GLIDE: A LANGUAGE FOR DESIGN INFORMATION SYSTEMS

Charles Eastman and Max Henrion

Institute of Physical Planning,
School of Urban and Public Affairs
and Department of Architecture,
Carnegie-Mellon University

keywords: design, database, architecture, engineering, languages.
CR Classifications: 3.20, 3.40, 4.22, 8.2.

1. INTRODUCTION

A primary means for evolving more powerful computer software is through the embedding of higher level features into problem-oriented languages. To late, CAD (computer -aided design) systems have had to begin development from FORTRAN, PL/1 or language of similar level. Yet there are needs common to many CAD applications which could be provided in the language or system foundation from which they are developed. GLIDE (Graphical Language for Interactive DEsign) is an attempt to organize the commonly -needed database features and operations for the design of physical systems into a high level computing environment. By "physical system" we mean artifacts such as buildings, ships and machines, that are made up of a 3-dimensional components and in which spatial arrangement is an important concern. GLIDE is intended to provide an efficient computer representation for physical systems in sufficient detail for their design and construction.

The provision of a complete and coherent computer-based model of 3-dimensional objects and their spatial arrangement offers many advantages over conventional specifications and drawings. Given a complete 3-dimensional spatial representation of the artifact being designed, the designer is guaranteed that all 2-dimensional projections generated from it will be consistent. Shape information, normally represented in drawings, can be integrated with functional and performance information so that application programs can access and manipulate both kinds, without the manual translation now required for drawings. These application programs can check data consistency, evaluate the design's structural, thermal or other properties, estimate cost, or add conventional details. Others can generate displays, construction drawings and numerical control tapes from the stored shape information, which will have guaranteed correspondence with the design. Many such programs already exist and more can be imagined for specific applications in different fields of design.

One can distinguish three levels of representation for 3-dimensional elements:

1 An IMAGE that contains sufficient information for generation of graphics, possibly including hidden line or surface elimination. This usually consists of an unstructured set of planar or curved faces.

2 A SHAPE model represents elements as solid bodies, usually polyhedral. It is spatially complete, meaning that it has a closed, volume-containing surface which contains sufficient information to determine whether any point is inside or outside. For the shape sculpting operations of union, intersection and difference on spatial domains, such a representation is essential [3].

3 An OBJECT model extends this shape model to include functional information, such as material, weight, rigidity etc. Thus shape may be only one attribute among many others which describe the object.

There now exist a wide range of computer languages for 2-D graphics applications, implemented as subroutine packages, or as extended or entirely new languages. While many of these can and have been extended to deal with 3-D IMAGES, only recently has there been work in developing general Shape models [1,17].

Most systems capable of full Shape modeling are controlled by a non-extendable interactive command language [2,3,8]. A few are imbedded in general procedural languages, both interpretive and compiled [4,10,11]. Several of these can store command strings on disk and in this way represent large physical system projects that can be repeatedly accessed and modified over a period of time. A few store information on disk in a run-time format, allowing interactive manipulation of large amounts of data [8,16] Several large CAD systems also support object models with comprehensive non-spatial attributes, but whose shape models do not extend to general polyhedra [5,12,20].

In GLIDE, we have attempted to combine the most desirable mix of these features that would be useful as a high level environment for physical systems CAD. The authors are members of

Siggraph '77, July 20-22 San Jose, California

a team at Carnegie-Mellon University that earlier implemented a prototype interactive design database called BDS (Building Description System) (8]. GLIDE incorporates many of the principles developed in BDS, including the compact representation of a large number of complete shape models, the shape -manipulating operations of union, intersection and difference, and several methods for the user definition of shapes. It embodies those facilities in a general high-level language and extends them with additional constructs for making functionally complete object descriptions and for relating objects to each other.

GLIDE is intended to combine the advantages of an interpreted command language for interactive design with those of a procedural programming language. It does not provide highly specialized programs for particular applications, but rather is a general language, with object modeling capabilities in a database environment that incorporates disk management facilities and specialized accessing schemes. It is intended to form a convenient basis for constructing a new generation of more powerful CAD applications. In this paper, we outline the conceptual approach taken in the design of GLIDE. We shall focus of the features which we consider to be of particular importance in the design of design information systems.

The complete specification of GLIDE is available in [6]. It has been implemented on a PDP-10, under the TOPS-10 operating system. using the Bliss implementation language. Implementation is also planned for a PDP-11/34, under the UNIX operating system. Versions are anticipated for both storage tube and refresh display graphics.

2. USER ENVIRONMENT:

In its development GLIDE was considered from two different perspectives: Firstly as a set of data types and commands for interactively defining, arranging and inspecting a design; and secondly, as a general high level language for extending the range of structures and operations available at the first level so as to meet the requirements of a particular application. From the first perspective, the language ought to be easily used by a designer with minimal computer experience. From the second perspective, it should provide good facilities for the experienced applications programmer, so as to allow easy extension of the capabilities of the system.

