This paper argues that there is a need for a problem-oriented language to handle three-dimensional geometric information, and proposes a set of language facilities that illustrate how this need should be met.

The emphasis is on the facilities needed for describing solid objects and their placement in space, and for defining and operating on configurations of objects.

INTERACTIVE GRAPHICS IN DATA PROCESSING

A language for three-dimensional geometry

by P. G. Comba

Many branches of engineering and technology give rise to problems of a geometric nature that cannot be conveniently formulated and solved with currently available programming languages and systems.

In this paper, we discuss in some detail one such problem (often referred to as the *placement problem*), we formulate the general requirements for a language that addresses itself to problems of this type, and we present an outline of the specific features that this language should have. For convenience and brevity, the proposed language is called Geometry Language (GL). A fuller set of specifications for GL may be found in Reference 1. It should be pointed out that GL has not been implemented.

Language requirements

placement problem

The so-called placement problem is a major and recurring problem faced by the designer of physical systems (e.g., ships, chemical plants, jet engines). It arises when a large number of objects of many different shapes (components, subsystems, pipes, cables) have to be positioned in a restricted space in such a way that no two objects are assigned to the same space.

The restriction that two objects cannot occupy the same space at the same time is, of course, a basic property of physical bodies. The problem of verifying that the plan of a system satisfies this *no-overlap* condition can be very difficult. Many techniques have

been used in attempting to solve this problem,[2] and some of them have been partially successful in special situations. For example, one can subdivide the whole design space into a set of cubes and keep track of which cubes are occupied and which are free. The trouble with this approach is that either it is too coarse or it generates an exhorbitant amount of data. Both of these objections are avoided by the geometric and hierarchical approach of GL.

Another aspect of the design process is the use of different media to *represent* the system being designed. At least three levels can be distinguished:

- Physical: scale models
- Graphic: sketches, drawings, blueprints
- Symbolic or abstract: equations, formulas, geometric concepts

The Geometry Language is a tool for working at the symbolic or abstract level.

If a scale model were used as the main design tool, the placement problem would vanish: since different physical elements of the model cannot occupy the same space, neither can the corresponding elements of the system. It is obvious, however, that this approach is impractical. In a typical design project, the specifications of the components and subsystems undergo a large number of changes before they reach final form, and many people must have access to those specifications. A scale model, on the other hand, is expensive and time-consuming to build and modify, not to mention the problems of duplication and dissemination. Furthermore, it may be difficult to read dimensional information off a scale model without taking it apart.

The graphic approach to the placement problem is also inadequate, since it is impossible in general to represent fully a three-dimensional object with curved surfaces on a flat surface such as a sheet of paper. A partial exception to this statement is the technique of stereographic three-dimensional views, developed by Strauss and Poley[3] to produce "wire-frame" representations of systems of pipes. If this technique can be extended to more complicated systems of objects, it is very likely that it will require an input language with many of the features of GL.

We are thus led to the abstract approach. The feasibility of this approach has been recently demonstrated;[4] specifically, it has been shown (for a certain class of geometric shapes) how the no-overlap condition can be formulated and tested directly in terms of the equations of the surfaces bounding the objects.

language design criteria

The placement problem is an instance of a larger class of problems of space allocation, partitioning, and accessibility that occur in engineering and architectural design. It is the author's contention that *the most natural language for handling such problems is one that deals directly with the geometric properties of space.*

If the designer's mind thinks in terms of planes, spheres, and cylinders, the language should deal with planes, spheres, and cylinders, and with the properties of these objects. In other words,

the symbols used in the language should designate geometric objects and relations rather than graphic constructs. (Whether these symbols should be alphanumeric characters or pictographs is a separate question, to be discussed below.)

