiguous representation. Bhanu and Ho [Bha87] have resolved this problem in their implementation using EGIs by decomposing each concave object's surface and building an EGI for each patch. This is not an ideal solution however as subdividing the object to yield a unique EGI for each part is not a simple process. Roach et al. [Roa87] have also devloped a CAD system using Gaussian spheres. To avoid the ambiguity problem, a *dual spherical representation* is used that combines the Gaussian image with *dual space*, a representation which contains both the orientation and position of the planes in an object. The users construct three-dimensional models using CSG primitives and boolean operators, with the primitives being represented internally by polyhedral surface descriptions generated by the dual spherical representation. For a more detailed explanation, see [Roa87].

2.1.5 Summary

Above is a discussion of the more common modelling techniques, each possessing certain advantages and disadvantages. No one method can as yet adequately represent a variety of shapes and still provide efficiency in terms of storage requirements and response-time for interactive modification and display. CSG can represent a wide variety of shapes with good accuracy but has slower response-time as the surface information isn't stored with the model. Combining CSG with other representations (e.g., B-rep) solves this problem but also adds to the complexity of the representation. Sweep and Gaussian representations tend to be limited in terms of the shapes that can be modelled and are also more difficult to modify interactively. The best approach for our application is B-rep as it is relatively simple and allows for easy interactive editing of the models. Currently, MOLASYS uses polyhedral approximations of curved surfaces with adequate success; extending the representation to model elastic surfaces as well could provide a more robust modelling capability.

2.2 Constraint-based Systems

A growing trend in interactive modelling systems is the use of constraints. A constraint expresses a relation that must hold true for a given model, or models. The relation can be defined as a mathematical quantity and from many such relations, sets of equations can be formulated to represent the constraints for a particular modelling configuration. The methods for specifying and defining these relations, in addition to solving the resulting system of equations, are varied and include object-oriented approaches (e.g., ThingLab [Bor81]), numerical constraint-satisfaction systems such as Juno ([Nel85]), symbolic constraint-satisfaction systems as implemented by Brüderlin [Brü86], and systems using a sequential evaluation of constraints (e.g., Rossignac [Ros86] and Bier [Bie86a], [Bie90]). While the main goal of most of these systems is the design of 2D or 3D models, the applications of the models varies from CAD and industrial design purposes, to animation ([Thi90], [Barr88], [Pla88]) and articulated figure positioning ([Phi90]), to representing deformable and free-form shapes, to developing a constraint language for drawing pictures. An example of the latter is IDEAL by Van Wyk [Van82], in which the user specifies relationships among variables the system then solves for to produce 2D graphs and geometric drawings.

In the following subsections the above approaches will be discussed in more detail. To start this section however, **Sketchpad** by Sutherland [Sut63] should be introduced as it was the first graphical system to use constraints and hence provided a basis for much of the later work. Well ahead of its time, Sketchpad was an interactive two-dimensional system for drawing shapes. The user employed a light pen to sketch lines and arcs, and could move and join the resulting objects. The constraints represented such relations as the connectivity of the objects, and the parallelism, congruence and orientation (e.g., vertical, horizontal) of the lines. Sutherland's constraint-solver then satisfied these constraints using a relaxation method. Many later projects extended this innovative work, such as **ThingLab** discussed below.

2.2.1 Object-Oriented

One of the earlier object-oriented systems was ThingLab by Borning [Bor81], a constraint-based kit for building such things as geometric demonstrations and simulations of physics experiments. ThingLab was a two-dimensional interactive graphics system that employed constraints to specify the relations among the objects and parts, aiming to allow the user to create objects graphically, i.e. without programming. The user specifies what relations are to hold and the system decides how to maintain those relations.

ThingLab was written in Smalltalk in which objects send and receive messages from each other. The objects in ThingLab can be points, lines, quadrilaterals, etc. and are organized into classes. Each class has a specific internal storage structure, a dictionary of messages it can receive, and methods (procedures) for computing responses. Each instance (object) of a class holds specific values for that object.

ThingLab extends Smalltalk by adding constraints and a constraint satisfaction process. Constraint satisfaction is done in two stages: planning, in which a constraint-satisfaction plan is developed and compiled; and run time, at which point the compiled code is executed for the object being altered. In planning, an object receives a message plan - a description of a message that might be sent to the object later. The object then generates a plan to be used at run time if this message is received. To generate this plan, the object creates an instance of ConstraintSatisfier which then collects the constraints that would be affected by the change and plans a method for satisfying them. The constraint satisfier first attempts to find a one-pass ordering for satisfying the constraints using one of two techniques: propagation of degrees of freedom and propagation of known states. If neither of these can satisfy the constraints, relaxation is used.

Propagation of degrees of freedom proceeds by searching for a part with sufficient degrees of freedom so that it can be altered to satisfy all of its constraints. As it is difficult to define degrees of freedom for nonnumeric objects, the part is considered to have sufficient degrees of freedom if there is only one constraint that affects it. If such a part is found, that part and its constraint are removed from further processing. This may enable another part to acquire sufficient degrees of freedom to satisfy its constraint, and so on. The process continues until either all constraints have been satisfied or until no more degrees of freedom can be propagated.

Propagation of known states is similar to the above technique. In this method the constraint satisfier searches for parts that have no degrees of freedom. If such a part is found, the constraint satisfier then looks for constraints that will allow other parts to be completely known. These constraints must connect a known part (i.e., a part with no degrees of freedom) to another part in one step and must also determine the other part's state uniquely. This process is then repeated.

When the system can not find a one-pass ordering for solving the constraints, the numerical method of relaxation is attempted (a method very close to that used in MOLASYS). In relaxation, each of the object's numerical values is changed one at a time and the effects of each change on the constraints is recorded as a set of linear equations. The coefficients of the linear equations are calculated by finding the derivatives of the error expressions of the constraints with respect to the initial errors. A least-mean-squares fit to this set of equations is found and the values for the parts are recalculated. This process repeats until all the constraints are satisfied (within some predetermined cutoff), or until the system decides that the constraints can not be satisfied (i.e., the errors are no longer decreasing for each iteration). To prevent more parts from being relaxed than is necessary, the following approach is used. It is assumed that the state of the part with the largest number of constraints connecting it to other unknown parts is known. By propagating known states, any parts that would become known as a result of the above step, along with the original part, are removed from the set of parts to be relaxed. This process is repeated until the set of parts to be relaxed is empty. Then, at run time, only the parts that had been assumed to be known are relaxed. As each of these parts is relaxed, the system calculates the new states of the parts which had become known as a result of assuming that that part was known. Relaxation is not that effective a technique in ThingLab but most sets of constraints that arise can be satisfied with one of the other two methods.

The constraint-satisfaction process in ThingLab must be rapid as the system must ensure the constraints are satisfied, for example, each time a new image of a moving figure is shown. The code generated in the planning stage is run efficiently in the execution stage resulting in a good response time. However, the planning process can take an appreciable amount of time, especially if new constraints are created. The planning is only done once for a particular message but planning time will increase for significant numbers of new messages. This problem is more noticeable when structural changes are made frequently, a problem encountered in MOLASYS also. Borning and Duisberg [Bor86] are exploring alternate strategies for constraint satisfaction for these cases.

One of the strengths of ThingLab is its uncomplicated approach for describing the relations between the objects and parts. A variety of relations can be described as constraints, and the constraints are maintained in an efficient manner. Examples of possible constraints are constraining a line to be horizontal, a triangle to be twice the size of another, a resistor to obey Ohm's law, a grey-scale level of a region to be in a certain range, etc. Another advantage is the modularity derived from the use of object-oriented programming techniques. The shared substructure of an object is important for constraint-satisfaction as it facilitates determining what is affected by each change instead of requiring this to be recalculated each time the constraints are solved, as in MOLASYS.

ThingLab provided an example of an interactive graphics system implemented from an object-oriented approach in an efficient and useful manner. As it is only two-dimensional, the practical applications are limited, but it demonstrated the potential for constraint-based programming and gave direction for many later systems.

2.2.2 Numerical Constraint-Satisfaction

Juno, by Nelson [Nel85], is also a two-dimensional graphics system. However, Juno uses iterative numerical methods to solve sets of constraints on point coordinates. Juno is relatively simple as it displays only 2-D line drawings and has only four types of constraints: congruence of distances between pairs of points, parallelism of lines, and specifying a given line to be horizontal or vertical. Combining the first two of these yields a great variety of other geometric constraints. Juno also allows initial estimates of the coordinates where the solution will be found. As Juno permits non-linear constraints, these initial estimates guide the constraint solver in converging to a solution. The nonlinear solver behaves predictably if the initial positions are not too far from the final solution. Juno uses Newton-Raphson iteration to solve the constraints and Nelson has reported acceptable performance.

The object representation is quite simple: the only data object is the point, consisting of two coordinates. The system uses only line drawings and the lines are represented simply as pairs of points. The constraints are specified interactively by selecting icons and points. The points can be moved, added, or deleted, and the constraints can also be added or removed.

Juno is relatively limited but the goal was only to create an image editor for drawing figures. Nelson wanted to be able to create images interactively using a mouse (and avoiding the difficulties of keyboard commands such as being required to keep track of point names instead of simply pointing) and with the ability to specify constraints thus letting the constraint solver perform the often tedious steps for aligning points accurately.

An earlier application for constraints is that developed by Ambler and Popplestone ([Amb75]) who were constructing a robot to be used for automatic assembly and proposed using constraints to instruct the robot. The constraints specified the spatial relations (e.g., connectivity) between the parts of the robot being manipulated in the assembly process. A set of simultaneous equations was derived from these intermediate goal states which was then (partially) solved using propagation of known states. Using this method, Ambler and Popplestone were able to calculate expressions for the positions of three-dimensional bodies from spatial constraints.

A growing area of applications of constraint-based modelling is that of animation. More systems are using some form of physical constraints to create "realistic" motion (e.g., [Thi90]), with a variety of methods for implementation of these constraints. Barzel and Barr [Bar88] have created a modelling system to build and animate graphics models. The models are built from a collection of primitives (e.g., spheres, rods) using constraints to form the objects. The constraints are also employed to position and animate object models. Given the constraints on the behaviour of an object, the system solves an inverse dynamics problem to determine the forces that would produce the desired (constrained) behaviour. Thus the constraints are converted into constraint forces. As the models move, the constraint forces are continuously calculated to maintain the constraints. The motions of the objects are due to the effects of inertia and externally applied forces (e.g., gravity, springs) and also to the geometric constraints. The constraint forces assemble and hold together the objects, representing forces which could be used to assemble real-world objects. Some examples of these constraints are *point-to-point*, in which a joint is formed between two bodies which may continue to move freely on the condition that the two constrained points remain in contact; and *point-to-path* constraints, in which a point on an object is constrained to follow a user-specified path.

Each constraint is described by a measure of its deviation, i.e., deviation is zero when the constraint has been satisfied. The constraint forces are computed by solving a *constraint-force equation* - a multidimensional linear equation of the form $MF + B = 0$ where F is a collection of constraint forces. A standard linear-system solver is used to solve the equation, the result being the net forces that act on each of the objects. The system deals with underconstrained equations by selecting the solution which is smallest in magnitude, and for overconstrained data the least-squares fit usually yields a reasonable solution.

On a similar idea, Platt and Barr [Pla88] have developed a system that applies mathematical constraints to the physical behaviour of object models to create realistic animation of flexible models. Two types of constraints are used in this system: reaction constraints

(RCs), and augmented Lagrangian constraints (ALCs). Reaction constraints supply reaction forces to cancel forces that violate constraints. RCs are fast and can be used to force a point to follow a path, and to prevent collisions. In animated flexible models the constraints will often be more complicated, that is, we want to be able to constrain the models to be incompressible and moldable. For these cases, ALCs have been implemented. ALCs add differential equations that compute Lagrange multipliers of the physical system. Both types of constraints eventually satisfy the specified constraints exactly. In adding constraints to flexible models, Platt and Barr can control the direction and location of movement (without being required to specify each small step) of the objects while maintaining physically realistic motion.

Badler et al [Bad86] developed **POSIT** to simplify the process of positioning and orienting three-dimensional models. The models in this study are articulated figures, i.e., human forms. POSIT uses multiple constraints and inverse kinematics to position and orient the various joint parts of a model into a user-specified goal position. This goal position is broken down into many smaller goals for the different points of the figure, creating a system of simultaneous constraints. POSIT then attempts to solve for the "best-fit" to these goals, using a feature not a part of any of the previously discussed systems. In POSIT each of the goals can be assigned a *strength* value - a measure of the importance of that goal. When it is not possible for each goal to be satisfied, the strength values are used to decide which points must be closer to their goals. For example, if one goal is four times the value of a second goal, the first point will be positioned four times closer to its goal than the second point is to its goal. In POSIT, a body is represented as a hierarchical tree. It can be defined recursively with the lower torso as the root of the tree. The constraints are then satisfied simultaneously using a tree-traversal algorithm.

POSIT is used in connection with a six degree-of-freedom sensor (to be discussed in a later section) and an Iris for fast display. The resulting combination allows for more rapid and intuitive positioning of models, avoiding many of the tedious steps of manually specifying various angles, orientations, etc.

