Production Systems and Information Engineering
Volume 6 (2013), pp. 101-112

EFFICIENT 2D SOFTWARE RENDERING

PETER MILEFF
University of Miskolc, Hungary
Department of Information Technology
mileff@iit.uni-miskolc.hu

JUDIT DUDRA
Bay Zoltán Nonprofit Ltd., Hungary
Department of Structural Integrity
judit.dudra@bay-zoltan.hu

[Received March 2012 and accepted September 2012]

Abstract. The market of computer graphics is dominated by GPU based technologies. However today's fast central processing units (CPU) based on modern architectural design offer new opportunities in the field of classical software rendering. Because the technological development of the GPU architecture almost reached the limits in the field of the programming model, the CPU-based solutions will become more popular in the near future. This paper reviews the problems and opportunities of two-dimensional rendering from a practical point of view. An efficient, software based rasterization method is presented for textures having a transparent color component. The applicability of the solution is proved through measurement results compared to other methods and the GPU based implementation.

Keywords: real-time rendering, software rasterization, optimization

1. Introduction

Computer graphics has gone through dramatic improvements over the past few decades, where an important milestone was the appearance of the graphic processors. The main objective of the transformation was to improve graphical computations and visual quality. Initially, the development process of the central unit was far from being a fast paced evolution like today. So based on industry demands, there was a need for dedicated hardware which takes over the rasterization task from the CPU.

Graphical computations have different requirements from the other parts of the software. This allowed for the graphics hardware to evolve independently from the central unit opening new opportunities before developers, engineers and computer games. From the perspective of manufacturers and industry, primarily speed came to the fore against programming flexibility and robustness. So in recent years the development of videocard technology focused primarily on improve the programmability of its fixed-function pipeline. As a result, today's GPUs have quite effectively programmable pipeline supporting the use of high-level shader languages (GLSL, HLSL, CG).

Nowadays, the technological evolution proceeds to a quite new direction introducing a new generation of graphics processors, the general-purpose graphics processors (GPGPU). These units are no longer suitable only to speed up the rendering, but tend the direction of general calculations similarly to the CPU.

However, the problems of the GPU-based rasterization should be emphasized. The applied model and the programming logic slowly reach its limits, the intensity of progress is apparently decreasing. Even there are fast hardware supported pipeline in current graphics cards, they do not provide the same level of flexibility to the programmer to manage the rendering process as CPU based rendering. Although the existing pipeline and 3D APIs provide many features for developers, if we would like to deviate from the conventional operation, we encounter many difficulties.

Today's GPU architecture is questionable and needs to be reformed, as leading industrial partners strongly suggest [12,13]. The video card industry is currently under development. For example AMD's new Fusion technology (APU - Accelerated Processing Units) represents a whole new generation by the integration of the graphical processor and the central unit.

For the problem posed also by game industry leaders [12], the solution is to return to the software rendering technique, where the display content should be programmed logically and technically using the same language as the application.

This would permit creating a more flexible development environment driving computer graphics to a new direction.

A good basis for this is provided by the huge revolution of the CPUs occuring in recent years. Processor manufacturers have responded with extended instruction sets to market demands making possible faster and mainly vectorized (SIMD) processing also for central units. Almost every manufacturer has developed its own extension e.g. the MMX and SSE instruction family which are developed by Intel and supported by nearly every CPU. Due to new technologies, a software can reach about 2-10x speedup by exploiting properly the hardware instruction set.

This paper investigates practical realization questions of two-dimensional software rasterization. A special optimization solution is presented, which helps to improve the non GPU based rendering for transparent textures.

2. Related work

The software based image synthesis has existed since the first computers and it has been focused even more with the appearance of personal computers until about 2003. After this time almost all the rendering techniques become GPU based. However there were born many interesting software renderers during the early years. The most significant results were the Quake I, Quake II renderers in 1996 and 1998, which are the first real three-dimensional engine [8]. The rendering system of the engines was brilliant compared to the current computer technology and was developed by the coordination of Michael Abrash. The engine was typically optimized for the Pentium processor family taking advantage of the great MMX instruction set. The next milestone of computer visualisation was the Unreal Engine in 1998 with its very rich functionality the time (colored lightning, shadowing, volumetric lighting, fog, pixel-accurate culling, etc) [12]. Today Unreal technology is a leader in the area of computer graphics.

