A Display Controller for an Object-level Frame Store System [*]

J.A.K.S. Jayasinghe, A.A.M. Kuijk, and L. Spaanenburg

In [3] and [1] a new architecture for a Computer Image Generating (CIG) system designed to have optimal interaction support for realistic 3D graphics has been presented. There it was stated that —from an interaction point of view— there is no need to have access to an image representation as low as the pixel level. This, and the fact that the performance and resolution to a major extend has been limited by the pixel update speed enforced by memory technologies, led us to the conclusion that it should be investigated whether a CRT display could be refreshed from an object-level representation of the frame instead of the conventional pixel-level frame store.

In this paper we present as a result of this study an architecture of a (multi-processor) Display Controller that is capable to directly refresh a raster display from such an object-level frame representation.

CR Categories and Subject Descriptors:
B.7.1 *[Integrated Circuits]* : Types and Design Styles — *VLSI*
C.1.m. *[Processor Architectures]* : Miscellaneous — *Hybrid systems*
C.3 *[Special-Purpose and Application-based Systems]* : — *Real-time systems*
I.3.1 *[Computer Graphics]* : Hardware Architecture — *Raster display devices*
I.3.3 *[Computer Graphics]* : Picture/Image Generation — *Display algorithms*

Key Words & Phrases: Display Controller, Computer Image Generation, Raster Graphics, Object Representation, Massive Parallelism, RISC, VLSI.

1. Introduction

In present day workstations, high quality visualization and interaction facilities are becoming essential features. Recognizing this, system designers paid special attention to the image generation pipeline in order to improve both image quality and interaction behaviour. By improving the image generation pipeline, the frame buffer access bottleneck became more and more apparent. To overcome this problem, all sorts of partitioning

* This study is partly funded by the Dutch Technology Foundation (STW).

strategies have been developed, without asking the basic question: "do we need a pixel-level frame buffer?"

The basic justification of a frame buffer in graphics systems is the need to uncouple the real-time refresh process from the computation intensive image generation process. In order to separate these two processes, storage of the image is needed in a representation suitable for the refresh process. Due to ever increasing demands on image quality and image complexity, even the vast evolution of hardware we could witness the last decade did not result in an image generating system that could meet the timing requirements imposed by the refresh process. This justifies the expectation that uncoupling of the image generating and the refresh process will always be needed. Realising this, the basic question posed above can be changed into: "do we need a frame representation level as low as the pixel level?"

To answer this question from an interaction point of view an inventory of the types of graphics based interaction [3] shows that these interactions basically act on three representation levels (see Table 1). These levels are: *Low:* visible parts of objects (LDF), *Medium:* objects as a whole (MDF) and *High:* the image as a whole (HDF).

Note that there are no interactions that address individual pixels at all, so the answer to the last question from interaction point of view is *no.*

LDF	MDF	HDF
Highlight	Priority	Viewing Control
Blink	Visibility	Grouping
Depth Cue	Transparency	
Pick	Shading, Reflection	
	Scale, Translate, Rotate	
	Clip	
	Change, Replace	

Table 1: Examples of some graphics based interaction operations and the representation levels on which they operate. LDF is the level of visible parts of objects, MDF is the object level and HDF is the image level.

Based on this inventory, we designed a workstation architecture where all three levels mentioned are accessible for interaction purposes [1, 3]. Since these three levels are present in our architecture, it is only a small step to come up with the final question: "is it possible to refresh directly from the lowest representation level needed for interaction?"

In order to answer this question, firstly some details on this lowest representation level (LDF[†]).

[†] LDF stands for Low level Display File or alternatively Linear Display File following Carlbom [2]

In our architecture, the LDF is a bucket-sorted structure of primitives called *patterns* It is the result of a hidden surface removal algorithm that operates in object space [5]. It is essential to note that, since only the visible parts of objects are in this file, *the patterns in this LDF are non-overlapping*. The geometrical properties of these patterns are described by *domains* (D(x, y), in the form of a sorted list of slices designed for efficient HSR and scan conversion) whereas the colour properties are described by *colour-functions* (C(x, y)).

What properties should a Display Controller (DC) have that can indeed refresh directly from this LDF, the lowest but still structured object-level representation we have accessible for interaction?

It should be noted that since the number of patterns in the LDF can be very high for a complex scene (in the order of 100 K), the band-width of the LDF/DC interface will be a prime factor that could limit the performance of the DC. Because —as mentioned in the above— patterns in the LDF are non overlapping, only a few patterns, the so called Active Patterns, contribute to a given pixel-row. Active Patterns in general will contribute to several pixel-rows. Due to high refresh speeds, on-chip storage of the Active Patterns will be necessary to reduce the band-width requirements of the LDF/DC interface. Since each "slice" of a domain has enough information to paint the pixels up to the next slice, the on-chip storage could be kept to a minimum (i.e. instead of storing the complete active pattern in the on-chip storage, only one slice of an Active Pattern needs to be stored).

As real-time scan-conversion is a very demanding process, full exploitation of coherence so that incremental calculations can be done is essential. For each pattern, the colour of adjacent pixels as well as the intersections of the edges of a pattern with the next pixel-row can be calculated incrementally. The exploitation of these coherencies of the patterns reduces the processing power requirements of the display controller dramatically (see the Appendix). Even with these incremental calculations several hundred MIPS are required for real-time scan-conversion. However, due to the technological limitations we have to face today, the capacity of processing elements will be limited to an order of 10 MIPS.

Therefore, if we stick to the idea of refreshing from the LDF, the bandwidth requirements as well as the processing requirements enforce a multiprocessor implementation of the Display Controller.

In this paper we present the basic structure of the Display Controller as shown in Figure 1, designed with the above considerations in mind. The Display Controller consists of an Increment Processor (IP) capable of painting pixels on the display at refresh speed, an Active Patterns Store (APS) implemented as an on-chip memory of the IP, a high band-width LDF/DC interface, and a Pattern Loader (PL) which loads the active patterns from the LDF into the APS. Note that both the IP and the PL must be realized as multiprocessor arrays.

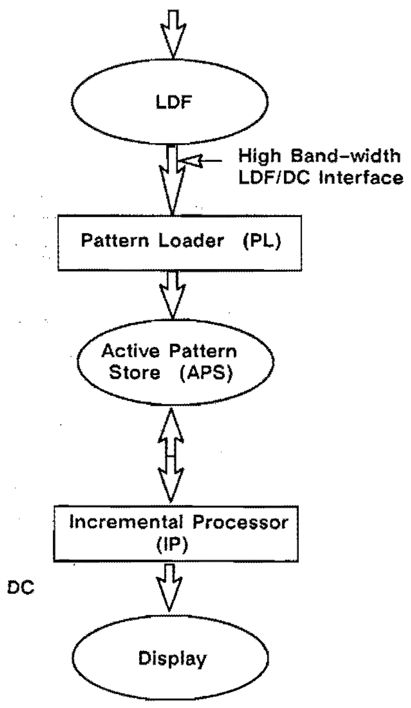

LDF
High Band-width LDF/DC Interface
Pattern Loader (PL)
Active Pattern Store (APS)
Incremental Processor (IP)
DC
Display

Figure 1: Functional block diagram of the Display Controller (DC). It consists of a Pattern Loader (PL) that loads the active patterns in the on chip Active Pattern Store (APS). These Active Patterns are used by the Increment Processor (IP) to produce the pixel stream. Note that PL as well as IP are multiprocessor arrays.

2. A Multiprocessor Display Controller.

During the design of the multiprocessor Display Controller, the following aspects have been taken into account.

- Since all of the elements in the DC cannot be integrated into a single chip, the DC must be partitioned. This partitioning should not cause any considerable degradation of the DC performance.
- The system should be scalable. That is, it should be adaptable to different resolutions and complexity demands.
- The adaptability should show a performance improvement linear with the increase of hardware.
- Ability to use the same architecture for increased packing densities (i.e. the architecture should not be determined by current VLSI technology).

- Ability to handle images consisting of lots of small patterns having rather simple colour functions as well as images consisting of fewer patterns, but having more complex colour functions.
- Implementable in VLSI.

