ADVANCED CONCEPTS FOR HIGH-LEVEL GRAPHICS LANGUAGES

Günther F. Schrack

Departments of Electrical Engineering and Computer Science
The University of British Columbia
Vancouver, B.C. V6T 1W5

ABSTRACT

Much effort has been expended in the past years on the specification of the CORE and the GKS subprogram system standard proposals. Both have a common intent: They standardize the input/output interface between an application program and the graphics terminals. Such systems can not, however, address the modelling problem, i.e. ways and means to create or define graphical objects. This task is at present still left to the application program.

As the demand for increasingly sophisticated graphical output rises, more effective software tools must be developed. It is the purpose of this paper to demonstrate the capability and effectiveness of a high-level graphics programming language for modelling. In particular, high-level operations required for modelling, such as the explicit definition as well as functional generation of graphical objects, and the systematic manipulation, editing, and substitution of graphical data are discussed. Several explicit examples of modelling of graphical objects with the aid of essential program fragments and their resulting output are presented, by this means simultaneously high-lighting some of the characteristics of the Language for Interactive Graphics (LIG6).

KEYWORDS: high-level graphics programming languages, modelling of three-dimensional objects, programming tools for computer graphics

In the past few years, significant accomplishments have been achieved in computer graphics on both the hardware and the software level, which, without doubt, furthered its cause and its acceptance as an effective tool in many human activities. The market which supports computer graphics has already grown at a phenomenal rate; yet it is predicted that such growth rates will continue for some time in the future.

To sustain such growth, productivity needs to be increased, particularly on the software level, since hardware prices are falling and software costs are rising. In this context, software means software creation and hence, to a large part, it means programming. Therefore, it is essential that progress occurs in programming support and in improved programming tools for computer graphics.

Advances in high-level graphics programming languages will become as important as the advances made in the past in general-purpose high-level languages. It is the purpose of this paper to show that such advances are possible by illustrating the effectiveness of a high-level graphics language. To this end, a number of advanced concepts are discussed.

The idea of high-level graphics languages is not a new one, as the survey of McLean [McLe78] shows: The first proposals were published as early as 1967. In recent years, the reports of Barth et al. [Bart82], Magnetat-Thalmann et al. [Magn81], and van Wyk [VanW81] are particularly relevant.

Advances at the Language Level

The CORE and GKS standard proposals have influenced the graphics community in numerous ways, e.g. by unifying many technical terms, by providing a perceptual model for graphics output

systems, and by clarifying important concepts such as, e.g., windows vs. viewports in a two- and three-dimensional environment. In particular, the proposals are to be credited with establishing clearly the distinction between modelling of graphical objects on one hand and their display on an output surface on the other.

The expressed task of computer graphics is the synthesis of pictures, not their analysis. Thus, modelling of objects is a challenging problem, particularly since the CORE/GKS proposals specifically exclude that activity, concentrating on the display and output of pictures.

It is becoming increasingly clear that all possible graphical objects cannot be collected into one single class of objects. On the contrary, the world of graphical objects consists of many classes which must be catered to individually. This can become possible only by providing different graphical languages or procedure packages. The individual characteristics of an object class can then be exploited to advantage. Of course, the more objects belong to a class, i.e. the larger the domain of a graphics language is, the more versatile that language will be.

The Language for Interactive Graphics, version 6 (in short, LIG6), is a language which allows modelling structured objects in two or three dimensions by means of lines or flat polygonal faces; they may be coloured or rendered in grey scale. LIG6 is a true high-level graphics language, as its characteristics will witness.

Data types.

Two system-defined data types are provided, GRAPHICAL and VECTOR. The latter functions in a supporting role, allowing triples of type REAL in the form of vector variables, constants, and casts. The usual vector operators, such as the dot and cross product, are also provided.

The data type GRAPHICAL plays a central role, allowing the naming of any simple or complex object or subobject. Its data structure is not fixed, as it can be regarded as a set of nodes were the structure of a node is a fixed record, but the number of nodes per variable can expand or contract.

Each node stores a number of attributes such as position, scaling, rotation, colour, and, additionally, two pointers. The pointers refer to other nodes; the value pointer must always be present, and may point to a graphical primitive, the super pointer may be set to NIL.

Operators

A large number of monadic and dyadic operators have been defined which create and or operate on the data structure and nodes. Among them are modelling transformation operators for geometric transformations, and modelling attribute operators which set or reset values such as colour, lightness, saturation, and pattern.

