MATHEMATICA MONTISNIGRI
Vol XLVII (2020)

COMPARATIVE STUDY OF HIGH PERFORMANCE SOFTWARE RASTERIZATION TECHNIQUES

V. F. FROLOV[1,2], V. A. GALAKTIONOV[1,*] AND B. H. BARLADYAN[1]

[1]Keldysh Institute of Applied Mathematics of RAS,
Miusskaya Sq. 4, Moscow, Russia, 125047

[2]Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russian Federation

*Corresponding author. E-mail: vlgal@gin.keldysh.ru, web page: http://keldysh.ru/

DOI: 10.20948/mathmontis-2020-47-13

Abstract. This paper provides a comparative study and performance analysis of different rasterization algorithms and approaches. Unlike many other papers, we don't focus on rasterization itself, but investigate complete graphics pipeline with 3D transformations, Z-buffer, perspective correction and texturing that, on the one hand, allow us to implement a useful subset of OpenGL functionality and, on the other hand, consider various bottlenecks in the graphics pipeline and how different approaches manage them. Our ultimate goal is to find a scalable rasterizer technique that on the one hand effectively uses current CPUs and on the other hand is accelerating with the extensive development of hardware. We explore the capabilities of scan-line and half-space algorithms rasterization, investigate different memory layout for frame buffer data, study the possibility of instruction-level and thread-level parallelism to be applied. We also study relative efficiency of different CPU architectures (in-order CPUs vs out-of-order CPUs) for the graphics pipeline implementation and tested our solution with x64, ARMv7 and ARMv8 instruction sets. We were able to propose an approach that could outperform highly optimized OpenSWR rasterizer for small triangles. Finally, we conclude that despite a huge background high-performance software rasterization still has a lot of interesting topics for future research.

2010 Mathematics Subject Classification: 78-04, 65C05, 65C20.
Key words and Phrases: software rasterization, graphics pipeline, real-time rendering.

1 INTRODUCTION

Modern hardware-accelerated graphics pipeline consists of dozen stages and has great flexibility [1]. However, it is not always possible to rely on existing graphics hardware for various reasons. At first, various embedded applications do not have dedicated graphics processors and thus forced to use software implementation. Second, a huge amount of popular Linux distributives uses open-source graphics drivers with partial or complete software implementation of rasterization (Mesa OpenGL). Particular applications use specific hardware anyway [2, 3]. In such cases, software implementation should be able to provide real-time rendering sacrificing graphics quality for the sake of correctness or clarity of the displayed information. In such situations programmable functionality of OpenGL shaders, for example, can be excluded or restricted.

At the same time, processors are greately evolved over the past decades and therefore software rasterization methods that were relevant a couple of decades ago may not be the best ones for today. This gives rise to a fundamental contradiction in the design of the rasterizer: it is necessary to pay attention to efficient loading of hardware units of a modern CPU when we come to its peak performance, but on the other hand we don't want to depend too much on any particular hardware. At last, software rasterization is still remaining a widespread challenge in graphics community and thus, have a scientific interest to study within itself.

1.1 Need for software rasterization

Today, almost all rendering techniques have become GPU based. Software solutions, however, do not lose their relevance. For example, Linux uses widely open-source software graphics drivers (Mesa OpenGL [4]). GPU driver installation is not always easy and even not possible on some Linux systems (running, for example, on a custom CPU development board which is quite common for embedded systems). Microsoft also has its own software rasterizer in DirectX10 and DirectX11 called "WARP". WARP rasterizer scales well into multiple threads, and in some cases is known to be faster than low-end GPUs [5]. Besides, software graphics pipilene is more flexible and can directly use system memory. Thus it is useful in scientific visualization of large data sets [6, 7]. At last, the recent development of CPUs sets a new round in software rendering research since many applications for which it was previously impossible to achieve high speed pure in software are enabled now.

1.2 Graphics pipeline

Before moving on, we would like to shortly describe a subset of graphics pipeline that we took for our research and point out why this subset is important and challenging to accelerate on CPU. Useful graphics pipeline requires at least 5 stages:

1. vertex processing;
2. primitive assembly;
3. triangle rasterization;

4. pixel processing;

5. alpha blending.

Both first and second stages are quite simple, especially if we don't have to consider triangle clipping. Vertex processing consists of multiplying points by a matrix and is implemented trivially. Primitive assembly consists of the formation of triangles by indices of its vertices and thus, is mostly trivial. Also, these stages are rarely a bottleneck due to vertices and triangles amount is considerably less than pixel amount.

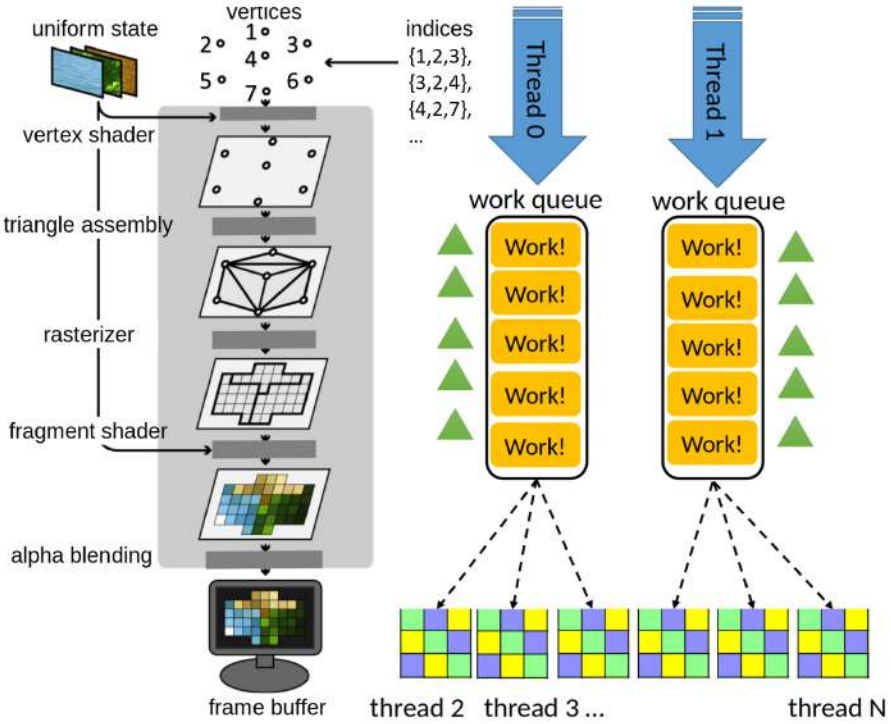

Fig. 1. Graphics pipeline forming producer-consumer scheme where some threads (0 and 1) push triangles (or some other portion of work) to queue and other threads process pixels and behave like consumers taking work from the queue.

However, the following 3 stages are not so simple. It becomes especially noticeable for multi-core implementation where triangle rasterization became a sort of work distribution for pixel processing forming a producer-consumer scheme (fig 1). Alpha blending should be mentioned separately due to it assumes fixed order for processing of pixels for different triangles. The situation is complicated by the fact that not all rasterization algorithms and not all methods of efficient pixel processing (using instruction level parallelizm for example) can be easily used together. This happens due to algorithms have different optimal data structure layout and different access patterns to frame-buffer data. When performance is a goal, these problems became essential. Programmable functionality of OpenGL shaders, on the other side, can be excluded from consideration without loss of generality due to it influences mostly on the pixel processing computation complexity. Thus, we can model its influence if consider heavy pixel processing cases (heavy in comparison to vertex processing and triangle assembly, for example).

1.3 Scientific problems

With the extensively developed graphics hardware last decades many research topics in the area of real-time software rendering became abandoned. At the same time CPUs have evolved significantly:

1. deep out-of-order pipelines, speculative execution, SIMD and various CPU architectures;
2. multi-level caches and tremendous gap between memory and processor speed;
3. true multi-core systems, the number of cores increases significantly;
4. The "relaxed memory model" have appeared and efficient sharing of the cache by many threads has become non-trivial task, especially when increasing number of cores.

Thus, many algorithms and optimizations that were populular 20 years ago (the dawn of graphics hardware development) mostly useless and even performance-harmful for modern CPUs. The goal of our work is to explore different techniques together (considering the influence of all factors upon each other) and find the most practical and scalable approach for software implementation of OpenGL graphics pipeline on modern multicore CPUs which is, in our opinion, is not solved.

2 HIGH PERFORMANCE SOFTWARE RASTERIZATION TECHNIQUES

2.1 Triangle rasterization basics

Before considering triangle rasterization algorithms, we should note that in the existing graphics pipelines (for example OpenGL, DirectX or Vulkan) there is a certain agreement about drawing triangles. A pixel is considered as overlapped by a two-dimensional triangle if its **center lies inside the triangle**. Thus, the pixel-triangle overlap test is called a "coverage test" (fig. 2).

