2 XInPosse: Structural Simulation for Graphics Hardware

M.A. Guravage
E.H. Blake
A.A.M. Kuijk

ABSTRACT A structural simulator is used both to test hardware and to visualize software that should run on that hardware. In a layered set of graphical hardware simulators, a structural simulator bridges the gap between hardware fidelity on the one side and sufficient performance to visualize graphics algorithms on the other. Essential design requirements were code extensibility and reusability. In order to achieve this, object-oriented methods were adopted. Important design criteria for graphical hardware simulators at this level are that both the exact digital state of the hardware and the graphical output be visualized interactively. The experience with using the XInPosse simulator is presented and analysed. XInPosse simulates a large systolic array in custom VLSI for second order interpolation; in this case to produce shaded scanlines. XInPosse provides the user with a means of tracing commands within the array while interactively setting breakpoints and displaying processors of particular interest. It verified that the hardware could execute the graphics algorithms correctly and that the limitations on numerical accuracy and range were graphically acceptable. An unexpected use was to facilitate communication between chip designers and the graphics researchers. Problems in the documentation of the hardware and workarounds for hardware "bugs" were found more easily through the common reference frame provided by the simulator. It is the intention of the authors to use the modularity provided by the object-oriented design to produce a toolkit for building graphical hardware simulators.

2.1 Introduction

Software simulation is a common if not compulsory part of hardware development. A simulator allows system designers to visualize the internal operation of the hardware interactively using both text and graphics. This paper discusses one such simulator for graphics hardware based on custom VLSI. We place equal emphasis on the need for a simulator, on its design, and on the lessons learnt from using it.

The research described here stems from a project whose aim is the development of a prototype graphics system based on *"A new technology for raster graphics on the basis of* VLSI[1]". In our system [3], a systolic array of FUSSOI[2] processors occupies the last place in the image-generation pipeline, that of rasterization and display. The preceding phases perform perspective projection, visible-surface calculation, and shading; the results of

[1]This study is partly funded by the Dutch Technology Foundation(STW) CWI79.1249
[2]Fast Universal Systolic Second Order Interpolator

which are quadratic shading functions [4, 1] which approximate Phong shading. Processors, allocated one per pixel along a scanline, interpolate these functions generating pixel intensities at screen refresh rate.

The XInPosse[3] simulator models this systolic array of processors at the functional level with fidelity and granularity sufficient to predict and evaluate their behaviour. Its graphical user interface allows one to initiate, control, and observe the execution of commands in the array. Individual processors can be displayed and their states observed whilst commands pass through them. An intensity graph for each scanline is drawn; along with a complete output image.

In the midst of designing a complete graphics system, we observed our occasional difficulty in coordinating and communicating our work. We will show that, apart from its utility for software and hardware designers, our simulator supported and facilitated communication among our disparate group of specialists.

This first section discusses the antecedents to the need for a structural level hardware visualizer. In the remaining sections we discuss: The major consideration which drove the design and implementation of the simulator, our experience in using it so far, concluding remarks, and an outline of further work.

2.2 The Need For Hardware Simulation

The simulator was developed in a project whose overall aim is to build a fast interactive graphical workstation aimed, in the first instance, at the Computer Aided Design market. A substantial part of the hardware is based on custom VLSI [3].

In developing custom (VLSI) hardware a number of simulators may be built; together they represent an integrated set of layered simulation tools. These include at least the following:

A. Electronic, or mask level, simulation. An analog simulation of the digital hardware.

B. Exact low level technology independent logic simulation. This is used by the chip designers.

C. Structurally and functionally accurate simulation. Registers, flags and logic are exactly simulated. This is the simulator discussed here.

D. Algorithmically accurate simulation. Need not use exactly the same data and control as the hardware. Used for fast software implementations to test algorithms.

The reason for using less accurate simulators at all is that accuracy is very expensive in terms of computing resources. Low level simulators (A and B) are, generally speaking, utterly impractical for testing software.

The algorithmic simulator (D) is used primarily to verify that algorithms will perform as expected in practice. It has sufficient speed to allow simple graphical results to be

[3] in posse, Latin. Literally "in potentiality."

visualized. The memory requirements are such that a complete hardware (sub-)system can be simulated on a workstation.