As naive users of the system gain experience, they should be encouraged to expand their use of the more general features. While we do not see designers becoming programmers (or vice versa) overnight, we hope that such a system will allow the naive user to find the system useful with minimal learning and that as he or she demands more powerful capabilities, they can be easily learned in small increments. We see this as a way of reducing the gap between these two classes of people. This is a prerequisite for the development of systems which are tailored closely to the requirements of individual designers and their practices and which incorporate valuable kinds of design knowledge. This is in contrast to most existing systems, which enforce a strict separation between the command language and the system implementation language.

The requirements of interactive design dictate that GLIDE be interpretive. Each command entered by the designer is executed immediately so that its effects can be inspected. An important form of inspection is graphical; the user is able to view the spatial effects of his actions as they are made.

Programs incorporating special operations relevent to a particular design field or to the practices of an organization also need to be entered as procedures and stored for subsequent use. These are checked syntactically as they are entered, a line at a time, by an incremental translator. The pre-translation of procedures requires that translation be independent of the particular state of the database and that any external data or procedures needed are linked at run-time.

A design project will be developed over an indefinite period of time and many terminal sessions, requiring millions of words of storage. Thus a project, while under development, should exist in a run-time format directly available online. The GLIDE operating environment keeps a small subset of the database in core and the rest on disk, and the management and swapping of records is handled automatically without conscious intervention by the user.

3. BASIC FEATURES:

GLIDE incorporates the basic features of a general purpose Algol-like language. These are fairly conventional and so we will not describe them in detail. They include the following: the simple data types of INTEGER, REAL, BOOLEAN and TEXT (i.e. character string); variables of these types mus't be declared before use and may be scalar or vectors; the standard arithmetic, comparative and logical operators; the single-value assignment operator; control structures including, the IF <bool> THEN e ELSE e, and FOR and WHILE loops; a hierarchical structure of blocks delimited by BEGIN and END, with dynamic declaration of local variables, and user-definable procedures. Flow of control can also be changed by the escape statements: LEAVE, EXIT and RETURN, for escaping from a BEGIN...ENO block, the body of a loop, and of a procedure, respectively.

The language is "expression-oriented" that is, every statement returns a value and can be used as part of a higher level expression. The value of an assignment is the value assigned.

The value of an IF statement or block is that part of the last statement executed. Procedures may act as functions and return a value when called. The escape statements return the value of the expression following it to the construct from which they escape, eg. the value RETURNed by a procedure. Wherever a statement is treated as an expression, its type must match that expected by the context.

As a database definition language GLIDE also contains the means for creating record types for defining objects, their shapes and their relationships. The general record type for defining a class of objects is a FORM, whose instances are COPIES. There are also two special record types for defining shapes: TOPO for defining surface topologies, and POLY for defining polyhedral shapes. A SET consists of a collection of Copies or other Sets. Copies and Sets can be treated as equivalent in many contexts and the union of their types is an ITEM. These record types will be described in more detail.

4. FORMS AND COPIES:

FORMS are user-defined complex datatypes somewhat analogous to the Record of PL/1 and Pascal, Struct of Algol-68, or Record class or the Relation of database definition languages. In design databases it is usual to encounter families of objects that have many but not all properties in common. Examples might be a class of doors in a building which have similar shape hut different locations, finishes and handles; or a class of container vessels in a chemical process plant. In this situation it is convenient to define a FORM not only as a "schema" which defines the ATTRIBUTES of a class of objects, but also as a "prototype" which contains the initial default values for objects derived from it. For this reason the instances or occurences of a Form are known as its COPIES. To define a COPY it is only necessary to specify its Form and those Attribute values which differ from it. This allows more concise entry of data and is also used in the current implementation for achieving compact storage of data. The advantages of this organization will become especially apparent when we consider the spatial Attributes for shape and location.

The functional properties of importance in different areas of design vary greatly. User defined Attributes are provided for representing these properties of an object. Attributes may be of any simple type, including vectors of fixed size, or any record type (actually references to records). Since the same Attribute name, eg. COLOR or MATERIAL, may be relevant to many different Forms, they are declared global rather than local to a particular Form definition. Attribute declarations are introduced by the keyword ATTRIB:

```
ATTRIB TEXT title,notes;
ATTRIB REAL shelflength,deskft,filespace;
ATTRIB BOOL phone;
```

A Form definition consists of a block containing Attribute initializations and possibly other statements enclosed between BFORM and EFORM. (These block delimiters are intended to aid type checking, but for conciseness can be replaced by "(" and "}", as can the delimiters for all the other record definition blocks to be described.)

```
FORM workspace =
        BFORM title←'Secretary';
        shelflength←15; deskft←5;
        phone←true; notes←'none'
        EFORM;
```

Copies are identified by subscripts appended to their Form. The Form in its role as "prototype" may also be treated as the zeroth Copy.] In the Copy definition the subscript may be specified or defaulted, as the integer after the previous highest. The Copy definition block delimiters are "I" and "}".

```
COPY workspace[10] = {;phone←FALSE};
COPY workspace = {;title←'chairman';
    deskft←10; notes←'likes red wallpaper'};
```

