Automated Testing of Computer Graphics Systems

Joaquim Armando Pires Jorge
INESC
Rua Alves Redol, 9
1000 Lisboa, Portugal

Rolf Ziegler
Martin Göbel
ZGDV
Wilhelminenstraße,7
D-6100 Darmstadt, FRG

Abstract

Graphics systems have become increasingly complex over the last years. Computer graphics standards provide the means to implement portable graphics applications. However, the goals aimed by the use of graphics standards cannot be achieved, unless the standards are not lent to certification. Methods and tools for testing graphics systems have been developed, such as the GKS conformance testing package.

Current test tools rely heavily on operator judgement of pictorial output produced by implementations being tested. As graphics systems are becoming richer in functionality, the design of test cases becomes more difficult and error prone.
Automated test methods should be devised to provide for certification methods more precise than those involving human judgement. In particular, methods for automatic picture testing should be devised. Graphics systems should also include functions to assist in automatic testing, in order to prevent the need for special external devices such as cameras and image processing equipment.

This paper focuses on the issues raised by automatic testing of graphics systems. Standard components are studied with regard to their ability to being tested. Recent technologies and their application to automatic testing are discussed.

Introduction

Graphics standards describe a model of a system, interactions between its components, and externally visible results of these interactions. The implementation of these results is not specified by these standards.
Standards specify the flow of information across an interface and the data that is to be exchanged, but not the internal workings of the different functional units.
Figure 1 shows the generic interfaces within the model of a computer graphics operating environment. It highlights the concept of multiple active workstations which may have different capabilities ranging from input-only devices to intelligent interactive workstations.
Obviously the layer of device dependent software needs to be adjusted to support such radically different environments and the application software will need to be able to inquire the capabilities of the underlying system in order to maximize its efficiency in using the underlying hardware.
The interfaces of the pipeline to any individual device are usually called the Applications Programmer Interface (API) and the Virtual Device Interface (VDI). At the operator interface, pictorial output is presented as a result of graphics commands processed by the pipeline, or as a response to operator action on input devices that produce an echo.

It can be seen that one of the major benefits arising from the use of standards is device independence. The user can deal with one or more abstract graphics devices with a full range of input and output capabilities [ArBo-88].
The precise specification of functions and data provided by language bindings makes it possible to write portable applications software through the correct use of a standard.

By stating that standards allow the implementation of portable programs, we assume that all realizations of a standard will provide a unique functional behavior to the user. Therefore, there is a need for certification methods to validate an implementation of a graphics standard. This will prevent the proliferation of "compatible" and "look-alike" implementations claiming to conform to the "general philosophy" and "spirit" of a given standard.

Conformance testing aims to ensure that an implementation of a standard produces correct results and also that the syntax of commands has been implemented correctly.

The extent to which standards can be tested depends heavily on the standard specification. Up to now standards documents consist of informal specifications written in natural language. A great deal of the standardization work consists of illuminating ambiguities and clarifying dubious points. It would be desirable to have a formal specification for each standard if time, resources and tools were available.

In the absence of formal means of ascertaining the correctness of an implementation, one must use a falsification strategy: a complete implementation is subjected to a series of tests attempt to discover errors.
Thus, falsification testing can determine non-conformance to a standard but can never assure complete conformance to it. If formal specifications were available a validation strategy could be used: an implementation is derived by a sequence of transformations from the specification. If each transformation is correct, the end product will be itself correct. Formal proofs of correctness have been applied with success to small pieces of software. It is doubtful if the same techniques could be applied to graphics packages such as GKS which contain hundreds of functions.

Falsification tests separate clearly the specification, implementation and certification phases. This practical approach that aims to detect errors as a proof of failure, has been used extensively to validate compilers and operating systems.

The development of test packages for graphics standards must take into account the pictorial nature of graphics output, in contrast to other standards such as programming languages where the desired results can automatically be compared with actual test results and produce a decision. Up to the present, the subjective evaluation of pictorial output has played a major role in testing graphics systems.

Experience with established Test Services

The strategy for conformance testing of GKS was developed in the

early 1980's. A major influence was the success of compiler validation, in which a language compiler is subjected to a large suite of test programs. These test programs are written in the language being tested, and are designed to be self-checking.
However writing a GKS validation suite highlighted problems not found in compiler testing. For some functions, such as error reporting and inquiries, the 'result' can indeed be returned to the test program and verified. For example, an error situation can be detected by the test program, using GKS error and inquiry mechanisms. But for the major graphics functions such as output primitives and attributes, the 'results' are in the form of a picture which has to be evaluated by a human tester.

