International Journal of Computational Geometry & Applications

DCEL: A POLYHEDRAL DATABASE AND PROGRAMMING ENVIRONMENT

GILL BAREQUET*
Department of Computer Science, Tel-Aviv University, 69978 Tel-Aviv, Israel.
E-mail: barequet@math.tau.ac.il

Received 14 August 1996
Revised 24 May 1997
Communicated by M. C. Lin and D. Manocha

ABSTRACT

In this paper we describe the DCEL system: a geometric software package which implements a polyhedral programming environment. This package enables fast prototyping of geometric algorithms for polyhedra or for polyhedral surfaces. We provide an overview of the system's functionality and demonstrate its use in several applications.

Keywords: geometric software, databases, programming environments, polyhedra.

1. Introduction

Computational geometry has offered a large amount of algorithms during the last two decades. Software implementation of these algorithms makes them valuable not only for theoreticians but also for practitioners in academia and industry. This is in many cases the appropriate tool for choosing the best algorithm for a specific problem in a given context: hardware platform, operating system, programming language, typical inputs of the application, robustness considerations, etc. The importance of applied computational geometry is now being recognized.[10] Dedicated international events, e.g., the workshop of the geometry center in Minneapolis, MN (January 1995) and the ACM workshops in Philadelphia, PA (May 1996) and in Nice, France (June 1997), are held. Large-scale geometric libraries such as the European CGAL project[13] are designed and implemented.

Implementing geometric algorithms may be more difficult then implementing other numerical algorithms.[10,25] An important reason for this is the fact that most of the existing programming languages support data types for representing numbers (e.g., integral, real, complex) and related operations, but not for geometric entities (e.g., points, planes) and operations. The implementor usually has to create the

*Present address: Center for Geometric Computing, Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA. E-mail: barequet@cs.jhu.edu

needed infrastructure, to reuse pieces of code taken from other programs, or to use suitable geometric software packages.

Many geometric programs written for specific purposes, such as for computing Voronoi diagrams and convex hulls, are available in the public domain. Several packages which include geometric data structures and libraries of basic algorithms are available too. These include packages which support 2-dimensional geometry (such as XYZ GeoBench,[25] GeoLab,[24] and LEDA[19,20]) or 2- and 3-dimensional geometry (e.g., PLAGEO/SPAGEO[14]). The XYZ GeoBench and GeoLab systems have graphics and animation capabilities. Other systems, such as GASP[26] and GeoSheet[17] support visualization of geometric algorithms. GASP is an animation system used for preparing demos and video tapes, while GeoSheet is an interactive tool for visualizing geometric algorithms in distributed environments.

The system described in this paper is a database and programming environment for polyhedra (objects with piecewise-linear boundary), featuring data structures and basic algorithms, some graphics functionality, and, most important, a ground for easy prototyping of geometric algorithms for polyhedra.

Geometric software environments can make the implementation of complex algorithms easier. The use of already coded (and debugged) primitive functions may save a significant amount of time in implementing a high-level application. This is however not always trivial. Verifying that they run on the same hardware and operating system is a minor problem: portability between platforms usually goes without saying. In many cases the main source of problems is the interface between the application and the package. Using a geometric package requires that the user understand its terminology, use its (usually complex) data structures, consider its side-effects, obey its general-position requirements, follow the needed sequences of operations (e.g., invoking preprocessing functions before calling the desired function), and so forth. In addition, it is not always possible to use the graphics capabilities of the package (e.g., when the workstation running the application does not contain a specific piece of hardware), or to provide its memory-consumption needs.

Imagine, for example, how simple it is to call a function which computes the inverse of a matrix. Arrays of numbers (real numbers, in this case) are elementary and make the following C interface self-explanatory:

```
extern int inverse ( /* Returns TRUE if regular, otherwise FALSE */
   int      dim,       /* Number of rows and columns */
   double * in_matrix, /* Input matrix */
   double * out_matrix /* Output matrix */
);
```

This interface is probably legible also to users oriented at other programming languages. Now consider calling a function which computes the convex hull of a set of points in the plane. Its C prototype may look like:

```
extern int conv_hull ( /* Returns TRUE on success, otherwise FALSE */
   int       n_points,      /* Number of points */
   Point     points [],     /* Point array */
   int     * n_hull_points, /* Number of hull vertices */
   Polygon * hull           /* Convex hull */
);
```

The type `Point` already needs some explanation even in the simple 2-dimensional case: it may be an *array* of two real numbers, a *structure* with two fields (say, `x` and `y`), each of which is a real number, or another data type. The interface in this example becomes much more complex when we consider the type `Polygon`. It may be a simple array of points, or a linked-list of points, or a doubly-connected linked-list, or a more complex structure. Each option may include for each hull vertex the actual coordinates or may refer to vertices by indices in the input point list. Whatever the data structures are, the programmer should either use them throughout the application or implement an intermediate level that converts all the structures in each call to a function of the package.

Using an object-oriented programming language (thus hiding a lot of unnecessary information) may be extremely helpful in organizing an application and in facilitating its design and implementation. We claim, however, that this is insufficient when the "spirit" of the interface is based on a hierarchy of complex structures (e.g., by using C++'s class inheritance). In this case the interface is as complex as in the C version and a long learning phase is required. A C++ interface hides the actual implementation of a structure but provides numerous 'methods' for accessing the data in the structure. It is naturally a "cleaner" interface but it is not necessarily simpler.