Form identifiers and Copies are bound permanently to their definition and cannot be reassigned. The bind operation is denoted by =, as distinct from the assignment operator "←". However, the individual Attribute values can be modified and a Form or a Copy can be dynamically expanded by the addition of new Attributes. Note that changing a Form Attribute will affect all Copies with the default value; but new individual values may be assigned to an Attribute of a Copy that was originally defaulted. These changes are achieved in a modification block, attached to the object name with a ":":

```
workspace[11]:  {title←'chairperson'};
ATTRIB REAL filespace;
workspace:  {filespace←3.25; deskft←12};
```

This dynamic extension of record formats is not allowed in most languages and it adds some complexities to implementation. However we feel it to be important in a database system oriented to interactive design, where simple structures may be created initially which are later refined and elaborated in ways that could not have been anticipated at the start of the process. It also allows set membership information to be added to objects incrementally without any prior restriction on the range of Sets. Attributes needed for a particular kind of analysis may also be appended to objects whenever the need is identified.

Outside a definition or modification block an Attribute value can be accessed by the syntax:

<attrib> OF <copy>

Since the names of Forms and hence of Copies are permanently, bound it is frequently desirable to use a variable to refer to such objects. These may be declared to be of type ITEM. An Item can be either a Form, a Copy or a Set of Items. An Item variable may be assigned and reassigned by the "←" operator.

```
ITEM boss; boss ← workspace[11];
```

5. PROCEDURES:

Procedures serve both their conventional role in the development of code for application programs and as means to extend GLIDE as a high-level user-oriented command language adapted to particular design tasks. Accordingly, in addition to the conventional calling syntax with the parameters enclosed in parentheses; a "command" syntax is provided. In this syntax the parentheses are omitted and alphanumeric separators can be used instead of commas.

```
REVERSE(a,b,c);
ALIGN a WITH b ALONG c;
REPEAT window[1] BY 8.5,0,0 TIMES 25;
```

The type of a procedure and of its parameters are given in the declaration, together with the seperators if any.

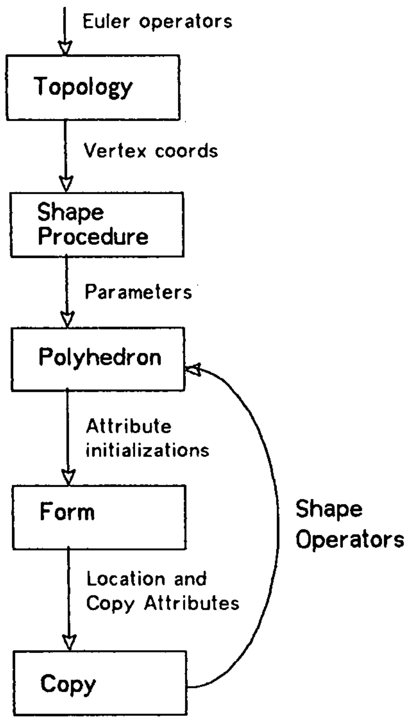
```
REAL PROCEDURE max(REAL x,y)=
            IF x GTR y THEN x ELSE y;

SET PROCEDURE repeat(ITEM a
    BY REAL dx,dy,dz TIMES INTEGER n)=
    BSET ITEM c; c←a;
    FOR I TO n DO c←COPY a = {c dx,dy,dz}
    ESET;
```

6. SHAPE DEFINIIION:

Shape is an Attribute of particular importance for physical systems design. While Shape may represent a solid object it may also represent a void, such as the space in a room, or the envelope of a set of objects. It is represented in GLIDE by a record of type POLY, representing planar-faced polyhedra. A Form record may be associated with a Polyhedron- by binding it to the SHAPE Attribute of the Form. All Copies of this Form will have the same Shape, possibly with parametric variations.

The representation of a general polyhedron needs a large amount of data and efficient means of entry are required. Below, we describe the language constructs for defining polyhedra. Several different methods can be used, each based on a partial description that can be created and named for later reference. The partial descriptions are related hierarchically, as shown in Figure 1. This hierarchy facilitates the entry of information needed to define a Polyhedron and is also used in GLIDE as a means to greatly compact the storing of shape information [7]. We introduce the shape definition facilities in the approximate order in which they might be learned, from the simplest to the most complex to use, in order to emphasize the possibility of a gradual progression, as users become more familiar with the facilities. Thus in the later examples we shall show how to construct entities such as Topologies, which are earlier treated as primitives.

Figure 1: Object definition hierarchy



6.1 Copies:

Given an existing Form with defined Shape, new objects are easily derived as Copies at different locations. Location consists of the system -defined Attributes: LX, LY, LZ, which define a translation, and AX, AY, AZ which define an orientation specified as rotations in degrees round the three axes. These can be initialized and accessed like other Attributes, but a more convenient syntax for location is:

```
<item> <lx>,<ly>,<lz> \ <ax>,<ay>,<az>
```

The transform is relative to the <item>, or if that is omitted, is relative to "world" axes. Omitted trailing values default to zero.

```
COPY door[3] = {door[2] 15,20\0,90};
COPY door = {10,12,10; handle←'BRASS'};
```