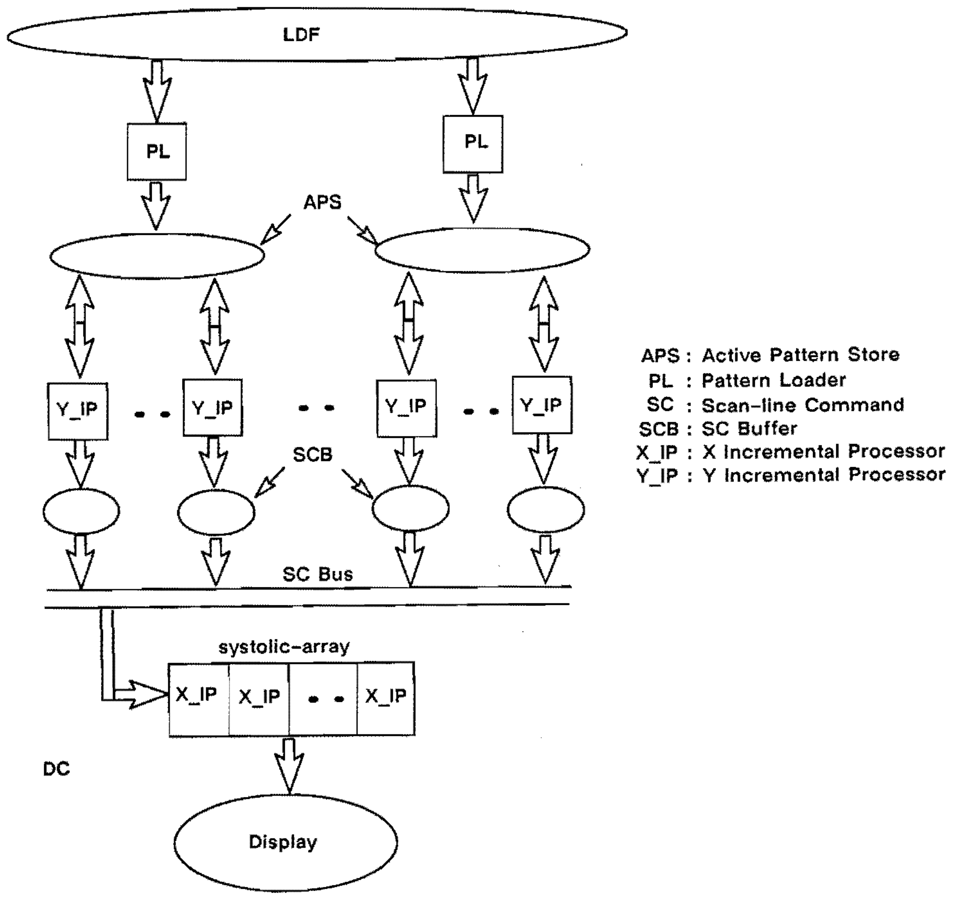


Figure 2: The architecture of the Display Controller.

The architecture presented in Figure 2 satisfies these requirements. This figure merely presents the data flow through the system. The control flow will be discussed in the next section. The high band-width of the LDF/DC interface, has been achieved by parallel accesses of the LDF by multiple PLs. Due to the fact that patterns in the LDF do not overlap, the PLs can work completely independently.

The nature of the DC process —incrementally generating streams of pixels, row by row— suggests a splitting of these incremental operations in horizontal (x) and in vertical (y) direction. The y-increments to calculate both the intersection D(x) of a pattern with the current scanline as well as the colour function C(x) of that pattern on the current scanline. The x-increments to calculate the pixel values of that section. Since the time constraints of these two processes differ in an order of the number of pixels on a scanline, it is reasonable to suggest an implementation of the IP array by two different types of processor elements, X_IP and Y_IP.

The Y_IPs calculate the intersections of the pattern edges with the current pixel-row, the colour values at the left most edge, the incremental colour values along the pixel-row direction, etc, and generate from this the scanline commands that are sent into the Scanline Command Buffer (SCB).

The X_IPs are pixel processors, of which there are as many as there are pixels on the pixel-row. The X_IPs are connected as a systolic-array. The left-most X_IP is fed by scanline commands from the Scanline Command buffers. Each X_IP performs incremental calculations on the scanline command it receives which is then passed on to its right neighbour. Depending on the type and destination range of a command, internal registers of the X_IPs are updated. Between commands from one scanline and the next, a special Refresh command is fed to the X_IP array, which causes the X_IP that receives this command to output the resulting colour value and reset its internal registers to be able to start calculations for the next pixel-row. Since due to this mechanism Refresh commands directly control the pixel flush, they will have to come at regular intervals, dictated by the total line time.

The architecture as presented in Figure 2 inherits the following features:

- The number of X_IPs can be adapted to any display resolution desired.
- The ratio of processor elements can be tuned to maximize the throughput of the DC. (e.g. if the throughput of a Pattern Loader seems to be larger than that of a Y_IP, one PL can be made to serve several Y_IPs.)
- PLs and Y_IPs can be added to the DC in order to increase the processing power. With this, the performance of the system can be increased up to a level where it can render the most complex pictures (i.e. pictures without any coherence between the pixels).
- Realistic pictures with multiple light sources, and objects with diverse shading properties can be scan-converted in real-time. The realism of the generated picture can be improved by anti-aliasing the edges of the patterns (see the Appendix).

As the processors in the systolic-array have to perform some calculations before they transfer data to their neighbouring processors, the transfer speed and consequently the maximum resolution is limited (with the targeted technology this will be in the order of $1\,K \times 1\,K$). Due to the anti-aliasing capabilities, the effective resolution of the display can be improved beyond that. Alternatively, if a higher resolution is essential even without anti-aliasing, one can use a number of X_IP arrays in parallel. As the speed of VLSI implementations is continuously increasing, it can be expected that within a few years time, also very high resolution displays can be refreshed using a single X_IP array.

3. Partitioning the Multiprocessor Display Controller

As sub-micron technology and wafer scale integration are not available on a cheap commercial basis, the DC has to be partitioned into several chips. As we mentioned before, this partitioning must not degrade the performance of the DC. Figure 3 shows the different levels of partitioning that can be done.

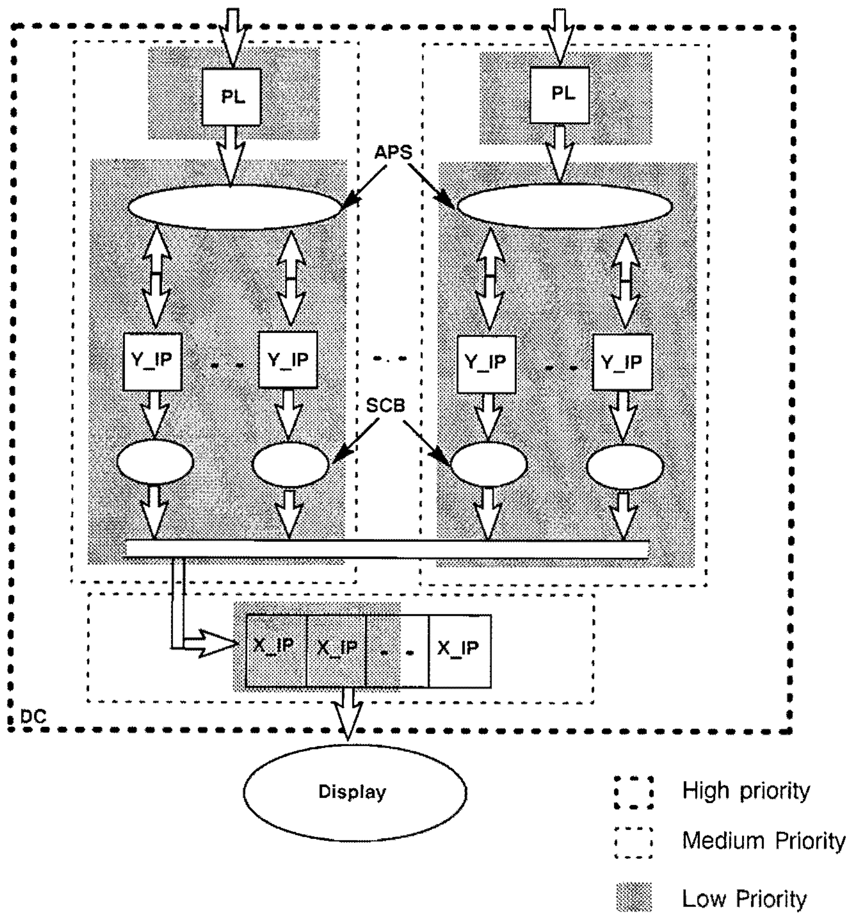


Figure 3: The different levels of partitioning of the Display Controller.

As indicated by the thick dashed box, it would be desirable to have one complete, maximally configured, DC in a single chip. This however is impossible, not only due to limitations of the silicon technology, but also due to the large number of I/O pins needed for a high band-width LDF/DC interface (which is realized by multiple busses). For this, the DC can be partitioned as indicated by thin dashed boxes. This partitioning includes a PL and some Y_IPs on one chip, and the X_IPs on another chip. If even more partitioning is enforced by the technology available, we can do it as indicated by filled boxes. Here only a few X_IPs are integrated on one chip, the PL on a separate chip, and some Y_IPs with the APS on another chip.

4. Implementation of the Multiprocessor Display Controller

4.1. The Pattern Loader

As the output of the X_IP array is directly used to refresh the display, the DC will have to fulfill high throughput requirements. As a result programmability of the DC must be kept to a minimum. On the other hand, in order to be able to adapt to future developments in lighting models, mapping techniques etc, we do not want to impose a severe restriction upon the representation in the LDF. Therefore any mismatch between the data representations in the LDF and the representations as needed by the increment processors must be resolved by a data dependent mapping, i.e. by making the PL programmable.

