
Production Systems and Information Engineering
Volume 10 (1), pp. 1–14
doi: 10.32968/psaie.2022.1.1.

EFFECTIVE PIXEL RENDERING IN PRACTICE

Péter Mileff
University of Miskolc, Hungary
Department of Information Engineering
mileff@iit.uni-miskolc.hu

Judit Dudra
Bay Zoltán Nonprofit Ltd. for Applied Research, Hungary
Department of Structural Integrity and Production Technologies
judit.dudra@bayzoltan.hu

Abstract. The graphics processing unit (GPU) has now become an integral part of our lives through both desktop and portable devices. Thanks to dedicated hardware, visualization has been significantly accelerated, softwares today only use the GPU for rasterization. As a result of this development, now we use only triangle-based rendering, and pixel-based image manipulations can only be performed using shaders. It can be stated that today's GPU pipeline cannot provide the same flexibility as the previous software implementation. This paper discusses an efficient software implementation of pixel-based rasterization. After reviewing the current GPU-based drawing process, we will show how to access pixel level drawing in this environment. Finally, a more efficient storage and display format than the classic solution is presented, which performance far exceeds the previous solution.

Keywords: Software rendering, optimization, pixel rasterization

1. Introduction

High quality computer visualization is one of the most important areas and requirements these days. Modern image synthesis is now practically present in almost every field: in computer gaming, in multimedia applications, in design or other graphics softwares. Today, the realm of visualization is dominated by GPU-based rendering. The hardware itself is beyond great development, the industry has nearly 20 years of development behind it. Its main driving force was the computer game industry, where there is a constant pursuit for better rendering quality even closer to reality. The use of GPUs has become virtually standard, all mainstream operating systems have closed or even open

source drivers, despite the fact that the architecture of video cards is not open to those engineers who would like to write a driver.

However, the GPU is not a miracle tool. Their price and power consumption has increased significantly in recent years. The power consumption of a modern video card can be even 280 W during active usage. Their appearance significantly transformed the previously known process of visualization. Developers had to break with previous software visualization methods and adapt to the available programmable API of video card drivers (Glide, OpenGL, DirectX). This particular process of visualization has not changed much over the years. Initially, a so-called fixed-function pipeline was used in the image synthesis, later with the advent of shaders, various parts of the pipeline became programmable for the developers, thus paving the way for custom solutions used in programs.

To put it very simply, we can say that in the pre-GPU period of visualization, the programmer had full control over every pixel on the screen or in the main memory. With the advent of GPUs, this high degree of flexibility has disappeared, especially before the shader world. Today's visualization is based on a wireframe model. Every object must be a mesh of triangles that defines the "shape" of the object. The texture is applied to this wireframe using the perspective texture mapping technique. For fast visualization, the video card is designed to have its own memory. It must contain all the elements and objects that will be drawn. Therefore, one of the first steps of any software (e.g. loading a level) is to load all the elements into the GPU RAM before drawing. So the GPU is like a separate island.

The triangle-based model is efficient and suitable for most display purposes [1]. However, the programmer is forced to the solutions provided by the GPU APIs. It would be the use of shaders that gives us the flexibility we had in early times to work with even pixels, but unfortunately these fixed processes do not satisfy all needs. Today, we can no longer deal with pixels and the entry point for those who want to learn computer visualization has increased significantly.

The purpose of this article, therefore, is to examine the possibilities by which the pixel-level programmability can be achieved. We will show how to draw efficiently with the help of the CPU, which will be verified with measurement tests.

2. General steps of GPU rendering

The methods that can be used in today's computer visualization are limited, determined by GPUs. GPU-based display is currently present in virtually any

area of the world, dominating the area. Therefore, we can only effectively apply in practice methods that are supported by the GPU and the driver. Typically, this type of task is pixel-based drawing. Modifying or accessing a pixel is not explicitly supported, and there are situations where it would be necessary.