A regular arrangement of Copies can be created with the use of a FOR loop:

```
! a spiral staircase, see fig 5;
FOR i FROM 1 TO numsteps DO
        COPY step={0,8.5*i \0,30*i};
```

6.2 Parameter ized Shapes:

Any well-used system will have built up a large library of standard shapes which may suffice for simple applications, but clearly there must be means for creating new ones. One method is to use parameterized shapes, which are procedures that define and return a Polyhedron that is defined parametrically from within a class. A simple example is Cuboid, parameterized by length, width and height.

```
POLY box1 = cuboid(0.002,0.003;0.005);
POLY collar = column(12,riser,r);
POLY plate = wedge(radius,th,angle);
```

Storing a catalog of parameterized Shapes has become a conventional technique in most CAD systems. Different classes of applications may be served by libraries containing different sets, eg. pipes for process plants, I-section beams for structural steel design, etc. GLIDE provides the means of defining new ones as procedures which create classes of Polyhedra.

The shape of an object is specified by binding a Polyhedron record to the SHAPE Attribute within the Form definition.

```
FORM matchbox = BFORM
          num.matches ← 200; color←'red';
          SHAPE=cuboid(0.002,0.003,0.005)
          EFORM;
```

6.3 Shape Operators:

An extremely powerful method for constructing complex Polyhedra is by joining simpler ones together, or by subtracting one from another. The Shape operators that do this create a new Polyhedron, with a new Topology and Geometry. There are three to perform the union, intersection and difference operations on the spatial domains represented by the Polyhedra:

```
COMBINE <poly> WITH <poly>
LAP <poly> WITH <poly>
CUT <poly> FROM <poly>
```

An example of their use is shown in figure 2. The various cuboids ir 2a were located interactively and subtracted from the centre one, with the resulting Polyhedron in 2b.

The LAP operation is the ultimate test for spatial conflicts. Note that type coercion can occur from a Copy to its Shape and hence the above and other operations expecting a Polyhedron will also work with operands which are Copies of a Form.

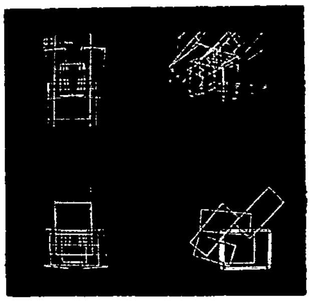


Figure 2a

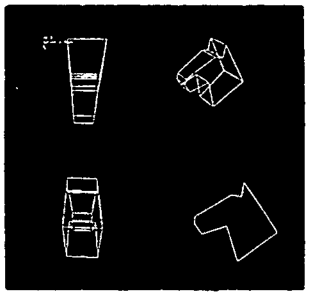


Figure 2b

6.4 Defining a Polyhedron:

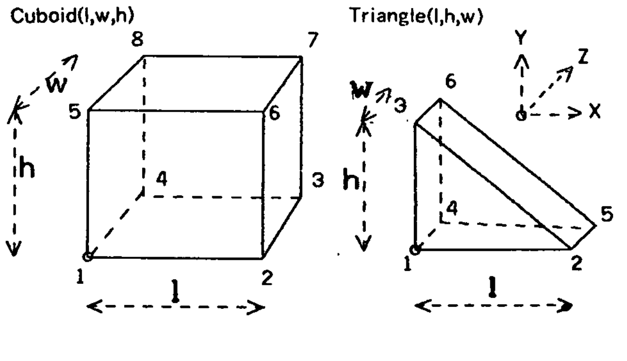
A Polyhedron consists of a number of components: FACES, EDGES and VERTICES. Each Face and each Vertex consists of an ordered ring of Edges and each Edge contains two Vertices and two adjacent Faces. The information associated with these components is divided into two parts: the TOPOLOGY and the GEOMETRY. Topology defines the number of these components and there interconnections. The GEOMETRY specifies their spatial position and physical dimensions. Note that here "topology" means the network of vertices, edges and faces of a polyhedral surface, which is not the quite the same as its use in the field of mathematics of that name. A Topology may be common to many different Polyhedra and so it can be entered independently from any particular one and referenced by rrame. See Figure 1.

A Polyhedron definition is delimited by BPOLY and EPOLY (or "{" and "}"), and starts with the specification of a Topology, which defines the numbers of faces, edges and vertices and their connections. The Geometry is then specified by a set of statements which bind to the vertex coordinates (VX, VY and VZ) a value relative to the origin. These have the form:

```
VX[<sublist>] = <real>
```

where <sublist> is a list of one or more vertex ind i ces. Those co-ordinates not mentioned default to zero.

```
POLY PROCEDURE cuboid(REAL l,w,h) =
     BPOLY hexa; ! A hexahedron Topology;
     VX[2,3,6,7]= l; VY[3,4,7,8]= w;
     VZ[5 TO 8]= h
     EPOLY;
```



```
POLY PROCEDURE triangle(REAL l,h,w)=
    BPOLY TOPOL= tria; VZ[4 TO 6]=t;
    VX[2,5]=l; VY[3,6]=h
    EPOLY;
```

After the creation of a Polyhedron, the system checks that the Vertex coordinates define planar Faces, and that the bounded domain is inside. A Polyhedron can be treated as a Form with only a Shape Attribute, and Copies can be made and moved, and other Attributes added.

