3D rendering using 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 complies with the newer, programmable standards. As the design implements a somewhat unique hidden surface removal algorithm originally created by PowerVR, the paper focuses on this part.

1. Introduction

One of the most rapidly developing filed in the hardware industry is 3D graphics rendering. Displaying lifelike virtual worlds has not spread only in workstations or PCs, but it starts to appear in mobile devices. From the starts, 3D algorithms try to model real world objects with models built up from triangles, however the simulation of lifelike, complex surfaces developed by huge margin.

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 (cell 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 an ideal 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 following basic steps are required to render a 3D scene ([1.], [2.], [4]):

- Transformation and lighting:
  - Tesselation
  - Transformation
  - Lighting
- Rasterization
  - Shading
  - Hidden surface removal

The first three steps are done for every vertex, while the latter two is done for every screen pixel.

Tesselation is only needed when the objects are not specified as a list of triangles, but other, higher order surfaces. In real-time rendering this function is still not widely supported.

Transformation transforms objects (actually their vertices) from their local coordinate system into camera space, and then applies perspective correct projection to move from 3D space into the screen's 2D coordinate system.

Vertex based *lighting* is applied during the transformation. For each vertex, two color components (diffuse and specular) are computed according to the light sources. A huge difference between real time and non real time rendering is that the former uses local illumination algorithms, that is, other objects do not have influence on the color of the current object.
Constant shading uses one vertex color on the entire triangle, while the more complicated Gouraud-shading interpolates between the vertices' colors to get the final result for the triangle's internal points.

The next step in traditional hardware architectures is *shading* (the computation of output pixel colors). This is one of the computationally most intensive parts of the rendering. In simple cases only the computed vertex colors are used, however much more complex surfaces can be simulated with texture mapping.

Hidden surface removal checks whether the rendered pixel of the actual triangle is visible or not. There are a lot of different algorithms (scan-line, list priority, Warnock, binary space partitioning), but hardware realizations exclusively use the Z-buffer method.

2. Texture Mapping

Before going into details, it is necessary to understand the basics of texture mapping ([2], [3], [4]), which is used to simulate detailed surfaces without too complex object geometry.

In the simplest case a texture is a two dimensional array representing the surface of an object. A single element in the texture map is called texel (texture element). Textures are assigned to triangles by defining the texture coordinates (commonly referred as *u* and *v*) at each vertex.
As perspective projection is not a linear transformation, a screen pixel transformed into texture space can have any shape defined with the four pixel corner points (Figure 1. shows the situation with the bounding rectangle grayed). The effect is that the area in the texture covered by the transformed pixel may be smaller or greater than an actual texel.

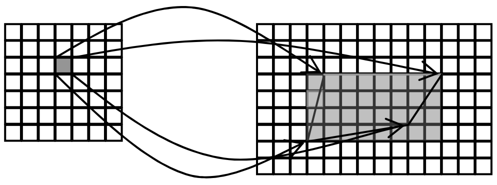

Figure 1. Screen pixel transformed into texture space

In the former case simple bilinear filtering is applied to reduce sampling errors.

The latter case is slightly more complicated. The most straightforward solution is to average all the texels that are covered by the transformed pixel. The only problem with this solution is speed – reading all the necessary texels from the external texture memory may require too much bandwidth or take too much clock cycles. To decrease the memory reads, pre-averaged versions of the original texture are stored. The algorithm is called Mipmapping, and the different resolution textures are called mipmap levels. The (i+1.) level has half the side size of the (i.) one, so a mipmapped texture exactly has 4/3 the size of the original texture (1 + 1/4 + 1/16 +…). The mipmap level used is computed from the bounding square (corner points are used to determine this) of the transformed pixel. If a pixel covers 8x8 texels size in the 0. mipmap level, it exactly covers one texel size in the 3. level. To eliminate shimmering with moving objects, bilinear filtering is applied. More advanced filtering algorithms use the bounding rectangle instead of the bounding square (our implementation uses 4D pyramid anisotropic filtering), and even allows the sides of this rectangle not to be parallel with the texture space's *u* and *v* axle.

