Applied Information Science, Engineering and Technology: Selected Topics from the Field of Production Information Engineering and IT for Manufacturing: Theory and Practice, Series: Topics in Intelligent Engineering and Informatics, Vol. 7, Chapter 5, Springer International publishing, 2014.

ADVANCED 2D RASTERIZATION ON MODERN CPUS

Peter Mileff[1], Judit Dudra[2]

[1] Department of Information Technology, University of Miskolc
Miskolc-Egyetemváros, 3515 Hungary,
[2] Department of Structural Integrity, Bay Zoltán Non-profit Ltd., Hungary

Abstract. The graphics processing unit (GPU) has become part of our everyday life through desktop computers and portable devices (tablets, mobile phones, etc.). Because of the dedicated hardware visualization has been significantly accelerated and today's software uses only the GPU for rasterization. Besides the graphical devices, the central processing unit (CPU) has also made remarkable progress. Multi-core architectures and new instruction sets have appeared. This paper aims to investigate how effectively multi-core architecture can be applied in the two-dimensional rasterization process and what the benefits and bottlenecks of this rendering model are. We answer the question of whether it would be possible to design a software rendering engine to meet the requirements of today's computer games.

1 Introduction

Computer graphics has undergone dramatic improvements over the past few decades, and one important milestone was the appearance of 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 the fast-paced evolution of today. So based on industry demands, there was a need for dedicated hardware to take over the rasterization task from the CPU.

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

Today technological evolution proceeds 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 have capabilities for general calculations similar to those of the CPU. Simultaneously with the soaring of the graphical chips, central units also have evolved. Although this line of development was not as spectacular as the development of GPUs, it was undoubtedly important. The appearance of a new GPU or video card was always surrounded by major media advertising compared to the attention given to central units. For CPUs, initially two development lines have evolved. The first approach considered it appropriate to increase the number of central processing units (Multiprocessing Systems) and then the objective was to increase the cores inside the central unit (Multicore processors). The first dual-core processors have appeared starting in 2005. A multicore system is a single-processor CPU that contains two or more cores, with each core housing independent microprocessors. A multicore microprocessor performs multiprocessing in a single physical package. Multicore systems share computing resources that are often duplicated in multiprocessor systems, such as the L2 cache and front-side bus. Multicore systems provide performance that is similar to that of multiprocessor systems but often at a significantly lower cost, because a motherboard with support for multiple processors, such as multiple processor sockets, is not required.

All of these gave a completely new direction to software development, making the evolution of multi-threaded technology possible. Today we can say that the core increasing process has significantly contributed to improving the user experience of the operating systems (multitasking) and the development of multi-threaded applications. Multi-threaded software which exploits properly the hardware features can far outperform the performance of software applying the classical, one-thread approach. Observing this trend we can see that processor manufacturers and design companies have established themselves in the production and design of (multi) core based central units. Today's mobile devices also have multiple cores (2-4) and in case of PCs the number of cores can reach eight units due to the Hyper-Threading technology.

Besides the core increasing tendency, the 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 families, which were developed by Intel and are supported by nearly every CPU. Initially, AMD tried to boost its position with its 3DNow instruction set, but nowadays the direction of development of mobile central units is the Vector Floating Point (VFP) technology and the SSE such as the NEON instruction set initially introduced in ARM Cortex-A8 architecture. For PCs Advanced Vector Extensions (AVX) open up again great new opportunities in extending perfor-

mance.

Due to new technologies, software can reach multiple speedups by properly exploiting the hardware instruction set. It is therefore appropriate to examine the speedup possibility of the rasterization process. Is there any point in developing a software renderer capable of meeting the needs of today's computer games? This paper investigates the practical implementation issues of two-dimensional rasterization. A special optimization solution is presented, which helps to improve non-GPU-based rendering for transparent textures, heavily utilizing the CPU cores for higher performance.

2 Related works

Software-based image synthesis has existed since the first computers and was focused even more with the appearance of personal computers, up until about 2003. After this time almost all the rendering techniques became GPU based. However, there were many interesting software renderers created during the early years. The most significant results were the Quake I and Quake II renderers in 1996 and 1998, which are the first real three-dimensional engines [8]. The rendering system of the engines was brilliant compared to the computer technology of that day, and was developed under 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 in computer visualization was the Unreal Engine in 1998 with its very rich functionality (colored lighting, 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 made in GPU rendering, software rasterization was increasingly losing ground. Fortunately, there are some notable great results today as well, such as Swiftshader by TrasGaming [2] and the Pixomatic 1, 2, and 3 renderers [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 maximizes rendering performance by modifying 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 in addition, its own software rasterizer (WARP) has also 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 video card within the *Larrabee* project [5]. In a technological sense, the card 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 it possible for graphic calculations to be programmed in a more flexible way than GPUs with an x86 instruction set [4].