In the following, the general process of GPU rendering will be reviewed, and then we will examine how to make possible the pixel-based drawing on today's GPUs.

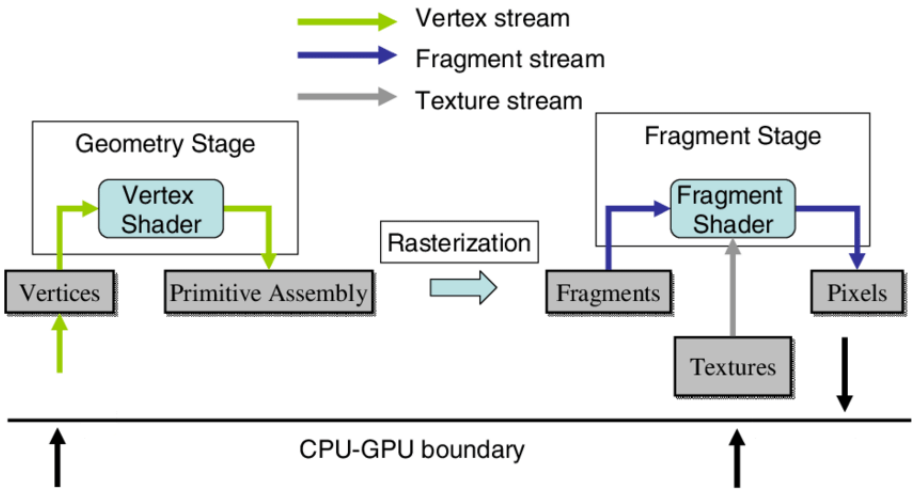

Figure 1. GPU rendering process from a general perspective

If we look at the figure, it is clear that the rendering process can be divided into two main parts on the GPU side. The first part (Geometry Stage) is responsible for vertex transformation. All operations related to any vertices are performed here. Typical tasks are: transforming vertices into the right place in the virtual world based on the world matrix of game objects, possibly calculating certain types of lights, or even inserting new vertices. So it is clear that anyone who wants to see anything on the screen has to work with vertex-based models and data in any case. And this is practically a big constraint compared to the software model where we want to display something without a GPU. For executing the vertex phase, the vertex (and geometry) shader is responsible.

After the geometric phase, we reach the level of the fragments. This is where the actual rasterization takes place on primitives assembled and transformed in the vertex phase. The fragment shader, which is executed once for each fragment, is responsible for executing the process. The output of this stage is the color of the current fragment.

In practice, this stage gives the opportunity to the programmer to control how the color of a given fragment should be modified. Different fragment-level algorithms can be used here. This typically includes the calculation of Per-Pixel lighting, but also the application and calculation of shadows or other image space effects (e.g. Bump mapping, Normal Mapping, etc.). While we have the ability to influence pixels, this solution offers nowhere near as much flexibility as software rendering, where we essentially do what we want with a pixel.

Current GPU-based rasterization works well, a huge industry is built on it. The advantage is that – according to a fixed programming logic – the programmers get a uniformly higher level of abstraction. There is no need to program the visualization at a low level. With the advent of shaders, it was a big step in GPU-based visualization. This made it possible to replace the fixed-function pipeline and introduced better programmability from a developer side. Today, all software need to use shaders if they use GPU ror rendering, this is a hard requirement from the GPU APIs (OpenGL, DirectX). Although they greatly improve the programmability of GPUs, cannot and will probably never achieve the capabilities of software rasterization. As in this case they would have to become virtually a CPU, losing the acceleration benefits of a specialized architecture. A further disadvantage is that, for beginners who want to deal with computer visualization, the entry level has been greatly increased. Creating an API-level "Hello World" application is not easy either, as geometry, matrices, shaders, and other tasks need to be addressed, and two-dimensional rendering is no exception.