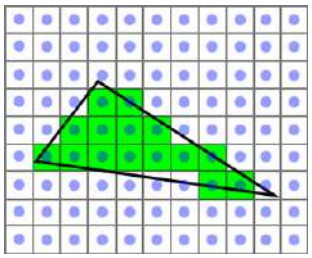

Fig. 2. Standard agreement about covered pixels. A pixel is considered as overlapped by two-dimensional triangle if its center lies inside the triangle.

Probably the most well known **scanline** algorithm [8] sudbidives a triangle into 2 adjacent triangles with horizontal edges. Then it is proposed in some way to move along the edges of the triangle and paint the area between the edges line by line. A straitforward way is to move along edges using finite differences (equations 1 and 2) [9].

$$\Delta_{xy1} = \frac{v2.x - v1.x}{v2.y - v1.y}; \quad \Delta_{xy2} = \frac{v3.x - v1.x}{v3.y - v1.y}. \tag{1}$$

$$\begin{aligned} y &:= y + 1; \\ x1 &:= x1 + \Delta_{xy1}; \\ x2 &:= x2 + \Delta_{xy2}. \end{aligned} \tag{2}$$

We will refer to this algorithm **"scanline"**. Despite the simple idea, we should pay attention to the fact that the algorithm has certain problems:

- The known algorithms for moving along edges (Brezenham [10], Fujimoto [11], or algorithm with finite differences discussed above) do not allow us to say whether the edge pixel is covered by a triangle or not. This means that such a rasterization algorithm itself does not comply with the agreement adopted in OpenGL. For its correct implementation it is necessary to add a pixel-triangle overlap test (so-called "coverage test", fig. 2).
- The algorithm should be additionally limited to a rectangle (built around a triangle), because scanline uses division by the difference between the coordinates of 2 vertices, which under certain conditions became a small number (though zero, as a rule, is excluded by a separate condition that the triangle does not degenerate into a line). This leads to the fact that the offset in y by 1 pixel gives a huge offset in x, which can even go beyond the limits of the screen. The reason for this problem is that according to the OpenGL standard, the coordinates of the triangle's vertices when moving to screen space should be floating point numbers (or at least, have 4-bits subpixel precision [12]). They can not be just integer pixel coordinates. Therefore, strictly speaking, the Bresenham algorithm cannot be used to move along edges.

2.2 Related Work

An improved scanline implementation can be found in [13]. It moves along the longest edge, drawing lines between edges. In comparison to the prevoius naive scanline approach, this algorithm is simpler for CPU due to it has less branches and special cases and it doesn't have a *near zero* division problem because it doesn't use finite differences. However, it does not eliminates the need for the coverage test and the original version does not implement it. We will refer to this algorithm as **"scanline(fast)"** and will test its original implementation without coverage test. Such algorithm would be equivalent to the classic version using Bresenham for movement along the edges.

In [14] **half-space** rasterization was proposed. This paper introduces the concept of *edge-function* (equations 3–6) which was later adopted as a kind of standard agreement for "coverage test" that we discussed before. This method is based on the fact that a line in 2D subdivides the space (plane) into two half-spaces (half-planes). If we substitute the coordinates of the center of the pixel P into the equation of a line, we can obtain the sign distance to this line (equation 3). The *edge-function* is a special case of well known cross

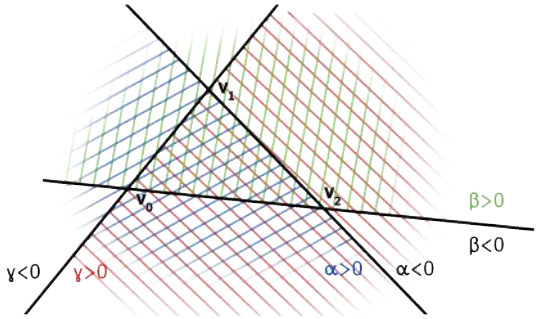

Fig. 3. Half-space algorithm idea

product and it allows calculating the signed distance from pixel center (x, y) to some edge — $(\alpha, \beta, \gamma$; equations 4–6). If all signed distances are greater than zero, the point lies inside the triangle (fig. 3).

$$E(A, B, P) = (P.x - A.x)(B.y - A.y) - (P.y - A.y)(B.x - A.x) \tag{3}$$

$$\begin{aligned} E_\alpha(x, y) = E(A, B, P) &= (x - A.x)(B.y - A.y) - (y - A.y)(B.x - A.x); & (4)\\ E_\beta(x, y) = E(B, C, P) &= (x - B.x)(C.y - B.y) - (y - B.y)(C.x - B.x); & (5)\\ E_\gamma(x, y) = E(C, A, P) &= (x - C.x)(A.y - C.y) - (y - C.y)(A.x - C.x). & (6) \end{aligned}$$

The most useful property of the *edge-function* is that it can be **evaluated incrementally** when rasterizer moves along pixels (figure 4) [14]. Besides, baricentric coordinates (u, v, w) also can be evaluated directly from *edge-function* by multiplying its value with inverse triangle double area which is also evaluated with the *edge-function* (equations 7–9).

$$\begin{aligned} u(P) &= \frac{E(A, B, P)}{E(A, B, C)}; & (7)\\ v(P) &= \frac{E(B, C, P)}{E(A, B, C)}; & (8)\\ w(P) = 1 - u(P) - v(P) &= \frac{E(C, A, P)}{E(A, B, C)}. & (9) \end{aligned}$$

The most significant advantage of half-space rasterizer is extremely simple kernel of the algorithm, especially in comparison with scanline approach. No more difficult to fill the rectangle (fig. 4). This property allows branch prediction mechanisms working efficiently and this is also the reason for the popularity of hardware solutions. The disadvantage of half-space approach (in comparison to scanline for example) is the presence of idle iterations since inside the bounding rectangle; there can be a rather large area which is not covered by a triangle. However, this disadvantage is easily fixed by a serpentine traversal algorithm [14] or Blocked based version of half-space rasterization [15].

```
 1: for y in range minY .. maxY do
 2:     Cx1 := Cy1;
 3:     Cx2 := Cy2;
 4:     Cx3 := Cy3;
 5:     for x in range minX .. maxX do
 6:         if Cx1 > 0 and Cx2 > 0 and Cx3 > 0 then
 7:             u = Cx1*TriAreaInv;
 8:             v = Cx2*TriAreaInv;
 9:             framebuffer[x,y] := DrawPixel(u, v, 1-u-v);
10:         end if;
11:         Cx1 := Cx1 - Dy12;
12:         Cx2 := Cx2 - Dy23;
13:         Cx3 := Cx3 - Dy31;
14:     end for;
15:     Cy1 := Cy1 + Dx12;
16:     Cy2 := Cy2 + Dx23;
17:     Cy3 := Cy3 + Dx31;
18: end for;
```

Fig. 4. Half-space rasterization kernel. $Cx*$ and $Cy*$ variables store edge-functions for line and colum respectively. $TriAreaInv = 1/E(A, B, C)$ is a constant inverse triangle double area. A triplet of $(u, v, 1 - u - v)$ represents baricentric coordinates of a pixel center.

Blocked based half-space method was also suggested in [14] but well-developed much later in [15]. The main idea of blocked version is that if we perform coverage test check (via evaluating *edge-function*) for 4 corner points of a pixel block (4x4 or 8x8 for example) and all tests have passed then the block is covered by triangle and we can fill/process all internal pixels in parallel (for example using SIMD instructions). Several blocked versions of half-space rasterizer were proposed and tested in [15]. The most complex version (called "Block-based Bisector Half-Space Rasterization") processes triangle in such a way that it minimizes checks for empty blocks due to a quick cut of empty space from inside triangle bounding box. The advantages of "bisector" algorithm appear only on extremely large triangles and simple fill modes (without texture for example) because incremental *edge-function* evaluation is quite cheap in comparison to pixel processing for a fully-covered or even partially-covered block. At the same time average amount of blocks for most of triangles is usually just a little: 4-8 blocks. As a result complication of the algorithm leads to poor performance due to branch misprediction simultaneously with winning of empty blocks tend to zero. We will refer to the blocked version of half-space rasterizer as **"blocked half-space"**. The main advantage of blocked version (over previous half-space approach) is the possibility of parallel processing of pixels via SIMD instructions. Besides, blocked half-space algorithm processes empty space faster. The disadvantage of blocked version appears with small triangles — not all calculations that were performed for 4x4 tile (for example) are useful.

2.2.1 Floating point vs fixed point

When choosing between a floating point and a fixed point, two cases should be distinguished: (1) rasterization algorithm itself and (2) pixel operations. When speaking about rasterization — current graphics hardware uses "28.4" or "15.8" (or other) fixed point

format with 4 or 8 bit subpixel precision [12, 1, 16] and there is simply no any reason for using floating point to process the triangle in the rasterization algorithm. This is so because a fixed point has deterministic behavior and is not subject to rounding errors; therefore, it's not even about speed, but rather about correctness. Both half-space and scanline approaches are known to be implemented in fixed point well [17, 18].

