D. D. Grossman

Procedural Representation of Three-dimensional Objects

Abstract: A system of PL/I procedures has been written that permits geometric objects to be described hierarchically. The objects are themselves represented as PL/I procedures, allowing very general use of variables. By effectively intercepting subprogram calls, the system provides a means of modifying the semantics associated with any object without modifying the object's procedural description.

Introduction

Substantial prior literature exists on methods of modeling geometric objects. In the field of artificial intelligence, Gelernter [1] represented two-dimensional figures in the form of list structured numerical data. This representation was used to generate proofs of theorems in plane geometry. Roberts [2] tackled problems involving the three-dimensional scene analysis of polyhedra. Binford [3] developed an alternative representation in terms of "generalized cones." In a different vein, the research of Winston [4] was concerned with representing the structural rather than the spatial relationships in scenes. Recently, Baumgart [5] has developed a topological representation for arbitrary polyhedra that is particularly well suited to constructing complex polyhedra through a range of operations including set union and intersection.

The field of graphics is concerned principally with the appearance of geometrical objects. The high level of complexity of three-dimensional objects that can be represented is exemplified by the architectural modeling work of Greenberg [6]. The modeling routines of Appel [7] allowed component parts of an assembly to be juxtaposed. Some systems have the capability of generating pictures of articulated mechanisms. For example, in the system described by Csuri [8], simple macros could be written to draw an airplane with retractable landing gear and wheels that turned. Catmull [9] modeled the articulations of the human hand. However, representations that are specifically suited for describing the appearance of objects tend not to generalize easily to other applications. In particular, such systems cannot easily count the number of windows on the second floor of a building, compute their total surface area, or determine the height of an aircraft's wheels above a runway.

Another facet of graphics consists of systems intended for computer assisted design, and the literature on this topic is extensive [10]. One of the earliest attempts at designing three-dimensional objects was due to Luh [11]. From a description in the form of quartic bounding surfaces, this system could draw a simple object, compute its center of mass and moments of inertia, and even generate a numerical control tape to manufacture the object. Comba [12] worked on a geometry language suited to determining whether or not two objects interfered with each other. A system intended for architectural design applications has been described by Wehrli [13]. This system included a language called SPACEFORM, which contained a set of primitive solids. Two new design systems have been developed. The first, due to Braid [14], is based on a set of six primitive solids, i.e., cuboid, wedge, tetrahedron, cylinder, sector, and fillet. The second, due to Voelcker [15], involves a language called PADL, which is currently based on the cuboid and cylinder primitives. In both systems highly complex objects are constructed as hierarchical combinations of more primitive objects, under the operations of set union and intersection. A unique aspect of PADL is its ability to describe tolerances of mechanical parts. Each PADL program is transformed by a parser-interpreter into an object file, which may, in turn, be accessed by output programs. Currently, the only output program is one that generates graphics files for display purposes.

Another field that relates to parts description is the programming of numerically controlled machine tools [16]. The most widely used language for numerical control is APT. Each APT program is a representation of the object that the program can physically generate. The

APT compiler transforms the source program into a tool path sequence, which can drive a numerically controlled machine tool. The compiler, therefore, is a program that derives information about an object directly from its APT representation. The complexity of this compiler is an indication, however, that it is a formidable programming job to extract information from an APT source program in this manner. For example, it would be difficult to compute the volume of an object from its APT source program.

This paper reports a new system for describing geometric objects in the form of procedures. The system can model any three-dimensional physical object that can be represented as the sum or difference of subparts, including mechanisms with rotational or linear joints. A significant difference between this approach and that of other procedural representations is the manner in which information about objects is extracted from the representation. In both the PADL and the APT approaches, information can be extracted from object procedures by programs that syntactically process the procedures and translate them into data structures. In the system described here, *the object procedures themselves are actually executed.*

Another difference between this system and the others is the emphasis placed on the use of variables to characterize *arbitrary attributes* of objects, not merely lengths and angles. This emphasis was motivated by the desire to represent objects of generic shape and even objects that are mechanisms.

