GRAPHICS IN SPRITESLAND

by

F. Mavaddat
N. Cutcliffe
R. Ellis

Department of Computer Science
University of Waterloo
Waterloo, Ontario

Research Report CS-83-30

October, 1983

GRAPHICS IN SPRITESLAND

F. Mavaddat

N. Cutcliffe

R. Ellis

Department of Computer Science

University of Waterloo

Waterloo, Ontario N2L 3G1

Canada

ABSTRACT

A friendly programming environment for Texas Instrument's 9918A Graphic Processor is discussed which, while hiding some of its more difficult features of low level programming, enhances the concept of sprites. Enhanced sprites, which we call Graphic Objects, are more powerful than their constituent sprites in size or number of colours and can change shape to represent time varying objects. This environment also facilitates the writing of Graphic Processes. Graphic Processes are self contained, relatively independent programs which look after the movement of closely related Graphic Objects without much attention to simultaneous control of other objects in the scene. Graphic Processes simplify the handling of complex and structured scenes. This paper ends with a discussion of implementation issues.

1. INTRODUCTION

Computer graphics is enjoying a deservedly high profile and widespread recognition. There is almost no doubt that graphic and voice will be the predominant forms of communication with computers in the future.

CRT based graphic equipment is still quite expensive. Any high resolution colour enriched

graphic output station is worth many tens of thousands of dollars. There are some graphic applications that even at such high prices are still very cost effective. The use of such graphic systems in the cinematographic industry is now a proven success [3]. Other known cost effective applications are flight simulators and oil tanker simulators. Many other applications are not cost effective as yet but are pursued in the hope of future reductions in the cost of graphics hardware. Certain graphic applications require much lower hardware costs for them to be cost effective. No revolutionary reduction in the cost of high resolution graphics hardware can reasonably be expected in the foreseeable future. Therefore new approaches are necessary to address such applications in a cost effective way. Examples of applications are electronic video games, computer aided instruction, and colour computer terminals. All these applications require hardware costs below a few hundred dollars if not sometimes much less.

There are certain characteristics of these applications which makes their cost effective implementation possible with even the present state of hardware technology. The first characteristic of significance is the acceptability of low resolution graphics. Instead of 1280 x 1024 pixels which seems to be the de facto standard of high resolution graphics [2], these applications are sufficiently satisfied with considerably lower resolutions. The other characteristic which often adds to the reduction of cost is the lack of any need for pixel intensity definition. Most of the applications discussed require only one bit of information per pixel. This bit is used to turn the pixel on or off in black and white applications, or distinguish between two colours in colour applications. The combined effect of lower picture resolution and binary pixel information is to reduce the need for high bandwidth access to frame buffer memory.

Another characteristic which contributes to low cost is the relatively few number of colours needed to satisfy many applications. While hundreds of colours are typically provided by more advanced graphic systems, low cost applications require not more than 8 or 16 colours. This obviously contributes to further cost reductions.

Low resolution, less pixel information and a reduced number of colours still satisfies many applications because they are not trying to duplicate reality and the loss of realism does not seri-

ously damage the applications' central themes [4,7]. As an example, in an alien's invasion game, representing the aliens by simple box shape figures and with coarse resolution is definitely not realistic, but its effect on the main theme of the game is not serious. In other words, the main theme is the human reaction to some threatening objects, aliens or otherwise. It is probably better to call them aliens because of the increased flexibility that this allows in representing them. The number of colours is another limitation which is not going to seriously affect the applications discussed. If representing reality is the main theme then we are in need of considerable colour control. The more familiar the object to be presented is, the more control that is needed in its generation. For example, the variety of colours needed in generating a human face makes it suitable only to the most advanced of graphic systems, while an alien's colour is open to imagination and probably the stranger the more real!

Limited requirements of this class of graphic applications prompted the manufacturers into the construction of single board graphic processors. Most of these were used in arcade games. Success of the arcade games and their subsequent proliferation into home entertainment enlarged the market to the point of making the development of custom ICs profitable. Initial ICs were limited in their function and capabilities. Recently a few graphic ICs of reasonable power were introduced into the market. This report discusses one such IC, TI's 9918A Video Display Processor. A high level graphic package to be used for developing application programs is presented for that IC. These are all part of an ongoing project in our Microsystems laboratory.

2. FEATURES OF TI's 9918A GRAPHIC PROCESSOR

The Texas Instruments 9918A is a single chip video display processor (VDP) which supports up to 16K of raster display picture memory (frame buffer) in the form of dynamic RAM. This memory is not part of the host processor's address space. It is accessed by the host using the VDP as a middle man. The VDP also performs the task of generating a composite video signal output of the raster image described by the frame buffer.