The advances in engineering and technology in the last decades have made it possible to design systems of great complexity where hundreds of people are involved in the design process. This has brought about new problems of control and documentation: what space belongs to whom, and who has placed what where. *A language for geometric processing must have facilities for defining and handling sets and configurations of spacially and functionally related objects.*

written vs. pictorial form

The question whether the symbols of the language should be alphanumeric characters, or graphics elements such as points, lines, and curves, is related to the question whether the language is intended to be used in a batch mode or in an interactive mode. The point of view taken here may be summarized as follows:

- Both modes are necessary. Many geometric problems can be adequately expressed in a programming language where one writes statements on coding forms; it is then unnecessary and wasteful to use a graphics console. Conversely, there are problems where interaction is essential and where it is more natural to draw with a light pen or a stylus and to point to the elements of a figure.
- Although the "written" and the "pointing" language are intended for different uses, one can establish a correspondence between the elements (variables, commands, etc.) of the two languages. In fact one can think of them as two forms of the same language. For this reason, only the written form of GL is discussed in this paper. Besides, the emphasis here is on what the language can do, i.e., on its meaning rather than its form.

GL and COGO

The approach advocated here can be contrasted with the design philosophy of the widely used Civil Engineering Coordinate Geometry System (COGO).[5]

COGO deals primarily with points, straight lines, angles, and circular arcs, whereas GL deals with solid objects and the surfaces bounding them.

An engineer working with COGO "writes the description of his problem and how to solve it as if he were solving it by hand"; in other words, he specifies a linear sequence of instructions. By contrast, GL is intended to be imbedded in a full-fledged programming language, with facilities for branching, looping, subroutine linkage, and interrupt handling.

COGO operates on a simple data structure, whereas GL requires a complex data structure.

why three dimensions

Having established the need for a system for manipulating three-dimensional geometric figures, the question arises whether one should design a more general system to handle n-dimensional geometry; this system could then be used to solve problems in 2-, 3-, or 4-dimensional space.

While this approach is appealing, it is quite impractical, as shown by the following considerations of efficiency and relevance.

Efficiency. If the dimensionality n of the space is treated as a parameter to be set at execution time, the allocation of storage space for the data structures on which the system operates becomes more complicated. For example, the coordinates of a point would require a variable amount of storage depending on n. There is also another source of inefficiency: if all the problems that the system is intended to solve have to be formulated and coded in n dimensions, many subroutines will be much harder to write and test, and more time-consuming at execution time.

Relevance. Many problems in three-dimensional geometry become either trivial or irrelevant or incompletely defined in a space of different dimensionality. For example, questions of visibility and calculations of shadows are relevant only in three-dimension, since the world in which we live is three-dimensional. A two-dimensional space (i.e., a plane) should not be viewed as a special case of a three-dimensional space, since a three-space contains infinitely many planes. (A system for three-dimensional geometry should, of course, provide facilities for defining planes and for computing cross-sections of three-dimensional objects. The point here is that if the natural way to formulate a problem requires only two-dimensional concepts, the problem should be solved by using a two-dimensional system.) In the opposite direction, the claim is sometimes made that problems involving time, as in kinematics and dynamics, require four-dimensional capabilities. Actually the time variable occurs only as a parameter and it does not interact with the space variables in the way these variables interact with each other. For example, while it is meaningful to rotate an object in space, it is usually meaningless to rotate it in space-time.

outline of language capabilities

The global entities on which the language operates are called *models.* A model is a data set containing a representation of a region of space and (in that region) of a system or assembly of graphic and/or physical objects.

The process of building a model is analogous to that of creating or updating a file. Data is written in a file according to a specified format; analogously, geometric data is placed in a model with respect to a specified coordinate system.

The actual steps involved are the following:

1. Defining a set of objects of type *geometric.* This is done by means of a set of built-in *primitive geometric functions,* which can be combined to form geometric expressions, whose value can be assigned to geometric variables.
2. Optionally defining one or more coordinate transformations, using the *transformation* statement. These two steps are independent of the specific model being built.
3. Defining or retrieving the model being operated on, and optionally a submodel (or configuration) within it. The data type

model and the *configuration* statements are provided for this purpose.