A good example of a generic shape is the machine screw. Although all machine screws have essentially the same shape, they may be individually classified by head type, material, length, diameter, and pitch. Other examples of generic shapes are washers, nuts, screwdrivers, wrenches, etc. Associated with each such shape is a set of attributes that can be given values to specify a particular instance of that shape. The system described in this paper is designed to allow variables to be used for these attributes, so that an application program can refer, in principle, to objects of the form

```
SCREW(HEAD,MATERIAL,LENGTH,DIAMETER,PITCH)

WASHER(THICKNESS,OUTERDIAMETER,
                                INNERDIAMETER)

TWOBARLINKAGE(ANGLE1,ANGLE2)
```

Application programs need to query the representation to extract data of specific interest. For example an application program might need to know the total volume of an object, or the count of machine screws.

One of the simplest application programs is one that just draws a picture of the object on a graphics output device. This particular application has the pedagogical advantage that it is immediately obvious from the output whether or not the program and the underlying system are working properly. For this reason, the examples given in this paper relate to graphics applications. The reader should appreciate, however, that the system is designed for arbitrary applications and that much of the complexity of the system is due to the need for generality.

System overview

The system for representing geometric objects is written entirely in PL/I. Writing the system in an existing high level language has the great advantage that it is easy to interface the system with application programs written in the same language. On the other hand, one effect of the decision to stay with PL/I is that application programs include a large number of CALLS to system subprograms, whereas in a special purpose language the word CALL could have been eliminated and in many cases generic operators could have replaced the subprogram names. Also, some of the data type declarations in this system could have been made transparent in a special purpose language.

The system consists of three essentially independent modules, as shown in Fig. 1. The CREATE module oversees the hierarchical construction of geometric objects out of their subparts. It is because this routine is called recursively that the entire system was not coded in FORTRAN. The PARTLIB module is a library of routines that represent geometric objects. The COORD module is a package of routines that provides transformations between the coordinate frames of different objects.

Of fundamental importance is the fact that geometric objects are represented as PL/I procedures rather than as numerical data. This representation makes it possible for variables to specify attributes of objects. This representation also has the advantage of having a natural mechanism for describing the hierarchical construction of an object, namely, by calls to subprograms representing the subobjects.

When objects are represented by numerical data in conventional parts description systems, there is no fundamental impediment to writing arbitrary new application programs that may access the data. When objects are represented by procedures, however, these procedures usually have some specific semantic content, i.e., they actually do something. The difficulty that arises is that different application programs frequently require different semantics. For example, an application program that computes the volume of an object needs different semantics from one that generates a drawing of the object.

Clearly, any approach that requires a different representation for each application is totally unsatisfactory.

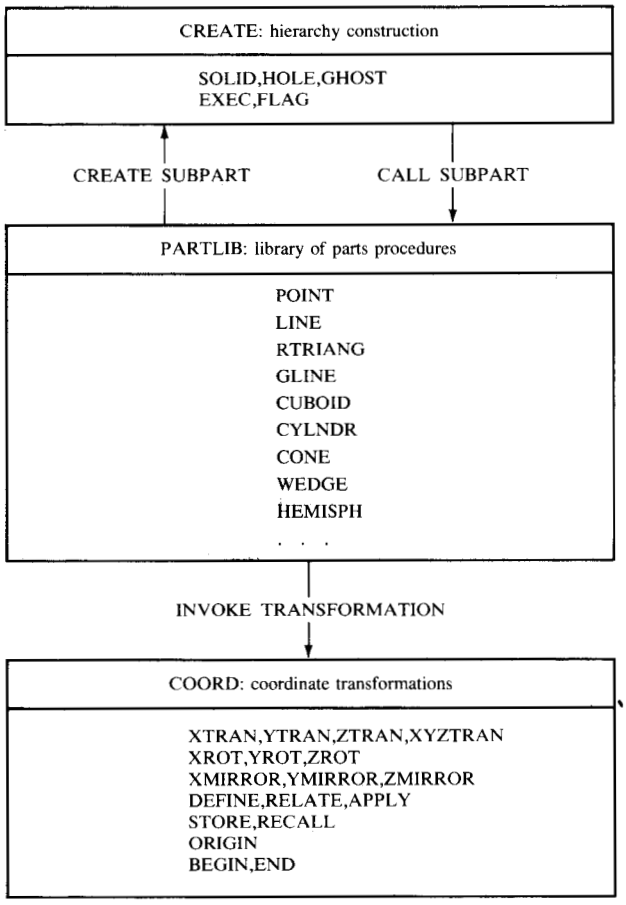

Figure 1 System overview.

