Implementing a Programmable Pixel Pipeline in FPGAs

Péter Szántó, Béla Fehér

Department of Measurement and Information Systems
Budapest University of Technology and Economics
H-1117 Budapest, Magyar Tudósok krt. 2.
szanto@mit.bme.hu, feher@mit.bme.hu

Abstract.
Complex three dimensional graphics rendering is computationally very intensive process, so even the newest microprocessors cannot handle more complicated scenes in real time. Therefore to produce realistic rendering, hardware solutions are required. This paper discusses an FPGA implementation which supports programmable pixel computing.

1. Introduction

Advanced 3D rendering is computationally a very intensive task, so that even the highest end CPUs cannot do it in real time, therefore dedicated hardware is required. Our thought is a mobile device (phone, PDA) which can do a lot of different tasks, eg. decompression of motion pictures and/or music files or 3D gaming. As most of these functions are not needed simultaneously, it is logical to use a shared hardware to realize them. On the other side, these require quite different algorithms and a lot of processing power – therefore an FPGA may be a good solution. At power up it can be configured to attend basic, frequent tasks; however when the user wants to play a game, the FPGA can be configured as a 3D accelerator.

The implemented graphics accelerator supports hidden surface removal and shading in hardware. Transformation from local model coordinate system into the screen space's coordinate system, clipping and vertex based lighting must be computed by the host processor.
The hidden surface removal (HSR) part is based on the idea originally created by PowerVR: tile based deferred rendering [6.]. This method ensures that no processing power is wasted on computing non-visible pixels, and also saves memory bandwidth.
The shading part follows Microsoft's Pixel Shader 1.4 specification, which – among others – defines the instruction set and computation precision to support. This compliancy guarantees that complex shading algorithms created for current desktop graphics chips can be used without any modification. Version 1.4 is the most advanced specification which does not require floating point arithmetic during the color computing, but it must be noted that DirectX9 and OpenGL 2.0 supports more advanced shading capabilities.
To maintain high picture quality, anisotropic filtering is supported, while color computing is done with 16 bit precision per channel (64 bit per pixel).
The current implementation does not support edge anti aliasing, a feature which is very important for small displays – like the ones in handheld devices – but the architecture can be further developed to implement this without major modifications.

This paper concentrate on the shading part of the system, however to understand the whole functionality, the hidden surface removal unit is also presented briefly.
The specific implementation details of the arithmetic units – such as floating and fix point adders, multipliers and divisor – are not presented here.

2. Implementation

Figure 1. shows the complete block diagram of the implemented features.
Most of the blocks will be discussed later in details, with the exception of data storage elements.
Most blocks on the left side of the diagram represents external memories.
Vertex Data Memory stores all information associated to triangles and vertices, such as transformed coordinates, the number of textures assigned to triangles, texture coordinates and so on.
Texture Memory stores textures in a mipmapped format, as shown on Figure 4.
Frame Buffer stores one final output color for every screen pixel with 8 bit resolution per channel.
There are also intermediate storage elements stored in on-chip Block RAMs.
The *Z-buffer* stores Z (depth) values for the pixels in the tile which is under visibility testing process.
The *Triangle Index Buffer* stores a pointer for the same pixels, which defines the visible triangle on that location.
Triangle Data Cache is also an on-chip buffer storing values required during the shading process. These are the coefficients of the values to be linearly interpolated – texture coordinates, RHW and color data.