Our geometric software package, DCEL, handles polyhedral surfaces. It provides a database for storing a description of a polyhedron (or polyhedra) and an easy-to-use ground for implementing algorithms for polyhedra. The package supports the communication with external software, e.g., other geometric systems or graphics packages. It was used for the implementation of various algorithms.[1,2,4,5] The entire package was implemented in standard C. It runs on PC's, Silicon Graphics workstations (Indy's and Indigo's), Sun workstations, Digital workstations (Vax machines and DecStations), and IBM workstations (the RS6000 family).

The motivation for the system is twofold. The first goal is the ability of the application not to be aware of the actual implementation of even a single geometric data type of the database (although it may still recognize and use some internal types). For this purpose every entity in the database is associated with an *id* number (a non-negative integer) and every attribute of every entity is accessible (for read and write operations) through appropriate C macros. All the macros have the same form: `ATTRIBUTE(id)` and `SET_ATTRIBUTE(id,value)`. Global properties which do not depend on any id are accessible through macros of the form `PROPERTY` and `SET_PROPERTY(value)`. For example, fetching the x-coordinate of the ith vertex in the database is performed by the macro `X(i)`, while setting its value to w is performed by `SET_X(i,w)`. The current number of facets, for example, is always

accessible through the macro `N_FACETS`. Nested macros are also permitted, e.g., `SET_X(i,Z(j))`. A complete listing of the available macros is given in Section 2.

The second goal is the ease of communication with external applications and devices. For this purpose we chose the 'file-interface' approach. Our system is capable of loading and saving files in a variety of ascii and binary formats. These file formats are recognized by most commercial CAD systems and by some packages used by the computational geometry community. Communication with graphics packages is also performed by using files. For example, on-line graphics display is available through the Geomview package[21] by using a pipeline into which our system writes ascii data. (This mechanism also serves for off-line display.)

We note that the DCEL system, at least in its current released form, does not contain robustness mechanisms for ensuring correct execution of geometric algorithms with degenerate inputs. However, any external mechanism can easily be adopted into the system.

The rest of this paper contains a detailed description of the DCEL system. We describe the concepts on which it is built and the tools it provides, and demonstrate how they allow easy and fast implementation (or, rather, prototyping) of geometric algorithms that fit into the polyhedral context. We survey the features of the system and supply a few implementation examples.

The paper is organized as follows. In Section 2 we describe the abstract data structures supported by the polyhedral database and how to access them. In Section 3 we detail the kernel operations of the system. We overview in Section 4 the toolkit functions provided by the programming environment. In Section 5 we sketch some of the algorithms that have been implemented on top of our system. We outline in Section 6 a few possible extensions of the system. We conclude in Section 7 with some closing remarks.

2. Basic Data Structures

The name of the package, DCEL, stands for Doubly-Connected Edge List, suggested by Preparata and Shamos[22] for the description of polyhedral 2-manifolds. This simple data structure is based on a list of edges, where each edge e is characterized by its two endpoints (vertices), the two (oriented) faces f_1 and f_2 that share e, and the two edges that follow e along f_1 and f_2. Guibas and Stolfi[15] give a formal definition of a similar data structure (so-called *quad-edge*) that describes simultaneously the graph of the polyhedron boundary, its dual graph, and its mirror graph. They have used only two basic operations: one for the creation of an isolated edge, and the other (so-called *splice*) for connecting or disconnecting an edge from the graph. As the authors mention, the quad-edge data structure is a variant of the *winged-edge* data structure of Baumgart[6] (also described elsewhere[7,12]). Dobkin and Laszlo[11] generalize the quad-edge data structure to 3-manifolds; Brisson[8] generalizes it later to n-manifolds.

The basic entities of our database are vertices, edges, polygons, and facets. In our implementation, a polygon of the polyhedron boundary should be oriented clockwise when it is viewed from outside the polyhedron. In addition, the database

supports connected components of the manifold and some storage of general information.

The DCEL system provides the following macros that access data of vertices: `X(id)`, `Y(id)`, `Z(id)`, and `RANK(id)`. (The rank of a vertex is the number of edges incident to the vertex.) The respective modifying macros are `SET_X(id,value)` and so forth. Valid id numbers are in the range 0 through `N_VERTICES-1`. The pseudo variable `N_VERTICES` is a macro that always returns the current number of vertices. Optionally, all the macros deny access to vertices with illegal id numbers, thus they also serve as a debugging tool. Turning off the id validity-checks (at compilation time) speeds up run-time macro execution.

The edge is the central entity of the database. The macros used for accessing an edge are `ORIGIN(id)`, `TERMINUS(id)`, `CARDINALITY(id)` (number of parallel edges sharing the same pair of endpoints), `PREV(id)`, `NEXT(id)`, `OPPOSITE(id)`, `ANGLE(id)` (the dihedral angle between a pair of facets that share an edge with cardinality 2), `FACET_NUM(id)`, `WINDOW_NUM(id)`, and `POLYGON_NUM(id)` (see below). The macro `N_EDGES` returns the current number of edges, so that valid edge id numbers are in the range 0 through `N_EDGES-1`.

In our implementation we maintain a separate entity for each edge in a polygon. Therefore, a regular edge of a polyhedron corresponds to two opposite edge entities (both having cardinality 2). The cardinality of an edge on a border of a 2-manifold is 1, and the edge is the opposite of itself. In case two (or more) portions of the manifold are tangent in an edge, the cardinality of all the corresponding entities is larger than 2 and the 'opposite' relation is a cyclic operator that defines a ring of edges.