Instead, the system described here provides a special means by which application programs may append their own semantics to a given procedural representation of objects. It is this mechanism that renders the representation of objects by procedures feasible, and it is therefore the essential new feature of the system described in this paper.

PARTLIB: library of parts procedures

Every object in the system is represented by an entry in the library of parts, called PARTLIB. In most cases the body of the code for a part consists of calls to other entries that represent subparts of the object. Those part subprograms that do not call subparts are the primitive objects of the system.

In the current implementation, the only primitive object is POINT. The entire PL/I code that represents this primitive is

```
POINT: ENTRY;
RETURN;
```

The only object immediately above POINT in the current implementation is LINE. At this level a variable appears that represents the length of the LINE.

```
LINE: ENTRY(LENGTH);
CALL SOLID(POINT);
CALL XTRAN(LENGTH);
CALL SOLID(POINT);
RETURN;
```

The routine SOLID, which is invoked in the second and fourth statements of this procedure, is in the module CREATE, which is discussed in a later section of this paper. At this point it is sufficient to consider that

```
CALL SOLID(POINT);
```

is superficially equivalent to

```
CALL POINT;
```

The routine XTRAN, which is called in the third statement of LINE, is in the module COORD and translates the coordinate frame by LENGTH in the positive X direction. Thus the LINE procedure says essentially that a LINE of arbitrary LENGTH consists of one POINT at the origin and another POINT a distance LENGTH away along the X axis.

Right triangles may be built out of three LINES, as shown in the procedure

```
RTRIANG: ENTRY(BASE,ALTITUDE);
CALL YTRAN(ALTITUDE);
CALL ZROT(-90);
CALL SOLID(LINE,ALTITUDE);
CALL ZROT(90);
CALL YTRAN(-ALTITUDE);
CALL SOLID(LINE,BASE);
CALL XTRAN(BASE);
CALL ZROT(90+ATAND(BASE,ALTITUDE));
CALL SOLID(LINE,SQRT(BASE**2+ALTITUDE**2));
RETURN;
```

In this manner, successively more complicated objects are constructed hierarchically from simpler ones. At the volume level there are five basic procedures called CUBOID, CYLNDR, CONE, WEDGE, and HEMISPH. At still higher levels, the objects proliferate rapidly and in ever increasing complexity.

With the passage of time, more specialized shapes may be added to the PARTLIB. It might be useful to have such objects as camshaft levers, piston rods, motor blocks, etc. The implementation of the system requires only that each new object have a unique name with at most seven alphanumeric characters. The current implementation ignores all problems related to program management facilities.

In the course of the preceding discussion, the reader has probably become aware of the rather discomforting

fact that in a certain sense the PARTLIB is devoid of semantic content. This fact is particularly clear in the procedure for the POINT primitive, which immediately returns control without doing anything. In the same spirit, since LINE calls two POINTS, it essentially does nothing twice. Nowhere in the PARTLIB is there any semantic identification of POINTS with geometric points, LINES with geometric lines, and so forth. In particular, nowhere in the PARTLIB are there any semantic routines that generate graphics output. How then is any picture generated?

The answer to this apparent paradox was alluded to in the previous section. Since the semantic routines are usually specific to a particular application program, a mechanism is provided so that they may be coded as part of the application program module. For example, an application program to count the number of POINTS in an OBJECT needs basically to make the semantic association that a POINT is an object that causes a counter to be incremented by 1. To produce graphics output, the semantic association is basically that a LINE is an object that causes a geometric line to be drawn between the origins of the coordinate frames of two POINTS.

The mechanism for attaching semantics is provided in the routine SOLID, which causes application program routines to be invoked automatically as a preface and an epilog to the execution of PARTLIB routines. The technical details of this mechanism are described in a subsequent section.

The fact that semantics are attached after the PARTLIB is coded is essential to the generality of this method of representing geometric objects. This feature permits a single PARTLIB to represent a set of objects for a wide and open-ended range of potential applications. New applications may be added without modifying the PARTLIB, and the PARTLIB can usually be expanded without affecting old applications.

