
*Production Systems and Information Engineering*
Volume 6 (2013), pp. 57-68


# MODERN 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 January 2012 and accepted September 2012]

**Abstract**. The computer visualization process, under continuous changes has reached a major milestone. Necessary modifications are required in the currently applied physical devices and technologies because the possibilities are nearly exhausted in the field of the programming model. This paper presents an overview of new performance improvement methods that today CPUs can provide utilizing their modern instruction sets and of how these methods can be applied in the specific field of computer graphics, called software rendering. Furthermore the paper focuses on GPGPU based parallel computing as a new technology for computer graphics where a TBR based software rasterization method is investigated in terms of performance and efficiency.

*Keywords*: software rendering, real-time graphics, SIMD, GPGPU, performance optimization

## 1. Introduction

Computer graphics is an integral part of our life. Often unnoticed, it is almost everywhere in the world today. The area has evolved over many years in the past few decades, where an important milestone was the appearance of graphic processors. Because the graphical computations have different needs than the CPU requirements, a demand has appeared early for a fast and uniformly programmable graphical processor. This evolution has opened many new opportunities for developers, such as developing real-time, high quality computer simulations and analysis.
The appearance of the first graphics accelerators radically changed everything and created a new basis for computer rendering. Although initially the graphics pipeline and the programmability of the cards followed a very simple model, the previously existing software rasterization quickly lost its importance because CPUs of that

time were not able to compete with the performance of the graphics hardware. 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 improving the programmability of the fixed-function pipeline. As a result, today's GPUs have quite effectively programmable pipelines supporting the use of high-level shader languages (GLSL, HLSL, CG).
Nowadays, technological evolution is proceeding in quite a new direction introducing a new generation of graphics processors, the general-purpose graphics processors (GPGPU). These units are no longer suitable only for speeding up the rendering, but tend the direction of general calculations similarly to the CPU.
However, the problems of GPU-based rasterization should be emphasized. The applied model and the programming logic slowly reach their limits, the intensity of progress is apparently decreasing. Though there are fast hardware supported pipelines in current graphics cards, they do not provide the same level of flexibility for the programmer to manage the rendering process as CPU based rendering. The reason for this is that the pipeline is adapted to hardware limitations and there are many other limitations in utilization of shader languages.
Although the existing pipeline and 3D APIs provide many features for developers, if we would like to deviate from conventional operation, we encounter many difficulties. The general purpose programming of the GPU unit is limited because of the memory model and the fixed-function blocks, which are responsible for performing parallel thread executions [7]. For example, the sequence of pixel processing is driven by rasterization and other dedicated scheduling logic. This is clearly demonstrated by the uniform and predictable look and attitude of today's computer games [1].
Today's GPU architecture is questionable and needs to be reformed, as leading industrial partners strongly suggest [17,18]. What if developers could control every aspect of the rendering pipeline? The answer is practically a return to *software rendering*. A good basis is provided for this by the huge revolution in CPUs occuring in recent years. Processor manufacturers responded with extended instruction sets to market demands making faster and mainly vectorized (SIMD) processing possible also for central units. Almost every manufacturer has developed its own extension, like the MMX and SSE instruction family which are developed by Intel and supported by nearly every CPU. Initially, AMD tried to strengthen with its 3DNow instruction set, but nowadays the direction of development is the Vector Floating Point (VFP) technology and the SSE like NEON instruction set initially introduced at ARM Cortex-A8 architecture.
Due to new technologies, software can reach about 2-10x speedup by exploiting properly the hardware instruction set. All this combined with the GPGPU technology, the question arises: Why could a full software implemented graphical pipeline not be developed where all parts are programmable? Although the

performance probably would not compete completely with a graphical unit, it would offer a more flexible solution than today's only GPU-based solutions.
The main aim of this paper is to examine how it is possible to develop a software renderer built on modern basis, which points forward, is fast enough and takes advantage of technological opportunities inherent in today's central units.