After the continuous headway of GPU rendering software rasterization was increasingly losing ground. Fortunately there are some notable great results also today, such as the Swiftshader by TrasGaming [2] and the Pixomatic 1, 2, 3 renderes [14] by Rad Game Tools. Both products are very complex and highly optimized utilizing the modern threading capabilities of today's Multicore CPUs. The products have dynamically self-modifying pixel pipelines, which maximises rendering performance by modifing its own code during runtime. In addition, Pixomatic 3 and Swiftshader are 100% DirectX 9 compatible. Unfortunately, since these products are all proprietary, the details of their architectures are not released to the general public.

Microsoft supported the spread of GPU technologies by the development of DirectX, but beside of this its own software rasterizer (WARP) has been implemented. Its renderer scales very well to multiple threads and it is even able to outperform low-end integrated graphics cards in some cases [3].

In 2008 based on problem and demand investigations, Intel aimed to develop an own software solution based videcard within the *Larrabee* project [5]. The card in technological sense was a hybrid between the multi-core CPUs and GPUs. The objective was to develop an x86 core (many) based fully programmable pipeline with 16 byte wide SIMD vector units. The new architecture made possible to graphic calculations to be programmed in a more flexible way than GPUs with x86 instruction set [4].

Today, based on the GPGPU technology, a whole new direction is possible at a software rendering. Loop and Eisenacher [2009] describe a GPU software renderer for parametric patches. Freepipe Software rasterizer [Liu et al. 2010] focuses on multi-fragment effects, where each thread processes one input triangle, determines its pixel coverage and performs shading and blending sequentially for each pixel. Interestingly, recent work has also been done by NVidia to create a software pipeline which runs entirely on the GPU using the CUDA software platform [7]. The algorithm uses the popular tile-based rendering method for dispatching the rendering tasks to GPU. Like any software solution, this allows additional fexibility at the cost of speed.

A new SPU (Cell Synergistic Processor Unit) based deferred rendering process has been introduced in today's leading computer game, Battlefield 3[13]. Its graphical engine, Frostbite 2 engine makes possible to handle large number of light sources effectively and optimized.

In publication [1] a modern, multi-thread tile based software rendering technique was outlined where only the CPU has been used for calculations and had great performance results.
Thus, recent findings clearly support the fact that CPU-based approaches are ready to come back in order to improve performance and flexibility. So, the aim of this publication is to investigate performance in the area of the 2D software rendering utilizing today CPUs.

3. Software rendering in practice

Software rasterization is the process, where the entire image rendering is carried out by the CPU instead of a target hardware (e.g. GPU unit). The main memory stores the shape assembling geometric primitives in in the form of arrays, structures and other data. The logic of image synthesis is very simple: the central unit performs the required operations (coloring, texture mapping, color channel contention, rotating, stretching, translating, etc.) on data stored in main memory, then the result is mapped into the *framebuffer* and the completed image are sent to the video controller. Framebuffer is an area in memory which is being streamed by display hardware directly to the output device. Usually it is on video card, but can be mapped into address space of the application and accessed directly like normal RAM.
Software image synthesis has many advantages over the GPU-based technology. As the CPU performs the whole processing, there is less need to worry about compatibility issues because we do not have to adapt to any special hardware, or follow its versions. The image synthes can be programmed uniformly using the same language like the application, so there is no restriction on the data (e.g. maximum texture size) and the processes compared to GPU language shader solutions. Every part of the entire graphics pipeline can be programmed individually. Preparing the software to several platform causes less problems because displaying always goes through the operating system controller, there is no need for special video card driver.
Developing a fast software renderering engine requires lower level programming languages (e.g. C, C++, D) and higher programming skills. Because of the utilized techniques it is necessary to use operating system-specific knowledge and codings. A typical example is when the framebuffer should be copied into the video card's memory for displaying. To support this data transfer, several solutions are developed in practice.
Firstly, we can use the operating system routines for framebuffer transfer, but it is strongly platform-dependent. This method requires writing the bottom layer of the software separately for all the operating systems. A more elegant solution is to use the OpenGL's platform-independent (e.g. glDrawPixels, texture) [6] or the DirectX (e.g. DirectDraw surface, texture) solutions.