In a later paper, Phillips, Zhao and Badler [Phi90], have created a more advanced system. **Jack** is an interactive interface for modelling articulated figures; it provides for interactive manipulation of the figures and joints using mouse and keyboard input. As in POSIT, the bodies are segments connected by joints being represented in a tree-structure. In Jack, the constraint-satisfaction process has been enhanced from that in POSIT. In this process, each goal/constraint is expressed as a function which describes both the position and orientation of the end effector and is a function of the joint angles. The goal is achieved when a set of joint angles has been found which place the end effector at the goal. A characteristic vector function is then formed for each function such that the corresponding goal is satisfied if and only if the charateristic function applied to the end effector yields the zero vector. To solve the resulting system of characteristic equations, the Newton-Raphson method is used with some modifications to ensure convergence. For more details, see [Phi90]. Also in Jack, a *drag* function has been implemented to allow the user to drag the goal positions and have the end effector follow. The above inverse kinematics algorithm is executed at every refresh of the screen during interactive manipulation. To ensure a minimal delay between the motion of the mouse and the response (i.e., motion of the model), a limit is set on the amount of time for articulated figures, i.e., human forms. POSIT uses multiple constraints and the inverse kinematics algorithm. Since the algorithm is iterative, at each iteration the end effectors move closer to the goal. If the solution can't be computed sufficiently quickly, an intermediate solution is then accepted. The net result is that a good response rate, with less "dead-time", is achieved although the end effectors move more slowly towards the goals.

In the above paragraphs, several systems implementing numerical constraint-satisfaction have been discussed. Many of these applications use a linear-system solver such as the Newton-Raphson method, and report good results. This appears to be the best approach to the problem (and is what is implemented in MOLASYS) as large systems of simultaneous constraints can be solved both accurately and efficiently. However, alternate approaches (e.g., Badler) provide interesting options, such as a weighting function indicating the relative importance of the constraints, that are not currently possible in the other systems.

2.2.3 Symbolic Constraint-Satisfaction

Brüderlin [Brü86] has developed a constraint-based system with similar goals to those of MOLASYS, although the constraint-satisfaction algorithm is quite different. The aim of Brüderlin's project was to simplify the process of building object models by allowing the user to specify constraints on the objects and then having the system do the rest of the work. Brüderlin's system allows the user to interactively create three-dimensional geometric objects through the use of constraints. The constraints are first evaluated symbolically in Prolog and then the results of this step are converted into coordinates by procedures written in Modula-2.

The system initially provides a set of primitives, such as cubes, prisms, and pyramids, and a set of mouse-controlled operations (e.g., scale, rotate, translate) that can be performed on these primitives. Accurate feedback is given to the user by the display of both visual and numeric results of the transformations. Set operations defined by boolean expressions are also provided, allowing objects to be added or subtracted from each other. Once the initial 3-D shape of the model has been defined using the above tools, geometric constraints can be used for the more complicated transformations. To facilitate the process of specifying the constraints, one or two perpendicular projections of the 3-D model are displayed in separate windows. To input the constraints, points in any of the windows can be selected with the mouse and any numerical values needed for the constraints (distances, angles, etc.) can be entered from the keyboard.

Brüderlin's system, like MOLASYS, describes the objects using a language based on relations between points. Lines are defined simply by their two endpoints, but other relations are expressed as predicates. The predicates specify such things as the distances, slopes, and angles between points, the congruence of distances, slopes, and angles of sets of points, and the symmetry of points. The constraints are expressed using these predicates and their combinations. Construction rules are used to calculate the coordinates of the points. These rules are again written in the form of predicates and express geometric constructions, for example: given the angle α between three points A, B, and C, and the slope β between B and C, then the slope between B and A is $\beta + \alpha$.

To satisfy the constraints, Brüderlin's system tries to apply each of its rules. The user-specified constraints, expressed as predicates, are stored as facts in a Prolog database. When facts corresponding to the precondition of a rule are found, the rule fires. The process stops when no further rules can be applied. As the constraints aren't evaluated as a system of equations being solved for a set of parameters, occasionally the constraint-satisfaction algorithm terminates before solving for all the points in the object(s). If the object is underspecified, more constraints can be added and the process is retried. If there is a contradiction, the user is informed and can cancel or replace some of the constraints. This is a different approach than that used in numerical methods techniques in which the system still attempts to achieve a best-fit to the constraints, even if the system is over or under-constrained. For an under-constrained system, in the worst case, the results are not what was desired but the user can then specify more constraints. In the best case, the results are satifactory and no further input is needed. The same holds for contradictory constraints. Numerical methods are a definite advantage as it is preferable to find a solution than to go through the complicated process of determining when there is a contradiction and then to rearrange the constraints.

When the constraint-satisfaction algorithm has terminated, the database contains symbolic predicates for constructing the points which are then evaluated by Modula-2 procedures. When more than one solution is possible, an alternate solution is automatically displayed and the user can select the better of the two. In addition, a fact is stored for every rule that fires, allowing each step to be traced and interpreted later on. The explanations can then be shown to the user both in text and graphically.

Brüderlin's system is similar to MOLASYS in many ways: volumetric primitives, initial operations to define their shape, constraints for more complicated transformations, and a boundary representation of the models consisting of point coordinates, point-edge, edge-surface etc. relations. The main differences are the types of constraints and the method for constraint-satisfaction. In MOLASYS, the constraints are used to define an objct with respect to a background image, hence there is no real need for numeric values for the constraints as the model points can be measured directly to the points in the image. Also, the constraint-satisfaction is carried out as a system of equations. Brüderlin's symbolic approach seems more complicated but has the advantage of being able to detect and explain inconsistent and incomplete constraints and to provide alternative solutions. The former isn't necessarily an advantage as the system is unable to deal with these situations without more information.

Overall, the system accomplishes its main goal well: to provide an interface to facilitate the construction of 3-D models by constraints. Its constraint-satisfaction algorithm is perhaps more complicated than is necessary, but it does provide additional flexibility in evaluating the solution of the constraints.

2.2.4 Sequential Constraint-Satisfaction

A somewhat different approach to solving constraints was proposed by Rossignac [Ros86]. Most current systems convert constraints into a system of equations to be solved simultaneously by iterative numerical methods. In Rossignac's system however, constraints are evaluated one at a time in an order specified by the user. Rossignac is concerned with solid modelling applications in industrial design and believes that this system will provide a powerful tool for specifying and interactively editing parametrized models of mechanical parts.

As was described in an earlier section, Rossignac's work focusses on a system that is a dual representation, combining CSG and BRep using natural quadrics. The geometry is represented as a graph; the nodes indicate the objects and the branches are the constraints. The user can scroll through the constraints attached to each node, modifying or deleting the constraints or inserting new constraints. A constraint is evaluated by a method, producing a rigid motion (in the form of rotations and translations about a current axis) to satisfy that constraint. Evaluation of the graph is initiated by the user and the constraints are evaluated one at a time. When the evaluation is complete, the new positions of the objects are calculated and displayed. Instead of recording the rigid motions computed from the constraints, Rossignac's system stores only the unevaluated constraints. This has the advantage of easy editing of the constraints as they are avaliable to the user to be modified, however the disadvantage is that all of the constraints will have to be reevaluated each time.

Another disadvantage of this system is due to the sequential evaluation of the constraints. When the constraints are evaluated simultaneously, a best-fit for all the constraints is obtained. However, in sequential evaluation, the execution of a later constraint can invalidate the solution to an earlier constraint. Solving these situations involves either more complicated processing of the later constraints to ensure (if possible) that their solutions are compatible with those of the earlier constraints, or more care by the user to avoid such situations (again, not always possible), or both.

The goal of this system was to simplify the specification of geometry, not to develop a problem-solver, hence the system is not capable of solving problems where many constraints must be satisfied simultaneously. However this approach does avoid the problems involved in converting the results of simultaneous constraint evaluation into CSG representation (while CSG is a common representation, it is difficult to interactively modify as displaying objects in CSG is computationally intensive). Also, Rossignac's procedural approach may be more appropriate for the industrial domain, i.e., sequences of rotations and translations can be viewed as a description of assembly processes. A transformation specified by a set of constraints is broken down in Rossignac's system into a sequence of operations by the user. Rossignac's system provides a tool for interactively modifying parametrized models of mechanisms and is a possible alternative to constraint-based systems involving the simultaneous evaluation of large sets of equations. However, only methods for computing rigid motions from simple constraints and for modifying/processing the constraint graph have been implemented so far. It is not clear whether the system can be developed to a point where it will be of use in the applications for which it was intended as the sequential processing can not handle more complicated constraint specifications easily. The small advantages gained by the approach may not offset the possible losses in efficiency and robustness.

Another system using a sequential constraint-satisfaction approach is that of **Snap-Dragging** by Bier and Stone [Bie86a]. Initially, this was a two-dimensional program for the interactive creation of precise line drawings. The constraints are in the form of relationships that hold between points and lines. Snap-dragging uses the draftsman's ruler and compass metaphor to construct the shapes accurately.

Snap-dragging snaps the cursor to points and curves using a gravity function. There is also a set of gravity-active points, lines (of varying orientations), and circles called *alignment objects* available from mouse-sensitive menus. These alignment objects are used to design models by allowing intersection points and curves to be computed automatically. the user can snap to the vertices and points of intersection, using the geometry of the alignment objects to create the models. For example, an equilateral triangle is built from a straight line and two circles of equal radius. First, the two base verteces are selected on the line. Then two alignment circles of a specified size are generated, each centered on one of the vertices. The third vertex is then calculated at the intersection point of the circles. When the triangle is formed, the alignment objects disappear. Translation, rotation and scaling are also provided. The system has been extended to three dimensions ([Bie88], [Bie90]), incorporating surfaces (such as quadrics and spline patches) defined by control points. A surface is transformed by applying the snap-dragging operations to the surface's control points.

Snap-dragging was designed to accomplish the goal of making precise line drawings. Using this system, creating the models would be relatively easy from a set of measurements (i.e., for the domain of mechanical assembly for which it was intended), but quite difficult from images as the constraints require quantitative information. Bier and Stone believe that conventional constraint-based systems are difficult to control as these systems use large sets of simultaneous equations. The solution to these equations can affect many parameters and possibly have multiple solutions. To avoid these potential problems (but not necessarily serious problems, as will be discussed later), in snap-dragging only one point is moved at a time and the constraints are satisfied immediately and then discarded. Bier and Stone claim this approach is easier to use, providing the operations a designer would use to construct an object. However it also results in a much less powerful system, possibly requiring several sequential steps that could be accomplished in one larger step in a constraint-based system.

Another application for sequential constraint-satisfaction is that of modelling complex shapes. Complex surfaces are frequently represented as polygon meshes, consisting of specifications of vertices, edges, and faces. Instead of specifying the vertices directly, the relationships between the vertices can be defined, resulting in a set of constraints. These relationships can constrain the vertices to be colinear or coplanar, can specify the area of the faces, can set the length of the edges between the vertices, etc. Prusinkiewicz and Streibel [Pru86] have used the latter form of constraints in a modelling system for complex three-dimensional shapes. In addition, lines can be constrained to be parallel to a plane and specific vertices can be fixed in space. The mesh consists of polygons expressed in terms of edges, which in turn consist of vertices. The system of constraints for n vertices consists of $3n$ equations (each representing a constraint) with $3n$ unkown vertex coordinates. The equations (quadratic) are solved simultaneously using Newton's method. However, to avoid either underconstrained or overconstrained ystems, an additional technique was implemented. The mesh being defined is thought of as the last element in a sequence of submeshes. the first element in the sequence is simply defined and solved, then each subsequent submesh differs from its predecessor by a few additional vertices and edges. When calculating a new submesh, the vertices of the previous submesh are already known so only a small set of equations needs to be solved at each step. Thus the constraints are evaluated sequentially in an order imposed by the sequence of submeshes. While there are meshes that can not be decomposed into a sequence of submeshes, in many cases this decomposition is very natural, i.e., the sequence corresponds to the steps involved in the real construction of an object. The technique can also be used for modelling shapes found in nature as the generation of successive submeshes also imitates growth. For a more detailed explanation, refer to [Pru86]. This relatively uncomplicated technique provides an interesting alternative in constraint-based modelling although its potential for interactive model design appears limited.

2.2.5 Deformable Models

Several references have already been made to modelling representations for free-form, or deformable surfaces, e.g., [Ost86], [Thi90]. A deformable surface is typically represented as a grid of points, the points being able to move in relation to each other in a manner determined by the properties of the surface. For example, in a model representing an elastic surface the grid points are connected by springs. Deformable surfaces are not simply surfaces with deformation operations applied to them as they possess properties independent of the operations, e.g., the spring relation between the points. When an external force is applied to a set of grid points, the points' motion simulates the stretching of a real elastic surface.

Terzopoulos et al. ([Ter87], [Ter88]) use *symmetry-seeking deformable models* constrained by strain energy functions to create simulated elastic shapes. The models initially consist of a deformable tube and attached deformable spine centered in the tube by radial spring forces. The shape of the model (tube) is controlled by expansion/compression forces radiating from the spine. These forces are applied to the model based on cues derived from images. The user initially establishes the spine against a background image. The spine is centered inside the image, running along its length. The model (spine, at this stage) than begins to inflate (due to internal expansion forces) and as it deforms, it is dynamically projected onto the image. The boundaries are attracted to significant intensity gradients in the image and the shape of these model boundaries is controlled by external forces derived from the image data. The models reconstruct 3-D shapes by finding a stable equilibrium between the externally applied forces (derived from the image or the user) and the internal forces due to the model's deformation constraints. In this way, models are created from image data with some user interaction. The user can guide the reconstruction process by interactively specifying additional forces or altering the model's material properties. While a variety of objects can be formed, the domain is limited to elastic shapes possessing axial symmetry. It does not appear possible to use this method for polyhedral objects, as in the domain for MOLASYS.