3. Pixel level software rasterization

In the early stages of computer visualization, all graphical applications used only software, CPU-based rendering, because no GPU device was available at the time. Despite the fact that the hardware at the time did not perform very well, the memories were slow, we still got very nice results if we look at the history of the computer games. These softwares handled the pixels themselves, and according to the standardized interface of VESA VBE (VESA BIOS Extensions), no special video card driver or other API (OpenGL, DirectX) was required. VBE was a complement to the INT 10h BIOS video features. Achieving the desired resolution and pixels (read / write) from a few lines of program code was feasible even for a beginner. Games born in the DOS period were built on this solution, using so-called memory mapped display RAM. The video card memory in windows (128K, 64K, 32, 32) [2] was practically directly accessible to the program. There were no shaders or other API calls, if the correct byte was written to the correct location in the display RAM, the result was

immediately visible. However, this possibility has disappeared with the advent of graphics cards in today's sense.

3.1. Pixel based rendering on modern GPU-s

With the advent of modern video cards, direct support for pixel-based drawing has been removed. So if someone wants to work with pixels today, they have to look for other ways. Unfortunately, there is little information available on the internet to achieve this effectively. In practice, two possible directions are available, which are described below.

3.1.1. Drawing with the operating system

Every modern operating system offers the ability to work with pixels. Of course these solutions in the background use the video card hiding the shader and other difficulties from the programmer. For example, in Windows, we can create Bitmap objects that have a *SetPixel* function. In addition, the operating system has a *BitBlt* function that allows block-based transfers of rectangular pixel arrays from one source to another. This essentially makes software drawing possible.

For Linux systems, programmers have a similar opportunity. For systems that use the X11-based display, XLib is available as an operating system-level library. XLib has a lot of features. The easiest way is to draw something to the screen we need to create a Display object that represents the screen / window, and use XLib functions such as *XDrawPoint*, *XDrawLine*, etc. An alternative to Windows's BitBlt function is the *XCopyArea* function.

Pixel-level drawing is thus possible with the above solutions, but in practice programmers do not choose these solutions. One reason for this is that the above solutions are platform dependent. Of course this limitation can also be solved by developing an abstraction layer above these. However, development is more difficult in this case because it must be tested on multiple operating systems at the same time, at least until the finished abstraction layer can serve all needs and is massively stable. The most important disadvantage is that since pixel drawing is implemented through the operating system, it is not the most efficient solution in terms of performance.

3.1.2. Rasterization through hardware APIs

The capabilities of today's video cards can only be properly exploited through the two available APIs (OpenGL, DirectX). If we want to achieve high performance, it is advisable to start from these, follow the process that is required

and expected by the APIs. In this paper, we will choose the OpenGL API to demonstrate the implementation of pixel-based visualization.

Before describing the only suitable option for pixel drawing, it should be mentioned that there was an option in OpenGL from the beginning that made this possible. The *glDrawpixels* function was an extremely easy-to-use option for moving a rectangular set of pixels, a specific area of the main memory to the framebuffer of the video card. Software rendering is accomplished the way that the programmer created an array of bytes corresponding to a given resolution and color depth in the main memory and then drawing into it. Drawing was essentially a series of memory operations. Then, after drawing, we could use *glDrawpixels* to move the finished array to the framebuffer, during which the contents of the array were displayed on the screen. Although it was a very convenient solution, it was removed from the OpenGL 3.2 standard and is no longer supported. And for portable / mobile devices, it was never part of the standard (OpenGL ES). In addition, the performance of the method was not necessarily satisfactory. The reason for this is described later along with the presented methods.

3.1.3. The method of drawing at the pixel level

The GPU is designed to work efficiently with vertex and texture data. It is therefore worth choosing a solution that uses these processing steps at some level. By default, no special APIs other than OpenGL or DirectX are required to implement pixel-level drawing. The key idea of the solution is to use an orthogonal projection (like in 2D games) and create a texture that matches the resolution of the screen. This screen texture is formed by two triangles and is defined and positioned depending on the resolution so that it completely covers the screen. With this, we practically create a virtual canvas where we need to draw somehow.