GKS validation tests fall into two classes:

-Application interface tests: These follow the compiler validation model, in that the test programs are self-checking, with the exception of input functions. Although input returns data through the application interface, checking is best done by returning data as output and visually comparing it to the input entered.

-Operator interface tests: These produce pictorial output, and may envolve a fair amount interaction as well. A human tester compares the pictures generated by a candidate implementation against a set of reference pictures.

Since June 1987 a GKS conformance testing service is available for the FORTRAN language binding. Testing is done by running a set of test programs and comparing its results to expected values. A thorough coverage of an implementation can be achieved for all levels of GKS.
A client may use the test software to run self-tests on his implementation. At this stage difficulties can be sorted out and different interpretations of the standard can be discussed with the Testing Laboratory. When the client orders a formal test all programs are run again under supervision of a Testing Laboratory staff member. The tester writes a report and has it accepted by the client. A certificate can be issued based on this report.
The GKS test suite has been designed to check all the prescriptions of the GKS standard and to detect deviations from the standard in a GKS implementation. This involves five test series:

-data consistency test series
-data structure test series
-error handling test series
-input/output test series
-metafile test series

The data consistency tests examine GKS description tables. The values in these tables are checked for consistency and conformity to the GKS standard.

The data structure tests check that the values in the GKS state lists are manipulated correctly, by setting, modifying and inquiring these values. These tests do not check that data is correctly interpreted when output is generated - they just verify that data held in the state lists agrees with expected values.

The error handling tests produce error situations and then check that the error response of the implementation complies to the GKS standard.

The input/output tests provide a check of the GKS System as a whole, through a comprehensive set of tests which exercise all the input and output capabilities of GKS. Pictures are produced by the tests and visually checked against a set of carefully designed reference pictures. Input is tested by a set of defined operator actions which should produce specific results on a display.

The metafile test series checks that the GKS metafile is used correctly, through operator evaluation of reference pictures stored in metafiles and then read back.

The development of the GKS test suite has shown that it is possible for relatively simple strategies, to achieve a good degree of coverage of an implementation. The development of test software was useful in highlighting inconsistencies and ambiguities in the GKS document, contributing to its improvement.
Although operator tests have the merit of exercising the system as a whole, demonstrating its ability to produce sensible pictures, experience has shown that they are a limiting factor in the testing process. Only a limited amount of tests can be run, thus rendering exhaustive testing impossible, no matter how carefully test pictures

are designed. Testing becomes very time-consuming to carry out, the dependence on human judgment is unsatisfactory and to a certain extent unpredictable - results accepted by one operator can be rejected by another.

Testing at component interfaces

As shown in previous paragraphs, a functional standard may extend its reach across several interfaces defined in the reference model. In order to do a thorough certification, one should be able to perform tests on all these interfaces. Figure 1 identifies major interfaces relevant for conformance testing.

Data is exchanged at the Application Program Interface using function calls and data structures defined by a language binding. The GKS test suite development has shown that data structure, consistency and error tests can be automatically performed at this interface if external (visual) effects are not taken into consideration.

The Virtual Device Interface stands between the device-independent and device-dependent components of a graphics system. Information exchange may take the form of function calls or a data-stream encoding, as specified by emerging standards such as CGI[ISO-88a]. Automated testing at this level, must make use of a reference implementation, which generates reference data that will be compared to the output of a candidate implementation. The development of a comparator program is still an open research topic, due to the behavior latitude allowed in current standards.

At the operator interface, data take the form of visual information, and functional requirements become less precise. All graphics standards allow considerable variations in the visual appearance of graphics primitives, to interface a broad range of graphics devices. Automated test methods for pictorial output become very difficult to implement, even if the device provides means for inspecting the contents of its display surface without operator intervention. The problems raised in developing an automated test suite for CGI, are developed in the next section of this paper.

Survey on Automatic Testing methods

One of the main lessons gained from the experience with GKS conformance testing was to test as much as possible automatically in CGI [ISO-88a]. CGI contains a more general set of primitives than GKS (e.g., circles and ellipses are included), so additional tests are needed; . Therefore the test suites should be designed in a way to perform automatic testing.

