11 An Architecture for a High Performance Rendering Engine

Hans-Josef Ackermann
Christoph Hornung

ABSTRACT
We present an architecture for a high-performance programmable rendering engine. This chip or chip-set will be able to deliver one Gouraud-shaded, z-buffered, texture-modulated and alpha-blended pixel every clock cycle. Focus of the paper is the derivation of the architecture of the pixel processing block from the applied algorithms.

11.1 Introduction

Nowadays graphics systems often consist of high performance RISC-processors for geometry calculations and specialized ASICs for rendering. Such an architecture provides high performance for both geometry processing and rendering, but offers only little flexibility in the rendering area as discussed in [1]. Yet there is a clear trend towards more flexible rendering algorithms. Beyond Gouraud-shading, alpha-blending to implement transparency and anti-aliasing as well as different kinds of texture and data mapping become standard features of graphics applications [2]. Even more [3], [4] require configurable and multiple stage shading algorithms using iterative and/or recursive techniques. This chapter addresses this problem and proposes an architecture for a programmable rendering engine (PRE). This chip or chip-set will provide both high rendering performance by executing basic shading functions in a single clock cycle and flexibility by being programmable. Besides the overall architecture of PRE, the chapter will mainly focus on the pixel processing block described in Section 11.4.

11.2 Goals

As the Triangle Shading Engine described in [5], the PRE-processor is intended to work together with one ore more Digital Signal Processors in a system, which is balanced in its geometry calculations to rendering performance ratio. The main goals of the proposed architecture are to reach both high performance and wide flexibility. Besides this, interfacing the PRE-processor should be kept as easy as possible in order to achieve a simple overall system design minimizing necessary PCB space and optimizing cost / performance ratio.

11.2.1 Performance

To achieve high performance, the PRE will be able to output a pixel, processed using one of the following basic algorithms, each clock cycle:

- z-buffering
- Gouraud-shading
- α-blending
- texture-mapping

Compared to the intended cycle time of the PRE, access to standard dynamic memory used as r,g,b,α and z-buffer is a bottle-neck. Thus the memory interface will be designed to allow one read and one write access each clock cycle using interleaving technique.

Depending on the available process, a single cycle execution time down to 25ns corresponding to a rate of 40 Mpixel/sec is projected for a single PRE system. For high end applications, multiple PRE processors could be used within a distributed graphics system to achieve scalable performance.

11.2.2 Flexibility

To gain the aspired degree of flexibility, the PRE-processor will be on-line micro-programmable. Thus simple shading algorithms as well as complex and lately developed ones will be supported. The "Shade Trees" approach [4] has been chosen as a test-bed to prove the flexibility of the PRE architecture. Although the architecture of the PRE-processor is fit for a wide variety of shading algorithms, it will be optimized to support the basic algorithms listed before. All data will be processed using a fix-point representation.

11.3 Architecture of the Programmable Rendering Engine

The Programmable Rendering Engine consists of the following components as shown in Figure 11.1:

- edge processing block
- pixel processing block
- control block
- memory interface

The edge processing block scan converts the edges of the rendering primitives. The pixel processing block forms the inner loop by doing the pixel processing for a scan line. the control block contains the micro sequencers. The memory interface provides flexible and fast memory access according to different memory organizations.

11.3.1 Edge Processing Block

The edge processing block interpolates the different data (coordinates, z-values, colors, alpha, texture coordinates) along an edge of the rendering primitives. Rendering primitives supported will be:

- lines
- triangles
- planar trapezoids
- triangle strips

The edge processing block must be able to do proper subpixel calculations to support alpha-blending, texture-mapping and anti-aliasing. This can be reached using the algorithm proposed in [5]. The input data consisting of coordinates, color-, and z-values of the primitives edges as well as edge and span increments will be processed in fix-point representation. The resolution of the fractional part will be sufficient to avoid rounding errors in any case.

11.3.2 Pixel Processing Block

The pixel processing block interpolates the different data (coordinates, z-values, colors, alpha, texture coordinates) along a span. This block implements point sampling at the centers of pixels according to [5]. This approach avoids all the artifacts known by using DDA-algorithms [1]. Details will be described in section 11.4.

11.3.3 Control Block

The control block contains micro sequencers with their associated microcode RAM. The two processing blocks and the memory interface are controlled independently. The operation of the basic building blocks in the pixel processing block are configurable in parallel with a long instruction word in a cycle to cycle fashion.

The set-up data interface will support 32 bit and 64 bit operation. The interface will provide automatic loading of data sets from an externally connected FIFO. The read cycle will be 25ns. Loading set-up data will work independently from interpolation using two sets of set-up registers.