However, the process of drawing is not entirely trivial. This is because any graphical object we want to display, the graphical APIs (OpenGL, DirectX) require that all the data in the object should be in GPU memory (except for very large worlds [streaming]). Of course, this is where the need arises for GPUs to have more and more memory, as all the objects of the game to be displayed must be available in the GPU. For OpenGL, VAO / VBO is the expected storage structure for efficient rasterization speeds. The problem comes from the fact that the GPU stores the data structure, the memory area which needs to be modified from the CPU side. There are several ways to modify the pixels of a texture object, but in each case we have to move the texture data and memory area between the GPU memory (GPU side) and the

main memory (CPU side) several times. The following figure illustrates this process:

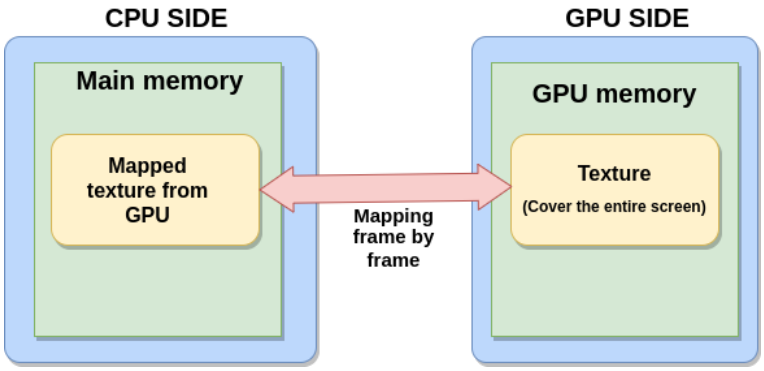

Figure 2. The process of software rendering on today GPU's

In the first case when we want to draw, the data movement is performed from the GPU to the main memory, and the second time, when we are done with the pixel modification, the data is loaded into the video RAM again. Plus, it's all at least 50-60 times per second [1], which is a critical requirement for games. In order to illustrate the amount of data being transferred, a 1920 × 1080 pixel image in RGBA quality means 8,294,400 bytes of movement, which must be done 50 to 60 times. In a computer game, we rarely work with images of this size (possibly in case of the background), but there are usually many smaller image on the screen at a time.

So whatever solution we choose from those provided by the OpenGL or DirectX APIs, we are limited by the BUS speed of the current architecture. Most modern machines today use PCI Express 4.0, a 2011 standard with a theoretical maximum baud rate of 16 GT / s bits that just doubles PCI Express 3.0. Although the PCI Express 5.0 (2017) and PCI Express 6.0 (2022) standards are already available, the family of hardware that supports them will not be available for years to come.

There are several ways to modify the pixel data in the GPU memory on the CPU side in OpenGL:

- **glTexImage2D/ glTexSubImage2D:** a classic solution is to use the glTexImage2D function to create a texture and the glTexSubImage2D function to modify parts of it or even the whole as needed.
- **Pixel Buffer Object (PBO):** The main advantage of PBO is fast pixel data transfer to and from a graphics card through DMA (Direct

Memory Access) without involving CPU cycles. And, the other advantage of PBO is the asynchronous DMA transfer.

4. Software rendering in practice

The practical effectiveness of software rasterization depends largely on how it is implemented. Achieving high performance and optimization is essential and requires serious expertise in many cases. In the following, we review some important rules and techniques.

4.1. Storing the image data

Today's software works with 32-bit (4 byte – RGBA) color depth images. Textures can be divided into two groups based on color channels: there are image elements with and without alpha channels. The distinction is important because the display procedure and the rendering acceleration options are different for the two groups. Images without alpha channel have no transparency, only RGB color components. This means that any two objects can be drawn on top of each other without having to mix the colors of the objects below each other, making their drawing mechanism faster and easier [3].