The structural level simulator (C) must have a sophisticated user interface to allow interactive inspection and debugging of executing programs. The simulator must allow visualization of important aspects of the modelled processor. An applications programmer must be able to verify that the output from a shading algorithm, say, will be accurate on the target hardware. A hardware specialist must be able to trace the data and commands as they pass through the simulated hardware and be able to interrogate the state of any part of the system at any stage.

It is this dual visualization that makes this simulator a useful adjunct in dealing with one of the problems which crop up in hardware design. Research in hardware based on custom VLSI requires close cooperation within a large team of specialists with very different backgrounds. Intensive and detailed interaction is needed because:

1. The hardware is closely coupled to a specific and very demanding application domain (i.e., graphics).

2. The ways in which the hardware can be realized are not fixed and there are a number of trade-offs between various sub-systems.

This means that a successful project is vitally dependent on dialogue between system and hardware designers. Once a VLSI design has been decided upon[4] The VLSI design team has to document their design. Unfortunately it may well be that these efforts are not effective in communicating with the higher level designers. The VLSI team is trained to communicate with peers and fail to understand the difficulties of non-specialists.

In this paper we will be considering simulators as an essential element in the communications between specialists. The low level simulators (A and B in the list above) are not suitable for this since they cannot execute realistic code and they are comprehensible only to hardware designers. The algorithmic simulators provide no information which a hardware designer can use. The structural level simulators are particularly useful in giving the system designers a concrete idea of what the hardware designers have implemented. On the basis of the simulation, questions can be formulated by the higher level users in terms which can readily be understood by the hardware designers.

2.3 Design Issues

Garzia [2] defines modelling, in very general terms, as a representation of an entity other than the entity itself. The primary requirement of our model was that it exhibit replicative and predictive validity on a functional basis. The nature of the components to be modelled also suggested a structural validity – that our software structures correspond to and interact as their counterparts in the modelled system [2]. This perspective lead quite naturally to an object-oriented approach.

[4]The process by which the design is finalized won't be discussed here. It is even more the result of intensive communications between graphics and interaction specialists and VLSI designers.

COMMANDS	DESCRIPTION
refresh	output the accumulated intensity
acc_mode()	*disable accumulation of negative numbers*
seti(x, i)	set intensity in one processor
setdi(x, di)	set first derivative in one processor
setddi(x, ddi)	set second derivative in one processor
dis(x, dx)	disable a range of processors
eval0(x, dx, i, di, ddi, dddi)	interpolate a cubic shading function
eval1(x, dx, i, di, ddi)	interpolate a quadratic shading function
eval2(x, dx, i, di)	interpolate a linear shading function
eval3(x, dx, i)	interpolate a constant shading function
eval4(x, dx, i)	interpolate a constant shading function
setpi(x, dx, i)	set intensity in processors x,x+dx, ...
setpdi(x, dx, di)	set first derivatives in processors x,x+dx, ...
setpddi(x, dx, ddi)	set second derivatives in processors x,x+dx, ...
nop()	no operation

TABLE 2.1. FUSSOI scanline commands

To demarcate the user interface and simulation functions logically and functionally, it was originally decided that each would occupy it own distinct UNIX[5] process. Unfortunately, for reasons described below, the asynchronous communication mechanism which connected the two processes proved too complicated and error prone for the volume of data processed. These problems were resolved, without compromising modularity, by merging the user interface and simulator processes into a single process.

2.4 Interaction

The simulator is logically and functionally split in two halves. The first half is an X-Windows program which manages the user interface – its graphics, event handling, and command parsing. The second half performs the simulation.

We concern ourselves in this section with the design of the user interface and begin with a description of the FUSSOI commands, the parser, and the graphical user interface.

The data that drive a simulation are commands which describe shading functions along scanlines; see Table 2.1. The arguments to the commands specify the shading function's initial values for intensity f(x), and optionally, its first, second, and third forward differences.