We allow a facet of the polyhedron to contain windows (holes), so that it may not be simply connected. Hence, we represent each facet by a boundary polygon and zero or more window polygons. For each facet we keep the (polygon) id of the facet boundary polygon, the number of windows it encloses (zero or more), and the normal to the plane that supports the facet. These are accessed through the macros `BOUNDARY(id)`, `N_WINDOWS(id)`, and `NORMAL(id)`. The three components of the normal vector can be accessed directly by `NORMAL_X(id)`, `NORMAL_Y(id)`, and `NORMAL_Z(id)`. The macro `N_FACETS` returns the current number of facets, so that valid facet id numbers are in the range 0 through `N_FACETS-1`.

For each polygon structure we keep its complexity (number of vertices, denoted as the "length" of the polygon) and the id of one of its edges. These are accessed through the macros `LENGTH(id)` and `EDGE_PTR(id)`. We also maintain a linkage between all the polygons (boundary and windows) of a facet: each polygon structure contains the id of the next window in a linked-list. The first polygon is the boundary, and the last window does not point to any other polygon. This field is accessed through the macro `NEXT_WINDOW(id)`. The macro `N_POLYGONS` returns the current number of polygons, so that valid polygon id numbers are in the range 0 through `N_POLYGONS-1`.

General data is also maintained. The current implementation supports the name and title of the object, creation and reception date, the names of the CAD system

and the operating system (if applicable), name and details of the (human) originator or designer of the object, the units in which the object is specified, etc. General geometric information is also available, e.g., the number of connected components the object consists of, the surface area of the object and its volume (when the object is closed), etc. In addition, there are global flags that indicate whether the normals (to facets) are computed (this information may not be updated continuously through the course of an algorithm), whether the surface fully encloses a volume, etc. Other pieces of information are subject to the interpretation of each application. These include a password and a protection level, general comments, etc. All the general information fields can be accessed through appropriate macros. Each general data field named, say, `item` is accessible through the macros `ITEM` and `SET_ITEM(value)`. The value argument must be of the appropriate type of the data field, otherwise a compilation error occurs.

In addition, the DCEL system supports user-defined fields. For each entity type the system maintains few general-purpose fields of all the basic C types. Polygon fields, for example, are named `p_int_field_1`, `p_double_field_2`, etc. These general-purpose fields may be referred to by using the macros `P_INT_FIELD_1(id)`, `SET_P_INT_FIELD_1(id,value)`, etc. In order for an application to reset the field `p_int_field_1` to `cw` (for indicating whether polygons are oriented correctly) it needs only to define the macros:

```
#define CW(_p)              P_INT_FIELD_1 (_p)
#define SET_CW(_p, _value) SET_P_INT_FIELD_1 (_p, _value)
```

This allows the application to have code of the form:

```
SET_CW (p1, TRUE);
if (! CW (p2))
   DCEL_invert (p2);
```

3. Kernel Operations

In this section we describe the kernel operations of the DCEL system.

3.1. File Operations

Since every system lacks more than it contains, it should communicate with other systems which may provide it with more functionality. It is therefore good software-engineering practice for a system to have the ability to share data with its environment. We chose an easy communication mechanism through files with agreed-upon format. File interface is much easier to implement and to use than a functional interface because the latter is more sensitive to the platform, the operating system, and other environmental conditions, and because syntax semantics is usually defined more strictly than that of a functional interface.

Here are the main file formats supported by our system:

OFF A very simple ascii file format used by the package Geomview.[21] This file format first contains an indexed list of vertices, then a list of polygons, each consists of a list of references (indices) to the vertex list.

QUAD Another ascii file format used by Geomview. This format supports quadrilaterals only and does not contain a list of vertices: each quadrilateral is specified by the coordinates of its vertices.

MATH (write only) An ascii file format used by the Mathematica package.[27]

GASP (write only) A file format which consists of a sequence of calls to the GASP[26] functional interface. A separate off-line interpreter processes files of this type and displays them by using GASP.

STL An ascii file format[28] which is the de-facto standard of the model rapid-prototyping industry. (STL is an acronym for stereolithography, the leading technology in this industry.) This format contains triangles only, and it is recognized by most commercial CAD systems.

BSTL The binary form of the STL file format.

SLC An ascii file format used for describing a series of parallel polygonal slices, where each slice contains any number of polygons with an arbitrary hierarchy of containment. It is used mainly for medical imaging and for GIS applications.

DTM (read only) An ascii format used for describing a digital terrain model as a list of height measurements (z coordinates) taken over a regular xy-grid.

DCEL A binary-dump format of the DCEL database. It enables extremely fast I/O operations, since it simply maps the database dynamic memory into disc space. Except for data transfer no parsing operations or any other computations are performed during reading or writing the data.

The ability to read and write files in various formats allows the user to convert files between them. For example, the following C code (that does not check for errors) converts a BSTL (binary STL) file to an OFF (Geomview's) file:

```
DCEL_load (in_file_name, DCEL_BSTL);
DCEL_save (out_file_name, DCEL_OFF);
```

In case the target format is limited, the appropriate adjustments are performed automatically. Thus the contents of the database may be triangulated, facet windows may be eliminated (by adding "bridges" for splitting every multi-connected facet into several simply-connected polygons), or the normal vectors may be recalculated (when the output file format requires this information).

3.2. Atomic Operations

3.2.1. Entity operations

The DCEL system features the following entity operations:

1. Adding (or deleting) a single vertex, edge, polygon, or a facet to (from) the database. A suitable interface allows these operations also on sequences of new (or old) entities. In fact, deleted entities are only *marked* for deletion but are actually removed only when the database is "condensed" (see Section 3.2.2). Before doing that every "deleted" entity can be recovered.

2. Bypassing a vertex in a polygon. Let v_1, v_2, v_3 be three consecutive vertices along some polygon p. Bypassing v_2 in p means connecting v_1 and v_3 by an edge that replaces the edges from v_1 to v_2 and from v_2 to v_3.