Early hardware implementations supported only one texture for a triangle (single texturing), while the next generations supported more, however the supported blending modes for the textures were quite limited. Newest standards ([7.]) make texture combining a programmable process. The unit responsible for this task is called PixelShader, the implementation complies with version is 1.4 (this is the most advanced which does not require floating point color support).

3. Implementation

There are a lot of different implementations of 3D graphics algorithms in hardware. Desktop chips nowadays use floating point computation throughout the entire pipeline (and also accelerates transformation and lighting), and they have multiple pipelines working in parallel. Of course such processing power requires a lot of hardware resources (eg. ATI's R300 has more than 100million transistors) which is not available in FPGAs, but our goal was not to implement an FPGA based solution comparable to high-end desktop chips, but a design capable of accelerating advanced shading algorithms (a requirement earlier implementations ([5], [8], [9]) do not fulfill) and still fitting into an FPGA and having enough power for small displays can be found in handhelds (about 320x200).

The implemented unit supports hidden surface removal and shading in hardware. Figure 2. shows the schematic diagram of the complete implementation. The execution units will be discussed later, however there are also storage elements. The blocks on the left side represents external, off-chip memories, used to store vertex data, texture information and final output color values.

Traditional 3D rendering architectures 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 a Z value 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 it must be noted that with optimization such as early Z check a lot of unnecessary work can be eliminated).

Our implementation works a bit differently: it waits all triangles to be transformed, and then starts the rendering by dividing the screen into tiles [6]. A tile is small enough (actually 32x16 pixels) to store its Z buffer and frame buffer in on chip memory. Tiles have an index buffer associated to them, which stores a pointer to triangles which cover a pixel in that tile. 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. 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. The HSR unit will be discussed in detail later (section 3.1). The HSR is followed by a Grouping Unit (GU) which collects the tile pixels covered by the same triangle into groups to minimize multiple triangle data reads from the external memory during the shading.

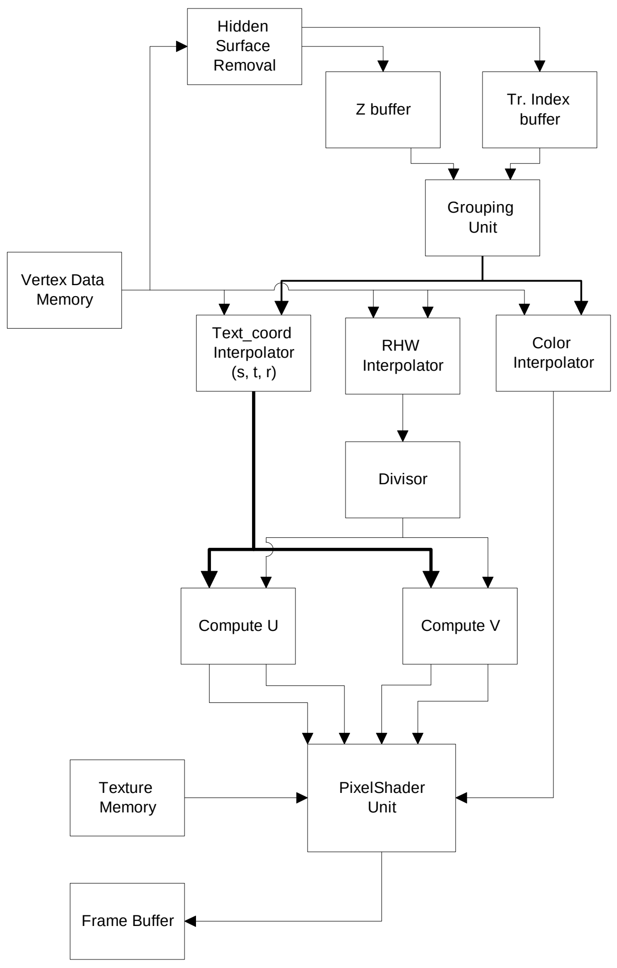

Figure 2. Complete schematic diagram

The actual shading process starts after the GU. The shading pipeline first interpolates the color components, texture addresses and RHW coordinate (reciprocal homogeneous W, computed at the transformation stage) for the actual pixel using the given values for the triangle vertices. Unfortunately texture coordinates do not change linearly with screen space *x* and *y* coordinates (they do so in camera space), so another coordinates are used: *s* and *t*. First, the *s* and *t* are computed for the vertices with the following formula:

$$s = u * RHW$$
$$t = v * RHW \qquad (1)$$

These can be linearly interpolated using the screen's *x*, *y* coordinates. However, to address textures, *u* and *v* must be computed, so the interpolated *s* and *t* must be divided with the interpolated RHW value. After the texture coordinates are determined, the PixelShader Unit computes the final output color.

On the schematic diagram a MAX unit is also present: this unit is used to implement Cubic Environment Mapping. This texture mapping technique requires an interpolated three dimensional vector and a divisor to divide two components of the vector with the maximal amplitude component. So, in case of cube map texture, our implementation defines the maximum component, properly arrange the other two (this means switching and/or negating) and performs the division.

To understand the following decisions, we have to look into PixelShader specification a bit.

A PixelShader program is divided into two phases (phase one and two). In both phases, texture addressing instructions are followed by arithmetic instructions. The difference is that in the first phase only *Texture coordinate registers* are available to address textures (these store the result of the interpolation process), while in the second *Temporary registers* may also be used – and these registers get their values during the execution of the first phase's arithmetic instructions. To determine which mipmap level to use, the covered texture area must be computed – so texture coordinates must be computed not just for the pixel center, but also for the pixel corners – this can be done with four times more PS units (one for the pixel center and another four for the corners), or in four more clock cycles.

To avoid this problem, the hardware does not compute mipmap level for every pixel, but for every 2x2 pixel block. The first phase instructions are executed on every pixel of such a block, and then mipmap levels are computed using the available data at the four pixel centers. This may degrade effectiveness if the pixels in the block are not covered by the same triangle, and make the Grouping Unit a bit more complicated, but the fact that only one PixelShader and divisor is needed compensates this.

3.1 The Hidden Surface Removal Unit

As mentioned in the general description, the HSR determines the visible triangle for every pixel in a given tile. To achieve this, it must decide whether a pixel is covered by the currently processed triangle, and in case it does, the Z value must be computed and compared with the one in the Z buffer. To make this process fast, 16 parallel units are used. Each unit works on one line of the tile (32 pixels), and has a double buffered, dual port Z buffer memory and a double buffered Index memory, just as Figure 3. shows. A tile is small enough to have these memories on-chip, so external memory bandwidth is not wasted during this process. Triangles are defined with vertices sorted by growing *y* coordinates.

Covering is determined using the coefficients of the triangle sides. The Compute_M0 computes the *x* coordinate for all three sides using the following formula:

$$x = A * y + B \qquad (2)$$

where *y* is the *y* coordinate of the top pixel line *A* and *B* are the coefficients defined with the appropriate screen space vertex coordinates (x_{s0}, x_{s1}, y_{s0}, y_{s1}).

$$A = \frac{x_{s0} - x_{s1}}{y_{s0} - y_{s1}} \qquad (3)$$
$$B = x_{s0} - A * y_{s0}$$