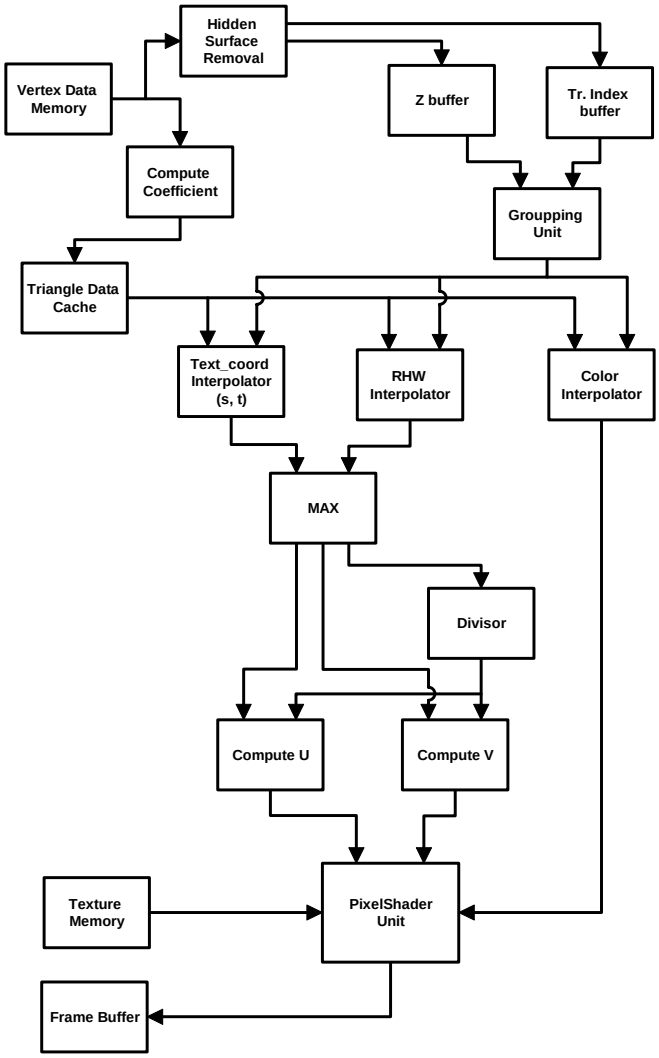

Figure 1. *Complete Block Diagram*

2.1 Hidden Surface Removal

Traditional architectures ([3.], [7.]) work triangle by triangle: after a triangle is transformed, they compute an output color and a Z value for every screen pixel covered by the actual triangle. Then the Z value is compared against the value stored in the Z-buffer (a buffer in external memory that stores the Z value of the actual visible triangle for every screen pixel), and if it is less, then that the color value is written into the frame buffer, and the Z buffer is updated with the actual value. There are two problems with this method: first, a lot of work is wasted on computing non visible pixels, and second, it needs a lot of memory bandwidth due to Z-buffer reads and writes (however with optimizations such as early Z check a lot of unnecessary computation can be eliminated).

Our implementation works a bit differently: it waits all triangles to be transformed, and then starts the rendering process by dividing the screen into tiles [6.]. A tile is small enough (currently 32x4 pixels) to store its Z-buffer and (temporary) frame buffer in on chip memory. Tiles have an Index Buffer associated to them, which stores a pointer for every pixel pointing to the triangle visible there. Rendering is done tile by tile, and starts with visibility testing: all triangles in the actual tile are processed to determine the visible one for every tile pixel. During this process, the dual-port internal Z-buffer is read in every clock cycle, and updated if necessary. When the Z-Buffer is written, the Index Buffer is also updated with a pointer of the actual triangle. Output computing is only done once for every pixel using the values generated by the Hidden Surface Removal Unit (HSR): a triangle index for the visible triangle, and the Z value.

Unlike traditional architectures, where a triangle is processed once – so all triangle data are read from the external memory once – our implementation works on a tile basis. As the pixels in a given tile covered by the same triangle may not be neighboring, it is not an option to step through the tile, and render pixels one by one. Instead, the HSR is followed by a Groupping Unit (GU) which collects the tile pixels covered by the same triangle into groups to minimize multiple triangle data reads from the external memory. Another problem is that a triangle may cover pixels in more than one tile, so triangle data must be read for every such tile.
The output of the GU is a memory containing triangle pointers and *x*, *y* coordinates.

2.2 Shading pipeline

The actual rendering starts after the GU. The shading pipeline consists of two main blocks: the first interpolates the color components, texture addresses and RHW coordinate (reciprocal homogeneous W, computed at the transformation stage) for the actual pixel, while the second (Pixel Shader) computes output color values.