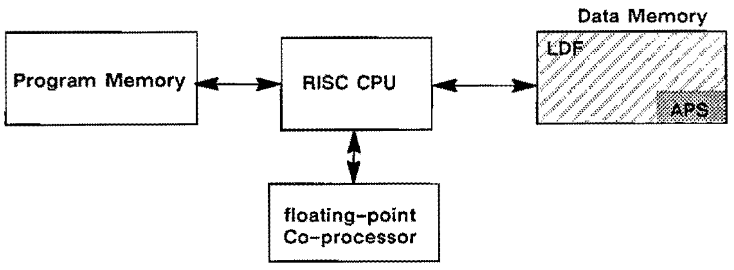


Figure 4: The basic block structure of the Pattern Loader.

Because of the high throughput requirements, calculations performed by the increment processors will have to be done using fixed-point numbers. Floating-point numbers, however, are indispensable for up stream processes in the image generating pipeline, such as the hidden-surface removal algorithm.

As a result, we propose the PL to be a pipe-lined RISC processor with a floating-point to fixed-point converter (preferably on-chip for maximum throughput). Since the LDF contains the data of the patterns, a separate memory is employed to store the RISC processor program. Figure 4 shows the basic blocks of the Pattern Loader. The data memory of the RISC processor consists of the LDF memory and APS memory. The PLs will operate in MIMD (Multi-Instruction Multi-Data) mode, to be able to handle differently structured data.

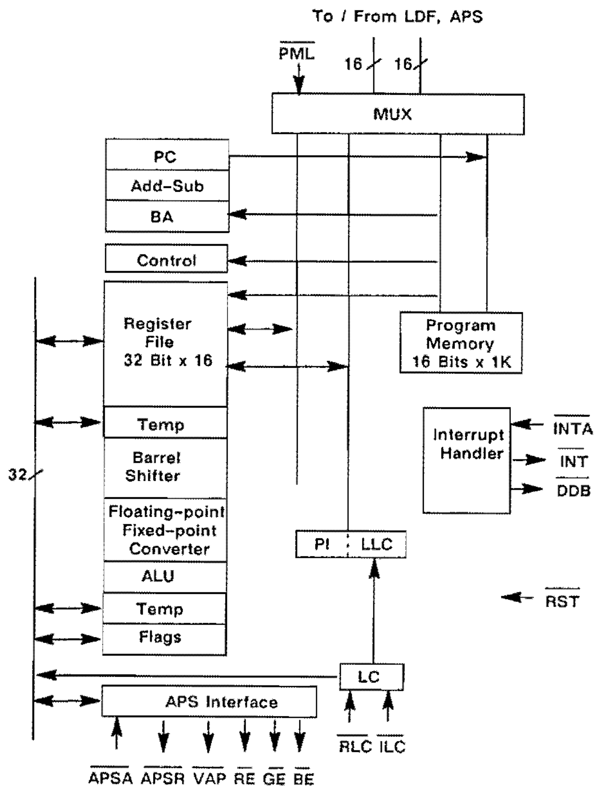
The function of the Pattern Loader (PL) is to transfer the active patterns from the Low-level Display File (LDF) into the Active Pattern Store (APS). The internal architecture of the proposed RISC CPU, and it's instruction set are given in Figure 5 and Table 2 respectively. The estimated packing density is one PL per 1 μ CMOS chip. The number of I/O lines connected to the PL is about 40. The PL could also be implemented using a general purpose processor, probably at the cost of some performance degradation. Functions to be performed by the PL's hardware are described in the coming sub-sections.

4.1.1. Program Loading Mode. In this mode the $\overline{\text{PML}}$ signal will be kept low by the host, and the PL's program will be down loaded via the PL's data bus, otherwise used for LDF and APS memory access only. In this mode the normal operation of the PL will be suspended. Once the complete program memory is down-loaded, the $\overline{\text{PML}}$ line will be set on a high level again and the PL can start the pattern loading process.

4.1.2. Index Table Construction Mode. Although it may be possible to find the patterns contributing to the pixel-row being scan-converted on the fly, it is much more efficient to make use of an index table, i.e. a table, indicating which patterns are contributing to a given pixel-row. Modification of such an index table has to be carried out during the vertical retrace time, since at that time, no patterns are scan-converted. The index table can be stored in the LDF memory. There should be an entry in the index table for each pixel-row on the display and each entry of the index table must indicate which patterns become active on the pixel-row in question. This index table mechanism can be used to assure that pattern loading can be sustained at a sufficient rate.

4.1.3. Pattern Loading Mode. In this mode the active patterns indicated by the index table, will be transferred into the APS. As fixed-point numbers are used within the Display Controller, floating-point numbers will be converted into fixed-point representation. Complex patterns which, due to the fixed sized APS segments cannot be stored as a whole, must be decomposed into simple patterns. Sudden peaks in the pattern loading process can be minimized by loading patterns in advance.

The signals $\overline{\text{APSR}}$, $\overline{\text{APSA}}$, $\overline{\text{VAP}}$, $\overline{\text{RE}}$, $\overline{\text{GE}}$ and $\overline{\text{BE}}$, are used to load the active patterns into the APS. The PL will lower the signal on the $\overline{\text{APSR}}$ line (by a RAPS instruction) in order to get a free segment of APS memory. In return $\overline{\text{APSA}}$ will be lowered, if a free APS memory segment is available. The PL will wait for this signal if the WAPSA instruction has already been executed. The $\overline{\text{RE}}$, $\overline{\text{BE}}$ and $\overline{\text{BE}}$ signals are used for selective transfer of colour dependent data. Once an active pattern is loaded, the PL signals on $\overline{\text{VAP}}$, indicating the incremental processors can scan-convert the pattern.



APSA	APS Acknowledge
APSR	APS Request
BA	Branch Address Reg.
BE	Blue Enable
GE	Green Enable
INT	Interrupt
INTA	Interrupt Acknowledge
IRC	Increment LC
LC	Line Counter
LLC	Last LC Reg.
MUX	2:1 Multiplexer
PC	Program Counter
PI	Processor Identity Reg.
RE	Red Enable
RLC	Reset LC
RST	Reset
Temp	Temporary Reg.
VAP	Valid Active Pattern

Figure 5: Register level architecture of the Pattern Loader.

The on-chip counter LC indicates which pixel-row is being scan-converted, whereby the PL can check whether it is loading the patterns in time. Such a check is needed, since a delay in the $\overline{\text{APSA}}$ signal might delay the PL. The LC counter is incremented by the signal on $\overline{\text{ILC}}$ and reset by $\overline{\text{RLC}}$ (Note that if a general purpose CPU is used for the PL, LC must be implemented in external hardware). If the patterns could not be loaded in time the host computer must be informed about this situation. The $\overline{\text{INT}}$ signal, generated by the GINT instruction, is used to report pattern loading problems to the host. The $\overline{\text{INT}}$ signal will transfer the value of the LC when the problem occurred (which is stored in LLC register) as well as a pattern identification number (which is stored in the PID register), when the host responds via the line $\overline{\text{INTA}}$. In order to reduce the pin count of the PL, these data will

Data Transfer instructions		General Control instructions	
LD Ri, *M	Ri <- <*M>	BRA *Ri	Branch always to *Ri
LDI Ri, D	Ri <- D	BRC *Ri	Branch on C to *Ri
MV Ri, Rj	Ri <- Rj	BRZ *Ri	Branch on Z to *Ri
ST *Rj, Ri	<*Rj> <- Ri	CALL *Ri	Branch to subroutine *Ri
ST *Rj+, Ri	<*Rj> <- Ri	CLC	Reset C flag
	Rj <- Rj+1	FPC Ri, Rj	Ri <- fixed point Rj
ST *-Rj, Ri	Rj <- Rj-1	NOP	No operation
	<*Rj> <- Ri	RST	Return from subroutine
		SETC	Set C flag
Arithmetic / Logical instructions		**Pattern Loading Control instructions**	
ADD Ri, Rj	Ri <- Ri+Rj	COLE D	Colour Enable
ADDC Ri, Rj	Ri <- Ri+Rj+C	GINT	Generate interrupt
ADDI Ri, D	Ri <- Ri+D	RAPS	Request APS segment
ADDIC Ri, D	Ri <- Ri+D+C	RDLC Ri	Ri <- LC
AND Ri, Rj	Ri <- Ri And Rj	SVAP	Signal VAP
ASL Ri, Rj	Arithmetic left shift	WAPSA	Wait until APSA
	Ri by <Rj> bits		
ASR Ri, Rj	Arithmetic right shift		
	Ri by <Rj> bits		
LSL Ri, Rj	Logical left shift		
	Ri by <Rj> bits		
LSR Ri, Rj	Logical right shift		
	Ri by <Rj> bits		
NEG Ri	Ri <- -Ri		
NOT Ri	Ri <- Not Ri		
OR Ri, Rj	Ri <- Ri Or Rj		
SUB Ri, Rj	Ri <- Ri-Rj		
SUBC Ri, Rj	Ri <- Ri-Rj-C		
SUBI Ri, D	Ri <- Ri-D		
SUBIC Ri, D	Ri <- Ri-D-C		

Table 2: Instruction set of the Pattern Loader. Ri, Rj are registers R0, R1,..R15. *M and *Ri is the address given by M or Ri and <*M> and <*Ri> is the data at the corresponding address.

be sent on the PL's data bus, so that during this time the pattern loading will temporary be suspended. The line $\overline{\text{DDB}}$ will be lowered by the PL in order to disconnect the data bus from the LDF.

