Direct Visualization of Quadrics

Laporte H.
Nyiri E.
Froumentin M.
Chaillou C.

ABSTRACT
Today, most of the powerful graphic systems are based on 3D-triangle display methods. However, this approach generates well-known problems, like the low quality of contours and shading, and the necessity to have large amounts of primitives to display complex scenes. A way to solve these problems is to use higher level primitives, among which a very interesting one is the quadric surface. We study here direct visualization of quadric surfaces (quadrics for short).
Although we study all the rendering process, we focus on the scan conversion stage. First, we present mathematical and modeling backgrounds where we show that a quadric surface can easily be rendered in scan-line. Then we give the general algorithm and details about the difficult part, i.e. the bounding plane algorithm. A functional description of a scan converting processor is proposed, using a modular approach. In the last part, we give a hardware implementation of each module and the whole processor. We also do an estimation of the silicon cost. The conclusion is that our quadric patch scan converter can actually be realized.

1.1 Introduction

Today, machines dedicated to image synthesis use polygons as display primitives. There are several reasons to this. First polygons (and more particularly triangles) are very simple shapes so complexity is low and few calculations are required to render them (i.e. bilinear interpolation). Furthermore there are lots of optimized algorithms and special purpose components for scan conversion. Today the most efficient machines are able to compute more than one million Gouraud shaded triangles per second [5].

Nevertheless drawbacks exist. We can point out the problems of approximated contours (because of the tessellation of modeling primitives such as B-spline or Bézier patches) and the fact that a huge number of triangles is necessary to represent a scene.

To avoid these drawbacks, using higher level primitives is promising. A few attempts have been made especially in the Pixel-Plane 5 machine [4]. Our goal is to go further in this way. Using quadric surfaces seems interesting as they are defined by a quadratic implicit equation, depths and normal vectors are easily computed for scan-line Z-buffer algorithms.

However this alternative to the facet involves important modifications. The complete study is devided into two parts. The first part deals with modeling problems. The second one considers the rendering pipeline, mainly including host requirements, anti-aliasing, texture mapping and of course direct visualization. This paper only discusses direct visualization issues.

In the first part of this paper, we present the quadric and the modeling background. Then, visualization algorithms are given. In the third part, we propose a complete functional description. In the end, we study hardware implementation and estimate the VLSI cost.

1.2 Quadric Surfaces

1.2.1 Presentation

A quadric is a 3D surface defined by the implicit equation:

$$Q(x,y,z) = ax^2 + by^2 + cz^2 + dxy + exz + fyz + gx + hy + iz + j = 0 \tag{1.1}$$

For each couple (x,y) we can easily find the back and front depths (z) where the quadric is present (if any):

$$z = F1(x,y) \pm \sqrt{F2(x,y)} \tag{1.2}$$

where $(c \neq 0)$

$$\begin{array}{l} F1(x,y) = \frac{-e}{2c}x + \frac{-f}{2c}y + \frac{-i}{2c} \\ F2(x,y) = \frac{e^2-4ac}{4c^2}x^2 + \frac{f^2-4bc}{4c^2}y^2 + \frac{ef-2cd}{2c^2}xy + \frac{ei-2cg}{2c^2}x + \frac{fi-2ch}{2c^2}y + \frac{i^2-4cj}{4c^2} \end{array} \tag{1.3}$$

If $F2(x,y)$ is positive, then the quadric intersects the line defined by x,y an parallel to the z-axis.

The coordinates of the normal at (x,y,z) are the partial derivatives of $Q(x,y,z)$:

$$\left\{ \begin{array}{l} NX = \frac{\partial Q}{\partial x}(x,y,z) = 2ax + dy + ez + g \\ NY = \frac{\partial Q}{\partial y}(x,y,z) = 2by + dx + fz + h \\ NZ = \frac{\partial Q}{\partial z}(x,y,z) = 2cz + ex + fy + i \end{array} \right. \tag{1.4}$$

The depth values are used in the Z-buffer and the normal vector for shading.

1.2.2 Modeling background

Usually, quadrics are used as volumes, within a Constructive Solid Geometry environment. But CSG is not well-suited for real-time rendering. Therefore, an approach based on quadric *surfaces* has to be considered to define a Z-buffer graphics system. A "Quadric Patch" is then defined including the surface and bounding planes. Recent studies [6, 3] show that using the Bernstein-Bézier representation for quadrics simplifies the construction of piecewise quadratic surfaces with G^0 or G^1 continuity. This representation uses tetrahedra as bounding volumes. Then, four bounding planes appear to be necessary.