Assignment Statements

A distinction is made between three levels of programmer sophistication: basic, medium, and advanced [Ross82a]. At the basic level, the programmer models objects entirely at the syntactic level. The medium level introduces additional language features, e.g. several identification statements. At the advanced level, the programer is given complete access to the internal data representation, requiring him to understand and know the data structure thoroughly. Four different assignment statements, the synonym assignment, the copy assignment, the value assignment, and the super assignment aid in this task.

Input/Output Statements

A number of input statements, allowing interactive execution of an application program, are provided. Graphical objects can be output to several terminals such as a Tektronix 4027 colour raster display and a Printronix printer-plotter. Several output statements generate perspective projection; a camera model has been adopted in their design.

Example

Figure 1 shows a short program, defining a cube of cubes, as well as the output it created. The program is written at the basic level, as only a few

of the actually available modelling operators were required for the task. No direct data structure manipulation or interaction (input) statements are used.

```
C
C  LIG6 PROGRAM TO CREATE A 3-DIMENSIONAL CUBE OF CUBES          (Trent Wagner)
C
      GRAPHICAL A, CUBE(3), SPAR(5), ROW, TEMP, SIDE, OBJ, TEXT
C
C  First create a prototype cube with one corner at origin
C
      A :- (POLY FROM (0.0,0.0,0.0) TO (0.0,0.5,0.0) TO (0.5,0.5,0.0)
                                    TO (0.5,0.0,0.0)) <LIGHTNESS 100.0>
      CUBE(1) :- A + A <ROTX 90.0 'DEG'> + A <ROTY -90.0 'DEG'>
                   + A <TRANS(0.0,0.0,0.5), LIGHTNESS 25.0>
                   + A <ROTX  90.0 'DEG',TRANS(0.0,0.5,0.0), LIGHTNESS 50.0>
                   + A <ROTY -90.0 'DEG',TRANS(0.5,0.0,0.0), LIGHTNESS 75.0>
C
C  Next form spars which will join the cubes
C
      SPAR(1) :- LINE FROM (0.50,0.25,0.25) TO (1.00,0.25,0.25)
      SPAR(2) :- LINE FROM (0.25,0.50,0.25) TO (0.25,1.00,0.25)
      SPAR(3) :- LINE FROM (0.25,0.25,0.50) TO (0.25,0.25,1.00)
      SPAR(4) :- SPAR(3) + SPAR(3) <TRANS(1.,0.,0.)> + SPAR(3) <TRANS(2.,0.,0.)
      SPAR(5) :- SPAR(4) + SPAR(4) <TRANS(0.,1.,0.)> + SPAR(4) <TRANS(0.,2.,0.)
C
C  Form a row of cubes aligned with X-axis
C
      CUBE(2) :- CUBE(1) <TRANS(1.0,0.0,0.0)>
      CUBE(3) :- CUBE(1) <TRANS(2.0,0.0,0.0)>
      ROW     :- CUBE(1) + SPAR(1) + CUBE(2) + SPAR(1)<TRANS(1.,0.,0.)> +CUBE(3)
C
C  Form a wall of cubes (3 * 3) in Z-plane
C
      TEMP :- ROW + SPAR(2) + SPAR(2)<TRANS(1.,0.,0.)> +SPAR(2)<TRANS(2.,0.,0.)>
      SIDE :- TEMP + TEMP <TRANS(0.0,1.0,0.0)> + ROW <TRANS(0.0,2.0,0.0)>
C
C  Create final (3 * 3 * 3) object and the title
C
      OBJ :- SIDE + SPAR(5) + SIDE <TRANS(0.,0.,1.)> + SPAR(5) <TRANS(0.,0.,1.)>
                            + SIDE <TRANS(0.,0.,2.)>
      TEXT :- 'Edmonton ''83' <SCALE(6.0,6.0,1.0), TRANS(0.12,-0.50,2.30)>
C
C  Set camera model parameters and display all parts
C
      VIEW POINT = (5.5,5.5,8.0)
      AIM POINT  = (1.0,1.0,1.0)
      VIEW WIDTH =  3.00
      VIEW UP VECTOR = (0.0,1.0,0.0)
      DISPLAY (OBJ + TEXT)
      STOP
      END
```