The VDP is able to operate in four different modes to generate its primary raster display. All of these modes produce cell graphics. Cell graphics operates by partitioning the frame buffer into a collection of sub-buffers called cells [5]. Each cell describes the raster image of a fragment of the overall image. The main display is composed of an array of pointers to cells. The usual arrangement for cell graphics is such that there are insufficient cell definitions (usually 256) for a unique cell to appear on every cell position of the screen. Because of this, it is not possible to uniquely control every pixel on the screen. The result is that cell graphics is a very awkward system for image production. However, the 9918A has a cell graphics mode available which overcomes this problem and allows the frame buffer to behave like a simple bit map. This is done by describing the screen with three separate cell definition tables of 256 cells each. The resulting raster buffer is described by 32x24 cell buffers. Each cell is 8x8 pixels giving an overall image of 256 pixels on the horizontal and 192 pixels on the vertical.

For each raster description cell there is a corresponding colour description cell of 8 bytes. Each byte in the colour cell corresponds to a row of pixels described by the raster cell. The VDP has available a 16 colour set which is encoded in the colour cell as a 4 bit binary value. The first four bits of a byte in a colour definition cell defines the colour of the ones and the last four bits is the colour of the zeros. The result is that even though 16 colours are available, the colours can only resolve down to a row of 8 pixels which severely limits the generality of colour use.

By far the most interesting feature of the VDP is that in addition to the primary raster frame buffer whose features were described above, there is a facility for multiple plane frame buffers [5]. The Texas Instruments literature refers to these small buffers as "sprites" [8]. A sprite is similar to a raster cell except for its ability to overlay and also its ability to be positioned at any pixel position. There are 32 sprites available. The sprites are available in four formats described by a size and magnification. Sprites may be 8x8 or 16x16 pixels which may be magnified to 16x16 and 32x32 pixels, respectively. When magnified, each pixel of the original raster becomes 2x2 pixels on the screen. All sprites must share the same size and magnification at any one time. There is only one of the 16 possible colours assigned to a sprite. Each of the pixels

associated with a one take on this colour. All of the zeros in the sprite buffer take on the special value of transparent.

The most attractive properties of sprites are their abilities to overlay and to be placed at any pixel position on the screen. These features eliminate from the host processor the most computationally expensive portion of moving images on a raster display. The cost of moving a 16x16 pixel image becomes equivalent to the cost of placing a pixel. These powerful features make real-time animation a viable consideration for even the lowest powered computer systems.

3. A HIGH LEVEL USER INTERFACE

Like any other piece of hardware, low level programming of the VDP is tedious and error prone. It is subject to difficulties, similar to machine code programming and therefore all the known (and of course some unknown) errors. In a similar way, most of these problems are eliminated through the use of automatic programming techniques and high level user interfaces, such as symbolic naming, and automatic number conversion. Achievements at this level are similar to those of assembly language capabilities. Another motivation for a high level interface is that of expanding the machine capabilities beyond what is apparently available through the hardware. Similar capability expansions are often known as virtual capabilities, virtual resources, or virtual machines. A well known example of this is virtual storage.

Yet another outcome of a higher level interface is the abstraction. An abstraction hides the irrelevant (or less relevant) details while emphasising some or all of the desirable features at the same time. It may even represent an unfamiliar picture of the original hardware, one more suited towards the intended applications. In our attempts at creating a higher level interface to the VDP, a great amount of emphasis is placed on abstraction, extension, and manipulation of sprites. This, in turn, makes our interface more suitable for the development of games or other animated applications.

The following pages describe the semantics of the tools developed for easy interface to the VDP. Our plan has been to develop functional tools which are called within a host programming environment. We believe that this is the best method for the initial developments and gaining of experience. Except for a very primitive set of single key graphic commands to be used for a quick test of ideas for demonstration [1], no attempt has been made to define any syntax for the interface.

3.1. GRAPHIC OBJECT

Graphic objects are a logical extension of the sprite concept. Though the sprites are limited to definitions of 8x8, or 16x16 there is no similar restriction on the size of a Graphic Object (GO). Similarly, colour restrictions imposed on the definition of Sprites are relaxed in case of the GOs. This gives the user an opportunity for handling larger sprites with more colour selections. Obviously, the range of available colours are still limited to those available from the VDP hardware.

Furthermore, the shape or colour of portions of GOs can change in time. This is useful for creating the sensation of moving or changing GOs.

