INFOTECH
STATE OF THE ART
REPORT

SERIES 8
NUMBER 5

COMPUTER GRAPHICS

INVITED PAPERS

INFOTECH
STATE OF THE ART
REPORT

SERIES 8
NUMBER 5

COMPUTER GRAPHICS

INVITED PAPERS

Published by Infotech Limited,
Maidenhead, Berkshire, England.

Printed by Vlasak and Company Limited,
Marlow, Buckinghamshire, England.

UDC 681.3
Dewey 658.505
ISBN 8553-9660-1

THE INFOTECH STATE OF THE ART REPORTS 1971-1979

1 The Fourth Generation
2 Giant Computers
3 Real Time
4 Computing Terminals
5 The New Technologies
6 Computer Networks
7 High Level Languages
8 Application Technique
9 Incompatibility
10 Interactive Computing
11 Software Engineering
12 Computing Economics
13 Minicomputers
14 Operating Systems
15 Data Base Management
16 Computing Manpower
17 Computer Design
18 Computer Systems Measurement
19 Commercial Language Systems
20 Computer Systems Reliability
21 Management Information Systems
22 Input/Output
23 Microprogramming & Systems Architecture
24 Network Systems and Software
25 Data Base Systems
26 Virtual Storage
27 Structured Programming
28 Real Time Software
29 Multiprocessor Systems
30 Program Optimisation
31 Distributed Systems
32 System Tuning
33 Minicomputer Systems
34 Software Engineering Techniques
35 Microprocessors
36 On-Line Data Bases
37 Software Reliability
38 Distributed Processing
39 Future Systems
40 Performance Modelling & Prediction
41 Structured Analysis & Design
42 System Reliability & Integrity
43 Minis Versus Mainframes
44 Data Base Technology
45 Future Networks
46 Future Programming
47 IBM
48 Microcomputer Systems
49 Software Testing
50 Managing the Distribution of DP
51 Structured Software Development
52 Convergence: Computers, Communications and Office Automation
53 Man/Computer Communication
54 Distributed Databases
55 Microcomputer Software
56 Supercomputers

CONTENTS

Invited Papers

DATA MANAGEMENT FOR INTERACTIVE GRAPHICS

M P Atkinson

University of Edinburgh
Edinburgh
UK

ACKNOWLEDGEMENTS

The work on the data curator is currently being supported by a grant from the British Science Research Council, grant number GRA 86541.

M P ATKINSON is lecturer in the Department of Computer Science at the University of Edinburgh. He formerly held positions of lecturer and researcher at the Universities of Lancaster, Cambridge and East Anglia. He has consulted on databases and software design with a number of organisations, including ICL, SPL International and British Aerospace. His current research includes databases for interactive large-scale design and production planning. Dr Atkinson is secretary of the British Computer Society's Computer-Aided Design Group.

DATA MANAGEMENT FOR INTERACTIVE GRAPHICS

Graphics technology has always been bedevilled by particularly acute problems of data structuring and storage. Solutions drawn from conventional database technology have not generally met the unique requirements of graphics systems, but new solutions have been evolved. The paper describes the latest developments in data design for increased programmer productivity and efficient operation, with underlying advantages to users.

INTRODUCTION - THE DATA

Before we can understand how to manage any resource we must have a clear perception of what it is, and how it is used. The data used for interactive graphics has a number of features which combine to distinguish it from other data. These features are as follows:

1 Its intimate relationship with program structure.

2 The complexity, variety and unpredictability of its access paths.

3 The high performance of algorithms and accesses which are demanded by ergonomic considerations.

4 The relationship it may have to large bodies of application-related data.

These features are listed in order of importance in the way they influence the data management task, and require us to develop different techniques from those used in, say, conventional database management. Other features which may be of importance in a particular application are:

- The volume of data involved
- The longevity or transience of the data.

These various features will now be discussed to illustrate the nature of the data being processed.

CONVENTIONAL DATABASE STRUCTURE

In many applications, where significant volumes of data are handled, the data structure may be designed with little cognisance of the programs and algorithms which are to be applied to it. In these circumstances the data relates closely to the perceived structure of the relevant objects in the real world. Once this perception has been realised and identified, then the logical structure (which in this case is the predominant structure) can be defined to reflect that perception (001). Even in that context the task is not always straightforward (002) but it is a possible and practicable approach in, say, business and administration systems for the following reasons:

1 The loss in efficiency and speed of response for individual programs engendered by

this approach is more than outweighed by the economic gain of the improved planning of system construction and the adaptability to change (data independence) so achieved.