Handling images with an alpha channel is not more difficult, but needs more computational power. This is because within a texture, the transparent and non-transparent parts (e.g character animation, cloud, particle effects, etc.) can change randomly, which causes the drawing to be performed pixel by pixel [3]. When it comes to software rasterization, the pixels of an image object are usually stored in main memory in the following form:

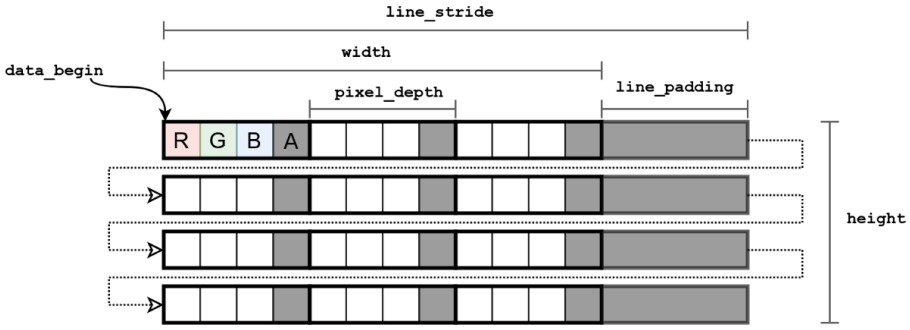

Figure 3. The common way to stora pixels in the main memory

The pixels are composed of four components (R, G, B, A), each component is 1 byte. These value types are important when the pixel array needs to be

moved to the video RAM. On the CPU side, when calculations are performed, the pixels are often stored as floats in the interval [0,1].

In the following, through C++ examples we will show how to store and draw the pixels and the image itself efficiently. It is important to note that because drawing per pixel requires significant performance due to the large amount of memory read and write operations, any non-optimized operation in the drawing cycle represents a strong decrease in final performance.

4.1.1. Naive pixel rendering

The simplest form of pixel representation is to store each component of a pixel as a separate unsigned char (0–255). This storage format, as long as we do not require any special storage format, is virtually the same as the texture (RGBA) created in the GPU storage format which can be accessed when mapping it to the CPU side. Based on this, an image storage structure can be easily created:

```
struct texture_t
{
    unsigned int width;
    unsigned int height;
    unsigned int internalFormat;
    unsigned char *texels;
};
```

From the above definition, texels will be the memory block that will contain the pixels of the loaded image, which will be created for an RGBA image as follows:

```
texinfo.internalFormat = 4;
texinfo.texels = (unsigned char*)malloc(sizeof(unsigned char) *
texinfo.width * texinfo.height * texinfo.internalFormat);
```

Drawing a pixel array created in this way is relatively simple. The example assumes that the texture and the underlying screen-covering mesh have been already created on the GPU side. Texture (screen) pixels can be accessed by covering them with the *gGraphics.GetFrameBuffer()* wrapper class. The mechanism of drawing in this case:

```
unsigned int wHelp = mTexture.width*4;
CFrameBuffer fbuffer = gGraphics.GetFrameBuffer();

for(unsigned int i=0; i < mTexture.width; i++){
    for(unsigned int j=0; j < mTexture.height; j++){
     unsigned int iHelp = i*4;
       unsigned char a = *(mTexture.texels + j*wHelp + iHelp + 3);
       unsigned char r = *(mTexture.texels + j*wHelp + iHelp + 2);
```

```
        unsigned char g = *(mTexture.texels + j*wHelp + iHelp + 1);
        unsigned char b = *(mTexture.texels + j*wHelp + iHelp);

        // no draw if pixel is fully transparent
        if (a == 255) continue;

        if (position.x + x >= fbuffer.width || position.y + y >= fbuffer.height)
           continue;

      unsigned int offset = position.y+y*fbuffer.width + position.x + x;

      fbuffer.mFrameBuffer[offset] = r;
      fbuffer.mFrameBuffer[offset + 1] = g;
      fbuffer.mFrameBuffer[offset + 2] = b;
      fbuffer.mFrameBuffer[offset + 3] = a;
  }
}
```

