SANDIA REPORT
SAND98-2684
Unlimited Release
Printed December 1998
Second Printing November 2001

Content-Based Search on a Database of Geometric Models

Identifying Objects of Similar Shape

Patrick G. Xavier, Tyson R. Henry, Robert A. LaFarge, Lilita Meirans,
and Lawrence P. Ray

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/ordering.htm

SAND98-2684
Unlimited Release
Printed December 1998
Second Printing November 2001

The only change to this SAND Report is the distribution limitation, which has been changed from Patent Caution to Unlimited Release.

Content-Based Search on a Database of Geometric Models

Identifying Objects of Similar Shape

Patrick G. Xavier, Tyson R. Henry, and Robert A. LaFarge
Intelligent Systems Principles Department

Lilita Meirans and Lawrence P. Ray
Applied Systems I Department

Intelligent Systems and Robotics Center
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1008

Abstract

The Geometric Search Engine is a software system for storing and searching a database of geometric models. The database may be searched for modeled objects similar in shape to a target model supplied by the user. The database models are generally from CAD models while the target model may be either a CAD model or a model generated from range data collected from a physical object. This document describes key generation, database layout, and search of the database.

Intentionally Left Blank.

Contents

Figures

Intentionally Left Blank.

Content-Based Search on a Database of Geometric Models

Identifying Objects of Similar Shape

Introduction

This work is inspired by the use of search engines to locate text documents of interest by finding specified words. But not all information is expressed in words. The work described in this report expands the range of search queries by developing a system which can search a database of geometric models for objects having a similar shape to a specified object. We call this system the Geometric Search Engine (GSE).

One possible use for the GSE is querying CAD databases. With the advent of powerful CAD software, most manufacturing companies use this software to design and model their parts. While CAD software has greatly improved the design process, it has not addressed the issue of finding previous designs and inventory. Ideally, once a design engineer has a basic design, one should be able to determine whether the part has been already designed or manufactured. The problem is that determining the existence of a previous design usually involves the tedious activity of looking through text records kept in computer files or paper records. For a large company, nomenclature may vary slightly from location to location, complicating the process even if an electronic database of part descriptions is kept. For large international companies, even the language used in the textual description may vary from location to location. The ideal CAD database should be able to search on geometry rather than text. With such a search capability, a designer can avoid the overwhelming response that a text search on "fastener" might yield.

The "eigenfaces" approach to face recognition (Turk and Pentland 1991) is a key influence on the Geometric Search Engine, although the algorithms are very different. In contrast to "feature recognition" techniques, this approach adopted a "training" paradigm in which key vectors were computed based on global properties for a set of training images. Faces were represented as weighted sums of a set of "eigenfaces," computed as the eigenvectors of the covariance matrix for a set of face images. Recognition was performed by comparing the weights required for reconstruction of a face image with weights for a known set of faces.

Murase and Nayar (1995) extended the approach to object recognition, training on images of objects in various orientations. Lighting variation was accommodated with multiple training images under different lighting. This work was extended to implement a color-based object recognition system which automatically sampled training images for objects placed on an automated turntable (Nayar et. al. 1996). Krumm (1997) used binary feature vectors as keys instead of eigenvector principal components; by this means, he was able to recognize partially occluded objects. These methods all performed recognition from 2D optical images; recognition was based on similar appearance.

The primary objectives of the Geometric Search Engine are different: to recognize objects by shape rather than appearance and to identify similar objects rather than just to recognize an object. The GSE shows that the general approach of computing key vectors from multiple training images can successfully achieve these objectives using 3D range data. The range data can be computed from 3D models using computer graphics algorithms or measured from physical objects using 3D sensors. A new key vector algorithm is developed for the 3D data. Rather than computing eigenvectors and principal components for the entire image set, we compute a key vector from a simulated grasp of the object. Unlike the eigenfaces method, this process is not invertible, but it allows an object to be added to the database without recomputing keys for the entire set of objects. (However, we do have an option to apply principal component analysis for key compression. Optimal key compression requires recomputation for the entire set when objects are added.)

An alternative approach to 3D shape similarity is presented by Zhang and Hebert (1997). Based on 2D multi-resolution approaches, they compute a similarity measure between an object and various template objects at multiple scales. This allows classification of objects into similar classes, but use of the method for recognition or identifying similar objects from a database has not been investigated. The method requires a complete description of an object's surface, while the GSE requires only data taken from a single view; for some applications, a complete model will not be available.

Overview

The goal of searching for objects of similar shape immediately raises two questions. What does it mean for the shape of two objects to be similar? How do we accommodate the fact that the appearance of most objects varies with the position of the viewer?

The Geometric Search Engine considers only the range (distance from the viewer) of points on an object's surface *as seen from a particular viewpoint*. Portions of the object not visible from that viewpoint are ignored. We use the term *object-viewpoint* to refer to an object seen from a specific viewpoint. Viewing positions are normalized in various ways to reduce the number of degrees of freedom for a viewpoint to two angles, termed *azimuth* and *elevation*.

The GSE compares two object-viewpoints by calculating the Euclidean distance between the *keys* calculated for each object-viewpoint. A key is a vector of real numbers describing an object as seen from a particular viewpoint. The use of keys is our answer to the question "What does it mean for the shape of two objects to be similar?" It means that the distance between their keys is small for at least one pair of viewpoints. Note that the principle could easily be modified to compare objects from multiple viewpoints for each object in order to search for objects that have a similar shape as seen from all directions.

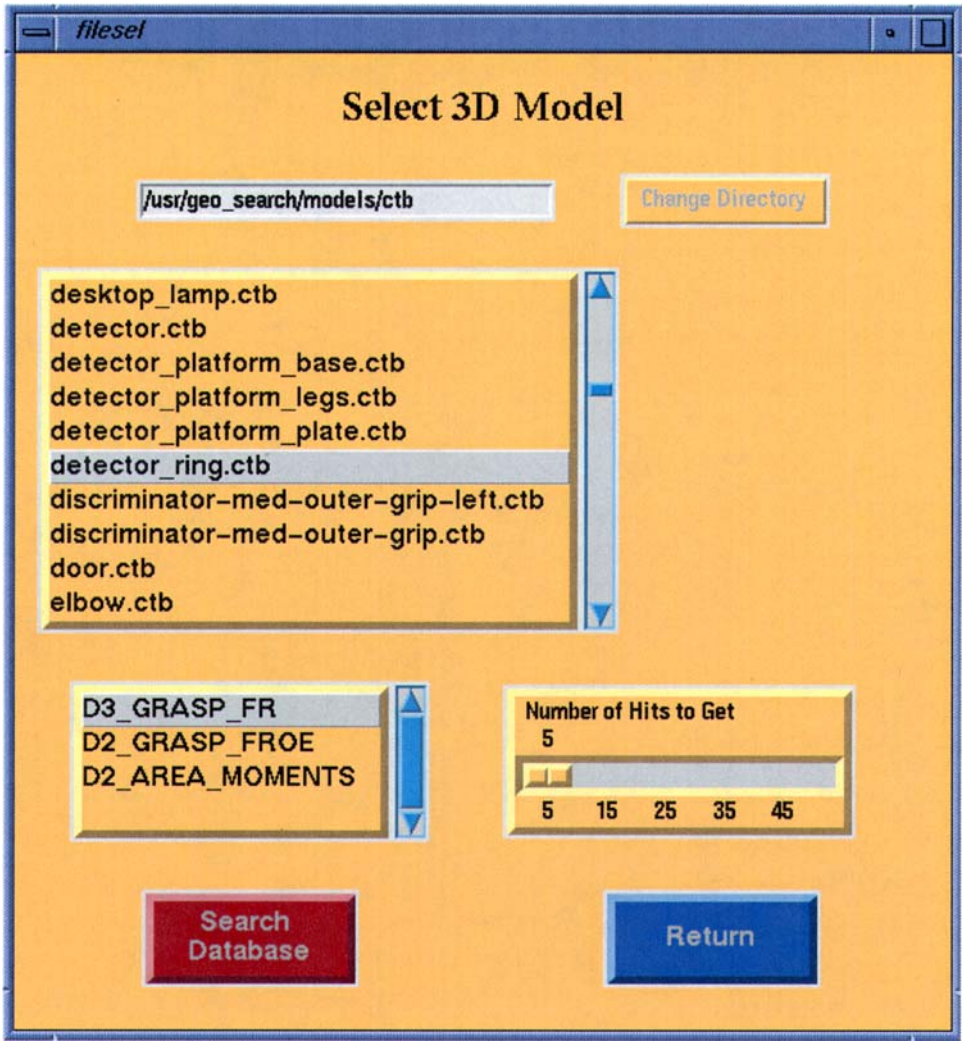
Object shape variation with viewer position has been accommodated by modeling each object with many object-viewpoints. A database of keys is prepared by computing and storing key vectors for objects represented in the database. For each object, the key vector for each of many viewpoints is included in the database. In the tests described in the Performance section, each object was represented for 1332 viewpoints. See the Key Computation section for more detail.