Forward differencing is an efficient method for evaluation of polynomials. A polynomial, e.g., $f(x)$, is specified using its initial value at the location $x = t$ and the first forward difference, $\Delta f(t) = f(t+1) - f(t)$. One then uses initial value at t, and adds the first

[5]UNIX is a trademark of AT&T Bell Laboratories.

forward difference to the original value to arrive at the value $f(t+1) = f(t) + \Delta f(t)$. The n-th forward difference is constant. For example, evaluation of the function $f(t) = t^3 + t^2 + t + 1$ over the range $t = 0$ to $t = 6$ is performed as follows: the first forward difference is $\Delta f(t) = ((t+1)^3 + (t+1)^2 + (t+1) + 1) - (t^3 + t^2 + t + 1) = 3t^2 + 5t + 3$, the second $\Delta^2 f(t) = (3(t+1)^2 + 5(t+1) + 3) - (3t^2 + 5t + 3) = 6t + 8$, and the third $\Delta f(t) = (6(t+1) + 8) - (6t + 8) = 6$. The initial values for the third-order forward difference are then $f(0) = 1$, $\Delta f(0) = 3$, $\Delta^2 f(0) = 8$, and $\Delta^3 f(0) = 6$. This method computes the values of the polynomial for each location in the range using n additions per location; see Fig. 2.1.

	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$
$\Delta^3 f(t)$	6	6	6	6	6	6	6
$\Delta^2 f(t)$	8	14	20	26	32	38	44
$\Delta f(t)$	3	11	25	45	71	103	141
$f(t)$	1	4	15	40	85	156	259

FIGURE 2.1. Forward differencing

Scanline commands are decomposed into subscanline commands; fulfilling the systolic array's requirement that commands complete execution in one systolic pulse; see Table 1.2. It is these subscanline commands and their data which are read by the simulator and are the basic unit of execution on the systolic array.

Subscanline commands propagate along the systolic array with every processor executing its instructions each systolic pulse. Fig. 2.2 shows the state of seven processors during seven cycles as they execute an eval2 command (linear shading function) followed by a refresh command. The x,dx command specifies that commands will be active in processors 2 through 4. The first occurrence of forward differencing is at t=4 when the di command reaches processor 2; The new di, next used in at t=5 in processor 3, is the sum of current di and ddi values.

The primary element of the user interface is the command parser which implements the code produced by the well-known UNIX lexical analyzer and parser generator LEX and YACC. The parser provides a "little language", with modest syntax checking, which parses scanline commands and their data, decomposing them into subscanline commands.

The graphical interface is an X-Windows program composed of widgets from the OSF-MOTIF widget set. Upon invocation, three separate panels appear: a control panel, a scanline panel, and an image panel; see Fig. 2.3.

The control panel has five components: a file selection widget which controls the parser,

COMMANDS	slot #1	slot #2	slot #3	slot #4	slot #5	slot #6
refresh	refresh	clear				
acc_mode()	(en)disable					
seti(x, i)	x	i				
setdi(x, di)	x	di				
setddi(x, ddi)	x	ddi				
dis(x, dx)	x,dx					
eval0(x, dx, i, di, ddi, dddi)	x,dx	dddi	(ddi)+dddi	(di)+ddi	(i)+di	acc
eval1(x, dx, i, di, ddi)	x,dx	ddi	(di)+ddi	(i)+di	acc	
eval2(x, dx, i, di)	x,dx	di	(i)+di	acc		
eval3(x, dx, i)	x,dx	i	acc			
eval4(x, dx, i)	x,dx,i					
setpi(x, dx, i)	x,dx	i				
setpdi(x, dx, di)	x,dx	di				
setpddi(x, dx, ddi)	x,dx	ddi				
nop()	no-op					