In the cases of state list checks, error report checks, error reaction checks, and tests of output of CGI raster functions (on a CGI system supporting the raster part) automatic testing is feasible. Problems arose in testing output and input automatically.
When testing output functions by automatic means, the test strategy is to compare candidate pictures with the expected results, as described in the testing of GKS.
A first approach for a comparison makes use of a pixel by pixel comparison between the rendered image and the image stored by a reference implementation of the CGI. This reference implementation would have to be properly configured to match the candidate implementation's specific characteristics. Tis approach is impaired by the huge number of allowable variations in the rendering characteristics of output primitives, coupled to areas of the rendering process about which CGI (and other graphics standards as well) says next to nothing. These include:

- Algorithms for selecting pixels to be set in rendering primitives (e.g., Bresenham, DDA, with/without aliasing etc.).
- The algorithm used to combine the new pixels with the old
- Precision of coordinates, rounding strategy and fence post problems.
- Attribute handling - definition of different styles (e.g. dashed) along with the rendering across multiple line segments (e.g., continuous around corners v. restart for each segment).
- Potential transmutation from one primitive to another during clipping (giving different rendering methods for the resulting primitives).
- Allowable levels of support for different attributes (e.g., transparency, fill styles, segment display priority, etc.).

With this number of variations a pixel by pixel comparison has almost no chance of coming up with meaningful results, however sophisticated the basis for the comparison.

Several other methods can be devised for testing picture output. One must bear in mind that testing strategies that involve the use of additional hardware, may prove to costly to implement:

1. Check whether the primitive lies inside a given area.
It is very approximate to check whether a certain primitive fits into a given area because this may lead to the answer 'the output is ok' although it is not (i.e., horizontal lines do not appear as horizontal).

2. Compare with several reference algorithms.
This approach has the disadvantage of making provision only for the "most common algorithms". Also needs to be modified to match ideosincrasies of particular implementations.

3. Apply methods of statistical analysis.
This could be a good approach to get a statement about the quality of implemented pixel setting algorithms of the candidate implementation. The idea is to compare the generated pixel data with the ideal output by means of statistical analyzing methods. Such a valuing algorithm can deliver a characteristic for a qualitative "good" pixel setting algorithm.
First steps were made in implementing an algorithm for evaluating generated (two-point) polylines. This algorithm delivers a value for the deviation of the generated pixel line from the geometric polyline [LeWe-87]. Problems arose in defining an appropriate measure value which will be the upper (or lower) value for a qualitative "good" polyline (or pixel setting algorithm).

4. Compare by means of computer vision.
Recent advances in computer vision [BoTh-87] make this an alternative means of checking the correctness of pictures. Techniques such as Hough transforms allow the identification of areas; and character recognition too was shown to be possible.
For devices which use a bitmap, it may be possible to interrogate the bitmap directly using the GET PIXEL ARRAY function; otherwise some image capture device can be used to collect a representation of the picture produced by the candidate implementation.

To sum up briefly, experience has to be gained in automatic picture comparison concerning computer vision and statistical analysis algorithms. Research will have to be done in order to determine if

these methods are suitable for comparing pictures.

Conclusions

Conformance testing provides a means of certifying the adherence of implementations to a given functional standard. As graphics systems become more complex, writing, maintaining and running test suites for these systems is getting an increasingly harder task. This paper attempts to highlight the components of a standard that are better suited to be tested automatically. Testing for visual effects seems to be harder part of the testing task, given the loose specification of visual output in current graphics standards. From the standpoint of emerging standards such as CGI, new test methods based on image processing present interesting research possibilities.

References

[ArBo-88] Arnold, D.B., Bono, P.R.: CGM and CGI, Metafile and Interface Standards for Computer Graphics, Springer-Verlag, January, 1988.

[BoTh-87] Boyle, R.D., Thomas, R.C.: Computer Vision, A First Course, Blackwell Scientific Publications, October, 1987.

[BoHe-87] Bono, P.R., Herman, I.:GKS Theory and Practice, Springer-Verlag, January 1987.