The database keys are computed from three-dimensional geometric object models, rendered by computer graphics techniques in the form of a depth map, or Z-buffer. At present, the models are polygonal surface representations, but the method is readily adaptable to other three-dimensional representations.

In operation, the Geometric Search Engine allows a user to interactively search the database, which has been previously prepared. Although the key computation and database search techniques could be applied using any of a variety of interfaces, we describe the graphical user interface used for development work as an example of one form of usage.

The user selects the geometric model for the *target* object (see Figure 1). The target is the basis for the search; objects similar to the target will be returned as *match* objects. The

geometric model for the object may come from CAD data, or a polygonal model constructed from 3D range data. In this work, we tested the GSE with both CAD models and range data models.[1] The user may also select the number of match objects to return (called "hits" in the user interface). There is also a control for selection of the key computation algorithm. An image of the model is displayed on the computer screen in a second window (see Figure 2). The user rotates the model to present a characteristic view; it is this view which the search will attempt to match.[2]



[1] The authors thank Charles Little and Dan Small of the Intelligent Systems and Robotics Center at Sandia National Laboratories for applying results of their structured-lighting and model-building research to collect range data and build polygonal models of several components for use in this project.

[2] The authors thank Peter Watterberg of the Intelligent Systems and Robotics Center at Sandia National Laboratories for providing and customizing the software package used to display polygonal objects, including interactive viewpoint and lighting adjustment.

Figure 1. User Interface for Selection of the Target Object.

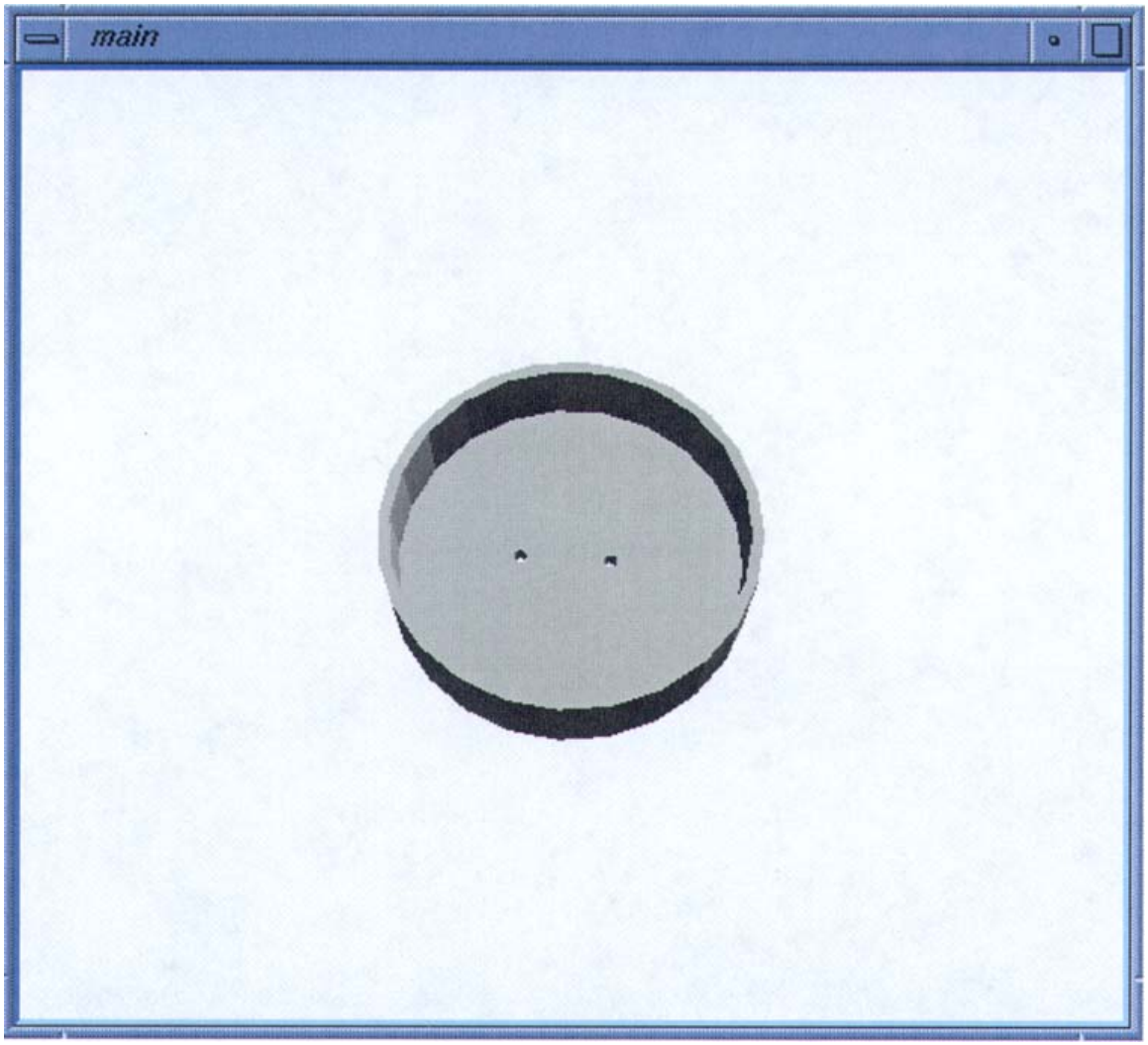


Figure 2. Target Object of Search.

The user then initiates the search. The key value for the selected object and viewpoint is computed, and the database is searched for keys having similar values. The object-viewpoints having the most similar key values are returned as matches. Only one viewpoint is returned for each object – the viewpoint with the most similar key. Also returned from the database search is the path and name for the object's geometric model file.

A list of matching objects is displayed (see Figure 3), and the user selects which of these objects to view (see Figure 4). The distance between keys for the selected object and the target object is displayed. The selected object is displayed on screen, initially as seen from the best-match viewpoint; the user may rotate the object to view it from other angles and understand the overall 3D shape.

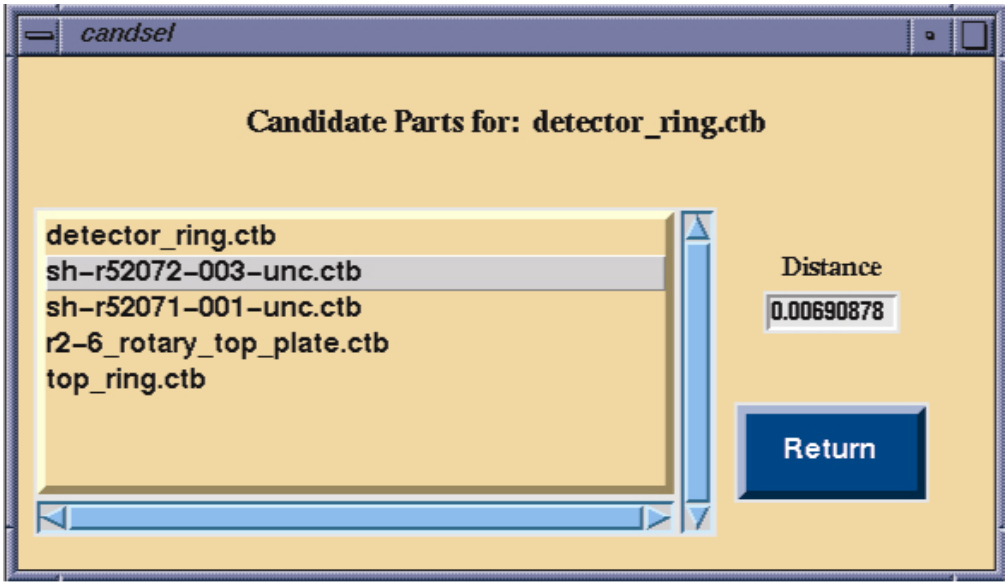


Figure 3. List of Match Objects.