2 The recognition of the essential entities and relationships has evolved over many years, and they are largely tangible and easily visualised (but see (001)).

The programs then cluster around this data, and are defined in terms of it, taking a subsidiary role and implementing operations which reflect events in the modelled world, or generating reports and extracts of data to correspond to requests for data. Recent interest (003,004,005) in the relationship between software engineering and database design shows the important role the specification of the data structure has in determining the form and development of the total software system. This is not a trivial role, as the control and organisation of the software is usually a major component of the cost of such systems and is certainly critical to the system's success.

GRAPHICS DATA AND PROGRAM STRUCTURE

The data and program structure in a graphics system may be seen as a marked contrast to the conventional system just described. Predominant here are the programs used to implement operations on the data. This is exemplified by the plethora of papers on hidden-line-removal algorithms, or by the researches of workers such as Braid and Forrest. It proves sufficiently difficult to devise algorithms - which are adequate in performance and accuracy - to implement operations on two and three dimensional objects, even though the operations are well known and easily specified. In devising these algorithms, significant gains may be made by using appropriate data and data structures. Thus, often the data structure used may be subservient to the algorithms and programs implementing the graphics system.

Further, the implementation of this software is a major component of the cost of the graphics system, and getting it to perform correctly and satisfactorily is a substantial challenge to the implementors. If their task is made more difficult by an imposed inappropriate data structure*, the difficulty may become insuperable. Certainly the time to complete the task will be extended, and the performance of the algorithms impaired. If the complexity of the programs is increased due to a cumbersome interface to a data handling package, then in the worst case a threshold in complexity may be reached making it impossible ever to debug the program completely. Even if these very worst effects are avoided, the time, cost, performance and reliability of the software will be adversely affected. As this is an important factor in data management we will return to this issue later in the paper.

An example of a data structure, which is typical of graphics applications, is taken from a paper by Braid and Hillyard (006) and reproduced in Figure 1.

In this example the data structure definition language is that of ALGOL 68 (007). Similar data structures can be defined in PASCAL (008) and in many other languages. Indeed they can be written in the new FORTRAN 7 standard (009) and in the proposed language ADA (010).

---

* In a survey of a number of programs recently conducted by IBM (011), in a screen-based administrative system, 60% of the code dealt with interfacing to the display, 25% with interfacing to the database management system and 15% with implementing the application's algorithm.

```
mode  vector  =  struct  (real x, y, z),
mode  trans   =  struct  (ref matrix tm,      # pointer to 4 × 4 matrix #
                          ..... ),            # other fields omitted #
mode  curve   =  struct  (int ck,             # curve type #
                          ref [ ] real cf),   # equation coefficients #
mode  surf    =  struct  (int sk,             # surface type #
                          ref [ ] real sf)    # equation coefficients #
```

Figure 1: Simplified part of a data structure used for graphics

Looking at that fragment of data structure definition we may observe the following:

1 The object system modelled is not part of the real world. Vectors, transformation matrices, curves and surfaces are all abstractions which have been chosen to model real objects. The abstraction step for which the tools of data modelling are being developed has already been taken.

2 The use of ref here is an important step in partitioning the definition and/or in specifying semantic information. The ref matrix allows the details of the representation of a matrix to be specified elsewhere, avoiding distracting detail here and possibly postponing a decision to a more appropriate moment, or localising it in a more appropriate site. The use of ref [ ] real (a reference to an array of reals) may also be allowing the localisation of instances of the parameters of a curve, so that the structure may indicate that two or more curves (or surfaces), have the same equation (see Figure 2).

3 The programmer is using references and arrays. Constructs, which typically, are not well supported by traditional database systems.

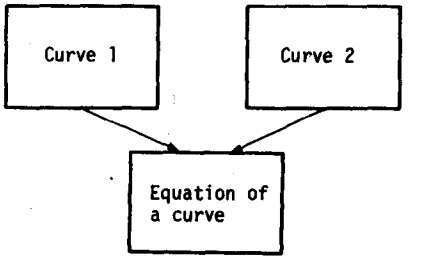

Figure 2: Two curves may share same equation

As a further example, consider another fragment of the declarations used by Braid and Hillyard, shown in Figure 3.