4. Optionally selecting a subregion of the model space for the purpose of qualifying the drawing and placement statements and the interference checking (Step 7). This is done with the *region* statements (the keywords are IREGION and WREGION).
5. Optionally defining and selecting sets of (already drawn or placed) objects for further qualifying the interference checking and graphic display. This is done with *set* variables and *set* assignment and selection statements (the keywords are SET and ISET).
6. Optionally selecting a coordinate transformation, to facilitate the next step. The *coordinate* statement is used for this.
7. Designating certain geometric objects to represent graphic or physical objects and incorporating these representations into a configuration. This essential step is done with the *drawing* and *placement* statements, respectively. The term *component* is generally used hereafter to mean "the representation of a (graphic or physical) object." The distinction between geometric, graphic, and physical is discussed more fully below. Two kinds of tests are performed when an attempt is made to draw a graphic component or place a physical component:

   - the component must lie in the selected working region.
   - if it is a physical component, it must not interfere with (i.e., overlap) any other physical component in the selected interference set and in the selected interference region.

   If either test fails, the drawing or placement statement is not executed and an interrupt takes place.
8. Displaying a set of components in a selected display region, with various options as to type of projection, shading, appearance of hidden boundaries, etc. This is done with the *display* statement. Further options provide for saving display-generated data.
9. Optionally using the *attach* and *merge* statements to combine independently defined configurations and models.

The interrelation of the concepts introduced in Steps 1 through 7 is illustrated in Figure 1. Each entry depends on (i.e., is defined or executed with reference to) the entities pointing to it.

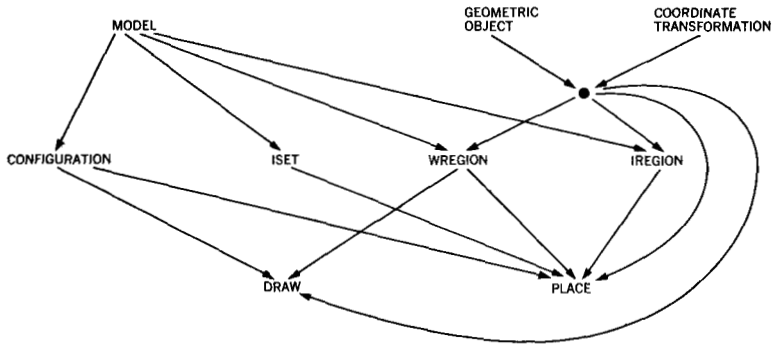
remarks on language capabilities

Only the geometric and modeling capabilities of GL are described in this paper. It is assumed, however, that the full capabilities of a higher-level language such as PL/I are available. In fact GL has been designed as a possible set of extensions to PL/I.

Whenever an available option is not used, a *default* definition is provided for it. For example, if no coordinate transformation has been selected, all operations involving coordinates are executed with reference to an implied absolute coordinate system.

The concept of *selection* statement deserves further discussion. A selection statement (for selecting a coordinate transformation or

Figure 1 Interrelation of concepts



an interference set, for example) may be thought of as setting a new default condition that will govern the subsequent execution of certain other statements (e.g., a PLACE statement). It must be noted, however, that selection statements are used more extensively in GL than in conventional programming languages. Thus in the analogy between building a model in GL and writing a file (say in PL/I), the analogue of a GL selection statement would be a statement that selects a FORMAT so that subsequent WRITE statements are governed by that FORMAT without explicitly referencing it. There are several reasons for having selection statements in GL rather than following the alternative approach of requiring each DRAW and PLACE statement to refer explicitly to the conditions by which it is governed:

- As can be seen from Figure 1, the DRAW, and especially the PLACE, statements are subject to many conditions, and it would be quite unwieldy to have to reference or specify them in each such statement.
- In most programs, there is expected to be a high ratio of DRAW and PLACE statements to selection statements: in other words, the conditions affecting successive DRAW and PLACE statements are relatively stable. This is due both to the nature of the computer-aided design process and to the fact that each DRAW or PLACE can operate on only one object. The latter fact in turn is determined both by the considerations in the next paragraph, and by the need for a simple interpretation of an interrupt occurring during the execution of a DRAW or PLACE.
- The written form of the language must contain the facilities that are needed in the pointing (conversational) form. In a conversational language, the statements should be designed for simplicity of form and richness of implied or contextual meaning.