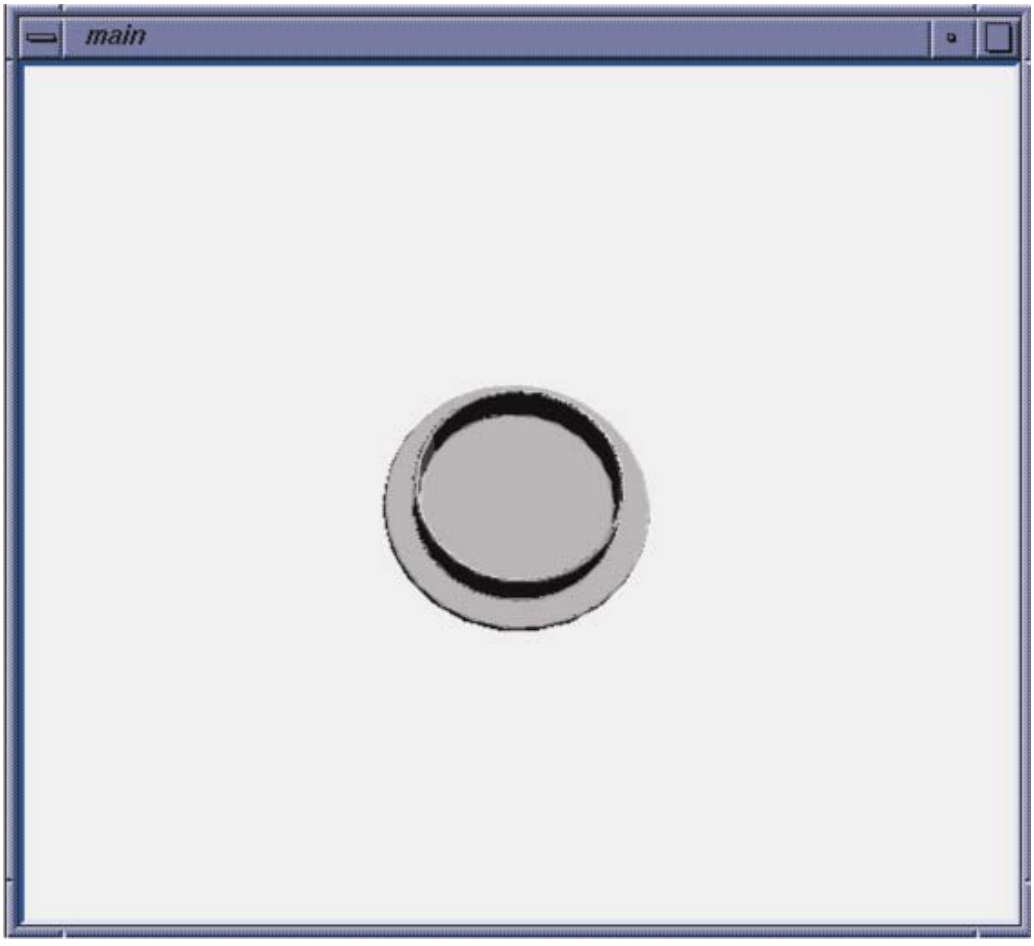


Figure 4. One Match Object Returned by Query.

Key Computation

In our scheme for searching for objects by overall shape, an index, or database, contains the union of sets of key vectors, with each key vector corresponding to a particular object-viewpoint. We now describe several techniques we used to generate sets of key vectors for a given object. We ignore object scale (size), choosing to concentrate instead on shape.

The first subsection, Baseline, describes the fundamental key vector computation for the experiments described here. The second subsection, Compression with Principal Components, describes the way in which the baseline key vectors were compressed for improved performance. And the third subsection, Feature Recognition, describes exploratory work toward a richer set of information for key vectors including machined features of objects.

Baseline Keys

In this section, we describe the basic techniques used to construct key vectors. We first outline a general framework that we used for both 2D keys and 3D keys. 2D keys are constructed using the depth map only to indicate presence or absence of an object; the same information could be generated from 2D optical images. 3D keys are constructed using the full range information in the depth map. We then provide the details of the methods we used to generate keys for an index to be used for queries based on 2D information. Finally, we describe how we generalized some of these methods to construct keys to be used for queries based on 3D information.

Outline of Process

For each object to be represented in an index, we construct a set of keys that correspond to that object viewed from a sample set of viewing directions. Each key is a vector, or *key vector*, of real numbers that are the *key elements*. These key elements are computed from a 3D geometric model of the object. At query time, a *query key vector*, or more simply a *query key*, is constructed from the query object-viewpoint geometry and matched against the keys (vectors) in the index.

This part of our approach is shared by many researchers in the image indexing community (Turk and Pentland 1991; Murase and Nayar 1995; Nayar et. al. 1996; Krumm 1997). In the general theory, the set of all keys corresponding to an object will determine a 2-dimensional *key surface* embedded in the space of all possible keys. To find the object best matching a query is then equivalent to finding the surface that comes closest to the query key. However, in practice, one searches the index for the key

vector that most closely matches the query key. Clearly, the more densely the set of viewing directions is sampled in constructing the key set for an object, the more likely one of the object's keys will be closest to a query key when the object's key surface is the key surface closest to the query key.

Recall that in our framework for general shape recognition, 2D query data are basically images, and 3D query data are faceted range data. To compute keys from query data, we adjust the viewing parameters that determine how the data appears when rendered with 3D computer graphics algorithms, render the data, and then perform computations on the bitmaps making up the depth map (Z-buffer) for the rendered image. When generating keys for a given object for insertion into the index, for each viewing direction θ, we similarly adjust the viewing parameters, render the object, and perform computations on the associated pixel arrays. In the tests described here, we used the OpenGL programming facilities on an SGI Octane, taking advantage of the high-speed graphics hardware to speed key computation.

Specifically, given an object and a viewing direction θ, or given query data, we adjust the viewing parameters to *normalize* the view. This means that when the data is drawn, it will all fit in the window, and that sets of view data with identically-shaped silhouettes will be rendered so that their silhouettes align identically.

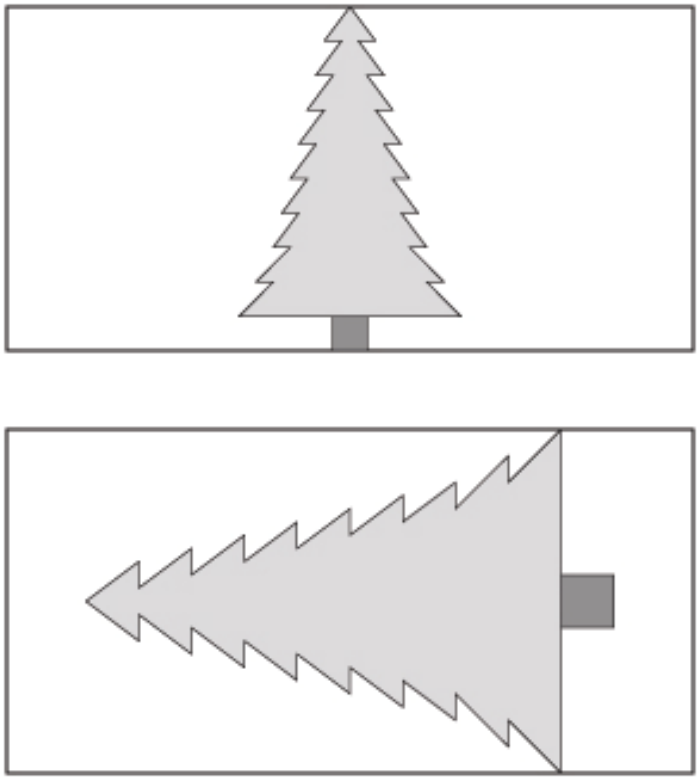


Figure 5. Normalization Process. Initial View Above, Normalized View Below.

To normalize the view, we first compute the minimal axes-aligned bounding box around the geometric data and set the orthographic projection parameters (i.e., use **`glOrtho(..)`**) to contain the bounding box. We then draw the data using orthographic projection in a bitmap of a standard aspect ratio, which we have chosen to be 2:1 in width:height. The **`zNear`** and **`zFar`** projection parameters must match the z-axis parameters of the bounding box. Next, we compute the (2D) principal axes of the silhouette of the drawn data; the silhouette pixels are those with non-background color. We then rotate the data about the viewing axis so that the major principal axis of the silhouette aligns with the window's X-axis, compute a new axes-aligned bounding box around the data, and adjust the orthographic viewing parameters so the bounding box fits maximally in the drawing window (see Figure 5).

After we clear the window and re-draw the (rotated) data, we are ready to apply a key computation algorithm to the rendered data.

Key Computation for Indices for 2D Queries

We now describe several ways to compute key elements from 2D geometry data. We assume the data has been rendered with normalizing viewing parameters (i.e., rotation, projection) into a standard-sized bitmap.