11.3.4 Memory Interface

Compared to the estimated internal processing times of the PRE- processor, data transfer from and to frame-, z- and α-buffer is a bottle-neck when assuming standard dynamic memory cycle times. The Triangle Shading Engine [5] reaches a cycle time of 150ns for a read-modifiy-write-cycle using non- interleaved DRAMs in fast page mode. To achieve a performance of 40 Mpixels/sec, one read and one write cycle of frame- an z-buffer (plus α- and texture-buffer if involved) has to be completed in each clock cycle which is projected to be 25ns. A solution to this problem is the use of dual-ported frame-, z- and α-buffers realized with standard video-RAMs. Hence page mode access time using the random ports of the VRAMs is at least 50ns, the memory chips have to work in an interleaved fashion. The interleave factor will be four. The serial port will serve as read port only whereas the random port will perform write and control operations. An additional advantage when using VRAMs is the possibility of fast buffer clearing by successively writing back the contents of the cleared shift register to all rows of the memory array.

Unlike to frame- z- and α-buffer, the texture-buffer requires random access. Because of the limited size of the texture-buffer (64K by 32 bits) and to satisfy the required access times, fast static memory will be used.

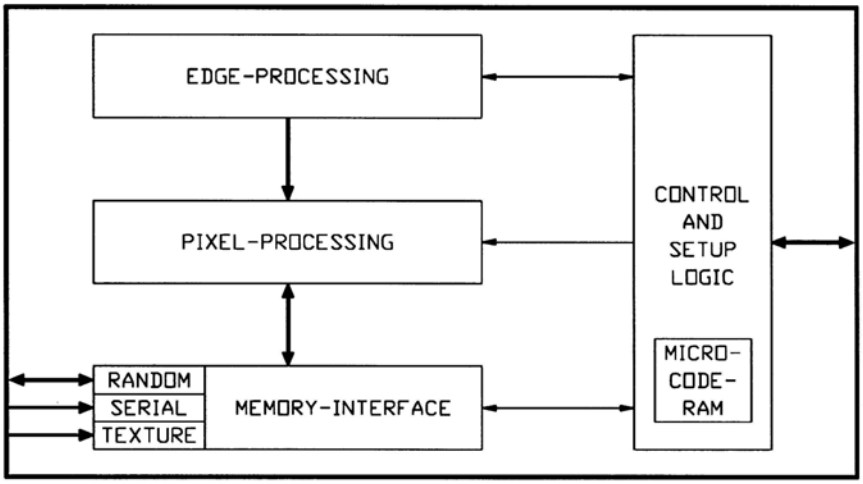

FIGURE 11.1. Block Diagram of the Programmable Rendering Engine

11.4 Derivation of the Architecture of the Pixel Processing Block from the Applied Algorithms

11.4.1 General Remarks

In this section, the pixel processing block will be described. An outline of its architecture based on the requirements of basic shading algorithms will be given. The pixel processing block consists of the following conceptual pipeline:

- reading of buffer-values
- pixel generation
- pixel blending
- writing back processed values

The read unit is used to read the different values from the frame-buffer, the z-buffer and the texture- buffer. The pixel generation unit processes single pixel values, normally by linear interpolation. The pixel blending unit does a modification of pixel values. This is essential for the calculation of one resulting pixel value out of two input values. The final pixel values (z and rgbα) are written back by the write unit. In this section, first the elementary design of the different stages is described. Then, the different stages are extended to allow more complex algorithms.

11.4.2 Memory Layout

The z-buffer will be 24 bit wide. It will be addressed in strictly sequential order, with address increments of either 1 or -1. Moreover, parallel read and write is necessary. These requirements can be fulfilled using VRAMs for the realization of the z-buffer.

The rgbα-buffer will be 32 bit wide, containing one byte for r, g, b and α each. It is organized in the same way as the z-buffer.

The texture-buffer is 32 bit wide, containing a byte for r, g, b and α each. This allows true color textures with an additional α-channel. Textures will be assumed as toroidally closed and being of a size of 2^n x 2^m. As will be seen later, the texture-buffer is randomly addressed. Therefore, fast SRAM is used.

11.4.3 Basic Algorithms

The algorithms of the following sections are described in a C-like notation. The C-code pieces represent the inner loops of the underlying algorithms. There are no explicit type declarations. Table 11.1 gives an explanation of the used types and symbols.