6.5 Defining a Topology:

A Topology may be constructed by means of Euler operators, which create and link new Vertices, Faces and Edges. (They are named after Euler who showed that they are sufficient to construct any legal polyhedron. For a description of their use see [2,8].)

These Euler operations and other statements are combined into a Topology definition block enclosed between BTOPO and ETOPO. BTOPO generates a primitive point Topology consisting of a single Vertex on a Face. Further Faces. Edges and vertices are added and linked by the following Euler operations:

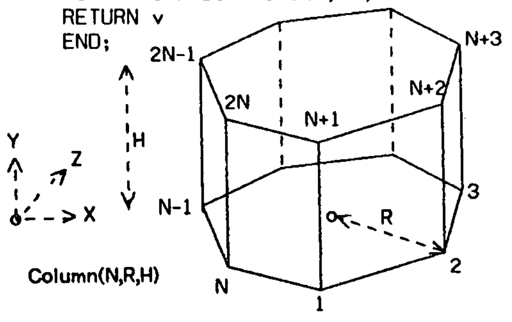
CVE(V, F): To create a new Vertex (and Edge) attached to Vertex V on Face F.

CFE(V1, V2, F): to create a new Face (and Edge) by linking vertices V1 and V2 on Face F.

MERGE(F1, F2, V1, V2) to merge two Faces F1 and F2 starting by merging V1 on F1 and V2 on F2, and finally eliminating the two Faces. Thus it can create a Polyhedron with a hole, or merge two into one.

Variables of type VERTEX, FACE and EDGE may be declared within a block defining a Topology or Polyhedron, or as parameters of a new Euler procedure. The Faces,Edges and Vertices are uniquely numbered for each Topology in the order they are created. This index can be used to identify them, and thus type coercion can can take place from an integer to the expected subrecord type.

```
    ! To create a chain of N Vertices
starting at V on F;
    VERTEX PROCEDURE chain
        (VERTEX v; FACE f;INTEGER n)=
        BEGIN
        FOR i TO n DO v←CVE(v,f);
        RETURN v
        END;
```



Column(N,R,H)

```
    !Extrusion (or prism) Topology;
    TOPO PROCEDURE extrude(INTEGER n)=
        BTOPO
        chain(n-1,1,1);  !lower Face;
        CFE(n,1,1);      !closes face;
        chain(n,1,1); ! upper Face;
        CFE(n+1,2*n,1);!closes upperface;
        ! make the side Faces;
        FOR i FROM 2 TO n DO CFE(i,i+n,1)
        END;

!Hexahedron (eg cube) and triangular prism;
    TOPO hexa = extrude(4),
         tria = extrude(3);

    POLY PROCEDURE column(INTEGER n;REALr,h)=
        BPOLY EXTRUDE(n);
        REAL ang; ang←.360/n;
        FOR i TO n DO BEGIN
                VY[i+n] = h;
                VX[i,i+n] = r*SIN(i*ang);
                VZ[i,i+n] = r*COS(i*ang)
                END
        EPOLY;
```

6.6 Summary:

Other complex shapes may most easily be defined graphically, as for example in several programs which can construct a 3-0 shape from two or more 2-D pictures drawn on the tablet[13,18,19]. We regard these as a particular class of application program, which can be interfaced to GLIDE.

Notice that the hierarchy of Polyhedron definition facilities, when combined with a procedural language and database features, gives the user a range of alternatives for storing information, both procedurally or as data. Thus a Polyhedron may be defined as a procedure and invoked only when it is needed, with only local declarations. Alternatively it may be stored as data permanently. Thus no particular bias is made with regard this sometimes hotly debated issue [14,15]. The application programmer can select that means of storing information that best reflects the conditions regarding space, speed and needed information.

7.1 ACCESSING SHAPE INFORMATION:

The Attributes of a Shape can be accessed in the same way as the other Attributes of an object, either within a modification block or as <attrib> OF <item>. Modification of these Attributes, however, is limited. The following Attributes describe the number of Topological components, the Vertex coordinates and the Face coefficients (which are computed from them):

```
NUMVERTS, NUMFACES, NUMEDGES
VX[v],VY[v], VZ[v]
FACEA[f], FACEB[f], FACEC[f], FACEK[f]
```

The following system-supplied "clocking" functions provide ways of accessing the topological relations:

```
EDGE PROCEDURE NEXT.EV(EDGE e; VERTEX v)
EDGE PROCEDURE NEXT.EF(EDGE e; FACE f)
VERTEX PROCEDURE NEXT.VF(VERTEX v; FACE f)
FACE PROCEDURE NEXT.FV(FACE f; VERTEX v)
VERTEX PROCEDURE OTHER.VE(VERTEX v; EDGE e)
FACE PROCEDURE OTHER.FE(FACE f; EDGE e)
```

The NEXT functions produce the next component following the the first argument in the ring, clockwise looking inwards on a Face and anti-clockwise round a Vertex. The OTHER function produce the other component to the first argument on the given edge. In the case that the first argument is null (index 0) they return any component of the requested type that is connected to the second argument.