Figure 1a

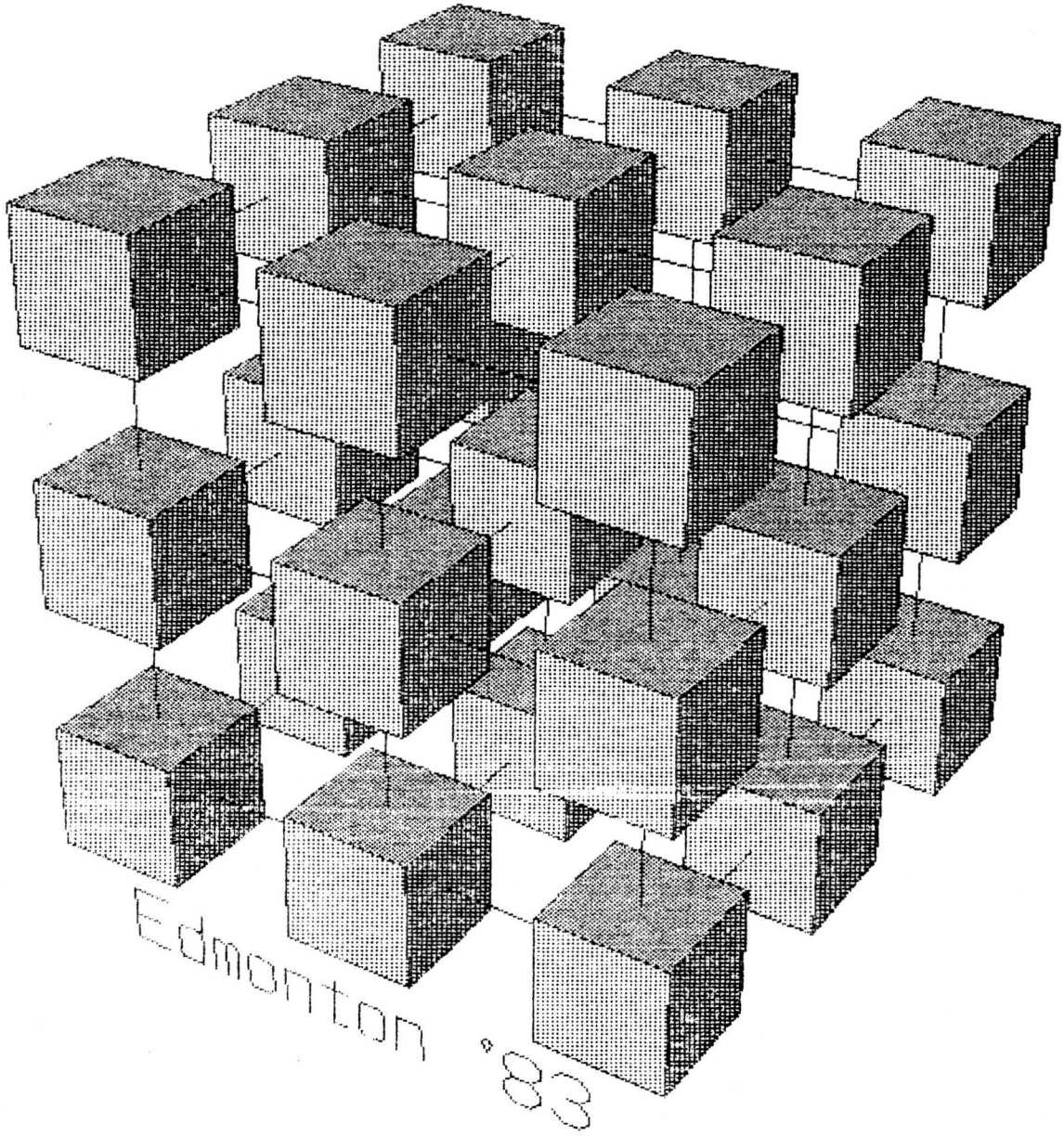

Figure 1b

Advanced Level Programming

Experience with previous versions of the language LIG led to several conclusions which influenced the design of LIG6 [Ross82b]. "Real life" graphical objects are usually very complex; indeed, there is no upper limit of complexity an object can possess. Thus, it cannot be stored in a static data structure; the structure must be dynamic, i.e. must be able to be expanded and contracted. A graphical object is often structured, and it is possible to exploit this characteristic in its definition, storage, and manipulation. The data structure of a graphical object is not unique; different data structures can represent an object with the same external representation, i.e. the same output, independent from the viewpoint specified [Requ80].