The basic distinction between *geometric*, *graphic*, and *physical* entities can be viewed in several ways. Functionally they are used in different ways in the language, as explained in later sections: geometric objects are defined independently of a model, whereas

graphic and physical components are instances of geometric objects attached to specific configurations, hence to specific models; and physical components are the only ones for which interference testing is done. Intuitively a geometric object is an abstract entity; a graphic component is like a figure drawn on a piece of paper, possibly for reference or construction purposes; and a physical component is a figure drawn for the purpose of designating a physical object.

Geometric manipulation

A formal definition of the main features of GL is now presented. A fuller definition may be found in Reference 1. Where necessary, the so-called SRL[6] notation is used for displaying the form of the statements in the language.

The attribute TRIPLE may be used for defining arrays of three arithmetic scalar elements; it is introduced merely as a convenience.

coordinate transformations

The coordinate transformation selection statement has the form

COORDINATE trans;

where "trans" has the form

$$\left\{\begin{array}{l}\text{TRANSFORMATION ([t1], [t2], [s] [,trans])}\\ \text{label-designator}\end{array}\right\}$$

The first form of "trans" may be termed an immediate transformation designation, and the second form a remote designation. If the immediate form does not contain the optional clause ",trans" it is a simple transformation, defined with respect to the absolute coordinate system of the space of geometric objects; if it does contain the clause ",trans" it is a compound transformation, i.e., the transformation specified by the first three arguments is defined with respect to the coordinate system defined by the ",trans" clause. In the remote form, the label designator must designate the label of a TRANSFORMATION statement (defined below). These definitions make it possible to compound transformations, both immediately and remotely designated, to any depth.

As to the first three arguments of the transformation, t1 is a triple expression (i.e., an expression of type TRIPLE) that defines the origin of the coordinate system, t2 is a triple expression interpreted as a triple of Eulerian angles that defines its orientation in space, and s is an arithmetic scalar expression that defines its scale. If any of these arguments are omitted, the triples are taken as zero and s is taken as 1. (The use of Eulerian angles to represent a rotation in space is discussed in most textbooks on classical mechanics: for example, see References 7 and 8.)

Like all selection statements, the COORDINATE statement conditions the executions of several other statement types (to be described in later sections) until superseded by the execution of another COORDINATE statement. If a transformation contains

nonconstant expressions, the latter are evaluated whenever one of the statements which it conditions is evaluated.

The coordinate transformation definition statement has the form

label: trans1;

where "trans1" is the immediate form of "trans" defined above.

geometric expressions

The attribute GEOMETRIC is used for declaring identifiers of data of type geometric. The latter can be scalars, arrays, function procedures, and formal parameters.

There are no geometric constants. The effect of geometric constants can be achieved by using the built-in primitive geometric functions with constant arguments.

Geometric elements can be combined to form geometric expressions. A geometric expression is any of the following:

- A geometric variable
- A geometric function reference
- An expression enclosed in parentheses
- Any two expressions connected by one of the infix operators +, *, /

The meaning of the operators (listed here in order of decreasing binding strength) is:

/ Relative complementation
* Intersection
+ Union

Thus A/B is that portion of space contained in the geometric figure A but not in the geometric figure B. Similarly, intersection and union are interpreted as referring to the sets of points comprised by the geometric entities. Mixed expressions of geometric and other data types are meaningless.