Sederberg and Parry [Sed86] have used a different approach to model deformation. They have developed a technique for deforming solid models created with existing methods, e.g., CSG, B-rep. The *free-form deformations* (FFDs) are defined as mappings from $\mathbf{R}^3$ to $\mathbf{R}^3$ using Bernstein polynomials. Sederberg and Parry claim FFDs can be applied to virtually any geometric model and modify only the geometry without altering the integrity of the model. Thus the result is a valid model, allowing the calculation of points, etc. A wide variety of objects can be created (e.g., telephone receivers, trophy cups) although the ease of use of the system wasn't clear from the paper.

One of the better known methods for deformable models uses *superquadrics*; shapes formed by a 3-D vector sweeping out a parametrized surface resulting in cubes, spheres, pyramids and many intermediate shapes. **Supersketch** by Pentland [Pen86] is a three-dimensional interactive modelling system which deforms the superquadrics by stretching, tapering, bending, etc. and combines them to form complex shapes. A similar idea was used in **Thingworld** [Pen90] which has the added feature of automatic modelling, i.e., Thingworld can automatically generate models from measurements of real objects. The user first interactively positions deformable models approximately over each part in the image to be modelled. Thingworld then calculates and numerically minimizes the error between the model's initial position and the measured point data, producing a three-dimensional model (composed of deformed parts) that closely fits the object's measurements. The process of "fitting" is accomplished by giving each data point (of the object's measurements) an artificial gravity. Forces are then generated from these gravity fields causing the model to deform dynamically to fit the data. Pentland reports good response time for a variety of object models. This method shows a lot of promise for our application as well as the models could be fit directly to the images by deriving the object measurements automatically from the image. This might then be easier than the MOLASYS approach as the user would not be required to specify the constraints manually.

2.3 Interface Design

2.3.1 Human-Computer Interaction

In recent years there has been a perceptual shift from viewing a user interface as something grafted on to an existing system to the idea of designing the complete system (including the interface) to fit a model of how people function. For modelling, this means examining how people think about and describe shapes, and how this changes over the course of creating a model from the initial sketching of the shape to detailing the model ([Pen87]). The user interface then becomes a logical extension of the system as a whole, and not a separate entity added on to make the system useful. To accomplish this task, we need to consider how people interact with the computer, i.e., how the users perceive, process, and respond to the information presented by the system. The goal then becomes one of reducing the work the user must perform in each of those functions to provide an intuitive, efficient system.

In using an interactive system, the user is influenced by psychological factors such as personality, experience, and knowledge. As the goal is to design a system to aid the user to maximum advantage in performing a certain task, the interface (i.e., style of interaction) should reflect and enhance the user's perceptual and cognitive processes. While numerous aspects of human-computer interaction remain poorly understood, many studies have been performed into a variety of factors (e.g., perception, knowledge acquisition) affecting successful user interaction, such as [Fol84], [Mor87], [Dil87], [For86].

Perception is the process of reception and recognition of physical stimuli by the sense organs - mainly visual organs in computer use but also auditory and tactile (for interaction devices). As most of the tasks in using an interface involve locating menus, cursors, and a variety of other entities, a major consideration is displaying information in a manner that allows for quick recognition of needed items. To attract the user's attention to specific parts of the display variables such as colour, layout, brightness, motion, reverse video, line thickness, fonts, etc. can be altered. Cognitive processing involves the acquisition, organisation, and retrieval of information. An appreciation of cognitive factors provides insights into how users learn to use interaction techniques. Learning a task involves organising the relevant information. If the information can be easily sorted into categories or concepts, learning occurs more rapidly. Hence the study of cognition provides guidelines for structuring hierarchical menus, grouping choices (e.g., a menu of twenty choices is easier to understand if the choices are grouped into several logical subgroups), selecting command names, etc.

A system that is easy to use will be predictable, require user input for a task in proportion to the difficulty of the task, and provide good error-handling facilities (e.g., detect and correct errors, give suitable messages, and prevent the loss of work). The user should be able to perform the desired work with minimal conscious attention to the tools of the interactive system. The tools should thus be intuitive (i.e., the system structured to allow the user to perform tasks in a natural, logically structured manner) and efficient (i.e., no long delays between steps). Another desirable feature is to facilitate user "trial and error" approaches, i.e., allow the user to backtrack, or undo, certain steps.

Above is a brief examination of some of the factors affecting the success/failure of a user interface. As this field gains in importance and experience, better models and guidelines will become available allowing for the creation of truly "user-friendly" systems.

2.3.2 Input Devices

A related issue specific to the domain of three-dimensional modelling systems is that of selecting input methods that allow users to design three-dimensional models using as feedback only two-dimensional images. Several input devices exist which permit the user to work in three dimensions at once, but this often leads to confusion as only two dimensions can be displayed at a time resulting in inadequate or confusing feedback to the user. The next few paragraphs will briefly discuss some existing approaches to this problem and the difficulties and successes encountered.

Chen et al. [Che88] were concerned with finding a method that would allow easy manipulation and control of the position of an object in three dimensions. Their approach was to focus on virtual controllers in conjunction with a mouse. Sliders and menus were considered but these have poor kinesthetic correspondence between the direction of mouse movement and that of object rotation. Other systems, such as touch-sensitive tablets (e.g., [Hil87]), can provide good accuracy but are difficult to learn. The virtual sphere controllers implemented by Chen et al. simulate a 3-D trackball that can freely rotate about any axis in 3-space. On the screen the virtual sphere is overlaid on the object to be rotated and the user can imagine viewing the object encased in a sphere. Rotation is then accomplished by rolling the sphere (and object) with the mouse. Up-and-down and left-and-right motion at the centre of the circle produces rotation about the x and y axes respectively, and rotation about the z-axis is accomplished by movement along the outside of the sphere. While this results in intuitive, easy-to-use rotation, it is also more complicated to implement. Instead, MOLASYS uses menus to rotate about each of the three axes separately. The facility provides straightforward rotation although the process is slightly less flexible and slightly more time-consuming for the user.

Edwards et al. [Edw88] adopted a different approach in designing a 3-D modelling system. The central concept, the *Cutplane*, consists of a plane that moves through space under mouse control. The intersection of the plane and any object is highlighted and only this region can be selected for manipulation. In this way, Edwards et al. hoped to create a system that would allow for intuitive and unambiguous manipulation of points, vertices and faces in three dimensions. Badler et al. [Bad86] experimented with using the *Polhemus*, a six degree-of-freedom sensor, for manipulating and positioning 3-D objects. The Polhemus wand may be freely translated or rotated in space, and senses its position and orientation relative to a magnetic source. While it might seem that the user should be able to move objects in an intuitive manner, Badler et al. discovered the opposite. The 2-D screen image didn't provide adequate feedback for operating the Polhemus in three dimensions. Also, positioning and orienting an object at the same time proved difficult as there were too many degrees of freedom to control simultaneously. There is a balance between allowing the user sufficient degrees of freedom to be able to use the system to best advantage and providing some constraints on the environment so that there are not too many choices as to be overwhelming. Providing adequate spatial feedback is another important problem as only two dimensions (for now) can be displayed in an image. Providing several views (both orthogonal and perspective) appears to be the best solution so far.

Chapter 3

MOLASYS

As has been evidenced, graphical systems for object modelling are steadily improving with many recent advances in the area of interactive three-dimensional modelling systems. In the systems examined previously, a variety of different interface styles were designed with the goal of minimizing the losses in performance due to tradeoffs between accuracy, flexibility, speed, memory costs, and ease-of-use. As a system can only provide the above features with varying degrees of success, it is necessary to design the modeller considering the specific features that are important to the applications for which the modeller will be used.

MOLASYS has been designed for computer vision applications which frequently use three-dimensional models for matching and recognition tasks. A current problem in this area is the creation/acquisition of these models. There was a specific need for a system that would provide for the interactive construction of models from input images. The images contain the objects to be modelled, viewed from several different viewpoints. The models are overlaid on the images (presented in adjacent windows), allowing the user to build the models from the objects in the images. This approach takes advantage of the natural constraints (e.g. dimensions, shape) for constructing the models provided by the image, thus simplifying the design of the three-dimensional models.

3.1 Modelling Representation

The primitives are displayed with a boundary representation using polygonal approximations for the surfaces. Back surface removal is provided although the representation has otherwise been kept relatively simple. More complicated rendering functions (e.g., illumination, shadows) can be added from a standard graphics package.

While many aspects of MOLASYS are in the computer graphics domain, the models created will be used for computer vision tasks such as matching, and will have different requirements from the models typically used in graphics applications. For example, graphics models are frequently used to display realistic, detailed scenes involving a complicated rendering process for the model surfaces, whereas vision applications are more concerned with matching to image features and contours. Hence, a system is needed which can calculate points, edges, etc. and also quantities such as partial derivatives with respect to model parameters rapidly and accurately. The system should also be hierarchical in nature, allowing easy access to the component parts and parameters of the model.

From these considerations, a modelling language was developed [Low89] to describe models and their internal parameters. Called ModelScript, the language is an interpreted, device-independent language that uses Lisp syntax. As an example, the definition of a simple rectangular solid is shown in figure 3.1. The primitives currently defined are shown in figure 3.2.

The top level of the model is the object structure containing pointers to the faces, edges, and points that define it. Each face consists of pointers to its bounding edges in addition to surface normal data for calculating the orientation of the face. Each edge contains pointers to both of its endpoints as well as to the one or two faces it delimits. A location vector is stored in each point, along with a list of the edges to which the point belongs and a list of the variables in the location expression. The lowest level of the model is the variable which contains only its current value and the range of values it can inhabit.

When the object is first built from this input, the pointers are established and the surface information and point locations are calculated. There is a caching mechanism for the visibility and location information to prevent calculation of the information more than once for the same viewing position. The representation also contains facilities for specifying edges in a model to be "occluding". For example, the edges along the length of the cylinder primitive could be defined in this manner and then would only be displayed if exactly one of an edge's adjoining faces were visible. Thus, only the bounding contours

```
;;VARIABLES
(variable length  0.1  3.0)
(variable width   0.1  3.0)
(variable height  0.1  3.0)

;;POINTS
;;point names are of the form xyz :
;;  x - 'l' for left,  'r' for right,
;;  y - 'l' for lower, 'u' for upper,
;;  z - 'f' for front, 'b' for back.
(point llf ((* width -5.0) (* height -2.0) (* length  3.0)))
(point rlf ((* width  5.0) (* height -2.0) (* length  3.0)))
(point luf ((* width -5.0) (* height  2.0) (* length  3.0)))
(point ruf ((* width  5.0) (* height  2.0) (* length  3.0)))
(point llb ((* width -5.0) (* height -2.0) (* length -3.0)))
(point rlb ((* width  5.0) (* height -2.0) (* length -3.0)))
(point lub ((* width -5.0) (* height  2.0) (* length -3.0)))
(point rub ((* width  5.0) (* height  2.0) (* length -3.0)))

;;EDGES
;;edges between left and right points
(edge lower-front llf rlf)
(edge upper-front luf ruf)
(edge lower-back  llb rlb)
(edge upper-back  lub rub)
;;edges between upper and lower points
(edge left-front  luf llf)
(edge right-front ruf rlf)
(edge left-back   lub llb)
(edge right-back  rub rlb)
;;edges between front and back points
(edge left-lower  llf llb)
(edge right-lower rlf rlb)
(edge left-upper  luf lub)
(edge right-upper ruf rub)

;;FACES
(face front lower-front right-front upper-front left-front)
(face back  lower-back  right-back  upper-back  left-back)
(face upper upper-front right-upper upper-back  left-upper)
(face lower lower-front right-lower lower-back  left-lower)
(face left  left-lower  left-back   left-upper  left-front)
(face right right-lower right-back  right-upper right-front)

;;OBJECT
(object rect-prim front back upper lower left right)
```

Figure 3.1: definition of a rectangular primitive using Modelscript

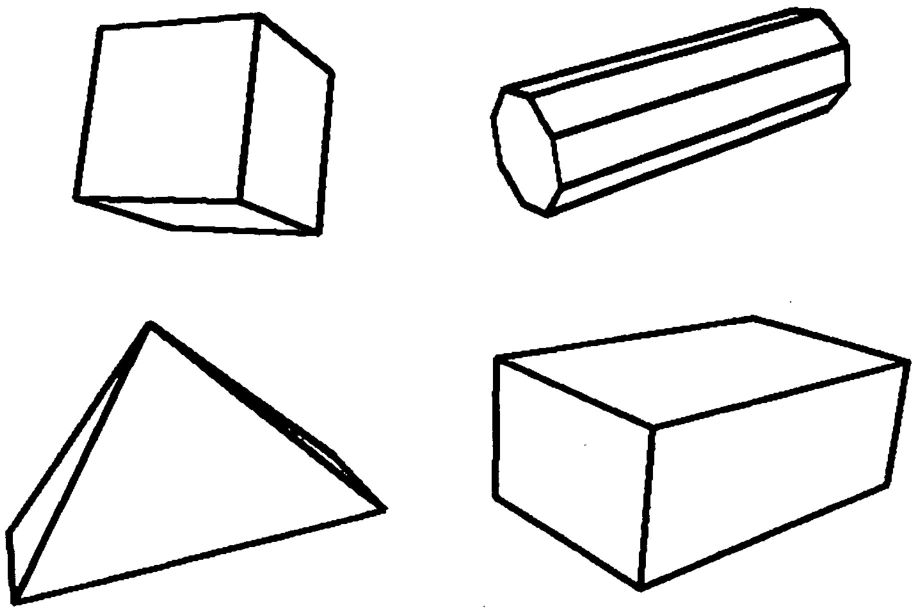


Figure 3.2: three-dimensional primitives defined in MOLASYS

are displayed and not the interior edges, resulting in a smoother appearance.

The locations of the points are defined in terms of variables to allow for flexible manipulation of the object's dimensions without affecting the object's basic shape. The point locations are expressed with respect to an origin at the centre of the three-dimensional solid, thus allowing the model to be stretched in either direction along each of the three axes.