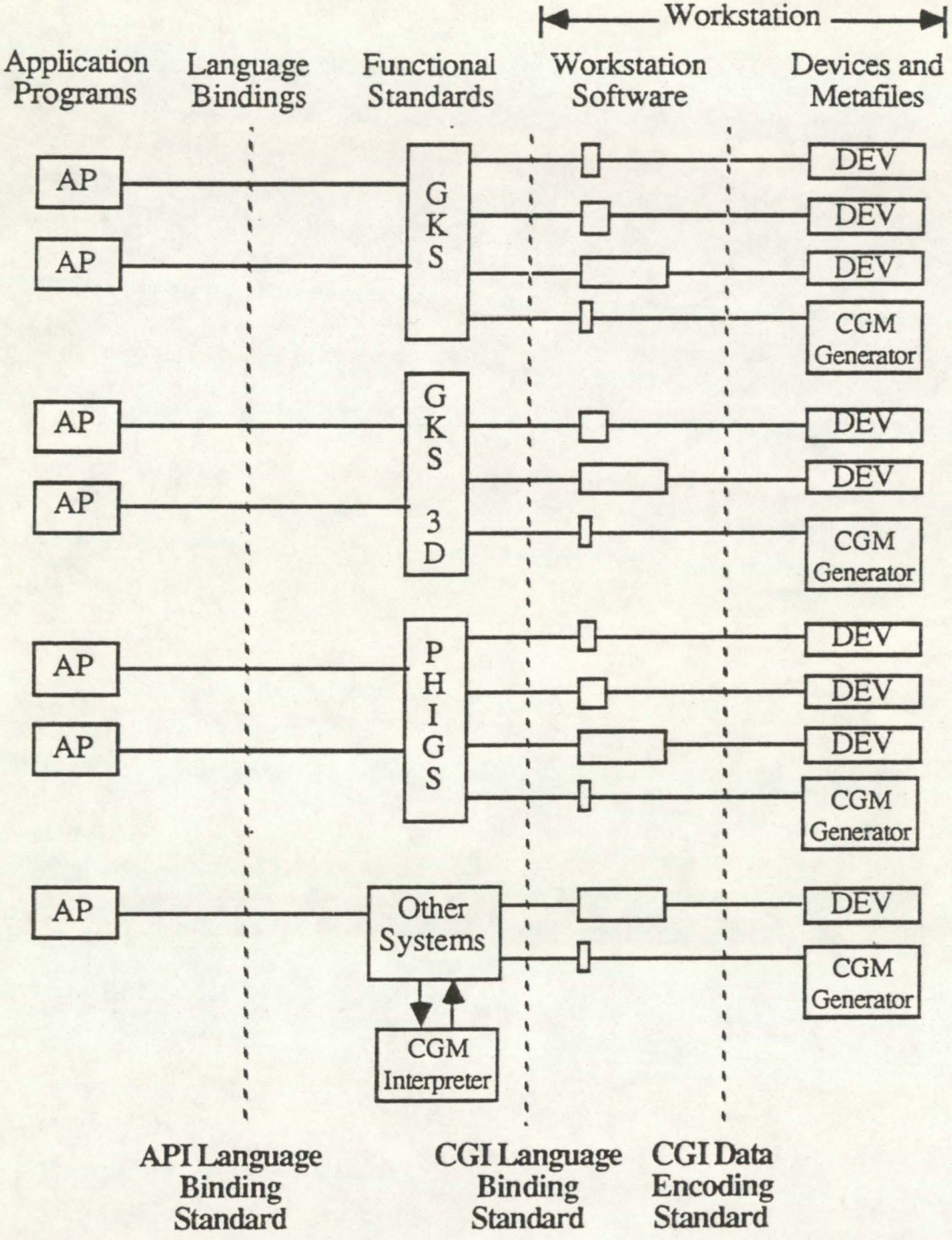
[Bro-83] Brodlie, K. W. "GKS Certification - An Overview", Computers and Graphics Vol. 8, No 1, pp 13-17, 1984

[ISO-88a] ISO/DP9636(CGI) - Interim Draft, Information Processing Systems - Computer Graphics - Interfacing Techniques for Dialogues with Graphics Devices - Functional Specification, April, 1988.

[ISO-88b] ISO/IEC JTCI/SC24 NI85, Conformance Testing of Implementations of Computer Graphics Standards, Initial Draft Document, July, 1988.

[LeWe-87] Lehn, Wegmann: Einführung in die Statistik, Teubner-Verlag, 1987.

[ZiGo-88] Ziegler, R., Göbel, M. (Eds.): Workshop Report "The Computer Graphics (Device) Interface: Applications and Test Methods for CGI", August, 1988.


Workstation
Application Programs
Language Bindings
Functional Standards
Workstation Software
Devices and Metafiles
AP
AP
GKS
DEV
DEV
DEV
CGM Generator
AP
AP
GKS 3D
DEV
DEV
CGM Generator
AP
AP
PHIGS
DEV
DEV
DEV
CGM Generator
AP
Other Systems
CGM Interpreter
DEV
CGM Generator
API Language Binding Standard
CGI Language Binding Standard
CGI Data Encoding Standard