Parallel works at the LIFL include, on one hand, modeling with quadric patches as primitive, on the other hand, "quadric tesselation" algorithms, i.e. algorithms which convert complex primitives like B-spline surfaces into a mesh of quadric patches.

1.3 Algorithms

The algorithm works in scan-line, using the formulas given in 1.2.1. Note that the projection of the patch is displayed inside the projection of the bounding box. Shading (true Phong shading can be used [9]) and Z-buffer operations are post-processed.

1.3.1 General algorithm

Assuming that $ZP1$, $ZP2$, $ZP3$, $ZP4$ are the plane depths and $NXBASE$, $NYBASE$, $NZBASE$ are intermediate results for the normal component computation (see 1.4.3) the algorithm is:

```
Initialize zone
FOR each line DO
        initialize line
        FOR each point DO
                Compute F1, F2, ZF, ZB, NXBASE, NYBASE, NZBASE, ZP1, ZP2, ZP3, ZP4
                Execute boundary-plane algorithm {described below}
                compute NX, NY, NZ
        END
END
```

1.3.2 Boundary-plane algorithm

For visualization, a boundary-plane is used as follows: the plane cuts 3D space into two parts. If the quadric is in the correct half-space we keep it, else we cut it. To determine which half space is the correct one we use the z-coordinate of the normal vector to the plane (called CP). The main task is to compare quadric depths (ZF is the front depth and ZB is the back one) with plane depths (called ZP) and to select the correct depth. To perform this, we use two flags (called EF and EB), one for each quadric depth. A value of one means that the depth is able (i.e. not eliminated by a cut into the quadric due to a boundary plane) and of course a value of zero means that the depth is unable. At the beginning of the algorithm they are both set to one. Then we cut the quadric with the four boundary planes. In the end, there are three possibilities. $EB = EF = 0$ means that the whole quadric is eliminated so we need $ZMAX$ (the maximum depth value). If $EF = 1$ then we keep ZF (no matter what the value of EB is). If $EF = 0$ and $EB = 1$ then we keep ZB.

Figure 1.1 shows in 2D a quadric patch with one front plane and one back plane.

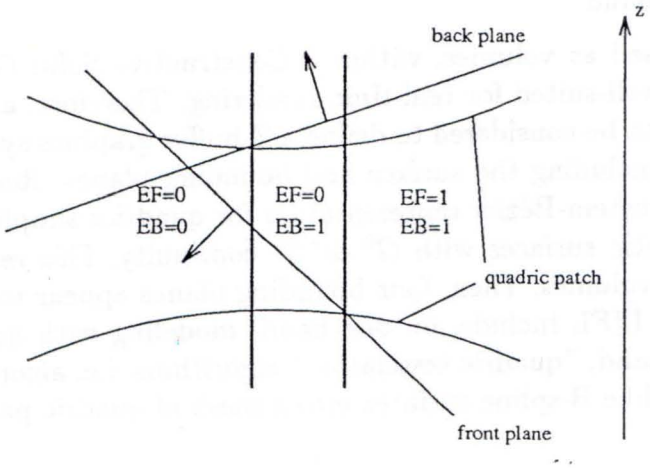

FIGURE 1.1. General diagram

Notice that Z' is the depth which is used to compute the normal coordinates whereas $ZOUT$ is the output depth. If $ZOUT = ZMAX$ Z' has no meaning and the normal vector neither.

```
{initialization}
EF=1;EB=1;
FOR each plane i DO
    {Cutting-plane algorithm}
    IF CPi=0 {perpendicular plane}
```

```
        IF ZPi<0
           EF=0; EB=0
    ELSE IF CPi<0 {front plane}
              IF ZPi>ZB
                  EF=0; EB=0;
              ELSE IF ZPi>ZF
                  EF=0;
           ELSE {back plane} IF ZPi<ZF
                  EF=0; EB=0;
              ELSE IF ZPi<ZB
                  EB=0;
END
{selection of the right depth}
IF EF=1 Z'=ZF ELSE Z'=ZB
IF (EF=0 and EB=0) ZOUT=ZMAX ELSE ZOUT=Z'
```

1.4 Functional description

Our aim is to define a processor for quadric surface calculations. After studying the mathematical basis and the algorithms the next step is a functional description of this processor. First we present a general description and then we give details about each of its modules.