The evaluation of a geometric expression has a meaning different from the case of an arithmetic expression. The evaluation of a (scalar) arithmetic expression involves the successive execution of a sequence of arithmetic operations, from which a single value is produced. The evaluation of a geometric expression involves the setting up of a data structure as well as the performance of numerical calculations; the entire data structure is the value of the expression.

built-in geometric functions

The built-in functions provided by the system for handling geometric information can be grouped into four classes, depending mainly on the type of the function arguments and values. This is shown in Table 1.

The first class contains a few functions that facilitate the handling of triples of numbers.

The second and most important class is that of the *primitive geometric functions* on which all geometric constructions are based. The execution of a primitive geometric function results in the creation of an internal data structure, where information is kept

Table 1

Function arguments	*Function value*
arithmetic	arithmetic
arithmetic/point	geometric
geometric	arithmetic
geometric	geometric

Table 2 Examples of built-in geometric functions

Genus of function value	*Function*	*Meaning*
arithmetic to arithmetic		
t	TR(r,r,r)	Triple whose elements are the arguments.
t	DIR(t)	Normalized triple, i.e., direction cosines corresponding to argument triple.
r	LENGTH(t)	Square root of the sum of the squares of the elements of the triple.
arithmetic or point to geometric		
(no genus)	NULL	The null object; used for freeing storage.
p	PT(r,r,r)	Point with given coordinates.
p	PT(t)	Point whose coordinates are the given triple.
s	SEG(p,p)	Line segment (directed) from first point to second point.
line	LINE(p,p) LINE(p,t)	Directed line defined by two points. Directed line defined by point and direction.
triangle	TRIA(p,p,p) TRIA(p,t,t)	Triangle with given vertices. Triangle defined by point and two vectors issuing from point.
plane	PLANE(p,p,p) PLANE(p,t)	Plane defined by three points. Plane defined by point and direction of normal.
sphere	SPHERE(p,r)	Defined by center and radius.
cone	CONE(p,t,r)	Right circular cone with center of base at p, height t, and radius of base r.
half space	HSPACE(p,t)	Defined by point and direction.
full space	FULL	Needed for taking complements.

Table 2 Examples of built-in geometric functions (cont'd)

Genus of function value	*Function*	*Meaning*
geometric to arithmetic		
t	TR(p)	The triple of the coordinates of a point.
t	TR(s)	The triple of the components of a segment.
t	DIR(s)	This function is defined for all argument genera to which a direction is associated.
r	LENGTH(s)	Length of segment.
geometric to geometric		
geom	COPY(geom)	Produces copy of object.
geom	AFFINE(geom,t,t,t,t)	Affine transformation of geometric object. The first 3 triples are the column vectors of the transformation matrix, the last one is a displacement vector.
geom	BDARY(geom)	Boundary of the geometric object.

Abbreviations used in this table:

type	*genus*	*abbreviation*
arithmetic	real	r
	integer	i
	real array of dimension 3 (triple)	t
geometric	point	p
	segment	s
	(other)	(no abbreviation)

about the genus of the corresponding geometric object, its component parts, their interrelations, and the numerical values associated with these components. The concept of data structure is discussed in the paper by C. I. Johnson elsewhere in this issue. The value returned by a primitive geometric function, besides being of type geometric, can be characterized by its genus. (By analogy, a number is of type arithmetic but has additional characteristics, e.g., scale and precision.)

The third class comprises such functions as the length of a segment.

The fourth class may be regarded as a set of transformations of geometric objects into geometric objects.

Examples of built-in geometric functions are shown in Table 2. The abbreviations used for specifying the type and genus of arguments and values are indicated at the end of the table.