3.1.1. DEFINITION OF GRAPHIC OBJECTS

One obvious way of defining a very general GO is to define it through its bit map. Some short hand notations like the use of octal or hexadecimal symbols can also be employed to reduce the burden of the job. Another method of more efficiency for most objects is to define them in terms of polygons. This can be done by first defining the boundaries and then assigning a colour to the enclosed area. As we said before, the syntax of such definitions has not been our prime interest so far. A few techniques for sophisticated paint systems are already widely known [6]. As GOs are often simple objects, we have adopted a simple notation for the definition of boundaries and specification of their colour. The same scheme is also used for the painting of the main frame buffer images. GOs defined at this level are called the primitive GOs. Two or more GOs

can be combined to form other GOs for two possible reasons. The first reason is to create a larger and more sophisticated GO. This is done through the relative positioning of the two origins of the constituent objects and naming the resultant object. The second reason is so that two or more GOs can be combined to form a cycling object. Only one of several GOs forming a cycling GO is visible at any one time.

3.1.2. CREATING AND REMOVING OF GRAPHIC OBJECTS

A GO does not exist until it has been created. The "create" command creates an instance of a GO and assigns it to one of many virtual planes. A created Graphic Object can move only within the confinement of its associated plane.

The "create" command, as part of the act of creation, has to specify the "definition" of which an instantiation is to be made. The position of the virtual plane and where in that plane it should initially be placed is also under the control of the "create" command. Any movement of the GO is subject to the control of other commands to be discussed later. Obviously it is always possible to have several instantiations of a GO definition active at any time in the system.

There is no limit to the number of virtual planes in the system and every such plane can be associated with only one GO instantiation at any time. There are no unallocated planes among those already associated and every plane is recognized by a pointer returned at the time of creation. When creating a new GO the user must specify the position of its associated plane with respect to those already in the system. This can be done by placing it in "front" or in "back" of all virtual planes or "before" or "after" a specific virtual plane.

The user may also "remove" a created GO or "purge" its definition altogether to make room for new definitions. When a GO is "removed" its corresponding virtual plane is also eliminated and its position claimed by those behind it.

Not all created GOs are visible at all times. Virtual planes are larger than the video window open unto them and only those within the bounds of the window are visible. A GO leaving

the window will re-enter the window later after traversing a virtual cylinder around which the virtual plane wraps. This is the main reason for expecting more virtual planes than there are physical planes.

3.1.3. MOVING OF GRAPHIC OBJECTS

Any movement of a GO is along some direction in its plane. Because such directions are unique to the GO, a simple "move" instruction, applied to any GO would move it along that direction by the amount specified. Successive execution of suitable "move" and "direction setting" commands moves a GO along a path of any desired complexity. To define a direction a point different from the current position of the GO is selected. The direction from the current position of the GO towards the latest target defined is the current direction of future moves. While defining the target points a distinction should also be made between those which simply define a direction and disappear and those which do act as the target and impede any further movement of the GO beyond the target point. Some simple horizontal or vertical directions of move can be simply defined as "up", "down", "left", or "right".

3.1.4. CONTROLLING OF CYCLING OBJECTS

A cycling object is a composite object which is defined by several versions only one of which is visible at any time. The selection of the visible version is done through the activation of the "switchto" command. The selected version will remain "on" until another version is selected through the execution of an appropriate "switchto" command or the GO is "removed" from the system.

3.2. SYSTEM TIMING

The rate of change in a scenario can be controlled by the appropriate insertion of the "wait" statements. A "wait" statement specifies the amount of delay to be inserted between the

completion of the execution of the statement immediately preceding the "wait" and the start of the execution of the statement immediately following it. All delays are specified in real time units.

3.3. GRAPHIC PROCESSES

Many applications require simultaneous handling of several Graphic Objects. Even in some simple games many things are moving along distinct paths doing different things. Each object is often independent and to a great extent autonomous. Putting the management of this complexity under the control of a single sequential program is an unnecessary burden on the programmer. This is a situation similar to that of writing operating systems prior to the development of more advanced techniques for managing the complexity.

A scene showing a car moving in a system of streets and intersections is in fact under the control of its driver who is following a reasonably independent algorithm taking him from point A to point B. Similarly, in a game of competition between the forces of two sides, individual men are following their own tactics of the game with the overall goal of achieving victory. All of these are comparable to the situation in an operating system where individual system components are performing their own functions individually (e.g. a tape unit is copying a buffer onto a tape while the disk drive is filling another buffer from disk and buffer to buffer transfers are done asynchronously and autonomously under the control of the CPU), while as a whole they are helping to push more jobs through the system. Based on these observations it becomes clear that a similar philosophy of breaking graphic activities into independent, though interacting processes, may be the answer to the management of this complexity.