Due to the non-linear nature of perspective correct projection (the transformation that translates three dimensional camera space into screen space), texture coordinates do not change linearly with screen space *x* and *y* coordinates (they do so in camera space), so to implement perspective correct texture mapping, another set of coordinates are used for linear interpolation: *s* and *t*. First, the *s* and *t* are computed for all vertices with the following formula:

$$s = u * RHW \qquad t = v * RHW \tag{1}$$

These can be linearly interpolated using the screen's *x*, *y* coordinates. Interpolation is done using the coefficients of the plane equation for all values (*k* represents any value to be interpolated, while *x* and *y* are vertex coordinates):

$$\begin{aligned} A_k &= (y_{s1} - y_{s2}) * (k_{s1} - k_{s0}) - (k_{s1} - k_{s2}) * (y_{s1} - y_{s0}) \\ B_k &= (k_{s1} - k_{s2}) * (x_{s1} - x_{s0}) - (x_{s1} - x_{s2}) * (k_{s1} - k_{s0}) \\ C_k &= (x_{s1} - x_{s2}) * (y_{s1} - y_{s0}) - (y_{s1} - y_{s2}) * (x_{s1} - x_{s0}) \\ D_k &= -(A_z * x_1 + B_z * y_1 + C_z * k_1) \end{aligned} \tag{2}$$

Interpolated values can be then computed using:

$$k_i = \frac{(-A_k * x) + (-B_k * y) + (-D_k)}{C_k} = E_k * x + F_k * y + G_k \tag{3}$$

E, *F* and *G* values are computed in hardware during triangle data reads, while *1/C* is computed by the host processor (this division must be done only once per triangle, as *C* only depends on the *x* and *y* coordinates, but is independent from the value to be interpolated).

However, to address textures, *u* and *v* texture coordinates are needed, so the interpolated *s* and *t* must be divided with the interpolated RHW value. The Divisor unit is based on an iteration algorithm with ROM based starting values.

CubeMap textures require a bit different addressing method than simple 2D textures. They are addressed using an interpolated 3D vector (*R*). For this purpose, texture coordinate and RHW interpolators are used. The component with the maximum absolute value defines the CubeMap face to use (hence the MAX unit in the block diagram), while texture addresses within that face are computed using the other two components and the reciprocal of the maximal component (division is done in the RHW division unit), as shown in (4) and Table 1.

$$u = \frac{\frac{uc}{|m|} + 1}{2}, v = \frac{\frac{vc}{|m|} + 1}{2} \tag{4}$$

R component with maximum amplitude (m)	uc	vc
R_x	$-R_z$	$-R_y$
$-R_x$	R_z	$-R_y$
R_y	R_x	R_z
$-R_y$	R_x	$-R_z$
R_z	R_x	$-R_y$
$-R_z$	$-R_x$	$-R_y$

Table 1. *Addressing Cube Map with a 3D vector*

The interpolated texture and color coordinates are fed into the Pixel Shader unit, which computes the output color.

For *s*, *t* and RHW interpolation, IEEE single precision floating point arithmetic is used unless it is absolutely unnecessary. The multipliers in (3) are mixed inputs units: the pixel coordinates are integers, while the coefficients are floating point values. The outputs of the units are normalized 32 bit floats. Multiplication takes three clock cycles to execute, and the results are fed into a three input floating point adder, which takes five cycles to compute the result. Division (more exactly reciprocal computing) is also done using floating point values; however the mantissa is a bit less precise, but still requires nine clock cycles. The "Compute_U" and "Compute_V" units are floating point multipliers with fixed point output, as the results are used to address textures in the memory.