The geometric assignment statement has the form:

geometric assignment statement

geom-variable [,geom-variable] . . . = geom-expression [,trans];

The geometric variables on the left can be simple or subscripted variables. The assignment statement produces the following sequence of events:

1. The geometric expressions on the right-hand side are evaluated;
2. If the optional clause ",trans" is used, the designated transformation is evaluated and applied to the value of the expression; the resulting value is then assigned to the identifier(s) on the left-hand side;
3. Otherwise, if a transformation has been selected by the previous execution of a statement, that transformation is evaluated and applied to the value of the expression; the resulting value is then assigned;
4. Otherwise, the value of the expression is assigned.

The geometric assignment statement can be executed only if its right-hand side contains only references to variables which have already been assigned values, or constants (including variables and constants appearing as function arguments). In general, the value of a geometric expression is a data structure, hence it is possible to change the value of a variable without the explicit assignment of a new value to it. For example, if the geometric assignment statement

A = B + C;

is followed by a statement that changes the value of C, the latter statement changes the value of A as well. If one wishes to avoid this, one can use the COPY function and replace the first assignment statement by

A = COPY(B + C);

user-defined geometric functions

The user can define function procedures of type geometric by using the normal formalism for procedure definition. Functions defined in this way differ from the primitive geometric functions in that they do not have a genus; in this respect, they resemble PL/I structures, which can comprise items of heterogeneous characteristics.

Model building

models and configurations

Since GL is defined as a set of extensions to a higher-level language (such as PL/I), it is assumed that adequate facilities for the defining, opening, and closing of files are available in that language. It is further assumed that a new file organization attribute, GRAPHIC, can be added to the language. A file with that attribute is henceforth called a *model*.

All the statements to be described in this and the following sections, except attaching and merging, can only be executed with

respect to a designated model. This designation is done with the model selection statement

MODEL identifier;

where the identifier is a model name.

The structure of the data in a model is described in terms of *configurations*. A configuration is intended to represent a grouping of functionally related objects and can be represented by a tree. An outer configuration, i.e., one not contained in other configurations, is a model. This is illustrated in Figure 2 where the dots represent configurations, the uppermost dot represents a model, and the circles represent components.

The configuration statement has the form:

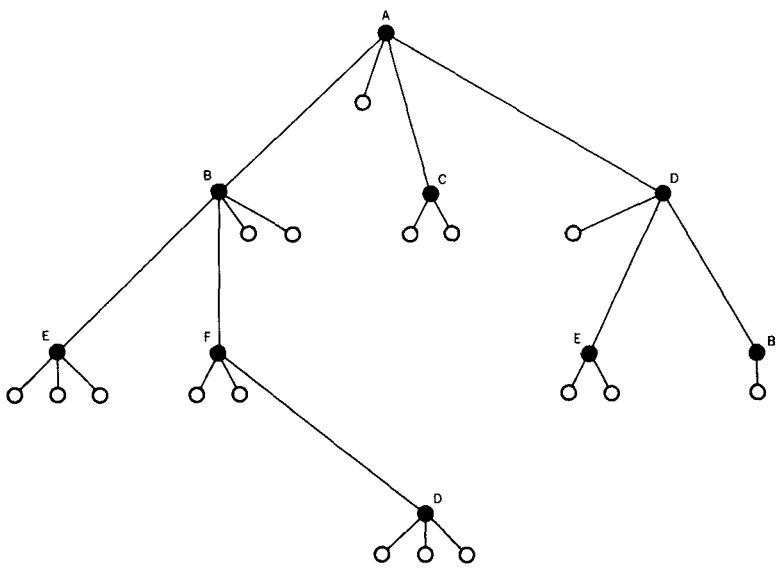
CONFIGURATION identifier;

The identifier must designate a unique configuration within the selected model; if necessary, this can be achieved, as in PL/I, by name qualification. The function of the statement is:

- If the identifier is the name of a configuration within the selected model, that configuration is selected.
- If not, a new configuration within the model is defined and named, and selected.

The execution of subsequent drawing and placement statements has effect with respect to the selected configuration, i.e., any components drawn or placed become elements of that configuration. If no configuration statement is given, the model itself is the selected configuration.