TABLE 2.2. FUSSOI subscanline commands

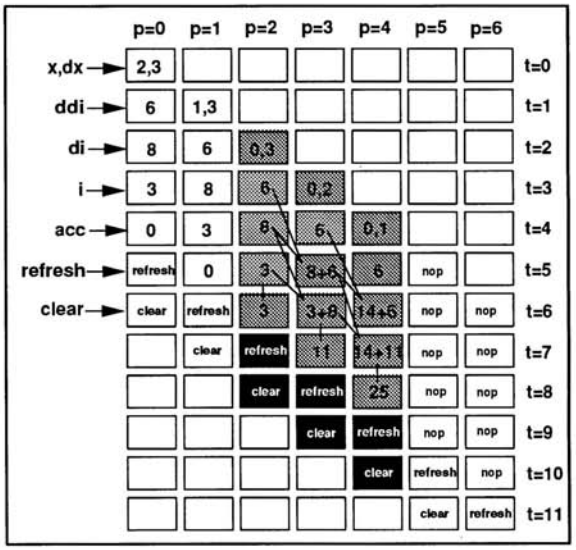

FIGURE 2.2. Systolic array

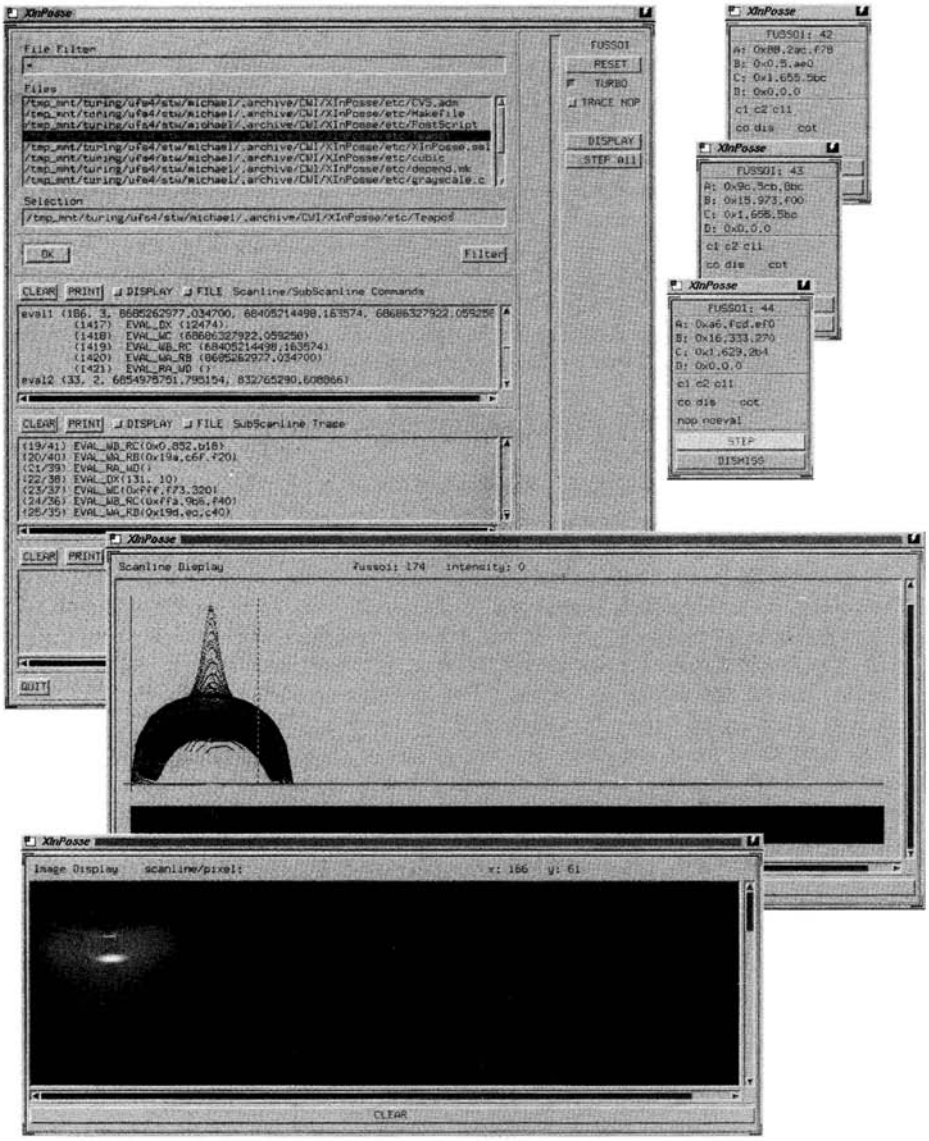

FIGURE 2.3. XInPosse user interface

a scanline widget which displays the scanline read by the parser and subscanline commands sent to the simulator, a subscanline widget which displays an execution trace for each command as it passes through the systolic array, a status widget which displays error and status messages, and lastly a slider widget which allows one to choose and pop-up individual processors.