4. 2D software rendering solutions

Two-dimensional visualization plays an important role beside today's modern three-dimensional rasterization. We can say that the world moved to the direction of the field of 3D vizualization, but the two-dimensional solutions have always been and will be present as a complementary technique. Several layers of the computer applications belong here: any software that has some kind of graphical menu or windowing system, and mainly the two-dimensional computer games. There are always attempts to make menu systems more illustrative using three dimensional graphics, but these solutions usually return to 2D mapping. The rendering speed at these systems is not critical compared to today's system performance. Mostly a small number of static images vary, other transformations (e.g. rotate, stretch, etc.) are not very typical.
In computer games the rasterization performance requirements are the opposite of this. Generally a large number of continuously changing and moving sets of objects have to be rendered, which consumes significant system resources. Typical features of today's systems are high screen resolution, large texture sets, animations and transformations to reach a higher user experience.
In the following several 2D rasterization techniques are presented, which make it possible to develop a high performance and robost software-based rendering system.

4.1 Classical 2D rendering

The classical two-dimensional software rasterization has to solve a great number of difficulties. The main disadvantage of the GPU-based solutions is that there is no special hardware for rendering, any calculation should be done on the CPU.
The basic building blocks of 2D rendering are two-dimensional images (textures) and objects (animations). The graphics engine is responsible for image generation, it takes objects one by one and draws them into the framebuffer. So the final image is created as a combination of these. In all cases, textures are stored in the main memory represented as a block of arrays. Arrays contain color information about the objects, their size depends on the quality of the texture. Today software works with 32-bit (4 bytes - RGBA) type color images, where image resolution can be up to 1024x768 pixels depending on the requirements of the items to be displayed.
Textures can be classified into two main groups based on the color channels included: there are visual elements with and without alpha channel (A). The distinction is important because the displaying process, the potential acceleration techniques differ in these two groups.

4.1.1 Rendering textures without alpha channel

Textures without alpha channel lack transparency and only have RGB color components. This means that any two objects can draw on each other without

merging any colored pixels of the overlapping objects. The rendering process will be faster and simpler.
The classical mechanism for drawing any type of texture is the per-pixel rasterization (just like in 3D area), which is slow in terms of today's high quality requirements. The graphical engine asses through the graphical objects pixel by pixel and generates the final image. The disadvantage of this is that many elements, which can also consist of many points, have to be drawn on the screen. The per-pixel drawing requires thousands of redundant computations and function calls. In case of each pixel the color information should be read from memory, then depending on the environmental data its position should be determined and finally the color should be written into the framebuffer (e.g. pFrameBuffer[y * screenWidth + x] = color). So the pixel-by-pixel realization is not enough to provide a fast solution because too many small operations are performed, which consumes CPU resources. In a real-time computer game up to 100 different moving objects should be drawn simultaneously to the screen.
In order to achieve the appropriate speed, additional solutions and optimizations are needed. In case of textures without alpha channel, the solution is relatively simple. Move the picture array at once in one or more blocks into the frame buffer and not pixel by pixel. So the main objective of a rendering optimization (for 3D as well) is to try to perform all the operations in blocks as wide as possible. This minimizes unnecessary movement of data and calculations. The following figure shows the process:

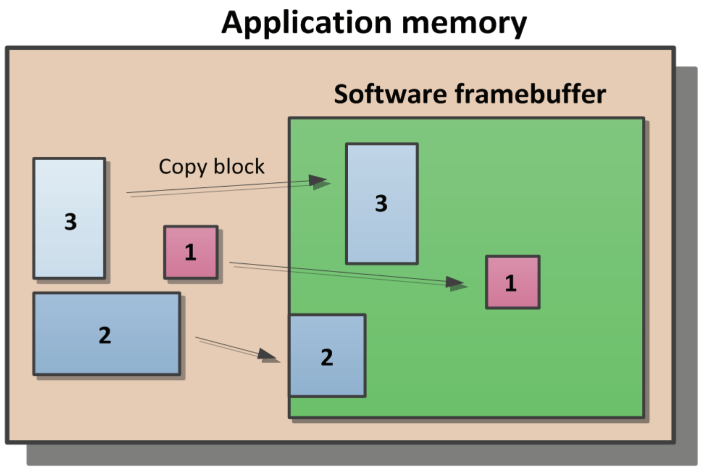

Figure 1. Block oriented texture copy

In this case drawing means that the central blocks of the main memory are copied to a specified area of the framebuffer using system level memory copy operations (e.g. C - *memcpy()*). The solution can achieve significant performance speedup.
However, the method is not complete. The reason is that at pixel level rasterization screen bounding check calculations can be performed easily, but in case of block

oriented rendering the data blocks should be segmented based on the object's position. If any object locates out of the screen bounding rectangle in any direction, a viewport culling should be performed. Although it requires further calculations, the solution remains still fast enough.