Table 11.1: Basic Types, Structures, Variables and Functions

```
Types:
tCard8                                  cardinal value 8 bits
tCard12                                 cardinal value 12 bits
tCard24                                 cardinal value 24 bits
tCard32                                 cardinal value 24 bits
tFix8_16                                fixpoint value; 8 bit integer, 16 bit fract.
tFix24_16                               fixpoint value; 24 bit integer, 16 bit fract.
typedef struct
{ tFix8_16 x; tFix8_16 y; tFix8_16 z; } tFixXyz
typedef struct
{ tFix8_16 r; tFix8_16 g; tFix8_16 b; tFix8_16 a; }
tFixRgba typedef struct
{ tCard8_16 x; tCard8 y; tCard8 z; } tIntXyz
typedef struct
{ tCard8 r; tCard8 g; tCard8 b; tCard8 a; } tIntRgba
typedef struct
{
tFixXyz Par;                            fixpoint value used during interpolation
                                        of texture address
tIntXyz Addr;                           integer part of interpolated address
tCard16 Offset;                         texture address offset
tCard16 *pBase;                         pointer to texture baseaddress
tCard16 *pTxt;                          pointer to texture address in memory
}
tTxtMemInt
typedef struct
{
tIntRgba *pRgba;                        pointer to Rgba in frame-buffer
tCard12 dpRgba;                         delta pointer to Rgba in frame-buffer
tCard24 *pZ;                            pointer to Z in z-buffer
```

```
tCard12 dpZ;                    delta pointer to Z in z-buffer
tTxtMemInt Txt;
}
tMemInterface;
typedef struct
{
tFixRgba Rgba;                  fixpoint Rgba-value during interpolation
tFix24_16 Z;                    fixpoint Z-value during interpolation
}
tInterpolator;
typedef struct
{
tIntRgba Rgba;                  integer Rgba-value
tCard24 Z;                      integer Z-value
}
tPxlValue;
typedef struct
{
tIntRgba Front;                 Rgba-value of the frontzpixel during blending
tIntRgba Back;                  Rgba-value of the back pixel during blending
tIntRgba Rgba;                  Rgba-value of the pixel after blending
}
tBlend;
```

Variables:

```
tMemInterface Mem;              memory interface
tInterpolator Inter;            interpolation stage
tPxlValue Pxl, Buf;             pixel comparator stage
tBlend Blend;                   pixel blendingstage
```

Basic Functions:

```
FixToIntZ(tCard24_16);          returns the int. part of a tCard24_16 value
FixToIntRgba(tFixRgba);         returns the integer part of a tFixRgba value
```

11.4.3.1 Gouraud-Shading

Gouraud Shading along a span is one underlying algorithm of the PRE:

```
  Pxl.Rgba = FixToIntRgba(Inter.Rgba);
Mem.pRgba = Pxl.Rgba;
Mem.pRgba = Mem.pRgba + Mem.dpRgba;
Inter.Rgba = Inter.Rgba + Inter.dRgba;
```

This basic algorithm uses the pixel generation unit and the write unit.

The pixel generation unit consists of three interpolators which work in parallel. The components r, g, b and α as well as dr, dg, db and dα are represented in a fix-point

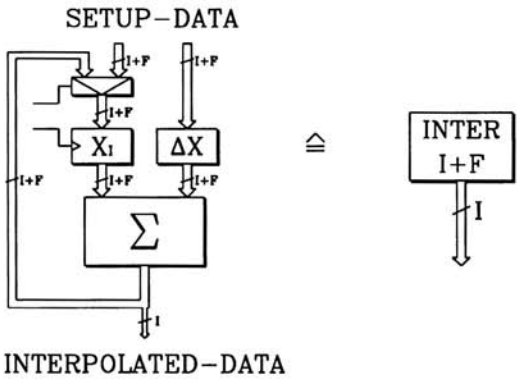

FIGURE 11.2. Block Diagram of a General (I+F)-Interpolator

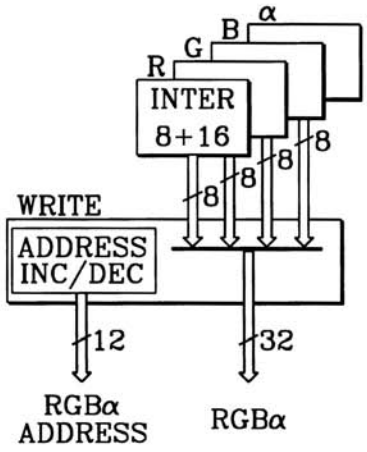

FIGURE 11.3. Block Diagram of the Gouraud-Shading Block

format. The integer part is 8 bits wide, the fractional part 16 bits. This is sufficient to avoid rounding errors with regard to the maximum screen resolution of 2048 by 2048 pixels. Once initialized with a starting value and an increment value, an interpolator produces an 8 bit integer result every clock cycle. We will call such an interpolator a (8+16)-interpolator, for short. The block diagram for a general (I+F)-interpolator is shown in Figure 11.2.