The particular PARTLIB implementation described here was based on a decision to model three-dimensional objects as polyhedra. Curved lines and curved surfaces were therefore omitted. This omission is not a fundamental limitation of the method, since curved lines and surfaces may be added to the PARTLIB. For example, a circular arc of radius RAD and angle THETA could be represented by the primitive code

```
CIRCLE: ENTRY(RAD,THETA);
RETURN;
```

The graphics application programs would then have to be expanded to provide the necessary semantic routine to draw circular arcs, and many other application programs might need similar additional semantics.

Other primitive objects that are substantially more complex than circles could also be added. For example, primitives could be provided for polynomial curves, conic sections, Ferguson-Coons patches [17], and so forth. In most applications, the supporting semantic routines for primitive objects of these sorts would be rather complicated.

CREATE: hierarchy construction

The principal entry point in the recursive module CREATE is called SOLID. The subsequent code oversees the hierarchical construction of each object out of its constituent subobjects. Entry at SOLID results in the following consecutive steps:

1. It dynamically allocates working storage for the particular subobject about to be created.
2. It performs "downward inference." This step involves copying data from the working storage of the parent object into the working storage of the new subobject. The data copied include the coordinate frame.
3. It calls all semantic routines whose names appear in a table called FLAG.
4. It calls the procedure for the subobject.
5. It calls all semantic routines whose names appear in a table called EXEC.

The data in each object's working storage include a set of upward and downward pointers, a coordinate frame matrix, and a polarity. The polarity is +1 for solid objects, −1 for holes, and 0 for "ghost" objects. Ghost objects are essentially the generalization of construction lines in drafting; they are used to generate the final object, but they have no semantic content. The entry point SOLID creates subobjects with the same polarity as the parent object, the entry point HOLE reverses the polarity, and the entry point GHOST generates a 0 polarity subobject. The system itself does not possess any physical model knowledge such as "Negative subparts must be entirely contained within positive objects," or "Two positive subparts may not occupy the same region of space." The modeling of unphysical objects is therefore not precluded.

All three of these entry points are written as PL/I generic entries to allow an arbitrary number of parameters to be passed from the parent object to the subobject. For example, LINE calls SOLID(POINT), which at step 4 above calls POINT. In this case, no parameters are passed to POINT. On the other hand, RTRIANG calls SOLID(LINE,ALTITUDE), which at step 4 above calls LINE (ALTITUDE). In this case, one parameter is passed to LINE.

The CREATE module also contains entries called FLAG and EXEC, which manage entries in the FLAG and EXEC tables referred to in steps 3 and 5. These routines provide the mechanism by which the application pro-

gram may attach semantic routines to the objects. For example, suppose that PLOTX is the name of a semantic routine for producing graphics output. A call to EXEC(PLOTX) would cause PLOTX to be placed in the EXEC table and be invoked at step 5 after each subobject was completed. PLOTX could then check whether the subobject just created was a LINE and, if so, plot it. A call to EXEC with no parameters erases the EXEC table. The entry point FLAG functions in a similar manner for semantics that must be invoked prior to subobject creation.

It can be seen that in essence CREATE intercepts subprogram calls between parent objects and their subobjects. This interception makes possible the addition of semantics without the need for changing the object procedures.

It is interesting to note that step 2 could be factored out and be called instead as the first semantic routine in the FLAG table. Actually, this step of SOLID is the only portion of CREATE that is specific to the topic of geometric construction. By replacing this step, it would be possible to adapt this geometry system to other fields in which it may be useful to intercept and interpret subprogram calls.

COORD: coordinate transformations

The COORD module contains a set of coordinate transformation routines that may be invoked to modify the coordinate frames of objects. These routines and the underlying data structures are very similar to those provided in the automation language AL [18]. All coordinate frames are represented as 4×4 floating point matrices with a top row of 1, 0, 0, 0. The remainder of the matrix can be partitioned into an origin vector and a 3×3 rotation matrix. This method of representing frames is normally referred to as homogeneous coordinates [19].