With this representation, the locations of points and edges of the model can be calculated accurately in an efficient manner. As will be seen later, the representation also allows for efficient calculation of the partial derivatives of the object parameters. The output of the modelling system will consist of the models in this format. These output files will then be read by C programs to carry out various vision applications.

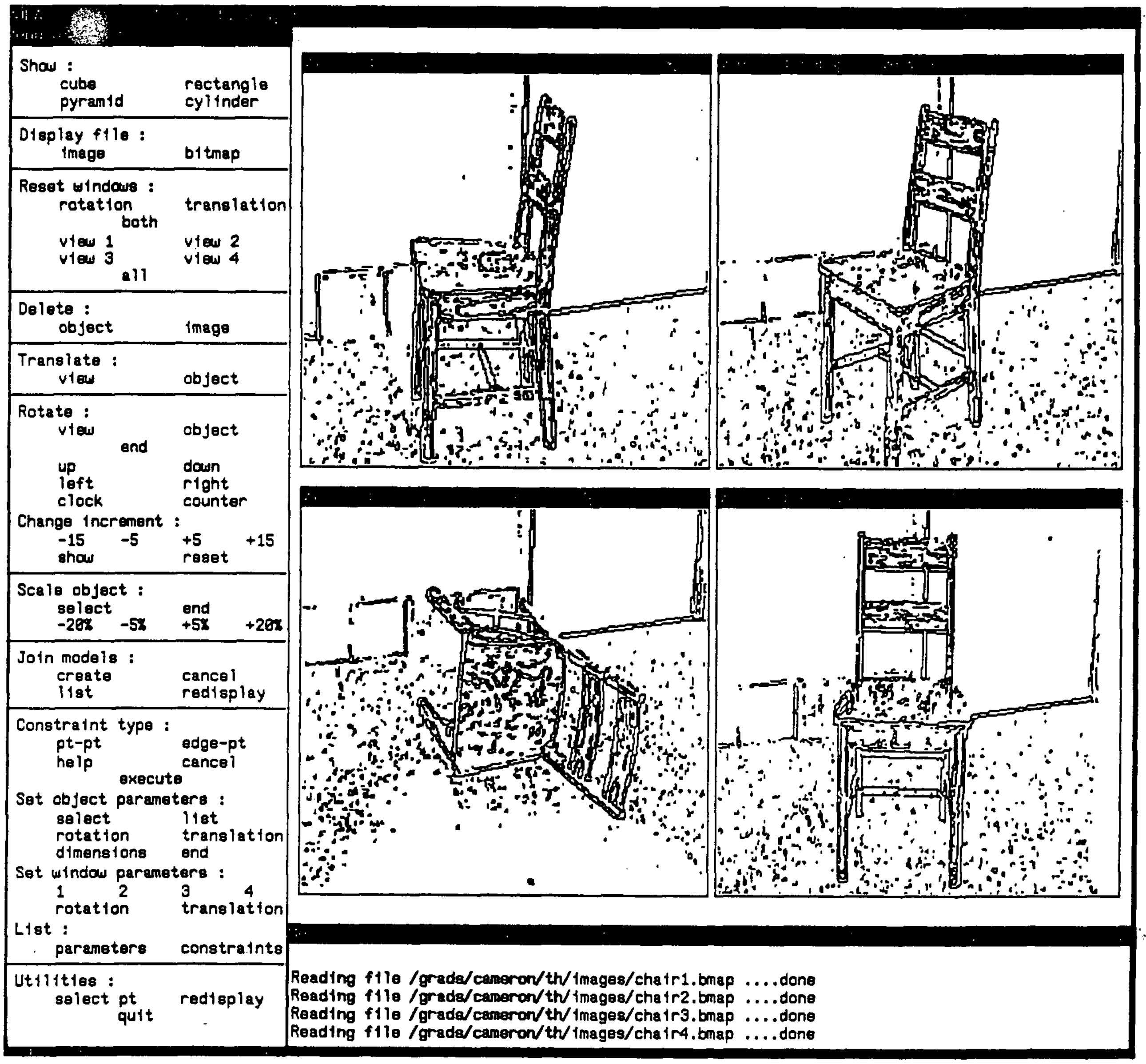
The modelling primitives currently available are still limited, but with the high-level specification of ModelScript, more can be added without too much difficulty. For more details, see [Low89].

3.2 Interface

The MOLASYS interface is shown in figure 3.3. There are three main parts to the interface: four display windows, a mouseable command menu, and a message panel. All of the commands are contained in the menu, no keyboard input is required. When the user must respond to a choice during execution of a command, a pop-up menu listing the choices is provided. The construction of the models takes place in the four windows. As can be seen in figure 3.3, images of an object or scene from four different views are displayed in the background of the windows. Each view corresponds as closely as possible with the initial orientation of the camera for each window. The camera for each window has six degrees of freedom: translation and rotation about each of x, y, and z. Initially, the cameras for all four windows have the x and y translation set to zero and a z translation of -100. This creates a perspective projection of the three-dimensional models when they are displayed with the centre of projection lying along the negative z axis, a distance of 100 pixels behind the plane of projection (i.e. the window). In this implementation, views 1,3, and 4 represent the xy, xz, and yz views respectively, while view 2 is a perspective projection rotated about the y axis somewhat. These views are arbitrary and the windows can contain any desired viewpoint. These views are accomplished by adjusting the rotation matrices of the cameras. The first window initially has the entries of its rotation matrix set to zero and is thus an xy view of the object/scene, projecting along the z axis. The third window shows the xz view of the object/scene, being rotated about the x axis by 90 degrees. The fourth window is rotated about the y axis by 90 degrees, offering the yz view. The camera of the second window is rotated slightly about the y axis, offering a different perspective of the object and giving a better impression of the three-dimensional structure of the object. All of these views are initial and can be easily modified either by the user from the command menu or in the source code.

3.3 Transformation Functions

In MOLASYS, the objects' shape, position, and orientation are manipulated primarily through the use of constraints. However individual transformation functions (translation,



```
Reading file /grads/cameron/th/images/chair1.bmap ....done
Reading file /grads/cameron/th/images/chair2.bmap ....done
Reading file /grads/cameron/th/images/chair3.bmap ....done
Reading file /grads/cameron/th/images/chair4.bmap ....done
```

Figure 3.3: The MOLASYS interface including a menu (left), four display windows, and a message panel (bottom)

rotation, scaling) are available to the user for the initial placement of the models and for minor adjustments.

3.3.1 Translation

Translation can be accomplished in two ways: by changing the translation matrix of the camera of a specific view and thus effectively moving all of the models in one window by the same amount, or by translating one object with respect to the rest of the models. In the latter case, the point coordinates of the individual model are changed to reflect the desired translation, and all four windows are redisplayed showing the resulting changes to the models' positions with respect to each other and to the scene. In both cases of translation, the user specifies (through use of the mouse) the window/object to be translated and the amount of that translation.

The translation is calculated using the following method. The image coordinates (u,v) of a point, rotated and translated by the camera for a given window, can be expressed as

$$u = \frac{f(x_r + T_x)}{z_r + T_z}$$

and

$$v = \frac{f(y_r + T_y)}{z_r + T_z},$$

where

f is the focal length of the camera

(x, y, z) are 3D coordinates of model point

$(x_r, y_r, z_r)^T = R(x, y, z)^T$ are model point coordinates rotated by the camera

$T = [T_x, T_y, T_z]^{-1}$ is the translation vector of the camera

R is the rotation matrix of the camera.

To translate an object, the user mouses on a vertex in the model and then on a point to which the vertex is to be moved. Considering only u (v can be calculated in a similar manner), the new image coordinate u' after the user specified translation is

$$u' = \frac{f(x_r + T'_x)}{z_r + T_z}.$$

u and u' are known from the coordinates of the moused points, and we are trying to solve for T'_x. So

$$u' - u = \frac{f(x_r + T'_x) - f(x_r + T_x)}{z_r + T_z}$$

and

$$T'_x = \frac{(u' - u)(R_z z + T_z)}{f} + T_x.$$

$T'_x - T_x$ is the user-specified translation along u, and together with $T'_y - T_y$ defines the neccessary changes to the translation vector of a camera for a view translation. For an object translation however, we want to express the translation as a change in the point coordinates of the model. So if

$$(x, y, z)_{r,t} = R(x, y, z) + T,$$

then

$$(x, y, z)'_{r,t} = R(x, y, z) + T',$$

where

$(x, y, z)_{r,t}$ are 3D model coordinates rotated and translated by the camera
$(x, y, z)'_{r,t}$ are 3D model coordinates rotated and translated by the camera and translated by the user-specifed translation
$T' = [T'_x, T'_y, T'_z]^{-1}$ is the translation vector of camera, including the user-specified translation.

Expressing the new point as a change in the point coordinates and not of the translation vector yields

$$(x, y, z)'_{r,t} = R(x, y, z)' + T,$$

where

$(x, y, z)'$ are the new coordinates of the model point translated by the user.

Therefore

$$R(x, y, z) + T' = R(x, y, z)' + T$$

and

$$(x, y, z)' = (x, y, z) + R^{-1}(T' - T).$$

The equation is simplified as $R^{-1} = R^T$, hence

$$(x, y, z)' = (x, y, z) + R^T(T' - T).$$

3.3.2 Rotation

Rotation can also be done by rotating either a view or an object. For rotation, once a view or object has been selected, it can be repeatedly rotated by a variable increment in either direction along any of the three axes. The increment is initially set to 15 degrees and can be changed by the user by adding or subtracting from this increment. This allows the user to choose a suitable accuracy for manipulating the objects.

Rotation of a view is calculated directly by multiplying the new rotation matrix derived from the rotation angle with the camera rotation matrix. Rotating an individual object is more complicated. Using the notation from the previous section,

$$(x, y, z)_{r,t} = R(x, y, z) + T, \tag{3.1}$$

and

$$(x, y, z)'_{r,t} = R(x, y, z)' + T.$$

Therefore,

$$(x, y, z)'_{r,t} - (x, y, z)_{r,t} = R((x, y, z)' - (x, y, z))$$

and

$$(x, y, z)' - (x, y, z) = R^{-1}((x, y, z)'_{r,t} - (x, y, z)_{r,t}). \tag{3.2}$$

Also,

$$(x, y, z)'_{r,t} = R_\theta((x, y, z)_{r,t} - T) + T,$$

where

R_θ is the rotation matrix representing the angle the object is to be rotated.

Using equation (1), the last equation becomes

$$(x,y,z)'_{r,t} = R_\theta(R(x,y,z)) + T. \tag{3.3}$$

Substituting (1) and (3) into (2),

$$\begin{aligned}(x,y,z)' - (x,y,z) &= R^{-1}(R_\theta R(x,y,z) + T - R(x,y,z) - T) \\ &= R^{-1}(R_\theta R(x,y,z) - R(x,y,z)).\end{aligned}$$

Hence,

$$\begin{aligned}(x,y,z)' &= R^{-1}R_\theta R(x,y,z) - R^{-1}R(x,y,z) + (x,y,z) \\ &= R^{-1}(R_\theta R(x,y,z)).\end{aligned}$$

To rotate about the centre of the object and not that of the window, the last equation becomes

$$(x,y,z)' = R^{-1}R_\theta R((x,y,z) + T) - T.$$

3.3.3 Utility Functions

The interface for scaling the object is similar to that for rotation. The selected object is scaled smaller or larger in increments of 5% or 20%, and the increments can be changed in the source code. As the point coordinates are defined using variable expressions, scaling is executed simply by altering the values of the objects' parameters by the specified percentage, thus maintaining the shape of the object while changing its overall dimensions.

Other functions provided are deleting of models and/or bitmaps, resetting of the original translation/rotation of the views, and a "select point" function. This latter function allows the user to mouse a model point and have it named and highlighted in the windows. This is helpful in mentally orienting the different views of the model.

3.3.4 Joining Models

Another feature that is provided in the MOLASYS interface is the ability to join objects. Using this command, more complex objects can be built up from the simpler models by specifying pairs of points that are to remain attached. A global list stores the *joins* as a list of sublists, each sublist consisting of the first object and its joined point followed by the second object containing the other joined point. The joins are specified simply by mousing on the two points to be joined, and can be cancelled in a similar manner. The system automatically aligns the two points when a join is specified. After a join has been declared, any operation that affects one point (i.e., translation and rotation, but not scaling and deletion) will affect the other. The constraint-satisfaction process will also maintain these joins when calculating the partial derivatives, as explained in the next section.

3.4 Constraint-Satisfaction Process

Do not worry about your difficulties in mathematics,
I can assure you that mine are still greater.

- Albert Einstein

The main feature of this object-modelling system is the *constraint-satisfaction* module. With these techniques, the user can specify directly by how much the models should be changed, and it is left to the program to determine how the individual parameters should be altered to reflect these changes. In the next few sections, the constraints, parameters and constraint-satisfaction algorithm will be discussed in detail.

3.4.1 Constraint Specification

The user specifies the constraints in one of two forms: *point-to-point* or *edge-to-point*. In point-to-point constraints, the user mouses a point in an object model and then a point in the background scene to which the object point is to be moved. The constraint is displayed as a dashed line drawn between the two points. In edge-to-point constraints, the user first mouses on the two endpoints of an edge in a model, and then on a point in the background to which the edge is to be moved. The edge and point are highlighted and a dashed line is drawn along the perpendicular distance between them. It is this perpendicular distance that is minimized during the constraint-satisfaction process. The edge is not necessarily kept parallel to its original orientation when it is moved, however if several constraints are specified the system will have sufficient information to yield the desired orientation of the object model(s).