7.2 FORMS WITH VARIABLE SHAPES:

Shape Attributes may not be directly updated, since they are bound at definition. However the vertex coordinates may be bound to a Real Attribute of the Form, and hence changing the value of the Attribute referred to by the cometry also alters the Shape. Thus by initial ICl defining a Polyhedral geometry in terms of a set of Attributes, the Shape can be parametrically variable for each Copy.

```
ATTRIB REAL LENGTH,WIDTH,HEIGHT;
ATTRIB TEXT MANUF;
FORM matchbox = BFORM
    manuf ← 'sun match inc';
    length←5.6;
    width←2.6;
    height←4.825;
    SHAPE = BPOLY hexa;
        VX[2,3,6,7] = length;
        VY[3,4,7,8] = width;
        VZ[5 TO 8] = height
        EPOLY
    EFORM;
```

By setting up a Form in the above manner, each of its Copies may have a cuboid Shape but with different dimensions. The initial values assigned to the Attributes are the Form's default val ues.

```
COPY matchbox = {100,100 90; width←3.0;
        length←3.95};
```

By allowing copies of a Form to vary in this way, great convenience is allowed in the treatment of objects cut from standard stock. Al pipes or beams of a certain section may be Copies of the same Form, but with variable lengths, allowing quantities of materials to be conveniently aggregated.

8. SETS:

The Set provides a way of referencing a number of objects together, allowing them to be treated as a single entity. Conceptually, a Set is an unordered list and it can contain objects and also other Sets. Thus multi-level hierarchies can be defined. All operations which take an Item for their argument can operate on a Set.

A Set may be defined simply by a list of Items, between the delimiters BSET and ESET (or "I" and "}" again.) A list of several Copies of the same Form can be specified concisely by listing the index range as the subscript. The subscript ALL means all the Copies of the Form. An origin for rotation may be specified as the first statement:

```
SET boxes = BSET 100,100;
        matchbox[ALL]; soapbox[1,3,7];
        box[1 TO 25,30,40]
        ESET;
```

Like other definition blocks a Set definition can contain general statements, and if any of these evaluate to an Item, then that will be also placed in the Set. This provides a convenient means of collecting and accessing all the objects created or accessed during a delimited part of a design session, and encourages (although it does not enforce) a hierarchical structure for organizing the design database.

```
SET staircase = BSET
    FOR i FROM 1 TO N
       DO COPY step={0.4*i 0.8*i};
    rail[1 TO n]; post[1 TO n]; centre
    ESET;
```

In other contexts members of a Set are inserted or removed thus:

```
PUT <item> INTO <set>
TAKE <item> FROM <set>
```

Two special loop statements have been provided for accessing the members of a Set:

```
FORALL <name> IN <set> DO <stmnt>
FORMEM <name> IN <set> DO <stmnt>
```

The name is implicitly declared as an Item with the scope of the loop body, which is executed for each successive Item. The difference between them is that FORMEM accesses on ly the top members of the set, whereas FORALL accesses all the objects at the lowest level contained within sets within the set, recursively.

```
        ! test if a set contains an
Item;
    BOOL PROCEDURE member(ITEM e; SET s)=
        FORALL m IN s DO
            IF m EQL e THEN RETURN TRUE;
            !NB exhausted loop has value false;

        SET PROCEDURE union(SET s1, s2) =
            BEGIN SET s3=s1;
            FORALL m IN s2 DO
                IF NOT member(m, s1)
                    THEN PUT m INTO s3;
            RETURN S3
            END;
        !procedure to enter Attributes in
inverted list form so that all objects with a
particular Attribute may be directly accessed;
        PROCEDURE collect(ATTRIB SET att
            in ITEM obj is SET attset)=
                BEGIN
                obj: {att←attset};
                PUT obj INTO attset
                END;
```

9. SPATIAL SEARCH:

Accessing objects by their location in space is an important method of access to a database representing a physical system. No general methods of database organisation are sufficient to perform this efficiently, but a number of special techniques have been developed to deal with it [9]. In Glide the function

```
        FIND <poly>
```

will generate the Set (which may be empty) of objects in the vicinity of the given Polyhedron.

```
        SET PROCEDURE findlap(ITEM i)=
                BEGIN SET conflicts = FIND i;
                FORMEM m IN conflicts DO
                IF (LAP m WITH i) EQL NULL
                THEN REMOVE m FROM conflicts;
                RETURN conflicts
                END;
```