4.1.2 Rendering textures with alpha channel

Textures having also a fourth, alpha (A) color channel belong to another group of images. The role of this type of images has increased today, they are used in many areas in order to improve visualization experience (e.g. window shadows, animations with blurred edges, semi-transparent components, etc.). Handling this extra information is not more complicated, but more computing-intensive. The reason is that transparent and non-transparent areas can arbitrarily vary within a texture image (e.g. character animation, particle effects, etc.). Due to this the rendering process is made at per-pixel level because transparent or semi-transparent parts of the objects should be merged with the overlapped pixels. As mentioned earlier, basically the procedure is not very computation intensive, but in systems working with large amounts of objects and larger textures, the solution performance will be insufficient. The following diagram illustrates the problem:

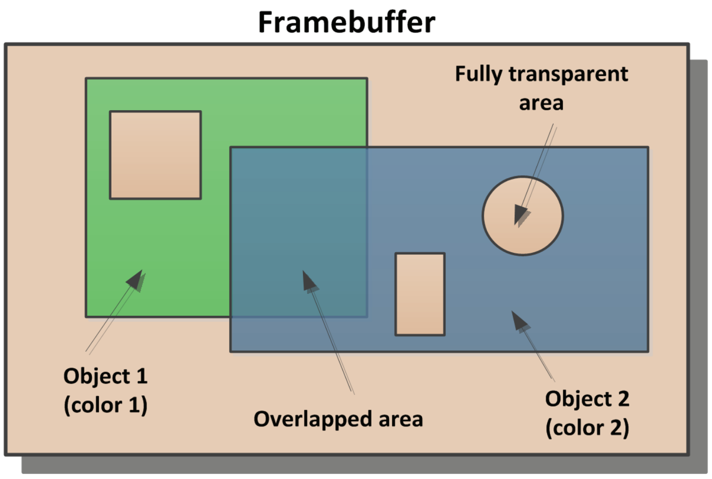

Figure 2. Overlapping RGBA textures

4.2 RLE like object rendering

The above implies that the main problem of two-dimensional software rasterization is to handle semi-transparent textures or textures with 'hole' areas in a performance friendly way. In the following we present a technique which offers an efficient solution to this problem at the cost of some compromises.

It is foreseeable from the above that a process needs to be developed which somehow tries to take advantage of block-based visualization capabilities handling pixels in blocks. To resolve this issue, let us start from the investigation of a typical texture. If we analyze pixels, we can see immediately that the majority of images have parts where the colors of adjacent pixels are equal to each other or there are fully transparent areas. This provides a good basis to develop an algorithm with a custom data structure, which works like RLE (Run-length Encoding) encoding and can group the same color and adjacent pixels into blocks.

4.2.1 Texture pre-processing

The basic idea of the solution is to collect the same colored pixels into blocks. This requires pre-processing operations for all the loaded textures. During the process the appropriate describing data structure will be established, which helps performing the block oriented rendering in the rasterization stage. The following figure shows the steps and logic of the pre-processing task.