## 2. Related work

Computer graphics has always been a very crowded area over the years, but with the spread of tablet PCs and mobile devices it has come fore even more. It is common in almost all fields whether physical simulation, modeling, multimedia or the area of computer games. However, although the GPU based display has a detailed literature, the area of software rendering has only a small number of new publications since the release of GPUs.
Software based image synthesis has been there since the first computers and it was focused even more with the appearance of personal computers until about 2003. Thereafter almost all visualization became GPU based. Among the software renderers born during the early years, the most significant results were the Quake I, Quake II renderers (1996), which are the first real three-dimensional engines [5]. These graphics subsystems were developed by the coordination of Michael Abrash, and were typically optimized for the Pentium processor family taking advantage of the great MMX instruction set. Among the later results, the Unreal engine (1998) can be highlighted, whose functionality was very rich at the time (colored lightning, shadowing, volumetric lighting, fog, pixel-accurate culling, etc) [13].
After the continuous headway of GPU rendering, software rasterization was increasingly losing ground. Despite this, some great results have been born, such as the Pixomatic 1, 2, 3 renderers [15] by Rad Game Tools and the Swiftshader by TrasGaming [2]. Both products are highly optimized utilizing the modern threading capabilities of today's Multicore CPUs and have dynamically self-modifying pixel pipelines. In addition, Pixomatic 3 and Swiftshader are 100% DirectX 9 compatible.
Microsoft supported the spread of GPU technologies by the development of DirectX, but besides 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 its own software solution based videocard within the *Larrabee* project [7]. The card in a technological sense was a hybrid between the multi-core CPUs and GPUs. The objective was to develop a fully programmable software pipeline using many x86 based cores [4].
Today, based on the GPGPU technology, a whole new direction is possible in 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 [8]. 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.
Today's leading computer game Battlefield 3 [14] introduced a new SPU (Cell Synergistic Processor Unit) based deferred rendering process, which makes it possible to handle and optimize a large number of light sources.
In [1] the author outlined a modern, multi-thread tile based software rendering technique. The solution utilized only the CPU and had great performance results.
Thus, recent findings clearly underline the fact that in order to increase power and flexibility CPU-based approaches come to the fore again.

## 3. Software rendering

The imaging process is called software rasterization when the entire image rasterization process is carried out by the CPU instead of a target hardware (e.g. GPU unit). The shape assembling geometric primitives are located in the main memory 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 the main memory, then the result is stored in the *framebuffer* (holding pixel data) and sends the completed image to the video controller. The following figure shows a general pipeline of a software renderer:

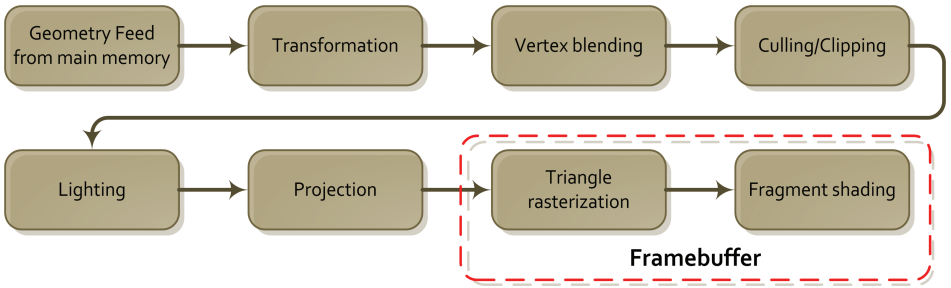



**Figure 1.** General graphics software pipeline

If we look at the pipeline stages, we can see that two dominant groups are formed during the image rasterization process. The first group includes mainly *vertex transformation operations*, which takes up to framebuffer operations. In these phases, the pixel level rasterization is prepared by several matrix and vector transformations (e.g. coordinate system, vertex, cameras, cutting). The second group includes *per-pixel operations*, such as triangle discretization and pixel

shading. For graphics engines from the perspective of rendering these two groups, but mainly the second, are the computationally intensive task.
However, optimizing stages in both groups can result in significant speedup. In the following several modern performance improvement techniques are outlined.

### 3.1 Benefits of software rasterization

The 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 synthetis can be programmed uniformly using the same language as 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 platforms causes fewer problems because displaying always goes through the operating system controller, there is no need for a special video card driver.
In summary, software rendering allows more flexible programmability for image synthesis than GPU technology.