Today, based on the GPGPU technology, a whole new direction is possible in software rendering. Loop and Eisenacher [17] describe a GPU software renderer for parametric patches. The FreePipe Software rasterizer [10] 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 the GPU. Like any software solution, this allows additional flexibility 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, a Frostbite 2 engine, makes it possible to handle a large number of light sources effectively and optimized. In [1] a modern, multi-thread tile based software rendering technique is outlined where only the CPU is 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 paper is to investigate performance in the area of 2D software rendering utilizing today's CPUs.

3 Basics of 2D rendering

The name 'software rasterization' originates from the imaging process where the entire image rasterization process, the whole pipeline, is carried out by the CPU instead of target hardware (e.g. a GPU unit). In this case the graphics card is responsible only for displaying the generated and finished image based on a framebuffer array located in the main memory. The main memory holds also the shape assembling geometric primitives in the form of arrays, structures and other data, ideally in an ordered form. 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 the completed image is sent to the video controller.

A framebuffer is an area in the memory which is streamed by the display hardware directly to the output device. So its data storage logic needs to meet the requirements (e.g. RGBA) of the formats supported by the video card. To send the custom framebuffer to the video card, several solutions have arisen in practice. First, we can use the operating system routines (e.g. Windows - GDI, Linux - Xlib) for 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 OpenGL's platform-independent (e.g. GLDrawPixels or Texture) [6] or DirectX (e.g. DirectDraw surface or Texture) solutions.

3.1 Benefits of software rendering

Although software rendering is rarely applied in practice, it has many advantages over the GPU-based technology. The first and most emphasized point is that there is less need to worry about compatibility issues, because the pipeline stages are processed entirely by the CPU. In contrast with the GPU, the CPU's structure changes less rapidly, thus the need for adapting to any special hardware/instruction set (e.g. MMX, SSE, AVX, etc.) is much lower. In addition, these architectures are open and well-documented, unlike the GPU technology.

The second major argument is that image synthesis 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. Because displaying always goes through the operating system controller, preparing the software for several platforms causes fewer problems. Today's two leading GPU manufacturers publish their drivers only in closed form, which leads to significant problems in performance with the Linux platforms. Driver installation is not easy on certain distributions; the end of the process is often the crash of the entire X server. The alternatively available open source drivers are limited in performance and other areas.

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. When the CPU needs to modify these segmented data

frequently, this can cause a 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. Within the period of 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 the case of a 1024x768 screen resolution with 32 bit color depth, one screen buffer holds 3 MB of data.

Moreover, developing a fast software rendering engine requires lower level programming languages (e.g. C, C++, D) and higher programming skills. Because of the techniques used, it is necessary to use operating system-specific knowledge and coding.

4 Overview of 2D rendering

Two-dimensional visualization plays an important role in addition to today's modern three-dimensional rasterization. Computer applications using graphical menu or windowing systems belong in this area, but the most obvious example is the desktop of the operating system. Undoubtedly the main users of the technique are the two-dimensional computer games. Over the years, there has been an increasing demand to improve the visual quality of the virtual world. Today a complex game operates on a large number of continuously changing and moving sets of objects. Thus the rasterization process consumes significant system resources, changing dynamically depending on the moving objects. Typical features of the renderer of these complex systems are a high screen resolution, large texture sets, animations and transformations to achieve better user experience. The screen resolution has been also increased. While in the past the 320x200 and 640x480 dimensions were sufficient, nowadays large high-quality textures (32 bit) are indispensable for higher screen resolutions (e.g. higher than 1024x768). All of these increase the requirements for performance, inducing continuous development of the rasterization models and techniques.

Based on the needs of computer games, in the following it will be shown what the main difficulties of two-dimensional rendering are and why the rasterization stage is so performance intense.

4.1 Characteristics of 2D rendering

Two-dimensional rendering operates on images (textures) and objects (animations) using 2D algorithms. During the image generation process, the graphics engine is responsible for objects being drawn into the framebuffer one by one,

based on a predefined drawing logic. 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.