Because these types of images also contain an alpha channel, they are drawn pixel by pixel. If there are several larger objects on the screen, refreshing the screen 50-60 times per second will require significant performance. Of course, the above sample code already includes one or two optimizations *(wHelp, iHelp)* that will help, but unfortunately they won't be enough either. It is important to note that the above example does not deal with the case, when writing a given pixel, it should be mixed with the color of the pixel below it.

Storing pixel components separately is not only ineffective for drawing. Any other operation required to perform on the image (rotate, flip, stretch, etc.) will never be satisfactory.

4.1.2. Advanced pixel representation

A more efficient approach is to somehow handle the components of the pixels together. In the case of RGBA, each component requires 1 byte, so 4 bytes = 32 bits to store one pixel. And this size practically corresponds to an integer value. So the components of a pixel can be "wrapped" into an integer variable with the following bit positions: blue component: 7-0, green component: 15-8, red component: 23-16, alpha component: 31-24.

To take advantage of this new form of storage, existing pixels in main memory must be converted to this form. It is best to do this immediately after loading the images from the file system. The following sample code illustrates this conversion:

```
int texel_length = mTexture.width*mTexture.height;
```

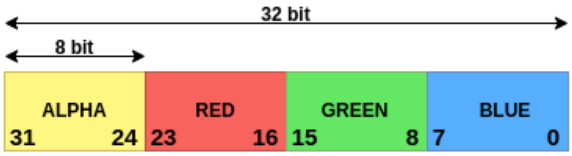

Figure 4. Pixel component in an 32 bit integer

```
mTexture.int_texels = (uint32_t *)malloc(sizeof(uint32_t)*texel_length);

for(unsigned int i=0; i < mTexture.width; i++){
    for(unsigned int j=0; j < mTexture.height; j++){

    unsigned char r = *(mTexture.texels+j*mTexture.width*4 + (i*4) + 2);
    unsigned char g = *(mTexture.texels+j*mTexture.width*4 + (i*4) + 1);
    unsigned char b = *(mTexture.texels+j*mTexture.width*4 + (i*4));

    uint32_t rr = (uint32_t)(r);
    uint32_t gg = (uint32_t)(g);
    uint32_t bb = (uint32_t)(b);
    uint32_t aa;

    unsigned char a = *(mTexture.texels+j*mTexture.width*4 + (i*4) + 3);
    aa = (uint32_t)(a);

    uint32_t color = (aa << 24) | (rr << 16) | (gg << 8) | bb;
    mTexture.int_texels[j * mTexture.width + i] = color;
    }
}
```

Although it is not specifically highlighted in the sample code, the *mTexure* structure includes a `uint32_t *int_texels;` this variable has also been added to store an array of integer-based pixels. The process goes through each pixel and creates its 32-bit integer equivalent with the required bit operation:

```
uint32_t color = (aa << 24) | (rr << 16) | (gg << 8) | bb;
```

The final result is that each pixel is represented by a single integer value, which will provide significant benefits in further operations.

4.1.3. Working with integer pixels

One of the effective way of increasing the performance of computer programs from any field is to be able to process a set of data in larger units or blocks.

The above integer-based pixel storage conforms to this principle. Whatever operation is performed, it is expected that fewer operations or iterations will be required to process the pixel set. In the following, we present two main operations.