3. Splitting an edge into two edges. Splitting an edge $e = \overrightarrow{pq}$ at some point t means replacing e by the two edges $\overrightarrow{pt}$ and $\overrightarrow{tq}$ and by replacing the opposite edge $\overrightarrow{qp}$ (if it exists) by the two edges $\overrightarrow{qt}$ and $\overrightarrow{tp}$.

4. Discretizing an edge. This operation is performed by repeatedly splitting the edge at equally spaced points along the edge.

5. Splitting a polygon into two polygons. The polygon $p = (v_1, \ldots v_{i-1}, v_i, v_{i+1}, \ldots, v_{j-1}, v_j, v_{j+1}, \ldots, v_n)$ is split from v_i to v_j (two non-consecutive vertices along p) by adding the two oppositely oriented edges connecting v_i and v_j to the database and by replacing p by the two polygons $(v_1, \ldots, v_{i-1}, v_i, v_j, v_{j+1}, \ldots, v_n)$ and $(v_i, v_{i+1}, \ldots, v_{j-1}, v_j)$.

6. Simplifying non-simply-connected polygons by eliminating loops. Simplifying the polygon $(v_1, \ldots, a, s, b, \ldots, c, s, d, \ldots, v_n)$ at s is performed by replacing it by the two polygons $(v_1, \ldots, a, s, d, \ldots, v_n)$ and $(s, b, \ldots, c)$.

7. Merging two polygons into one. This operation has two modes:

   (a) Removing two oppositely oriented edges that the two polygons share. Namely, two polygons $p_1 = (v_1, \ldots, a, s, t, b, \ldots, v_n)$ and $p_2 = (u_1, \ldots, c, t, s, d, \ldots, u_m)$ are merged at $\overline{st}$ by removing the two oppositely oriented edges $\overrightarrow{st} \in p_1$ and $\overrightarrow{ts} \in p_2$ and by replacing the two polygons p_1 and p_2 by the polygon $(v_1, \ldots, a, s, d, \ldots, u_m, u_1, \ldots, c, t, b, \ldots, v_n)$.

   (b) Adding a "bridge" of two oppositely oriented edges between vertices of the polygons (one of each) and merging the sequences of vertices by using that "bridge". Namely, two polygons $p_1 = (v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n)$ and $p_2 = (u_1, \ldots, u_{j-1}, u_j, u_{j+1}, \ldots, u_m)$ are connected between v_i and u_j by adding the two oppositely oriented edges $\overrightarrow{v_i u_j}$ and $\overrightarrow{u_j v_i}$ to the database and by replacing the two polygons p_1 and p_2 by the polygon $(v_1, \ldots, v_{i-1}, v_i, u_j, u_{j+1}, \ldots, u_m, u_1, \ldots, u_{j-1}, u_j, v_i, v_{i+1}, \ldots, v_n)$.

   Note that coplanarity of the merged polygons in not checked.

8. Inverting a polygon. A polygon $(v_1, v_2, \ldots, v_{n-1}, v_n)$ is inverted by replacing it by the polygon $(v_n, v_{n-1}, \ldots, v_2, v_1)$.

3.2.2. Database operations

The DCEL system features the following global operations:

1. Printing the contents of the database to the standard output in a legible form. This operation has several levels of detail controlled by the application.
2. Condensing the contents of the database. This operation frees all the memory allocated for "deleted" entities which cannot be recovered any more.
3. Erasing the entire contents of the database. The erased data are lost and cannot be recovered.
4. Recomputing all the connectivity information in the database. That is, recomputing all the neighborhood relations between polygons, reflected by pairs of opposite edges that point at each other. (This information may not always be updated continuously for efficiency purposes.)
5. Recomputing the ranks of all the vertices.
6. Recomputing the axis-parallel bounding-box of the object.

3.3. Iterators

The DCEL system provides several iterators for all the entities of some type (vertices, edges, polygons, or facets), for all the vertices (or edges) of some polygon, etc. Assume that an application wishes to print the following line:

```
The neighbors of polygon 17 are: 12 5 19 32
```

(Given that the 17th polygon is a quadrilateral and that the neighboring polygons are indeed 12, 5, 19, and 32, in this order.) The application may accomplish this task by invoking the macro `LOOP_POLYGON_EDGES` as follows:

```
int e, p = 17;
printf ("The neighbors of polygon %d are:", p);
LOOP_POLYGON_EDGES
   (p, e, printf (" %d", POLYGON_NUM (OPPOSITE (e))););
printf ("\n");
```

The actual implementation of the macro `LOOP_POLYGON_EDGES` provided by the DCEL system is:

```
#define LOOP_POLYGON_EDGES(_poly, _edge, _command) { \
   int __i, __l; \
   for (_edge = EDGE_PTR (_poly), __l = LENGTH (_poly), __i = 0; \
        __i < __l; _edge = NEXT (_edge), __i ++) { \
      _command \
   } \
}
```

The argument `_command` of the macro may be any valid (grammatically correct) piece of C code. Nested macros are also allowed. The following piece of code prints the above line for all the polygons:

```
int e, p;
LOOP_POLYGONS (p,
               printf ("The neighbors of polygon %d are:", p);
               LOOP_POLYGON_EDGES (p, e,
                  printf (" %d", POLYGON_NUM (OPPOSITE (e))););
               printf ("\n");
              );
```

The macro `LOOP_POLYGONS` provided by the DCEL system is:

```
#define LOOP_POLYGONS(_poly, _command) { \
   for (_poly = 0; _poly < N_POLYGONS; _poly ++) { \
      _command \
   } \
}
```