The explicit coordinate transformations provided in COORD perform translations, rotations, and reflections with respect to the object's frame [20]. In this frame, translations along the axes are called XTRAN(X), YTRAN(Y), and ZTRAN(Z). In addition there is a composite translation XYZTRAN(X,Y,Z). Rotations about the coordinate axes of the object's frame are called XROT(THETA), YROT(THETA), and ZROT(THETA). Reflections along the coordinate axes are called XMIRROR, YMIRROR, and ZMIRROR.

In addition to the explicit coordinate transformations itemized above, COORD provides a set of implicit transformations that can be used to attach one object to another. The routines that effect these implicit transformations are called DEFINE, RELATE, and APPLY.

The statement

CALL DEFINE(name,subaddress)

is used to assign a unique name to any subobject that has already been created. The subaddress vector is used to specify the relationship of the subobject in the hierarchy. The addressing method chosen relies on the natural ordering of subpart procedure calls within any given object's procedure. For example, in the RTRIANG procedure, subaddress (3,1) would refer to the third LINE, first POINT.

Specification of an object's subaddress requires detailed knowledge of the entire structure of which that object is a part. If the system described here were provided with some form of interactive graphics, the pointing of a light pen could cause an automatic determination of an object's subaddress. If a pointing were ambiguous, the system would have to provide the user with some means of indexing through alternate subaddresses to resolve the ambiguity.

Returning from these speculations to the description of the actual implementation, the unique name assigned by DEFINE can be used to refer to the origin of the coordinate frame of the relevant object. The statement

CALL ORIGIN(name,vector)

may be used to return the origin vector associated with the object of any particular name.

A composite new frame may be constructed from three such origin vectors in the following manner provided there is no degeneracy: The new frame's origin coincides with origin 1. The new frame's x axis points in the direction of origin 2. The new frame's XY plane includes origin 3.

The statement

CALL RELATE(name1,name2,name3,name4,name5,
name6,matrix)

computes the 4×4 transformation matrix that brings the composite frame specified by name1, name2, and name3 into coincidence with the composite frame specified by name4, name5, and name6. There are also two-dimensional and one-dimensional versions of this procedure.

The final stage of the implicit transformations is accomplished with the statement

CALL APPLY(matrix)

which applies the transformation matrix computed by RELATE to the current frame.

COORD also supplies several utility routines to allow programs more direct access to coordinate transformations and frames. These include the statements

CALL STORE(matrix)
CALL RECALL(matrix)

which, respectively, save the current frame in a matrix and restore a matrix to be the current frame. In addition,

routines MATMPY and MATINV are provided for direct multiplication and inversion of frames and transformations.

Finally, there are routines in COORD that initiate and terminate the hierarchy of objects. The statement

```
CALL BEGIN(SIZE)
```

allocates a storage area of the specified SIZE in which CREATE subsequently dynamically allocates working space to objects. BEGIN also causes the first working storage space to be allocated and initialized to be at the top of the hierarchy, have the identity frame, and have polarity +1. Lastly, BEGIN clears the FLAG and EXEC tables. The statement

```
CALL END
```

frees the storage area allocation performed by BEGIN. The only reason that BEGIN and END are in COORD rather than in CREATE is so that they may serve as models for users who wish to write application programs that themselves contain the storage area for dynamic allocation.

Application programs

• *Semantics for graphics*

The technique of writing application programs is best demonstrated by means of an example. The example chosen for this purpose is graphics output.

If one ignores all consideration of having the graphics output look aesthetically pleasing, it is extremely easy to write a naive graphics semantic routine to produce an output file that can be used to draw a picture of any object. This simplistic routine is shown below:

```
PLOTX: PROCEDURE(NODE); /* SIMPLE GRAPHICS SEMANTICS */
DECLARE NODE ENTRY, (PT1,PT2) POINTER,(A(3),B(3)) FLOAT;
IF NODE=LINE THEN DO; /* IS OBJECT A LINE? */
  CALL DEFINE(PT1,1); /* NAME OF FIRST POINT */
  CALL DEFINE(PT2,2); /* NAME OF SECOND POINT */
  CALL ORIGIN(PT1,A); /* ORIGIN OF FIRST POINT */
  CALL ORIGIN(PT2,B); /* ORIGIN OF SECOND POINT */
  PUT SKIP DATA(A,B); /* OUTPUT BOTH ORIGINS */
END;
END;
```

Not shown in this code is a standard set of declarations that resolve references to system entry points. The IF statement checks whether or not the object just created is a LINE. If it is a LINE, the DEFINE statements assign the names PT1 and PT2 to the subaddresses of the LINE that correspond to its first and second POINTS. The calls to ORIGIN then extract the origins of the coordinate frames of these two points, and the PUT statement produces these vectors. An independent graphics package may subsequently use the output file to control the generation of a picture.