Any number of constraints can be defined on any or all of the models. The constraints are displayed in the windows in which they were specified. Initially, it is assumed that the system will solve for all possible parameters: window translation and rotation (about each of x, y, and z), and the translation, rotation, and dimensions (width, length, and height) for each object. This can lead quite rapidly to a large number of parameters, however performance has so far been good. Parameters not affecting the constraints aren't changed by the algorithm. If efficiency becomes a problem, parameters can be removed from the list of parameters for which the system will be solving.

3.4.2 Algorithm

When all the desired constraints have been entered, the system will attempt to solve all the constraints simultaneously. This may not always be possible, and there are techniques for systems that are either under or over-constrained. Stabilizing methods have also been employed to improve the performance of the system. The constraints are carried forward after each execution and any new constraints will be added to the current list unless specifically deleted by the user.

The main steps in the control loop for the constraint-satisfaction process are as follows:

1. specify the constraints and parameters
2. calculate the partial derivatives
3. solve for the parameters
4. compute new parameter values
5. if constraints aren't satisfied and system hasn't converged
   then return to 2 using new parameter values

6. redisplay models using new parameter values

The constraint-satisfaction process is initiated by the user from the menu and iterates until all the constraints have been solved or the system has converged to a best-fit solution. The constraints are considered solved when each of the error measurements in the constraints has been reduced to within a predetermined threshold (e.g., the points are aligned to within a few pixels). The process also halts when the corrections to the parameters have become quite small. In these cases, the constraints can't all be satisfied completely. When the computed corrections to the parameters are small (e.g., less than 5-10% of the value of the parameter), the iteration is stopped and the system is said to have converged to a solution that best fits the data. In the event that the system is not converging, or has not been solved, after a threshold number of iterations (e.g., 6-10), there is a check that halts the iteration and displays the current estimates of the parameters.

3.4.3 Constraint Evaluation

The constraints specified by the user are represented as a vector of error measurements, **e**, between components of the model(s) and the background image. Using Newton's method, MOLASYS iteratively alters a vector of non-linear parameters, **p**, (containing a subset of the object and camera parameters discussed above) to reduce these errors to zero. This process of satisfying the constraints is implemented as a system of linear equations

$$\mathbf{J}\mathbf{x} = \mathbf{e},$$

where **x** represents the vector of corrections to be made to the parameters to minimize the errors. **J** is the Jacobian matrix of partial derivatives of the errors with respect to the parameters,

$$J_{ij} = \partial e_i / \partial x_j.$$

The effect of each parameter correction x_i on an error is x_i multiplied by the partial derivative of the error with respect to that parameter. Thus each row i of the above matrix equation expands to

$$e_i = \frac{\partial e_i}{\partial x_1} x_1 + \frac{\partial e_i}{\partial x_2} x_2 + \cdots + \frac{\partial e_i}{\partial x_m} x_m,$$

where m is the number of parameters, stating that each error e_i is equal to the sum of the changes in that error resulting from the parameter corrections. Thus on each iteration, the system is solving for the vector $\mathbf{x}$ of corrections to be subtracted from the current vector of parameter estimates, $\mathbf{p}$:

$$\mathbf{p}^{(i+1)} = \mathbf{p}^{(i)} - \mathbf{x}.$$

3.4.4 Jacobian Matrix

The first step in the process of evaluating the constraints is calculating the Jacobian matrix of partial derivatives. As stated earlier, each entry J_{ij} represents the partial derivative of a constraint error with respect to a parameter. Each row of $\mathbf{J}$ contains the partial derivatives of one error with respect to each of the parameters, and each column contains the partial derivatives of all the errors with respect to one parameter. Hence to calculate the Jacobian matrix, the following algorithm is used:

```
for column 1 to the number of parameters do
    perturb the value of the parameter corresponding to the current column
    if the parameter is an object parameter and there are joined models
    then change the joined models to reflect the change
    for row 1 to the number of constraints do
        measure the constraint error using the perturbed parameter value(s)
        $J_{row,col}$ = change in the error/change in the parameter
    return the parameter(s) to the original value(s).
```

The parameters are altered by 1% of the range of values the parameter can possess, to a minimum of 0.01, and the constraints are recalculated to measure the effects of this change. Any columns that contain only zeros (i.e. the parameter has no effect on any of the constraints) are removed from the Jacobian matrix, as are the corresponding parameters from the vector of parameter corrections, $\mathbf{x}$, for which the system is solving. The user can also "turn off" parameters that he/she does not want to be altered. The system then does not solve for these parameters and they retain their current values. Functions are provided in the menu to turn any of the parameters on or off, i.e., to add or remove the parameters from $\mathbf{x}$.

To maintain the connectivity of objects that have been joined, any object model joined to an object with a parameter being altered by the above algorithm may also be altered (depending on the parameter). For object rotation and translation parameters, the joined object models are rotated/translated by the same amount as the current object. For object dimension parameters, the joined points are checked. If the (u, v) coordinates of the points are no longer equal, the joined models are translated to compensate for the change in the size of the model. In this way the implementation of joined objects is achieved without being required to change the object models.

3.4.5 Solving the System

As there will be more constraints than parameters in some cases (i.e., the system is overconstrained), we solve for an **x** that minimizes the 2-norm of the residual instead of solving for it exactly. This is expressed as $min \parallel \mathbf{Jx} - \mathbf{e} \parallel^2$, and since $\parallel \mathbf{Jx} - \mathbf{e} \parallel^2 = (\mathbf{Jx} - \mathbf{e})^T(\mathbf{Jx} - \mathbf{e})$, the equation becomes

$$\mathbf{J^T J x} = \mathbf{J^T e}.$$

Thus, each iteration of Newton's method computes $\mathbf{J^T J}$ and $\mathbf{J^T e}$ and solves for **x** using Gaussian elimination with partial pivoting.

When there are more constraints on the solution than unknowns (parameters), the above method will usually converge to a solution. However there will be cases with more unknowns to solve for than there are constraint errors. To stabilize the solution for these underconstrained systems (and for other systems which have ill-conditioned solutions for various other reasons), prior constraints are added that specify the defaults to be used when there is insufficient data. The prior constraints are implemented by adding rows to the existing system, specifying values for the parameters:

$$\begin{bmatrix} J \\ I \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{e} \\ \mathbf{d} \end{bmatrix}.$$

The identity matrix **I** adds one row for each parameter i, assigning it the value d_i. In this implementation, $d_i = 0$ for all i as we would like the defaults to be zero changes for the parameters. In the absence of futher constraints on the solution, the parameters should stay at their current values. To define the importance of the user-specified constraints versus the prior constraints, each added row of the matrix equation is weighted by a diagonal matrix $\mathbf{W}$. Each weight w_{ii} in $\mathbf{W}$ is inversely proportional to the standard deviation σ_i for parameter i, i.e. $w_{ii} = 1/\sigma_i$. Thus each constraint contributes in proportion to the amount of deviation from its expected value. (The constraints from the image are assumed to be scaled to have unit standard deviation). The resulting equation is

$$\begin{bmatrix} \mathbf{J} \\ \mathbf{W} \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{e} \\ \mathbf{Wd} \end{bmatrix},$$

which can be solved by

$$\begin{bmatrix} \mathbf{J}^{\mathbf{T}} \mathbf{W}^{\mathbf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{J} \\ \mathbf{W} \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{J}^{\mathbf{T}} \mathbf{W}^{\mathbf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{e} \\ \mathbf{Wd} \end{bmatrix}.$$

This yields

$$(\mathbf{J}^{\mathbf{T}}\mathbf{J} + \mathbf{W}^{\mathbf{T}}\mathbf{W})\mathbf{x} = \mathbf{J}^{\mathbf{T}}\mathbf{e} + \mathbf{W}^{\mathbf{T}}\mathbf{Wd},$$

and since $d_i = 0$ for all i (explained above)

$$(\mathbf{J}^{\mathbf{T}}\mathbf{J} + \mathbf{W}^{\mathbf{T}}\mathbf{W})\mathbf{x} = \mathbf{J}^{\mathbf{T}}\mathbf{e}.$$

Since $\mathbf{W}$ is a diagonal matrix, $\mathbf{W}^{\mathbf{T}}\mathbf{W} = \mathbf{W}^2$, again a diagonal matrix. Thus the computational cost of stabilizing the solution is minimal as it simply involves adding small constants along the diagonal of $\mathbf{J}^{\mathbf{T}}\mathbf{J}$. For a more detailed explanation, see [Low89].

3.5 Implementation

The methods described in this chapter have been implemented in MOLASYS, written in Sun Common Lisp using the Sun Windows environment. The examples in the next section were obtained using a Sun 3/260. The interface is scaled automatically upon specification of the window size (minimum dimensions of 850x710 pixels) to fit the window. The images are read from either bitmap files or image files containing lists of point coordinates grouped by connectivity.

The menu is implemented as a control loop, started when the menu is first displayed and ended upon the user mousing the quit function. The control loop continually reads the mouse event queue, processing each moused command and responding with the appropriate function call.

Chapter 4

Results

4.1 Examples

An example of model creation using MOLASYS is displayed in the next several figures. A model of a chair is created using the rectangle and cylinder primitives to match an image of a chair. Photographs of a chair were taken from approximately the three orthogonal viewpoints plus one perspective viewpoint. The photographs were digitized and processed by a standard edge-detection algorithm. The resulting edge files were converted and displayed as bitmap files by MOLASYS.

Figure 4.1 shows the initial set-up for the modelling process. The rectangular primitive is displayed and has been translated and scaled using the menu functions to approximately the position and size of the seat of the chair. The views (cameras) of the four windows were also rotated slightly to better align the models with the images. The initial rotation matrices of the cameras project the object models to approximately the same orientation (from the current viewpoint) as the image objects, although the alignment of the views requires minor fine-tuning. As mentioned previously, the viewpoints are arbitrary and can be changed to any desired viewpoint. The initial rotation and translation matrices of the camera can be changed correspondingly by the user from the menu or in the source code.

Figure 4.2 displays constraints defined by the user to model the seat of the chair using the rectangular primitive. Several constraints have been specified in each window on the points visible from that viewpoint. Using many constraints results in better performance. In this example, all of the parameters have been left active for the constraint-satisfaction