Examining Figure 3 the reader may note the following:

1 The predominance of reference components which point to or name other data objects.

2 The use of previously defined data types, such as vector, instances of which may exist, both in their own right, and as part of a larger data object.

3 The variety and complexity of data traversal paths which are provided, e g to scan the edges of a face, the edges around a vertex, the faces around a vertex, the vertices around a face, the instances in an object, and the instances in which an object is instanced.

We may be sure that these features are important. For example, consider the preponderance of reference variables. They have clearly proved a useful modelling tool otherwise the programmer would not have described his data that way. Some of the semantics

```
mode object = struct (int ni,                          #no of times an object is
                                                        instanced#
                      ref inst bi,                     #first of chain of instances
                                                        in the object#
                      ref face bf,                     #a face of object#
                      ref edge be,                     #an edge#
                      ref vertex bv),                  #a vertex#
mode inst   = struct (ref object ib,                   #object instanced#
                            iw,                        #object containing instance#
                      ref trans it,                    #transformation applied to
                                                        instance#
                      ref inst ii),                    #next in instance chain of iw#
mode edge   = struct (ref vertex epv,env,              #vertices at end of edge#
                      ref face epf,enf,                #faces on either side of edge#
                      ref edge epcw,epcc,encw,encc,    #winged edges#
                      ref curve ec),                   #equation of curve of edge#
mode face   = struct (ref edge fe,                     #an edge of the face#
                      ref surf fs),                    #equation of surface of face#
mode vertex = struct (ref edge ve,                     #an edge of the vertex#
                      vector vs);                      #vertex coordinates#
```

Figure 3. A further fragment of the data description shown in Figure 1

of these references are captured in the definition, others are not. For example, the type of the referend is identified, and its uniqueness and existence implied, but the paths which form loops, or those which form inverses of one another are not. At the same time, the programmer uses this construct freely, as he believes the task of finding the referend (dereferencing), given the reference, to be of low cost. They are central to the performance of the algorithms, and implement the multiplicity of paths they use. One might prefer it if programs did not contain such structures. The proliferation of references permits the construction of complex structures, prone to error, and liable to subtle faults which may not easily be revealed or repaired. Some can be avoided by other constructs (012) and it may be possible, using data abstractions, to constrain the pointer structures to have appropriate safe structures, without loss of efficiency (013) However, at present, these are avenues of research, and whatever its outcome, the pointer or reference is at present, and will probably remain, an important programming tool. All we can expect is that its use will become more disciplined leading to more reliable software.

In summary, we have looked at the data to be managed, and have observed the following:

1 The predominance and significance of reference variables.

2 The use of a wide range of data types, which are extended and nested.

3 The use of a data definition language to define the data, (but without the strong binding of the permitted operations to those types which ought to be present)

4 The provision of complex traversal paths through the data, upon which algorithms depend for their efficiency.

5 The task of defining this data is not one of 'modelling reality'.

THE OPERATIONS ON GRAPHICS DATA

Having considered what is to be managed, it is necessary to consider what is happening to that data, to make it need managing.

The total set of operations on the data can be divided into four groups as follows:

1 Creation.
2 Manipulation.
3 Storage.
4 Termination.

These operations interact with the operations carried out by a user. The user, typically an engineer, arrives at the workstation, and indicates to the system that he wishes to start or resume a task, and indicates his area of interest. Typically, the user is working in the context of his own and others' previous work, so the system assembles some subset of the data generated by that previous work. The user then manipulates that data creating more, and finally concludes the task and saves or extracts the relevant data he has generated. Although the computer may operate on the task almost indefinitely, the user does not have this stamina. He may tire, lose concentration, need to consult references, think, or fulfil other commitments. Thus it is necessary for him to be able to leave off a task and return to it. When using graphics, some workstations may be more suited to one part of the task, and some to another, they may be on different machines supported by different software. Thus the task may be started on one machine and then resumed on another. As with any other software system, some of the programs will always be undergoing development, change or repair.

Thus the task of maintaining the data needs to be stable over a number of changes. The *creation* of the data may be an absolute creation, as when a user draws a new arc on a diagram, or it may be that data is brought to the machine from some external system, a component supplier's database or another data source within the company. Once created, the data would normally be *manipulated* by the operations available within the programming language of the applications and graphics software. *Storage* is often achieved in present day systems by translating the data to and from textual form or other alien representation. This is highly inefficient in a number of ways:

1 The programs which perform the translation have to be written, maintained and accommodated in memory.

2 The translation process has to be applied to all of the data before useful processing can commence. This is often wasteful as only a small part of the data may be touched in one session, before the translation is reversed, and hence there is a redundant wastage of both instructions executed, and data transferred.