The write unit must be able to write a 32 bit word each clock cycle. For reasons of generality and to support advanced object shading algorithms, we will support to write in two directions (drgba = 1 or drgba = -1). If pipelined operation of the interpolators and write unit is assumed as well as single cycle propagation delay for each block, one

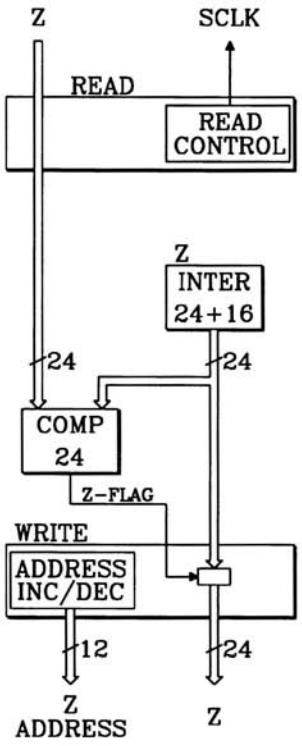

FIGURE 11.4. Block Diagram of the Z-Unit

Gouraud-shaded pixel can be delivered in each clock cycle. The pipeline length is two.

11.4.3.2 Z-Buffering

Z-buffering consists of the following steps: extracting the integer part of the pixel's actual z-value out of the interpolator, reading the buffer's actual z-value from the z-buffer, comparing both z-values, and writing back the pixel's z- value if necessary. Then the address for the next z-buffer access is calculated and the z-value for the next pixel is interpolated. This algorithm looks as follows:

```
  Pxl.Z = FixToIntZ(Inter.Z);
Buf.Z = *Mem.pZ;
if (Pxl.Z > Buf.Z)
Mem.pZ = Pxl.Z;
Mem.pZ = Mem.pZ + Mem.dpZ
Inter.Z = Inter.Z + Inter.dZ;
```

This basic algorithm already makes use of all the conceptual functional blocks of the

PRE. In addition to an interpolator and a write unit, a comparator and a read unit are necessary.

The read unit realizes a 24 bit interface to read a z-value which is stored in the z-buffer as an integer. The read is done in a sequential order with an increment of 1 or -1.

The pixel-generation unit implements a (24+16)-interpolator. The design is similar to the one of the rgbα-interpolators.

The comparator unit compares two 24 bit words. The result is the z-flag indicating whether the first value is greater than the second one. In this case, the interpolated z-value (as well as rgbα-values) of the pixel must be written to the z-buffer by the write unit.

This basic algorithm then leads to the block diagram shown in Figure 11.4. As mentioned above, it is necessary to read and to write a z-value at the same time. This requires a dual ported memory. The z-buffer will be realized with VRAMs using the random port as write port.

11.4.3.3 Texture-Mapping

Doing texture-mapping, the color of a pixel is determined indirectly. The vertex data are interpreted as addresses that point into a texture plane or texture space. These addresses are evaluated per pixel and the corresponding texture value is read. In our approach, we only support point sampling and no filtering yet. Textures are assumed to be toroidally closed. Both 2D and 3D texture-mappings are supported. The texture is supposed to fit into 64K words. In the 2D case, textures may be up to 256 x 256, in the 3D case, up to 32 x 32 x 32 texels. The access to texels is random. Therefore, a random address generator is required. The x,y,z-components of the texel addresses are assumed to be (8+16). During simple texture-mapping, no modification of rgba-values is done. The algorithm uses the texture-read unit and the write unit only. The texture logic can be realized using three (8+16)-interpolators and a texture address generator. This texture address generator calculates the final texture address by merging the three 8 bit outputs of the interpolators and a 16 bit base address to a physical 16 bit memory address. The write unit moves the rgbα-values, read from the texture-buffer to the frame-buffer. The write unit is the same one for any write access to the frame- buffer. The matching algorithm is given below:

```
  Mem.Txt.Addr = FixToIntRgba(Mem.Txt.Par)
Mem.Txt.Offset = (((Mem.Txt.z & (1 ≪ Mem.Txt.Dim.z)-1)
≪ Mem.Txt.Dim.y) + (Mem.Txt.y & (1 ≪ Mem.Txt.Dim.y)-1))
≪ Mem.Txt.Dim.x)) + (Mem.Txt.x & (1 ≪ Mem.Txt.Dim.x)-1)
Mem.Txt.pTxt = Mem.Txt.pBase + Mem.Txt.Offset
Pxl.Rgba = *Mem.Txt.pTxt
Mem.pRgba = Pxl.Rgba;
Mem.pRgba = Mem.pRgba + Mem.dpRgba
Mem.Txt.Par = Mem.Txt.Par + Mem.Txt.dPar;
```