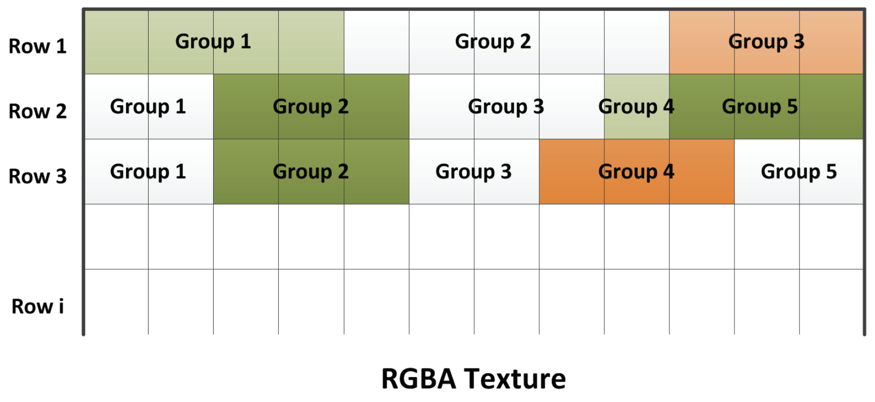

Figure 3. Preparing textures for block oriented rendering

The figure illustrates that an even more complex structure is needed to store the color blocks. Pre-processing is performed row by row, where a separate block is created for all the coherent sets of pixels. For this the following should be stored: color and length of the group, whether the group is transparent or not, and finally a pointer pointing to the first pixel in the original texture address space. Moreover a global data structure should be prepared, which stores the precalculated color groups in rows, their counts and a pointer to the original texture memory space.
The proper implementation of the storage is especially important because in case of systems containing a large amount of objects, the number of loops, function calls, and operations are significant and slow down the rendering process.

4.2.2 The rendering process

During the rasterization stage object mapping is performed by the prepared data structure. This data makes it possible that while the image has 'holes' (fully transparent parts), a color group oriented block based blitting can be achieved on the

framebuffer. The result is that the rendering performance of images with alpha-channel can be considerably increased. The rasterization will consist of so many blocks as many were created during the texture pre-processing stage. In addition, the rasterization is performed row by row. One reason is that the framebuffer has been implemented in a row oriented form like in most systems, so a row is a logical unit.
Another reason is that each object can overrun the screen. Although the colors are grouped, parts that are out of screen should be culled during the rasterization.
The row based approach results again in a speed improvement here because if the beginning or the end of the row is out of the screen, other parts (groups) of the row should not be checked. This prevents performing additional computions. The following code summarizes the drawing process as a sample.

```
CFrameBuffer* framebuffer = g_Graphics->GetFrameBuffer();
  for(unsigned int i=0; i < row_group.size(); i++){
    vector<CRLEColor*> image_row = row _group[i];
    for(unsigned int j=0; j < image_row.size(); ++j){
      CRLEColor * c = image_row[j];
      if (c->invisible == false){
        framebuffer->BlitArray(c->offset,c->length,pos.x+c->x,pos.y+c->y);
      }}}
```

4.3 Test results

The following section presents the performance differences of the rasterization techniques with the help of three test cases. The test programs were written using the C++ language applying the GCC 4.4.1 compiler and the measurements were performed by an Intel Core i7 870 2.93 GHz CPU. The chosen screen resolution and color depth were 800x600x32 in windowed mode.
Because of the results' validity we considered it important to implement all the test cases with the GPU based technology as well. With this reference value, the relative performance ratio of the methods will be visible and clear. The hardware used for the test was an ATI Radeon HD 5670 with 1 GB RAM. To display the framebuffer, the OpenGL glDrawPixels solution was applied in an optimized form.
The GPU based reference implementation was also developed with the OpenGL API, where all visual elements were stored in the high-performance video memory and the VBO (Vertex Buffer Object) extension was applied for the rendering. Currently, VBO is the fastest texture rendering method in the GPU area.
The alpha-channel images used in the tests contained an average number of transparent pixels.

Test case 1: during the test we were looking for an answer to the question of how the presented methods can handle a relatively big texture. Although the RLE-based solution logically does not fit this example because the picture does not contain transparent areas, it is advisable to perform this measurement.

Test case 2: the aim of this test is to measure the speed of the RLE based rendering implementation against the classical method in case of an average size (256x256) image with alpha-channel. The fully block oriented approach cannot be used for this type of images.

Test case 3: in test three a heavily loaded rendering system was simulated applying 200 64x64 size RGBA type textures.

During the tests the average *Frame Rate* (Frames Per Second) was recorded at least one minute run-time. The following Table contains the results for all test cases.

Table 1. Benchmark results

		Speed of rasterization methods (FPS)			
	Count	**Pixel level**	**Block oriented**	**RLE based**	**GPU based reference implementation**
800x600 texture	1	119	1290	1224	3522
256x256 texture (with alpha)	1	910	-	1798	3689
64x64 texture (with alpha)	200	143	-	794	1690

The findings demonstrate that pixel-level rendering was proved to be the slowest because of the large number of operations. However the RLE based approach has good results in all test cases. The frame rate was only lower in the first test, because RLE based rendering requires extra data structures and loops to rasterize the image. This supports the fact well that moving pixels in larger blocks results in significant performance improvements.