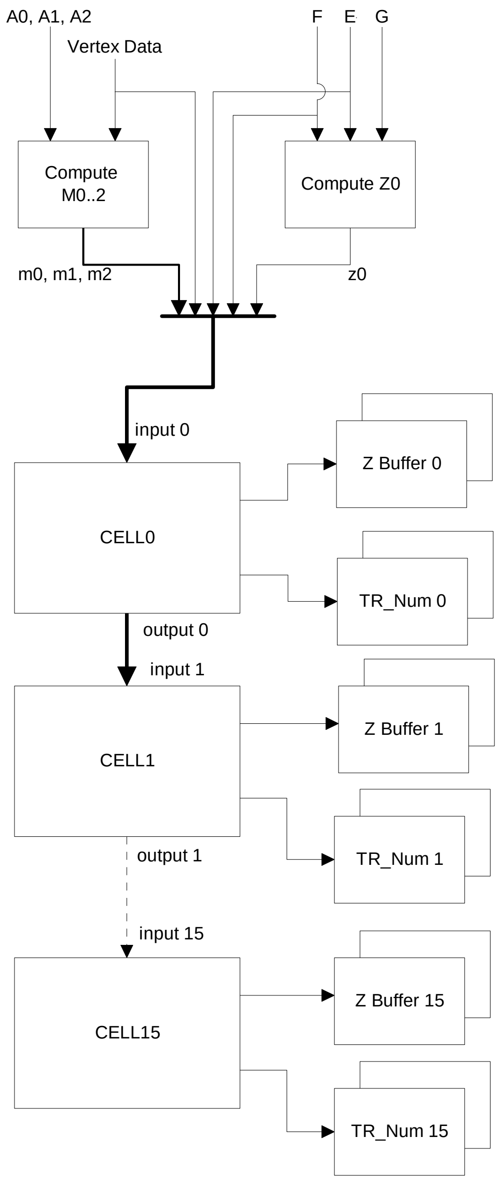

Figure 3. Hidden Surface Removal

For the Z value interpolation, the coefficients of the plane equation are used, defined by the formulas shown below (using screen space vertex coordinates):

$$A_z = (y_{s1} - y_{s2}) * (z_{s1} - z_{s0}) - (z_{s1} - z_{s2}) * (y_{s1} - y_{s0})$$
$$B_z = (z_{s1} - z_{s2}) * (x_{s1} - x_{s0}) - (x_{s1} - x_{s2}) * (z_{s1} - z_{s0}) \qquad (4)$$
$$C_z = (x_{s1} - x_{s2}) * (y_{s1} - y_{s0}) - (y_{s1} - y_{s2}) * (x_{s1} - x_{s0})$$
$$D_z = -(A_z * x_1 + B_z * y_1 + C_z * z_1)$$

$$E_z = -\frac{A_z}{C_z}$$
$$F_z = -\frac{B_z}{C_z} \qquad (5)$$
$$G_z = -\frac{D_z}{C_z}$$

A Z value is then calculated using

$$z = E_z * x + F_z * y + G_z \qquad (6)$$

expression. The current implementation does not compute these coefficients in hardware, the host has to do this.

For every pixel, only an addition (or subtraction) is needed to generate new Z value or intersection points. To decide which two of the three intersection points have to be used to determine covering, a mode_y value is generated. The definition of mode_y is shown on Figure 4.

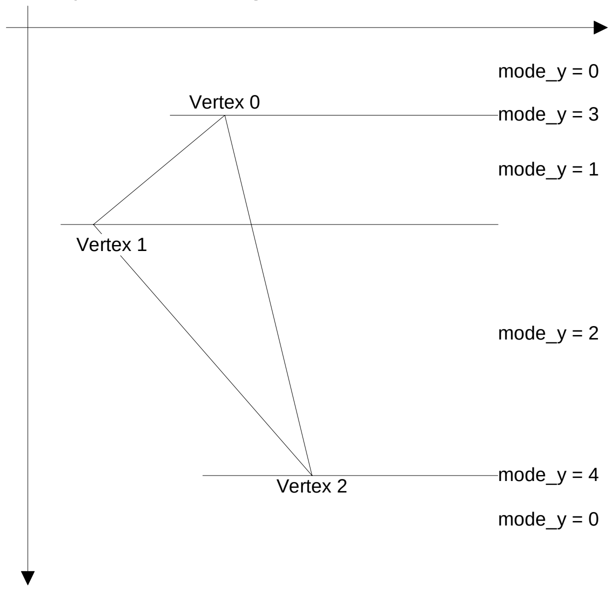

Figure 4. Mode_y definition

The meaning of Mode_y:

- Mode_y=0: this line is outside of the triangle
- Mode_y=1: the *y* coordinate is bigger than vertex0's *y* coordinate and smaller than or equal to vertex1's *y* coordinate. Intersection 0 and 1 should be used.
- Mode_y=2: the *y* coordinate is bigger than vertex1's *y* coordinate, and smaller than vertex2's *y* coordinate. Intersection 1 and 2 should be used.
- Mode_y=3: the *y* coordinate is equal to vertex0's *y* coordinate.
- Mode_y=4: the *y* coordinate is equal to vertex2's *y* coordinate.