The scanline intensity panel displays shading curves, where intensity is proportional to amplitude, and gray-scale values for the scanlines being processed over the array of processors.

The image panel displays a complete image constructed from all the scanline commands.

Optionally, individual array processors can be popped-up; displaying their contents and status. Breakpoints are automatically set which suspend execution of commands in these processors. Commands can then be single stepped individually or as a group. If pop-up processors are dismissed, breakpoints are removed and execution continues.

2.5 Object-Oriented Implementation

This section describes the models, written in C++, which implement the simulator. First, the class int36 which implements thirty-six bit arithmetic. Then the classes FUSSOI & OPCODE which implements the simulation proper.

The class int36 is the simulator's arithmetic class – actually a set of collaborating classes. The processors perform their interpolation to a precision of thirty-six bits. Though packages and classes for multiple precision and arbitrary precision arithmetic are available, we wanted a class optimized for our needs. Specifically, to provide thirty-six bits of precision, a double is used and shifted to use the thirty-six most significant bits of the fifty-two bit mantissa. Familiar operators are overloaded for the class.

The class OPCODE is the meta-class for the set of derived classes, each implementing the functions of one subscanline command. When an opcode and its data are read by the simulator, they are placed on an execution list (a circular link list), the entire list is traversed at each simulated systolic pulse and the opcodes applied to their destined processors on the systolic array. The data encapsulated in an instantiation of an class OPCODE are the necessary data passed from one processor to the next during execution.

The class FUSSOI implements all the functions of a processor, i.e., reading from and writing to registers and performing calculations. Here, structural simulation is most evident. The function of the processors is so modelled that the architecture of the class was coded directly from the hardware controller state diagrams. Each member function of the class corresponds to one subscanline command. An array of instances of class FUSSOI implements the systolic array.

2.6 Program Extensibility and Reusability

At first sight a simulator might seem a one-of-a-kind program written to simulate one particular processor. However, in an experimental environment the demands placed on the simulator, particularly in terms of what can be visualized and the ease with which

important aspects may be accessed, will change. Experience will tell what the important questions are which are put to the simulator. As confidence with some parts of the hardware increases, there is also increasing demand that more detailed simulation, on occasion, be dispensed with in order to speed up processing of the remainder.

A general problem with writing simulators is getting them finished in time to be useful in designing the hardware. The present simulator is a case in point. It will serve a number of useful purposes but influencing the design of the first chip is not one of them. Realizing this, it was decided to concentrate on making the various modules as reusable as possible. The other hardware simulators we need should be much quicker in arriving.

These requirements indicate a thoroughgoing modular approach. We opted for an object-oriented design. We hoped that the ability to make incremental changes to previously defined modules, via the inheritance mechanism common to object-oriented systems, would assist in software reuse and program extension.

Currently, the simulator generates gray-scale intensities only. Colour could be implemented by including colour data and corresponding methods in the OPCODE meta-class which would in turn be inherited by its sub-classes. With three modelled systolic arrays, each would calculate pixel intensities for the colours red, green and blue respectively.

2.7 Experience with Using the Simulator

The two primary tasks XInPosse was designed and written for are:

1. Verifying the functionality of processors, individually and arranged in a systolic array.

2. Testing of various shading algorithms which eventually will be implemented in silicon in a level directly above the systolic array.

Algorithmically, the simulator is being used to test routines implementing angular interpolated Phong shading as well as those for anti-aliasing via exact area integration. The simulator is the only medium, short of the hardware itself, which offers the exact numerical accuracy necessary to compute pixel intensities.

One of our first observations was that subscanline commands which set and evaluate intensities do so one processor, or pixel, too late – thus leaving pixel zero unreachable. This irregularity necessitated a change in the shading algorithms to compensate for the unexpected shift. The simulator therefore provided insight into the processors' architecture which compelled changes in higher level code.

Another observation was that, while processors can be instructed to ignore i.e. (clip) negative numbers, it was observed that highly negative numbers eventually wrapped around and become positive once again.

The components of the simulator were written to correspond directly to their hardware counterparts, Once the components were modelled, the processors' architecture could be coded directly from the defining hardware controller state diagrams. Straight away, the simulator exhibited unexpected behaviour which strongly suggested errors in the controller's documentation. The simulator allowed us to explicitly describe our observations,

ask the appropriate questions, and interpret the answers to resolve the errors. All were attributed to errors in the documentation.