Starting with the bounding box, the easiest key element to compute is the aspect ratio of the bounding box of the image. Using a pixel array representation of the silhouette, we also compute the fill-ratio, the ratio of silhouette pixels to total pixels in the 2D bounding box. Informal experiments showed that these two measures could be surprisingly useful in determining the pose of a single object.

Another method for generating key elements was inspired by soft-fingered squeeze-grasping, earning it the name *virtual soft-fingered grasp measurement*. In a simple form of this method, we divide the bounding box of the image into a small number of vertical bands each approximately the same number of pixels wide (see Figure 6). In each band, we compute two measurements: (a) the average fraction of box height from the top of the box to the silhouette pixel nearest the top in each pixel column, and (b) the average fraction of the box height from bottom of box to the silhouette pixel nearest the bottom in each pixel column. This procedure yields twice as many key elements as vertical bands. We chose to use five bands *a priori*, thus obtaining ten key elements.

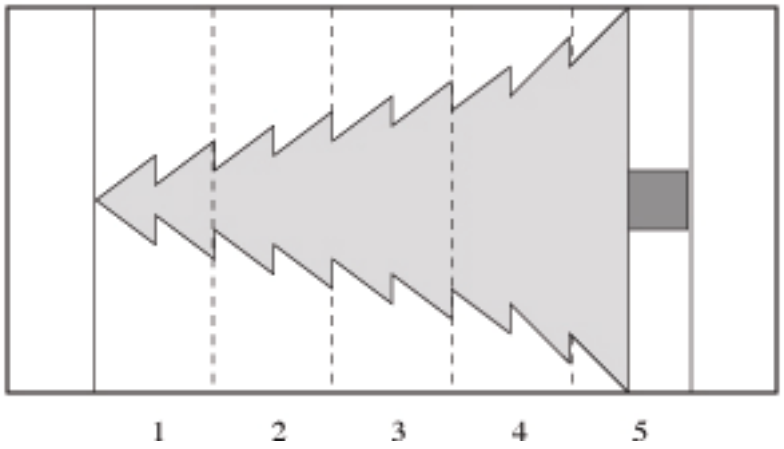


Figure 6. Vertical (in plane) Virtual Soft Fingers.

In computing key elements from silhouettes using virtual soft-fingered grasping, we also considered horizontal bands of the pixel array similar to the vertical bands. From each band, we obtain two measurements: (a) average fraction of box width from left side of box to nearest silhouette pixel, and (b) average fraction of box width from right side of box to nearest silhouette pixel. Obviously, this yields twice as many key elements as horizontal bands, and we chose to use three bands.

Additional Keys for 2D Queries

Although not ultimately included in the baseline keys, occluding edges are another type of 2D geometric data that can be automatically extracted at index-creation time. We describe the basic method we use for finding occluding edge pixels. Our code for extracting occluding edges for a given object model at a given viewing direction θ assumes that the model is a proper faceted boundary representation. After computing the normalizing viewing transformations, we render the object in a designated body color. Then, for each edge, we compute the dot products of the normal directions of its adjacent facets and the viewing direction vector. If the dot products have different sign, render the edge in the designated edge color (different from the body color). Using the Z-buffer when rendering ensures that non-visible edges are discarded. We then turn off all the pixels on the silhouette boundary. The remaining pixels that are the designated edge color are the pixels of the occluding edges.

We computed a key element from the occluding edge pixels: the ratio of the number of occluding edge pixels to the number of silhouette boundary pixels. We chose to wait to see how well users would identify occluding edges at 2D query construction time before expending further resources along this direction.

Key Computation for Indices for 3D Queries

When 3D geometry data is the input for computing key vectors, it is straightforward to extend the virtual soft-finger technique to make use of the values in the Z-buffer array. This extension constructs key elements from the rectangular region of the Z-buffer corresponding to the bounding box of the geometric data. The region is first divided into an array of equal-sized rectangular sampling regions; it is convenient for the width and height of these regions to match the width and height of the vertical and horizontal bands used for the silhouette. With five vertical and three horizontal bands, there are fifteen rectangular regions. For each sampling region, we compute the average depth value, which lies in the range [0, 1]. As described above, the z-axis projection parameters are set to match the bounding box of the geometric data, so rendering in the Z-buffer results in normalized measurements. Therefore, it is important to include the ratio of the depth of the bounding box to its height as a key element. We also note that it seems intuitive to "invert" the values in the Z-buffer via the function $z'(x,y) = 1 - z(x,y)$ so that its contents can be visualized as mountainous terrain instead of a trough.

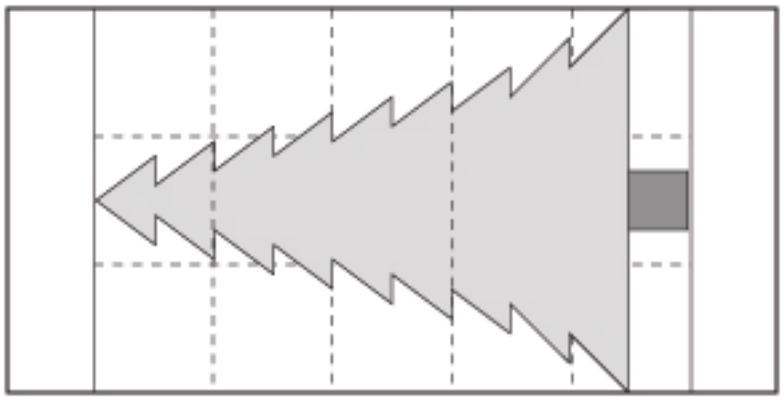


Figure 7. Sampling Regions.

In summary, ten key elements are computed from vertical bands, six from horizontal bands, and fifteen from depth sampling regions. In addition, there are the aspect ratio and fill ratio, making a total of thirty-three key vector elements.

Compression with Principal Components

As described above, the baseline key computation algorithm generated a key consisting of thirty-three real numbers. The elements of the key show significant correlation over the database of keys, as is shown in Figure 8, which shows a grayscale image of the matrix of Pearson correlation coefficients, after absolute value is taken. Each cell shows the correlation coefficient for the key elements in the corresponding row and column.

Brighter cells correspond to stronger positive or negative correlation, and darker cells to poor correlation. Of course, the diagonal elements are white; each key element is perfectly correlated with itself.

We used principal components analysis to generate shorter keys having nearly the same information content. If the original key vector is $k = k_0, k_1, ..., k_{32}$ then the first principal component is the linear combination $a_0 k_0 + a_1 k_1 + ... + a_{32} k_{32}$ having the maximum variance among all normalized linear combinations - those for which $\sqrt{a_0^2 + a_1^2 + ... + a_{32}^2} = 1$. The Nth principal component is the linear combination with the largest variance among all these combinations which are orthogonal to principal components 1 through N-1.

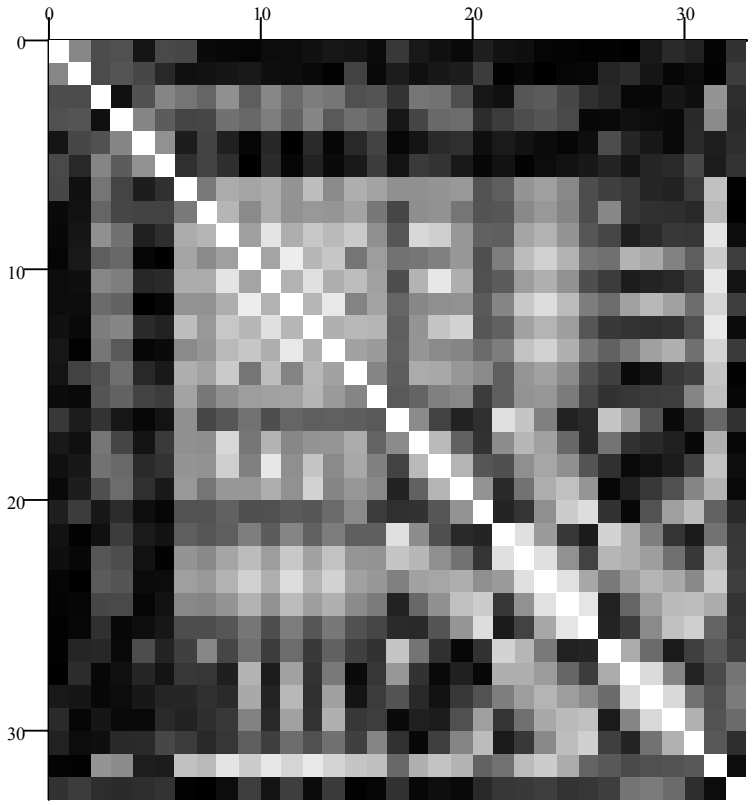


Figure 8. Correlation Coefficients of Key Elements (absolute value).