Fig. 5. PlayStation1 (right) didn't have correct texture mapping due to absence of floating point for pixel operations [19].

While speaking about pixel processing — it depends on the hardware. Early versions of gaming consoles didn't have floating point support [20] so they had visible problems with texture mapping and Z-buffer (fig. 5). There are still processors without a floating point and SIMD support (or its performance may be not enough), therefore, fixed-point can be an option [18]. Also, if we do not need rendering in three dimensions, we can get by with a fixed point. Otherwise, we believe that for pixel operations it's better to use floating point in conjuction with SIMD. Here are our reasons:

1. Rendering in 3D is difficult to be correct without a floating point (fig. 5).
2. SIMD and floating point can be used together. If SIMD instructions are enabled, there should be no need in complex and chip-expensive Out Of Order execution mechanisms to speed-up floating point operations. Blocked based half-space always has a lot of independent work (at leats 16 operations for 4x4 pixel block), so coarse-grained instruction parallelism [21] can be used. GPUs actively use this idea sending commands to the pipeline from different micro-threads [22]. This is why they are so good at floating point operations and have high memory bandwidth. Thus, even straitforward implementation of SIMD floating point should work well.
3. Almost all CPUs have different register sets for integer and floating point numbers. Using both (we must use integer registers for fixed point rasterization anyway) will increase the effective number of processor registers and in this case reduce register pressure.
4. A CPU may not have SIMD for integers (for example, SSE1 doesn't have them).

5. Half precision reduces architectural state by a half and thus more pixels can be processed in parallel or we may use less transistors for the CPU or, at least, reduce necessary memory bandwidth. Processors with half precision support for neural networks are currently becoming popular (for example late ARM and Intel CPUs).

Thus in our experiments we used fixed point for triangle rasterization algorithms and SIMD floating point operations for pixel processing.

2.2.2 Multi-threaded implementations

Multithreaded implementation of a graphics pipeline is a challenging task. Figure 1 shows it in general. An unknown number of triangles of arbitrary size is fed to the input of the graphics pipeline in general (so it is hard to say in advance exactly at which stage of the pipeline there will be the bottleneck). Non uniform work distrubution is easy arising here. Triangles could significantly overlap each other. Moreover, if alpha blending is enabled, a certain order of pixel processing for triangles must be preserved: if the triangle A was filed into the graphics pipeline before the triangle B, then A must be drawn before B and its pixels must be processed before the pixels of the triangle B. Otherwise, we will get an incorrect image.

One of the first papers about software rasterization on modern CPUs is [23]. In this paper, SSE instructions and multithreading capabilities were exploited. Binned implementations of rasterizer was used (which is known as a **"sort-middle"** approach [1]). In this paper, screen is subdivided into large bins/tiles (in size of 64x64, 128x128 or 256x256 pixels). Once all primitives are binned, threads switch over to tiles for rasterization and fragment processing work. Thus, in this paper, for each bin there is its own queue of triangles, which is first completely filled with all the threads, and then all the queues are emptied in parallel. One tile is processed at a time by only one stream. The blocked version of half-space rasterization was used with 8x8 block size for SIMD processing of pixels. An advantage of *sort-middle* approach from [23] is the correct alpha blend support by default due to each bin is processed in a single thread. The disadvantage is a limited parallelization capability due to different bins could have significantly different numbers of triangles and thus some bins will hang for a while in a single thread when all the others bins/threads have already finished. A performance growth demonstrated in [23] was measured on a quite heavy pixel operations (which reduces the described disadvantage) with shadow mapping, and even in this case was not perfect. Authors of [24] simply split screen in 4 parts and [25] also didn't introduce any new technique.

In [18] disadvantages of *sort-middle* approach was also noted and a solution was proposed that is parallelized almost perfectly — render different frames completely in different threads. This idea is similar to Nvidia SLI and AMD Cross-Fire GPU solutions [26]. The reason for such successful results is that this work bypasses the Amdahl law, making sequential calculations parallel via pipelining. Unfortunately, it has at least 2 drawbacks. First, this method of parallelization does not reduce the latency of rendered information. It makes the animation smoother, but the user sees the information on the screen with such a delay as if the whole rendering has occurred in a single thread. In automotive and avionics applications, for example, such disadvantage became serious, because a person in critical situation may wrong react to displayed information due to a time lag. Second,

a processor memory bus has limited bandwidth and thus SLI-method has a physical limitation on parallelizability on a single device (so, Nvidia and AMD use it for multi-GPU setup) due to each thread accesses its' own frame buffer and the total amount of memory moved along the bus increases with the number of threads.

Unlike previously discussed papers, an older work GRAMPS [27] uses the approach which is known as **"sort-last"**[1]. This approach parallelizes individual operations on pixels or groups of pixels, and unlike *sort-middle* does not require screen to be split into bins. Thus, different triangles can be processed by different threads. The main focus of [27] was done on prototyping and simulating graphics hardware. So, there was no information about efficiency of this approach for software implementation on practice.

2.2.3 Hardware solutions: sort-middle vs sort-last

Modern graphics hardware has a tremendous amount of parallellism inside. However, before fragments/pixels finally got to the frame-buffer they have to be sorted in some way to form a correct image. This becomes especially important if alpha blending is used. Current graphics hardware can be divided into 2 large classes based on what stage of the graphics pipeline this sorting takes place: *sort-middle* and *sort-last* [1].

Desktop GPUs have a high memory bandwidth and uses *sort-last* approach implementing the ordering of fragments inside Render Output Unit ("ROP") hardware units. Same units are known to be used for atomic operations in GPGPU, so, ROPs are useful units anyway. Mobile GPUs are aimed more at energy efficiency than at performance and use a *sort-middle* method (except Nvidia Tegra). This approach is more energy efficient because it allows performing fewer operations to DRAM keeping a small piece of framebuffer (for example 64x64) in the on-chip memory (cache). The disadvantage of *sort-middle* approach for GPUs is lower performance with a large number of triangles due to vertex shader and triangle set up executes several times (thus multiplying the cost of geometry stages with the number of tiles).

2.2.4 Software rasterization on GPUs

First succesfull software GPU implementation "in compute" (i.e. without using dedicated rasteriszation units) was proposed in [12]. This implementation was a three-level (bin-raster, coarse-raster, fine-raster) and used *sort-middle* on desktop GPUs. More advanced approach was suggested in [28] which reduces memory transactions in comparison to [12]. Due to efficient usage of shared memory and the extremely high computing power of the GPU, good results were obtained in both papers described above. Combined with a heavy pixel shader software rasterizations may have almost the same speed than hardware implementation but it may have higher flexibility.

Larabee [29] uses 4x4 blocked half-space with 16-wide vector instructions and the algorithm was recursive: each triangle evaluates 16 blocks of pixels at a time to figure out which blocks are even touched by the triangle, then descended into each block that's at least partially covered, evaluating 16 smaller blocks within it, continuing to descend recursively until we had identified all the pixels inside the triangle [16]. Thread paralellizm used *sort-middle* approach.

2.3 PERFORMANCE EXPERIMENTS AND ANALYSIS

We tested various methods on the fixed set of scenes. However, the purpose of our experiments was to select successful methods for a wide range of scenes. Therefore one of the most important criteria for an objective study is the correct choice of test scenes.

2.3.1 Test scenes

Our test scenes are presented at fig. 6 and 7. We chose these scenes so that the bottlenecks are presented in different parts of the graphics pipeline. Here is the description of these scenes and their rasterization modes/states:

1. T1: 18 triangles, color interpolation with perspective correction and Z-buffer;
2. T2: 8K triangles, color interpolation without perspective correction (2D mode);
3. T3: 92 triangles, texture with bilinear fetch, perspective correction and Z-buffer;
4. T4: 4K triangles, same rasterizer state than a previous one;
5. T5: 37K triangles, same rasterizer state than a previous one;
6. T6: 131K triangles. same rasterizer state, lighting was baked in the texture.

T1 scene is simple in all stages: geometry, rasterization and pixel processing. T2 scene is simple in pixel and geometry processing, but more complex for rasterizer itself due to it draws 8K small triangles. T3 scene is complex in pixel processing but simple at geometry and rasterization stages. T4 (4K triangles) and T5 (36K triangles) scenes are more or less balanced. T6 scene contains 131K triangles and is positioned as a complex scene for all stages. T6 scene has baked lighting. Therefore, having a small number of test scenes, we are able to study different bottlenecks in graphics pipeline ignoring irrelevant details of a complete OpenGL implementation in the same time.