### 3.2 Disadvantages of software rasterization

The main disadvantage of software visualization is that all data are stored in the main memory. Therefore in case of any changes of data, the CPU needs to contact this memory. These requests are limited mostly by the access time of the specific memory type. Frequent changes on segmented data in memory cause significant loss of speed.
The second major problem, which originates also from the bus (PCIe) bandwidth, is the movement of large amounts of datasets between the main and the video memory. During one second the screen should be redrawn at least 50-60 times, which results in a significant amount of dataflow between the two memories. In case of a 1024x768 screen resolution, 32 bit color depth, one screen buffer holds 3 MB data.

## 4. General acceleration opportunities and difficulties

Today's modern processor architecture offers many opportunities to increase the performance of the computationally intensive parts in the graphics pipeline. Naturally, to achieve really good results it is necessary to combine these methods, but due to space limitations this article focuses only on the most important techniques.

### 4.1 SSE based pipeline optimization

In recent years, the most important innovation was built around the SIMD processor instruction sets (Intel – SSE family, AMD – 3DNow, Apple – AltiVec, ARM –

NEON). These allow us to accelerate calculations in a vectorized way and can achieve multiple speed improvement in the pipeline. Besides, another important aspect is the question of programmability: How difficult is it to turn an existing code into an SSE code? Will the code be portable to other operating systems?
Among desktop computers the SSE instruction set is widely accepted today and the instruction set based programming is well-supported by compilers (e.g. GCC, Intel). Modern compilers provide several options to build SSE codes. It is possible to implement the code in assembly language, which requires a deep programming knowledge and the code will not always be portable. Another option is to use the higher level *Intrinsics library* of the compiler. This approach makes the programming level high enough and comfortable (e.g. GCC: __m128 z = mm_setzero_ps(); - fills the vector with zero bytes), and does not limit portability either.

### *4.1.1 SSE based vector optimization*

SSE (Streaming SIMD Extensions) is a SIMD instruction set family (currently SSE 4.2) developed by Intel for x86 architectures. The main innovation is that SSE originally added eight new 128-bit registers, known as XMM0 through XMM7. The extended instruction set provides the opportunity for the processor to execute an operation (e.g. multiplication) on the data of two vectors in parallel.

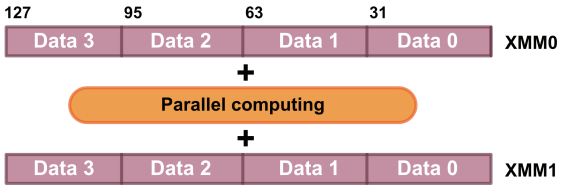



**Figure 2.** Parallel computing with SSE vectors

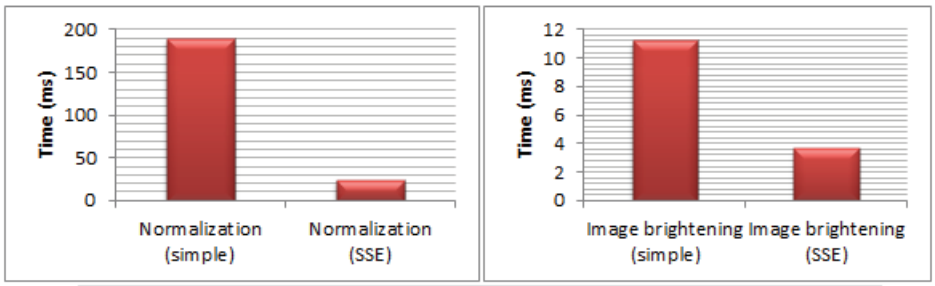
The first operation group of the pipeline typically consists of some kind of vector transformations performed on large datasets. In case of three-dimensional visualization, applying the SSE instruction set makes it possible to store four different 32 bit length floating point numbers (x,y,z,w) in a 128 bit length vector. This means that calculations can be made on these numbers at the same time, which results in significant speed improvements in the execution of the pipeline transformations.

### *4.1.2 SSE based image processing*

The SSE instruction family can be also successfully applied in the rasterization stage of the pipeline or in any other graphical transformation because most of the pixel operations are independent of each other and can be executed in parallel. Today's graphics engines use typically 32-bit (RGBA) color component framebuffers, where each component is 4 bytes long. This mapping fits well with the SSE approach because four pixels colors can be stored in a 128 bit length register and operations can be performed on it in parallel.