The first step in key compression was to prepare a key matrix *K*. *K* was 33 wide and 237,960 long; each row contained the 33 elements of the key for a particular viewpoint of a particular object. The key matrix contained most of the keys in the database – all 1322 viewpoints for each of 180 objects. (MathCAD version 7, the software used for the

computation, limits the size of arrays, restricting the number of objects which could be included.)

The covariance matrix *C* was then calculated. Each element $C_{i,j}$ contained the covariance of the ith column of *K* with the jth column (the covariance of k_i with k_j).

Finally, the eigenvalues of *C* and corresponding eigenvectors were computed. The magnitude of each eigenvalue is proportional to the variance of the linear combination with coefficients defined by the elements of the corresponding eigenvector. The eigenvector e_N for the Nth largest eigenvalue gives the coefficients for the Nth principal component.

The first *n* principal components were selected as the elements of the compressed key vector *c*:

$$c^T = \begin{bmatrix} e_1 \\ e_2 \\ \cdots \\ e_n \end{bmatrix} k^T \text{, where } n \le 33.$$

This created a key comprised of orthogonal elements having maximum variance for the data set. The shorter key reduced the database storage space and search time linearly with the reduction in key length; that is, the compression factor was $n/33$. And, because of the correlation between the original key elements, we were able to significantly compress the key and database using a key length of 15, with minor degradation of object recognition, as discussed in the Performance section.

Keys from Machining Features

Apart from the basic geometry keys (described in the Baseline section) that a geometric search engine for CAD application might use, the use of machining features as keys in the search should be useful in yielding an efficient response to a particular query. In the query example of "fastener", apart from the basic shape, information about the number of holes, their locations, diameters, and lengths would aid in reducing the number of fastener designs returned from the database.

A machining feature is one usually created by the removal of some material from the stock used to create the object, e.g. in a milling or drilling operation. For example, holes and pockets are considered machining features. Holes and pockets are the more prevalent features in most designs and the algorithms for their recognition will be described.

The algorithms for finding holes and pockets are based on the work of Ames (1991) in parts classification. It is assumed that the geometry of the part is represented in ACIS, a commercially available solid modeling software package. Translators are available from such common CAD packages as ProE and Catia to ACIS. ACIS uses a b-rep (boundary representation) to express the solid model in terms of lower level geometric and topological entities such as vertices, edges, and faces. Edges and faces each have underlying geometries associated with them such as straights and ellipses associated with edges and planes and cones associated with faces. The feature recognition algorithms are based purely on geometric and topological hints rather than on delta volume. The algorithms rely heavily on the construction of featurettes, which are low level features, such as sets of parallel edges or sets of faces forming a 360 degree cycle. Featurettes allow the feature recognition to proceed in smaller, simpler steps that are much easier to develop.

Hole Recognition Algorithm

When the ACIS solid model of a part is read into the feature recognition software, the first thing that is done is that the model is parsed to extract lists of all the faces, edges and vertices. These lists are traversed repeatedly, looking for certain characteristics that can start the definition of a particular feature. For example, in the case of trying to find the holes in a part, the list of faces is traversed until a conical, concave face is found. If that face is not already being used in another feature, then a test to see if the face is a turnable hole is performed. The test makes sure that the wall thickness next to the hole is above a minimum level. If this test is passed, then a hole facelist is created. The hole facelist consists of faces that are adjacent to the original face and to each other, are turnable, are either conical or planar, and terminate with either a cone or plane. Because the algorithm is geometry-based, it can identify holes as simple as one cylindrical face and as complicated as a hole with different diameters with other holes running through it; see Figure 9.

Pocket Recognition Algorithm

The algorithm for locating pockets looks for a closed set of faces, which is called a 2.5D faceset. This faceset, consisting of planes and half-cylinders, is essentially the sides of the pocket. This faceset could be defined by sweeping a face through the interval occupied by the faceset. The sides of the pocket are found by first finding sets of straight edges that are parallel and overlap. Then the faces that have these edges and are adjacent are traversed until the concave sum of the angles between the normal faces adds up to 360 degrees. Figure 10 illustrates how the algorithm found a pocket hogged

out from one side of a solid cube. The cube is drawn in a wire frame representation, and the sides of the pocket are solid.

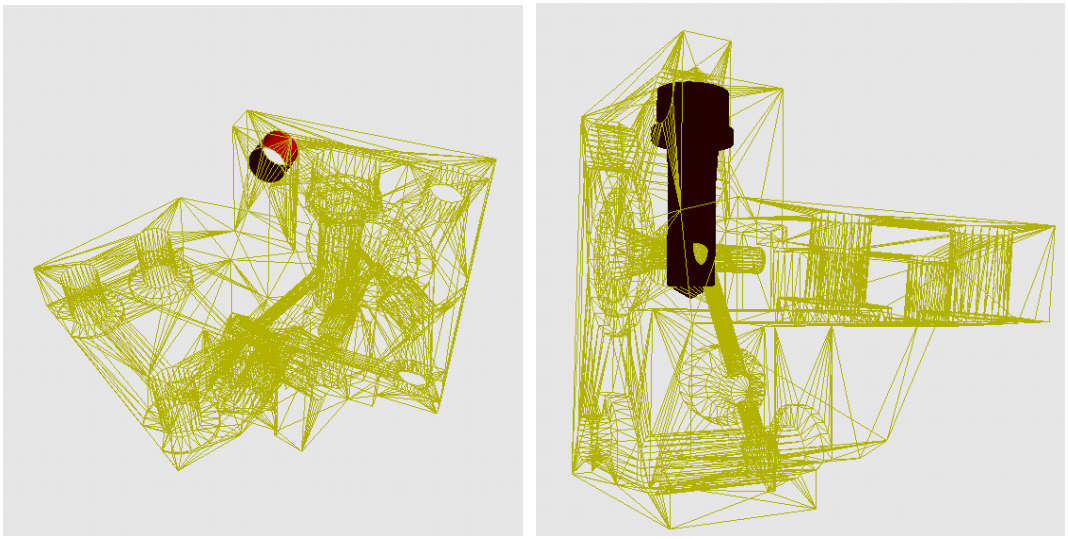


Figure 9. Simple and Compound Holes.

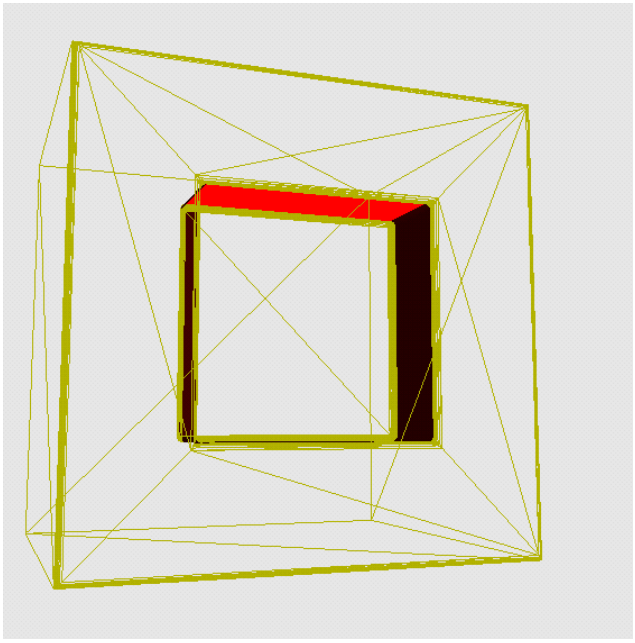


Figure 10. Example of a Pocket.

On Finding Features from Polygon B-Rep Models

We also did preliminary research into the problem of finding features directly from 3D b-rep polygonal models. Specifically, we developed and implemented a method for finding cylindrical holes. Although a particular technique is sensitive to how a hole has been faceted, other parts of the method are generally applicable.

The method has quite a few steps. First, we map all polygon edges that are concave to partitions of the sphere according to direction. Second, we use these partitions to efficiently group edges that are parallel to each other (with an epsilon tolerance) into collections. Each collection determines an equivalence class of possible holes whose axes are within epsilon of being parallel to edges in the cluster. Third, for each pair of polygons that co-bound an edge in a given collection, we hypothesize the line that would support the axis of a hole the polygons belong to. We then cluster the polygon pairs by these supporting lines. For each of these clusters, we form the connected components of polygons.