1.4.1 General description

We present in figure 1.2 the general diagram corresponding to the algorithm given in 1.3.1. In this figure, the bounding plane module corresponds to the algorithm given in 1.3.2

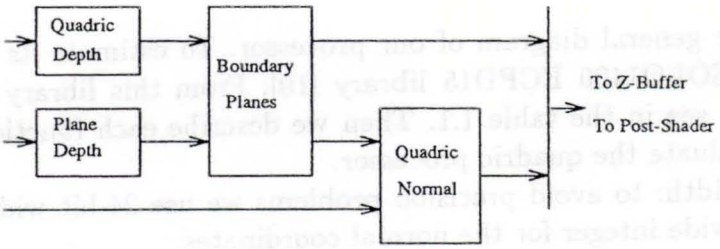

FIGURE 1.2. General diagram

1.4.2 Quadric Depth and Plane Depth Modules

Figure 1.3 shows the ZF and ZB depth calculations which are performed using formulas given in 1.2.1.

The computation of the depth of a single plane is given by a linear expression. It is a basic computation, hence we do not describe it.

1.4.3 Quadric normal module

The normal vector coordinates are computed in two steps. The first one is the calculation of the three linear expressions that we call the NXBASE, NYBASE and NZBASE components, assuming that $NXBASE = 2ax + dy + g$, $NYBASE = 2by + dx + h$ and

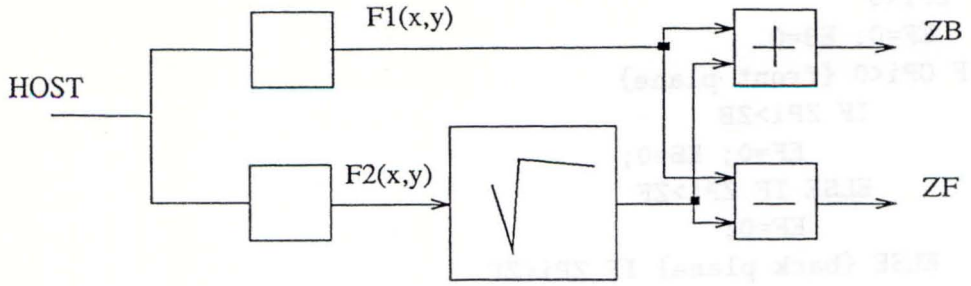

FIGURE 1.3. Quadric depth diagram

$NZBASE = ex + fy + i$.

The second step is then: $NX = NXBASE + e \times Z'$, $NY = NYBASE + f \times Z'$, $NZ = NZBASE + 2c \times Z'$ where Z' is wether ZF or ZB depending on flags EF and EB. Notice that as we can either see the inner side or the outer side of the quadric, we have to change the orientation of the normal if it is the back one. We just have to change the sign of NX, NY and NZ if $EF = 0$ and $EB = 1$.

1.4.4 Bounding-plane Module

For each plane, we have five criteria: sign of $ZPi - ZF$, sign of $ZPi - ZB$, sign of CPi (which is the z-coordinate of the plane normal), flag ($CPi = 0$?) and sign of ZPi. To perform the first two tests, we need to compute the differences $ZPi - ZF$ and $ZPi - ZB$. Then we determine the EF and EB flags. Besides we can add sign of $F2(x, y)$ which indicates wether the quadric is present or not. Then we can do the selection to obtain Z' and $ZOUT$.

1.5 Hardware implementation

Figure 1.4 presents the general diagram of our processor. To estimate its cost, we take figures from the ES2 SOLO1400 ECPD15 library [10]. From this library we construct basic cells that we can see in the table 1.1. Then we describe each functional block we need and finally we evaluate the quadric processor.

Consider the data width: to avoid precision problems we use 24-bit wide integers for the depths and 16-bit wide integer for the normal coordinates.

The unit for all the figures is the transistor.