2.3.2 Investigated and proposed techniques

Thus, we have implemented minimal but useful graphics pipeline subset. Such things as attribute interpolation, perspective correction and depth buffer during triangle rasterization are implied. Pixel processing includes texture mapping with bilinear filtering. However, we don't evaluate differentials (dFdx/dFdy [30]) for texture coordinates and avoid using MIP levels. For each OpenGL state we have implemented code generator using C++ templates for pixel processing excluding unnecessary code explicitly.

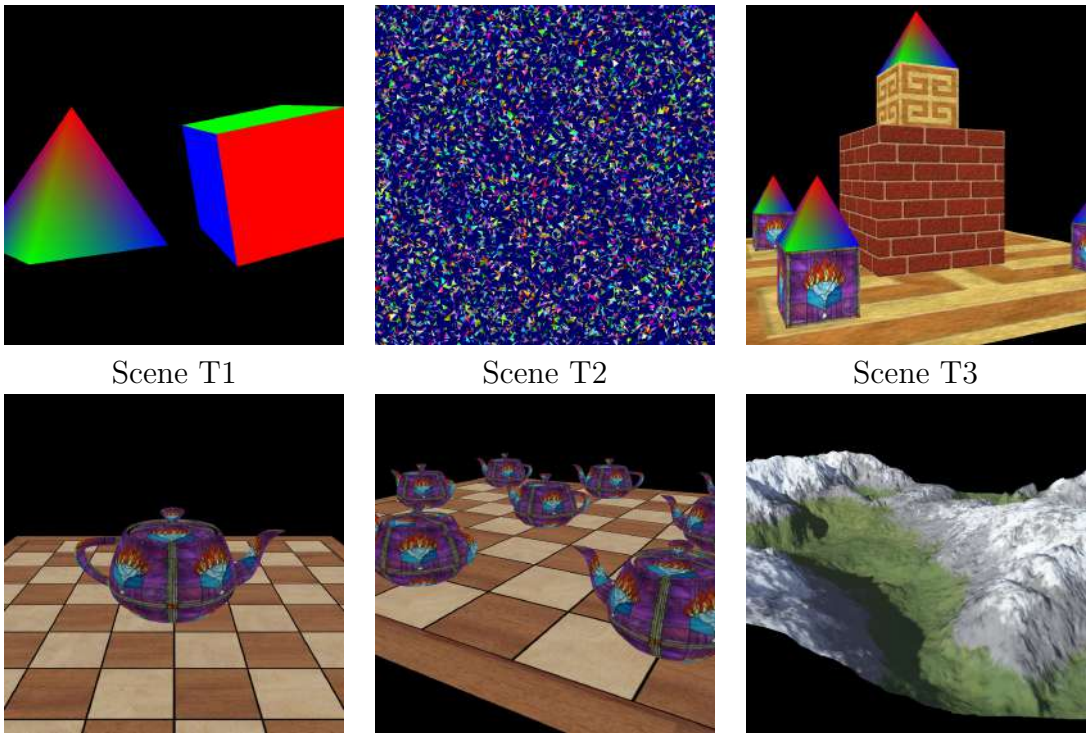

Fig. 6. Our test scenes rendered in solid mode to demonstrate their actual appearence.

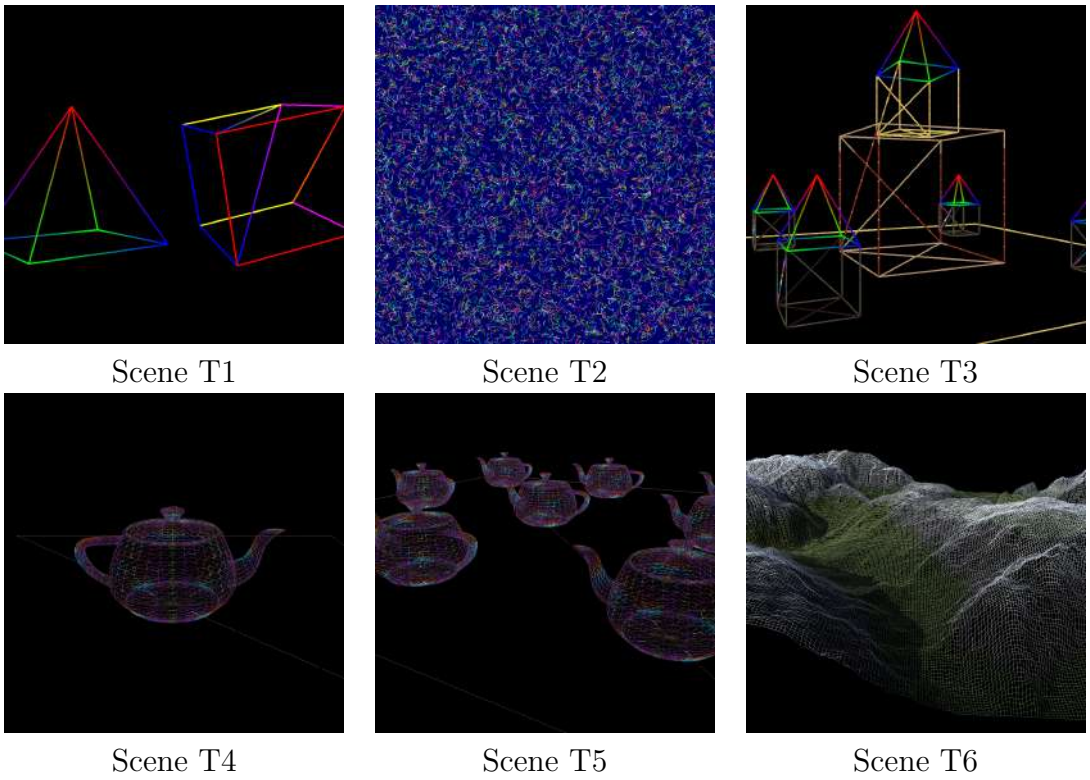

Fig. 7. Our test scenes rendered in wire frame to demonstrate triangles.

Thus, our pixel processing code doesn't have any branches except a depth test. For experiments we used three states/filing modes: (1) 2D color interpolation without texture (T2 scene); (2) 3D color interpolation with perspective correction and depth test (scene T1), and (3) 3D mode with texture mapping (other scenes), perspective correction, depth test and bilinear texture fetch.

Using compiler explorer [31], we have estimated that the first mode consists of approximately 68 instructions per pixel and the second takes 290 instructions (though each instruction processes line of 4 or 8 pixels for blocked half-space algorithm) for x64 CPU architecture (table 6). It may seem that having 68 instructions for just interpolating colors is too much. This is partly true; here we can see the disadvantage of the blocked half-space algorithm: it must evaluate half-space distances and baricentrics for all pixels in block while the iterative half-space evaluates them incrementally. On the other hand, texture mapping introduces significant amount of computation making this disadvantage irrelevant.

Rasterization algorithm: scan-line vs half-space. Our first experiment was about comparison of existing rasterizations algorithms on a single core (table 1). We used SSE processor instructions to accelerate computations where possible. For scan-line and half-space columns we vectorized the calculations by coordinates and image channels (we call such approach "sse1" in table 2). For blocked half-space we used pixel vectorization (i.e. single command processes a bunch of pixels; we call this approach "simd(sse4)" and "simd(avx8)" depending on instruction length). Rasterization algorithms themselves were implemented in a fixed point. We further studied optimal tile size (which is related to vector length) in our experiments (table 2).

scene	half-space	blocked half-space	scan-line	scan-line (fast)	fill_color
T1	286 FPS	294 FPS	158 FPS	**400 FPS**	625 FPS
T2	**650 FPS**	417 FPS	83 FPS	117 FPS	667 FPS
T3	68 FPS	**91 FPS**	61 FPS	73 FPS	500 FPS
T4	76 FPS	**87 FPS**	48 FPS	53 FPS	400 FPS
T5	**57 FPS**	51 FPS	35 FPS	46 FPS	250 FPS
T6	**50 FPS**	40 FPS	19 FPS	22 FPS	116 FPS

Table 1: Time for different rasterization algorithms. Each implementation was accelerated with SSE instructions. All numbers (FPS, Frames Per Second) are measured for single thread and 1024x1024 resolution. The higher is better. The last column fill_color is a tiled half-space algorithm filling all pixels with white color (like memset). We consider the performance of this case as the best possible one and compare the rest with respect to it. For this experiment we have used Intel Core i7 (3770, 3.4Ghz) CPU.

Experimental results show that the scan-line approach does have an advantage over half-space on large triangles and simple filling modes if a coverage test is removed (table 1, first row, scene T1). However, this advantage is easily eliminated by increasing block size in blocked half-space algorithm (table 2, fist row, avx8 column): blocked half-space gives

448 FPS versus 400 FPS (*this comparison, though is not quite correct since the numbers in tables 1 and 2 were measured on different processors, but we can rely on it because Xeon with a lower frequency in a single thread is usually slower than the Core-i7*) for scan-line (fast). In all other cases, half-space and blocked half-space show absolute advantage over scan-line approach.