The operation of the PL is pipelined and external data and address busses are allocated to the LDF and APS memory by time multiplexing.

4.2. The Active Pattern Store

For maximum parallelism of the PLs and IPs, the APS must be implemented as dual-port memory. The memory locations which contain the Active Patterns must be well protected from the PL access. A segmented APS will simplify the management, while supporting the throughput requirements. For efficient usage of the APS memory, the size of the segments must be kept to a minimum. On the other hand, such a strategy will reduce the overall throughput rate due to increased PL work load. Therefore the APS must be able to store patterns containing a few pattern slices (say 5-10). Because of the pattern representation used [5] any complex pattern can be sub-divided into simple domains very easily so that the PL is indeed able to decompose complex patterns on the fly into the simple patterns which can be stored in APS segments.

The APS management will be implemented by a flag mechanism (See Figure 6). As soon as the PL has loaded an active pattern into the APS, a flag is raised in the corresponding APS segment, indicating that this segment contains an active pattern. We refer to this flag as Active Pattern Flag. The Y_IPs must process the APS segments of which Active Pattern Flag is raised. As soon as the data of the APS segment becomes outdated, Y_IP must reset the Active Pattern Flag and raise another flag indicating that the corresponding segment is free for PL access. The above flag mechanism protects the APS segments containing valid data and provides a fast response to the requests made by the PL.

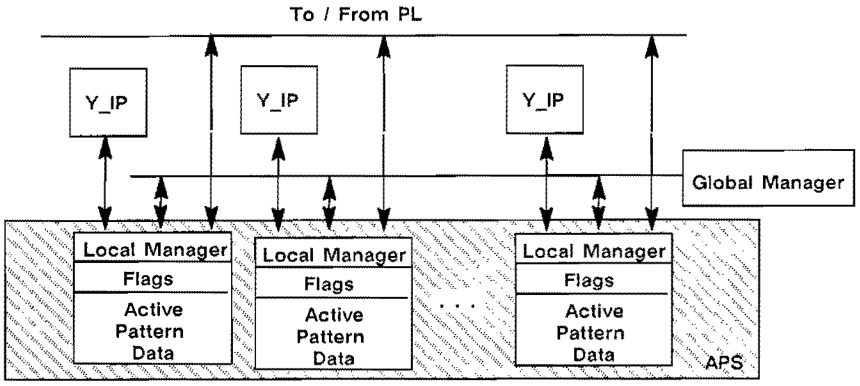


Figure 6: Implementation of the Active Pattern Store.

The Global Manager keeps track of all free APS segments, while the Local Manager connects its APS segments either to the PL or to the Y_IP depending on the commands received by the Global Manager and the flag settings. One Global Manager is assigned to each Pattern Loader. Due to the Local Manager switch mechanism, the number of APS segments available is transparent to the PL, so that a variation of the number of APS segments can be done very easily. Furthermore it reduces the width of the address bus between the PL and the APS.

4.3. The Y Increment Processor

Now let us turn our attention to the implementation of the Y_IP. Due to the diversity of the patterns (they may be textured or not, have different number of slices etc.), a highly specialized processor is not suitable. On the other hand the high throughput requirements can only be met by a highly specialized processor. Therefore, the Y_IP has to have a restricted programmability. For this, the Y_IP will also have to be a pipe-lined RISC processor. Its instruction set will be small compared to that of the PL.

As a large number of Y_IPs will be needed —more then there are PLs—, an instruction transmission mechanism is proposed. Instructions are generated by the Instruction Generator and flow through the Y_IPs as shown in Figure 7. Therefore, the Y_IPs operate on SIMD (Single Instruction Multi-Data) mode with some phase difference. The reasons for this instruction transmission mechanism are two-fold: As described in the Appendix, only a few number of multiplications are required for the scanline command generation. Hence one multiplier can be used by neighbouring Y_IPs when they operate in different phases. Furthermore an instruction transmission mechanism reduces the driving requirements of the Instruction Generator.

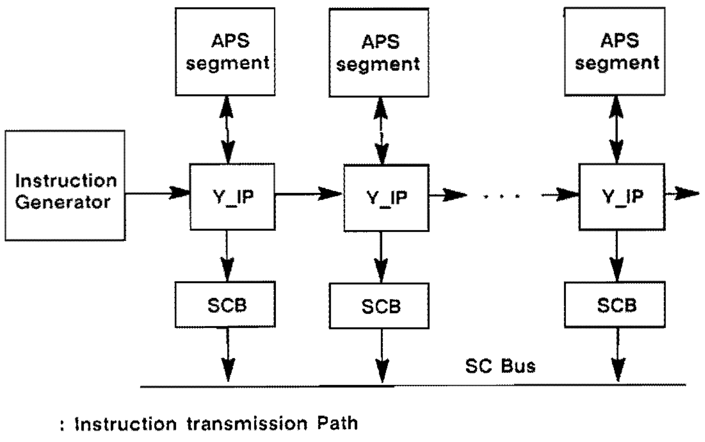


Figure 7: The instruction transmission mechanism of the Y_IP array.

The Y_IPs update the y-dependent data items in the APS such as the intersections points of the edges and colour values and they generate Scanline Commands. These commands will be stored in the SCB. As all Y_IPs must send their scanline commands to the X_IP array, the SC Bus must be properly arbitrated. This SC Bus arbitration is discussed in next the section.

The scanline commands to be generated consist of EVAL and SET commands described in section 4.5. The architecture of the Y_IP processor is presented in Figure 8. The active patterns are stored in the APS. Generated scanline commands are stored in the FIFO

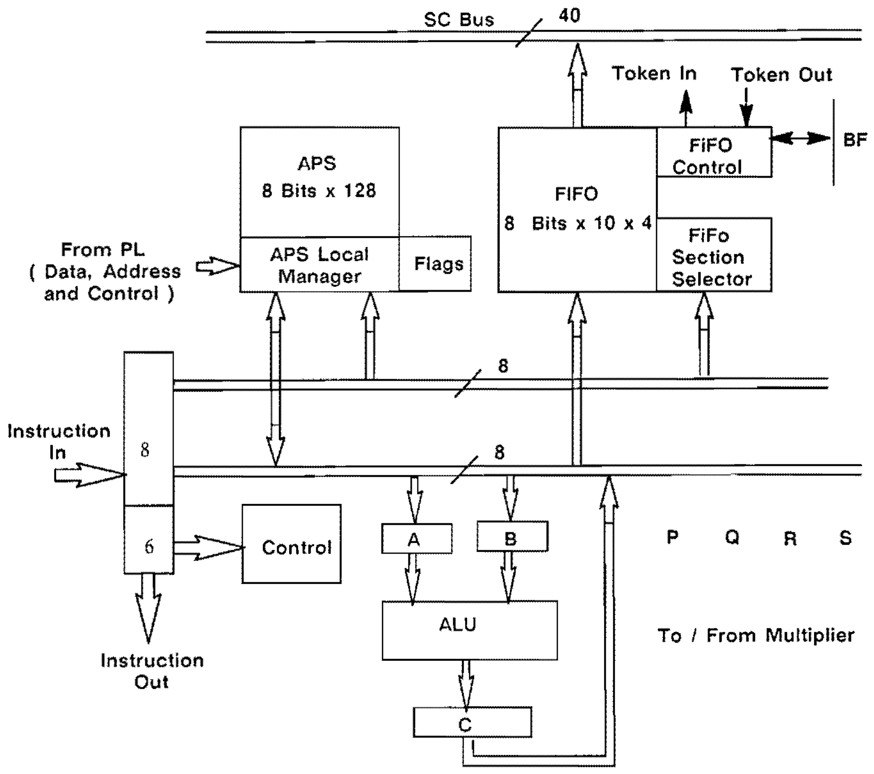


Figure 8: Architecture of the Y_IP processor. A,B,C are registers whereas P,Q,R,S are virtual registers shared by several Y_IP processors.

memory. Although 32 bit representation is used for the intensity data in the scanline commands (see section 4.5), the internal busses in the Y_IP processor are not 32 bit wide. Reasons to use more narrow data busses in the Y_IP processor are three fold. In the first place, the number of Y_IP processors integrated in one chip should preferably be as large as possible. The second reason is that data paths of different widths are required for the intensity and pixel location data. Although 32 bit data are needed for the intensity data, for the pixel location data 10 to 12 bits are sufficient. The third and main reason is the necessity of a multiplier for the scanline command generation. If the data path of the Y_IP processor is small, a multiplier could be integrated into the Y_IP processor chip quite easily. The timing constraints for the scanline command generation is not as severe as the evaluation of scanline commands. Hence smaller data paths within the Y_IP processor will not noticeably degrade the performance of the system. However, if the PL can only access the APS memory via narrow data paths, the speed of the pattern loading will become insufficient. Hence the PL must be able to write into APS memory via sufficiently wide data paths.