Therefore, graphics programming requires skills of the programmer which are different from other programming activities, namely to understand in detail the structures and properties of graphical objects which are to be created. This ability becomes even more important if the programmer knows only the class of objects with which his program has to deal, but not the individual objects from that class which a user will create.

For these reasons, a language must assist a programmer to a large degree. Conversely, the skilled programmer must know the programming language intimately, particularly he must know the internal representation of the objects, thus having complete control in their creation. This philosophy represents a departure from classical programming, where the programmer does not need to be familiar with the internal representation of, say, INTEGER, REAL, or CHARACTER types. Furthermore, it does run counter to recent trends in abstract data types, which suggest to hide and prevent access to "unnecessary" information of a given type.

Data Structure Manipulation

Two graphical objects can be combined into one by superimposing one onto the other. A superposition operator (symbol "+") achieves the desired effect; superposition does not change its operands. The operation expands the data base by a node which stores the modification operator values (transformation, colour, etc.); also, the value pointer and super pointer are set to point to the nodes of its operands. The data base can be accessed directly by referring to an object's identifier. Subobjects can be accessed indirectly, with either a system function or an assignment statement. Two graphical system functions, VALUE and SUPER, return values of its argument, as follows. Let P, Q, R, and S be declared graphical variables and assume that P, Q and R have been assigned graphical values. Let S be the object obtained by superposition of instaces of R and Q on P, syntactically expressed by

S :- P <mod1> + Q <mod2> + R<mod3> ...(1)

Here, <mod1> etc. indicate arbitrary attribute or transformation modifications, as, e.g., geometric transformations. The first operand of the graphical expression is an instance of P, having the topology of P but modified by the modifications <mod1>. Once defined, it can be referred to by SUPER(P) and SUPER(SUPER(P)), respectively. The language allows nesting of graphical expressions in a "natural" fashion, i.e. with the use of parentheses. For example, assignment statement (1) could be written as

S :- (P<mod1>+Q<mod2>)<mod4>+R<mod3>..(2)

This statement will have a different internal storage structure, even if the modification <mod4> is missing; it will produce a visually identical output as statement (1). Whereas execution of assignment (1) creates a data structure as shown in Figure 2, assignment (2) results in a different structure, as depicted in Figure 3.

The synonym assignment creates new nodes and sets pointers, but never copies part of a structure, as may have become clear by above examples. Therefore, care must be exercised by the programmer when redefining the value of a graphical variable: if that variable appeared in a graphical expression previously, the object of which it forms a part of will be changed as well.