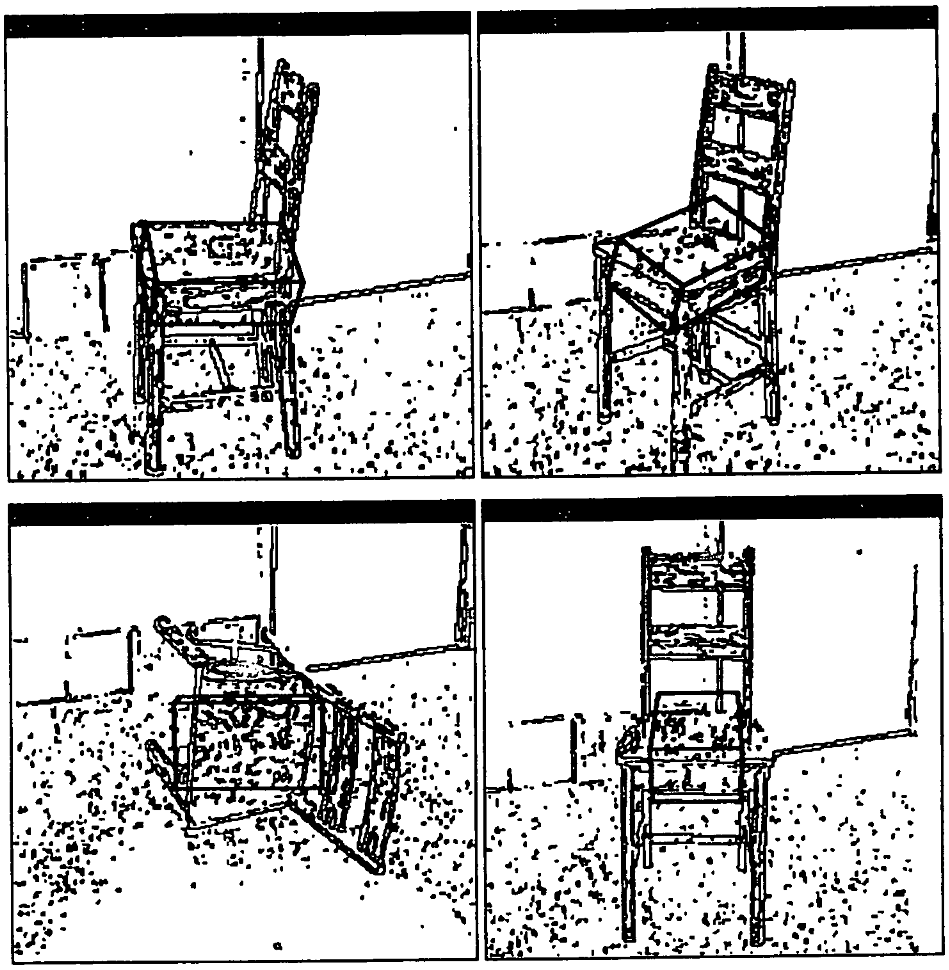


Figure 4.1: Initial configuration for the chair model

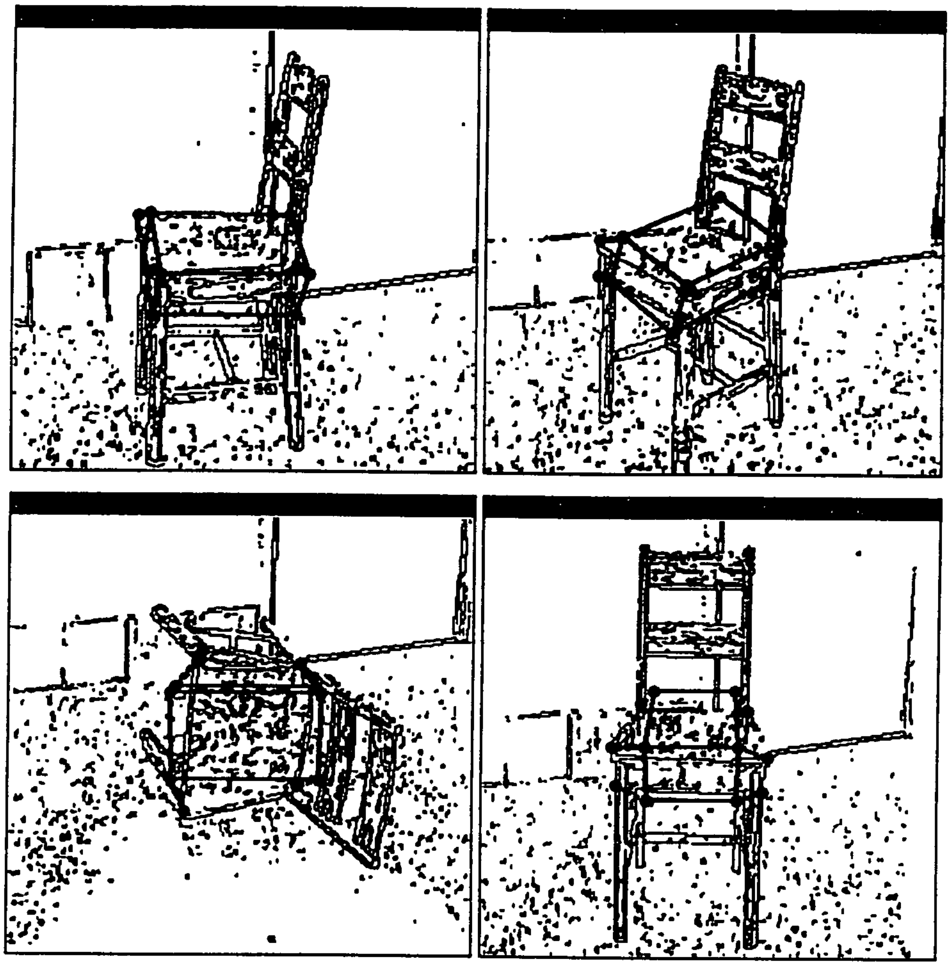


Figure 4.2: Specification of constraints to model the chair seat

process so the vector **x** potentially will contain the six degrees of freedom of each window, and the nine degrees of freedom for the model (translation and rotation about x, y, and z, and three object dimensions). Any irrelevant parameters, i.e., any parameters not affecting the constraints, are removed from further evaluation by MOLASYS. In this example, no parameters were removed, resulting in a system of 33 parameters and 50 constraints (each point-to-point constraint yields one constraint on each of u and v). Typically, only window translation and rotation parameters for windows containing no constraints are removed from **x** by MOLASYS. The user can remove any parameters that he/she would like to remain fixed at their current values.

Figure 4.3 displays the results of the constraint-satisfaction algorithm for the constraints displayed in figure 4.2. Seven iterations of Newton's method were needed in this case to satisfy the constraints. The fit is good for views 1, 2, and 4 and slightly off in view 3 as only four points of the seat were visible (i.e., could be constrained) from this viewpoint. The next step is to create the legs of the chair using cylinders. The cylinder primitives are transformed initially by the user, as was done for the rectangular primitive, and then constrained to the desired positions. They can also be joined to the seat, although in this case it isn't necessary as the overall model doesn't need to be rotated or translated, and the cylinders can be constrained into position next to the seat.

Figures 4.4 and 4.5 show the constraints and results of creating the first two legs of the chair, while figures 4.6, 4.7 and 4.8 display the last stages of constructing the model. For the two cylinders of figure 4.4, there were 18 parameters and 42 constraints. The parameters for the windows and the seat model were turned off from the menu to reduce the number of parameters for which to solve. Thus the system was only solving for the object parameters of the two cylinders, leaving the windows' translation and rotation and the seat model fixed. Also, the previously defined constraints were erased. Alternatively, all the constraints and parameters could have been kept active, and the system would then have solved for all the variables. However, while providing marginally better results, this approach would have also slowed down the iteration to a moderate degree. For the two chair leg models, eight iterations were required to converge to the solution displayed in figure 4.5. The three cylinders in figure 4.6 contained 27 parameters and generated

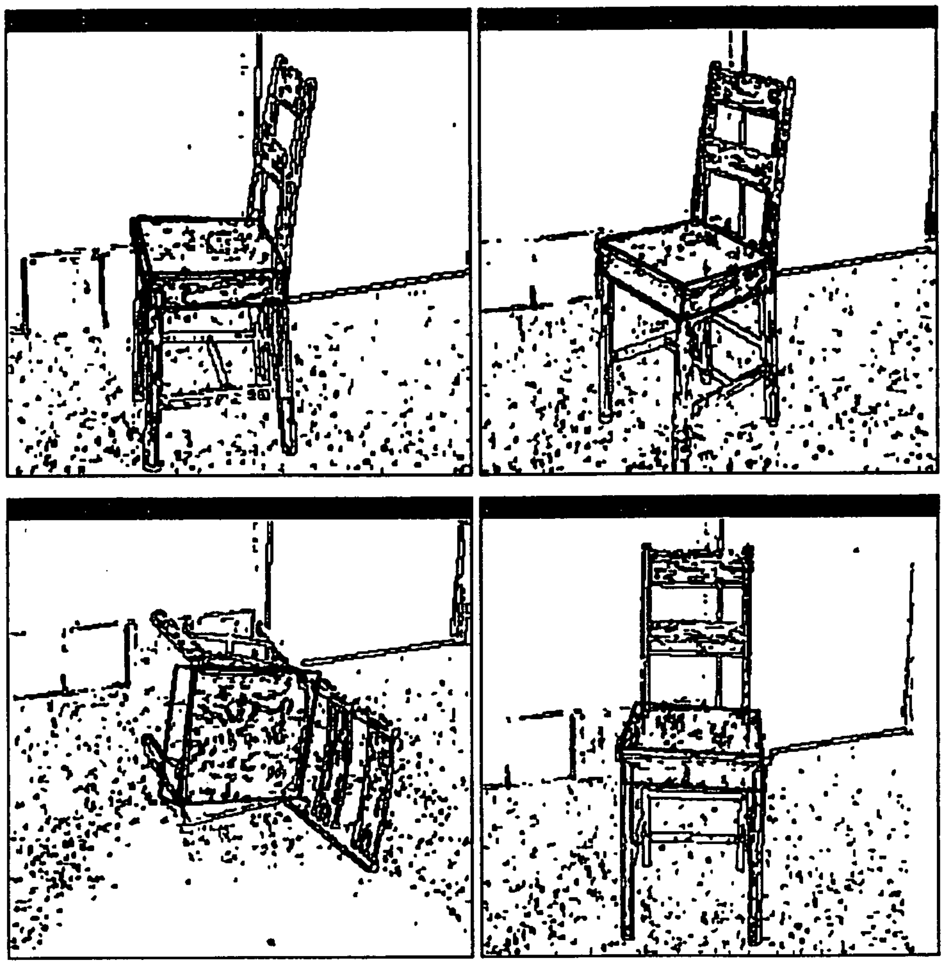


Figure 4.3: Results of constraint-satisfaction for chair seat

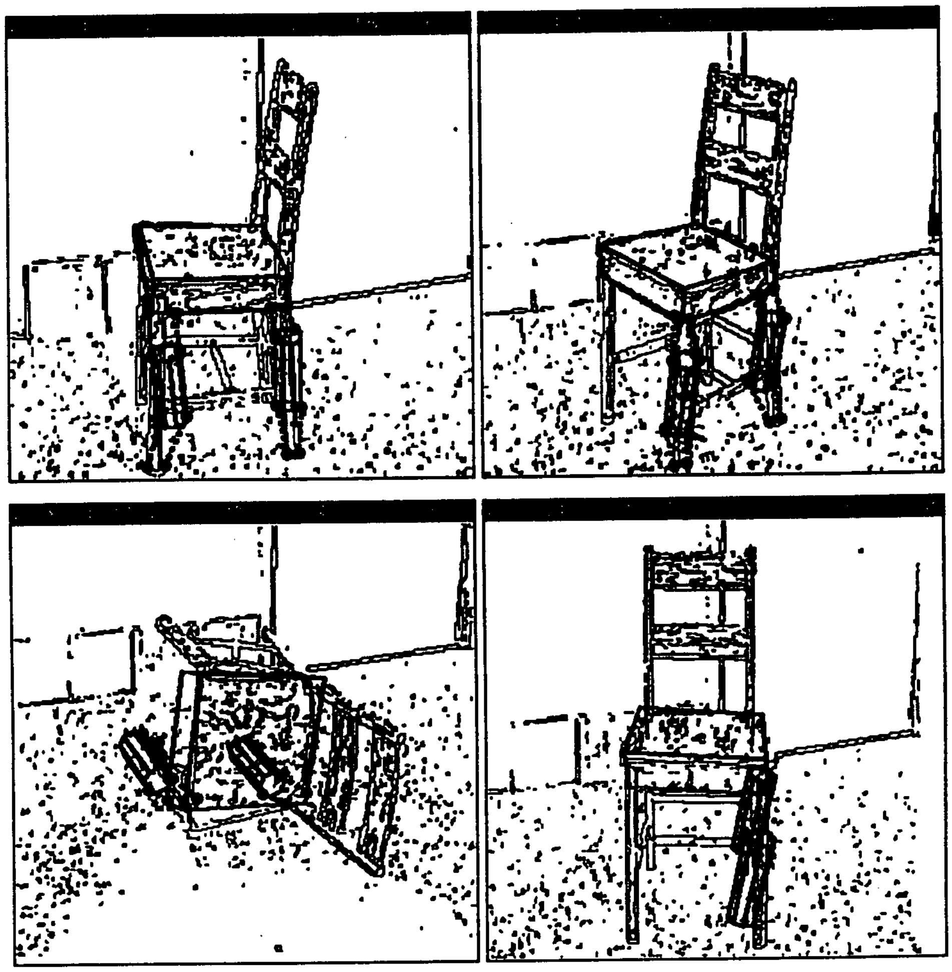


Figure 4.4: Constraints for the chair leg models

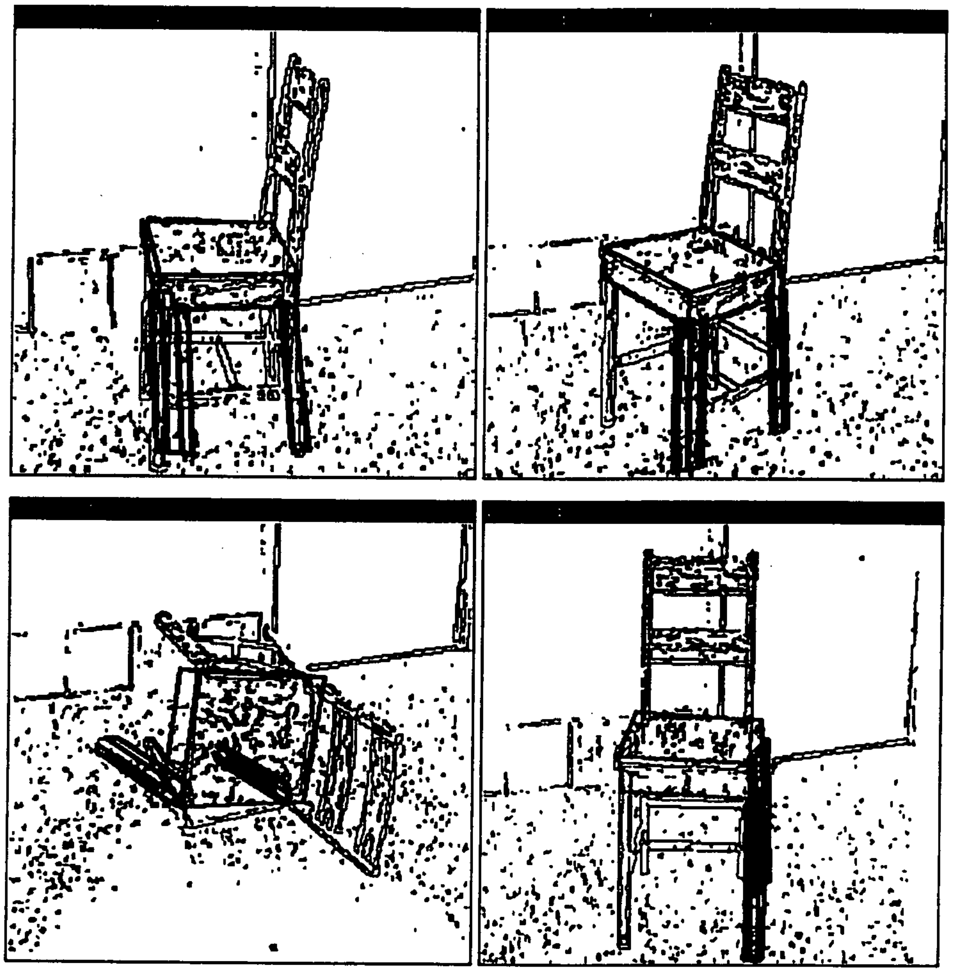


Figure 4.5: Results of constraint-satisfaction for chair legs

70 constraints. The system was stopped after eight iterations as it had not converged, although the results are reasonable. Minor modifications were made using the menu functions to improve the appearance of the models slightly.