A connected component must pass two tests before it is classified as a cylindrical hole. First, we look for chains of polygon edges that lie in planes perpendicular to the candidate axis. There must be cyclic chains with non-repeating vertices that lie in a perpendicular plane in order for the component to be a hole. Second, we test whether the interior of the convex hull of the cyclic chains is empty and accessible from at least one direction. To do this, we first hypothesize a sphere whose center lies on the possible hole axis and whose diameter is just small enough so that it should be able to translate into the hole from an open end just outside the hole without colliding with any of the model. This is called the *accessibility test*, and we perform it by using the swept-volume distance computation algorithm described by Xavier (1997). The connected component of polygons is classified as a hole if and only if at least some fraction of its interior is accessible by the sphere.

Obviously, the technique we use to identify polygon pairs that might belong to the same hole is only applicable to faceting algorithms that result in each polygon of a given hole having at least one edge parallel to the hole axis. This technique will not work for perfectly symmetrical triangular facetings of cylindrical holes.

Although we have not followed up on generalizing this technique as a part of this project, these preliminary results were among those supporting the proposal of the "Cloud to CAD" LDRD research project.

Database Search

Each geometric object is represented by an ordered set of real numbers called a *key*. Figure 11 shows a pictorial representation of a single key of size *n*. The length of the key can be varied, but must be the same for all keys in a given database. A database consists of an unordered set of keys, plus the model filename and viewpoint for each key. The database lookup algorithm finds the key in the database that best matches the given key--the one that has the smallest distance from the given key. The weighted sum of the squares of the distances of each pair of elements is used to calculate the goodness of the match, i.e., the distance between keys $\boldsymbol{a}$ and $\boldsymbol{b}$ is $\sum_{i=0}^{n-1} w_i (a_i - b_i)^2$, where *n* is the size of the keys and $\boldsymbol{w}$ is a vector containing a weighting value for each key element. Unless otherwise specified, all key elements are weighted equally.

Linear Search

The naïve search algorithm starts by looking at the first key in the database and progresses linearly through the database. The problem with this approach is that every element of the database must be considered before the best match can be determined. Such an algorithm is said to have linear or O(n) (read "order of n") run-time complexity because the time required grows proportionally as *n* increases, since all *n* elements must be considered. An algorithm with lower complexity, which will be faster for large *n*, is presented below.

3.12	2.544	11.90	0.003	…	1.201
0	1	2	3	…	n - 1

Figure 11. Sample Key of Length n.

Non-Linear Search

Nene and Nayar (1996) present an algorithm to search for nearest neighbors in high-dimensional spaces. We have further developed the algorithm to support search for *m* nearest neighbors, and to automatically choose an appropriate search range distance ε (see below). Figure 12 represents a sample database of 6 keys ($\boldsymbol{k_0} - \boldsymbol{k_5}$), each of length 5 ($k_{n0} - k_{n4}$). Each row in the diagram represents a single key. Each box represents a single

element of a key. Each box is labeled k_{ij} where i is the key number and j is the element of that key. This database and this graphical notation will be used to illustrate the database lookup algorithm.

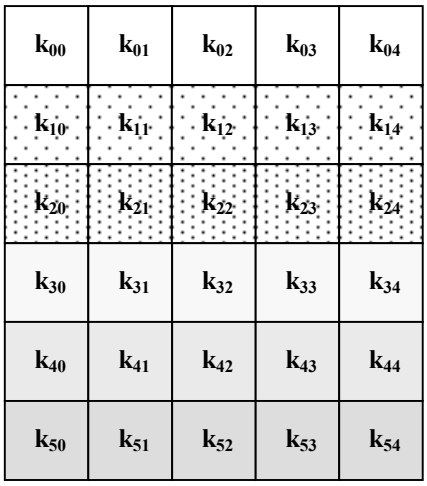


Figure 12. Sample Database of 6 Keys, Each of Length 5.

In order to improve the run-time complexity of the database lookup, the elements in the database must be ordered. As an analogy, consider the task of looking up a word in a dictionary. Once again the naïve approach is to start at the front of the dictionary and look at every word until you either find the word you are looking for or reach the end of the dictionary; the time complexity, once again, is O(n). Clearly such an approach is absurd. Since a clear ordering exists for words in the dictionary, a more direct method can be used to lookup words. Specifically, binary search, which cuts the search space in half each time, can be used to look up a word in a dictionary in $O(\log_2 n)$ time.

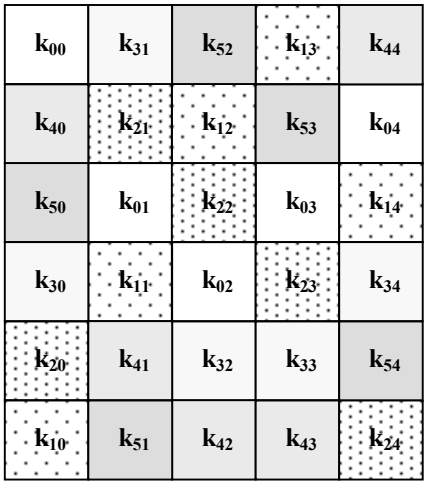


Figure 13. Sorted Sample Database.

Since the elements of the geometric keys are equally significant, a search algorithm must consider each element with equal importance. Thus, there is no simple way to create a single ordering of keys. We solved this problem by creating n orderings of the keys (where n is the length of the keys). This can be achieved by sorting each column in the database. Recall that the i^{th} column contains the i^{th} elements of all the keys. Figure 13 shows the sample database in Figure 12 after sorting all the columns.

Assume that the key k_3 is to be looked up in the sample database. In other words, assume the key being looked up exactly matches a key in the database. Since all a key's elements will have the same importance, the order in which the elements are looked up does not matter. Thus the key's elements will be looked up 0 through n order.

For all elements in the key, the corresponding column will be searched. For example, the 0^{th} column will be searched for the 0^{th} element of the key. Since the k_3 key is in the database, an exact match with each element of k_3 will be found in each column. The run-time complexity of searching a single column is $O(\log_2 m)$ (assume m is the number of keys in the database; the length of the column). Assuming n is the length of the key. The run-time complexity for the entire key lookup is $O(n \log_2 m)$.

One of the most significant contributions of this research is the ability to find not only exact matches but also similar geometric objects. Thus the lookup algorithm must be able to find close matches. This can be achieved by searching for matches within a range of the search key. Figure 14 illustrates the first phase of the closest match algorithm (spacing has been added between the columns to allow room for the annotations). Let T represent the target key to be looked up. Let ε represent half the size of the lookup range.

The first step of the search is to find all the elements in each column that are within ε of the target's key. More formally, find all elements in column i that are greater than or equal to $(T_i - \varepsilon)$ and less than or equal to $(T_i + \varepsilon)$. In Figure 14, the elements that are within the range are drawn with thicker lines. Finding all elements within the ranges has run-time complexity $O(n \log_2 m)$. (While it does take twice as much work as finding a single key, constants are ignored in run-time analysis.)

The second step is to identify which keys have all their elements within the range $[(T_i - \varepsilon), (T_i + \varepsilon)]$. These are the keys that are closest to T. In the sample problem shown in Figure 14, only keys k_2 and k_3 fall within the range. The third step of the lookup algorithm is to apply the distance function to these keys and produce an ordering.

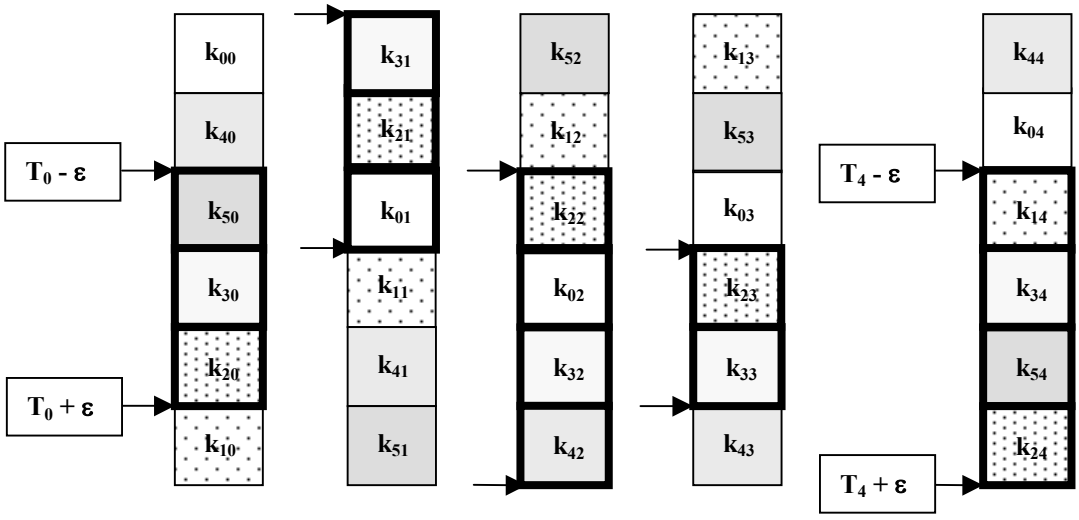


Figure 14. Searching for Closest Match.