TABLE 1.1. Basic Cells

Cell	Transistors
24-bit register without clear	24×26 = 624
24-bit register with clear	24×32 = 768
24-bit adder	24×36 = 864
48 bit-to-24 bit multiplexer	24×14 = 336
24-bit tristate inverter	24×8 = 192

1.5.1 The EP1 and EP2 elementary processors

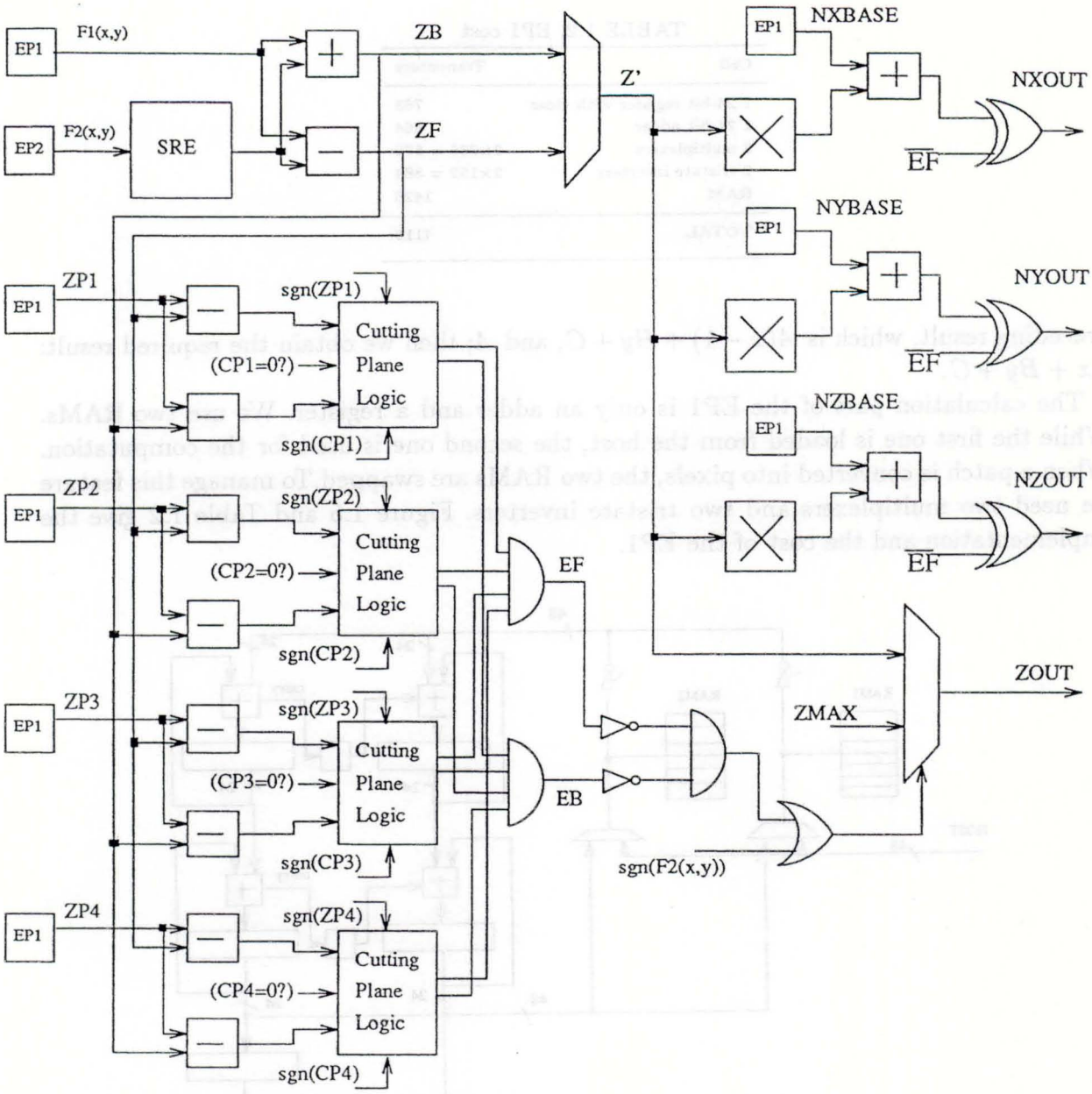

FIGURE 1.4. Quadric patch processor: general diagram

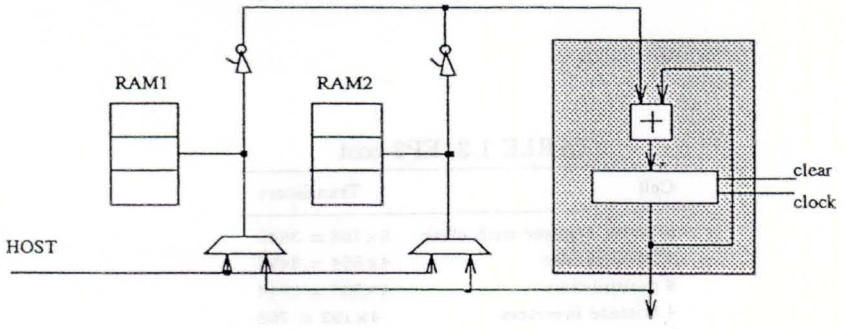

FIGURE 1.5. EP1 implementation diagram

These two cells use a forward differencing technique [1]. The EP1 computes a linear expression: $F1(x,y) = Ax + By + C$. At each line, we compute $By + C$ by adding $B(y-1) + C$, which is the result of the preceding line, and B. Then at each pixel we add the

TABLE 1.2. EP1 cost

Cell	Transistors
1 24-bit register with clear	768
1 24-bit adder	864
2 multiplexers	2×336 = 672
2 tristate inverters	2×192 = 384
RAM	1428
TOTAL	4116