3 Channel or communication resources are wasted.

This process of data translation for storage is, however, quite prevalent as it does also have a number of advantages as follows:

1 The stored data may be read, or created by hand which is useful during system development.

2 The necessary machine and software independence is assured.

3 It is intrinsically simple, and lends itself to being a clear interface between various programs.

The final point is perhaps the most important, as it has a significant effect on the speed of implementation and the reliability of the final product. (Techniques for precisely defining the allowed syntax and semantics for a text are well known and it is easy to build programs for checking that the specification is met.)

Finally, the *termination* of the data is associated with some user's finishing actions. Temporary and support data may be simply deleted and lost. The important data will, however, be passed on to other processes or an archival store. This usually involves a further translation and transmission to a central engineering database.

These various translations to and from a storage format to the 'final' data form, and from some 'source' data form, may be viewed as overheads of the system. On the other hand, those to and from the external format may be looked upon as an investment. It is usual for the data to be held as a central model, perhaps of an object being designed (such as an aircraft, hospital or ship) or of an object being monitored and controlled (such as in air traffic control or a power station). This model will be organised for a variety of uses, and will contain much data that is not relevant to graphical interaction. It is, therefore, very unlikely that this data will be in an optimum, or even suitable, form for any interactive graphics operations. The translation, at the outset, of data which is only relevant to a graphics task into a form optimal for the graphics application, is then an investment yielding savings or performance gains during the graphics sessions. It is important that this investment is not squandered by translating data which is not needed. Appropriate techniques are therefore required to specify and obtain the relevant data, at the start of the task. It is also necessary to provide a means of incrementally obtaining supplementary data during the task, otherwise the user will 'over-order' to ensure he has everything he might need.

In summary we see that the following operations need support:

1 Creation of data structures.
2 Translation to and from more general data structures.
3 Program-oriented data manipulation.
4 Medium term storage or persistence of data.

SOME SYSTEM CONSIDERATIONS

Scale

We have viewed the extracted data for one user's task as a separate entity. If it is a separate entity we may exploit this. First, note that there is a limit to the complexity of a task one person can undertake. Ease of collecting data, and starting new tasks will encourage that person not to claim or hold significantly more data than he can effectively use. Consequently this puts an upper bound of a few megabytes on the amount of data that need be managed at one time. In fact, in a recently developed system for aircraft geometry design, the typical volume of active data was less than a megabyte, and rarely exceeded quarter of a megabyte. The most complex case envisaged, when the undercarriage mechanism was being designed, only just reached a megabyte. Consequently, much simpler techniques, such as mapping the data into the program's

address space are applicable here, although they are not applicable in conventional database work.

Concurrency

The fact that a user is modifying data through a high bandwidth interactive graphics terminal, has significant consequences. First, it is inappropriate for two people to attempt to do this to the same data simultaneously. They are liable to confuse or mislead each other if they do, and get into situations which are perilously inconsistent. It is unlikely they can effectively cooperate on a common task. Hence, provision of concurrent mechanisms at this level is inappropriate. They must be dealt with at a higher level, corresponding to the management decisions regarding allocation of tasks, and they must be enforced as the data is extracted for graphical processing from the central (engineering) database, and corresponding to the authorisation process, as it is returned afterwards. This corresponds closely to the practice that has existed in many engineering companies of controlling the access to, copying of and amendment to drawings held in a central repository, but withdrawn for various projects to different drawing offices.

Integrity and backtracking

Secondly, this use of the graphical interactive tools leads to massive and sweeping changes to data. For example a few fundamental points or dimensions are changed on a drawing, and all the other points and curves which are constructed from them are changed. This invalidates the assumptions on which many database management systems base their integrity policies. They usually assume that only small parts of the data will be changed in a sequence of transactions, and therefore store before-images (copies of the original pages or records of data) to permit recovery to the previous state. The user at an interactive graphics system certainly needs to be able to reverse his operations, as he and his machines are just as prone to error as the conventional database user. But the high rate and extensive scale of changes possible with interactive graphics may make before-images entirely uneconomic. One alternative is to permit return only to the start of the task. This costs little, as in most systems the initial data is still available from its external source, or in its storage form. With storage forms, the task can be suitably punctuated by the user to preserve critical states. With even less effort by the user, and with lower costs in machine cycles, copies of the data in its graphical form may be kept from time to time. With the limit in scale of the data, this is quite practical, and the cost is decreasing with decreasing memory and backing store costs. A further alternative is being investigated by Neil Wiseman (014), which is to provide an inverse for every high-level operation on the graphical model. A log of these operations can then be kept, and it can be run backwards to retrace steps in the design. This only helps with human failure, and not with software or equipment failure.