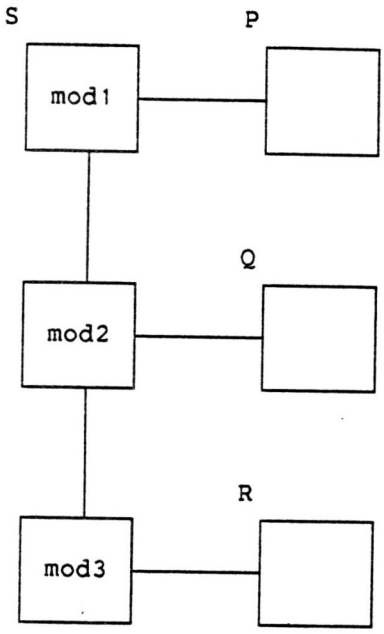

Figure 2

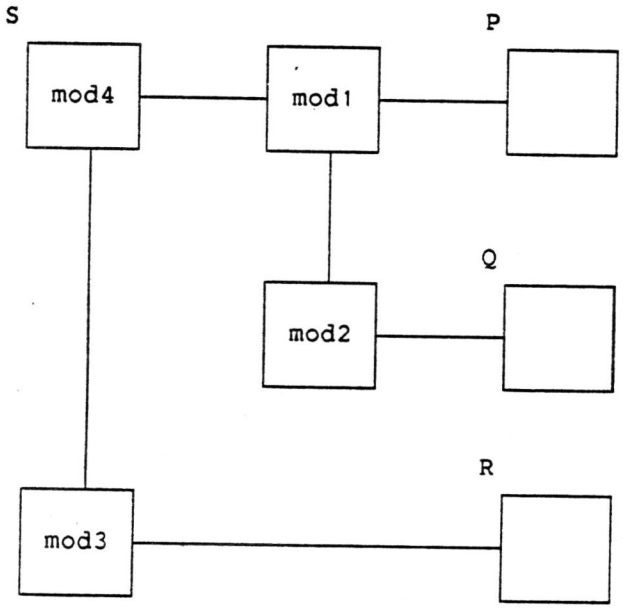

Figure 3

Such a dependence on other graphical objects can be avoided by duplicating the relevant parts of the data structure, using the copy assignment statement. It must be used judiciously, however, as the size of the data base may grow much more quickly than anticipated. With the system functions VALUE and SUPER, any part of a data structure can be accessed, no matter how complicated that structure may be. However, they do not allow alterations or replacement of the substructures they refer to. For this purpose, in addition to the synonym and copy assignments, two special assignments have been defined. With the value assignment, the defining instance of an object can be replaced by another instance, i.e. the value pointer and the transformations of an object can be reset to refer to a new and possibly entirely different instance.

As an example, consider the graphical variable P, defined initially as follows

```
P:-POLY FROM (V1) TO (V2) TO (V3)+Q<mod1>
                                    ... (3)
```

where V1, V2, and V3 are vector variables, Q is a previously defined graphical variable, and <mod1> are certain modifications.

With execution of the value assignment statement

```
P :> R <mod2>
```

the graphical value of P is altered by replacing the polygon with the instance R <mod2>; thus, P assumes a value as if the assignment

```
P :- R <mod2> + Q <mod1>
```

had taken place.

Similarly, the superpointer can be reset with the super assignment, but the transformations of that node remain unaltered. As an example, consider again the graphical variable P, defined as in statement (3). Execution of the super assignment

```
P :< R
```

results in a value for P as if the assignment

```
P :- POLY FROM (V1) TO (V2) TO (V3) + R
```

had occurred. More complex right-hand sides of the value and super assignments are possible, including references to subobjects with the aid of the system functions VALUE and SUPER.

Advances at the Programming Level

In addition to the language systems themselves (the compiler or preprocessor and the subprogram library), several programming tools have been created in the form of programs and subprograms to assist future graphics programming. At this stage, such efforts simultaneously serve to test the effectiveness of the language, providing further insight for graphics programming activities.

Translational Sweep

Modelling of three-dimensional graphical objects can be achieved in numerous ways, one of which is extrusion. Extrusion is an operation which allows creating an object from the specification of a cross-section, a closed polygon, and a direction, a vector. A translational sweep of the cross-section in the direction and for the length of the given vector defines the desired three-dimensional object. The object itself is defined by surface panels, stored in the data base which stores two end panels of the size and shape of the cross section, and a number of rectangular side panels which join the end panels. The column thus created has one side panel for each edge of the polygon. A subprogram which creates objects of this class requires only about 16 lines of high-level code (Figure 4).

Rotational Sweep

An immediate extension of translational sweep is rotational sweep, where the given end face is rotated about a given axis by a specified angle. The object is in fact created by heaping a series of identical wedges upon each other. If the object is to be built from flat faces, an additional parameter is required which specifies its resolution, i.e. the step size of the rotation. The LIG6 program wich accomplishes rotational sweep requires approximately 25 lines of code only (Figure 5).

Graphics Editor

As discussed above, a programmer will benefit by knowing the data structure that the subprogram system will create for an object. Similar to a text editor, a graphical editor which will aid a user in modelling objects would be very useful. Its tasks will be to make graphical objects visible and to carry out additions, deletions, and other operations, particularly those which affect the topology of an object. These operations will be easy to carry out when both the actual image of the object, as well as the internal structure of the object can be made visible. Such an interactive graphics editor has been implemented, written, of course, in LIG6. The editor displays the data structure as a binary tree, with a node represented by a small box containing the subobject which the node represents and with pointers as branches. A strict representation convention is used, with the value pointer always emanating from the right of a node, and the super pointer from the bottom (see, e.g., Figures 4 and 5). The user can operate on the pointers, creating or inserting new nodes, and deleting existing ones. Each operation is immediately reflected in the updated display of the object itself. At present, the experience gained with this programming tool is insufficient for reaching definitive conclusions. However, it is evident that the editor expects the user to know the internal data representation and its relationship to the external representation of an object. For further details, see [Ross82b].