Based on color channel information, textures can be divided into two types: images containing some transparent area (e.g. cloud, ladder), and images without transparent areas. The distinction is important because the rasterization and optimization methods differ for these types. In the following the implementation logic will be shown.

4.2 Rasterization model of non-opaque textures

Textures without transparent areas use only RGB color components, or the alpha value of all the pixels is maximal. So the image does not contain any transparent pixels ('holes'). This information is very important because it fundamentally determines the display logic. This means that any two objects can be drawn on each other without merging any colored pixels of the overlapping objects. The rendering process will be significantly faster and simpler.

The rasterization of images without transparent areas is relatively simple; the entire texture can be handled at once in one or more blocks and not pixel by pixel. The memory array of the texture is moved into the frame buffer using memory copy operations (e.g. C - *memcpy()*).

There exists only one criterion: to avoid framebuffer over addressing, attention should be paid to the edges of the screen and during copying, the data should be segmented based on the object's position. If any object is located out of the screen bounding rectangle in any direction, a viewport culling should be performed row by row at the texture. Although this requires further calculations, the solution is still fast enough. So this method was preferred in the early computer games.

4.3 Rasterization model of textures having transparent areas

In case of transparent textures, the part of the image contains transparent point groups, or the pixels of the image are opaque in some intensity. To implement the first category, a pre-selected but unused color has to be applied to mark the transparent pixels, this is called colorkey. Today, this is achieved using the alpha (A) channel associated with the image.

The role of this type of textures has increased today: they are used in many areas in order to improve the visualization experience (e.g. window shadows, animations with blurred edges, and semi-transparent components). Handling this extra information is not more complicated but is more computing intensive. The reason is that transparent and non-transparent areas can arbitrarily vary within a texture image (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. Figure 1 illustrates the problem.

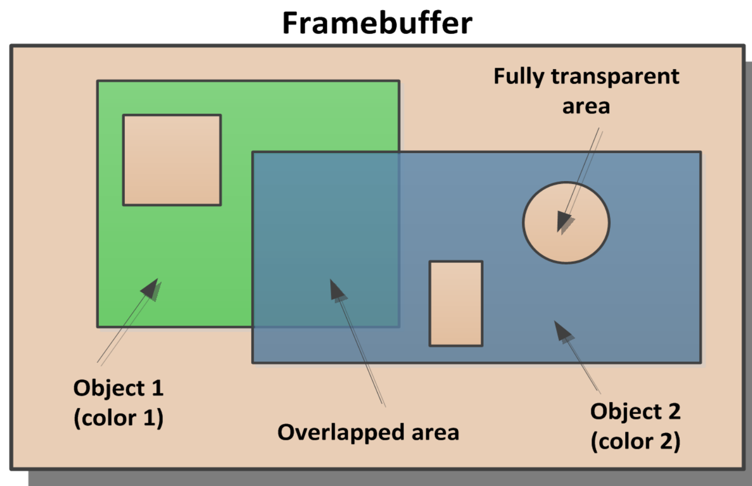

Fig. 1. Overlapping RGBA textures

The graphical engine assesses 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. For 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).

For this reason, this technique does not possess high enough performance to meet the requirements of today's complex computer games, where up to 100 different moving objects should be drawn simultaneously on the screen. Too many small operations should be performed, which consumes CPU resources.

5 Accelerate 2D visualization applying thread management

Given the process and the difficulties of rasterization, the question arises whether it is possible to find a more effective solution which is able to meet the growing demands of today's computer games. As long as the above model is implemented using classical programming technology, performance results will certainly not be satisfactory because the model does not take into account the characteristics of available hardware.

The following simple investigation will confirms this fact: while writing the paper, we implemented a simple one-threaded rasterizer (classical model). The objective was to make performance profiling utilizing transparent textures. The results showed clearly that a software running time of 97% exposes the rendering of the transparent textures: calculating color and position information of the pixel and writing them onto the framebuffer. This by itself does not indicate the inefficient use of the technology. Therefore, as a further investigation, it was observed how many CPU cores were active and to what extent during the runtime. The results showed that at the same time only one CPU core load was high, the load of the others was minimal. This confirms the fact that the simple rasterization model cannot properly take advantage of the features of the hardware.