The complete chair model is shown in figures 4.7 and 4.8. The fourth leg and two slats for the back of the chair were added using a cylinder primitive, two rectangular primitives, and the menu transformation functions. As can be seen in these two figures, the display is becoming cluttered, making it increasingly difficult to add more parts to the model. Specifying constraints is a problem as the primitives' points are on top of each other. Larger windows are needed, or perhaps a colour display, to improve the display of the model. In part for this reason, the overall time required to create this model was slow (approximately 50 minutes). This time would be reduced significantly with a better display as the constraints for more primitives could be specified in one step. Currently, it is a challenge to constrain more than two or three objects at a time as the windows become full.

Another problem illustrated by the chair model is the lack of a view for the back left portion of the chair. Orthogonal views were used as it was thought that the information contained in these views would be the most useful for creating the models. However, the perspective views (e.g., view 2) better show the three-dimensional structure of the model. Perhaps replacing view 3 with another perspective view similar to view 2 but from the other side of the object would be useful. The views used in any example will depend, at least in part, on the strucutre of the objects in the scene.

4.2 Performance

Performance of the algorithms has so far been good. The individual menu transformation functions produce results almost instantly, although the redisplay of the windows after each transformation slows down considerably for more object models. The constraint-satisfaction algorithm slows down as more constraints are added, but the delays have been relatively minor. The execution time increases reasonably slowly as the system grows larger. Each iteration of Newton's method requires at most 20 seconds in the examples tested to date (up to 33 parameters and 70 constraints) and the system usually has been

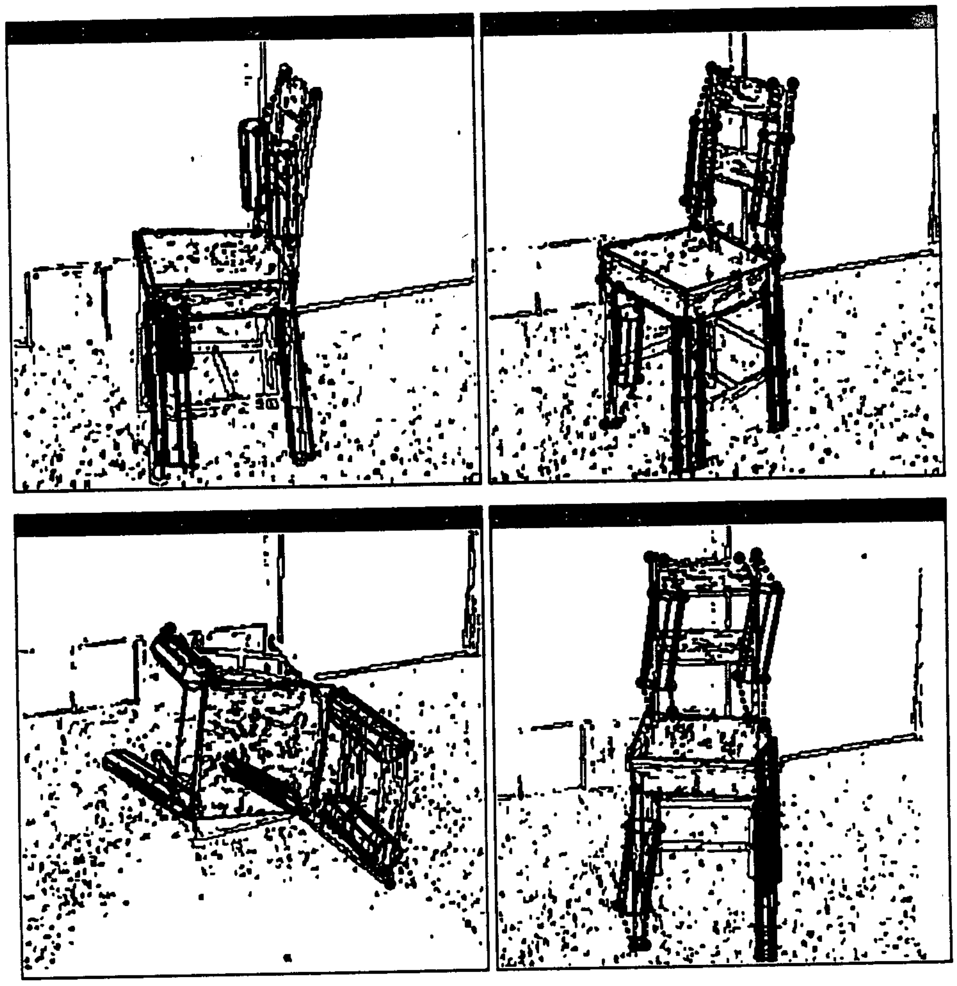


Figure 4.6: Constraints for the chair back and third leg models

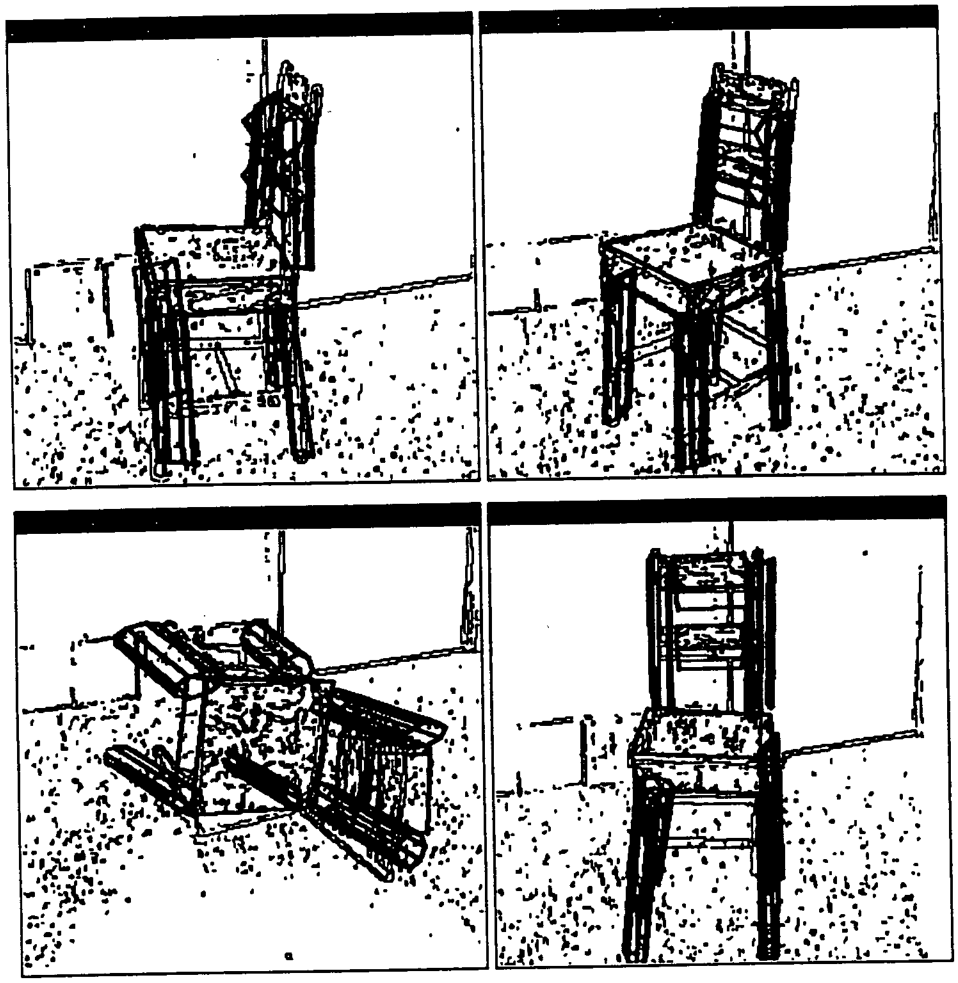


Figure 4.7: Chair model

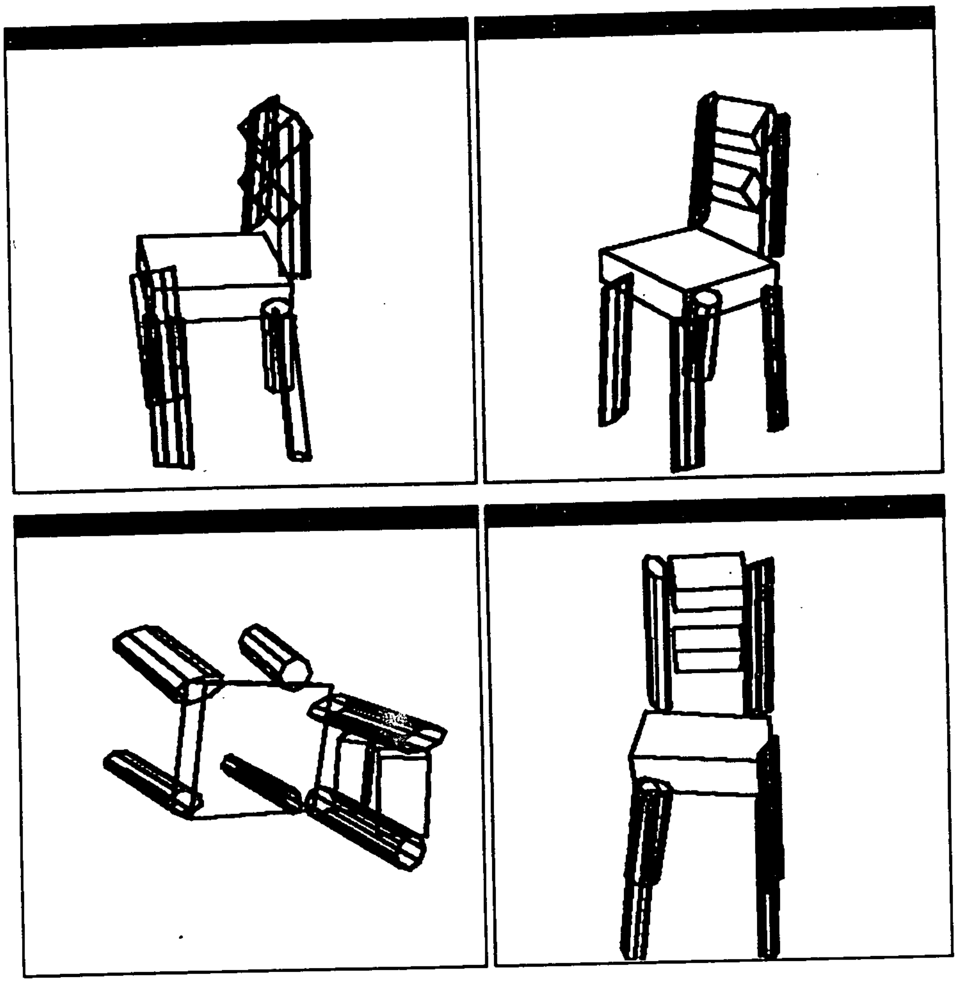


Figure 4.8: Chair model

solved or has converged in four to seven iterations. The simpler systems (i.e., fewer constraints in only 1-2 windows) converge quickly, giving accurate results. The larger systems, however, are occasionally slower to converge. There is a safety test after the eighth iteration to exit from the loop if no solution has been found at that point. The models are displayed using the best estimates obtained for the parameters and a suitable message is given to the user.

The interface is straightforward and easy-to-use, the trade-off being that a number of the steps involved can be tedious. In designing the interface, the author opted for the approach of simple and intuitive commands. More complicated functions can be more powerful (i.e. more flexible and requiring fewer steps) but they generally tend to increase the complexity of using the system. Creating the models using the constraints and multiple views implemented in MOLASYS was satisfactory, although more difficult than expected. This was due in part to the limitations of the primitives (i.e., in modelling various shapes gracefully).

Overall, MOLASYS provided a natural and useful tool for creating object models. The next chapter will discuss in more detail the problems in MOLASYS and future possible extensions to improve the system.

Chapter 5

Discussion

5.1 Goals

The aim of this work was to simplify the process of creating three-dimensional models by allowing the user to define constraints specifying the desired shape of the object and having the computer calculate how this would be best accomplished. The models are built on top of images of objects in four windows representing different views of the object and model. The constraints are defined interactively and specify relations that are to hold between parts of the model and points in the image objects. MOLASYS satisfies the constraints simultaneously using Newton's method to solve for a vector of corrections to the parameters to reduce the errors given by the constraints as much as possible.

There is currently no technique available to build object models from images, and MOLASYS was conceived to fill this need. By constraining the models to images, many tedious steps of aligning the points accurately are eliminated. The constraints for several primitives can be specified and solved at once, thus combining smaller sequential steps into larger and more efficient steps. The user specifies the changes that are to be made, and the computer determines how best to alter the parameters to complete these changes.

MOLASYS has provided an alternative approach to the process of model acquisition, enhancing the user's ability to create models efficiently by allowing the user to take direct advantage of the constraints present in the images. The constraint-satisfaction module presents a viable method for satisfying the constraints and producing 3D models.

5.2 Problems

New systems typically generate new problems, and MOLASYS is no different in this respect. There is a tendency for MOLASYS to alter the z translation parameter significantly (i.e., moving the models closer or further away) when the system can not be solved easily. This problem can be reduced by increasing the weight in $\mathbf{W}$ corresponding to that parameter.

Designing the interface to be natural and easy-to-use was difficult at times. The functions provided are straightforward and broken up into logical steps, resulting in good flexibility. The response rate, in most cases, is prompt although redisplaying the views becomes slow as more models are added.

There were also a few problems with the constraint-satisfaction process. Overconstrained systems generally yielded reasonable solutions (in good time), however the results for underconstrained systems were less consistent. This problem is still under investigation in an attempt to improve the performance. It was also noticed that larger systems (containing more constraints and parameters) often converge more slowly, requiring many iterations. The results are better if the initial estimates of the models are good (i.e., the objects are initially positioned near the image object before specifying the constraints).