Comparing half-space and blocked half-space approaches we can say that blocked half-space algorithm is usually better (table 1). The exceptions are scenes T2 and T6 where common half-space algorithms substantially defeated the vectorized version. This result is explained quite simply: T2 and T6 scenes contain a lot of small triangles which result in a large amount of partially-covered blocks for a block based rasterizer.

scene	pure_cpp	simd (sse1)	simd (sse4)	simd (avx8)	fill_color (sse4)
T1	147 FPS	297 FPS	427 FPS	**448 FPS**	588 FPS
T2	108 FPS	**204 FPS**	102 FPS	96 FPS	137 FPS
T3	35 FPS	61 FPS	83 FPS	**92 FPS**	500 FPS
T4	35 FPS	65 FPS	**74 FPS**	62 FPS	323 FPS
T5	26 FPS	**44 FPS**	42 FPS	33 FPS	119 FPS
T6	17 FPS	**36 FPS**	16 FPS	13 FPS	30 FPS

Table 2: Frames per second for different acceleration techniques for half-space (pure_cpp and sse1) and tiled half-space (simd(sse4), simd(avx8)) rasterizers. All numbers are measured for single thread and 1024x1024 resolution. The higher is better. The last column fill_color (sse4) is a tiled half-space algorithm filling all pixels with white color (like memset). We consider the performance of this case as the best possible one. For this experiment we have used Intel Xeon (5-2690 v4 2,6Ghz) CPU.

Combined approach. Such a result encourages us to combine sse1 and sse4 implementations: if a block is fully-covered, we used vectorized pixel processing; if a block is partially-covered we render its pixels subsequently using vectorization by coordinates or color channels (table 3, column "sse1+sse4"). It can be seen from table 3 that combined approach is good in average, but was not the best in all cases. We explain this by saying that blocked half-space implementation (and combined algorithm as follows) is much more complicated for branch prediction and speculative execution mechanisms. So, combined approach can be further improved: for triangles with small area use simple half space (sse1) and for other — cobmined (sse1+sse4) algorithms. This approach allowed us to fix performance for scenes with a large number of small triangles (T2 and T6).

Threads: sort-middle vs sort-last. As can be obvious from the previous work, most existing implementations use straitforward *sort-middle* approach subdividing image into bins. This approach supposes that pixel work dominates over geometry and rasterization itself. We also began with *sort-middle* approach but we have found that adding bins is in itself introducing essential overhead (table 4, second column). This happens due to essential duplicating of triangles that overlapped several bins and it becomes noticeable on geometrically-heavy scenes (T2, T5 and T6). Then we decided to try a different approach.

Having 4x4 blocked half-space algorithm, we decided to use spin-locks for 4x4 tile and thus implemented *sort-last*. We used *std::atomic_flag* [32] for spin-lock implementation.

The *sort-last*, in general (if we do not take into account the locks), should scale better due to it processes separate triangles in parallel. An additional advantage of this algorithm is locality and cache efficiency for triangles data: rasterized triangles are formed on the top the stack (or triangle queue) memory and they are in the cache.

If go further, *sort-last* could be optimized in such a way that it reads data directly from user input pointers, rasterizes triangles and immediately discards them (thus turning into a memory-compact and cache-effitient way). However, we did not do this because OpenGL has tremendous amount of ways for input user data layout.

scene	pure_cpp	simd (sse1)	simd (sse4)	simd (sse1+sse4)	fill_color (sse4)
T1	147 FPS	297 FPS	427 FPS	**430 FPS**	588 FPS
T2	108 FPS	**204 FPS**	102 FPS	197 FPS	137 FPS
T3	35 FPS	61 FPS	83 FPS	**84 FPS**	500 FPS
T4	35 FPS	65 FPS	74 FPS	**79 FPS**	323 FPS
T5	26 FPS	44 FPS	42 FPS	**46 FPS**	119 FPS
T6	17 FPS	**36 FPS**	16 FPS	31 FPS	30 FPS

Table 3: Comparison of suggested combined implementation (sse4+sse1). All numbers are measured for single thread and 1024x1024 resolution. The higher is better. The last column fill_color (sse4) is a tiled half-space algorithm filling all pixels with white color (like memset). We consider the performance of this case as the best possible one. For this experiment we have used Intel Xeon (5-2690 v4 2,6Ghz) CPU.

Although, the *sort-last* can be implemented in different ways, we used the simplest approach: a thread performs lock of 4x4 tile, processes pixels and then immediately unlocks the tile. For parallel processing of triangles we used a lock-free concurrent queue [33]. Some threads act as *producers* and push triangles into queue (1 or 2), while the others act as *consumers*, taking out triangles from the queue and performing rasterization. We did not limit the size of the queue, although we believe that for better cache efficiency it is worth doing, switching producer threads to consuming triangles when a limit has been exceeded.

Fig. 8 shows our experiment results. The *sort-middle* approach, as expected, was better for pixel-heavy scenes. However, for cases where pixel work was not enough, *sort-last* approach has won. The exception is T6 scene. This result seemed strange for us, especially in combination with the fact that *sort-last* has shown almost linear scaling on T2 scene. Nevertheless, this result may be explained. Scene T2 consists of 8K small random triangles (which bounding boxes overlap only slightly) where each next triangle is located at random position on the screen. Scene T6 consists of successive *triangle strips* and also triangle bounding boxes overlap much more. We were able to achieve a slight performance increase (15-20%) by increasing the pulling portion size for the consumer up to 4 triangles (this reduces conflicts of threads if they process a single trip). However, threads that handle different strips still conflict much. Moreover, T6 scene is heavier for pixel processing than T2, so *sort-middle* has won here.

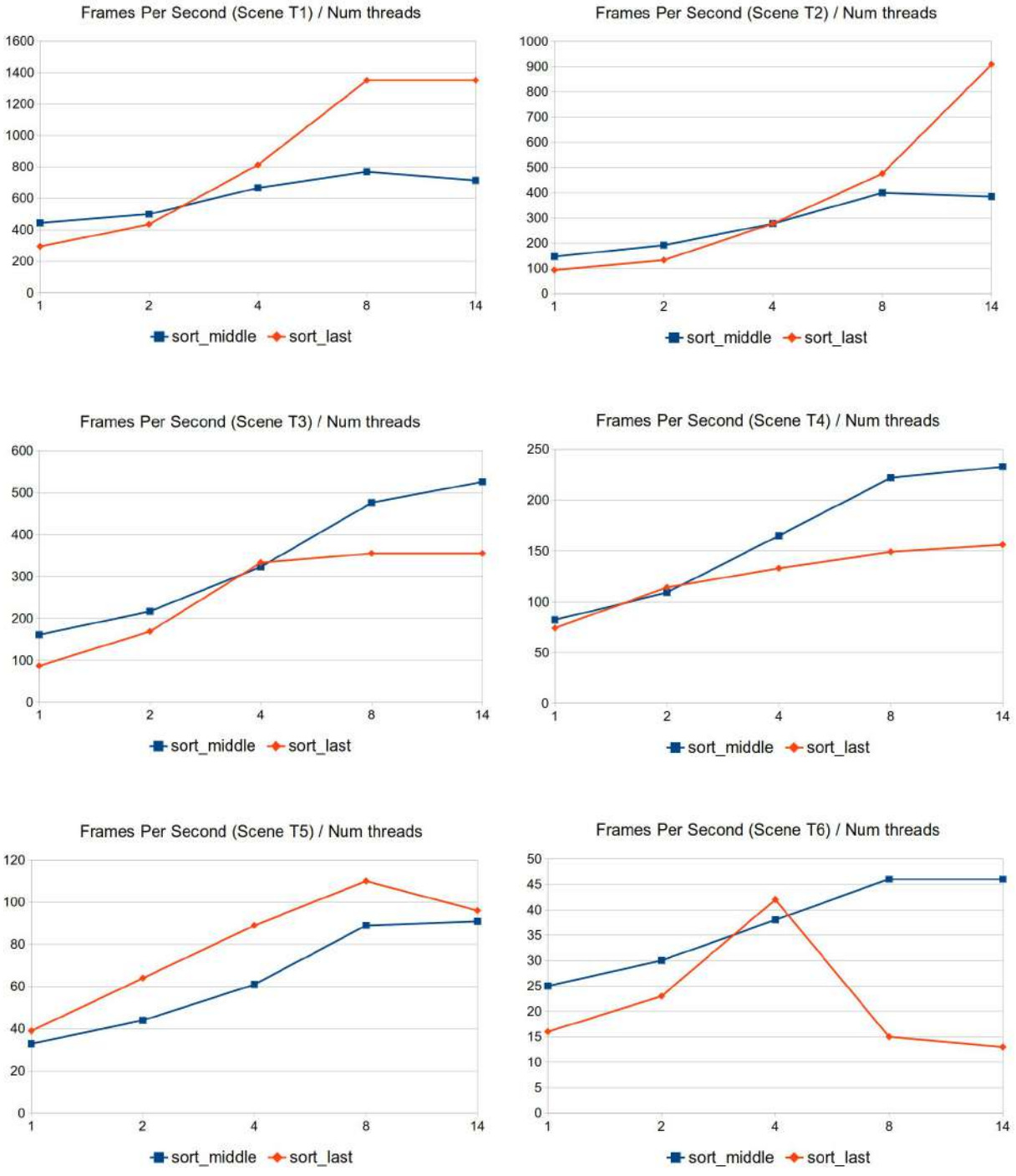

Fig. 8. Multithreading experiment. X axis — number of threads. Intel Xeon (5-2690 v4 2,6Ghz, 14 cores) CPU.

