
Production Systems and Information Engineering
Volume 13 (1), pp. 112–125.
https://doi.org/10.32968/psaie.2025.1.9

A HYBRID RENDERING PIPELINE COMBINING CPU RASTERIZATION AND GPU UPSCALING

PÉTER MILEFF
University of Miskolc
Hungary Institute of Information Technology
peter.mileff@uni-miskolc.hu

Abstract: In this paper, we present a hybrid rendering approach that combines CPU-based software rendering with GPU-accelerated image upscaling. The method allows developers to perform computationally heavy pixel-level operations at a lower resolution on the CPU and then efficiently upscale the result to the display resolution using hardware acceleration. This design significantly reduces CPU workload while maintaining visual fidelity, offering a practical solution for educational renderers, retro-style graphics, or systems without full GPU rendering pipelines. Experimental results show that this approach provides a balanced trade-off between performance and image quality, making it suitable for lightweight rendering engines and research in computer graphics optimization.

Keywords: *software rendering, GPU upscale, optimization*

1. Introduction

Software rendering remains a relevant topic in computer graphics, both as an educational tool and as a lightweight solution for systems with limited GPU capabilities. Traditional software renderers perform all pixel operations directly on the CPU, which can lead to significant performance bottlenecks when targeting high resolutions. Modern GPUs, however, are optimized for large-scale image processing tasks such as scaling, blending, and texture mapping.

This paper introduces a hybrid rendering technique that leverages both CPU and GPU resources efficiently. The proposed approach performs the actual rendering logic—such as rasterization, shading, or pixel compositing—on the CPU at a reduced resolution. Once the low-resolution image is generated, it is transferred to the GPU as a texture and then scaled up to the final display resolution. The scaling

process is entirely hardware-accelerated through the Simple DirectMedia Layer (SDL2) library, ensuring minimal computational overhead.

This architecture enables developers to focus on algorithmic aspects of rendering without being constrained by GPU programming or high-resolution performance costs. Additionally, it opens possibilities for integrating modern effects, post-processing, and real-time visualization in educational environments or low-power devices.

The following sections describe the system design, implementation details, and performance evaluation of this hybrid renderer, emphasizing the benefits and potential applications of CPU-to-GPU scaling in software rendering.

2. Literature Review

Software rendering has a long and rich history in computer graphics, and hybrid CPU–GPU rendering approaches have increasingly become of interest as heterogeneous hardware becomes more available. In what follows, we review key contributions in several relevant research areas: (1) software-based rendering pipelines, (2) hybrid CPU–GPU rendering architectures, and (3) lightweight/educational rendering frameworks and upscaling techniques. We then identify how the present work builds on and differs from these.

2.1. Software-based Rendering Pipelines

Early work on fully CPU-based rendering laid the foundation for understanding pixel-level algorithms, rasterization, shading and memory access optimization. For example, a case study titled *A Case Study of a Hybrid Parallel 3D Surface Rendering Graphics Architecture* (Holten-Lund et al., 1997) described a hybrid hardware/software architecture but emphasized software rasterisation engines capable of 300,000 triangles/second on a Pentium PC.

More recently, the paper *A High‑Performance Software Graphics Pipeline Architecture for the GPU* (Kenzel et al., 2018) explored the implementation of a software graphics pipeline on the GPU, investigating how to map software-rasterisation concepts onto massively parallel architectures.

These works highlight the performance challenges and algorithmic complexity of software rendering—particularly the high cost per pixel when operating at full display resolution.

2.2. Software-based Rendering Pipelines

With the advent of heterogeneous computing (CPUs + GPUs) researchers began exploring how to partition rendering workloads across devices. One example: *Hybrid CPU‑GPU Unstructured Meshes Parallel Volume Rendering on PC Clusters* (Juliachs et al., 2007) describes a hybrid object-space/image-space volume rendering method where the CPU handles flexible tasks (such as irregular mesh traversal) and the GPU performs repetitive compositing and sorting.

Another example: *Cloud‑Assisted Hybrid Rendering for Thin‑Client Games and VR Applications* (Tan et al., 2022) proposes a hybrid rendering architecture for thin-client devices where part of the workload is on the client (e.g., rasterization) and part on the cloud/GPU for ray-tracing or high-quality output.

Additionally, *RenderMan XPU: A Hybrid CPU+GPU Renderer for Interactive and Final‑frame Rendering* (Christensen et al., 2025) presents a production renderer (Pixar's RenderMan XPU) that shares code between CPUs and GPUs, achieving significant speedups by hybridizing platforms.