Figure 2 Configuration tree



It follows from the above that a configuration does not require a declaration independent of its selection.

sets

Set is a new type of datum which makes it possible to define and name arbitrary collections of components within a model. The attribute SET applies to identifiers of this type.

A set-expression is an expression formed by using the operators +, *, /, and the parentheses for grouping. Each operand in a set expression can be a set, a component (interpreted as a set of one element), or a configuration (interpreted as a set of elements). The operators, listed in order of decreasing binding strength, have the following meaning:

/ Relative complementation
* Intersection
+ Union

It is important to note that these operators are interpreted as operating on sets of elements, not on the elements themselves. For example, if A and B are overlapping geometric objects, their geometric intersection is not empty; whereas, if A1 and B1 are graphic components corresponding to A and B, the intersection of the set consisting of A1 and the set consisting of B1 is empty.

The set assignment statement has the form:

set-variable [,set-variable] . . . = set-expression;

The set variable may be simple or subscripted.

The interference set selection statement, which conditions the executions of subsequent placement statements, has the form:

ISET set-expression;

regions

A region is defined by a geometric expression (which should normally represent a three-dimensional object).

Region selection, which conditions subsequent drawing and placement statements, is done with the statements:

WREGION geom-expression [,trans];
IREGION geom-expression [,trans];

(for "working region" and "interference region"). The execution of a region selection statement is similar to the evaluation of the right-hand side of a geometric assignment statement: it is conditioned by the currently selected coordinate transformation unless overruled by the ",trans" clause.

The drawing statements have the following form:

drawing, placement, interference testing

DRAW (variable = geom-expression [,trans]);
REDRAW (variable [,trans]);
ERASE (variable);

The variable, which is also termed a *component name,* may be simple or subscripted, and it need not be declared in a separate statement. The execution of a DRAW statement involves the following steps:

1. The geometric expression is evaluated and the value transformed, as in the case of the geometric assignment statement.
2. The final (transformed) value is tested to determine whether it lies in the WREGION; if no WREGION has been selected, the model region (which is implementation defined) is used for the test. If the test fails, the execution of the statement is interrupted. The on-condition raised by this interrupt is called BOUNDS (see below).
3. If the test is passed, the value is assigned to the variable. If the latter had previously designated a component in the currently selected configuration, this assignment is an updating of its value; if not, the variable is entered as a new component name in the currently selected configuration.

The assignment that takes place in a DRAW statement differs from the geometric assignment in that subsequent changes in the value of the expression do not affect the value of the component (as discussed earlier).

The REDRAW statement is executed as follows:

1. If the variable does not designate a graphic component in the currently selected configuration, the statement is invalid.
2. If the statement is valid, a copy of the component is made and transformed according to the ",trans" clause if that clause is present, otherwise according to the currently selected transformation.
3. The transformed component is tested to determine whether it lies in the WREGION. If the test fails, a BOUNDS interrupt occurs and the variable still designates the original component.
4. If the test is successful, the transformed component is assigned to the variable, replacing the original one.

The ERASE statement, subject to Condition 1 of the REDRAW statement, removes the component designated by the variable from the configuration.

It should be clear from this discussion that the drawing statements do not cause a drawing to appear on a console: their purpose is to create a data structure that represents a drawing. To display a drawing one must use a *display* statement.

The placement statements have the same form as the drawing statements except that the keywords PLACE, REPLACE, and REMOVE are used. The function of these statements is also analogous to that of the drawing statements, except that the objects being operated on are *physical components.*

The PLACE statement is executed as follows:

1. As Step 1 in the DRAW statement.
2. As Step 2 in the DRAW statement.
3. If the WREGION test is passed, the interference test is executed. This consists of determining whether the final value of the geometric expression overlaps certain physical components or portions of them, i.e., whether it occupies space already occu-

pied. The interference test is done with respect to those parts of the components of the selected ISET which also lie in the selected IREGION. If no ISET has been selected, the entire model is implied. If no IREGION has been selected, the model region is implied. If the test fails, the execution of the statement is interrupted and the on-condition CLASH is raised.