### *4.1.3 SSE test results*

In the following, the efficiency of the SSE solution is presented through two test cases. The first test demonstrates a general calculation, vector normalization, which is often used by graphics engines. The formula is compute intensive because it contains square roots and divisions. During the test process 50,000 vector normalizations are repeated 200,000 times.
The second test demonstrates the strength of SSE in an everyday pixel-level image processing task. The test performs 1000 brightening transformations on a 32 bit, 1024x768 resolution image. The test programs were written using SSE2 instruction set, C++ language and GCC 4.4.1 compiler was used and the measurements were performed by an Intel Core i7 870 2.93 GHz CPU. The following table shows the results of each test case.




| Normalization (simple) | Normalization (SSE) | Image brightening (simple) | Image brightening (SSE) |
|---|---|---|---|
| 188,7903 ms | 24,1256 ms | 11,2148 ms | 3,6719 ms |

**Figure 3.** Comparison of the computing results

Measurement results prove the strength of the SSE-based programming. The performance of the calculations in pipeline bottlenecks can be multiple improved with the appropriate SSE code. While in the first case the speed improvement is 7.8x, the second test shows that SSE was 3.05 times faster compared to the conventional code.

### *4.1.4 Drawbacks of SSE programming*

Applying the SSE-based programming, certain compromises are required. Although this instruction set is well-supported by today's compilers (such as C++), an efficient, fast SSE-based code adaptation requires higher programming skills. SSE is not the Holy Grail, a poorly written code can be slower than the traditional approach.
The only restriction of SSE is that the stored and used data must be 16 byte aligned (evenly divisible by 16) in memory. Without this, the arithmetic instructions cannot be used directly. The disadvantage of the aligned memories is that data storage probably does not require 16-byte alignment, so basically we are wasting memory. In return we can gain high performance.

### 4.2 Working with Alignment

The graphics pipeline, thus rasterization speed can be even further improved if we store data properly aligned in memory. All data in memory have two properties: value and address. Data alignment means that the address of the data is divisible by one of the numbers (1,2,4,8) representing the byte length of the alignment. In other words, a data object can have 1-byte, 2-byte, 4-byte, 8-byte alignment or any power of 2. The CPU does not read from or write to memory one byte, instead accesses memory in 2, 4, 8, 16, or 32 byte chunks at a time.
Therefore if the related datasets of the graphics pipeline are not properly aligned to 4, 8, or 16 byte order, then these misaligned structures can cause serious performance losses, because CPU has to perform extra work (load 2 chunks, shift and combine) to access the data [12]. A simple case:

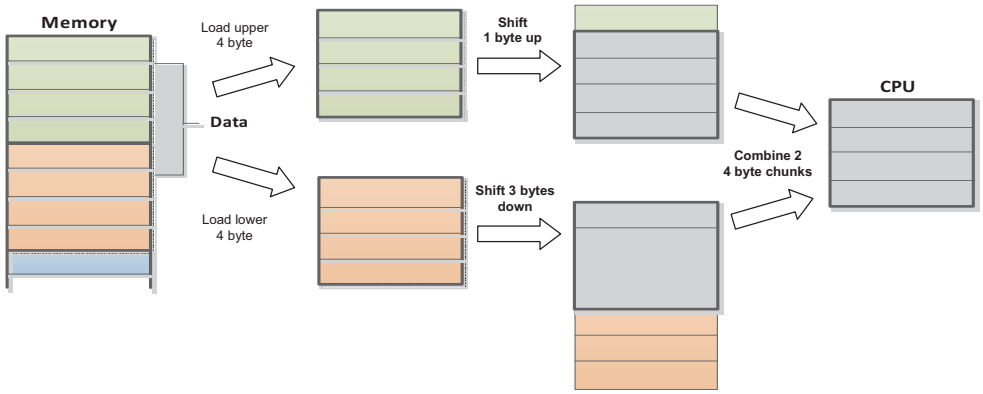



**Figure 4.** Unaligned data usage by CPU

The alignment problem of the pipeline structures is relatively easy to solve in lower-level languages (e.g. C, C++, D). The principle of member alignment is defined by the current compiler, but in most cases the rules are the same. The alignment should be always based on the most restrictive structure member, which is usually the largest intrinsic type. Therefore, members of a data storage structure should be ordered in descending order according to their size. Thus we get an aligned memory structure. In case of larger structure blocks this not only saves memory but the efficiency of rasterization can also be increased significantly.