In the field of parallelization, central units have developed substantially during the past years. The number of cores is continuously increasing and the instruction sets are slowly but also continuously evolving, e.g. the AVX instruction set applyied at Intel Sandy Bridge and AMD Bulldozer processors. Each core has its own cache memory. All of these provide a good basis for establishing intensified parallel computations. Today, having four cores in a central unit of a desktop computer is natural and mobile devices are slowly catching up in this area. An example is the Tegra 4, offering quad-core Cortex A15. It is therefore clear that these should be used in the rasterization process. It is indispensable to design and implement the rasterization model in such a way so as to be able to take into account the opportunities of today's hardware parallelism and dynamically adapt to them.

A software renderer developed by applying parallel technology properly is expected to achieve significantly better results than the classical approach. In the following such a model is outlined, supported by detailed test results.

5.1 The model of the distributed rasterizer

In the course of rasterization it is appropriate to develop a distributed model for the logic of the rendering engine. A solution should be designed which can be built from well parallelized processes. Naturally, the degree of parallelization has

a theoretical maximum upper limit defined by Amdahl's Law [15]: the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. Therefore it is important to investigate first to what extent the rendering process corresponds to the principles of parallelization.

The technology of parallelization is based on the appropriate utilization of CPU cores. This can be achieved most effectively by using hardware threads. However, thread management raises several optimization issues. As an important rule, the core rasterization model should be designed to take into account the potential number of cores in the particular hardware central unit. The parallelization is not optimal when the working threads of an application exceed the number of CPU cores (in the case of Intel processors this includes the virtual cores given the Hyper-Threading technology). When the number of logical threads reaches the number of available hardware threads, the performance slowly starts to decrease because of context switches [15].

Fortunately the process of two-dimensional visualization is simple from the mathematical point of view, therefore parallelization can be adapted more easily than in the case of 3D. The rendering is based on writing pixels into the framebuffer. Since pixels are independent of each other, thus the process of the rasterization can be well parallelized under certain criteria. The most important criterion required is that the synchronization between threads should be minimal. To achieve this, the only requirement is that the same elements of the framebuffer cannot be written by threads at the same time. If we can solve the problem that the threads should not wait for each other during the pixel writing process, the performance benefit is expected to be significant. The result is a multi-threaded rendering model implementing distributed rasterization.

Because the rendering process is mainly slowed by the rasterization of transparent objects, hereafter the paper primarily deals with this category.

Advanced distributed rendering model To design a distributed rasterization model achieving minimal synchronization, we should start from the logical division of the visualization area similar to GPU hardware and Tile Based rendering. Because this division also determines the logical division of the framebuffer, the areas should be designed to be independent. In this case, if the rasterization of the area is performed by separate processing threads, the necessary synchronization between threads will be minimal. The reason for this is that none of the render threads will do pixel operations on areas belonging to other threads.

Starting from the former idea that today a 4-core central processing unit is standard, we consider four identical-sized areas for the screen division. Thus the rasterization process can be performed using four threads. Naturally, the

efficiency of this process is optimal when the number of logical screen areas corresponds to the CPU cores. Figure 2 shows the logical steps of the rasterization process.