As stated before, the number of multiplications required for the scanline command generation is small, so that a serial multiplier and a divider is proposed (note that an array multiplier or divider needs a very large silicon area). The instruction set of the Y_IP processor is given in Table 3. All instructions, except MULT and DIV are single cycle instructions. Since the multiplier and divider are serial they need more cycles. In order to provide sufficient multiplications and divisions, a serial multiplier and divider which can supply results after w cycles, where w is the word length (8 bits), is proposed.

Data Transfer instructions		Arithmetic / Logical instructions	
LD Ri, *M	Ri <- <*M>	ADD	C <- A+B
LDI Ri, D	Ri <- D	ADDC	C <- A+B+Cin
MV Ri, Rj	Ri <- Rj	AND	C <- A And B
ST *M, Rj	<*M> <- Rj	ASL Rk, d	Arithmetic left shift
			Rk by d bits
General Control instructions		ASR Rk, d	Arithmetic right shift
			Rk by d bits
AYIP	Active if valid pattern	LSL Rk, d	Logical left shift
CLRCin	Clear Cin flag		Rk by d bits
DYIP	Deactivate	LSR Rk, d	Logical right shift
HTCin	Halt if Cin flag true		Rk by d bits
HTNCin	Halt if Cin flag false	MULT	R, S <- P×S
HTZ	Halt if Z flag true	NEG Rk	Rk <- -Rk
HTNZ	Halt if Z flag false	NOT Rk	Rk <- Not Rk
NOP	No operation	OR	C <- A Or B
NER	New pixel row	SUB	C <- A-B
SETCin	Set Cin flag	SUBC	C <- A-B-Cin
RST	Restart		

Table 3: Instruction set of the Y_IP processor. Ri is one of the registers A, B, P or Q. Rj is one of the registers C, R or S while Rk is register A or B. *M is the address given by M and <*M> is the data at the corresponding address.

Due to the diversity of the patterns, the data in the APS will not be uniformly structured. As instructions are transmitted along the Y_IP array, one can generate only scanline commands for those patterns which have similar attributes. Hence the processors which are assigned to differently attributed patterns than the one being processed must be disabled. For this purpose, we employ the HALT instruction. As the disabled processors will reduce the processor utilization, we can use a Data Independent Flow (DIF) type programming style to maximize the utilization of the processors. The estimated packing density of the Y_IP processor is about 16 processors per chip with associated memory in 1 μ CMOS technology. The number of I/O lines is expected to be about 100.

4.4. The Scanline Command Buffer

For the maximum parallelism of the processors in the DC, the SCB must be freely accessible for the Y_IPs and X_IP array independently. Hence a double buffered memory, or a FIFO must be used.

In order to arbitrate the SC Bus, a token mechanism is proposed. Nearest neighbour connection for the token flow is best for VLSI implementations. As shown in Figure 9, the left most SCB receives the token from the Instruction Generator. As soon as the SC bus is free (indicated by the BF line), the SCB which holds the token sets a "busy" on the BF line and starts to transfer SC commands. Each SCB keeps the token as long as it has SCs to be transferred to the X_IP array. When the SCB is ready it releases the busy from the BF line and passes the token to its right neighbour. The right most SCB returns the token to the Instruction Generator. Upon this, the Instruction Generator will feed NOP instructions to the X_IP array followed by the Refresh command. The Refresh command has to maintain synchronization, so that these NOP instructions have to fill up the time left before the Refresh command can be issued.

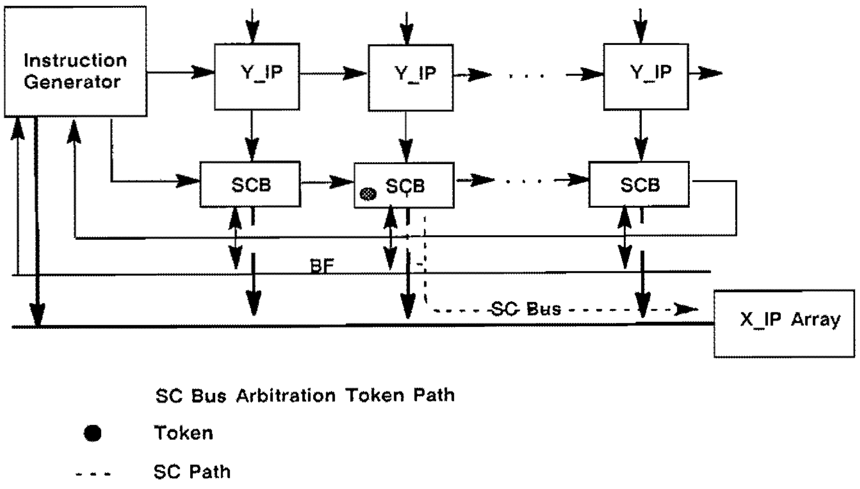


Figure 9: The Scanline Command Bus arbitration mechanism.

The space available in a SCB must be sufficiently large to store all the scanline commands required for the most complex pattern shading. In such complex cases, one pattern may generate several scanline commands. If a large number of patterns are complex shaded, the X_IP array may not be able to processes all scanline commands generated, in which case the token sent into the SCBs will not complete its round-trip in time. As the scan-conversion must be continued in real-time, such tokens must be purged, a new token must be issued and the host has to be notified. Such an exceptional situation can occur only if there are too many small patterns having a too high shading complexity. As a rule of thumb, patterns in the LDF should in principle not generate more instructions for a scanline than a constant times the number of pixels they cover on that scanline. Individual patterns may exceed this number, as long as this is averaged out by less demanding patterns.

4.5. The X Increment Processor

Ambient, diffuse, and specular components of the generally applied shading models as well as depth cueing and periodic textures, can be painted using incremental calculations. As described in the Appendix, piece-wise second order curves can be used to approximate the diffuse and specular reflections. The effect of depth cueing can be generated by changing the intensity gradients. Furthermore periodic textures can be generated by setting intensity, first derivative and/or second derivative at well defined pixel locations. Calculation of the effects of multiple light sources would be greatly simplified if partial contributions can be summed.

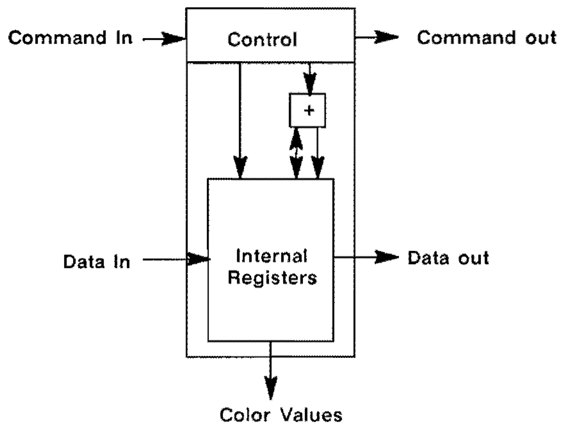


Figure 10: Basic structure of an X_IP element.

Given this, we designed a processor array of which the elements can execute the instruction set presented in Table 4. This array of processors will be able to support a fairly complex shading strategy. The instructions traverse the X_IP array and data associated with each instruction will be updated at each processor as indicated in Table 4. The block diagram of an X_IP processor is given in Figure 10.

Command	Description
SETI(X, DX, I)	Set the intensity at the pixel locations X, X+DX, .. to I.
SETDI(X, DX, DI)	Set the first derivative of the intensity at the pixel locations X, X+DX, .. to DI.
SETDDI(X, DX, DDI)	Set the second derivative of the intensity at the pixel locations X, X+DX, .. to DDI.
DIS(X, DX)	Disable the update of the R register

EVAL1(X,DX,I, DI, DDI)	Update the intensities between pixel locations X and X+DX. If pixels are between X and X+DX, then R, I, DI are updated as follows
	R <- R+I,
	I <- I+DI,
	DI <- DI+DDI
	If I, DI or DDI has been set by preceding SET commands, use the values provided by the SET commands.

EVAL2(X,DX,I, DI)	Update the intensities between pixel locations X and X+DX. If pixels are between X and X+DX, then R, I are updated as follows
	R <- R+I,
	I <- I+DI,
	If I or DI has been set by preceding SET commands, use the values provided by the SET commands.

EVAL3(X,DX,I)	Update the intensities between pixel locations X and X+DX. If pixels are between X and X+DX, then R is updated as follows R <- R+I, If I has been set by preceding SET commands, use the values provided by the SET commands.
EVAL4(X, I)	Update the intensities between pixel locations X and X+DX. If pixels are between X and X+DX, then R is updated as follows R <- I,
REFRESH()	Refresh the display by the content of R and reset all registers to be able to calculate the colours of the next pixel row.
NOP()	Do nothing. This command is used to maintain the synchronism.

Table 4: The X_IP command set.