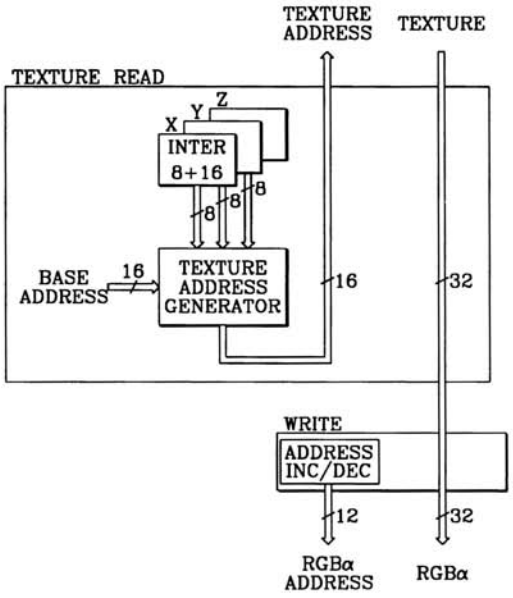

FIGURE 11.5. Block Diagram of the Basic Texture Logic

11.4.3.4 The Basic Design

Combining the three basic algorithms, we get the following basic design shown in Figure 11.6. This design implements Gouraud-shading or texture-mapping and z- buffering in one cycle.

11.4.4 Complex Algorithms

Up to now, we have just considered basic algorithms to calculate pixel values. In this section, we will combine these blocks to do more complex pixel calculations.

11.4.4.1 Alpha-Blending

Alpha-blending is the most complex basic algorithm of the PRE. It essentially consists of the blending of two pixels, Front and Back, controlled by the 8 bit α-value of Front. One of the pixels is interpolated, while the other one is read from the buffer. The decision is made depending on the z-value. For a detailed explanation, see [6].

The algorithm looks as follows:

```
Pxl.Z = FixToIntZ(Inter.Z)
Buf.Z = *Mem.pZ;
Pxl.Rgba = FixToIntRgba(Inter.Rgba)
```

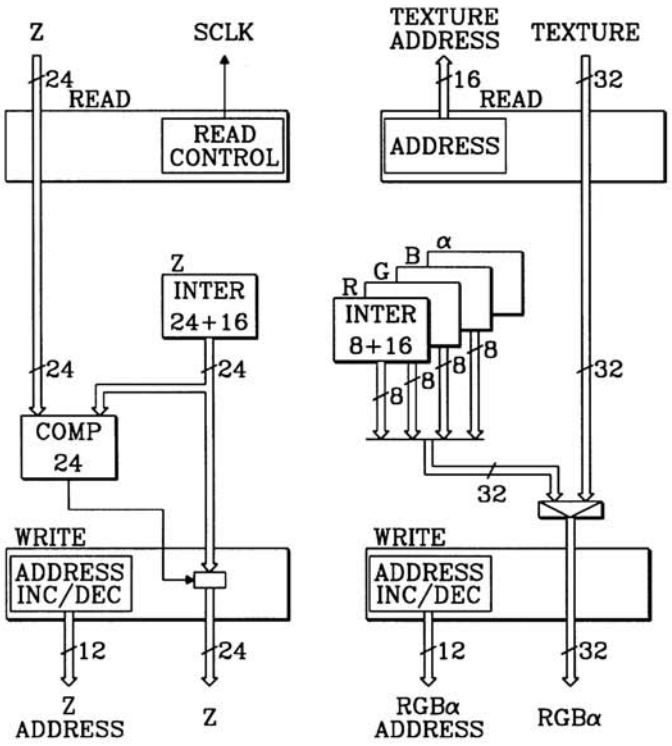

FIGURE 11.6. Basic Design

```
Buf.Rgba = *Mem.pRgba;
if (Pxl.Z > Buf.Z)
{
Mem.pZ = Pxl.Z;
Blend.Front = Pxl.Rgba;
Blend.Back = Buf.Rgba;
}
else
{
Blend.Front = Buf.Rgba;
Blend.Back = Pxl.Rgba;
}
Blend.Rgba = Blend.Front + (1-Blend.Front.a) * Blend.Back;
Mem.pRgba = Blend.Rgba;
Mem.pRgba = Mem.pRgba + Mem.dpRgba;
Mem.pZ = Mem.pZ + Mem.dpZ;
```