Conclusion

The major motivation for investigating high-level graphics languages is the potential of increasing productivity in writing interactive and passive graphics programs. All the well-known advantages of general-purpose high-level languages, such as ease of learning the language, writing, debugging, documenting, and maintaining programs, apply.

Due to the complexity of data in graphics programming, a higher level of sophistication should be expected of a professional programmer. A well-designed high-level graphics language and certain programming tools can be of much aid.

Acknowledgements

I am indebted to many of my students for assistance, ideas, and encouragement in this work, particularly to Robert Ross. Financial assistance was granted from the Natural Sciences and Engineering Council of Canada (Grant A-5148).

```
      GRAPHICAL FUNCTION EXTRUD(XSECT,DIREC)
         GRAPHICAL XSECT
         VECTOR DIREC, OLD, NEW
         INTEGER ORDER, PRILEN
         EXTRUD :- BLANK
         ORDER = PRILEN(XSECT)
         CALL LINPNT(XSECT,1,OLD)
         DO 10 I=2, ORDER
            CALL LINPNT(XSECT,I,NEW)
            EXTRUD :- EXTRUD + POLY FROM (OLD) TO
                              (OLD+DIREC) TO (NEW+DIREC) TO (NEW)
            OLD = NEW
   10    CONTINUE
         RETURN
      END
```

Figure 4

```
      GRAPHICAL FUNCTION REVOLV(XSECT,AXIS1,AXIS2,DEGS,STEPS)
         GRAPHICAL XSECT, MDSTR1, MDSTR2, ONEARC
         VECTOR AXIS1, AXIS2, NEW1, NEW2, OLD1, OLD2
         REAL DEGS
         INTEGER STEPS, ORDER, PRILEN
         MDSTR1 :- BLANK <MAP (AXIS1),(AXIS2) TO (0.,0.,0.),(0.,0.,0.),
                         ROTZ DEGS/STEPS 'DEG', MAP (0.,0.,0.), (0.,0.,0.)
                         TO (AXIS1), (AXIS2)>
         ONEARC :- BLANK
         ORDER = PRILEN(XSECT)
         CALL LINPNT(XSECT,1,OLD1)
         CALL APLYMD(MDSTR1,OLD1,OLD2)
         DO 10 I = 2, ORDER
            CALL LINPNT(XSECT,I,NEW1)
            CALL APLYMD(MDSTR1,NEW1,NEW2)
            ONEARC :- ONEARC + POLY FROM (OLD1) TO (NEW1) TO (NEW2) TO (OLD2)
            OLD1 = NEW1
            OLD2 = NEW2
   10    CONTINUE
         REVOLV :- BLANK
         MDSTR2 :- BLANK
         DO 20 I = 2, STEPS
            REVOLV :- REVOLV + ONEARC <MODIFICATION(MDSTR2)>
            MDSTR2 :- MDSTR2 <MODIFICATION(MDSTR1)>
   20    CONTINUE
         RETURN
      END
```

Figure 5

Literature

[Bart82] Barth, W., J. Dirnberger, and W. Purgathofer, The high-level graphics programming language PASCAL/ GRAPH, Comput. and Graphics, 6 (1982), 109-119.

[Magn81] Magnenat, N. and D. Thalmann, A graphical Pascal extension based on graphical types, Software -- Practice and Experience, 11 (1981), 53-62.

[McLe78] McLean, M.J., A survey of interactive graphics software, Austral. Comput. J., 10 (1978), 11-22.

[Requ80] Requicha, A.A.G., Representation for rigid solids: Theory, methods, and systems, ACM Computing Surveys 12 (4: December 1980), 437-464.

[Ross82a] Ross, R., Language for Interactive Graphics, Version 6: LIG6 User's Manual, Tech. Rpt., Department of Electrical Engineering, The University of British Columbia, Vancouver, B.C., 1982, 58 pp.

[Ross82b] Ross, R. and G.F. Schrack, The effectiveness of high-level graphical languages in dealing with various graphical domains, in: D.S. Greenaway and E.A. Warman, Eds., EUROGRAPHICS 82, North-Holland, Amsterdam, 1982, 339-356.

[vanW81] Van Wyk, C.J., A graphics typesetting language, SIGPLAN Notices, 16 (June 1981), 99-107.