Naturally the GPU based implementation produces always the fastest frame rate. But we must not forget that in this case the calculations are carried out by the dedicated hardware. All the data are stored in video ram, so there is no need to move data between the main memory and GPU memory.

4.4 Other optimization issues and features

Developing a really fast software renderer is not an easy task. During the implementation the programmer should take care of several seemingly small coding tricks, which have a strong influence on the performance. For example current 3D hardware is highly optimized for texture and vertex uploads, but framebuffer transfers have been neglected. Therefore the first optimization technique is to implement the framebuffer as an *uint32_t* type array and not as a *floating-point* or *unsigned char* type buffer. This storage type makes it possible to handle all the color components of a single pixel in one unsigned integer type variable, in one single block (e.g. color = A << 24 | R << 16 | G << 8 | B). With this modification at least 20% speed improvement can be achieved.

The built-in data structures (e.g. vector) provided by the C++ STL library are slow, it is not practical to use them [11]. The number of array iterations and unnecessary assignments should be minimized. To detect bottlenecks in the code, use a Profiler application and the component of the compiler which can display the assembly code of a specific code section. These can help to localize the problematic code segments.

Conclusion and further work

Although today's computer graphics is dominated by the GPU market, we cannot forget the opportunities offered by software based image synthesis. The central units have undergone a huge revolution during the recent years, which makes it possible to turn back to CPU based image rasterization in order to gain more flexibility. The techniques presented in this article highlight that developing a really fast software renderer requires a great deal of effort. It is essential to combine several technologies and to use lower-level languages (e.g. C, C++, D) for programming.
This paper discussed software rendering solutions in two-dimensional space. Our further objective is to perform a comprehensive analysis of the triangle rasterization problems and optimization techniques in the area of 3D computer graphics.

Acknowledgements

This research was carried out as part of the TAMOP-4.2.1.B-10/2/KONV-2010-0001 project with support by the European Union, co-financed by the European Social Fund.

REFERENCES

[1] ZACH, B.: *A Modern Approach to Software Rasterization*. University Workshop, Taylor University, 14. dec 2011.

[2] TRANSGAMING INC: *Swiftshader Software GPU Toolkit,* 2012.

[3] MICROSOFT CORPORATION: *Windows advanced rasterization platform (warp) guide*. 2012.

[4] ABRASH, M.: *Rasterization on larrabee*. Dr. Dobbs Portal, 2009.

[5] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., HANRAHAN, P.: *Larrabee: a many-core x86 architecture for visual computing.* ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2008 Volume 27 Issue 3, August 2008.

[6] ROST, R.: *The OpenGL Shading Language*. ADDISON WESLEY, 2004.

[7] LAINE, S., KARRAS, T.: *High-Performance Software Rasterization on GPUs.* High Performance Graphics, Vancouver, Canada, aug 5. 2011.

[8] AKENINE-MÖLLER, T., HAINES, E.: *Real-Time Rendering*, A. K. Peters. 3nd Edition, 2008.

[9] SUGERMAN, J., FATAHALIAN, K., BOULOS, S., AKELEY, K., AND HANRAHAN, P.: *Gramps: A programming model for graphics pipelines*. ACM Trans. Graph. 28, 4:1–4:11, 2009.

[10] FANG, L., MENGCHENG H., XUEHUI L., ENHUA W.: FREEPIPE: *A Programmable, Parallel Rendering Architecture for Efficient Multi-Fragment Effects*. In Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, 2010.

[11] AGNER, F.: *Optimizing software in C++ An optimization guide for Windows, Linux and Mac platforms*. Study at Copenhagen University College of Engineering, 2011.06.08.

[12] SWENNEY, T.: *The End of the GPU Roadmap*. Proceedings of the Conference on High Performance Graphics, pp. 45-52, 2009.

[13] COFFIN, C.: *SPU-based Deferred Shading for Battlefield 3 on Playstation 3*. Game Developer Conference Presentation, March 8, 2011.

[14] RAD GAME TOOLS: *Pixomatic advanced software rasterizer,* 2012.