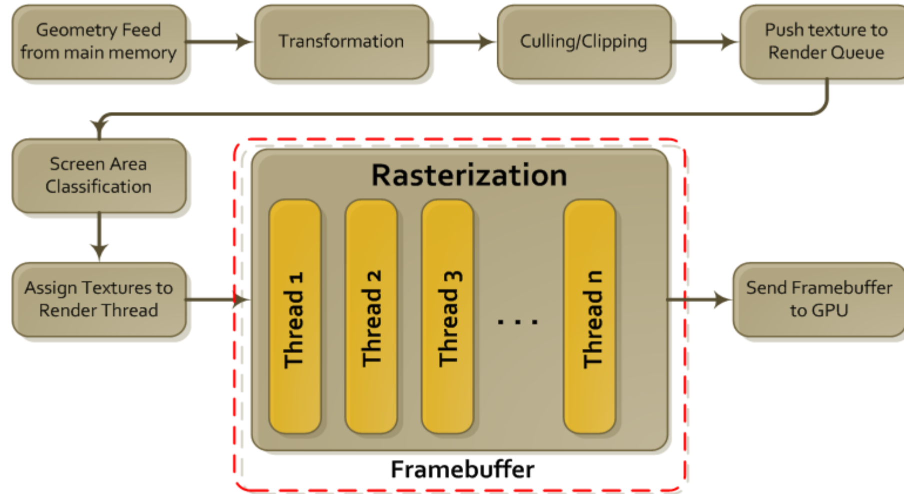

Fig. 2. Distributed rasterization logic

The first part of the applied graphics pipeline is identical to the classical solution. The position and orientation calculation of the 2D objects is performed and the objects are located in the region of the screen. If at least one pixel is visible of the object's texture, its rasterization is indispensable. However, this process differs from the classical method. While previously calling a *Draw()* method immediately writes the texture of the object into the framebuffer, this solution requires a container that collects and holds these textures on a list during the rasterization. The actual rasterization occurs when all of the object's texture is on the list intended for drawing.

The basis of the parallel rasterization is that the different areas of the framebuffer can be written independently and parallel with each other. To accomplish this, a fast classification algorithm is required, which determines the area where the object belongs and its render thread. However, the classification raises a further question. Surely there will be objects whose images overlap the logical areas. So during the classification process the texture will be associated with both threads for rendering, which violates the rules of parallelism laid down. One solution to these problems can be a simple model where the classification algorithm associates the textures with the areas in such a way that the elements belonging to more than one area are associated with the main thread of the

application.

Textures belonging to this thread should be drawn only before or after the rendering process of the other threads. Because of overlapping, the writing process of the thread affects the other thread's logical framebuffer parts. Although the solution is operational and fast, it is not applicable in every case. In computer games, a predefined rendering sequence is usually required (e.g. an airplane flies in front of certain clouds and behind certain clouds). This solution cannot keep the rendering order because of the overlapped areas. To extend the model in order to support the rendering order would require too much communication between threads, killing parallel performance.

In order to keep the advantages of the parallelism, the overlapping problem should be solved. Since we are in the two-dimensional screen space, the texture of each object can be described with a rectangle. This is the area bounded by the texture's width and height.

An effective approach to the problem is to cut the image of the objects along the logical area borders. For this, it is necessary to modify the classification algorithm. While previously all the textures overlapping the logical areas were associated with the main render thread, this model associates these textures with any logical area that is overlapped by the image. The most complex case is when the texture overlaps all four areas of the screen. This requires cutting the texture into four different parts and associating them with four threads. Figure 3 shows the logic of texture cutting and thread association.

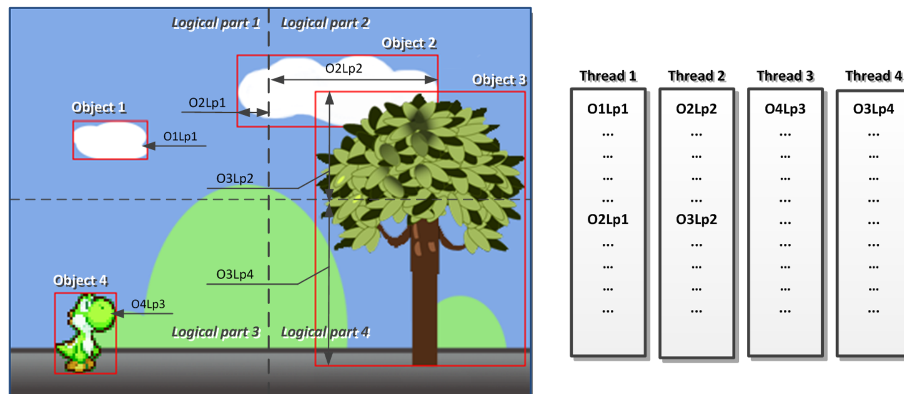

Fig. 3. Rasterization logic

During the visualisation process further checks and calculations are required. Because overlapping textures can belong to a logical area, during pixel level ras-

terization it is necessary to determine the exact pixel borders of the rendering. This avoids conflict with other threads. All of these operations require extra performance from the CPU compared to the previous simple solution, but this does not radically affect the rasterization speed. However, it is clearly visible that this model keeps the rendering order. If the display list is arranged in the correct order before the classification process, the order remains unchanged during rasterization.