preceding result, which is $A(x-1)+By+C$, and A; then we obtain the required result: $Ax+By+C$.

The calculation part of the EP1 is only an adder and a register. We use two RAMs. While the first one is loaded from the host, the second one is used for the computation. When a patch is converted into pixels, the two RAMs are swapped.To manage this feature we need two multiplexers and two tristate inverters. Figure 1.5 and Table 1.2 give the implementation and the cost of the EP1.

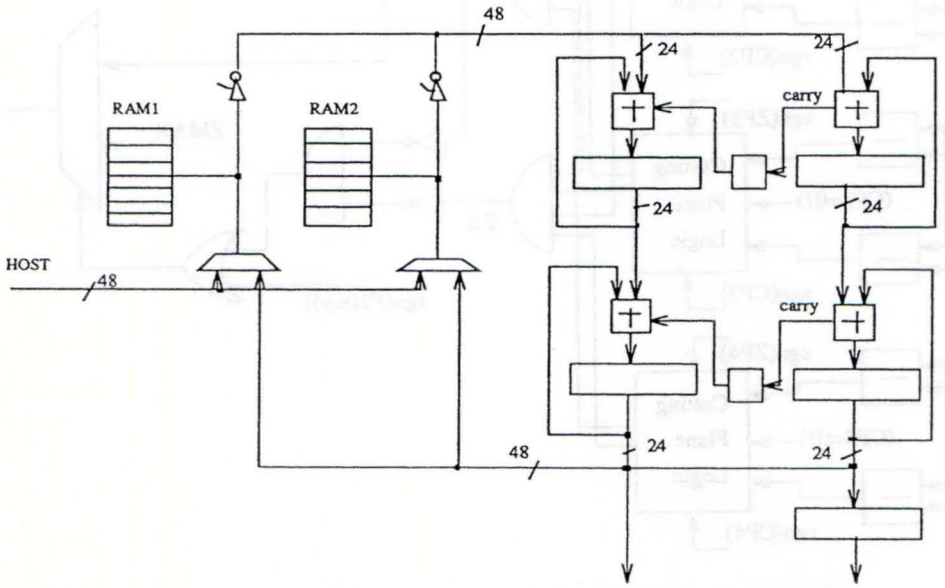

FIGURE 1.6. EP2 implementation diagram

TABLE 1.3. EP2 cost

Cell	Transistors
5 24-bit register with clear	5×768 = 3840
4 24-bit adder	4×864 = 3456
4 multiplexers	4×336 = 1344
4 tristate inverters	4×192 = 768
RAM	5712
TOTAL	15120

The EP2 computes a quadratic expression: $F2(x,y)=Ax^2+By^2+Cxy+Dx+Ey+F$. First assume that $F2(x,y)=Ax^2+Px+Q$ with $P(y)=Cy+D$ and $Q(y)=By^2+Ey+F$.