4. If the interference test is passed, the last step is like the third one in the DRAW statement, with "graphic" replaced by "physical."

Regarding the REPLACE statement, it is best described by saying that REPLACE is to REDRAW as PLACE is to DRAW. This means that the first three steps of its execution are analogous to those of the REDRAW statement; then the interference test is executed; then the last step is as in REDRAW.

The REMOVE statement is entirely analogous to ERASE.

As to the on-conditions BOUNDS and CLASH, they are always enabled and, as in PL/I, the programmer can write appropriate on-units to be executed if the interrupts occur. If the on-units are not included, an appropriate standard system action will be defined. It should be noted that, since the DRAW, REDRAW, PLACE, and REPLACE statements operate on only one component (at a time), it is easier for the programmer to interpret any interrupts that may arise than if these statements operated on lists of operands.

attaching and merging

All model building statements specified so far refer to a single model. The intermodel facilities needed in the language are briefly described here; a complete specification is not given, as it would depend on the interface between the language and the operating system.

Two kinds of facilities are needed:

- Attaching a model (as a configuration) to a designated configuration of another model.
- Merging two or more models into a new model.

The attaching operation subordinates one model to another; the merging operation coordinates the several models. With either operation there should be the option of requesting an interference test of the type associated with the PLACE statement, and specifying appropriate ON-condition actions.

Display

Only a brief outline of the display facilities is presented here. The DISPLAY statement has the form

DISPLAY data-list [format-list];

where data-list is a list of set-expressions.

The items on the format list specify the manner in which the corresponding sets of objects are to be displayed. The following remarks describe the main options that should be available:

- Having specified (by means of the data-list) the sets of objects to be displayed, one may request that only the graphic objects in those sets be displayed, or only the physical objects, or that both be displayed but with different renderings (e.g., dotted lines vs. solid lines).
- The further restriction may be imposed that only those objects that lie in a specified region be displayed.
- There are two main ways of rendering three-dimensional objects by means of drawings: cross sections and projections; the latter being the more widely used method.
- To specify a projection, one must give a viewing point and a projection plane.
- Shading may be requested by specifying a light source. The hidden lines (i.e., those lines that are hidden by the object itself or by some other object) may be rendered as dotted lines or suppressed altogether.

Conclusion

This paper has attempted to show why and how a language for dealing with space problems must focus on the geometric and structural aspects of those problems. By providing tools for working at a fairly high level of abstraction, the proposed language enables the user to state his problems without getting bogged down in a mass of system- and implementation-dependent details.

REFERENCES

1. P. G. Comba, *A Language for Three-Dimensional Geometric Processing-Written Form*, IBM New York Scientific Center Technical Report No. 320–2923, International Business Machines Corporation, New York Scientific Center, New York, New York (Nov. 1967).
2. *Three-Dimensional Placement Routing*, E20–0119, International Business Machines Corporation, Data Processing Division, White Plains, New York (1963).
3. C. M. Strauss and S. Poley, "3DPDP: A three-dimensional piping design program." To appear in *Proceedings of IFIP Congress* in Edinburgh (1968).
4. P. G. Comba, "A procedure for detecting intersections of three-dimensional objects." *Journal of the Association for Computing Machinery* **15**, No. 3, 354–366 (July 1968).
5. Civil Engineering Coordinate Geometry (COGO), IBM Application Program H20–0143, International Business Machines Corporation, Branch Office.
6. Systems Reference Library.
7. H. Goldstein, *Classical Mechanics*, Addison-Wesley, Reading, Massachusetts (1950).
8. W. V. Houston, *Principles of Mathematical Physics*, 2nd Edition, McGraw-Hill Book Company, New York, New York (1948).