Let c be the number of closest matches desired. In order for this algorithm to work correctly, there must be at least c keys that fall within the ranges. If there are more than c keys, the application of the distance function will find which are the closest c keys. If there are fewer than c keys, the size of the ranges must be expanded; in other words, ε must be increased, and the entire process must start again.

At this stage, it is possible that this search algorithm has not found the best match. The search algorithm first considers the keys that are a close match by doing an element by element comparison. The result is a set of keys for which each element is a close match to the same element of the search target key. Then the algorithm searches through these keys to find the closest match among them. However, it is possible that, for the best match, not all of the individual elements are close to the elements of the target; as a result, the best match may not be in the set of close matches. Consider the following 2 dimensional case:

Target key to be looked up: (10, 10)
Database: (10, 1000) (100, 10) (9, 9)

The key which is closest to the target is (9,9). However, if the ε for the search algorithm is set such that only elements within 0.5 distance of the target elements are selected, the first step of the search will find the keys (10,1000) and (100,10). The second step will only consider these two keys and thus (100,10) will be found to be the closest key.

To ensure finding the best match, the algorithm makes a second pass with a new value of ε. If c matches were requested, than ε is set to the c^{th}-nearest distance found at this

point. Repeating steps 1 and 2 ensures that all matches with a distance less than or equal to this value are found, possibly identifying better matches. For the example above, a second pass would be made with $\varepsilon = 90$ [the distance from (10, 10) to (100, 10)], ensuring that the key (9, 9) is recognized as the closest key.

The total run-time complexity depends on the data. In the best case, a good ε is chosen and the algorithm runs in $O(jn \log_2 m)$ where m is the number of keys in the database, n is the size of the keys and j is the number of keys within the matching range. In the worse case, ε must be increased several times in the first pass before enough keys are found (until j is large enough). *Enough keys* is a parameter to the lookup function; the user specifies how many matches are required. Assuming that ε is doubled each time, it might have to be doubled as many as $\log_2 m$ times before j is large enough. Thus the worst case time-complexity is $O(jn (\log_2 m)^2)$.

Performance

Evaluating the performance of this system raises the question of what standard to apply. How can one tell whether a match object is "similar" to a target object? And how similar are the two? We chose to assess performance in two ways, one objective and one subjective. For a series of target models chosen randomly from models contained in the database with random viewpoints, we measured the percentage of searches which returned the target object itself as the best match object. And we comment on the subjective sense we have when using the system. The system was tested on a database built from 213 CAD models generated over several years for other projects. Objects in the database ranged from spheres and cubes through fasteners and subcomponents to robot arms. There was no kitchen sink, but there was a faucet. Some objects proved to be quite similar to others; some objects were distinctive.

Object Recognition

When the target model is represented in the database, we expect the target model itself to be returned as the best match object. After all, the two objects are as similar as possible. However, since the database represents the model at discrete viewpoints, in general a target at an arbitrary viewpoint will not perfectly match any key in the database. If the system works well at recognizing similar objects, we may still expect it to return the target object as the best match, or perhaps a very good match.

Note that some objects may be indistinguishable from some viewpoints. For instance, several objects in the database have square bases. Seen from the bottom, all are simply squares. Also, the key algorithm used describes overall shape characteristics and is not sensitive to interior features; we found that it sometimes reported a "best match" with an object of generally similar shape.

We tested the system with a series of database searches using target objects taken from among the database objects. The particular target object and its viewpoint were selected randomly for each search using a pseudo-random number generator. For 1000 such searches in a database of 213 objects, the results are shown in Table 1. As can be seen, about 95% of the trials using linear search on baseline keys returned an ideal "best" match object, and 99% were deemed reasonable. Only 1% were poor matches. Even these duds showed a good match to the silhouette; for example, two long, thin objects with different cross-sections.

Recognition performance of the non-linear search was equivalent to the linear search. Compressing the key to 15 elements, and using linear search, lowered the number of

ideal matches to about 93%, although there was no significant change in the rate of reasonable matches. Compressing further to 10 elements resulted in continued slow degradation of performance, with 92% ideal matches and 2% poor matches. A compressed key with only 5 elements showed markedly poor recognition of the same object. Searches with 10 or 15 key elements returned some very poor matches (about 0.5%, and grouped with poor matches in the table), for which even the silhouette of the match object had little resemblance to the target object.

Table 1

Search Type	**Linear**	**Non-linear**	**Linear**	**Linear**	**Linear**
Key Elements (B: Baseline, C: Compressed)	**33 - B**	**33 - B**	**15 - C**	**10 - C**	**5 - C**
Trials	1000	1000	1000	1000	1000
Same object is "best match"	928	932	908	901	734
Indistinguishable object-viewpoint is "best match"	21	17	24	16	-
Reasonable "best match" with differing features	43	38	57	63	-
Poor "best match" with different general shape	8	13	11	20	-

User Observations

The Geometric Search Engine generally works as expected, and gives the impression of "understanding" shape. As a result, users tend to anthropomorphize the system and discuss "what it was thinking" or "why it chose this match."

Typically, the first few match objects returned by the GSE (from a database of 213 objects) seem similar to the target object. Beyond that, there is the sense that the system is "stretching it," but appropriately as there are no more good matches. There are occasional "flyers" for which a match object with a good distance score seems clearly less similar than other match objects with poorer scores. These are infrequent, about the same 1% rate as poor matches in the object recognition test. There are also occasional "happy surprises" when users are unexpectedly pleased by the appropriateness of the selected match object-viewpoint.

Use with a 15-element compressed key subjectively feels indistinguishable from uncompressed keys, but operation with a 10-element key is noticeably less satisfactory; match objects no longer seem as appropriate. Use with a 5-element key returns many laughable matches.

The GSE seems less responsive to depth information than to an object's silhouette. If no viewpoint yields a good match for both, the selected viewpoint often displays the match object from a steep angle in an apparent attempt to match the silhouette, even though the result is much greater depth variation in the match view than in the target view.

A perceived deficiency of the user interface is that it always returns the same number of match objects, regardless of their distances from the target object. For objects which are relatively distinctive, this means that some very poor matches are returned. Although the operator can use the displayed distance to assess the quality of a match, it might be more appropriate to limit matches to those within a specified distance.

Range Data

Polygonal models were constructed from 3-dimensional range data collected on: a wrench, a hex key, a square with a hole in the center, and a large hexagonal nut. Although the resulting models showed only the top surface and some of the sides, this was sufficient to use them as target models. The system correctly recognized the wrench and hex key. Due to poor reflection characteristics, the models for the square and nut had serious data dropouts and noise artifacts. The models were not recognizable either by humans or by the GSE.

Other

Since object silhouettes can be very distinctive, we wondered whether the GSE might perform well using only silhouette information from object models. To help assess the importance of the depth cues, the weights for key elements containing depth information were set to zero. Object recognition accuracy dropped off sharply, with only about 78% of random searches returning the correct object as the best match. Even more striking was the fact that lower-rated matches were no longer similar in overall shape. The system turned into a kind of parlor trick, showing that objects which resemble each other not at all can still have similar silhouettes from particular viewpoints.

We have not performed detailed studies of algorithm speed. The time for key computation is about 0.4 seconds on an SGI Octane with an R10000 CPU at 195 MHz. Linear database lookup using a key of 33 elements requires 1.3 sec. Non-linear search with the same key size ranges from less than 0.1 to over 20 seconds; the mean time is about 5 sec. The speed is aided by the fact that the entire database can be kept in RAM.

Directions for Future Work

Because of the pioneering nature of this work, there are many directions for future research. We divide these up into two general areas.

Improving Performance with Current Key Computation Methods

Here, we address the questions of how we might speed up search, increase accuracy, and/or decrease the amount of index space required for a given accuracy within the context of computing keys from virtual soft-fingered grasp measurements.

While informal experiments showed that accuracy improves with more-dense discretization of the set of viewing directions, we used the same set of 1322 viewpoints for all objects when building their key sets. This roughly uniform discretization of viewing directions was the first we tried that performed well for a preliminary 16-object database. By trying **Variable Viewpoint Discretization**, or locally varying how finely the set of viewing directions is discretized, it is possible that we can decrease the number of key vectors needed for a given accuracy and increase the accuracy without increasing the number of key vectors.