Properties of the rendering system Naturally, since threads are used in the visualization system, synchronization between threads is inevitable. The model requires two synchronization points. First, when the threads are assigned their tasks after the classification process and receive the 'start processing' signal. The second point should be at the end of rasterization, to wait for the work of all the threads to end. It is appropriate to place these points at the main thread and to accomplish variable sharing and synchronization through mutex variables. Since creating a new thread is time-consuming, the rendering engine should be implemented in order to avoid continuous thread (re)creation. Threads, for example, can be created after the classification process and ended after their rasterization process, but this will greatly slow down the rendering. Instead, threads live and run continuously. When their work is required, they will be activated, otherwise they will wait. Following these principles the benefit of the implemented renderer is its expected significant performance improvement. The disadvantage at the same time is that the implementation should be more complex.

Although the distributed renderer was emphasized for the rasterization of transparent textures, the principle is also applicable in the case of non-transparent textures. Their rasterization can also be integrated into the model extending the process. As previously mentioned, the rendering process of these types of textures is a set of fast memory copy operations. To keep these properties, the graphical engine should register the type of the textures after the image loading and determine which textures are transparent. During the rasterization process these types of textures are also placed on the same list as the transparent textures. The classification algorithm associates these similarly with one or more logical areas. However, in the actual rendering stage, the engine needs to know the exact type of texture, because rasterization of transparent textures is performed by pixel level and that of non-transparent ones is performed most effectively by memory block copy (including the checking and cutting process of the logical boundaries of the areas).

This way the complexity of the rendering engine increases, but it is worth distinguishing the two types in terms of performance.

6 Test results

The following section presents the performance results of the multi-threaded rasterizer for different test cases. During these tests we considered it important to compare the results to several different solutions. All different test cases have also been implemented by the classical, one-thread rendering solution. In addition, to validate the results the tests were also implemented with the GPU based technology. With this reference value, the relative performance ratio of the methods will be visible and clear.

The GPU based reference implementation was 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. It is important to emphasize that the drawing of textures was performed using GLSL, where two different test cases were distinguished. The role of the non-optimized type is important in mass texture drawing: a shader object is initialized before every object drawing and closed after it. The marker 'non-optimized' means the cost (performance) of the continuous shader changes. The optimal solution initiates the shader object only once before mass texture drawing and closes it after rendering.

The test programs were written in C++ applying a GCC 4.4.1 compiler and the measurements were performed by an Intel Core i7-870 2.93 GHz CPU. Due to Hyper-Threading technology, the CPU can run eight hardware threads in parallel. As a test environment, a 64 bit Windows 7 Ultimate Edition was chosen. The implementations did not use any hand optimized SSE such as code parts; only the compiler optimized code was applied. The chosen screen resolution and color depth were 800x600x32 in windowed mode. The hardware used for the test was an ATI Radeon HD 5670 with 1 GB of RAM. To display the software framebuffer, the OpenGL glDrawPixels solution was applied in an optimized form. The alpha-channel images used in the tests contained an average number of transparent pixels, about 50%. During the tests the average *Frame Rate* (Frames Per Second) was recorded for at least one minute run-time. It is important to highlight that in the case of software rendering, the frame rate was 1714 FPS without any drawing, when only the empty framebuffer was sent to the GPU. The pixel operations were optimized for both software renderer solutions (classical and multi-threaded). Framebuffer is defined as an *uint32 t* type array because 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) [16].

6.1 Simple pixel operation performance test

First a very simple but very practical test is worth examining the performance of pixel writing operations. The task of each test implementation was to fill the screen with a predefined color pixel by pixel. Because of the many pixel operations, processing is very computation-intensive. Table 1 summarizes the results.

Table 1. Pixel writing performance

	Speed of rasterization (FPS)	
	Simple rasterizer (1 thread)	Distributed rasterizer (4 thread)
Write 800x600 pixels to farmebuffer	614	1122

The results clearly show the advantages of the thread-based solution. The distributed version was almost twice as fast as the classical solution.

6.2 Renderer's compound test

In the following our objective is to present and compare the results of the different solutions, applying them to some test cases. Each test represents a special group of tasks. These groups are intended to highlight the most important tasks, those which often occur in computer games. They help to conclude how effective the multi-threaded rasterizer can be in different cases.
Test case 1: during the test we were looking for an answer to the question of how the presented methods can handle a large texture without any transparent areas.
Test case 2: the aim of this test was to measure the renderer's performance applying large and transparent textures. The test image contained an average number of transparent pixels, about 50%.
Test case 3: the test is a transition between the previous and the following cases. It renders 10 relatively large, non-opaque images.
Test case 4: applying the third test case with transparent textures.
Test case 5: a heavily loaded rendering system was simulated drawing 200 64x64 size non-transparent animated objects. Each object has eight different animation frames with identical sizes. Positions are randomly generated and uniformly distributed.
Test case 6: like test case 5 with transparent textures.