### 4.3 Minimize cache misses

Today's processors have at least one first-level instruction and data caches on chip, and may have second-level cache memory. Memory access speeds are much faster from these storages. If the pipeline wants to access some kind of data during its running and these data are not in one of the caches, then a *cache miss* event is generated and the data will be loaded into the cache. This event is very costly.

While a value of a variable can be loaded during some clock cycles from the cache, loading it from the main memory requires hundreds of clock cycles. Due to this rule an important goal is to reduce cache misses with the following proposals:

- Frequently used data should be stored together and not segmented,
- Avoid pointer indirection, store and access frequently used data in flat, sequential data structures,
- Accessing data sequentially minimizes cache misses, because each cache miss will load *n* number of new data into the cache,
- Group the functions which work on the same data.

## 5. GPGPU accelerated software pipeline

The GPU-based visualization technology is moving today towards to general purpose processing. This opens up new possibilities for computer visualization and engineering simulations because the graphics processors are no longer limited only to displaying graphics, but can be used for any calculations. The latest GPGPU cards are hiding huge computing power (~1 Tflops/s) because of the inherent growing number of streaming processors (e.g. NVidia GTX 480 has 480 CUDA cores), fast memory (GDDR5) and advanced technology. Since the traditional GPU-based pipeline is not flexible enough, why cannot we use the GPGPU solution of the graphics hardware to implement the pipeline entirely in software?
Logically, the general purpose options of the GPU can be used for any stages of the graphics pipeline with certain restrictions. Since rasterization is a much more computing-intensive task, the computations should be divided between the CPU and GPU in the way that the GPU performs the tasks from the projection stage. The GPU, due to its design and purpose, is very good at parallel task execution. And the process of rasterization typically belongs among well-parallelizable calculations.

### 5.1 Working together with GPU

The objective of the rasterization stage is to map triangles of the models to screen pixels considering the impact of lights, materials and any other factors. In case of software rasterization the typical rendering process is that the CPU takes triangles sequentially from memory, maps their points to the two-dimensional plane and finally calculates their pixels. The color of pixels is stored in a memory array, adapted to the screen resolution, called the *framebuffer*, and after the rasterization the buffer is copied to the video memory.
GPGPU support of the process can be achieved in several ways. For the ideal solution the characteristics and the programming language (OpenCL, CUDA) options of the GPU should be taken into account [9]. The smallest unit of their programming model is the work-item, which runs the implemented kernel code and is groupped into work-groups. Each work-group has a dedicated processor and runs

separately from the others. The group of work-items also runs inside the same computing unit. Therefore the proper rendering process should be chosen so that we could exploit the hardware features. For this, logically the Tile-Based Rendering [6] is the closest rasterization procedure. The following figure illustrates the TBR rendering solution from the perspective of the GPGPU.