10. GRAPHIC DISPLAY:

To obtain graphic output of the Shapes and arrangements of objects, two things are necessary: A specification of the VIEW and a specification of the DISPLAY SET. The View is a Copy of a system defined Form named VIEW, which contains the parameters needed for graphic display. This has a Shape consisting of a two vertex line, representing the reference point and view point. The reference point and line of sight of a View Copy is specified by its location and orientation. [The point of view can also be changed by the FROMX, FROMY and FROMZ Attributes.] Other Attributes control other parameters of the View. The default definition of the View is:

```
    ATTRIB BOOL PERSP,SECTION, HIDLINES;
    ATTRIB REAL CONE,FROMX,FROMY,FROMZ;
    FORM VIEW=BFORM
        PERSP←true;      !Perspective ororthographic;
        SECTION←false;   !section to be cut;
        CONE←2.0;        !cone of vision;
        HIDLINE←false;   !hidden line removal;
        FROMX←100;   !default view along X axis;
        FROMY←FROMZ←0;
        SHAPE = BPOLY BTOPO CVE(1,1) ETOPO;
                VX[2]=FROMX; VY[2]=FROMY;
                VZ[2]=FROMZ
                EPOLY
        EFORM;
```

```
        ITEM plan, birdseye;
        plan←COPY VIEW={\90; PERSP←false};
        birdseye←COPY VIEW={FROMY←500;FROMZ←40};
```

GLIDE provides two special Item variables: LOOK and OSET. To LOOK the user assigns the required current View (or Set of Views if several simultaneous views are wanted as in the traditional three orthographic engineering drawings:

```
        LOOK←{plan; front; side; birdseye};
```

OSET contains the current Set of Items to be displayed. Thus the basic DISPLAY commands can be defined as follows:

```
        DISPLAY x          PUT x INTO OSET
        ERASE y            TAKE y FROM OSET
        CLEAR              OSET←NULL ;
```

11. GRAPHIC INPUT:

GLIDE contains special functions which return information from a graphics satellite process which can read data from graphical input devices such as digitiser or light pen. LOCA returns a 3-Dimensional location in project coordinates and and PICK returns a reference to an Item which has been selected on the display.

```
        MOVE PICK TO LOCA;
```

In an environment with dynamic graphics these procedures pass control temporarily to the display process, which can allow considerable dynamic interaction before returning the final values to the GLIDE executer. These functions hopefully provide GLIDE programs with a degree of clevice -independence.

12. DECLARATIONS, BLOCK STRUCTURE AND SCOPES:

It expected that multiple users will access a design database. It would be impractical to anticipate all the variable names to be used in some project. Also, it must be possible to define application procedures independent from the database contents. Some procedures, however, should be able to add new records directly to the database and provide names for them.

Within this context, we believe that explicit declarations with local and global scope distinctions within a hierarchical block structure are the best means for allowing unambiguous referencing and efficient memory management. A design project may last over many sessions at the terminal. Conceptually the scope of a project corresponds with the outermost block level. This outer block begins at the initiation of a project and is the outer block level during a terminal session. Therefore entities declared at this top level are GLOBAL and continue to exist between sessions over the length of the project.

The scope of the name of a variable or record is the range of code over which it can be referenced by that name. The default scope of declarations is Local to the block in which they appear. However it is possible to override the default local scope by prefixing the declaration as GLOBAL. In GLIDE, definition blocks for all kinds of records can contain general statements including declarations and are as much part of the scope block structure as BEGIN ... END blocks. Declarations are not constrained to be at the beginning of the block.

Local variables of simple type (both scalar and vector) are allocated on a stack and hence are deallocated on exit from the block in which they were declared. Records on the other hand are allocated on a heap, managed by the database system. Even though they are no longer accessible by name after exit from a block in which they were declared as local, they may still be referenceable via any Global Sets or Item variables to which they have been assigned. For example any Items created within a Set definition block will automatically be put into that Set. Garbage collection of such records is managed by keeping a reference count for each one, and deleting only when this falls to zero.

13. FINAL REMARKS:

We have described only the major features of GLIDE. Other constructs are provided for moving Items, to change the coordinate axes for local detailing, for Copying Sets, and for input and output of alphanumeric data. In addition the Set type includes features to facilitate the definition of hierarchical and network database organizations. The language as we have descibed it here is a preliminary version, and it will evolve no doubt as feedback is gained from users, both designers and expert programmers.

We expect GLIDE to be used as a laboratory for developing a variety of design information systems. Initial application areas under investigation include building design, with interfaces to a variety of performance analyses, and the design of chemical process plants.

We have said little about the implementation and operating environment. However these are particularly critical to an interactive database system, since speed and convenience are essential if the full advantages of direct interaction are to be obtained. The system is modular and should allow investigation of alternative representation schemes. Thus the GLIDE interpreter could be interfaced with a standard database management system, or the shape modeling structures could be extended to include curved objects. In this way, we expect to use it both in the development of new CAD applications, and also as an experimental tool for developing new forms of support software.

This work was supported by the National Science Foundation, Division of Mathematical and Computer Sciences.

REFERENCES:

(1) Baer A., C.M. Eastman and M. Henrion, "A survey of geometric modelling systems", Institute of Physical Planning Research Report No 66, Carnegie Mellon University 1977.

(2) Baumgart, B. G., "A Polyhedron Representation for Computer Vision," Proceedings of the National Computer Conference, 1975.

(3) Braid, I.C. and C. Lang, "The design of mechanical components with volume building bricks" COMPUTER LANGUAGES FOR NUMERICAL CONTROL,. J. Hatvany, ed. North-Holland Publishing Co. London, 1973.