Finally, in a sister project, a pair of researchers are writing a PostScript[6] interpreter which produces FUSSOI commands. They are using the simulator to test the results of their work both algorithmically and graphically.

2.8 Problems

Since work on the simulator was begun after work on the hardware, and the simulator took time to write, its contribution to the hardware (first generation at any rate) was minimal.

For all the benefits of our object-oriented design and implementation, the present version require workarounds to cope with incompatible libraries and the fact that useful UNIX tools (LEX & YACC) don't generate C++.

The user interface was written with Xt and the OSF-MOTIF widget set[7], both C libraries. The simulator, on the other hand, was written exclusively in C++. Difficulties arose compiling files which contained both X-Windows and C++ code. This necessitated liaison functions to couple the disparate routines.

As stated earlier, the use of UNIX pipes was central to our original design. To that end, all the familiar functions: reading, writing, duplicating file descriptors, closing unused ends of pipes, (un)buffered I/O, etc., were encapsulated in a class definition for interprocess communication. Alas, managing the traffic on loops of pipes among separate processes was non-trivial. Difficulties arose when the depth of the pipe, usually 4k bytes, was exceeded. Beyond that threshold, and despite careful attention paid to avoid deadlock situations [5], spurious errors regularly occurred. Pipes were eventually abandoned in favour of a traditional single process data coupled communication model.

A problem akin to the previous one is that UNIX interprocess communication is inherently non object-oriented. Using pipes to connect processes permits the communication of objects as C-structures where the receiving process has prior knowledge of the structures' types and sizes. Objects cannot retain their identities when passed over pipes. This fact precludes the use of run-time binding, the propriety of virtual functions and derived classes that bind objects to functions dynamically.

2.9 Conclusion

Structural simulation has proven to be a useful metaphor for writing hardware simulators. Its ability to show both high level functionality and low level architecture simultaneously expands the usefulness of one simulator to both system and hardware designers. The structural validity implicit in structural simulation necessitates and attests to the usefulness of object-oriented design.

---

[6] PostScript is a registered trademark of Adobe Systems.

[7] At the time XInPosse was written, the U. Lowell C++ MOTIF binding was not available.

By testing sequences of instructions rather than testing the functionality of individual hardware instructions, we have observed the relationships (and problems) which arise only among streams of commands.

For graphical application, the final judgement of the correctness of an algorithm can often best be made by observing the graphical output. This simulator offers the ability to generating complete images while remaining absolutely true to the hardware architecture.

The simulator has served as a liaison between software and hardware engineers. Initially, it has been used to formulate and specify question about the hardware before asked of the hardware designers; and to evaluate their answers. Already, a number of non-trivial ambiguities have been resolved this way.

2.10 Further Work

Because of our object-oriented approach, coupled with a graphical system such as X-Windows, we now have the beginnings of a hardware visualization toolkit. Certainly, our C++ classes are reusable; as are the user interface widgets. We plan to collect and formalize our class library and set of user interface widgets into a hardware visualization toolkit.

Though the actual hardware is an MIMD machine, the processors are simulated serially. The simulator's performance would benefit greatly from a parallel implementation of the systolic array of FUSSOI processors.

2.11 References

[1] V.C.J. Disselkoen. Real-time Quadratic Shading. Technical Report CS-R9123, CWI, 1991.

[2] M.R. Garzia. Discrete Event Simulation Methodologies and Formalisms. *Simulation Digest, Communications of the ACM*, 21(1):3–13, 1990.

[3] J.A.K.S. Jayasinghe, F. Moelaert El-Hadidy, G. Karagiannis, O. E. Herrmann, and J. Smit. Two-Level Pipelined Systolic Array Graphics Engine. *IEEE Journal of Solid-State Circuits*, 26(3):229–236, March 1991.

[4] A.A.M. Kuijk and E.H. Blake. Faster Phong Shading via Angular Interpolation. *Computer Graphics Forum*, 8(4):315–324, 1988. Please note that on p. 321 the definitions of a and b should be swapped.

[5] J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *4.3 BSD UNIX Operating System.* Addison-Wesley Publishing Company, Reading, MA., 1988.