A Graphic Process (GP) owns one or more Graphic Objects and controls their movement. A GP is controlling the movement of a set of closely related objects and as such represents their close interaction in an effective way and without any need for consideration of others. Graphic Processes are written in independent segments and as such should be written either within a host system capable of multi-tasking or in their own artificial environment.

A Graphic Process can sense its environment through the use of the "find" command. The GP may ask for the position of a specific GO and receive its coordinate back through the "find" command. It may also ask for the position of the closest GO or the closest GO of a specific type.

4. IMPLEMENTATION

For implementing the environment three important decisions have to be made. These decisions are, first, how to represent the graphic objects, second, how to handle virtual planes with respect to the physical planes, and finally how to implement the graphic processes.

The most economical representation of a GO in terms of the number physical sprites is a desirable goal. Even with the most economical allocation the savings are not that great. Following our initial goal of finding the most suitable tools we have not considered the optimization issue yet and have settled for a simple mapping of Graphic Objects onto the sprites.

An m-sprite Graphic Object would require m physical planes when it is within the visible window. On the other hand the same object requires only one virtual plane when it is not within the view. The m physical planes have to be consecutive. This requires the re-assignment of physical planes each time an object is about to appear within the window and release of those planes after it leaves the view. This is a situation similar to the allocation of memory to incoming jobs in a multi-programmed system and reclaiming the area after the job leaves the system or is temporarily returned to backup store. The only difference is in the need for preservation of the order in the case of GOs in the physical planes as compared to their order in the virtual plane. Such considerations are often not necessary in the case of memory allocation.

The handling of processes is probably the most difficult of the implementation decisions that has to be made. Our current implementation techniques are similar to those of simulation languages. The list of future events is maintained by the process scheduler. Each list item, in addition to the time of the event, points to the program instruction (within the associated process) from which the program execution should proceed. Our implementation allows a program

execution to proceed until a wait instruction is reached which in turn schedules a new future event and relinquishes control to another process.

Unlike standard simulation techniques, when there is no current activity to be performed, time is not automatically updated to that of the next event. Real-time waits are enforced by a hardware timer and the next event on the list is started only after its real-time wait has passed. Under this scheduling protocol the smoothness of the changes is under the programmer's control. Should some changes be unacceptable it is the programmer's duty to refine his move steps. This in turn will further load the system and may slow the operations beyond that planned by the programmer. Like many other design tasks a good program is a compromise between quality of the product and realities of hardware available to the designer.

5. CONCLUSION

We have reported on some of the findings of our "Graphics in Spritesland" ongoing activities. A first version of the system was completed last January. In this system programs are written in the assembly language of the Intel 8085 microprocessor and debugged and tested on our Tektronix 8550 development system. Debugged programs are transferred to EPROM and placed in our dedicated system. The dedicated hardware is located on two circuit boards which, with a power supply, are placed in an enclosure. One board contains the VDP and its associated memory. The other board is a single board 8085 based microcomputer developed locally.

Our experience with the first version is highly favourable. A number of programs demonstrating the graphic capabilities of the system are written. A simplified version of Space Invaders was our first attack on a game project. A powerful version of Pac-Man was also written in less then one week and has been used extensively since.

Currently we are developing a new and enhanced version of the system with two cascaded VDP's. This should double the number of available physical planes.

REFERENCES

[1] Cutcliff, N., "A Functional Interface to Control Movement of Computer Generated Pictures, SPRITES, for Video Display Graphic Applications", Master's Essay, University of Waterloo, Waterloo, Ont., Can., 1983.

[2] Foley, J.D. and A. Van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, 1982.

[3] Kochanek, D.H.U., R. Bartels, K.S. Booth, "A Computer System for Smooth Keyframe Animation", technical report CS-82-42, University of Waterloo, Waterloo, Ont., Can.

[4] Malone, T.W., "What makes things fun to learn? a case study of intrinsically motivating computer games", technical report CIS-7, Xerox PARC.

[5] Newman, W.M. and R.F. Sproull, *Principles of Computer Graphics,* McGraw-Hill, 1973.

[6] Plebon, D.A. and K.S. Booth, "Interactive Picture Creation Systems", technical report CS-82-46, Unversity of Waterloo, Waterloo, Ont., Can.

[7] Shoup, R.G., "Colour Table Animation", *Computer Graphics* **13:2** 1979.

[8] *TMS 9918A Video Display Processor,* Microcomputer Series Manual, Texas Instuments Incorporated, 1981.