(4) Brun JM "EUCLID: Manual", Equipe graphique du LIMSI, B.P. 30, Orsay, France.

(5) Chalmers, J., "The development of CEDAR" International Conference in Computers in Architecture, York, 1972.

(6) Eastman,C. and M. Henrion, "Language for a Design Information System", Institute of Physical Planning Research Report No. 58, Carnegie-Mellon University, February, 1976 (revised).

(7) Eastman C, "The concise structuring of geometric data for CAD" in DATA STRUCTURES FOR PATTERN RECOGNITION AND COMPUTER GRAPHICS. A Klinger, H fu and T Kunii (eds) Academic Press 1976a.

(8) Eastman, C. "General Purpose Building Description Systems", COMPUTER AIDED DESIGN, 8:1(January, 1976c) pp.17-26.

(9) Eastman, C. and J. Lividini, "Spatial Search", Institute of Physical Planning Research Report No. 55, Carnegie-Mellon University, May 1975.

(10) Engeli, Max. "A language for 3D graphics applications" INTERNATIONAL COMPUTING SYMPOSIUM 1973, North Holland Press, 1974.

(11) Hosaka, M., T. Matsushita, F. Kimura and N. Kakishita, "A Software System for Computer Aided Activities", Proceedings IFIP W.G.5.2 Conference on CAD Systems, Austin, Texas, 1976.

(12) Hoskins, E.M.,"Computer aids in building",COMPUTER AIDED DESIGN, J.J. Vlietstra and R.F. Weilinga (eds.) American Elsevier, N.Y. 1973.

(13) Lafue, G. "Recognition of Three-Dimensional Objects From Orthographic Views", SIGGRAPH NATIONAL CONFERENCE PROCEEDINGS, ACM, N.Y., 1976, p.103-108.

(14) Newell, M. and D. Evans, "Modeling by Computer", Proceedings of the IFIP W.G. 5.2 Conference on CAD Systems, Austin, Texas, February, 1976.

(15) Newman, W. "Display Procedures" CACM 14,No 10 651 Oct 1971

(16) Production Automation Project "An introduction to PADL", TM-22, University of Rochester, NY 1974

(17) Shu, H. H., "Geometric Moleling for Mechanical Parts" 4th NSF/RANN Grantees' Conference on Production Research and Technology, Chicago, November 1976.

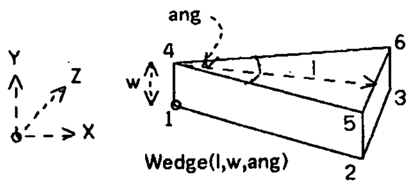
(18) Sutherland, I. "Three Dimensional Data Input by Tablet", PROCEEDINGS OF THE IEEE, 62:64, (April ,1974).

(19) Thornton, R., "MODEL: Interactive Modeling in Three Dimensions Through Two-Dimensiona Windows" unpublished, MS thesis, Cornell 1976.

(20) Brainin, Jack "Use of COMRADE in engineering design", 1973 National Computer Conference, AFIPS Press, Montvale NJ,1973

EXAMPLE:

Below is a simple example of some GLIDE code to generate the Spiral staircase shown in the illustration. It uses as primitives some of the procedures given previously in the text. "Spiral.step" is a parameterized shape procedure which constructs a Polyhedron from three elements: "Plate" is the surface of the step, "support" is a bracket and "Collar" attaches' the assembly to the "centre" column, which is passed as a parameter. These are welded together with the shape operator, COMBINE, and then the "centre" column is CUT out of the "collar" so that it fits closely around it. The "spiral.stair" procedure computes the number of steps and exact riser height to make the total height and calls "Spiral.step" to create an appropriate step. Repeated Copies of this are made at successive locations, and the procedure returns the "centre" and all the steps and the centre as a Set. It constitutes a simple detailing routine for creating a class of spiral staircases. The particular example illustrated is created as "stairl".



```
POLY PROCEDURE wedge(REAL l,w,ang)=
     BPOLY tria;
     VZ[3,6]= -VZ[2,5]= l*TAN(ang/2);
     VX[2,3,5,6]= l; VY[4 TO 6]= w
     EPOLY;
```

```
      POLY PROCEDURE spiral.step(POLY centre;
            REAL riser,radius,r,angle,th)=
         BEGIN
         POLY support =
            triangle(radius*0.95,-riser*0.8,th);
         POLY collar = column(12,riser,r);
         POLY plate = wedge(radius,th,angle);
         ! return the result of shape operations;
         CUT centre FROM  COMBINE collar WITH
                COMBINE support WITH plate
         END;

! To make spiral staircase, (dimensions in inches)
      SET PROCEDURE spiral.stair(ht,radius,angle)=
         BSET; INTEGER numsteps; REAL riser;
         numsteps ← ht/8.0;
         riser ← ht/numsteps;
         POLY centre = column(12,ht+32.0,5.0);
         POLY step = spiral.step(centre,
            riser,radius,3.0,angle,0.625);
         FOR i TO numsteps
            DO COPY step={0,riser*i \0,angle*i}
         ESET;

      SET stair1 = spiral.stair(100.0,46.0,30.0);
```