At last, we should make a note that in our comparison, a *sort-last* approach used pitch-linear buffer, and the *sort-middle* used a binned/tiled one. After comparing these two methods (pitch-linear vs tiled) in next subsection we can state that a *sort-last* approach can be even more efficient if it uses a tiled frame-buffer.

Framebuffer layout: pitch-linear vs tiled. Our next experiment was targeted to investigate memory subsystem efficiency when access frame buffer data. We assumed that frame buffer (and also depth buffer) access can be a bottelneck due to these buffers a priori can not be fit into the cache. Thus, some tiled frame buffer layout might be helpful because of less cache misses when accessing different rows (fig. 9).

We have investigated 4 different implementations (table 4):

1. pitch-linear frame and depth buffers layout. Default layout of 2d image by rows.
2. pitch-linear + binning overhead. This implementation has the same memory layout as a prevous one. However, it has bins for different 64x64 tiles and thus triangles that overlap several tiles should be duplicated. This implementation will show use binning overhead.
3. big tiles (64x64), i.e. bins. For this layout we split screen into 64x64 bins. For each bin inside we used pitch-linear layout.
4. Two-level tiling. At the first level, we split screen into 64x64 bins. At the second level we split each bin into 16x16 tiles thus making address linear inside the whole tile. Such layout will also allow wide vectors (for example, AVX512) being used for the whole 4x4 tile.

scene	pitch-linear	pitch-linear + bins	bins (64x64)	bins (64x64) + tiles (16x16)
T1	340 FPS	345 FPS	444 FPS	**476 FPS**
T2	113 FPS	103 FPS	147 FPS	**145 FPS**
T3	98 FPS	99 FPS	161 FPS	**169 FPS**
T4	99 FPS	82 FPS	132 FPS	**141 FPS**
T5	85 FPS	46 FPS	82 FPS	**91 FPS**
T6	40 FPS	20 FPS	33 FPS	**34 FPS**

Table 4: Comparison of pitch-linear and tiled frame buffer layouts. The higher is better. Blocked half-space algorithm was used (4x4). For this experiment we have used single thread and Intel Xeon (5-2690 v4 2,6Ghz) CPU. First column shows a default pitch-linear framebuffer layout. Second column demonstrates overhead we got from binned approach by itself: some triangles are duplicated due to they overlap several bins. Third column shows performance for binned approach and the last one — for two-level bins (64x64) + small tiles (4x4) approach.

Thus, memory layout has an extremely large impact on performance and tiled layout sould be definitely used.

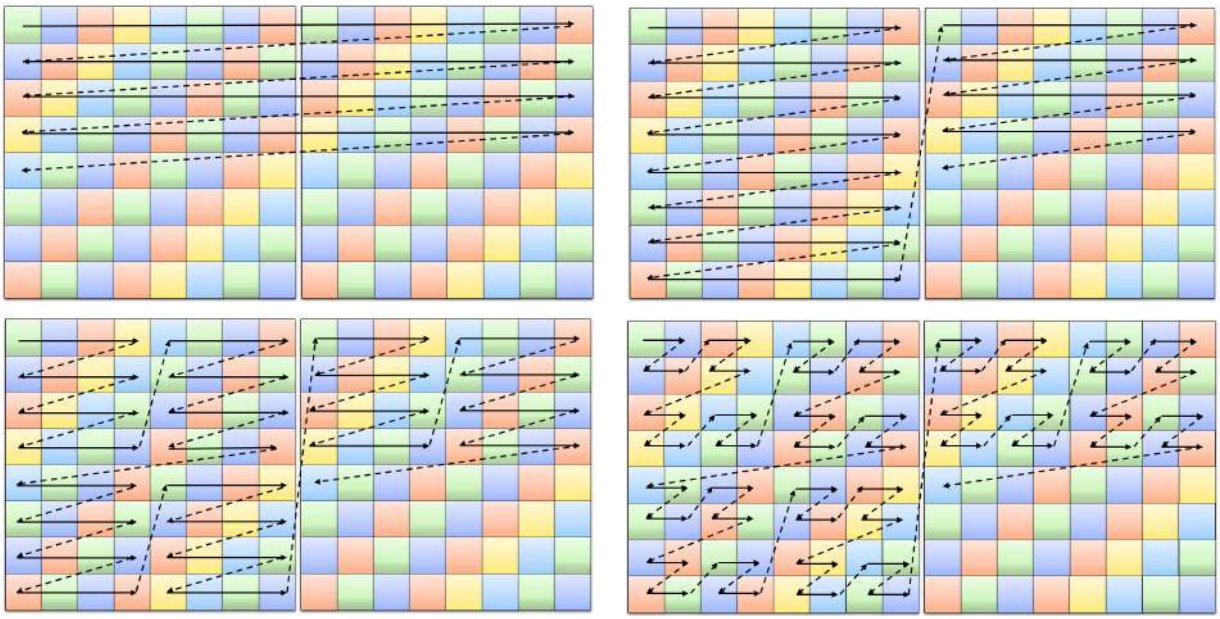

Fig. 9. Different framebuffer layout illustration. On this image, a bin size is shown to be 8x8, however on practice it was 64x64. Default pitch-linear is shown at top left image. Binned/tiled — top right. Two level (64x64 bins + 4x4 tiles) layout is shown at bottom left and Morton code layout [34] is shown at bottom right. With algorithmic point of view, the last one has better 2D locality [34]. However, Morton code evaluation is expensive and will also complicate half-space distances evaluation. At the same time, we would like to guarantee that all pixels in line have a subsequent addresses. This allow us reading/writing line of pixels with the single instruction and easily change length of instruction to test both SSE (for 4x4 tiles) and AVX (for 8x8 tiles).

CPU architecture: In Order vs Out Of Order. Our last experiment was aimed to study efficiency of different processor architectures for software graphics pipeline and rasterization. A trade off between performance and other CPU characteristics (such as energy efficiency, heat dissipation and cost) is essential for embedded systems. It is well known that the most significant performance gained on modern CPUs gives super scalar Out Of Order execution pipeline. This mechanism, at the same time, is the most expensive one. Our assumption is that with a large number of vector operations and independent instruction flow, software graphics pipeline should work well even on an in-order processor. Another reason we make this comparison is that in-order processors are more easily implementing precise exceptions which are important for safety-critical applications.

Since our blocked half-space algorithm is implemented via platform-independent light-weight vector library, we could easily port it to ARM. Unfortunalely our SSE1 implementation is heavily platform dependent (though, various options are exists [35]), so in this experiment we tested only pixel vectorization (blocked half-space algorithm). Using compiler explorer [31], we have counted instructions for different arhitectures and pixel processing modes (table 6). This information would allow us to more accurately evaluate how well the pipeline was loaded by the arithmetic instructions.

For this test we have selected several CPUs (table 7). First two processors (A83T and Cortex A53) are 2-way super scalar in-order machines. The i.MX6 (Cortex A9) has

scene	pitch-linear	bins (64x64)	bins (64x64) + tiles (16x16)
T1	340 FPS	444 FPS	**476 FPS**
T2	113 FPS	**167 FPS**	164 FPS
T3	98 FPS	161 FPS	**169 FPS**
T4	99 FPS	150 FPS	**155 FPS**
T5	85 FPS	102 FPS	**114 FPS**
T6	40 FPS	60 FPS	**63 FPS**

Table 5: Comparison of pitch-linear and tiled frame buffer layouts **without binning overhead**. The higher is better. Half-space block (4x4) algorithm was used. For this experiment we have used single thread and Intel Xeon (5-2690 v4 2,6Ghz) CPU. First column shows a default pitch-linear framebuffer layout. Second column shows performance for binned approach and the last one — for two-level bins (64x64) + small tiles (4x4) approach.