Pixel Shader Architecture

A Pixel Shader Unit (PS) is responsible for texture sampling and color computing. The following registers are the minimum specifications:

- 8 constant registers (read only)
- 6 temporary registers (read/write)
- 6 texture coordinate registers (read only)
- 2 color registers (read only)

Instructions are divided into two groups: texture addressing and arithmetic instructions. The four texture addressing instructions can use the texture coordinate registers (for texture sampling, or for copying for further use, as color data). For data modification, there are numerous arithmetic instructions with wide range of modification possibilities. Figure 2. shows the logical instruction flow of PS 1.4:

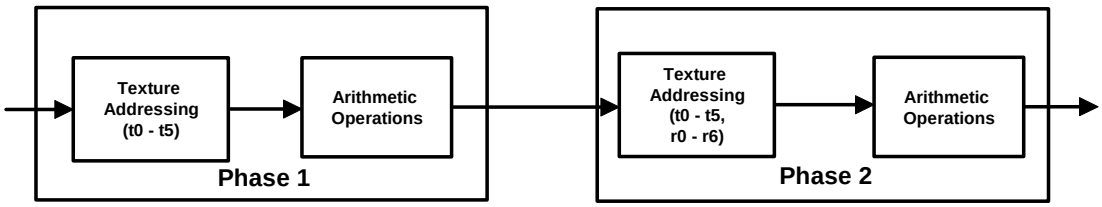

Figure 2. *Pixel Shader 1.4 Instruction Flow*

In the first phase Texture addressing instructions can only use texture coordinate registers (t) as an input (these values are computed by the interpolation pipeline). After texture sampling is done, arithmetic operators can modify the values.
In the second phase, texture sampling is also possible using values computed in the first phase (so dependant texture reads can be implemented).
Another key point is that texture coordinates and textures are not assigned permanently, so any of the coordinates can be used to address any of the textures assigned to the given triangle.

Pixel Shader Texture Addressing unit

The fact, that texture sampling is possible with values computed in the Pixel Shader makes mipmap level computing a little complicated. For determining mipmap level, the area covered by the screen pixel in texture space is required – so not only the pixel center's texture coordinates must be computed, but also pixel's corner points. The most straightforward solution is to perform the first phase instructions not only at pixel centers, but also at the corners. This method, however, requires at least three more PS units, and also increases texture memory reads.
Therefore our implementation follows a different way. Instead of computing mipmap level for every pixel, 2x2 pixels blocks are used, as shown on Figure 3. Darker vectors represent the values used by correct per pixel mipmapping, while brighter vectors are the ones used by our implementation.

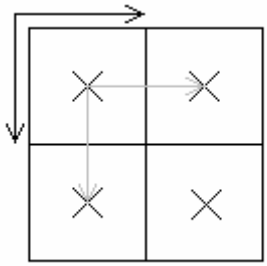

Figure 3. *Mipmap level computing*

This method involves that first phase instructions must be executed on four pixels even if only one of them is actually covered by the given triangle.
Beside this disadvantage there are also benefits. First, the same mipmap level computing can be used for texture coordinates which are computed in the interpolation pipeline, so only one divisor and simpler interpolators are needed. Second, as it takes four cycles for the PS unit to execute the same instruction on the four pixels, the PS unit can be pipelined (otherwise this cannot be easily done because instructions may depend on the previous results).

Final texture coordinates are stored in an 18 bit wide register to allow using 4096*4096 size textures, and still have enough precision to do bilinear filtering. The current implementation supports basic 4D Pyramid Anisotropic Filtering. The texture coordinates computed by the interpolator pipeline are translated into real addresses according to the mipmap level and the size of the texture being addressed.

The following steps are required during this process:

- Computing the maximal u and v differences using texture coordinates at the four neighboring pixels
- Determine the appropriate mipmap level to use independently in both u and v directions
- Computing final memory addresses according to the mipmap level, u and v, and texture size
- Sample the necessary texels and apply bilinear filtering

Textures are stored in texture memory with their appropriate mipmap levels, just as Figure 4. shows.