The following piece of code is part of an implementation of a DFS on the dual graph of the boundary of a polyhedron. Its task is to orient all the neighbors of some polygon `p` consistently with it. In case some neighbor polygon `_p1` was already visited in the DFS, its consistency with `p` is checked; otherwise `_p1` is oriented consistently with `p` and is pushed into the DFS stack. The implementation of the user-defined macros `VISITED(id)` and `SET_VISITED(id,value)` is explained in Section 2.

```
int e, p;
pop (p);
LOOP_POLYGON_EDGES (p, e,
                    int _ok, _e1, _p1;
                    _e1 = OPPOSITE (e);
                    _p1 = POLYGON_NUM (_e1);
                    _ok = (ORIGIN (e) == TERMINUS (_e1)) &&
                          (TERMINUS (e) == ORIGIN (_e1));
                    if (VISITED (_p1))
                       if (! _ok) {
                          printf ("Boundary is non-orientable!\n");
                          return (FALSE);
                       } else;
                    else {
                       if (! _ok)
                          DCEL_invert (_p1);
                       push (_p1);
                       SET_VISITED (_p1, TRUE);
                    }
                   );
```

3.4. Multiple Databases

The DCEL system allows the user to simultaneously use multiple databases. This feature is useful for divide-and-conquer algorithms and for operations whose

input and output need to be stored separately, e.g., in performing boolean operations. In addition, it allows version control of an iterative algorithm, e.g., in a mesh refinement process, where the user may step forward and backward between various versions of the same object with different levels of refinement.

There is always an *active* database on which the operations are performed. The default number of databases is 1 and then the id of the active (single) database is 0. The databases are numbered from 0 to $n-1$, where n is the number of currently available databases. The application may alter the number of databases or change the active database at any time. When the application increases the number of databases, additional empty databases are created. When the number of databases decreases, say, from n_1 to n_2 (where $1 \leq n_2 \leq n_1$), the highest $n_1 - n_2$ databases are deleted and their contents are lost. In case the currently active database is deleted, the application needs to specify the new active database. In the special case of decreasing the number of databases from n to 1, the application may choose to make the currently active database (rather than the 0th database) the single remaining database. The application may always switch the id numbers of two existing databases.

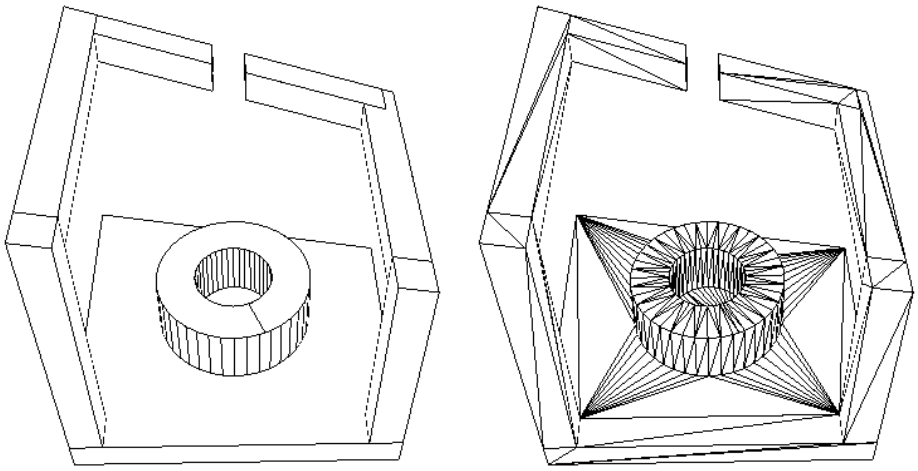
The databases may be totally disjoint or they may share data, e.g., the set of vertices. The system allows the user to copy or move entities from one database to another. These operations either append the new entities to the contents of the target database or overwrite them, but the active database does not change. The system ensures that an entity shared by several databases is not deleted as long as at least one database refers to it. Alternatively, the application may define one database as the *owner* of some entity type (e.g., vertices), so that entities of this type are removed only when the owner database is deleted.

The following is a typical application of multiple databases. The application triangulates the boundary of a polyhedron. It creates two databases that share the set of vertices. The polyhedron is loaded into the first database, then each of its facets is triangulated, and the newly created triangles are put in the second database. Finally, the second database becomes active and its contents are saved in a file. Fig. 1 shows the results of such as application. The original polyhedron is shown in Fig. 1(a), while the triangulated version of it is shown in Fig. 1(b).

3.5. Other Services

Our system provides a package for measuring the Euclidean distance between pairs of the basic geometric entities. An entity may be intrinsic (a vertex, an edge, a polygon, or a facet of the polyhedron boundary—all given by id numbers) or extrinsic (a point (specified by its three coordinates), a segment (defined by the coordinates of its two endpoints), or a polygon (an ordered list of points)).

The system also allows the application to measure time intervals (elapsed or CPU time). The application may define, reset, and take measurements of multiple "stop-watches" (timers). The resolution of each timer is defined either in seconds or micro-seconds. This mechanism is actually a friendly interface to the standard Unix functions `clock()`, `time()`, etc.



(a) Before triangulation (b) After triangulation

Fig. 1. Boundary triangulation

4. Toolkit Functions

Our system provides a set of toolkit functions built on top of its kernel. This toolkit includes some algorithms which are often used as building-blocks of real applications. Visualization means are also provided by the DCEL system. We mention below a few of these functions.