2-way super scalar out of order pipeline. The Core-i5 2410M (Sandy Bridge) has 4-way super scalar out of order pipeline. In addition to a wider pipeline, Sandy Bridge has many floating point ALUs, so it can execute 16 single precision floating point operations per clock (4 SIMD instructions per clock, each of 4 floats).

CPU arch/mode	Colored2D	Colored3D	Textured3D
x86/x64	68	100	290
ARMv7	79	110	500
ARMv8	60	86	250

Table 6: Comparison of instruction count per pixel for different rasterization states and CPU architectures. GCC compiler. 4x4 tiles were used. Colored2D includes color interpolation only. Colored3D — color interpolation with the perspective correction and a depth test. Textured3D adds bilinear texture fetch and perspective correction of texture coordinates to the previous mode. We have observed a significant increase in the number of instructions for ARMv7 and *Textured3D* mode due to spilling registers to memory. We used GCC 5.4.0 for both ARM cases.

We further introduce a special metric (equation 10, fig. 10) to compare *in-order* vs *out of order* from measured frames per second (table 7). We do this because in our experiments we used different CPUs with different architectures, manufacturing technology (for example 14 and 28 nm) and frequency. Our reason is straitforward: we don't want to compare the absolute performance values for different processors like table 7 does. Instead of that, we would like to approximately match instructions per clock for different CPUs to know whether *out of order* gives a benefit for our problem or not. Thus, if for a some CPU we have more instructions than for the other, we do not consider this a disadvantage for our comparison and we also do not want to take into account any inefficiencies introduced by the compiler. For this reason, the instruction count is in the numerator. At the same time frequency should be in denominator to bring all measures to a single frequency.

scene	A83T(ARMv7)		Cort.A53(ARMv8)		i.MX6 (ARMv7)		Core-i5 (x86/x64)	
	pure_cpp	SIMD	pure_cpp	SIMD	pure_cpp	SIMD	pure_cpp	SIMD
T1	16,7	17,5	26.3	35,9	13.9	19.2	96	191
T2	17,9	21,2	33,2	19,0	14,1	6,6	79	77
T3	5,8	6,5	7,4	14,5	4,5	6,3	27	67
T4	6,0	6,2	8,0	12,2	4,8	5,1	28	55
T5	4,5	4,6	6,1	7,0	3,7	3,0	22	33
T6	2,8	2,2	3,6	2,7	4,1	1,3	14	13

Table 7: Performance of a single-pixel (pure_cpp) and vectorized (SIMD) versions. Frames Per Second (FPS). Single thread, 1024x1024 and offscreen rendering. Binned frame-buffer (64x64 pixels) is used. A83T and Cortex A53 are in order machines; i.MX6 and Core-i5 are out of order ones. For this test we have used a laptop version of Core-i5 CPU (2410M, 2.3 GHz).

$$Efficiency = \frac{FPS * Instructions}{Frequency}. \tag{10}$$

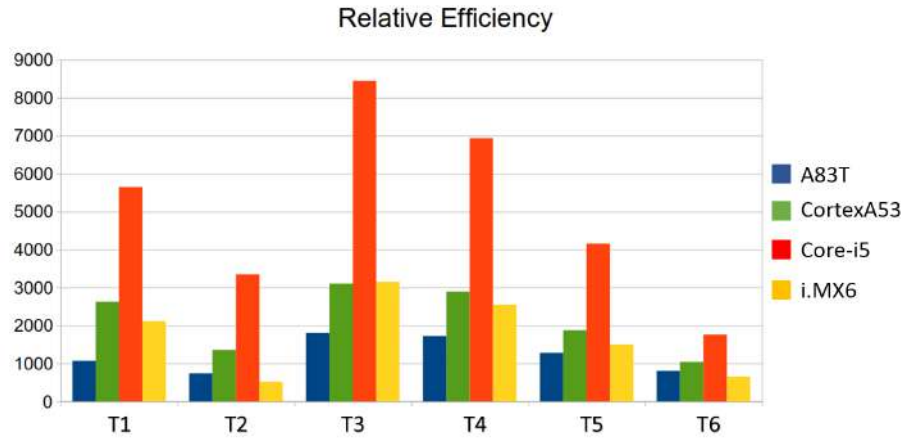

Fig. 10. Relative CPUs efficiency (equation 10). This efficiency could be thought as a relative instruction per clock (IPC). All histogram columns were obtained from SIMD columns of table 7.

Fig. 10 shows that the out of order (OOO) execution mechanism in itself gives only a very little benefit in average (compare A83T over i.MX6 — they both have 2-way execution pipeline, but i.MX6 have OOO and the A83T don't have it). The loss of the i.MX6 on T2 and T6 scenes can be easily explained — this is a result of expensive pipeline flush for the out of order CPU due to large amount of branch misprediction and complex code path in the blocked half-space algorithm; we rendered partially covered blocks with common (not vectorized) C++ code and therefore branch misprediction forces the CPU to flush pipeline and start executing another piece of code. Core-i5 has speculative execution mechanism and thus amortizes this problem. At the same time, in-order machine with greater amount of registers (ARMv8) shown better IPC (fig 10). Therefore, more registers combined with better code density for ARMv8 in Cortex A53 shown much better absolute performance than OOO execution added to ARMv7 in i.MX6 CPU (table 7).

2.3.3 Comparison with other implementations

We have compared our implementation to Mesa on A83T (ARMv7) and Mesa-OpenSWR on Intel Core-i7 CPUs (x86/x64). For Core-i7 we used high performance OpenSWR [6] implementation on Windows 7 and for A83T we used default Mesa 10.5.4 software rasterizer on Ubuntu Linux 16.04.6 LTS (xenial), BPI-M3 dev-board [36]. All comparisons were done in 1024x1024 resolution for windows and in 1024x640 for Linux on BPI-M3 due to maximum resolution limitation; please also note that this time (table 8) we have to include frame buffer display time into the comparison and therefore our numbers for A83T CPU in tables 8 and 7 are slightly differ.

Scn/CPU	A83T(ARMv7), 1 and 4 threads			Core-i7, 4 threads	
Scn/OGL	Mesa (1 thread)	Ours (1 thread)	Ours, 4 threads	OpenSWR	Ours
T1	5.6 FPS	7.0 FPS	**16.5 FPS**	**400 FPS**	240 FPS
T2	4.2 FPS	5.7 FPS	**15.0 FPS**	136 FPS	**150 FPS**
T3	1.0 FPS	6.7 FPS	**14.8 FPS**	**270 FPS**	210 FPS
T4	0.77 FPS	7.5 FPS	**8.2 FPS**	**220 FPS**	101 FPS
T5	0.45 FPS	5.0 FPS	**6.1 FPS**	**110 FPS**	63 FPS
T6	0.26 FPS	3.3 FPS	**4.8 FPS**	33 FPS	**40** FPS

Table 8: Comparison of our implementation to Mesa and OpenSWR OpenGL implementations. In this comparison, we used several optimizations altogether (such as tiled frame buffer and multithreading).

On x86/x64 our implementation [37] could not beat OpenSWR on pixel-heavy scenes (table 8). However, we were faster on T2 and T6 scenes where our combined approach (sse4+sse1, section 2.3.2) has shown its advantage. Our code was designed to quickly test the maximum number of different rendering techniques. So, considering that OpenSWR is made by Intel for the x86/x64 architecture only (and it simply can not run on the others), it would be naive to expect excellence from our experimental implementation for all cases. We believe that OpenSWR generates better vectorized code (processing a half of 4x4 tile with a single AVX instruction, for example). Also OpenSWR could proceed better with multithreading due to our experiments revealed problems for both studied methods (*sort-middle* and *sort-last*).

On the other hand, with the same software implementation, we can significantly outperforms default Mesa rasterizer on ARM which was the only avaliable software solution for BPI-M3 board during our work with it; according to our information there is no working graphics driver for Ubuntu Linux on BPI boards and therefore the whole rendering is performed actually in the software. Many other developent boards suffer the same problem on practice (along with patent issues [38]).

3 CONCLUSIONS

In this article we investigated various high performance graphics rasterization algorithms and techniques to be accelerated on different modern processor architectures. Prac-

tical and scalable solutions for software implementation of OpenGL graphics pipeline on modern multicore CPUs were elaborated.