• *More aesthetic graphics*

A more advanced graphics routine PLOT directly generates the LINES of each of the five basic volume shapes. In addition, PLOT produces an output file of objects as seen in the frame of the camera so that perspective views may be generated. The camera frame is taken to be the frame that was current when PLOT was originally called. Thus the viewpoint and orientation may be altered by moving the initial frame.

• *Polyhedral topology*

Semantics have been written by Lieberman [21] that permit this system to be used as a parametric preprocessor that generates topological files compatible with available programs involving polyhedral topology [7]. Algorithms have been added to perform the operations of set union, intersection, and difference. This method for generating descriptions of complex mechanical parts has been described elsewhere.

Example

The example of this section demonstrates how a very short procedure can represent a very complex object and demonstrates some semantics other than graphics. The recursive PL/I procedure shown below represents a two-dimensional botanical tree.

```
BRANCH: ENTRY(LENGTH) RECURSIVE; /* BRANCH BUILDER */
BEGIN;
  DECLARE(SPACE,SAVE(4,4)) FLOAT, (I,J,IMAX) BIN FIXED;
  CALL SOLID(LIMB,LENGTH); /* GENERATE LIMB */
  SPACE = SQRT(LENGTH);     /* SPACING OF BRANCH PAIRS */
  IMAX=(LENGTH-1)/SPACE;    /* NUMBER OF BRANCH PAIRS */
  IF IMAX<1 THEN RETURN;    /* DOES LIMB HAVE BRANCHES? */
  ELSE CALL STORE(SAVE);    /* FRAME OF CURRENT LIMB */
  DO I=1 TO IMAX;           /* SETS OF BRANCH PAIRS */
    CALL RECALL(SAVE);      /* FRAME OF LAST PAIR */
    CALL ZTRAN(SPACE);      /* AT INTERVALS OF SPACE */
    CALL STORE(SAVE);       /* FRAME OF CURRENT PAIR */
    DO J=1 TO 2;            /* PAIR OF BRANCHES */
      CALL RECALL(SAVE);    /* FRAME OF CURRENT PAIR */
      CALL ZROT(180*J);     /* ROTATIONAL SYMMETRY */
      CALL YROT(60*(1-I/(IMAX+2)));/* SPREAD ANGLE */
      CALL SOLID(BRANCH,(LENGTH-1*SPACE)/2);/*RECURSION*/
    END;
  END;
END;
RETURN;
```

The first action of BRANCH is to generate the main LIMB of the tree. Next it determines that pairs of secondary limbs will occur along this limb at intervals of SPACE equal to the square root of LENGTH. It executes a loop over these pairs, and nested within this loop it executes a loop over the two secondary limbs of each pair. Inside this loop, a recursive call via SOLID to BRANCH generates a secondary limb that is shorter than the main