Range searching. We have implemented a mechanism for 3-dimensional orthogonal range searching, in which the input is a set of points in space, and a query looks for all the points contained by some axis-parallel box. We have used a simple heuristic projection method, which projects the vertices onto each of the axes and sorts them along each axis. Given a query box, we also project it onto each axis, retrieve the three subsets of vertices, each being the set of vertices whose projections fall inside the projected box on one of the axes, choose the subset of smallest size, and test each of its members for actual containment in the query box. While this method may be inefficient in the worst case (requiring linear time per query), it works very well in practice. (Better asymptotic query time can be achieved by using the *range tree* data structure,[18] or by *fractional cascading*.[9])

Window elimination. We have implemented a "window-elimination" function for converting a perforated facet f into a simply-connected facet f'. First we construct a complete graph G in which each polygon (boundary or window) of f is represented by a vertex. The length of an edge of G is set to the Euclidean distance between the two polygons corresponding to the endpoints of the edge. Then we compute a minimum spanning-tree T of G by using the well-known algorithm of Prim.[23] Finally, we convert every edge in T to a "bridge" (a pair of oppositely oriented edges) which is added to f. No bridge intersects any of the polygons of f; otherwise it could not belong to a minimum spanning-tree of T. Let w be the

number of windows in f. The graph G contains $w+1$ vertices, thus T contains w edges. Since the minimum distance between two polygons in the plane always occurs at a vertex (or vertices) of at least one polygon, at most w new vertices are added. The number of new edges is always $2w$. The output facet f' contains only a boundary polygon. It touches itself along w pairs of edges but otherwise it is simply-connected.

Triangulation. Our system supports the triangulation of 3D polygons.[a] That is, the triangulation of closed spatial chains of straight segments. Although the spatial polygons are expected to lie *more or less* in the same plane, we do not make use of this assumption. Since every platform is limited in the precision of real numbers, the vertices of a general polygon in three dimensions (except for triangles) are practically never collinear. In addition, data originated by CAD systems as polyhedral approximations of smooth surfaces frequently contain non-planar 3D polygons. We closely follow the dynamic-programming triangulation technique of Klincsek.[16] Our system provides a few built-in objective functions (such as minimum area, minimum sum of edge lengths, and max-min angle), and also supports user-defined functions. Although the dynamic-programming method is asymptotically very inefficient (cubic in the complexity of the polygon), it still runs in practical time for reasonably complex polygons (up to a few seconds for a polygon of approximately 100 vertices on an SGI Indigo workstation). For coping with polygons with larger complexities we plan to implement a faster algorithm which actually triangulates the projection of the polygon onto a plane with the best least-squares fit with respect to the vertex set of the polygon.

Visualization. The DCEL system supports on-line visualization of the database contents through the Geomview and the GASP systems. A DCEL application may either display the entire contents of the database at any time (in this case the new display replaces the old one) or may display a specific entity (which is added to the display). The communication with Geomview is performed by using a pipeline. This is a stream file into which DCEL continuously writes data which are then read by Geomview. These data are written in the OFF and QUAD file formats.[21] The communication with GASP, however, is performed by using its functional interface. The DCEL system may be linked with the GASP library, which includes functions for displaying many geometric primitives. The GASP graphics window is opened as a separate process, to which the DCEL system transmits the displayed data through a mailbox communication channel. Both Geomview and GASP can capture snapshots of the graphics display into PostScript files.

Coherence tests. The DCEL system supplies a self-testing mechanism which checks whether the data contained in the database are consistent. This feature includes the following tests:

[a] The triangulation of such polygons is practical in the context of some algorithms.[3,4,5]

(i) Whether an entity referred to by another entity (e.g., a vertex by an edge, a polygon by a facet, etc.) actually exists.
(ii) Whether the relations between edges (next/previous, opposite) are consistent.
(iii) Whether all the vertices that belong to the same facet are coplanar (within some tolerance).
(iv) Whether the polygons are simple, and whether the boundary polygons of perforated facets fully contain their window polygons.

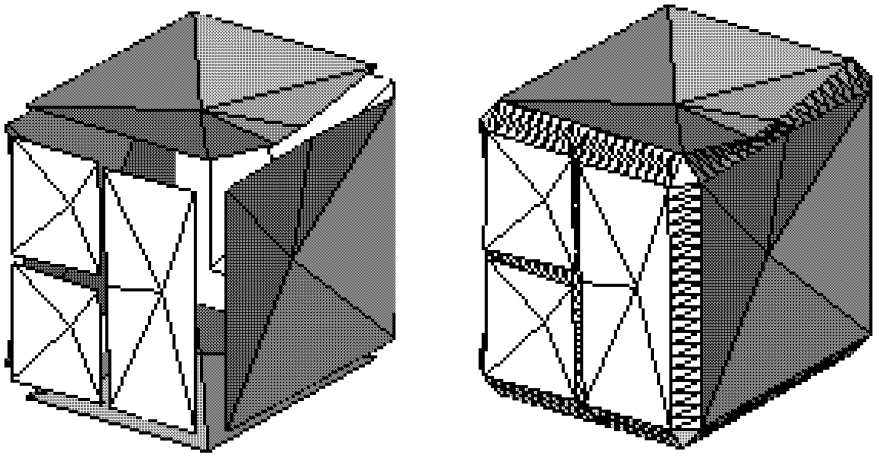
Analysis. The DCEL system can analize the polyhedral mesh according to the connectivity information it gathers beforehand. The analysis is based on a depth-first search performed on the dual graph of the polyhedral surface, where each polygon is represented by a vertex, and each pair of neighboring polygons is represented by an edge that connects the respective dual vertices. Polygons that belong to the same facet are also considered as neighbors. The analysis output consists of a list of the connected components of the surface, an indication for each component whether it is open or closed, the boundary polygons of the open components (so-called *borders*), statistics, and some topological information for each connected component: an indication whether it is orientable or not, and if so, which polygons, if any, are oriented incorrectly, etc.