Drawing the pixel buffer:

Of course, the most important operation is drawing, but virtually any operation we take will be very similar in nature to the code shown below:

```
CFrameBuffer fbuffer = gGraphics.GetFrameBuffer();

for(unsigned int i=0; i < mTexture.width; ++i) {
   for(unsigned int j=0; j < mTexture.height; j++) {
      uint32_t w = j * mpTexture.width + i;
      uint32_t color = *(m_pTexture->int_texels + w);

      unsigned int offset = (position.y+j)*fbuffer.width + (position.x + i);

      if (offset < screen_buffer_size)
                fbuffer.mFrameBuffer[offset] = color;
   }
}
```

It is clear that the drawing operation has been significantly simplified, we get a much more transparent solution with this new structure. In addition, of course, we hope a significant improvement in the speed of drawing.

Vertical mirroring:

Another interesting operation is to mirror the set of pixels which is often required for graphics applications. This example shows vertical mirroring. The integer pixel structure is also a big advantage in the operation.

```
void CTexture::FlipVertical()
{
    uint32_t *texels = new uint32_t[mTexture.height * mTexture.width * 4];

    for(unsigned int i=0; i < mTexture.height; i++){
      mempy(texels+(((mTexture.height-1)-i)*mTexture.width),
          mTexture.int_texels+i*mTexture.width,mTexture.width*4);
    }

    delete [] m_pTexture.int_texels;
    mTexture.int_texels = texels;
}
```

5. Test Results

Different rasterization techniques have different performances. In the following, we demonstrate the performance (Frame-Per-Second) of the two described pixel drawing methods using several test cases. The test programs were implemented using the C++ language and the GCC 11.2 compiler, and the measurements were performed on a Core i7-9700 3 GHz CPU on a Linux operating system. An Intel UHD Graphics 630 integrated video display was used for the tests. To display the software framebuffer, we used the OpenGL framework, where we chose a full-screen texture solution created in the GPU memory. We did not use CPU side parallelization in the rasterization of the pixels, the results reflect the performance of only one processor core.

The tests can be divided into three groups based on the image size used. The largest image is 1024×1024, then 256×256 is finally 64×64. Every image has a RGBA type. In order to show the efficiency of the algorithms, we drew from the same type of image in both small and large volumes.

		Rasterization speed (FPS)	
Size of texture	**Count**	**Classic rendering**	**Integer based pixel rendering**
1024x768	1	108	550
1024x768	10	15	398
256x256	10	139	658
256x256	100	16	323
64x64	100	225	727
64x64	200	124	655

Figure 5. Benchmark results

The results show well that integer-based pixel storage and drawing performed with the best results in all cases which meets the expectations. It is clear that a computer game, which needs high performance cannot be built upon the classic byte drawing technique using a single CPU core, while an integer-based solution would already be suitable for this task. In addition, it should not be forgotten that only one CPU core is involved here. With the help of multithreaded drawing, the above results can be far surpassed.

6. Conclusion

In modern computer visualization, today we can only encounter GPU-oriented approaches. The area has developed a lot in recent years and today has a huge market to offer. As development progressed, although better and better tools came into the hands of the programmer, the developer lost flexibility in graphics programming. Today, everything is made on a triangular basis, the pixel-level image manipulations and the inclusion of a CPU side in the rendering process has disappeared. This paper presented a technique that, although it is cumbersome, still allows the use of software solutions. It is clear from the performance data that high-level solutions could be developed that are not based solely on the GPU. The resulting applications are less dependent on the GPU, increasing cross-platform interoperability, and pixel-level accessibility would give new creative possibilities to developers.

References

[1] Mileff, P., Nehéz, K., and Dudra, J.: Accelerated half-space triangle rasterization. *Acta Polytechnica Hungarica*, **12**(7), (2015), 217–236.

[2] Video electronics standards association, vesa bios extension – core functions standard. version 3.0, 1998.

[3] Mileff, P. and Dudra, J.: Advanced 2d rasterization on modern cpus. *Applied Information Science, Engineering and Technology: Selected Topics from the Field of Production Information Engineering and IT for Manufacturing: Theory and Practice*, pp. 63–79, URL https://doi.org/10.1007/978-3-319-01919-2_5.