Because the computation of $F2(x,y)$ and $Q(y)$ is the same, we only present here how to compute $Q(y)$: assuming that $Q(y+1) = Q(y) + 2By + E + B$ with $R(y) = 2By + E - B$ and $R(y+1) = R(y) + 2B$, we have $Q(y+1) = Q(y) + R(y+1)$. Therefore, we can easily compute incrementally $Q(y)$ and then $F2(x,y)$.

The EP2 is implemented with 2 calculation cells (see 1.6). As the expression is quadratic we work with 48-bit integers and six word RAMs. Assuming that we cannot compute a 48-bit addition in one cycle clock we use two pipelined 24-bit adders for each cell. At the end of the computation, we use a synchronization register. The EP2 cost is given in table 1.3.

1.5.2 The square root extractor (SRE)

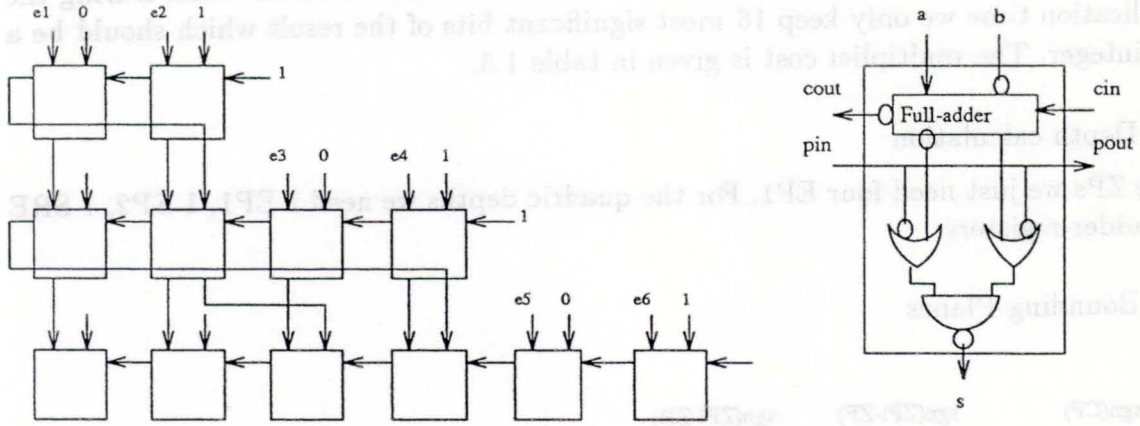

FIGURE 1.7. Square root extractor: array and basic cell

TABLE 1.4. SRE cost

Cell	Transistors
600 computation cells	600×56 = 33600
2423 D-latches	2423×26 = 62998
TOTAL	96598

The square root processor (figure 1.7) is a cellular array described in [8, 2, 7]. It computes a division where the divider and the result are equal. Each line computes one bit of the result. The basic cell computes a substraction and a selection (of the result of the substraction or the predeeding rest).

If n is the bit-width of the result the array is made with n lines, the i^{th} line made with 2×i cells. For n=24 we have 600 cells of 56 transistors each. Notice that a cell is roughly a full adder and a multiplexer (we will consider that its computation time is the same as that of an adder). Moreover we have to manage the pipeline. First we put a 48-bit register between each line of the array (cost: 24*48 = 1152 D-latches). But the lines in the second half of the extractor are larger than 24 cells so we cannot compute one line in one clock period. That is why we have to use 1271 additional registers (the pipelining is quite complex and we will not explain here these figures). The total number of D-latches is 2423. The SRE cost is given in table 1.4.

1.5.3 The multiplier

TABLE 1.5. Multiplier cost

Cell	Transistors
16×16 adders	16×16×36 = 9216
16×16 D-latches	16×16×26 = 6656
TOTAL	15872

The multiplier is a classical parallel one. However, the normal coordinates are 16-bit integers so we pick up the 16 most significant bits of Z' which is 24-bit wide. During the multiplication time we only keep 16 most significant bits of the result which should be a 32-bit integer. The multiplier cost is given in table 1.5.

1.5.4 Depth calculation

For the ZPs we just need four EP1. For the quadric depths we need 1 EP1, 1 EP2, 1 SRE and 2 adder-registers.