5.3 Extensions

The most obvious improvement to be made is the addition of more graphics primitives, preferably with a better approximation of curved surfaces. Perhaps some form of elastic or deformable surfaces could be used, as several systems mentioned in chapter 2 implement deformable surfaces using constraints. More flexible objects could be created from the current representation by adding more parameters to the models. The cylinder primitive currently has two different variables for the radius of each end, allowing for a tapering effect in its shape. The same could be done for the rectangular primitive to create less regular models. In addition, spheres could be created with the surface points defined using the distances from the centre of the sphere. This would require a separate radius parameter (initially all equal for a sphere) for each point. The models could then be stretched simply by altering the affected parameters.

The problem of clutter in the views mentioned in the previous chapter could be alleviated by the implementation of a *blow-up* function. This function could allow one view at a time to be enlarged to fill the display, allowing for much better resolution for specifying the constraints. Alternatively, this function could enlarge only a user-specified region of a view, filling either that view or a pop-up window temporarily placed on top of the current view(s).

Functions can be easily added to MOLASYS as it is implemented in modules (e.g., translation, rotation, scaling, constraint-satisfaction). Currently, translation of the models from the menu is provided but only in two dimensions (x, y) for each view. A *zoom* function would be useful as it would allow translation along the z-axis in each view, permitting the user to move the object further forward or further back.

Also, a facility to allow for back-tracking would be useful. Currently, there is no way to undo the execution of a set of constraints. If the system wasn't solved completely, the best-fit (or parameter estimates obtained after the maximum number of iterations allowed) may not be what was desired. Another useful tool would be to allow individual constraints to be deleted. Once this is implemented, it could also be possible to add weights to the constraints as a measure of their importance (as suggested by Badler [Bad86]). This could be helpful for systems of constraints that can't be satisfied completely (i.e., the system will converge to a best-fit).

More work also needs to be done on the convergence algorithm to ensure more reliable convergence in under and overconstrained systems. Currently, the performance is good although some cases diverge (e.g., continually translating along z), producing less than desirable results. Also, for larger systems requiring more iterations to converge, it should be possible to modify the algorithm to produce faster convergence. Experimenting with different weights, $\mathbf{W}$, or perhaps a better method for approximating the Jacobian matrix are potentially valid options.

Another area to explore is that of symbolic constraints, i.e., specifying a constraint on a parameter with respect to another parameter. For example, a symbolic constraint could specify that a model dimension be twice that of another. These constraints could be implemented in a similar manner to that for the *joins* specifying joined models. As the numerical constraints are being solved, any change to one of the parameters in a symbolic constraint would be applied to the other parameter as well. In this way, the numerical constraints would be solved while maintaining the symbolic constraints.

Another interesting feature would be the automatic programming by Pentland [Pen90] discussed briefly in chapter 2. The system would be presented with an image and would create a model using data measured from the image, requiring minimal assistance from the user. The constraints would be generated automatically between the model(s) and the measured data, and then solved. The system would have to be able to solve for different views of an object simultaneously in order to adequately represent the three-dimensional structure in the model.

The only perfect science is hind-sight.

- Murphy's Laws on Technology

Bibliography

[Amb75] A.P. Ambler and R.J. Popplestone, "Inferring the Positions of Bodies from Specified Spatial Relationships", Artificial Intelligence, vol. 6, pp. 157-174, 1975.

[Bad86] N.I. Badler, K.H. Manoochehri, and D. Baraff, "Multi-Dimensional Input Techniques and Articulated Figure Positioning By Multiple Constraints", Proceedings of the 1986 Workshop on Interactive 3D Graphics (Chapel Hill, NC, October 23-24, 1986), ACM SIGGRAPH, pp. 151-169, 1987.

[Bar84] B.A. Barsky, "A Description and Evaluation of Various 3-D Models", IEEE Computer Graphics and Applications, vol. 4, no. 1, pp. 38-48, January 1984.

[Bar88] R. Barzel and A.H. Barr, "A Modeling System Based On Dynamic Constraints", SIGGRAPH 1988 Conference Proceedings (Atlanta, Ga., August 1-5 1988), ACM SIGGRAPH, Computer Graphics, vol. 22, no. 4, pp. 179-188, 1988.

[Bha87] B. Bhanu and C. Ho, "CAD-Based 3D Object Representation for Robot Vision", IEEE Computer, pp. 19-35, August 1987.

[Bie86a] E.A. Bier and M.C. Stone, "Snap-Dragging", SIGGRAPH Proceedings 1986, pp. 233-240.

[Bie86b] E.A. Bier, "Skitters and Jacks: Interactive 3D Positioning Tools", Proceedings of the 1986 Workshop on Interactive 3D Graphics (Chapel Hill, NC, October 23-24, 1986), ACM SIGGRAPH, pp.183-196, 1987.

[Bie88] E.A. Bier, "Snap-Dragging: Interactive Geometric Design in Two and Three Dimensions", Report No. UCB/CSD 88/416, April 28, 1988, Computer Science Division, Dept. of EECS, UC Berkeley, CA.

[Bie90] E.A. Bier, "Snap-Dragging In Three Dimensions", ACM SIGGRAPH, Computer Graphics, vol. 24, no. 2, pp. 193-204, March 1990.

[Bor81] A. Borning, "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory", ACM Transactions on Programming Languages and Systems, vol. 3, no. 4, pp. 353-386, 1981.

[Bor86] A. Borning and R. Duisberg, "Constraint-Based Tools for Building User Interfaces", ACM Transactions on Graphics, vol. 5, no. 4, pp. 345-374, 1986.

[Boy82] J.W. Boyse and J.E. Gilchrist, "GMSolid: Interactive Modeling for Design and Analysis of Solids", IEEE Computer Graphics and Applications, pp. 27-40, March 1982.

[Bro81] R.A. Brooks, "Symbolic Reasoning Among 3-D Models and 2-D Images", Department of Computer Science, Stanford University, STAN-CS-81-861, June 1981.

[Brü86] B. Brüderlin, "Constructing Three-Dimensional Geometric Objects Defined By Constraints", Proceedings of the 1986 Workshop on Interactive 3D Graphics, (Chapel Hill, NC, October 23-24, 1986), ACM SIGGRAPH, pp.111-129, 1987.

[Chen88] M. Chen, S.J. Mountford, and A. Sellen, "A Study In Interactive 3-D Rotation Using 2-D Control Devices", SIGGRAPH 1988 Conference Proceedings (Atlanta, Ga., August 1-5 1988), ACM SIGGRAPH, Computer Graphics, vol. 22, no. 4, pp. 121-129, 1988.

[Dil87] A. Dillon, "Knowledge Acquisition and Conceptual Models: a Cognitive Analysis of the Interface", In D. Diaper and R. Winder, editors, People and Computers III, Proceedings of the 3rd Conference of the British Computer Society Human-Computer Interaction Specialist Group, University of Exeter, pp. 371-379, Sept. 7-11, 1987.

[Edw88] L. Edwards, W. Kessler, and L. Leifer, "The Cutplane: A Tool for Interactive Solid Modeling", SIGCHI Bulletin, vol. 20, no. 2, pp. 72-77, October 1988.

[Fol82] J.D. Foley and A. Van Dam, Fundamentals of Interactive Computer Graphics, Reading, Mass.: Addison-Wesley, 1982.

[Fol84] J.D. Foley, V.L. Wallace, and P. Chan, "The Human Factors of Computer Graphics Interaction Techniques", IEEE Computer Graphics and Applications, pp. 13-48, Nov. 1984.

[For86] A.R. Forrest, "User Interfaces for Three-Dimensional Geometric Modelling", Proceedings of the 1986 Workshop on Interactive 3D Graphics (Chapel Hill, NC, October 23-24, 1986), ACM SIGGRAPH, pp.237-249, 1987.

[Hea86] D. Hearn and M.P. Baker, Computer Graphics, Englewood Cliffs, N.J.: Prentice-Hall, 1986.

[Hil87] R.D. Hill, "Adaptive 2-D Rotation Control", ACM Transactions on Graphics, vol. 6, no. 2, pp. 159-161, April 1987.

[Klo86] F. Klok, "Two Moving Coordinate Frames for Sweeping Along a 3D Trajectory", Computer Aided Geometric Design 3, pp. 217-229, 1986.

[Kun85] T.L. Kunii and G. Wyvill, "A Simple But Systematic CSG System", Proceedings of Graphics Interface 1985, Canadian Man-Computer Communications Society, pp.329-336, 1985.

[Lai86] D.H. Laidlaw and J.F. Hughes, "Constructive Solid Geometry for Polyhedral Objects", SIGGRAPH 1986 Conference Proceedings (Dallas, Texas, August 18-22 1986), ACM SIGGRAPH, Computer Graphics, vol. 20, no. 4, pp. 161-170, 1986.

[Low89] D.G. Lowe, "Fitting Parametrized 3-D Models to Images", Department of Computer Science, University of British Columbia, 1989.

[Mir89] C. Mirolo and E. Pagello, "A Solid Modeling System for Robot Action Planning", IEEE Computer Graphics and Applications, vol. 9, no. 1, pp. 55-69, January 1989.

[Mor87] A. Morris, "Expert Systems - Interface Insight", In D. Diaper and R. Winder, editors, People and Computers III, Proceedings of the 3rd Conference of the British Computer Society Human-Computer Interaction Specialist Group, University of Exeter, pp. 307-324, Sept. 7-11, 1987.

[Nel85] G. Nelson, "Juno, A Constraint-Based Graphics System", SIGGRAPH 1985 Conference Proceedings (San Francisco, Ca., July 22-26 1985), ACM SIGGRAPH, Computer Graphics, vol. 19, no. 3, pp. 235-243, 1985.

[Ost86] E. Ostby, "Describing Free-Form 3D Surfaces for Animation", Proceedings of the 1986 Workshop on Interactive 3D Graphics (Chapel Hill, NC, October 23-24, 1986), ACM SIGGRAPH, pp. 251-258, 1987.

[Pen86] A.P. Pentland, "Recognition By Parts", Technical note no. 406, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, 1986.

[Pen87] A.P. Pentland, "Toward an Ideal 3-D CAD System", SPIE Conference on Machine Vision and the Man-Machine Interface, (Los Angeles, Ca., January 9-16 1987).

[Pen90] A. Pentland, I. Essa, M. Friedmann, B. Horowitz, and S. Sclaroff, "The Thing-World Modeling System: Virtual Sculpting by Modal Forces", ACM SIGGRAPH, Computer Graphics, vol. 24, no. 2, pp. 143-144, March 1990.

[Phi90] C.B. Phillips, J. Zhao, and N.I. Badler, "Interactive Real-Time Articulated Figure Manipulation Using Multiple Kinematic Constraints", ACM SIGGRAPH, Computer Graphics, vol. 24, no. 2, pp. 245-250, March 1990.

[Pla88] J.C. Platt and A.H. Barr, "Constraint Methods for Flexible Models", SIGGRAPH 1988 Conference Proceedings (Atlanta, Ga., August 1-5 1988), ACM SIGGRAPH, Computer Graphics, vol. 22, no. 4, pp. 279-288, 1988.

[Pru86] P. Prusinkiewicz and D. Streibel, "Constraint-Based Modeling of Three-Dimensional Shapes", Graphics Interface/Vision Interface 1986 Conference Proceedings (Vancouver, B.C., May 26-30 1986), Canadian Man-Computer Communications Society, pp. 158-163, 1986.

[Roa87] J.W. Roach, P.K. Paripati, and J.S. Wright, "A CAD System Based on Spherical Dual Representations", IEEE Computer, pp. 37-44, August 1981.

[Ros86] J.R. Rossignac, "Constraints in Constructive Solid Geometry", Proceedings of the 1986 Workshop on Interactive 3D Graphics (Chapel Hill, NC, October 23-24, 1986), ACM SIGGRAPH, pp.93-110, 1987.

[Sed86] T.W. Sederberg and S.R. Parry, "Free-Form Deformation of Solid Geometric Models", SIGGRAPH 1986 Conference Proceedings (Dallas, Texas, August 18-22 1986), ACM SIGGRAPH, Computer Graphics, vol. 20, no. 4, pp. 151-160, 1986.

[Sis88] J.C. Siska, R. Espinosa, D. Trevino, J. Rodriguez, J. Sanchez, and H. Ramirez, "3D Solid Modeling Software Development for Industrial and Academic Purposes", Computers and Graphics, vol. 12, nos 3,4, pp.381-389, 1988.

[Sut63] I. Sutherland, "Sketchpad, A Man-Machine Graphical Communication System", Ph.D. dissertation, MIT, MIT Lincoln Laboratory Technical Report No. 296, Lexington, MA, 1963.

[Ter87] D. Terzopoulos, A. Witkin, and M. Kass, "Symmetry-Seeking Models and 3D Object Reconstruction", International Journal of Computer Vision, vol.1, no. 3, pp. 211-221, October 1987.

[Ter88] D. Terzopoulos and K. Fleischer, "Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture", ACM SIGGRAPH, Computer Graphics, vol. 22, no. 4, pp. 269-278, August 1988.

[Thi90] J.A. Thingvold and E. Cohen, "Physical Modeling with B-spline Surfaces for Interactive Design and Animation", ACM SIGGRAPH, Computer Graphics, vol. 24, no. 2, pp. 129-137, Mar 1990.

[Van82] C.J. Van Wyk, "A High-Level Language For Specifying Pictures", ACM Transactions on Graphics, vol.1, no. 2, pp. 163-182, April 1982.