The function of the X_IP processor is to calculate the pixel intensities incrementally along the pixel-row direction, and to sum partial contributions generated for instance by individual light sources. The following code describes the operation of the X_IP processor in detail.

```
Read X, DX, I, DI, DDI provided with the command;
Use new data for I, Di, DDI if protect flags are false or use previously stored data when protect flag is true;

X--
switch (Command)
```

```
case "EVAL1": if (Eval == TRUE) {   if (X == 0) {  Eval = FALSE;
                                                   Command = NOP;        }
                                    else {         If (Dis == FALSE)     Colour += I;
                                                   I += DI;
                                                   DI += DDI;     }      }
              else if (X == 0) {    Eval = True;
                                    X = DX;
                                    if (Dis == FALSE)     Colour += I;
                                    I += DI;
                                    DI += DDI;                           }
              Protect_I = Protect_DI = Protect_DDI = FALSE;
              Dis = TRUE;
              break;

case "EVAL2": if (Eval == TRUE) {   if (X == 0) {  Eval = FALSE;
                                                   Command = NOP;        }
                                    else {         If (Dis == FALSE)     Colour += I;
                                                   I += DI;       }      }
              else if (X == 0) {    Eval = True;
                                    X = DX;
                                    if (Dis == FALSE)     Colour += I;
                                    I += DI;                             }
              Protect_I = Protect_DI = Protect_DDI = FALSE;
              Dis = TRUE;
              break;

case "EVAL3": if (Eval == TRUE) {   if (X == 0) {  Eval = FALSE;
                                                   Command = NOP;        }
                                    else           If (Dis == FALSE)     Colour += I;}
              else if (X == 0) {    Eval = True;
                                    X = DX;
                                    if (Dis == FALSE)     Colour += I;   }
              Protect_I = Protect_DI = Protect_DDI = FALSE;
              Dis = TRUE;
              break;

case "EVAL4": if (X == 0 && Dis == FALSE) Colour = I;
              Dis = TRUE;
              break;

case "SETI":  if (X == 0) {  X = DX;
                             Store_I;
                             Protect_I = TRUE;     }
              break;

case "SETDI": if (X == 0) {  X = DX;
                             Store_DI;
                             Protect_DI = TRUE;    }
              break;

case "SETDDI":if (X == 0) {  X = DX;
                             Store_DDI;
                             Protect_DDI = TRUE;   }
              break;
```

```
case "DIS":    if (X == 0 && Eval == FALSE){      X = Dx;
                                                            Eval = TRUE;
                                                            Dis = TRUE;    }
               if (X == 0 && EVAL == TRUE)        Command = NOP;
               break;

case "REFRESH": Output Colour;
               Reset Registers;
               break;

case "NOP":    break;
```

Command	Time slot #1	Time slot #2	Time slot #3	Time slot #4	Time slot #5
SETI(X, DX, I)	X, DX	I			
SETDI(X, DX, DI)	X, DX	DI			
SETDDI(X, DX, DDI)	X, DX	DDI			
DIS(X, DX)	X, DX				
EVAL1(X, DX, I, DI, DDI)	X, DX	DDI	DI	I	Accumulate
EVAL2(X, DX, I, DI)	X, DX	DI	I	Accumulate	
EVAL3(X, DX, I)	X, DX	I	Accumulate		
EVAL4(X, I)	X, I				
REFRESH()	Refresh				
NOP()	No operation				

Table 5: Decomposition of the scanline command set.

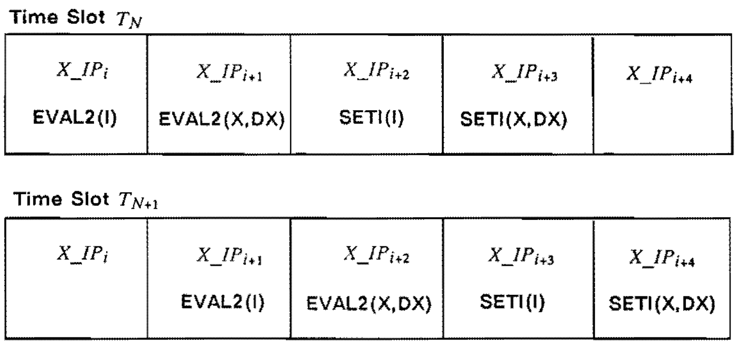
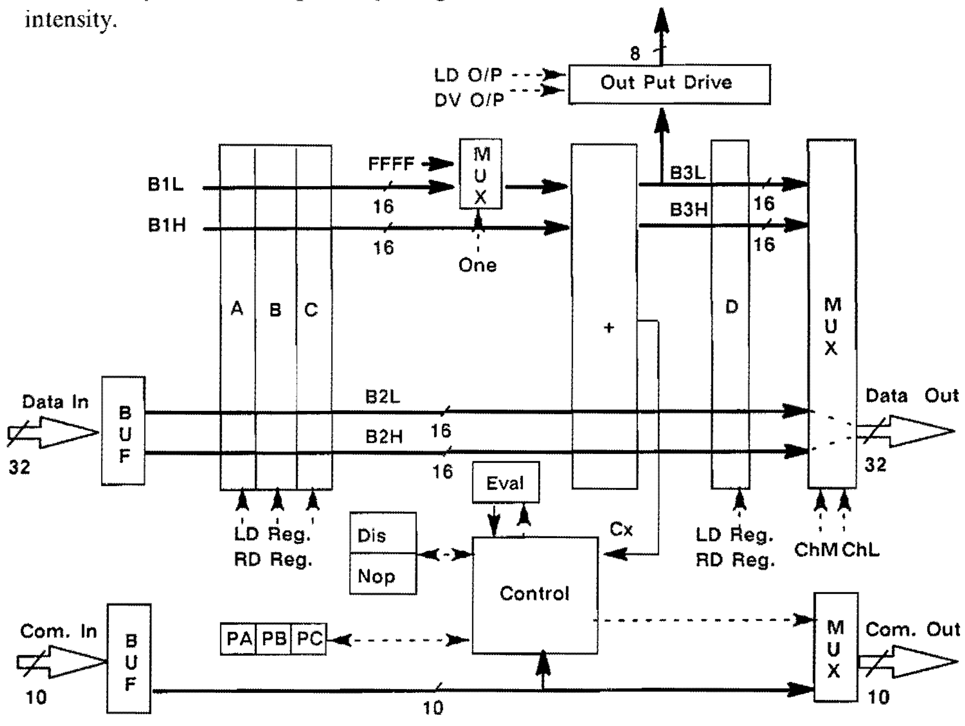


Figure 11: Distribution of the decomposed scanline command data among X_IP processors in two consecutive time-slots.

I, DI, DDI are represented by 32 bit fixed-point numbers in the realization proposed. Due to the large number of bits involved, the I, DI, DDI cannot be sent in parallel. Hence only one 32 bit input and one output data port will be used and consequently I, DI, DDI and the couple X, DX will be sent serially. Since 10-12 bits are sufficient for the representation of the X and DX, we can send the X and DX simultaneously on the LSB and MSB side of the 32 bit data bus. Table 5 shows the decompositions and the sequential order of the data as it will be sent into the X_IP array. The sequential ordering has been optimized to maximize the throughput of the X_IP processor. The decomposed data will be sent into the X_IP processor array in a pipe-lined fashion. Figure 11 shows an example of scanline data in two consecutive time slots.

Figure 12 shows the architecture of the X_IP processor at register transfer level. Registers A, B and C are used to store the intensity, the first derivative and the second derivative of the intensity function respectively. Register D is used to hold the accumulated colour intensity.



A,B,C,D	Internal Registers	LD Reg.	Load Register
PA,..PC	Protect Flags	RD Reg.	Read Register
Eval	Evaluate Flag	ChM	Change MSB Part
Nop	No Operation Flag	ChL	Change LSB Part
MUX	2:1 Multiplexer	B*L	Internal Data Bus Low Side
BUF	Buffer	B*H	Internal Data Bus High Side

Figure 12: Register level architecture of the X_IP processor.

The EVAL commands will update the appropriate registers when the processor location is between X and X + DX.

The SET commands will set the appropriate registers when the processor location is X, X + DX, X + 2DX,.. and raise the appropriate protect flag. When this protect flag is true the EVAL commands do not modify the content of the corresponding register. As the SET commands must effect only one EVAL command, the protect flags will be reseted during the last cycle of each EVAL command.

The processors which are between the pixel locations X and X + DX or processors on the pixel locations X, X + DX, X + 2DX, .. are identified as follows. At each processor the values of X is decremented by 1 and DX is substituted for X when X = 0. This mechanism will set Cx = 0 when X, X + DX, X + 2DX,.. are zero. (Cx is carry out from the MSB of X).

Upon the REFRESH command, the content of the D register will be transferred to the output circuitry and all registers and their flags will be cleared to be ready to process the scanline commands on the next pixel-row.

The estimated packing density of the X_IP processor is about 8 processors per chip in 1 μ CMOS technology. The number of I/O lines required is about 100.

Shading Method	Maximum throughput (in M quadrilaterals per sec.)		
	in general	512 × 512 pixels 50 Hz	1024 × 1024 pixels 50 Hz
Phong shading	K/9h	1.82/h	7.58/h
Gouraud shading	K/4h	4.09/h	17.09/h
Gouraud shading and depth cueing	K/8h	2.04/h	8.53/h
Constant shading	K/3h	5.46/h	22.74/h
Textured shading	K/((4+2q)*h) q = 2, 3,...	2.73/h for q=2	11.37/h for q=2

Table 6: Maximum throughput of the X_IP array.