The experimental results demonstrated unexpected results — even on fairly simple test scenes popular methods (*sort-middle* and blocked half space) substantially lost to rarely used ones (*sort-last* and simple half space). Relying on these results, a combined approach (blocked half-space + half-space) accelerated with SIMD instructions was introduced that have beaten extremely optimized OpenSWR implementation for 2 scenes. At the same time our implementation outperform default Mesa rasterizer on ARM CPUs an order of magnitude, which demonstrates the relevance of this area of research. We also offered a special metric for benchmarking relative Instructions Per Clock (IPC) for different CPUs without special tools, and this metric shows relatively low efficiency of the out of order mechanism itself for pixel processing. The more particular conclusions are shown further:

1. Half-space rasterization methods are absolutely better than scanline ones;
2. SIMD pixel processing for blocked half-space rasterizer gives essential benefit, but has limitations:
   (a) small triangles degrade performance, so the combined approach should be used;
   (b) wide vectors on architectures with low amount of vector registers may not have benefit due to high register pressure, increased number of instructions and spilling intermediate results to memory (table 7).
3. Even in such a computationally intensive task as pixel processing during rasterization (where the ratio of computational instructions to memory operations is greater than 100:1), memory access is still a seriously performance limit. Tiled frame buffer and depth buffers layouts increase performance up to 60%;
4. Despite our multithreading implementation is far from perfect (we don't have linear acceleration for most cases), we believe that the *sort-last* approach is more perspective, although it is non trivial.
5. For the considered problem out of order (OOO) machines have essential benefit if the OOO machine has significantly larger maximum instructions per clock than an in order one. It is more essential to have larger maximum thoroughput of floating point instructions (i.e. have for floating point ALUs).

When we first started our work, we were sure that it would be more technical and that all the research that could be done in this area had already been done due to the popularity of GPUs today. However, on practice, everything turned out to be differently. We could not find a single optimal approach for the implementation of software rasterization and graphics pipeline. Moreover, we found that with the advent of GPUs, researchers mostly ignore real-time software rendering. At the same time, processors were actively developing, so we believe that this field is the fertile ground for the future performance research.

REFERENCES

[1] T. Akenine-Moller, E. Haines, N. Hoffman, P. Angelo, I. Michal, and S. Hillaire.*Real-Time Rendering*, 4 ed. Natick, MA, USA: A. K. Peters, Ltd. (2018).

[2] G. K. Borovin and V. V. Lapshin, "Motion control of a space robot," *Mathematica Montisnigri*, **41**, 166–173 (2018). http://www.montis.pmf.ac.me/vol41/13.pdf

[3] K. V. Ryabinin and S. I. Chuprina, "Using scientific visualization systems to automate monitoring of data generated by lightweight programmable electronic devices," *Programming and Computer Software*, **44** (4), 278–285 (2018).

[4] OpenGL, "The mesa 3d graphics library." (2019) https://www.mesa3d.org/ (accessed August 8, 2019)

[5] Microsoft, "Windows advanced rasterization platform (warp) guide." (2018) https://docs.microsoft.com/en-us/windows/win32/direct3darticles/directx-warp (accessed August 8, 2019)

[6] OpenSWR, "A high performance, highly scalable software rasterizer for opengl." (2019) https://www.openswr.org/index.html (accessed August 8, 2019)

[7] M. M. Zieojevic, D. S. Jokanavic, and D. Baralic, "Sofware "cindrella" and its application in visualization of physics and mathematics," *Mathematica Montisnigri*, **34**, 86–93 (2015). http://www.montis.pmf.ac.me/vol34/6.pdf

[8] W. J. Bouknight, "A procedure for generation of three-dimensional half-toned computer graphics presentations," *ACM*, **13**(9), 527–536 (2019). http://doi.acm.org/10.1145/362736.362739

[9] B. Molkenthin, "Software rasterization algorithms for filling triangles." (2010) http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html (accessed August 8, 2019)

[10] J. E. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Syst. J.*, **4** (1), 25–30 (1965). http://dx.doi.org/10.1147/sj.41.0025

[11] A. Fujimoto, T. Tanaka, and K. Iwata, "Arts: Accelerated ray-tracing of system tutorial," *Computer Graphics, Image Synthesis, Computer Science Press, Inc, New York, NY, USA,.*, 148–159 (1988). http://dl.acm.org/citation.cfm?id=95075.95111

[12] S. Laine and T. Karras, "High-performance software rasterization on gpus." 79–88 *New York, NY, USA*: *ACM*. (2011) http://doi.acm.org/10.1145/2018323.2018337

[13] J. Beam, "Tutorial – introduction to software-based rendering: Triangle rasterization." (2009) https://www.joshbeam.com/articles/triangle_rasterization/ (accessed August 8, 2019)

[14] J. Pineda, "A parallel algorithm for polygon rasterization," *SIGGRAPH Comput. Graph.*, **22** (4), 17–20 (1988). http://doi.acm.org/10.1145/378456.378457

[15] P. Mileff, K. Nehéz, and J. Dudra, "Accelerated half-space triangle rasterization," *Acta Polytechnica Hungarica*, **12** (7), 217-236 (2015). https://www.uni-obuda.hu/journal/Mileff_Nehez_Dudra_63.pdf

[16] M. Abrash, "Rasterization on larrabee," *Dr. Dobbs*, **37** (2009). http://drdobbs.com/high-performance-computing/217200602

[17] Scratchapixel, "Rasterization: a practical implementation," (2016). https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-practical-implementation (accessed August 8, 2019)

[18] B. Barladian, A. Voloboy *et al.*, "Efficient implementation of opengl sc for avionics embedded systems," *Programming and Computer Software*, **44** (4), 207–212 (2018). DOI: 10.1134/S0361768818040059

[19] VCDECIDE, "Fighting force (nintendo 64 vs playstation) side by side comparison," (2019). https://www.youtube.com/watch?v=izOPJA2MyC0 (accessed August 8, 2019)

[20] Wikipedia, "Playstation technical specifications." (2019) https://en.wikipedia.org/wiki/PlayStation_technical_specifications (accessed August 8, 2019)

[21] E. Koskinen, M. Parkinson, and M. Herlihy, "Coarse-grained transactions," *SIGPLAN Not.*, **45** (1), 19–30 (2010). http://doi.acm.org/10.1145/1707801.1706304

[22] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient gpu architectures," *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 86–98 (2013). http://doi.acm.org/10.1145/2540708.2540717

[23] Z. Bethel, "A modern approach to software rasterization." University Workshop, Taylor University (2011).

[24] P. Mileff and J. Dudra, *Advanced 2D Rasterization on Modern CPUs.* Springer International Publishing, 63–79 (2014). https://doi.org/10.1007/978-3-319-01919-2_5

[25] P. Mileff and J. Dudra, "Modern software rendering," *Production Systems and Information Engineering*, **6**, 55-66 (2012).

[26] Y. Kim, J.-E. Jo, H. Jang, M. Rhu, H. Kim, and J. Kim, "Gpupd: A fast and scalable multi-gpu architecture using cooperative projection and distribution," *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 574–586 (2017).

[27] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "Gramps: A programming model for graphics pipelines," *ACM Trans. Graph.*, **28**(1), 4:1–4:11 (2009). http://doi.acm.org/10.1145/1477926.1477930

[28] M. Kenzel, B. Kerbl, D. Schmalstieg, and M. Steinberger, "A high-performance software graphics pipeline architecture for the gpu," *ACM Trans. Graph.*, **37** (4), 140:1–140:15 (2018).

[29] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, " *ACM SIGGRAPH 2008 Papers* "Larrabee: A many-core x86 architecture for visual computing *SIGGRAPH '08.* 18:1–18:15 *New York, NY, USA*: *ACM.* (2008) http://doi.acm.org/10.1145/1399504.1360617

[30] K. Group, "dfdx, dfdy – return the partial derivative of an argument with respect to x or y." (2019) https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/dFdx.xhtml (accessed August 8, 2019)

[31] M. Godbolt, "Compiler explorer," (2019). https://godbolt.org/ (accessed August 8, 2019)

[32] cppreference.com, "std::atomic_flag." (2019) https://en.cppreference.com/w/cpp/atomic/atomic_flag (accessed August 8, 2019)

[33] C. Moody Camel, "A fast multi-producer, multi-consumer lock-free concurrent queue for c++11." (2019) https://github.com/cameron314/concurrentqueue (accessed August 8, 2019)

[34] K. P. Lorton and D. S. Wise, "Analyzing block locality in morton-order and morton-hybrid matrices," *SIGARCH Comput. Archit. News*, **35** (4), 6–12 (2007). http://doi.acm.org/10.1145/1327312.1327315

[35] K. A. Batuzov, "The use of vector instructions of a processor architecture for emulating the vector instructions of another processor architecture," *Programming and Computer Software*, **43** (6), 366–372 (2017).

[36] Sinovoip, "Banana pi m3 octa core development board." (2019) http://www.banana-pi.org/m3.html (accessed August 8, 2019)

[37] V. Frolov, B. Barladian, and V. Galaktionov, "Swgl: Experimental opengl1 software implementation; github repository," (2019). https://github.com/FROL256/SWGL (accessed August 8, 2019)

[38] T. A. Polilova, "Ethical norms and legal framework of scientific publication," *Mathematica Montisnigri*, **45**, 129–136 (2019).

Received October 5, 2019