For filling, top-left convention is used, that is a pixel is covered by a given triangle in the following cases:

- It is inside the triangle
- It is on the left edge of a triangle
- It is on the top edge of a triangle (top edge is a horizontal edge)
- It has the same coordinates as vertex0 or vertex2

Any incoming triangle has the M0, Z0, A and mode_y for the 0. tile line already computed (this is done during - in parallel with - the memory reads to get the vertex data).

In every clock cycle, all even numbered cells step one pixel right: increment the *x* coordinate and compute a new Z value (add A to the previous value). Odd numbered cells step backwards, so they decrement *x*, and subtract A to get a new Z value. The computed Z values then used for the comparison with the Z buffer; to be able to do the comparison and the possible Z writing in one cycle dual port memory is used. Parallel to this, every cell computes a new mode_y value for the next cell.

Figure 5. shows the schematic diagram of a Z cell.

After all 32 pixels in the given line are processed, the Z values are incremented with coefficient B and passed to the next cell. At the same time, the 0. cell reads the data of the next triangle to process. If there is no more triangle in the current tile, processing of the next tile starts immediately using the second Z and Index memories. In order to maximize operating frequency, all on-chip memory reads and writes take place in different clock cycles, therefore for one pixel the visibility testing requires three clock cycles, while the complete line is processed in 34 clocks.

Z values' fractional part is stored with 24 bit precision (this value is fractional inside a triangle), however the HSR unit also computes Z value when a pixel is not covered by the triangle, and therefore additional eight bits are used to be able to compute them for the whole tile.

Computation is done with fixed point arithmetic to minimize area requirement and make addition as fast as possible without too much pipeline stages.