5. Implemented Algorithms

We now overview a few of the algorithms implemented by using the DCEL system. (These algorithms are not part of the system.)

Gap filling. In this problem the input is a "defected" polyhedral object whose boundary is broken by long thin cracks, and the goal is to "repair" the object so as to create a valid solid polyhedron. Such erroneous polyhedra are often produced by current commercial CAD systems when they approximate curved objects (built from smooth surfaces or solids) by polyhedra. This problem spoils the interfacing between CAD systems and external systems which require topologically-correct polyhedral input. An algorithm[4] for detecting and repairing these defects was implemented by using the DCEL system. The algorithm uses a partial curve matching technique for matching parts of the defects, and an optimal triangulation of 3D polygons for resolving the unmatched parts. Figs. 2(a,b) show a synthetic object whose boundary is broken, before and after the repairing process, respectively.

Inter-slice interpolation. In this problem the input consists of a series of polygonal (parallel) cross-sections of some unknown 3-dimensional object, and the goal is to reconstruct the object by computing a polyhedron whose intersections with the parallel planes that contain the input slices coincide with these slices. This is an important tool in understanding and reconstructing organs out of data obtained by medical scanners (such as CT, MRI, PET, etc.), and it is also useful for surface reconstruction from topographic data, as well as for other similar purposes. The



(a) Before curing (b) After curing

Fig. 2. Gap filling

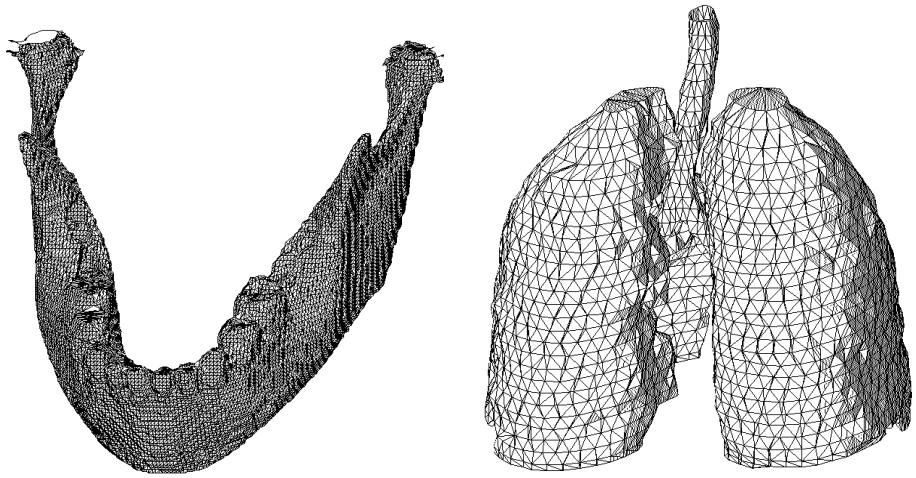
implementation of a new technique[5] for the reconstruction of an object from a series of parallel cross-sections uses the DCEL system. Similarly to the gap filling, the algorithm uses a partial curve matching technique for matching parts of the contours, an optimal triangulation of 3-dimensional polygons for resolving the unmatched parts, and a minimum spanning tree heuristic for interpolating between non simply-connected regions. This algorithm seems to handle successfully in practice any kind of data. It allows multiple contours in each slice with any hierarchy of contour nesting. The reconstructions of a jaw bone and a pair of lungs are shown in Figs. 3(a,b).

Collision detection. The *boxtree*[2] is a versatile data structure for representing triangulated or meshed surfaces in 3D. A boxtree is a hierarchical structure (a binary tree) of nested blocks (arbitrarily-oriented boxes or prisms with triangular cross-sections) that supports efficient ray tracing and collision detection. It is simple and robust, and requires minimal space. This hierarchy is applied for hit-detection between polyhedra. This data structure was implemented by using the DCEL system.

6. Future Extensions

We plan to implement the following extensions of the DCEL system:

1. Extend the toolkit library by functions for computing the convex hull of a set of 3D points, boolean operations on polyhedra, etc.
2. Implement robustness mechanisms. This aspect is currently totally ignored.
3. Implement additional algorithms on top of the system.



(a) Jaw bone (b) Lungs

Fig. 3. Fully reconstructed human organs

4. Implement a C++ version of the system.

7. Conclusion

We present in this paper the DCEL system, a polyhedral database and programming environment for fast prototyping of geometric software. Although it is modest in scale, we believe that the system provides a convenient ground for swift and easy implementation of software without concern about the data structures, input/output operations, visualization, etc. The primary goal of our system is to let the application concentrate on the *algorithm* rather than in the *representation* through software data types. This goal is achieved by providing an easy interface which requires a very short learning phase. We believe that this is an important factor in choosing an infrastructure package for the implementation of an efficient and reliable software.

The package is available in the public domain. A preliminary version is found at `ftp://ftp.cs.jhu.edu/pub/barequet/dcel` (a compressed `tar` file) and at a mirror site `ftp://ftp.math.tau.ac.il/pub/barequet/dcel`.

Acknowledgment

The author wishes to thank Micha Sharir for many helpful discussions of the DCEL system over the past several years. Work on this paper has been supported in part by the Israeli Ministry of Science and the Arts under Eshkol Grants 0562-1-94 and 0562-2-95.