1.5.5 Bounding Planes

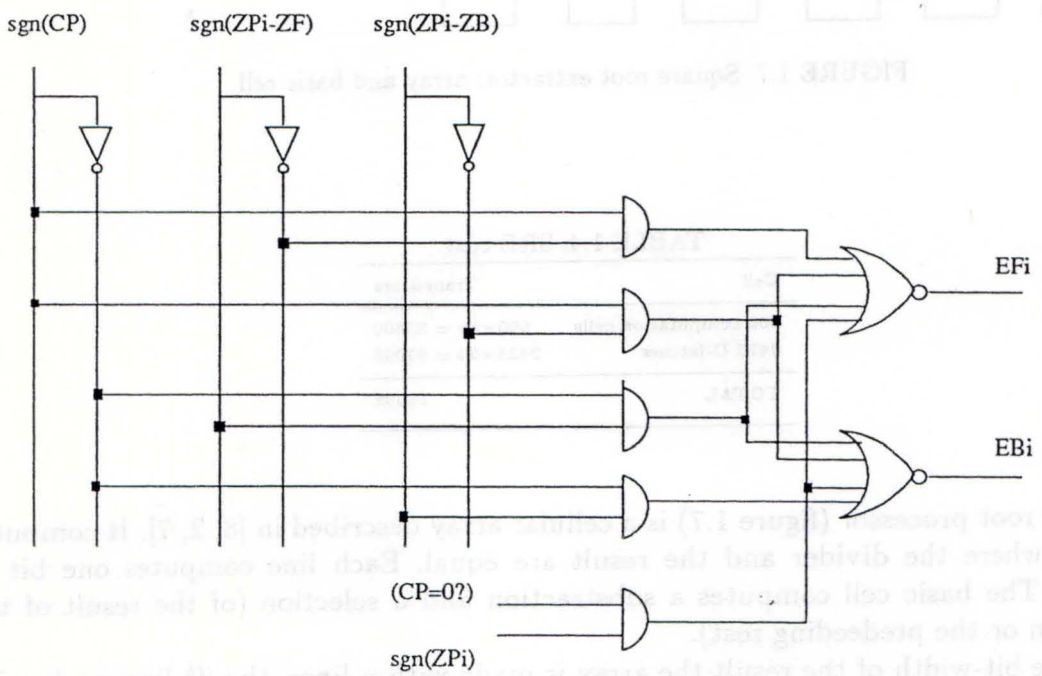

FIGURE 1.8. Cutting plane logic

First we need eight 24-bit adder-registers to perform the ZPi-ZF and ZPi-ZB differences. To describe one "cutting-plane" logic, we have to list all the cases where ZF and/or ZB have to be unabled. Assuming that $sgn(x) = 0$ when x positive and $sgn(x) = 1$ when x negative.

For ZF and ZB (i.e. when $ZOUT = ZMAX$): $(CP = 0?) = 1$ and $sgn(ZP) = 1$, $sgn(CP) = 1$ and $sgn(ZP - ZB) = 0$, $sgn(CP) = 0$ and $sgn(ZP - ZB) = 1$.

For ZF only: $sgn(CP) = 1$ and $sgn(ZP - ZF) = 0$.
For ZB only: $sgn(CP) = 0$ and $sgn(ZP - ZB) = 1$.

We group these conditions, taking care of the fact that we want to set EF and EB to zero and not to one (it means only a logic NOT). We obtain the diagram of figure 1.8. The flag ($CP = 0$?) is supplied by the host. All the cutting plane logics work simultaneously. We group the result with one 4-input logic AND for EF and another one for EB. Finally we select Z' and $ZOUT$.

We need one multiplexer for Z'. The control bit is EF. We need another multiplexer for $ZOUT$. The control bit is $\overline{EF} \bullet \overline{EB} + sgn(F2(x, y))$. The bounding plane control logic (with the two multiplexers and without adder-registers) is only $76 \times 4 + 46 + 672 = 1022$ transistors.

1.5.6 Pipelining and control

We do not focus here on the control of the processor because it is an external one. Indeed, the processor is used in a parallel architecture. So a dedicated component controls all the quadric patch processors. This control could be performed with a automaton or a micro-programmed circuit.

The pipelining is not very complex, given that each cell in the processor is already internally pipelined. We only have to add a few registers to synchronize each cell with the other ones. Table 1.6 shows the scheduling of calculations. The first column gives the start time and the end time of each step. The following column is the name of the step. The last one gives the need of extra-storage.