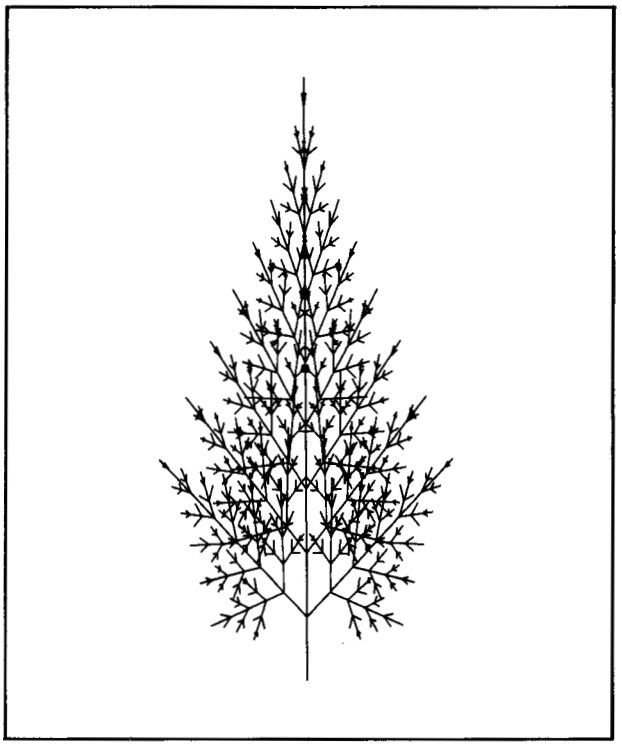

Figure 2 Picture of recursive tree.

limb. The recursion continues until the LENGTH of each limb is shorter than one and there are no further offshoots.

The semantic routine COUNTL, shown below, may be used to count the number of LIMBS in a tree of this form. For instructional purposes, some deliberate complications have been included in the program: Rather than counting all the LIMBS, COUNTL counts only those that have a length greater than an arbitrary threshold LMIN and that start with a Z coordinate height greater than HMIN.

```
COUNTL: PROCEDURE(NODE,LENGTH);
DECLARE NODE ENTRY, PTLIMB PTR, (LENGTH,A(3)) FLOAT;
IF NODE=LIMB THEN DO;/* ONLY COUNT LIMBS */
  CALL DEFINE(PTLIMB);/* NULL SUBADDRESS */
  CALL ORIGIN(PTLIMB,A);/* ORIGIN OF LIMB FRAME */
  IF LENGTH>LMIN & A(3)>HMIN THEN NLIMB=NLIMB+1;
END;
END;
```

Not shown in this code are the declarations and initial assignments of LMIN, HMIN, and NLIMB. The sixth statement of COUNTL increments NLIMB only when the specified thresholds are exceeded. It should be noted that LENGTH, which is a parameter of LIMB, is also a parameter of COUNTL. In general, all parts parameters are also available to the semantic routines.

With graphics semantics attached, a call to SOLID (BRANCH,90) generates the picture shown in Fig. 2. The tree in this picture consists of 747 distinct limbs. By putting a call to SOLID(BRANCH,N) inside a loop over N it would be possible to make a movie of the tree as it grows.

Limitations

This approach to representation of objects does raise two significant problems.

The first problem, which may limit the utility of this method in many potential applications, is its high consumption of computer resources. For example, in one program modeling a real object of moderate complexity, about 200 K bytes of working storage was required. To provide the large amount of memory needed, it was run under VM/370 on an IBM System/370, model 168. On this machine, execution of the program required about 3 seconds of virtual CPU time. Although these numbers are rather high, it should be noted that no attempt was made to minimize them. This storage requirement could be cut approximately in half by using a more compact method for encoding hierarchical pointers and coordinate frame data. The execution time could be cut by about one third if routines were provided to avoid the redundant computation of trigonometric functions.

The second problem associated with the use of PL/I procedures to represent objects is the difficulty of making the system interactive. This difficulty could have been avoided by writing the system in APL or LISP instead of PL/I, but then performance would have suffered. In the current system the effective turn-around time for debugging application programs is about 5 minutes. For the PL/I system to be made truly interactive, it would be necessary for a running application program to generate PL/I source code, compile it, load it, and execute it. Under VM/370 it is actually possible for a running program to invoke the PL/I compiler as a subprogram. It seems likely, however, that many nontrivial complications would arise in this sort of undertaking. In addition the performance would probably still be undesirably slow. For these reasons the system described in this paper is currently not interactive, although hope of someday making it interactive has not been totally abandoned.

Summary

A new method of modeling three-dimensional objects has been described. The system has the capability of describing any object that can be represented as the sum or difference of other objects. Each object is represented as a PL/I procedure, which in turn can invoke procedures representing the subparts. The use of variables makes it possible to describe generic shapes and mecha-

nisms. A technique of essentially intercepting calls to subpart procedures enables the user to supply semantic routines in the application programs, without modifying any object's representation.

The system described here is being used for the parametric generation of polyhedral topology files for complex mechanical parts. Currently, work is underway to apply the procedural system to Monte Carlo simulations of tolerancing and other forms of imprecision in discrete parts assembly.