5. Some Performance Figures

Let us assume the resolution of the display is M×N pixels and frame rate is R Hz. Furthermore assume the pixel clock speed is C_p MHz and one pixel row is refreshed within T_l μseconds. According to the design of the X_IP array we have $K = T_l N R C_p$ Mclock cycles per second to send the scanline commands to the X_IP array. Table 6 shows the maximum throughput that can be achieved by a single X_IP array. In order to load the X_IP array to its maximum capacity, PL's must be able to transfer the same amount of patterns to the APS. For geometrically homogeneously distributed pattern heights and pattern loading times (with average height h pixels rows and average loading time t seconds) the average number of APS segments occupied is given by [4]:

$$\mathrm{O} = \mathrm{L}\left[\frac{\mathrm{hT_l}}{\mathrm{t}}\right]\frac{\displaystyle\sum_{\mathrm{i}=1}^{\mathrm{i}=\#\mathrm{APS}}\left[\frac{\#\mathrm{APS}\,!}{(\#\mathrm{APS}-\mathrm{i})\,!}\right]\left[\frac{\mathrm{t}}{\mathrm{hT_l}}\right]^{\mathrm{i}}}{1+\displaystyle\sum_{\mathrm{i}=1}^{\mathrm{i}=\#\mathrm{APS}}\left[\frac{\#\mathrm{APS}\,!}{(\#\mathrm{APS}-\mathrm{i})\,!}\right]\left[\frac{\mathrm{t}}{\mathrm{hT_l}}\right]^{\mathrm{i}}}$$

where #APS is the number of APS segments connected to a PL, L is the number of PLs and O is the average number of APS segments occupied.

It can be shown that the best pattern loader utilization and APS segment occupancy occurs when $\#\mathrm{APS} \approx \frac{\mathrm{hT_l}}{\mathrm{t}}$ (see [4]). Based on these results, Table 7 shows the number of APS segments (hence number of Y_IPS) and PLs required for the maximum throughput.

Shading method	# PL cycles	Hardware needed for maximum throughput (PL cycle time = 50 ns, h = 10 pixel rows)					
		512 × 512 pixels 50 Hz			1024 × 1024 pixels 50 Hz		
		#APS	K	#APS × K	#APS	K	#APS × K
Phong shading	100	49	2	98	24	7	168
Gouraud shading	40	122	2	244	61	5	305
Gouraud shading and depth cueing	50	98	1	98	49	4	196
Constant shading	20	245	1	245	122	4	488

Table 7: Hardware requirements for maximum throughput.

6. Conclusions

We have shown that by exploiting massive parallelism and RISC technology, a multiprocessor Display Controller could be designed that is able to run the real-time refresh process from the lowest level frame representation we needed for interaction purposes. In doing so we are automatically assured of the maximum speed for interactions taking place at this lowest level. Since the frame representation at this level consists of a structured list of objects, it is both very compact as well as well suited for partial updates [1]. As a result, the access requirements of the frame store memory are reduced considerably. The real-time refresh process includes advanced scan conversion. Advanced in the extend that even complex shading methods such as Gouraud and Phong shading as well as periodic textures, multiple light sources and depth cueing are efficiently supported.

The Display Controller could be structured in such a way that scaling to both different display screen resolution as well as different image complexity can be done in a linear and independent way.

In spite of the massive parallelism, the number of chips needed for a 512 × 512 display will be below 100 with 1.2 micron technology.

As a result we obtained a powerful rendering device, which can be considered as our solution to the frame buffer access problem.

References

1. V. Akman, P.J.W. ten Hagen, and A.A.M. Kuijk, "A Vector-like Architecture for Raster Graphics," in *Advances in Graphics Hardware II*, ed. A.A.M. Kuijk, W. Strasser, EurographicSeminars, Springer-Verlag, 1988.
2. I. Carlbom, *System architecture for High-Performance Vector Graphics,* Dept. of Computer Science, Brown University, Providence, 1980. Ph.D. thesis
3. P.J.W. ten Hagen, A.A.M. Kuijk, and C.G. Trienekens, "Display architecture for VLSI-based graphics workstations," in *Advances in Graphics Hardware I*, ed. W. Strasser, EurographicSeminars, Springer-Verlag, 1987.
4. J.A.K.S. Jayasinghe, "Modeling and Performance Evaluation of a Real-Time Display Controller by Petri Nets," Report in preparation, University of Twente, Enschede, The Netherlands.
5. A.A.M. Kuijk, P.J.W. ten Hagen, and V. Akman, "An Exact Incremental Hidden Surface Algorithm," in *Advances in Graphics Hardware II*, ed. A.A.M. Kuijk, W. Strasser, EurographicSeminars, Springer-Verlag, 1988.

Appendix

This appendix will explain the use of incremental calculations for scan-conversion. In section A.1 we will have a look at the calculations needed to determine the geometry and in section A.2 we will discuss several calculations needed for pixel-colour evaluation.

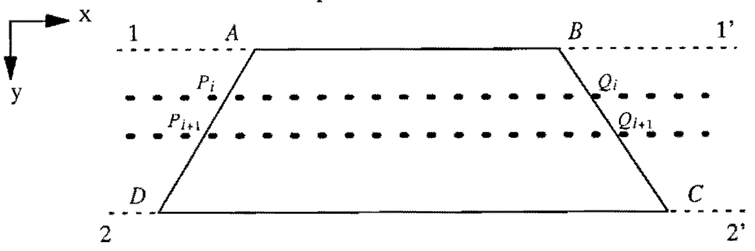


Figure A1: Scan conversion of an elementary pattern.

A.1 Scan-converting the Domains Using Incremental Calculations

Although domain representation can be used to describe any complex shapes such as concave shapes, partially filled shapes, etc, let us consider an elementary convex shape (Figure A1) for our discussion. The domain representation of the pattern ABCD contains two scanlines 11' and 22'. The scan-points A, B and gradients of AD and BC edges are associated with scanline 11'. Let edges AD and BC intersect the i^{th} and $i+1^{th}$ pixel-rows at P_i, Q_i and P_{i+1}, Q_{i+1} respectively. Let P_x, P_y denote the x and y coordinates of the point P. Then we can write:

$$P_{i+1_x} = P_{i_x} + 1/\mathrm{Grad(AD)}$$

$$Q_{i+1_x} = Q_{i_x} + 1/\mathrm{Grad(BC)}$$

If 1 / Grad(edge) is provided with the pattern, the intersections of the edges and the scanline can be calculated incrementally. Direct evaluation of the intersections would need multiplications, which are quite expensive in the sense of time and chip area. Incremental calculations on the other hand, require additions only. For the incremental calculations, fixed-point numbers can be used. A $2^m \times 2^n$ display needs, at least $\max(2m, 2n)$ bits for the variables.

A.2 Incremental Colour Evaluation

The realism of computer synthesized images can be enhanced by several techniques such as shading, texture generation, depth cueing, anti-aliasing etc. These techniques need a tremendous amount of calculations, hence real-time applications are difficult. Exploitation of coherency reduces the processing requirements and forthcoming sections present how this can be exploited for shading, texture generation, depth cueing and anti-aliasing.

A.2.1 Shading

In computer graphics, shading is the calculation of pixel intensities by estimating the contributions caused by different lightsources. These contributions depend on factors such as the colour of the light source, the relative positions, properties of the object, view direction etc. The contributions can be subdivided into three components, i.e. the ambient, diffuse and specular components. The ambient component is a general, lightsource independent term which depends on the object colour only. The diffuse component is a sum of terms, each of which depends on the distance, relative direction and colour of a particular lightsource and depends on the object surface colour and reflectivity. The specular component is somewhat similar, except that it is dependent on the view direction.

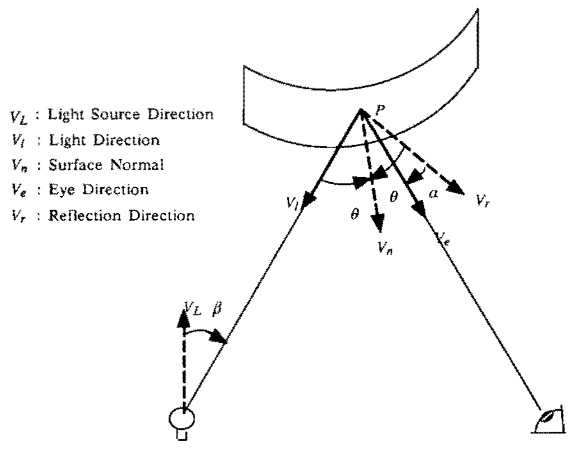


Figure A2: The vector and angle relations involved in shading calculations..