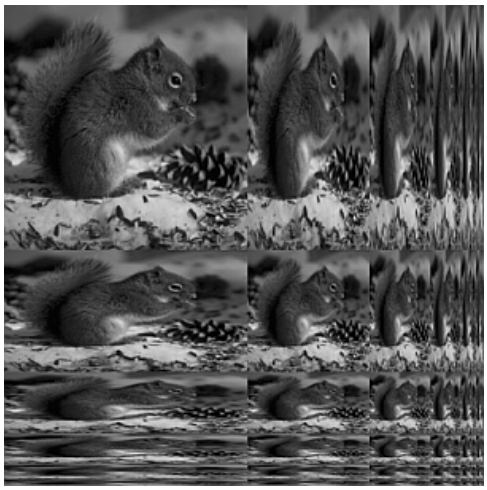

Figure 4. *Mipmapped texture*

In theory, address inside the texture can be computed using the following equation (shown only for the u corrdinate):

$$get\_u = \left(\sum_{k=0}^{(Mip\_Level\_U-1)} 2^{text\_size\_u-k}\right) + ((u << text\_size\_u) >> Mip\_Level\_U), \quad (4)$$

where text_size_u is two based logarithmic of real texture size in pixels. Note, that this method requires textures to be power-of-two sized. The second part ("in-mipmap-level address") is straightforward to implement in hardware, as it only requires shifting. This part is computed with 18 bit precision, the upper 12 bits are for addressing, while the lower 6 bits are for computing bilinear filtered results.

The 12 bit wide result is modified with the first part summa, which is also implemented using shift registers and a 13 bit constant (all bits are one). The first result is generated by shifting the constant left with (text_size+1), and the second is generated by shifting the constant right with (12-text_size+Mip_level). The final result is then computed from the two shifted values and the constant with bitwise XOR.

For example with a 512 pixel wide texture when reading from the second mipmap level, text_size is 9, while 12-text_size+Mip_level is 5, so the three registers contain:

$$FULL = 1111111111111$$

$$\mathrm{mod}\,0 = 1110000000000$$

$$\mathrm{mod}1 = 0000011111111$$

The result after the XOR operation is '0001100000000'=2^9+2^8, what is exactly the expected 768.

Figure 5. shows the block diagram of the Texture Addressing unit.

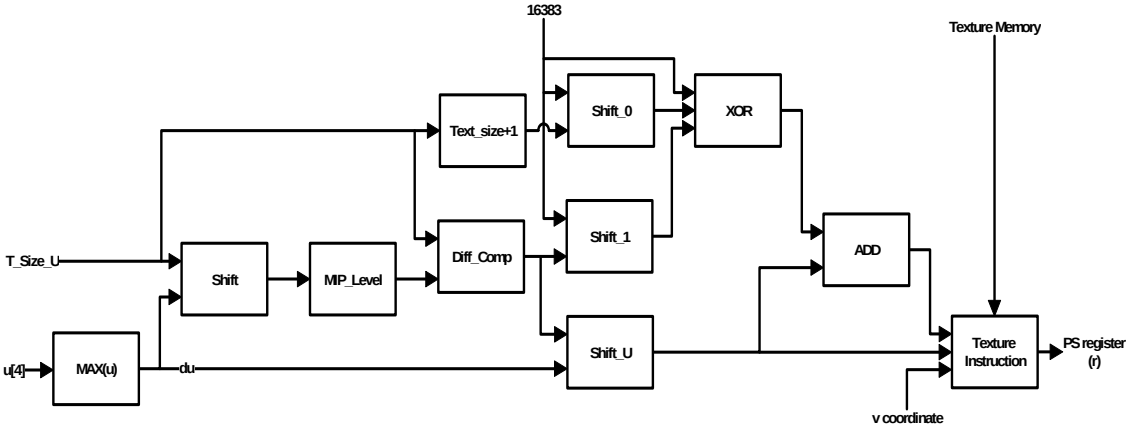

Figure 5. *Texture Addressing Unit block diagram*

Pixel Shader Arithmetic unit

The Pixel Shader (PS) unit is an arithmetic unit capable of executing specified instructions ([7.]). The unit works on four components: R(red), G(green), B(blue) and A(alpha). Figure 6. shows the data flow.