References

1. G. Barequet, "Applications of geometric hashing to the repair, reconstruction, and matching of three-dimensional objects," Ph. D. Thesis, School of Mathematical Sciences, Tel Aviv University, Israel, 1994.
2. G. Barequet, B. Chazelle, L. J. Guibas, J. S. B. Mitchell, and A. Tal, "BOXTREE: A hierarchical representation for surfaces in 3D," *Proc. Eurographics*, (eds. J. Rossignac and F. Sillion), Futureoscope-Poiliers, France, Aug. 1996; *Computer Graphics Forum*, **15** (1996) C387–396.
3. G. Barequet, M. Dickerson, and D. Eppstein, "On triangulating three-dimensional polygons," *Proc. 12th Annual ACM Symposium on Computational Geometry*, Philadelphia, PA, May 1996, pp. 38–47.
4. G. Barequet and M. Sharir, "Filling gaps in the boundary of a polyhedron," *Computer-Aided Geometric Design*, **12** (1995) 207–229.
5. G. Barequet and M. Sharir, "Piecewise-linear interpolation between polygonal slices," *Proc. 10th Annual ACM Symposium on Computational Geometry*, Stony Brook, NY, June 1994, pp. 93–102; full version: *Computer Vision and Image Understanding*, **63** (1996) 251–272.
6. B. G. Baumgart, "A polyhedron representation for computer vision," in *Proc. National Computer Conference* (AFIPS Press, 1975) pp. 589–596.
7. I. C. Braid, R. C. Hillyard, and I. A. Stroud, "Stepwise construction of polyhedra in geometric modelling," in *Mathematical Methods in Computer Graphics and Design*, ed. K. W. Brodlie (Academic Press, London, 1980) pp. 123–141.
8. E. Brisson, "Representing geometric structures in d dimensions: Topology and order," *Proc. 5th Annual ACM Symposium on Computational Geometry*, Saarbrücken, West Germany, June 1989, pp. 218–227.
9. B. Chazelle, "A functional approach to data structures and its use in multidimensional searching," *SIAM J. of Computing*, **17** (1988) 427–462.
10. B. Chazelle et al. (37 coauthors), "Application challenges to computational geometry," Technical Report TR-521-96, Princeton University, 1996 (available at `http://www.cs.princeton.edu/~chazelle/taskforce/CGreport.ps.Z`).
11. D. P. Dobkin and M. J. Laszlo, "Primitives for the manipulation of three-dimensional subdivisions," *Proc. 3rd Annual ACM Symposium on Computational Geometry*, Waterloo, Ontario, Canada, June 1987, pp. 86–99.
12. C. M. Eastman and K. Weiler, "Geometric modeling using the Euler operations," Research Report 78, Inst. of Physical Planning, Carnegie-Mellon University, 1979.
13. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr, "The CGAL kernel: A basis for geometric computation," *1st ACM Workshop on Applied Computational Geometry*, Philadelphia, PA, May 1996.
14. G.-J. Giezeman, "PLAGEO/SPAGEO: A collection of geometric C++ objects" (software available at `ftp://archive.cs.ruu.nl/pub/SGI/GEO`).
15. L. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams," *ACM Trans. on Graphics*, **4** (1985) 74–123.
16. G. T. Klincsek, "Minimal triangulations of polygonal domains," *Annals of Discrete Mathematics*, **9** (1980) 121–123.
17. D. T. Lee, C.F. Shen, and S. M. Sheu, "GeoSheet: A distributed visualization tool for geometric algorithms," *Int. J. of Computational Geometry and Applications*, (8,2) 1998, to appear.

18. K. Mehlhorn, *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*, eds. W. Brauer, G. Rozenberg, and A. Salomaa (Springer-Verlag, Berlin, 1984).

19. K. Mehlhorn and S. Näher, "LEDA—A library of efficient data types and algorithms," *Lecture Notes in Computer Science 379* (Springer-Verlag, Berlin, 1989), pp. 88–106.

20. K. Mehlhorn and S. Näher, "LEDA—A platform for combinatorial and geometric computing," *Communications of the ACM*, **38** (1995) 96–102 (software available through `http://www.mpi-sb.mpg.de/LEDA/leda.html`).

21. T. Munzner, S. Levy, and M. Philips, "Geomview: A system for geometric visualization," *Proc. 11th Annual ACM Symposium on Computational Geometry*, Vancouver, British Columbia, Canada, June 1995, pp. C12–13 (software available through `http://www.geom.umn.edu/software/download/geomview.html`).

22. F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, (Springer-Verlag, New York, 1985).

23. R. C. Prim, "Shortest connection networks and some generalizations," *Bell Systems Technical Journal*, **36** (1957) 1389–1401.

24. P. de Rezende, "GeoLab: An environment for development of algorithms in computational geometry," *Int. Computational Geometry Software Workshop*, Minneapolis, MN, Jan. 1995 (software available at `ftp://ftp.geom.umn.edu/pub/contrib/comp_geom/geolab/geolab.tar.Z`).

25. P. Schorn, "The XYZ GeoBench: A programming environment for geometric algorithms," *Lecture Notes in Computer Science 553* (Springer-Verlag, Berlin, 1991), pp. 187–202 (software available through `http://nobi.ethz.ch/group/projects.html`).

26. A. Tal and D. P. Dobkin, "Visualization of geometric algorithms," *IEEE Trans. on Visualization and Computer Graphics*, **1** (1995) 194–204 (software available at `ftp://ftp.cs.princeton.edu/pub/people/ayt/gasp.tar.Z`).

27. S. Wolfram, *Mathematica™: A System for Doing Mathematics by Computer*, (Addison-Wesley, 1988).

28. 3D Systems, Inc. (Valencia, CA), "Stereolithography Interface Specification," p/n 50065-S01-00, 1989.