Acknowledgments

My interests in this research were motivated in large part by P. Will. R. Evans contributed substantially to the coding of the coordinate transformations. Useful suggestions concerning the allocation of storage were made by L. Lieberman. M. Lavin improved on some of the initial data structures. Both Lieberman and Lavin have applied this system to the parametric generation of polyhedral topology files. I also profited from a general interchange of ideas with these three people as well as with A. Appel, D. Bantz, W. Burge, and M. Wesley. The graphics figures were plotted using a system provided by M. Loughlin and A. Stein. Finally, L. Junker helped teach me PL/I and assisted in correcting some of the more esoteric bugs.

References

1. H. Gelernter, J. R. Hansen, and C. L. Gerberich, "A Fortran-Compiled List Processing Language," *J. Assoc. Comput. Mach.* **7,** 87 (1960).
2. L. G. Roberts, "Machine Perception of Three-Dimensional Solids," Ph.D. Thesis, Massachusetts Institute of Technology, May 1963.
3. G. J. Agin and T. O. Binford, "Computer Description of Curved Objects," *Third International Joint Conference on Artificial Intelligence*, Stanford University, Stanford, CA, August 1973, p. 629.
4. P. Winston, Ph.D. Thesis, Massachusetts Institute of Technology, 1970 (reprinted as *MIT Project MAC Report MAC-TR-76*, September 1970).
5. B. G. Baumgart, "Winged Edge Polyhedral Representation," *Stanford Artificial Intelligence Laboratory Memo AIM-179, STAN-CS-320*, Stanford University, Stanford, CA, October 1972.
6. D. P. Greenberg, "Computer Graphics in Architecture," *Scientific American* **230,** 98 (1974).
7. A. Appel, "Modeling in Three Dimensions," *IBM Syst. J.* **7,** 310 (1968).
8. C. Csuri, "Real-Time Film Animation," *Annual Report to the NSF from the Computer Graphics Research Group*, Ohio State University, Grant No. GJ-204, January 1972 to January 1973, p. 76.5.
9. E. Catmull, "A System for Computer Generated Movies," *Proc. ACM Ann. Conf.* Boston, MA, August 1972, p. 422.
10. W. Newman and R. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill Book Co., Inc., New York, 1973, Part Four, "Three Dimensional Computer Graphics," p. 233.
11. J. Y. S. Luh and R. J. Krolak, "A Mathematical Model For Mechanical Part Description," *Commun. ACM* **8,** 125 (1965).
12. P. G. Comba, "A Language For Three-Dimensional Geometry," *IBM Syst. J.* **7,** 292 (1968).
13. R. Wehrli, M. Smith, and E. Smith, "ARCAID: The ARChitect's computer graphics AID," *University of Utah Report ITEC-CSc-70-102*, Salt Lake City, UT, June 1970.
14. I. C. Braid, *Designing With Volumes*, Cantab Press, Cambridge, England, 1974; also I. C. Braid, "The Synthesis of Solids Bounded by Many Faces," *Commun. ACM* **18,** 209 (1975).
15. "An Introduction to PADL," *Production Automation Project Technical Memorandum 22*, University of Rochester, December 1974.
16. *Numerical Control Users' Handbook*, edited by W. H. P. Leslie, McGraw-Hill Book Co. Inc., New York, 1970.
17. J. C. Ferguson, "Multivariable Curve Interpolation," *J. Assoc. Comput. Mach.* **11,** 221 (1964).
18. R. Finkel, R. Taylor, R. Bolles, R. Paul, and J. Feldman, "AL, A Programming System for Automation," *Stanford Artificial Intelligence Laboratory Memo AIM-243*, STAN-CS-74-456, Stanford University, Stanford, CA, November 1974.
19. W. Newman and R. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill Book Co., Inc., New York, 1973, Appendix II, "Homogeneous Coordinate Techniques," p. 467.
20. H. S. M. Coxeter, "Introduction To Geometry," John Wiley and Sons, Inc., New York, 1961, Chapters 3 and 7.
21. M. A. Lavin and L. I. Lieberman, "A System for Modeling Three-Dimensional Objects," *Research Report RC-5765*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1970.

Received June 23, 1975; revised January 2, 1976

The author is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.