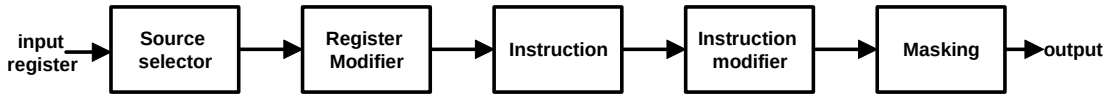

Figure 6. *Arithmetic instruction data flow*

Source Selector allows any of the source register components to be duplicated to the four parallel ALU channels.
Register Modifier modifies the data read from the register with basic modifier instructions (eg. bias, invert, negate, scale by 2), but does not change the source register's value.
Instruction executes the specified arithmetic instruction.
Instruction Modifier allows the result to be shifted left or right, or to be clamped from 0 and 1.
Masking selects the components to be updated in the output register.

The Register Modifier block is quite complicated, so Figure 7. shows its block diagram for one channel. As can be seen instruction selection is implemented by multiplexers. This unit has exactly the functionality PS 1.4 defines.

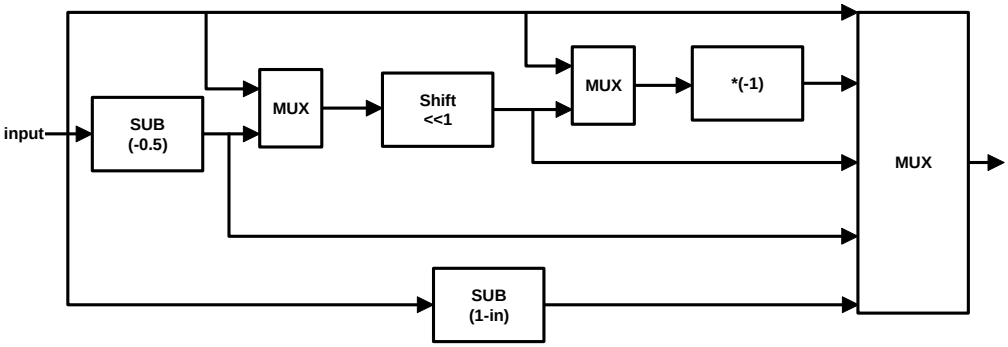

Figure 7. *Arithmetic Register Modifier*

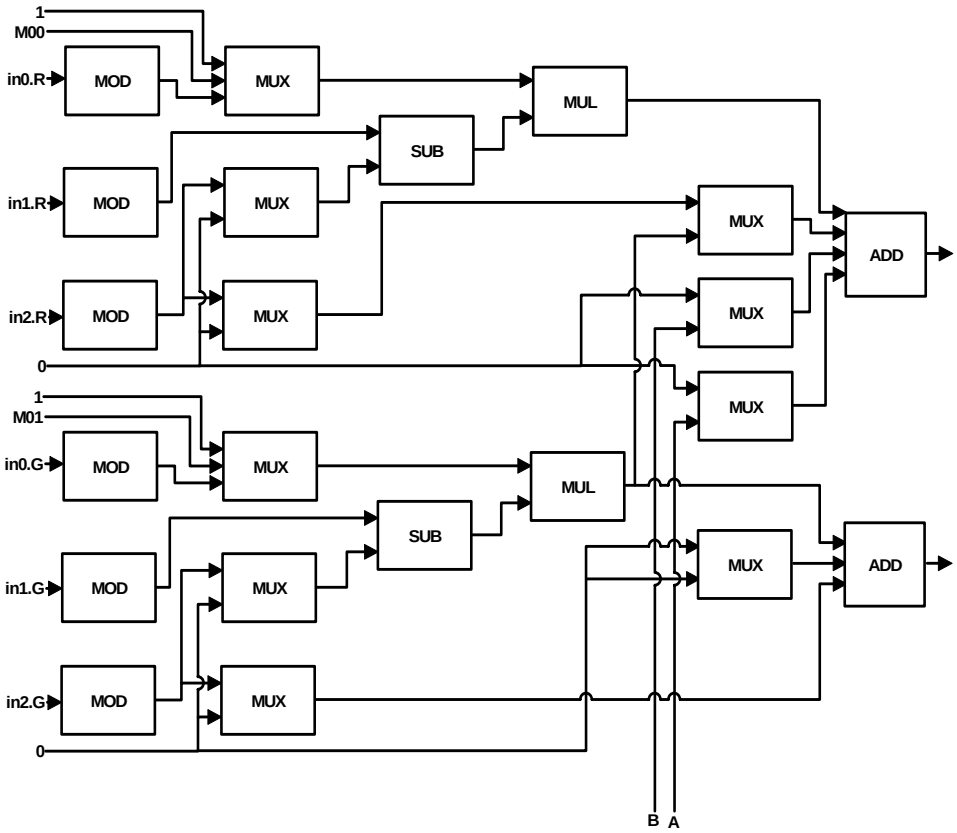

Figure 8. *Arithmetic Block*

The main unit in the Arithmetic block, which executes the instruction, is also programmed using multiplexers. The block diagram on Figure 8. shows only the R and G channels. B and A channels are almost identical; the only difference is that there are no multiplexers before the adder. For the R and G channels, these multiplexers are needed to be able to execute three- and four-component dot product in one cycle. To retain perspicuity, the blocks executing comparison instructions were removed.

The blocks on Figure 8. have the following functionality:

- MOD: Source Modifier (Figure 7.)
- MUL: Multiplication unit
- SUB: Subtraction unit
- ADD: Adder units

As mentioned earlier, both the Texture Addressing and the Arithmetic blocks are pipelined to achieve higher clock speeds.
The two main blocks (Texture Addressing and Arithmetic) work in parallel. However, as PS assembly programs follow exactly the instruction flow Figure 2. shows, parallelization must done while generating the byte code from the PS program.

4. Development tools

The implementation was not done using one of the most common hardware description languages (such as Verilog or VHDL), but an ANSI-C based tool, Celoxica's DK1 Design Suite.