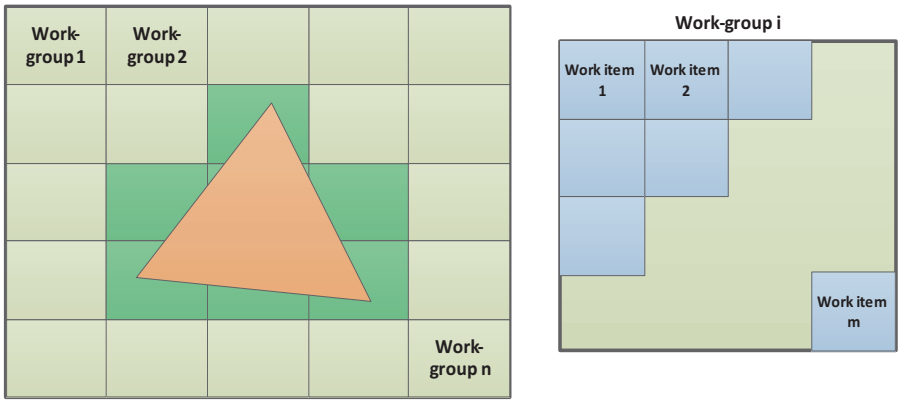



**Figure 5.** GPGPU model of Tile-Based Rendering

Based on the central idea of TBR, the framebuffer should be divided into equal-size areas, called *bins*. In order to exploit the parallel execution of the GPU, all areas should be assigned to a specific work-group, where work-items perform the computations. The rasterization logic is the following: all triangles of the pipeline are assigned to a tile (binning) based on their 2D mapping, whether the tiles overlap or not. All tiles are processed independently and parallelly on a separate processor, where the pixel level rasterization is performed by work-items.
In frame buffer distribution, typically 16x16 or 32x32 size parts should be chosen. The resolution is then not too high to take advantage of the card parallelism (e.g. Card cache size, local memory size, maximum numbers of work items, etc.), and not too small to result in many unnecessary calculations.
In a group, work-items are responsible for rasterizing the image of a tile. They take the list of triangles belonging to the group, calculate their boundaries and perform the pixel level rasterization. Within a group, work-items run also in parallel and share the same local memory. Each item is associated with a triangle, it is responsible for its rasterization. Because triangles can overlap according to their $Z$ value, synchronization is required between the items, which is supported by the languages (OpenCL, CUDA). The whole image (framebuffer) is ready to display when each group has completed its own task.

### 5.2 Constraints of GPGPU programming

The GPU and the main memory are away from each other, so moving data between them is strongly limited by the PCIe bus transfer rate (~2.4 GB/s). It is therefore

appropriate to store all triangle data of the pipeline in a card's memory and minimize data movement. As a short test, an empty, 1024x768 size, 32 bit framebuffer was shared with the GPU and performance was measured. No other calculations were performed, only data sharing and framebuffer displaying. The hardware used for the test was an ATI Radeon HD 5670 1 GB RAM. In the first test case an empty framebuffer was displayed on the screen and no GPU share was applied. The average rendering speed was 1100 FPS. In the second test, the software framebuffer was shared with the GPU in each rendering frame using OpenCL and the average performance dropped to 620 FPS without any GPU calculations.

Another problem is that running a kernel (even an empty one) requires a specific preparation time in execution. In the third test it was investigated how this kernel initialization affects the rendering performance. During the test process an empty kernel was created and four float type variables were shared with the kernel. Rendering speed dropped to 580 FPS this time.

Naturally, there are opportunities to improve the loss of speed arising from the communication. For example, if the entire framebuffer is stored as an OpenGL texture in the card's memory and is shared for GPGPU computations. However, applying this method, we lose a part of the characteristics of software pipeline.

We can say that the modern GPU is very efficient in parallel processing, but is not a wonder tool. Tests show that the technology can be applied in real-time applications, but it provides sufficient efficiency only in case of well-prepared and GPU uploaded data.

## Conclusion

It can be said that the future is bright for a software rendering revolution. The techniques presented in this article highlight the fact that developing a really fast software renderer requires a lot of effort. It is essential to combine several technologies and to use lower-level languages for programming. The CPU has evolved over the past few years: utilizing its potential properly, the performance of the software rendering pipeline can be improved to a large extent. This paper has presented how the GPGPU technology can be applied as a new approach in the rasterization stage. Its parallel potentials are adaptable to the TBR rendering method but only within certain limits.

## 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] EBERLY H, D.: *3D Game Engine Design*: *A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann/Academic Press, 2001.

[6] ANTOCHI, I.: *Suitability of Tile-Based Rendering for Low-Power 3D Graphics Accelerators.* Dissertation, Delft University of Technology, 2007.

[7] 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.

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

[9] OWENS, J., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRUGER, J., LEFOHN, A., PURCELL, T.: A *Survey of General Purpose Computation on Graphics Hardware*. Computer Graphics Forum. v.26, n. 1, pp. 80-113, 2007.

[10] 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.

[11] 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.

[12] 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.

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

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

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