Recovery

As an individual designer is probably working with any one set of graphical data, a failure of his system is less critical for a company than is a loss of data and system crash on a central database, say supporting a large transaction processing load. (This may not seem true from the viewpoint of the designer!) Consequently the speed with which the data can be recovered, and with which a roll-forward to the last

quiescent state of processing can be achieved, is less critical. It is therefore usual to invest less heavily in recovery data and procedures in the graphics application. This is important, as such procedures are expensive and impede performance. Dispensing with some of them usually makes a more responsive system possible, which is probably of much greater significance to the designer as equipment fails only rarely!

Consistency checking

In conventional database applications, checking the data as it is input (*validation*) and checking the stored data for correctness (*consistency checking* and *audit trails*) are frequently important. At the interactive graphics terminal it is less easy or less appropriate to apply such checks because:

1 The consistency constraints involved are much more complex.

2 Intermediate states which must occur as the model is being changed are often inconsistent.

3 Gross errors, which are easily detected by validation, are usually conspicuous on the displayed picture.

Consider each of these reasons in turn. The extra complexity comes from the mechanical nature of much that is shown on the screen. The following are examples:

- The intersection of the projection of these two lines must be at the centre of that circle
- These two parts must move independently so that they never collide
- All these parts must be machinable
- No subnetworks of this circuit may be isolated subgraphs
- No two areas on this mask may be within 1.5λ of one another.

The examples are legion, diverse, often difficult to specify in a computational form, and are frequently expensive to compute. It is sensible, therefore, either to enshrine them in the primitive operations of some application, or to write separate programs, which are envoked, to be rule-checkers, testing the model when the designer believes he has finished. They may operate on either the graphical data or the untranslated data.

A designer must be able to experiment and, to make the fullest use of graphics, view his experiments. He may only partially perform an experiment before he realises it does not lead to his desired goal. During these experiments, and as he specifies basic actions on the data, he will often leave the data in states which do not comply with the design rules. It is not possible for the machine to recognise when a user believes he has reached a correct state, and hence it is necessary for him to explicitly request tests.

Many tests to see, for example, if two parts collide or if they can be machined, are expensive. These are easily conducted, at least in the first instance by a glance at

a view of the system. The experienced engineer will spot most errors easily by seeing views of the object. The remaining errors have to be detected by running application programs as already described, but given the powerful parallel processing or pattern recognition of the human eye or brain it is foolish not to use it.

These aspects of consistency radically change the techniques used to achieve correctness from those used in a business database application.

Summary

The following system considerations have been shown to be radically different from those of a conventional database application:

1 The scale or volume of data.
2 The concurrency requirements.
3 The integrity and recovery requirements.
4 The approach to consistency checking.

It is important to exploit these differences in order to achieve the performance required for interactive graphics.

GRAPHICAL SYSTEMS AND SOFTWARE ENGINEERING

It is a major undertaking to construct a graphical software system, with the associated application, communication and interface software, command interpreters, rule checkers etc. Such undertakings need an appropriate software engineering approach, dividing the task into appropriate components, and ensuring that these components conform to the intended design once constructed. One general approach to this last step is to use the views, data description and constraints on data access provided by a conventional database management system. The author has written about the difficulties of using this approach in this context elsewhere (015). Other approaches using abstract data types also present difficulties, and a full discussion is not appropriate here; the interested reader is referred to the following references (003,004,005).

The productivity and success of a programming team is critically affected by the programming language it uses and by the programming environment in which it works. Various aspects of this environment are worthy of improvement, for example making it easy to conduct all debugging in the high-level language context, and making it easy and efficient to bring about changes in that context (019). Another improvement is to give the programmer a consistent development environment independent of for which machine he is developing the software. A good example of such an approach is the Bell Laboratory's programmer's workbench (020). The programmer himself can also benefit from a good interactive graphics facility (021).

SUMMARY OF REQUIREMENTS

We need to manage data which fulfils the following requirements: