5 The I.M.O.G.E.N.E. Machine: Some Hardware Elements

V. Lefévère
S. Karpf
C. Chaillou
M. Mériaux

ABSTRACT The goal of the I.M.O.G.E.N.E. project is to define a real time graphics system. We focus on true real time display, images being computed at frame rate, i.e 50 (or 60) times a second. The I.M.O.G.E.N.E. machine uses no frame buffer. We use a massive object parallelism; the graphics module is made of a large number of object-processors, each one handling one graphics primitive at pixel rate in raster-scan order. Shading computations are made in a deferred shading processor using Phong's method. After a brief presentation of Object-Oriented Architectures,we present new details about the hardware implementation of our Object Processors, and describe for the first time the shading processor.

5.1 Introduction

Many graphics systems with increasing performances have been proposed over the past several years. Current systems now offer performances of up to 1,000,000 small 3D triangles/second. These systems use parallelism in the scan-conversion processors as well as in the host computer. As far as scan-conversion is concerned, two kinds of parallelism are used: pixel parallelism (scan-conversion processors are associated to the pixels) and object parallelism (scan-conversionprocessors are associated to the objects). In both cases, current VLSI technology allows to build massively parallel systems.

Increasing scan-conversion performances involve increasing front-end computation power. Indeed insufficient front-end power may drastically reduce the performances of the whole system. The solution generally proposed consists of using a parallel host computer. The system proposed by Raster Technologies [13] uses 8 processors working in parallel. Up to date commercial systems use several RISC processors (4 in the Apollo [9], up to 8 in the Silicon Graphics [1]). The Pixel-Planes 5 team [7] intends to use up to 32 i860. Connecting these processors to the pixel-oriented graphics modules requires high speed links: the Silicon Graphics has two 64 bit buses (64 MByte/s). The StellarGS1000 [2] uses 512 bit datapath (20 Mhz). The full scale prototype of Pixel-Planes 5 will use eight 32 bit rings (20 Mhz each).

However, strong communication bottlenecks might appear between the host processors and the graphics module, and make performances rapidly decrease. Indeed the database stored in the host computer has obviously an object parallel structure. Therefore connecting an object-parallel host to a pixel-parallel scan-conversion system creates a bottleneck, because any host processor must be connected to any scan-conversion processor. We believe that the communications between the host and the graphics module will be the main bottleneck of future pixel-oriented graphics systems.

The object approach solves the problem of communication bottlenecks between the (parallel) host computer and the massively parallel graphics module. Indeed, since objects are independent, each host processor needs only to be connected to the Object Processors allocated to it.

The main recently proposed object-oriented systems are the Weinberg system [14], GSP-NVS [6] and PROOF [11]. All these systems implement a pipeline of Object Processors, each processor handling one graphics primitive (3D triangle) in raster-scan order. Hidden part elimination is achieved through the pipeline: each Processor receives the depth of its left neighbor, compares it with its own depth and transmits the visible object to its right neighbor. GSP-NVS and PROOF both propose a shading processor (Phong method) connected to the end of the rendering pipeline. PROOF and Weinberg also propose a solution for generating high quality anti-aliased images.

The performances of these systems depend on the number of Object Processors. Associating one object to one processor (in a given frame) seems quite inefficient, especially when displaying small objects. GSP-NVS proposes to allocate the processors at the beginning of every scan-line. However, this requires to sort the objects with regard to their Y coordinate, which may be difficult to achieve in real time.

The main advantages of object-oriented architectures are:

- no communication bottleneck
- no frame buffer is required
- highly modular architectures (the system can be expanded by adding Object Processors)
- True real time display (with a limited number of objects)

Of course, the performances of these systems depends on the number of Object Processors. However, Object Oriented architectures are very well suited for applications that do require real time performances. We will now present the solution we propose for a true real time graphics system. The I.M.O.G.E.N.E. system [4] is a real-time object-oriented graphics system (see Figure 5.1). It is composed of a large number of object-processors, each one handling one graphics primitive all over the screen in raster-scan order. Each Object Processor computes the depth and the normal vector of its own primitive. Hidden part elimination is achieved in the Decision Unit by means of a distributed Zbuffer (the Decision Unit can be either a pipeline or a pipelined binary tree). Then a deferred shading processor computes the shading (Phong's method) for all the pixels in the image.

The basic primitives handled by the system are classical 3D triangles and quadrics. We have presented these primitives in [8]. We will present in this paper the current status of the system, especially the hardware description of the Elementary Processors and of the shading processor.

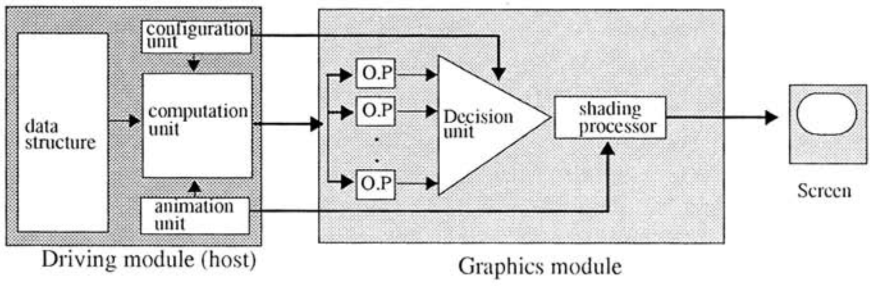

FIGURE 5.1. Architecture of I.M.O.G.E.N.E.

5.2 Some Hardware Considerations

5.2.1 Introduction

The Object Processors are built from two basic structures: the first order Elementary Processor EP1 computing incrementally at pixel rate the linear expression $Ax + By + C$, and the second order Elementary Processor EP2 computing incrementally at pixel rate the quadratic expression $Ax^2 + By^2 + Cxy + Dx + Ey + F$. A square root extractor is also needed for building quadrics [8].The EP1 and EP2 work at pixel rate in raster-scan order. We are building a 512×512, 50Hz prototype, so they have to work at 16 MHz (about 60 ns). We will see in the following how all the operators (mainly adders) succeed in sustaining that rate.

5.2.2 The Elementary Processors

We will now describe the First Order Elementary Processor. It computes at pixel rate in raster scan order the expression $Ax + By + C$, (x, y) being the pixel coordinates. $By + C$ is computed at the beginning of every screen line during horizontal retrace. Then $Ax + By + C$ is computed for every pixel of the line. The same structure can be used for both computations since A, B, and $By + C$ are saved in a backup RAM. The EP1 uses 24-bit words. However, the 24-bit adders available in our VLSI CAD software are to slow to sustain the screen rate. Therefore, the adder in the EP1 is pipelined and divided into two 12-bit adders. A 12-bit register is then required to synchronize the 12 LSB and the 12 MSB (see Figure 5.2). Coefficients A, B and C are computed by the host computer at frame rate according to the required animation. They are stored in a loading RAM, and transferred in the backup RAM during vertical retrace (this transfer is achieved simultaneously in parallel by all the Elementary Processors).

The microprogram controlling the EP1 is given (clock1 is the signal controlling the LSB register and the synchronization register, clock2 the signal controlling the MSB register and the carry flip-flop). The three coefficients of the frame to be displayed are assumed to be stored in the loading RAM.

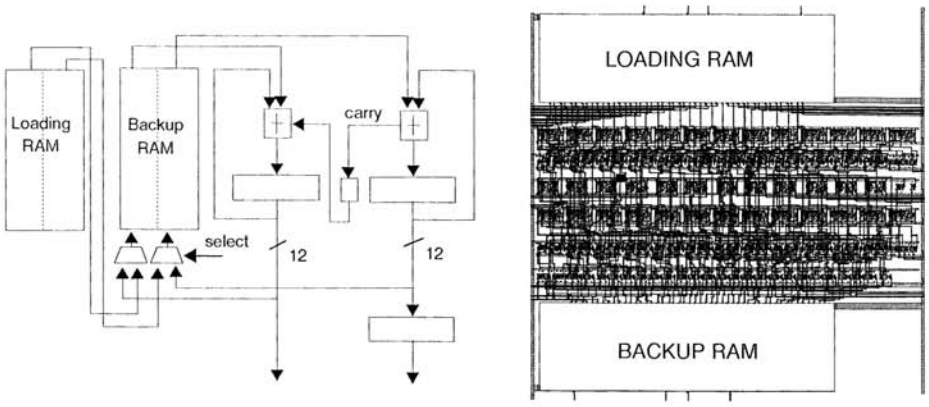

FIGURE 5.2. The first order elementary processor

```
At the beginning of every frame do
    Select = 1 {the backup RAM is connected to the loading RAM}
    Read_loadingRAM_adress0
    Write_backupRAM_adress0
    Read_loadingRAM_adress1
    Write_backupRAM_adress1
    Read_loadingRAM_adress2
    Write_backupRAM_adress2
    Select = 0 {the backup RAM is connected to the registers}
    At the beginning of every scan-line do
        Clear_all_registers
        Read_backupRAM_adress2 {reads old value of $BY + C$}
        Clock1, Clock2
        Read_backupRAM_adress1 {reads $B$}
        Clock1
        Clock2 {new value of $By + C$ is computed}
        Write_backupRAM_adress2 {$By + C$ is stored}
        Read_backupRAM_adress0 {reads $A$}
        Clock1
        For each pixel of the scan-line do
            Clock1, Clock2 {computes $Ax + By + C$}
```

5.2.3 The Object Processors

Two Object Processors have been studied: the Triangle Processor and the Quadric Processor. We focus here on the design of a complete Triangle Processor.

Each object is defined in every pixel by its depth, its surface normal vector and its color (or matter). The Triangle Processor is composed of seven EP1s (three to define the triangle border, one to define the depth and three to define the surface normal vector), one interface unit with the host processor, one Z-buffer unit and one pre-normalization unit.

The pre-normalization unit

Each surface normal vector component (N_x, N_y and N_z) is computed by one EP1, and thus is a 24-bit word. However, we wanted to reduce the pin count of the Triangle Procesor, and our simulations have shown that 12 bits/component are suited for an accurate shading. Thus the pre-normalization unit transforms each component into a 12-bit word. In fact it divides each component by 2^K. It first transforms N_x, N_y and N_z into $|N_x| + sign\ of\ N_x$, $|N_y| + sign\ of\ N_y$ and $|N_z| + sign\ of\ N_z$. This transformation is achieved with a XOR operator, so that a (negligible) loss of accuracy occurs (we assume that $-A = \overline{A}$ instead of $-A = \overline{A} + 1$). Then the highest component determines the value of K and controls an adaptive shifter. This shifter consists of four fixed shifters (1, 2, 4 and 8-bit shifter) that can be either passed through or bypassed. Figure 5.3 shows the pre-normalizaion unit (only the N_x component is shown).

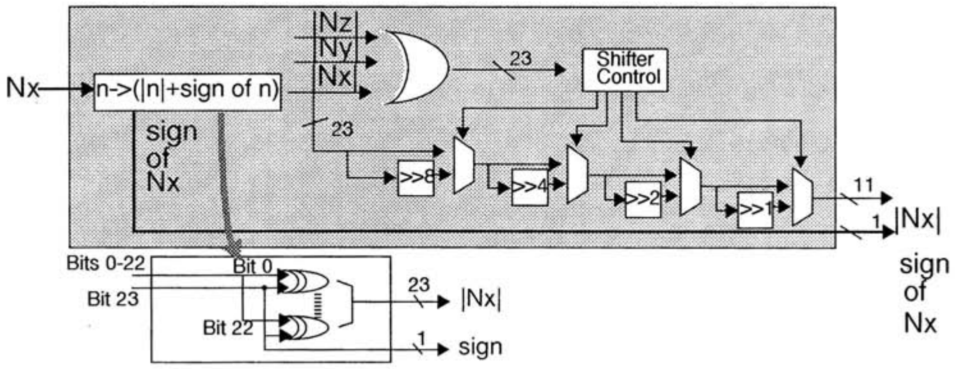

FIGURE 5.3. The Pre-Normalization Unit

The Z-buffer unit

The Triangle Processor that we are designing will be able to work either in a pipeline or in a binary tree. Therefore it includes a Z-buffer operator. This operator will be deactivated when used in a binary tree. When used in a pipeline, each Triangle Processor receives the depth of its left neighbor, compares it with its own depth, and transmit the closest to the screen reference to its right neighbor. The depth and the three components of the surface normal vector are computed by Epls, the color (or matter) is constant and stored in a register.

The Interface unit

This unit is connected to the Host bus. It receives coefficients from the host and dispatches them to the EPls. Each coefficient has a specific address indicating the number of the Triangle Processor, the number of the Elementary Processor and the address in the loading RAM. This unit will be implemented in a PLA.

Figure 5.4 shows the functional scheme of the Triangle Processor. Let us note that the actual design is a bit more complex, since any operator that cannot sustain the system clock rate must be split into several pipelined operators (e.g. the Z-buffer comparator). All the I/O pins are double clocked in order to reduce the pin count. The VLSI design of this chip is almost complete

5.2.4 Design of a Complete System

We plan to use the INMOS T800 Transputer (or the new T9000, named H1 for the moment) as a basic host processor. The animation unit is implemented into one Transputer. It receives animation data from the user, and broadcasts them to the Computation Units. Each computation unit is implemented in one Transputer, and handles a fixed number of Object Processors (this number has still to be determined). It calculates the coefficients of all its Elementary Processors according to the required animation. The coefficients of the next frame to be displayed are computed and stored in the Elementary Processors loading RAMs (connected to the Transputer bus) during display time. Then they are transferred during vertical retrace into the Elementary Processors backup RAMs. The main advantages of this solution are:

- Object Processors are seen by the Transputer as addresses in its own memory.
- Due to the use of a loading RAM, the host processor has only to modify the coefficients of the objects that have to be animated. Coefficients of fixed objects remain unchanged in the loading RAM, but will also be transferred during vertical retrace.
- The four external links available on the Transputer (about 2 Mbyte/s in the T800, 20 Mbytes/s in the T9000) enable to easily expand the system (see Figure 5.5)

5.3 The Shading Processor

5.3.1 Illumination Models

Phong presented the first complete illumination model by using the reflection vector [10].

$$I = K_a I_a + \sum_i (K_d \cdot I_i \cdot (\vec{N} \cdot \vec{L}) + K_s \cdot I_i \cdot (\vec{R} \cdot \vec{O})^E)$$

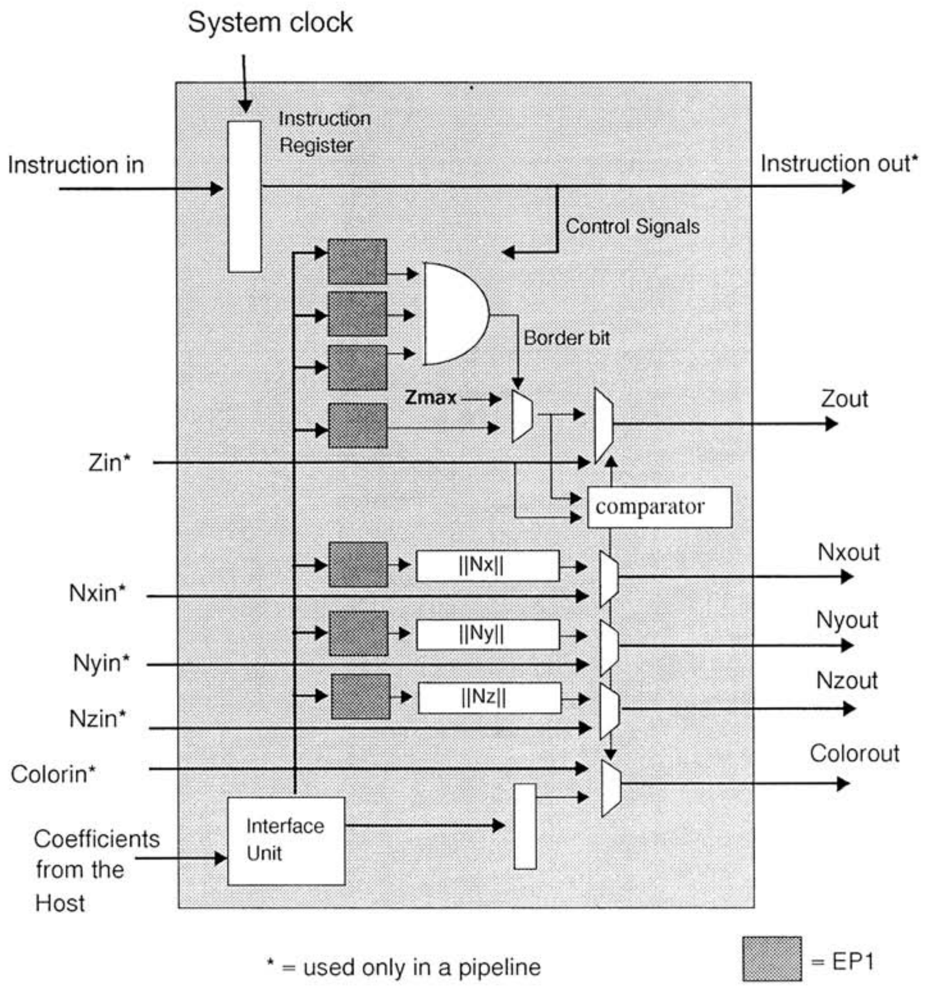

FIGURE 5.4. The triangle processor

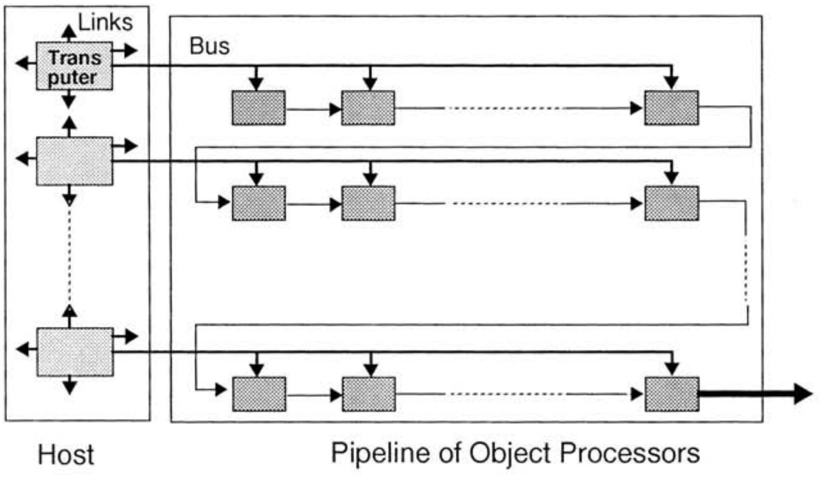

FIGURE 5.5. A complete system

Blinn modified this model [3, 5] using the highlight vector $\vec{H}$. The illumination formula is the following (see Figure 5.6):

$$I = K_a I_a + \sum_i (K_d \cdot I_i \cdot (\vec{N} \cdot \vec{L}) + K_s \cdot I_i \cdot (\vec{N} \cdot \vec{H})^E)$$

with

$\vec{N}$	the surface normal vector	$\vec{R}$	the reflection vector
$\vec{L}$	the light source vector	$\vec{O}$	the eye vector
K_a	the surface ambient coefficient	$\vec{H}$	the highlight vector
I_a	the ambient intensity	K_s	the surface specular coefficient
K_d	the surface diffuse coefficient	E	the surface specular power
I_i	the intensity of source i		

5.3.2 Deferred Shading

To defer the shading computations at the end of the visualization pipe-line (next to the hidden part elimination) has the following advantages:

- Shading is computed only for the visible pixels of the final image.
- Any object can be shaded provided that its normal vector can be computed at every pixel.
- The shading computations of all the pixels are independent (thus a massive parallelism can be used).

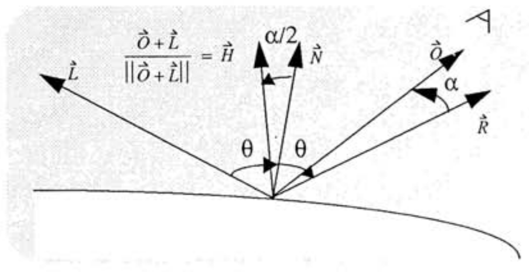

FIGURE 5.6. Vectors

The main drawback is obviously the large amount of computations. Hardware implementations of a deferred shading processor have been proposed for GSP-NVS [6] and for PROOF [5, 11]. The Pixel-Planes team [12] has developed a deferred Phong shading software for their 128x128 pixel-processors renderer.

5.3.3 First Choices

Our aim is to build a low-cost deferred shading processor. Therefore we will assume the following simplifications:

- In order to reduce the complexity of the shading operators, the surface normal vector components will be truncated into 12-bit words (11 bits + 1 sign bit). This provides an accurate diffuse shading (assuming three 8-bit DACs for Red, Green, Blue, i.e. 16 million colors) as we can understand by examining the following formula: $\delta I_D \leq \sqrt{3} I K_D \times \delta N \leq \sqrt{3} \times \delta N$, with δN the error on the diffuse shading and δN the error on the normal. As far as the specular shading is concerned, the bit count depends on the specular power. However our simulations have shown that 12-bit words allow to render usual materials.
- We will implement infinite distance point light sources (like all current hardware shading processors) with infinite distance viewer. Thus the light source vector $\vec{L}$ and the highlight vector $\vec{H}$ are constant for all the pixels in a given frame. This explains why we have chosen Blinn's model instead of Phong's one.
- We will try to use commercial components.

5.3.4 Our Proposal

Figure 5.7 shows the general scheme of the lighting equation with one light source, considering only the diffuse and specular components.

Since $\vec{L}$ and $\vec{H}$ are constant, the dot products computations can be simplified using multiplication Look Up Tables (Low size L.U.Ts can be used thanks to the reduced bit count of the incoming normal vector). The other multiplications can also be stored in

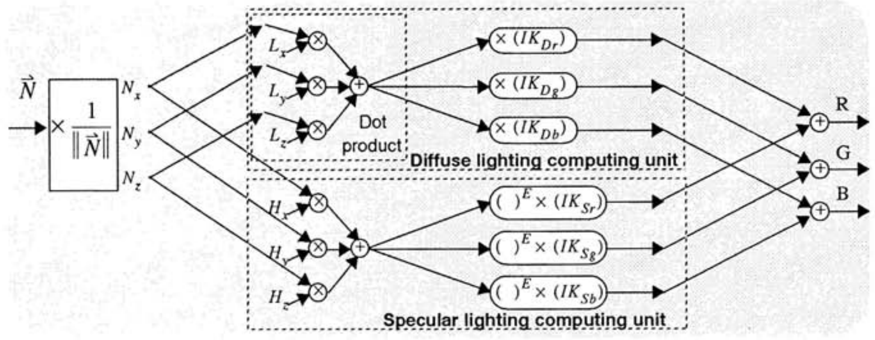
$\vec{N}$
$\times \frac{1}{\|\vec{N}\|}$
N_x
N_y
N_z
L_x
L_y
L_z
Dot product
$\times (IK_{Dr})$
$\times (IK_{Dg})$
$\times (IK_{Db})$
Diffuse lighting computing unit
H_x
H_y
H_z
$(\ )^E \times (IK_{Sr})$
$(\ )^E \times (IK_{Sg})$
$(\ )^E \times (IK_{Sb})$
Specular lighting computing unit
R
G
B

FIGURE 5.7. Scheme of the lighting equation

L.U.Ts. The architecture of the shading processor (assuming one light source and only 16 matters) is given in Figure 5.8.

To reduce by half the size of the dot product L.U.Ts, only the product of the absolute value are calculated by them, a XOR computes the sign of each result. This assumes the adder terminating the computation to deal with numbers coded with their absolute values and distinct signs. The complexity of this sort of adder is about the same as the one using the 2's complement form. We know that $-A = -\overline{A} + 1$, so some XOR could be used to make the conversion in the 2's complement form before being really added. Adding three numbers we may have an error of three LSB in the worse case, if we forget the correction between $-A$ and $\overline{A}$. Simulations of the architecture in OCCAM, on a transputer-based network, show this error is not important and is included in the other one we have already accepted to have. To assume the fact that only the surface seen by the light are lighted, the negative results of the dot product become the zero value for the next stage of the shading computation.

The L.U.Ts can be incrementally computed and loaded either during the previous frame by a classical microprocessor with at least a 24-bit ALU, or during vertical retrace by a wired sequencer.

5.3.5 The Incremental Algorithm

To fill the dot product L.U.T or the diffuse L.U.T, we start by putting 0 at the first address of the RAM and in the error register. We compute each following value by adding one of the numbers (Lx, Ly, ...) depending on the table, to the error register. At each overflowing of this error, we increase by one the storing value. This way of doing looks like the Bresenham algorithm for drawing line. Once the first three tables that are shown in Figure 5.9 could easily be computed, a less obvious solution had to be found to fill incrementally the last tables.

The exact algorithm depends on the hardware implementation of the control and loading unit. Some little modification may be done for each version, to reduce the hardware or software complexity.

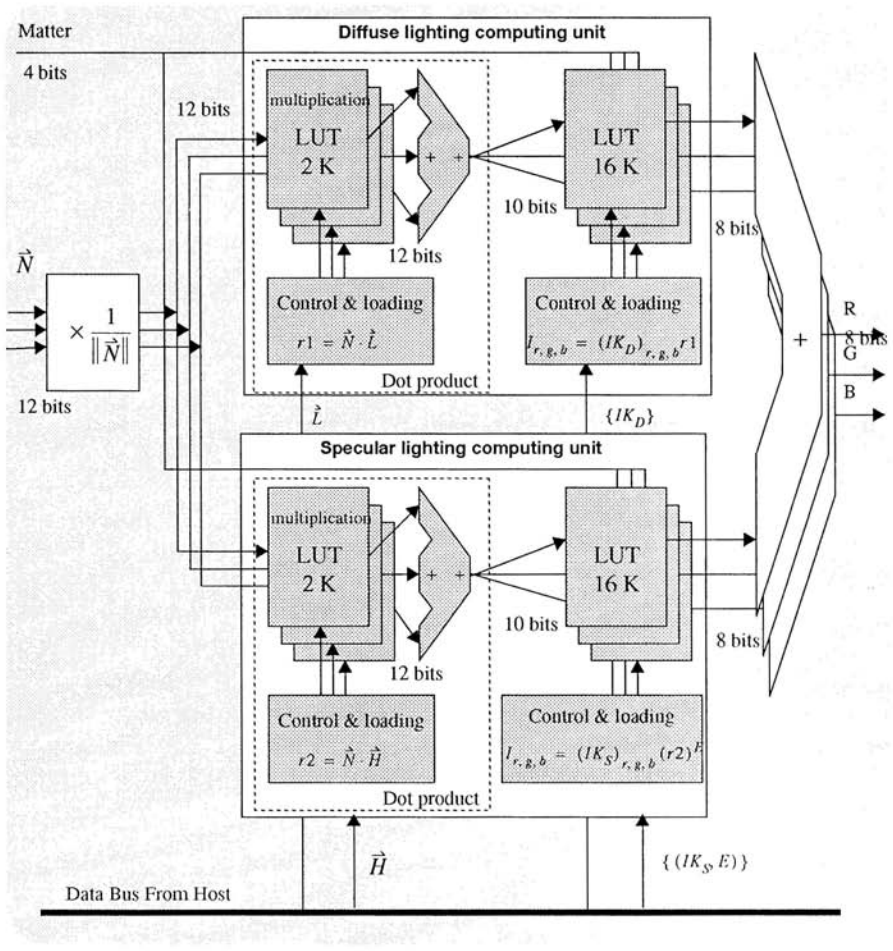

FIGURE 5.8. Hardware implementation

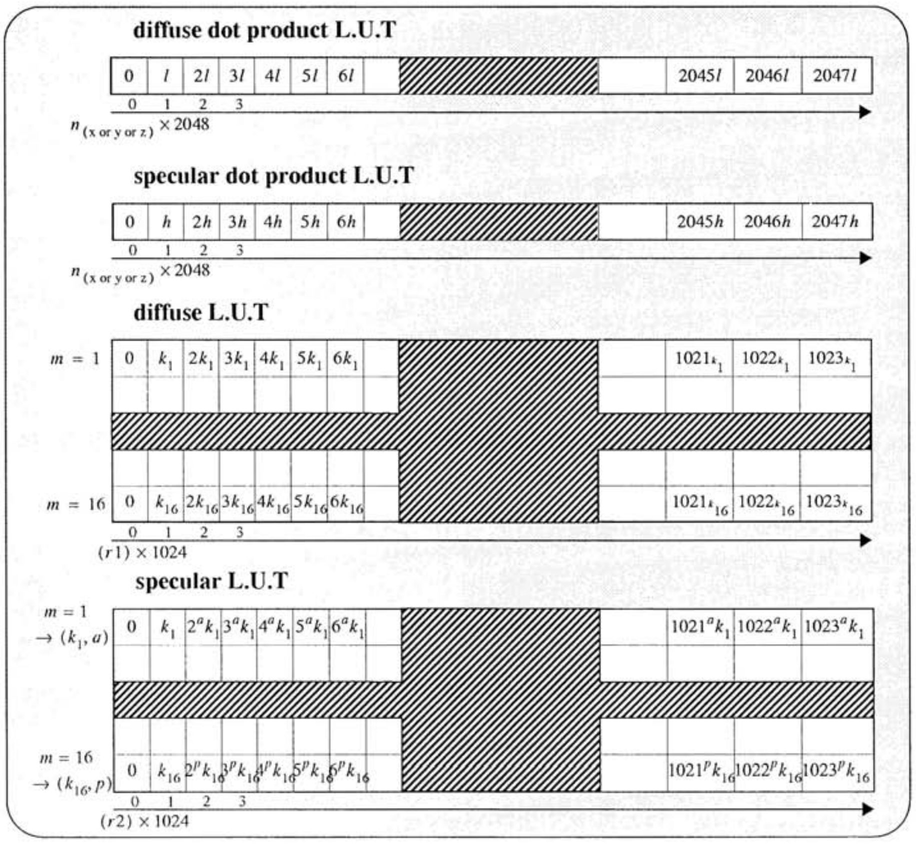
diffuse dot product L.U.T
0
l
$2l$
$3l$
$4l$
$5l$
$6l$
$2045l$
$2046l$
$2047l$
0 1 2 3
$n_{(x \text{ or } y \text{ or } z)} \times 2048$
specular dot product L.U.T
0
h
$2h$
$3h$
$4h$
$5h$
$6h$
$2045h$
$2046h$
$2047h$
0 1 2 3
$n_{(x \text{ or } y \text{ or } z)} \times 2048$
diffuse L.U.T
$m = 1$
0 k_1 $2k_1$ $3k_1$ $4k_1$ $5k_1$ $6k_1$ $1021k_1$ $1022k_1$ $1023k_1$
$m = 16$
0 k_{16} $2k_{16}$ $3k_{16}$ $4k_{16}$ $5k_{16}$ $6k_{16}$ $1021k_{16}$ $1022k_{16}$ $1023k_{16}$
0 1 2 3
$(r1) \times 1024$
specular L.U.T
$m = 1 \rightarrow (k_1, a)$
0 k_1 $2^a k_1$ $3^a k_1$ $4^a k_1$ $5^a k_1$ $6^a k_1$ $1021^a k_1$ $1022^a k_1$ $1023^a k_1$
$m = 16 \rightarrow (k_{16}, p)$
0 k_{16} $2^p k_{16}$ $3^p k_{16}$ $4^p k_{16}$ $5^p k_{16}$ $6^p k_{16}$ $1021^p k_{16}$ $1022^p k_{16}$ $1023^p k_{16}$
0 1 2 3
$(r2) \times 1024$

FIGURE 5.9. The contents of the L.U.Ts

The main difficulty is produced by the power function that could not be easily and incrementally calculated. Moreover, after having the result, we had to multiply it by the current value of K_s. To solve that, we have chosen to put in a ROM the information to produce incrementally each chosen power function. Assuming that the surface specular power may not be too high to have a correct picture, the number of power function that we have to put in the ROM is small.

Figure 5.10 shows us an example of a power functions drawn for a system that has 16 states to code the input numbers and 8 states to code the output numbers.

As, in fact, we want to build an algorithm which could compute incrementally the content of the specular LUT, we do not store the value of $f(x)$ for each x in ROM. After having computed the output state of the function for each input state, we code, and put in the ROM, the move shown by the arrow on Figure 5.10. All the power functions have

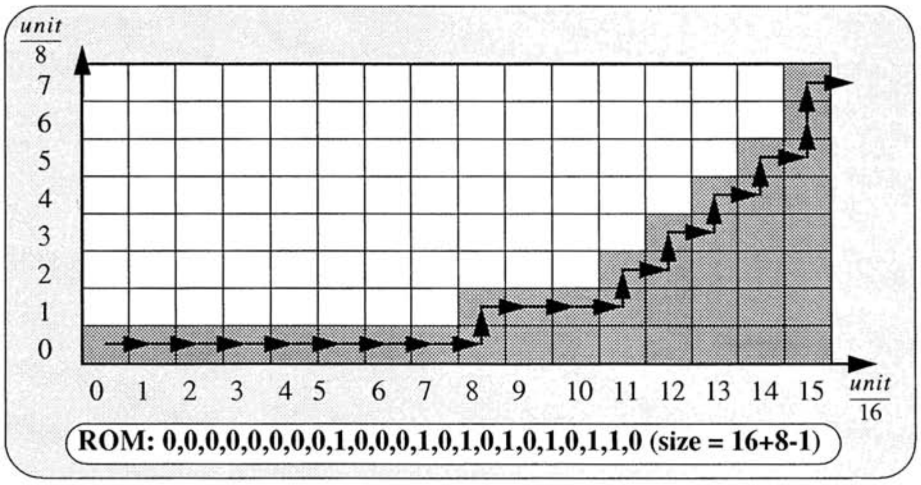
unit/8
0
1
2
3
4
5
6
7
8
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
unit/16
ROM: 0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,0,1,0,1,0,1,1,0 (size = 16+8-1)

FIGURE 5.10. The incremental drawing of the function $f(x) = x^4$

the same property:

$$f(0) = 0 \textit{ and } f(1) = 1$$

So the number of moves does not depend on the power function but only on the number of input and output states. We obtain the following expression:

$$length(ROM) = number\ of\ input\ states + number\ of\ output\ states - 1$$

In fact, we make a small error by assuming that 1 is equal to the final state (in our example, for the input states the final one is $15 \times \frac{unit}{16}$: and for the output one $7 \times \frac{unit}{8}$)

To compute the finally used function, including the product by K_s, we redraw the shape of the power function using a scaling on the output state. Figure 5.11 shows the execution of the scaling in the case of our previous example.

The time taken for filling the table depends on the length of the ROM. At each cycle, the control unit had to read the move in the ROM. For the horizontal move, the input state, indeed the address in the L.U.T, is increased by one. For the vertical move, the content of an error register is increased by the value of K_s and we increase by one the stored value when the error overflows, as we do for filling the other table. The number of states of the error may be at least the same as the number of output states, and the greater value of K_s is coded as the final state. After the execution, we obtain the function required as shown in Figure 5.12 for the case of our example.

The use of some diagonal moves is not interesting because then the time of computation may change depending on the surface specular power.

Cycle	In	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
prev L.U.T adresse	?	0	1	2	3	4	5	6	7	8	8	9	10	11	11	12	12	13	13	43	14	15	15	15
prev Error ×1/8	?	0	0	0	0	0	0	0	0	6	6	6	6	4	4	2	2	0	0	6	6	4	2	2
previous Value	?	0	0	0	0	0	0	0	0	0	0	0	0	1	1	2	2	3	3	3	3	4	5	5
Code (ROM)		0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	1	0	1	0	1	1	0
Stroring		▨	▨	▨	▨	▨	▨	▨	▨		▨	▨	▨		▨		▨		▨		▨			▨
next L.U.T adresse	0	1	2	3	4	5	6	7	8	8	9	10	11	11	12	12	13	13	14	14	15	15	15	0
next Error ×1/8	0	0	0	0	0	0	0	0	6	6	6	6	4	4	2	2	0	0	6	6	4	2	2	0
next Value	0	0	0	0	0	0	0	0	0	0	0	0	1	1	2	2	3	3	3	3	4	5	5	6
Move		R	R	R	R	R	R	R	R	N	R	R	R	U	R	U	R	U	R	N	R	U	U	R

The abbreviations of the move : Right , None , Up

FIGURE 5.11. Example of computing and storing the function $f(x) = \frac{6}{8} \times x^4$

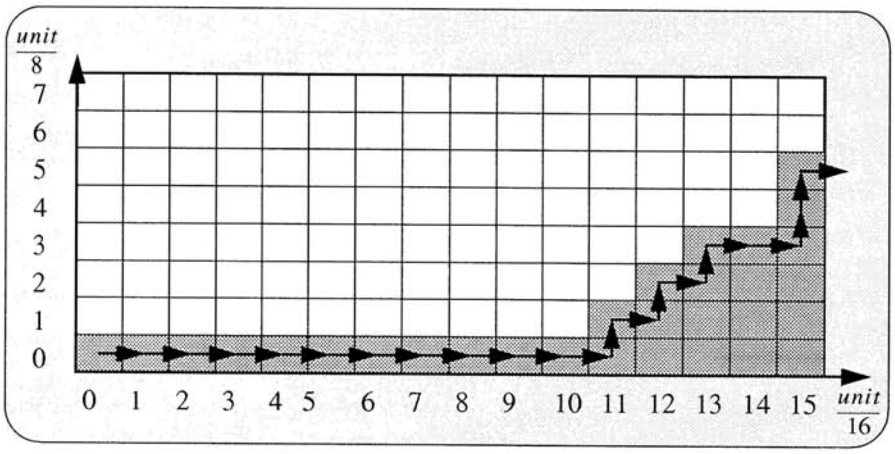

FIGURE 5.12. The final drawing of the function $f(x) = \frac{6}{8} \times x^4$

5.3.6 The Control and Loading Implementation

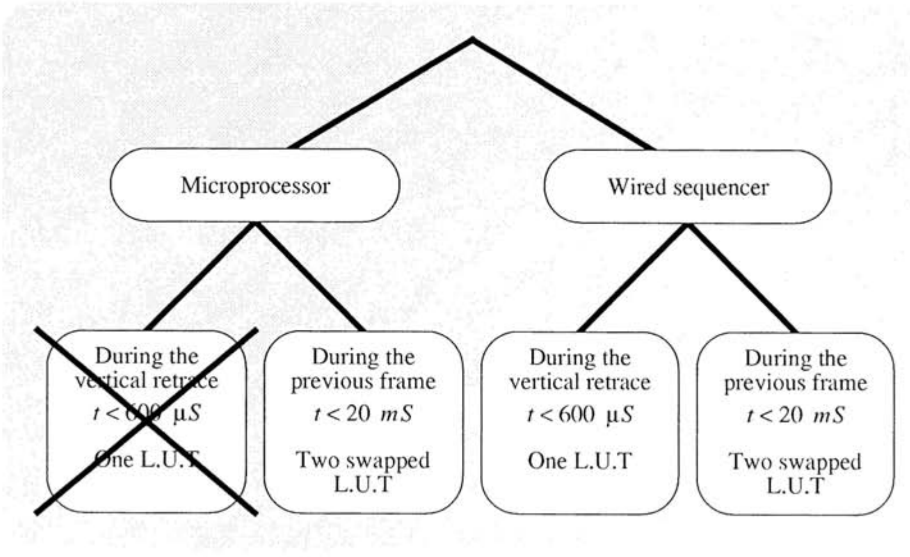
Microprocessor
Wired sequencer
During the vertical retrace $t < 600\ \mu S$ One L.U.T
During the previous frame $t < 20\ mS$ Two swapped L.U.T
During the vertical retrace $t < 600\ \mu S$ One L.U.T
During the previous frame $t < 20\ mS$ Two swapped L.U.T

FIGURE 5.13. The solution for a control and loading device

In each computing unit of the dot product, 6144 data have to be loaded for the shading of one frame. That can be done during the 20 *ms* of the current frame to be used at the next one, or during the 600 μs of the vertical retrace. To load the diffuse table, we have to store 16384 data in each of the three tables (one for each Red, Green and Blue color). Finally for the specular computation we need to store 16384 values calculated by an algorithm which takes 20480 cycles to compute. (We do not care about the time taken by each of the 16 changes of the arguments during the computation.)

Some classical microprocessors could divide the job to finish it during a frame. Their number depending on its clock rate and of the part they compute. But it is not feasible for them to make it during the vertical retrace. Their number will be too important. If we want absolutely to finish the job during the vertical retrace we must use a wired sequencer.

The microprocessor version

Figure 5.14 shows the implementation of the first solution, using a classical microprocessor (the MC68000 running at 8 Mhz) to compute the content of the dot product look-up table. We do not choose a transputer at this time because it is more expensive and more difficult to program in the assembling language.The connection with the host may be done by a transputer that gives data to all the CPU units of all the shading units. As we use a microprocessor having a 32 bits ALU, we integrate the error register and the counter used for the stored value, into D. Then the overflow of error and the increase of the counter become obvious and are running at the same time by the computation of D. For each calculated value, only the bits from 11 to 21 of the result are stored in the L.U.Ts. This implicit division by 2048 is done by the wiring between the microprocessor and the memories.

The control unit, named "CTRL" builds the chip select of each part of the architecture, such as the multiplexor for instance. The most expensive part of this hardware is the terminate adder that computes the sum of three terms. That may take a lot of place on a board if it is realized with TTL components.

Algorithm running at each frame

```
{ A <--- 0
  D <--- 0
  [A++] <--- D
  do 2047 times:
  {  D <--- D + $V_x$
     [A++] <--- D }
  A <--- 0
  D <--- 0
  [A++] <--- D
  do 2047 times:
  {  D <--- D + $V_y$
     [A++] <--- D }
  A <--- 0
  D <--- 0
  [A++] <--- D
  do 2047 times:
  {  D <--- D + $V_z$
     [A++] <--- D } }
```

execution time:
≈ 110592 Cycles = 13824 μs < 20 ms

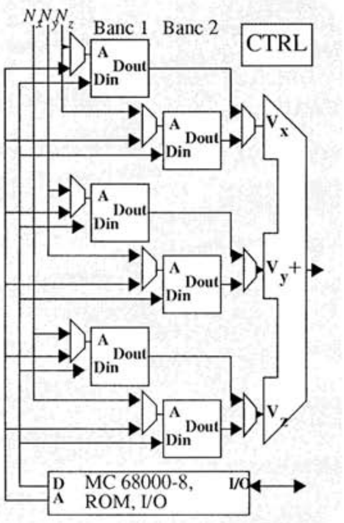

FIGURE 5.14. The algorithm and the microprocessor version

After the computation of dot product all the numbers treated are positive and are coded on 10 bits. For each of the three components of the light, 16384 data have to be stored in the LUT for the shade of the next frame. The diffuse component computation could be done by three CPU MC68000-16 (working with a 16 Mhz clock rate). Each of them works independently for one of the primary color (red, green and blue). These CPU run, for the 16 matters, the incremental algorithm to calculate the multiplication table by the value of $(I \cdot K_d)$.

The control and loading unit, of the specular part, has in its program, for each usable specular power, a coded version of the function. A MC68000-12 (working with a 12 Mhz clock rate) could compute all the values for a frame in less than 19 *ms*. Only 1280 operations have to be done to load the L.U.Ts for each matter. The "specular power ROM" that contains the way to draw the power function is the same as the ROM containing the program of the MC68000. In fact the way is stored as program and not as data in this case. There is an op-code used for the horizontal move and another one of the vertical move.

The wired sequencer version

On the other hand, we can choose the second solution. In the dot product unit, we use the same adder to calculate the sum of the three numbers of the dot product and to compute the data stored in the L.U.T. During the time of the vertical retrace no value had to be done by the dot product unit, so there is no trouble to work like that. The microprocessor disappears and is replaced by the wired sequencer that we can see on Figure 5.15.

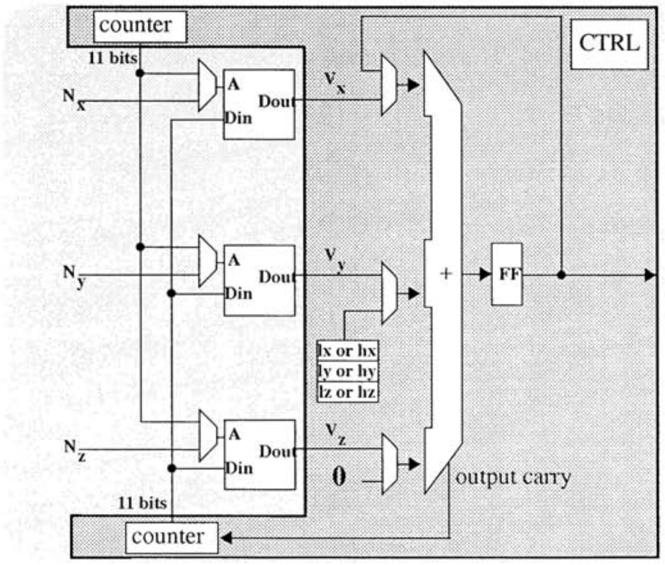

FIGURE 5.15. The wired sequencer

A wired sequencer, working for an 512×512 screen, with a clock time equal to 60 *ns*, takes about 370 *μs*, again the 600 *μs* of the vertical retrace, to execute the 6144 operations of loading in the dot product L.U.T. This sequencer could easily be done in an ASIC. But the performances of the EPLD continue to grow every year and offer us an interesting solution to make each CPU of the dot products. To give an example, the EPM5192 seems, by looking the data sheet, to be important enough for that work.

For the specular computation or for the diffuse computation, even by using a wired sequencer, we do not have the time to store the value during the vertical retrace, as long as we do not choose a quicker clock rate. The same rate clock will be able to support the 1024×1024 on an interlace screen. An other solution could also consists in reducing the number of bits used for the computation of the diffuse. Only 9 bits may be needed. The clock used to read and to store in the LUT could be different without any trouble.

The design of the specular wire sequencer is about the same as the diffuse one. The only difference concerns the drive of the counter that uses the content of the "specular power ROM" to know which counter is increased.

5.3.7 The Normalization Unit

Normalizing the incoming surface normal vector requires three square computations, two additions, one square root extraction and three divisions. The square root extraction and the three divisions can be replaced by three multiplications and one constant L.U.T. for the computation of $\frac{1}{\sqrt{x}}$ (see figure 5.16).

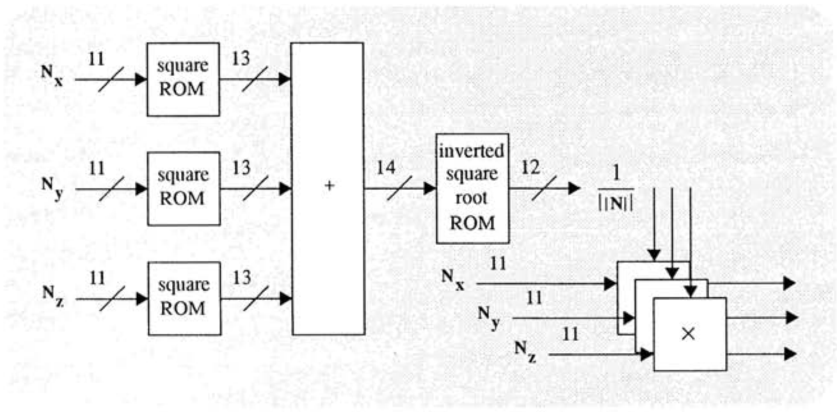

FIGURE 5.16. Scheme of the normalization unit

This part remains the most important because of the product we have not yet been able to eliminate. We continue our investigation to reduce that. A DSP-based system may be the solution. But we could say a solution exists, even if it requires a very important and expensive hardware.

5.4 Conclusion

We have presented in this paper some details of the hardware implementation of the I.M.O.G.E.N.E. system, a massively parallel object-oriented graphics system. We have described the design of the first order Elementary Processor and of the Triangle Processor.We have also presented the hardware implementation of the deferred shading processor computing a Phong shading . This processor can also be used in a classical frame buffer-based graphics system, provided that the surface normal vector be stored instead of RGB values.

Acknowledgements:

This work is supported by French Research Program on New Architectures (PRC ANM). We wish to thank Eric Nyiri for his work on the high level simulator of the system.

5.5 References

[1] Akeley, K. The Silicon Graphics 4D/240GTX Superworkstation. *IEEE Computer Graphics and Applications*, Vol. 9 Num. 4, July 1989, pp. 71-83

[2] Apgar, B. Bersack, B. and Mammen, A. A Display System for the Stellar Graphics Supercomputer Model GS1000. *ACM Computer Graphics*, Vol. 22 Num. 4, August 1988, pp. 255-262

[3] Blinn, J. F. Computer Display of Curved Surfaces. PhD thesis, University of Utah, Department of Computer Science, December 1978

[4] Chaillou, C. Karpf, S. and Meriaux, M. I.M.O.G.E.N.E: A Solution to the Real Time Animation Problem. In: R.L.Grimsdale, A. Kaufman (Eds.):*Advances in Computer Graphics Hardware V*, EurographicSeminars. Springer-Verlag, Berlin, 1992, p.139-151.

[5] Claussen, U. On Reducing the Phong Shading Method. *Proc. Eurographics'89*, Elsevier Science Publishers B.V., 1989, pp. 333-344

[6] Deering, M. Winner, S. Schediwy, B. and al. The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. *ACM Computer Graphics*, Vol. 22 Num. 4, August 1988, pp. 21-30

[7] Fuchs, H. Poulton, J. Eyles, J. and al Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *ACM Computer Graphics*, Vol. 23 Num. 3, July 89, pp. 79-88

[8] Karpf, S. Chaillou, C. Nyiri, E. and Meriaux, M. Real-time Display of Quadrics Objects in the I.M.O.G.E.N.E. Machine. *Proceedings ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications*, June 1991, pp. 269-277

[9] Kirk, D. and Voorhies, D. The Rendering Architecture of the DN10000VS. *ACM Computer Graphics*, Vol. 24 Num. 4, August 1990, pp. 299-307

[10] Phong, B. T. Illumination for Computer Generated Pictures. *Communications ACM*, Vol. 18 Num. 18, June 1975, pp. 311-317

[11] Schneider, B.O. A Processor for an Object-Oriented Rendering System. *Computer Graphics Forum*, Num. 7, 1988, pp. 301-310

[12] Tebbs, B. Neumann, U. Eyles, J. and al Parallel Architectures and Algorithms for Real-Time Synthesis of High Quality Images using Deferred Shading. *Proc. Workshop on Algorithms and Parallel VLSI Architectures*, 10-16 June, Part A pp. 152-156

[13] Torborg, J. A Parallel Processor for Graphics Arithmetic Operations. *ACM Computer Graphics*, Vol. 21 Num. 4, July 1987, pp. 197-204

[14] Weinberg, R. Parallel Processing Image Synthesis and Anti-Aliasing. *ACM Computer Graphics*, Vol. 15, Num. 3, August 1981, pp. 55-62