Table 2 summarizes the results of all solutions.

Table 2. Benchmark results

	Speed of rasterization (FPS)				
	Number of objects	Simple rasterizer (1 thread)	Distributed rasterizer (4 thread)	Non optimized GPU implementation	Optimized GPU implementation
800x600 texture (RGB)	1	1580	1710	3012	3012
800x600 texture (RGBA)	1	717	1180	3050	3050
256x256 texture (RGB)	10	1192	1336	2960	3056
256x256 texture (RGBA)	10	522	950	2987	3052
64x64 texture (RGB)	200	666	1002	532	1108
64x64 texture (RGBA)	200	380	766	538	1126

As we might expect, pixel level rasterization has the lowest performance in all cases. While the rendering performance of the non-opaque textures is higher, in the transparent case it was much worse. The reason for this is that the renderer draws non-opaque textures with memory copy operations, and transparent textures pixel by pixel. The performance values achieved underline the fact that a rasterizer using one thread is not able to exploit the available CPU resources.

The presented distributed renderer engine performed well in all cases. Although the implemented four- thread based prototype still does not properly take into account the hardware cores, the results are convincing. In one case, its performance was higher than for the non-optimized GPU solution. In addition, it should be noted that this approach scores better on the high graphical load. While in the second test case the rate of performance values of optimized GPU and the distributed approach was 2.58, the rate is 2.09 in the last test case.

The optimized GPU implementation has the fastest performance in all cases. But we should not forget that the calculations are performed by dedicated hardware. There is no need to move data between the GPU and the main memory.

Naturally, in practice there could be additional (exceptional) cases: one example would be in a game if all the objects are positioned in one logical area. In this case one render thread will render all the objects and its performance will be the same as that of the classical approach. Besides, the above examples do not take into account the case where the image of an object should be scaled or rotated. Compared to the GPU based implementation, this requires more resources from the CPU.

7 Conclusion and Further work

Although today the field of 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, offering new opportunities in this area. A powerful GPU cannot be defeated in rasterization performance, but a properly designed software renderer based on modern concepts and solutions is also able to achieve good results in rasterization, not only in speed, but also in flexibility. It should not be forgotten that a fully software based pipeline is less restricted compared to today's hardware solutions. In addition, the model discussed in this paper highlights that there are grounds also for developing two-dimensional games and other graphics applications using software renderers.

Further development of central processing units (e.g. the AVX instruction set) will open up more and more opportunities in this area. Naturally this will require a great deal of effort and applying lower level languages (e.g. C, C++, D) that can take advantage of these potentials of the central unit.

In further work we would like to find the answer to the question of to what extent the performance of the rasterization process can be enhanced by utilizing the CPU's SIMD extensions in both 2D and 3D cases.

Acknowledgements
The described work was carried out as part of the TÁMOP-4.2.2/B-10/1-2010-0008 project in the framework of the New Hungarian Development Plan. The realization of this project is supported 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, Intel Developer Site: http://software.intel.com/en-us/articles/rasterization-on-larrabee, 20. 02. 2013
[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 August 2008 **27** (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 5. Aug 2011
[8] Akenine-Möller, T., Haines, E.: Real-Time Rendering, Publisher: A. K. Peters./CRC Press, 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** (2009) 1–11
[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 20. 02. 2013
[12] Swenney, T.: The End of the GPU Roadmap, Proceedings of the Conference on High Performance Graphics (2009) 45–52
[13] Coffin, C.: SPU-based Deferred Shading for Battlefield 3 on Playstation 3, Game Developer Conference Presentation 8. March 2011
[14] RAD Game Tools: Pixomatic advanced software rasterizer, http://www.radgametools.com/pixomain.htm (2012)
[15] Akhter, S., Roberts, J.: Multi-Core Programming - Increasing Performance through Software Multi-threading, Intel Corporation; 1st edition (2006)
[16] Mileff, P., Dudra, J.: Efficient 2D Software Rendering, Production Systems and Information Engineering **6** (2012) 99–110
[17] Loop, C., Eisenacher, C.: Real-time patch-based sort-middle rendering on massively parallel hardware. Microsoft Research tech. rep., MSR-TR-2009-83 (2009)