The target hardware for our design was Celoxica RC1000 development board. This board has a Xilinx Virtex2000E FPGA, which has the necessary amount of logics and on-chip Block-RAM to suit our needs. The card itself is equipped with 8 Mbytes of SRAM, organized into four banks. Each bank can transfer 32 bit data in a clock, so a 128 bit memory interface is available. Memory can be accessed by the FPGA and by the host processor with DMA. Unfortunately, direct communication with the FPGA is somewhat limited, so one bank is always used by the host processor to upload triangle data.

5. Conclusion

The design proved that a 3D rendering hardware can be implemented in FPGAs without sacrificing image quality or limiting achievable effects.

On the other hand, there are clear disadvantages of the FPGA implementation. For such a complex design, a lot of hardware resources are needed, and the price of complex FPGAs may limit utilization of such systems. Achievable speed is far from current mainstream ASIC implementations, but for mobile devices this should not be a problem as low resolution displays need much less raw processing power.

References

1. Möller, Tomas, Haines, Eric: Real Time 3D Rendering, A. K. Peters Ltd., 2000.
2. Hecker, Chris: Perspective Texture Mapping, Game Developer Magazine, April, 1995. – April, 1996.
3. Watt, Alan: 3D Computer Graphics, Pearson Education Ltd./Addison-Wesley Publishing, 2000.
4. Dr. Szirmay-Kalos, László (editor), Theory of Three Dimensional Computer Graphics, Publishing House of the Hungarian Academy of Sciences, 1995
5. Styles, Henry, Luk, Wayne: Customizing Grpahics Applications: Techniques and Programming Interface, IEEE Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society Press, 2000.
6. Imagination Technologies, PowerVR Software Development Kit, Imagination Technologies, http://www.pvrdev.com
7. NVIDIA Corporation, NVIDIA Software Development Kit, NVIDIA Corporation, http://www.nvidia.com/developer
8. Microsoft, Microsoft DirectX9 Software Development Kit, Microsoft Corporation, http://msdn.microsoft.com