Objects with one or more axes of symmetry give rise to redundant (indistinguishable) viewpoints. For example, at present the database contains keys for each of 1322 viewpoints for a sphere, all of which are identical to each other. This is an extreme instance, but many objects have significant redundancy. The database could be compressed and search time reduced by storing only one viewpoint as representative of all viewpoints which have keys "near enough" to the representative. Conversely, for certain objects, key elements change rapidly enough near certain viewpoints to cause a loss of search accuracy. By increasing the fineness of viewpoint discretization where appropriate we would be able to increase system accuracy.

We see several criteria for adjusting the discretization to obtain sufficient accuracy. The *local discretization criterion* is that for each key vector of any given object, there must be some index key vector which is within some *maximum separation epsilon* and which also belongs to that object. The *global discretization criterion* is that for each key vector of any given object, there must be some other index key vector of that object that is closer than any key vector of any other object in the database, excluding those keys within some *indistinguishability epsilon*. The *redundancy criterion* is that if an index key vector is within the maximum separation distance of another belonging to the same object and that key vector can be removed without causing violation of the local discretization criteria, then it is redundant.

Since the number of key vectors for a given object is generally infinite, these criteria cannot be tested directly. A promising direction is to describe the sphere of viewing directions with an interpolating geometric subdivision scheme (Schroder and Zorin 1998) and let the subdivision vertices determine the discretized viewing directions. Starting with a nominal subdivision level, we would test each point of the corresponding discretization to check where it fails to meet the local criterion and where it meets the redundancy criterion. Locally finer or locally coarser discretizations would then be obtained by consulting the subdivision hierarchy. Local search starting at the vertices could be used to make testing of the discretization criteria more robust.

Speed might be improved through the use of alternative search algorithms, either through **specialized data structures** or exclusive use of **ordered keys**. It is easy to see that the algorithm could be significantly simplified and its running time reduced if an ordering could be imposed on the keys. Given this constraint, each column of the database would still be sorted, but the lookup algorithm could search the columns in order by their importance. This would allow for unsuccessful matches to be determined earlier. Pruning these unsuccessful matches would reduce the number of elements in each match set. The fact that the principal components method produces key elements in order of variance suggests that this ordering may be useful, but we have not investigated further.

General Shape Indexing

Our scheme of computing keys from virtual soft-fingered grasp measurements has a number of shortcomings and leaves much room for substantially different future work.

As the reader must realize by now, we chose to concentrate on developing methods for answering queries corresponding roughly to unidirectional surface scans. Research on key computation and indexing methods for 2D geometry data was put aside because of the great accuracy advantage the 3D version of the algorithm showed over the 2D version in preliminary tests later confirmed with the full data set. However, because 3D models are often not available, it would be extremely useful to be able to search a database by geometry using only a 2D query, such as one based on a photograph or line drawing.

One possible direction for improving the performance with 2D geometry data would be to **use occluding edge pixel information less naively**. Since the occluding edges are represented as a set of pixels, the many key element computation schemes might be applicable. For example, the principal axes and moment information are easily computed. The number of connected components and the number of branch junctions are also easily computed. The fractions of total pixels in horizontal and vertical bands could also be counted. Of course, the virtual soft-finger computations that we applied to

the silhouettes could also be applied. Undoubtedly, there are other functions that could be applied to the set of occluding edge pixels.

While we can compute these pixels for the object viewpoints generated at index creation time, there remains the question of how they would be identified at query time. For similar reasons, we have yet to **integrate feature recognition with general shape queries**, although we can identify certain features automatically on 3D CAD models. Although we can certainly use recognized features in computing key elements for object viewpoints, we have not explored the possibility of automatically identifying them in 2D query images. It is possible that this could be done for a set of parameterized features by building on computer vision techniques.

An alternative approach might rely on **interactive query construction** for identification of features in a 2D image. This is similar to the way IBM's QBIC system relies on the user to identify the silhouettes of significant objects in photographs, although we would only require this at query time since our database contains only 3D geometry. Interactive query construction might also enable the user to "clean up" dirty data, supply data missing to occlusion, and stretch and warp the query geometry, for 3D queries as well as 2D queries.

Finally, other methods for shape characterization, such as Fourier and wavelet decomposition, should be explored. While we have seen these techniques used in shape description in the literature, it remains to be seen how well they will work in shape recognition and with geometry with a wide range of scale and frequency.

Conclusion

The Geometric Search Engine creates a new search capability: search for objects whose shape is similar to a target object. In the present implementation, the shape need only be similar from a single viewpoint, although extension to similarity from multiple viewpoints is straightforward. The search is based on comparing key vectors computed for the target objects and for objects represented in a database. Algorithms for computing the keys based on object shape were presented, along with feature recognition algorithms which may be integrated in the future to achieve more advanced search. Keys computed offline are stored in a database, which may be searched in real time for a moderate number of objects. A non-linear search method was implemented for faster search, and a key compression method developed to reduce both search time and storage requirements.

The GSE has been implemented with a graphical user interface which allows users to select a target object, rotate it to present a characteristic view, search for similar objects, and interactively view alternative matches. We have demonstrated that the GSE can recognize objects from range data collected using physical objects, as well as recognize CAD-modeled objects. Users find that the system effectively selects similar shapes in most cases where such objects are available in the database, although poor selections are observed at a rate of approximately 1%.

The GSE was tested by searching the database for the best match to objects represented in the database, as seen from a randomly selected viewpoint. Results show that 95% of the searches returned as the best match are either the same object or an object indistinguishable from that viewpoint. Another 4% of searches returned a match deemed reasonably similar to the target object; less than 1% of searches returned a poor match.

Note that these results were obtained using naïve choices for two parameters which if optimized could significantly improve performance. These are the sampling density for viewpoints and the number of virtual soft-finger regions. In addition, these results are based on ignoring the size of objects.

References

A. Ames, 1991, Production Ready Feature Recognition Based Automatic Group Technology Part Coding, in *Solid Modeling Foundations and CAD/CAM Applications*, Association for Computing Machinery, New York, NY

J. Krumm, 1997, Object Detection with Vector Quantized Binary Features, in *Proc. of IEEE Conference on Computer Vision and Pattern Recognition*

H. Murase and S. Nayar, 1995, Visual Learning and Recognition of 3D Objects from Appearance, in *International Journal of Computer Vision*, vol. 14, no. 1, pp. 5-24

S. Nayar, S. Nene, and H. Murase, 1996, Real-Time 100 Object Recognition System, in *Proc. of IEEE International Conference on Robotics and Automation*

S. Nene and S. Nayar, 1996, Closest Point Search in High Dimensions, in *Proc. of IEEE Conference on Computer Vision and Pattern Recognition*

P. Schroder and D. Zorin, organizers, 1998, Course notes on Subdivision for Modeling and Animation, *SIGGRAPH 1998*, Orlando, FL, July 19-24

M. Turk and A. Pentland, 1991, Face Recognition Using Eigenfaces, in *Proc. of IEEE Conference on Computer Vision and Pattern Recognition*

P. Xavier, 1997, Fast Swept-Volume Distance for Robust Collision Detection, in *Proc. of IEEE International Conference on Robotics and Automation*

D. Zhang and M. Hebert, 1997, Multi-Scale Classification of 3-D Objects, in *Proc. of IEEE Conference on Computer Vision and Pattern Recognition*

DISTRIBUTION (FIRST PRINTING, DECMBER 1998)

1	MS	0188	LDRD Office, 4001
1		1002	P. J. Eicker, 9600
1		1006	P. Garcia, 9671
1		1006	L. Meirans, 9671
3		1006	L. P. Ray, 9671
1		1008	J. C. Fahrenholtz, 9621
15		1008	P. G. Xavier, 9621
1		1008	T. R. Henry, 9621
1		1008	R. A. LaFarge, 9621
1		9018	Central Technical Files, 8940-2
2		0899	Technical Library, 4916
1		0619	Review & Approval Desk, 15102 For DOE/OSTI
1		0161	Patent and Licensing Office, 11500

DISTRIBUTION, SECOND PRINTING, NOVEMBER 2001:

2	MS	0974	L. P. Ray, 6523
1		0986	R. A. LaFarge, 2662
1		1002	G. R. Langheim, 15200
1		1004	R. W. Harrigan, 15221
5		1004	ISRC Library, 15221
6		1004	P. G. Xavier, 15221
1		9211	Central Technical Files, 8945-1
2		0899	Technical Library, 9616
1		0612	Review & Approval Desk, 9612 For DOE/OSTI