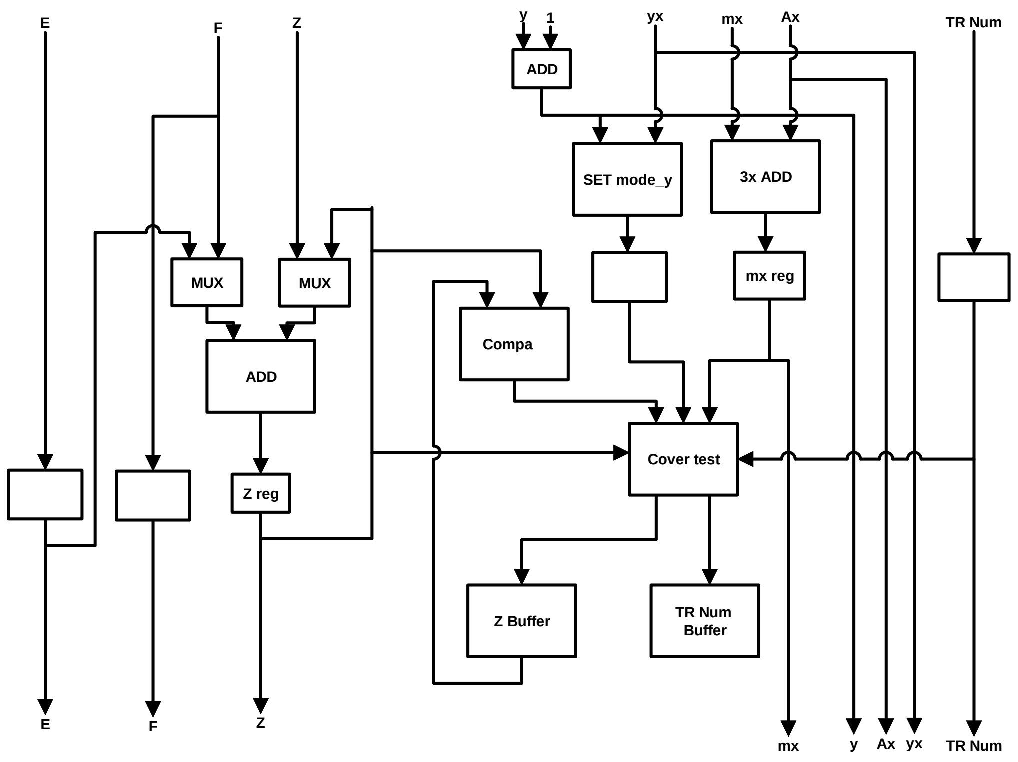

Figure 5. Z Cell schematic diagram

3.2 The shading pipeline

The first part of the shading pipeline (between the GU and the PS unit) works with IEEE single precision floating point data format, unless it is absolutely unnecessary (e.g. screen pixel coordinates are stored as integers). Texture coordinates and RHW value are interpolated the same way the Z coordinate was computed, using the coefficients of the plane equation. The difference - beyond the floating point format - is that these coefficients are computed in hardware to reduce external memory reads. The computed values are then stored in on-chip cache (Block RAM).

The interpolator has mixed floating point/integer multipliers and floating point adders. The former has three stages, while the latter has five. The Compute_U and Compute_V modules are nothing more than floating point multipliers with fixed point output (three stages). The divisor unit is based on an iterative algorithm, and is eight stages long.

The PS unit itself represents color values with 16 bits per channel (red, green, blue, alpha) during the computation, while final color values have 8 bits per channel. This unit is also pipelined, it has four stages and two parallel blocks (a Texture Sampling and an Arithmetic unit). The PS unit complies with Microsoft's Pixel Shader 1.4 specifications which defines register numbers and instruction set.

4. Experiences with DK1

The design was done using Celoxica DK1 Design Suite, the development board was a Celoxica RC1000 FPGA card.

With Handel-C (the C based hardware description language behind DK1) it is possible to use the traditional block based hardware description, however it also gives the designer the possibility for further abstraction, and let him concentrate on the pure algorithm regardless of the particular hardware realization.

Although a software version of the algorithms had been coded before the hardware design started, the FPGA implementation is based on an entirely different code, which was created using the traditional hardware design philosophy; the system was divided into functional blocks with well defined inputs, outputs and control signals. We found Handel-C supports this method well. However we feel that it is not too difficult to rewrite the current Handel-C code into VHDL.

The RC1000 FPGA card has a Xilinx Virtex 2000E FPGA on it with 8 Mbytes of SRAM arranged into four banks, which can be accessed parallel with 32 bit data width. The FPGA has the necessary number of Logic Cells and Block RAM size to implement the design. However the card itself has an unfortunate disadvantage, as communication is primarily supported through memory buffers using DMA, and direct communication with the FPGA is quite limited. Therefore at any time, one of the memory banks is reserved for the host to upload transformed vertex data into the card's memory.

5. Conclusion

The design proved that a 3D rendering hardware can be implemented in FPGAs. There are enough resources to implement the needed logic blocks, and they can be designed efficiently even with 64 bit color precision (16 bit per channel).

Our targeted clock speed was 20 MHz – as Celoxica DK1 could achieve not much more than this with the floating point multiplier – and this can be achieved with the design. Clock speed is mainly limited by the HSR (and the Block RAMs) and the interpolation pipeline, while the PixelShader's arithmetical unit can reach much more. The speed of the RAMDAC unit (which handles synchronization signals for the display) is different, and determined by screen resolution and refresh rate.

The current implementation does not use 16 Z processing unit as written earlier, but only 4. Therefore this is not the most resource hungry part of the design; the floating point interpolation and division unit occupy 31% of the chip. The one PixelShader unit needs 15%, the GU takes 10%, while the HSR consumes 8%.

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 Graphics 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] Microsoft, Microsoft DirectX9 Software Development Kit, Microsoft Corporation, http://msdn.microsoft.com
[8] Mrochuk, Jeff, Carson, Benj, Ling, Andrew: Manticore project, http://www.icculus.org/manticore/
[9] 3D Graphics Core Group: Low Power Core for 3D Graphics Texture Mapping, http://www-unix.ecs.umass.edu/~nramaswa/3dg/proj_details.htm