TABLE 1.6. Scheduling

Timing	Step	Extra storage
0/2	F2(x,y)	
2/38	SRE	
37/38	F1(x,y)	
38/39	ZF,ZB	
39/40	ZP-ZF,ZP-ZB	ZF,ZB
40/41	Z',ZOUT	Z',ZOUT
41/57	Multiplier	ZOUT
56/57	NXBASE,NYBASE,NZBASE	
57/58	NXOUT,NYOUT,NZOUT	ZOUT

We need $2+2+16+1 = 21$ extra-registers which costs $21 \times 26 \times 24 = 13104$ transistors. We can also see that the start up of the pipeline is 58 clock cycles.

1.5.7 Hardware cost of the whole processor

All the previous results are gathered in table 1.7.

A few words about these figures: first the total number of transistors needed is big but not huge. Moreover, the best VLSI-design softwares can reduce these figures. For example, the size of our D-latch is 26 transistors which is quite big. We also want to point out that the cost is paid for the square root extractor and the multipliers. To reduce the size of the processor, one must work on these two cells.

Let us have a quick comparison with triangles. We have designed a classical facet processor in the same condition than the quadric patch processor. Its size is about 35000 transistors, i.e. seven times smaller than the quadric one. Given that we think that a

TABLE 1.7. Processor cost

Cell	Transistors
8 EP1	8×4116 = 32928
1 EP2	15120
1 SRE	96598
3 multipliers	3×15872 = 47616
13 adder-registers	13×(864+624) = 19344
Bounding-plane logic	1022
Pipelining registers	13104
TOTAL	225732

quadric patch replaces at least ten triangles, this figure is very satisfying.

1.6 Conclusion and future works

The aim of this paper is to prove the feasability of a quadric patch processor. We have first described the quadric surface and given the scan-conversion algorithms. At that moment the main difficulty was the bounding of the patch. After a functional description we have focussed on hardware implementation. With a modular approach, we have described precisely each cells of the processor and evaluated the silicon cost. After having noticed that this time, the square root extractor and the three multipliers represent the two thirds of the circuit, we conclude that the quadric patch processor can actually be realized.

In this study, we have used a rendering software developed on a SUN workstation. Data come from a modeling software also developed by our team. We have implemented the algorithms, refined them and evaluated their costs. Of course, we have also implemented the preparation tasks and the shading. At this time we are writing a program to precisely emulate the whole processor.

Our goal is to display any kind of scenes, directly using quadric patches. Used with a post shading unit, we think that better quality is worth higher computation cost. First results are promising but there is still much to do. The next step is to implement an optimized algorithm on a DEC-Alpha board. Then we have to study the host requirements, the shader and then the whole architecture which combines quadrics and facets. Besides, we have to study quality improvement such as anti-aliasing or texture-mapping. The definition of the quadric patch processor is the first step toward the design of a new graphics system, which will improve quality without reducing performances.

1.7 References

[1] S. L. Chang, M. Shantz, and R. Rocchetti. Rendering cubic curves and surfaces with integer adaptative forward differencing. *ACM Computer Graphics*, 1989.

[2] Cowgill D. Logic Equations for a Built-in Square Root Method. *IEEE Transactions on Electronic Computers*, 1964.

[3] Wolfgang Dahmen. *Smooth Piecewise Quadric Surfaces*, chapter 23. Academic Press, 1989.

[4] Eyles J. Fuchs H., Poulton J. and al. Pixel Plane 5: A Heterogeneous Multiprocessor Graphics System Using Processor-enhanced Memories. *ACM Computer Graphics*, 1989.

[5] Silicon Graphics. Reality Engine in visual simulation. Technical report, SGI, 1992.

[6] Baining Guo. Representation of arbitrary shapes using implicit quadrics. *The Visual Computer*, 1993.

[7] Majithia J.C. Cellular Array for Extraction of Squares and Square Roots of Binary Numbers. *IEEE Transactions on Computers*, 1972.

[8] H. Laporte. Etude et conception d'un composant VLSI dans le cadre du projet IMOGENE : l'extracteur de racine carree. Master's thesis, University of Lille (France), 1991.

[9] V. Lefevere, C. Chaillou, and M. Meriaux. Low cost hardware for real time Phong shading. In *Proc. of Graphics Interface'92, workshop on local illumination*, 1992.

[10] European Silicon Structures. ECPD15 Process Databook. Technical report, ES2, 1989.