The equations 1 to 4 show the reflected light intensity mathematically. For an explanation of the vectors and angles which appear in these equations, please refer to Figure A2. Shininess of a surface is described by p, the specular exponent. This component is small for dull surfaces and can be as large as 100 for highly reflective (metallic-like) surfaces. The value of m determines the directivity of the light source.

$$I = I_a + \sum^{\text{lightsources}} (I_d + I_s) \tag{1}$$

$$I_a = K_a \tag{2}$$

$$I_d = K_d\,(V_l \cdot V_n)\,(V_L \cdot (-V_l))^m$$

$$I_d = K_d \cos\theta \cos^m\beta \qquad \text{for} \quad \pi/2 \geqslant |\theta|,\ |\beta| \geqslant 0 \tag{3}$$

$$I_s = K_s\,(V_r \cdot V_e)^p (V_L \cdot (-V_l))^m$$

$$I_s = K_s \cos^p\alpha \cos^m\beta \qquad \text{for} \quad \pi/2 \geqslant |\beta|,\ |\alpha| \geqslant 0 \tag{4}$$

In order to estimate the reflected light intensities, equation 1 has to be evaluated on all pixels covered by domains. Due to the vector dot product calculation and exponent calculations, real-time calculation of equation 1 would require a tremendous amount of hardware. For this, we approximate the term $\cos^p\theta$ piece-wisely by second order polynomials because second order polynomials can be evaluated iteratively using three additions per pixel location. Equation 5 can be derived by minimizing the absolute error.

$$\cos^p x = \begin{cases} 0 & \text{if} \quad |x| > x_0 \\ \dfrac{(|x| - x_0)^2}{x_0(x_0 - x_1)} & \text{if} \quad x_0 \leq |x| \leq x_1 \\ 1 - \dfrac{x^2}{x_0 x_1} & \text{if} \quad 0 \leq |x| < x_1 \end{cases} \tag{5}$$

where

$$x_0 = \frac{(p + 65.0)}{(5.0p + 31.7)}$$

$$x_1 = \frac{(p + 5.6)}{p(0.09p + 5.2)}$$

This approximation does not produce Mach-band effects because the approximated function and its first derivative have no discontinuities. The algorithm given below generates the approximated function iteratively.

```
X0 = (p + 64.0) / (5.0*p + 31.7) ,
X1 = (p + 5 6) / (p*(0.09*p + 5.2) ;

X = - Xmax,
DDI = 0,
DI = 0,
I = 0,

while (X < -X0) { X += 1, }
DDI = 2 / (X0*(X0 - X1));
while (X < -X1) { X += 1,
                              DI += DDI;
                              I += DI, }
DDI = -2 / (X0*X1),
while (X < -X1) { X += 1;
                              DI += DDI;
                              I += DI, }
DDI = 2 / (X0*(X0 - X1));
while (X < -X1) { X += 1,
                              DI += DDI,
                              I += DI, }
while (X <= Xmax)             ,
```

Note that only one comparison and three additions are needed per iteration step. This can be done using fixed-point numbers only. It can easily be proven that if at least $\max(2m, 2n)$ bits are provided for the fractional part, the use of fixed point numbers does not introduce quantization errors in the resulting intensity. Therefore a 1024×1024 display with 256 intensity levels needs at least 28 bits $(8+20)$ for the intensity data.

In the proposed architecture, Phong-like shading is evaluated by the scanline commands pumped into the X_IP array. The data for the scanline commands are calculated by Y_IP processors based on equation 5. The scanline commands are generated for each pixel-row, and the amount of calculations needed for the scanline command generation can be reduced by incrementally scaling the scanline command data.

A.2.2 Depth Cueing

Depth cueing is a technique to improve the realism of an image by fading the light intensities according to the distance between observer and surface. Figure A4 shows an appropriate depth cueing scale. The light reflected by the surfaces which are in front of the front depth plane and surfaces behind the back depth planes are scaled down by a constant factor and surfaces in between these two surfaces are scaled down by an interpolated factor.

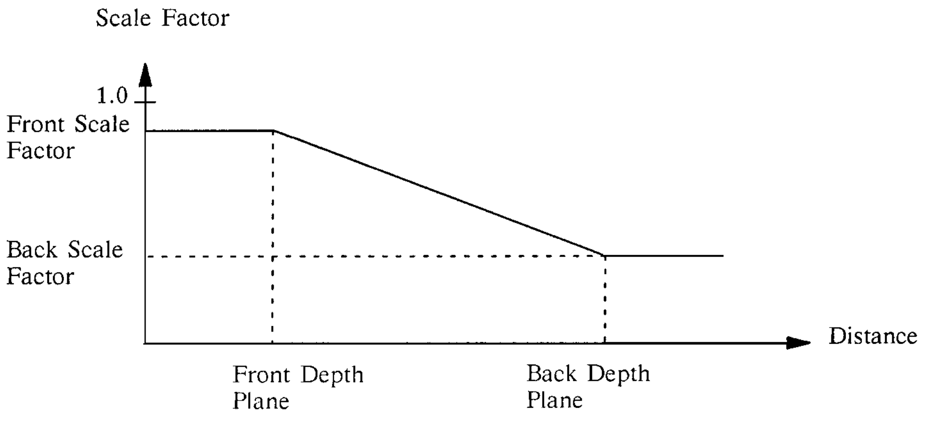
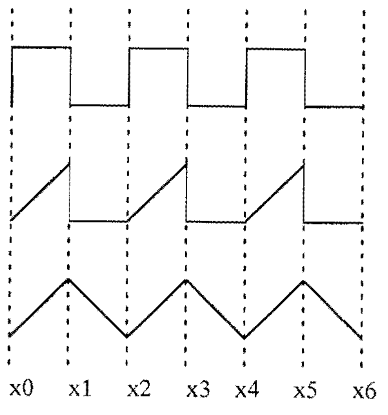


Figure A4: Depth cueing by scaling the intensity related to the view distance..

For exact results, the scaling would have to be performed at pixel-level, which would involve a lot of calculations. Therefore, we exploit the following simplified calculations. In case of constant shaded patterns, the edge intensities for each scanline are first scaled down according to the depth cueing scale factor. Given this, the intensities along the scanline can be linearly interpolated as in Gouraud shading to obtain the colours of the intermediate pixels. For Gouraud shaded patterns the same technique can be employed. Note that in both cases the intensity gradients are changed. In the case of Phong shaded patterns, one can get the appropriate effect by using different change over points (i.e. x_0, x_1) for negative and positive values of x. This has to be done carefully in order to avoid discontinuities in the intensity. In a forth coming paper we will describe Phong shading with depth cueing in more detail.



(a)

```
SETI(X0,2DX,I)
SETI(X1,2DX,0)
EVAL3(X0,6DX,0)

SETDI(X0,2DX,DI)
SETI(X1,2DX,0)
SETDI(X1,2DX,0)
EVAL2(X0,6DX,0)

SETDI(X0,2DX,DI)
SETDI(X1,2DX,-DI)
EVAL2(X0,6DX,0)
```

(b)

Figure A5: Periodic textures and the scanline commands to generate them.

A.2.3 Generation of Periodic Textures

According to section A.2.1, a complex shading method like Phong shading can be implemented using incremental calculations by changing the first and second derivatives of intensity at well defined pixel locations. Similarly, periodical textures can be generated using the same technique, in which case hardware designed for Phong shading can be used to generate periodical textured patterns. Figure A5(a) shows some simple patterns generated by incremental calculations just by altering intensity and first derivative and Figure A5(b) shows the scanline commands that go with it. Using this method, regular but fairly complicated textures can be generated quite efficiently.

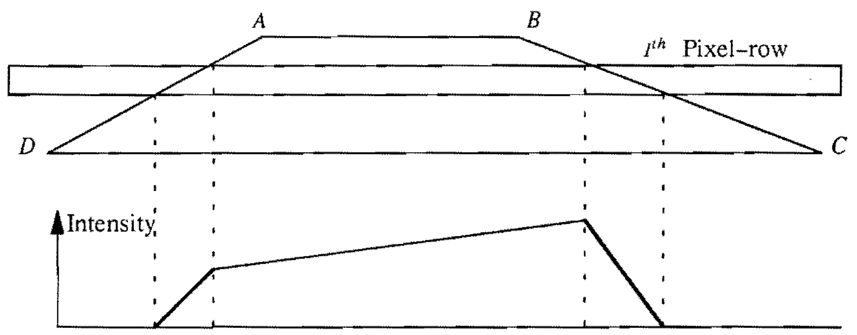


Figure A6: Anti-aliasing by modifying intensity function derivatives near the edges of a pattern.

A.2.4 Anti-aliasing the Pattern Edges

Anti-aliasing of pattern edges can be performed using incremental calculations by simply changing the derivatives of intensity. In this case the intensity gradients are changed for the pixels covered by pattern edges. Figure A6 shows an example of how this is done.

Due to the incremental methods we presented, the same hardware can be used for Phong shading, Gouraud shading, depth cueing, texture generation, and anti-aliasing of pattern edges. As the pixel colours are calculated in real-time, at least 100 MIPS processing power is needed for Phong shading on a display of 1024×1024 pixels at 50 Hz frame refresh rate. The processing power needed increases linearly with the number of light sources present. Therefore, the processing power of the scalable DC could be as high as several hundred (or even thousands) MIPS.