Inter.Rgba = Inter.Rgba + Inter.dRgba;
Inter.Z = Inter.Z + Inter.dZ

This algorithm makes use of all the building blocks of the pixel processor. The read unit reads a z-value in each cycle as well as an rgbα-value. The interface to the z-buffer has already been discussed (see 4.1.2). The rgbα-values are read in a strictly sequential order and 32-bit wide. Within the pixel generation unit, one rgbα-value is interpolated in parallel by four (8+16)-interpolators. This allows to interpolate the α-value across the span, which is not commonly done, but may be useful. The two rgbα-values are merged in the pixel blending stage in the next step. This stage consists of an adder, a multiplier, and an α-logic. The adder is 4x8 bit wide. The four multipliers multiply an 8 bit integer r,g,b or α-value by an 8 bit fractional 1- αFront-value each. The results are 8 bit integer r,g,b or α-values which are propagated to the next stage. The α-logic provides Blend.Front, Blend.Back and 1-Blend.Front.a. The length of the resulting pipeline for alpha-blending is five.

11.4.4.2 Texture-Modulation

Another advanced algorithm is texture- modulation. Textures may not only be used for mapping, but also to modulate a basic color or vice versa. The modulation may be done either by addition or by multiplication. In any case, it is an operation on the integer part of the color values only. To achieve texture-modulation, the stage shown in Figure 11.8 is included. This architecture again allows a single cycle output assuming that all building blocks produce a result in each cycle.

11.4.5 Programmability and Advanced Algorithms

Up to now, the design has fulfilled the initial requirements of being able to support single cycle execution of the basic shading algorithms. However, the resulting architecture is fairly general and allows, with just subtle extensions, the realization of wider ranges of algorithms.

11.4.5.1 Separation of Diffuse and Specular Color

During Gouraud shading, usually the final color value (sum of the diffuse and specular components) is calculated at the vertices and this color value is interpolated across the interior

of the primitive. While this works fine for objects with one intrinsic color only, it will lead to rather strange results when applied to texture mapped, transparent objects. Texture-mapping is conceptually a modification of the object's diffuse color. If first the resulting color is calculated and then this color is modulated with a texture, even the highlights seem to be 'textured', leading to a rather strange appearance. The following correct formula should be applied instead:

Pxl.Rgba = Diff.Rgba * Txt.Rgba + Spec.Rgba

If, however, the texture is used to model the surface's roughness, it should only be applied to the specular component. Then, the formula should be:

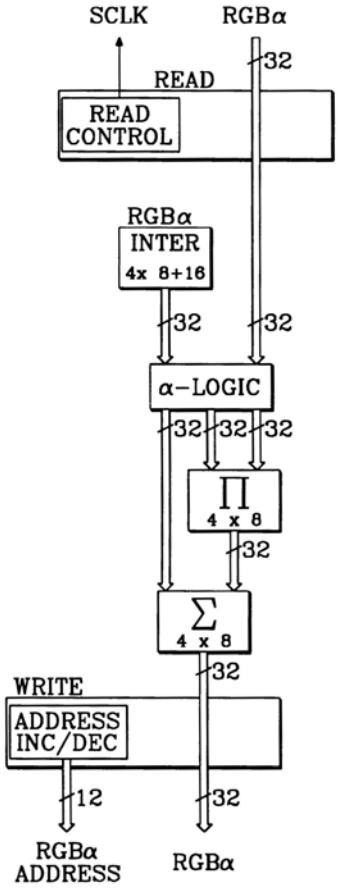

FIGURE 11.7. Block Diagram of the Alpha-Blending-Stage

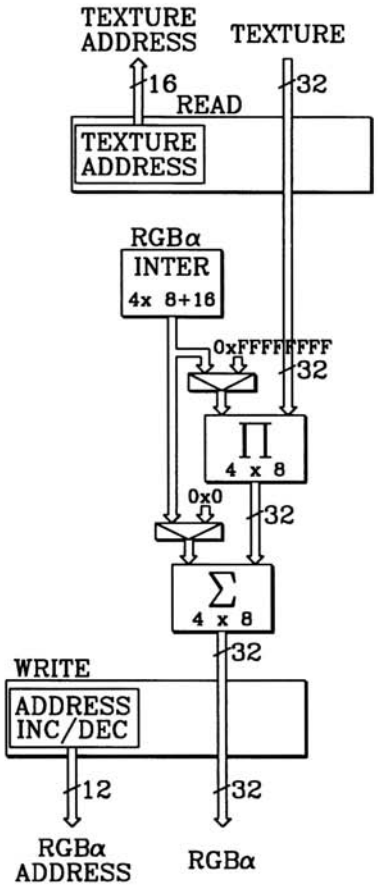

FIGURE 11.8. Texture-Modulation Stage

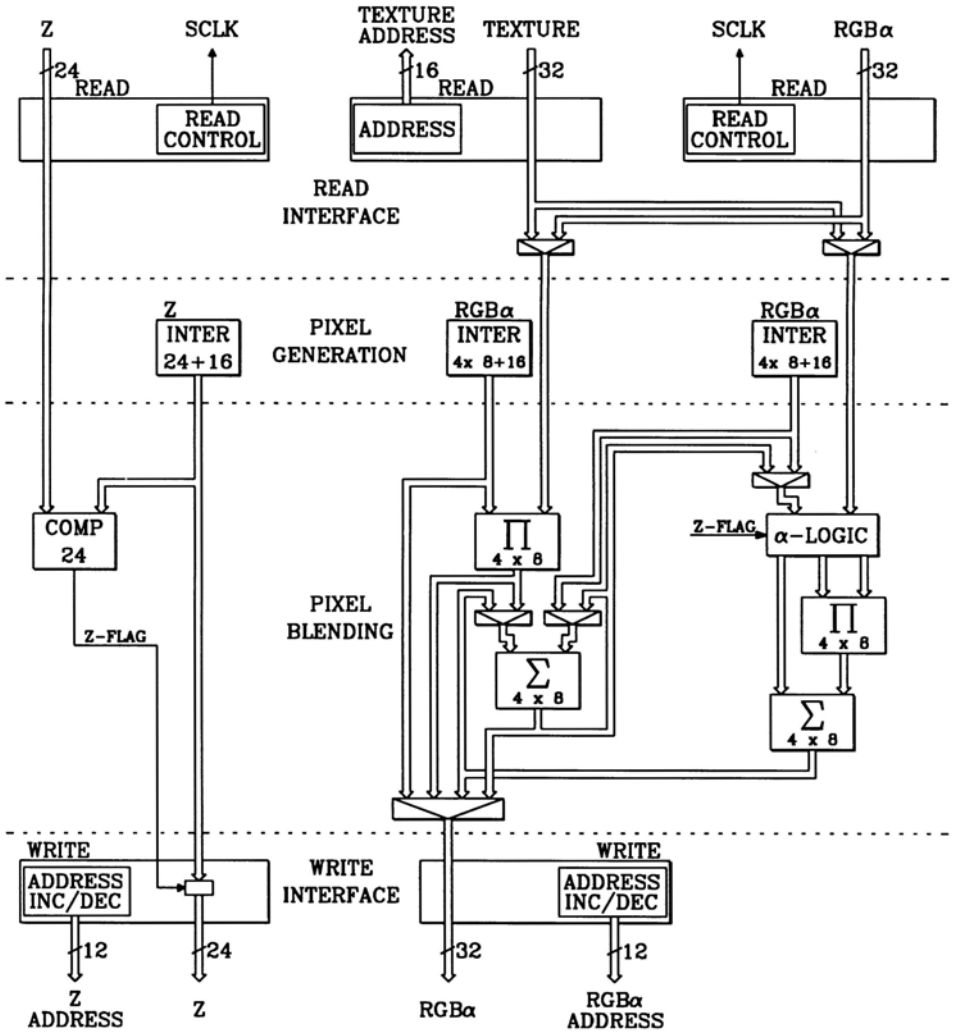

FIGURE 11.9. Final Design

Pxl.Rgba = Diff.Rgba + Spec.Rgba * Txt.Rgba

Similar arguments apply to transparent objects. A window, for example, is transparent, but has a clearly defined highlight. This can be realized only if we distinguish between diffuse and specular color. This advanced shading algorithms can easily be implemented by adding another interpolator for the second color components and connect it to the adder stage of the pixel blending stage.

11.4.5.2 Iterative Algorithms

To implement shade-trees, iterative shading algorithms must be supported. This requires both additional memory to store the results as well as additional busses to allow iteration. We want to implement such a feedback loop for each of the two internal stages of the pixel modification stage. This allows iterative pixel accumulation as well as iterative blending. The storage of results is done in FIFO registers which replace the initial value and increment registers of all interpolators. The depth of this FIFOs will be four. That means that a group of four parameter sets can be processed to iterate a final pixel value. In order to allow interpolating and initializing in parallel, there has to be a double set of FIFOs. Thus the final design of the pixel processing unit is shown in Figure 11.9. In case of using iterative algoritms, the time for generation of one result depends on the number of iterations. That means if the full iteration depth of four is used, a pixel can be produced every four clock cycles.

11.5 Expected Performance and Complexity

The simple assumption that the described pipeline of the pixel generation block delivers one pixel in each clock cycle implies, that even the slowest block of the pipeline requires less than one clock cycle to finish its operation. The critical blocks are obviously the multiplier units. As flash multipliers have been decided to consume to much of the available gate count, advanced Wallace Tree [7] multipliers will be used. Assuming the propagation delay times of an 1 μm CMOS process, a cycle time of 25ns is realistic. To assimilate the different propagation delay times of the different stages, the multipliers themselves could work in a pipelined fashion to increase the possible performance. But it does not make much sense to decrease the cycle time below the minimum required strobe pulse widths of the connected memory devices.

In any case, the system performance of the PRE will not only be determined by the achievable cycle time of the pixel processing block. As shown in [5], this performance, measured in primitives per second is determined by several other factors. The time necessary for transferring set-up data to the initial value and increment registers limits the maximum number of primitives per second. The time for one write cycle has as well been assumed to be 25ns. For simple Gouraud shading, eleven 64 bit words have to be transferred to the PRE. Even for complex iterative algorithms, this number may hardly exceed 25. On the other hand even a latest generation signal processor is not able to calculate the set-up data fast enough to keep the PRE working continuously when the area covered by the processed primitives decreases to a few pixels. An other determinant is the edge processing time. Unlike the Triangle Shading Engine [5], the PRE will provide an independent edge processing unit. It will be able to generate an edge value at least every two clock cycles. That limits drawing of vertical lines (which is the worst case operation) to 20 Mpixels per second. At last there is a loss of performance due to necessary memory refresh and control cycles. Projecting the performance of the PRE according to the performance of the Triangle Shading Engine leads to 100,000 up to 380,000 triangles (100 pixels, Gouraud-shaded, z-buffered, texture-modulated and alpha-blended), per second

and up to 1,000,000 Triangles (up to 8 pixels, Gouraud- shaded, z-buffered) per second, providing that set-up data can be accessed with maximum speed.

The complexity of the PRE can be estimated according to the components used in the different blocks. A detailed estimation can be given for the pixel processing block (estimation based on [7]):

Table 11.2: Estimation of Gate-Counts for the Pixel Processing Block

Component	quantity	Gate-equivalents
INTER 8+16	8	20000
INTER 24+16	1	4000
ADDER 8	8	1000
MULTIPLIER 8	8	10000
α-LOGIC	1	500
COMP 24	1	200
MUX 2:1	160	1000
MUX 4:1	32	300
Total		37000

Table 11.3: Estimation of Gate-Counts for the Chip

pixel processing	37000
edge processing	6000
memoryinterface	7000
controlunit	10000
Total	60000

The total gate-count will be about 60,000. This is about five times the complexity of the Triangle Shading Engine.

The approximate pin count will be 320 (Triangle Shading Engine 120 pins).

11.6 Future Work

The plan for the realization of the Programmable Rendering Engine has been started with the conception of the Pixel Processing Block. Depending on the available complexity of the process used for realization, functional blocks will be distributed to different chips. Future Work will concentrate on the other building blocks, their partitioning and realization of one or a set of chips.

11.7 References

[1] Kirk, D., Voorhies, D.: The rendering Architecture of the DN10000VS, *Computer Graphics* 24(4), August 1990 pp. 299-307

[2] Hanrahan, P., Lawson, J.: A Language for Shading and Lighting Calculations, *Computer Graphics* 24(4), August 1990, pp. 289-298

[3] Abram, G.,D., Whitted, T.: Building Block Shaders, *Computer Graphics* 24(4), August 1990, pp. 283-288

[4] Cook, R., L.: Shade Trees, *Computer Graphics* 18(3), 1984, pp. 223-231

[5] Ackermann, H.-J., Hornung, Ch.: The Triangle Shading Engine, In: R.L.Grimsdale, A. Kaufman (Eds.):*Advances in Computer Graphics Hardware V*,EurographicSeminars. Springer-Verlag, Berlin, 1991, p.3-13.

[6] Foley, J., van Dam, A.,Feiner, S. Hughes, J.: *Computer Graphics Principles and Practice*, second edition, Addison-Wesley, 1990

[7] Spaniol, O.: *Arithmetik in Rechenanlagen*, B.G. Teubner, Stuttgart, 1976

[8] Matra Harris: Databook *ASIC*, MHS, May 1989