These studies show that hybrid rendering can exploit the relative strengths of CPU (flexibility, branching, complex logic) and GPU (massive parallelism, throughput) but also highlight challenges: work partitioning, memory bandwidth, data transfer, load balancing.

2.3. Lightweight / Educational Rendering & Upscaling Techniques

While many hybrid rendering papers focus on high-end production or cluster systems, there is also research oriented towards more modest hardware or educational frameworks. For example, *Piko: A Design Framework for Programmable Graphics Pipelines* (Patney et al., 2014) explores a compiler/framework for building pipelines that can target both multicore CPUs and GPUs, enabling exploration of trade-offs in scheduling and architecture.

2.4. How this Work Relates and Contributes

The present paper proposes a hybrid rendering pipeline specifically targeted at software rendering at low resolution on the CPU, followed by GPU-accelerated scaling/presentation (via a texture blit) rather than full GPU rasterization. This niche—CPU renders low-res buffer → GPU handles scaling/listing/presentation—is less commonly addressed in research literature which typically handles either full CPU, full GPU, or more complex hybrid partitioning of major rendering tasks.

In light of the literature:

- Unlike full CPU software renderers (Section 1), our method deliberately uses a reduced resolution buffer, thereby reducing CPU workload.
- Unlike high-end hybrid renderers (Section 2), our method does not attempt to partition major rendering tasks between CPU and GPU at triangle or ray-level; instead it uses a simple, coarse division: CPU for the logic, GPU for the scaling blit/presentation.
- Unlike frameworks like Piko (Section 3) which aim at flexible pipeline generation, our focus is on a minimalist real-time approach using existing APIs (e.g., Simple DirectMedia Layer (SDL)) and leveraging GPU hardware scale abilities rather than custom shader complexity.

3. GPU Upscaling in Real-Time Rendering

In modern computer graphics and game development, GPU upscaling refers to the process of rendering an image at a lower internal resolution and then upscaling it to the display resolution using specialized algorithms executed on the graphics processing unit (GPU). This technique is widely adopted to balance visual quality and computational performance, allowing games to achieve higher frame rates while maintaining visually sharp output on high-resolution displays.

3.1. Motivation and Use Cases

The primary motivation for GPU upscaling is to reduce the rendering workload on the CPU and GPU by processing fewer pixels. In complex real-time scenes, rendering every frame at full native resolution (e.g., 4K or 8K) can be computationally expensive. By rendering at a lower resolution (e.g., 1080p) and upscaling the image to 4K, developers can significantly improve performance without an equivalent loss in perceived image quality.

GPU upscaling is commonly used in:

- Video games, to achieve stable frame rates on mid-range hardware.
- Real-time ray tracing, where upscaling reduces the cost of expensive light computations.
- Cloud gaming and streaming, to optimize bandwidth while preserving detail.

3.2. Modern GPU Upscaling Techniques

While early GPU-based upscaling relied primarily on simple texture filtering methods such as bilinear or bicubic interpolation, the 2010s marked the beginning of a new era characterized by advanced spatial and, later, AI-assisted techniques. Starting in this decade, GPU manufacturers began integrating specialized upscaling functions directly into their rendering pipelines to improve image quality without proportionally increasing computational cost.

One of the first significant steps in this evolution was AMD's Virtual Super Resolution (VSR), introduced in 2014. VSR allowed GPUs to render graphics internally at higher resolutions and then downscale them to match the display's native output, producing sharper and more detailed visuals. In the same year, NVIDIA released Dynamic Super Resolution (DSR), a similar approach implemented at the driver level. DSR utilized high-quality Gaussian filters to achieve smoother downsampling, effectively simulating higher-resolution rendering even on standard displays.

While both VSR and DSR provided perceptual improvements through spatial filtering, they remained purely algorithmic and lacked any form of temporal or contextual awareness. A major breakthrough occurred in 2018 with the introduction of NVIDIA's Deep Learning Super Sampling (**DLSS**), which marked the advent of AI-driven GPU upscaling. DLSS employs convolutional neural

networks trained on high-resolution ground-truth images to reconstruct missing detail from lower-resolution input frames in real time. This technique allowed games and visualization systems to render at significantly reduced internal resolutions while maintaining — or even exceeding — the visual fidelity of native rendering.

The following years saw the emergence of several competing and complementary technologies. AMD's FidelityFX Super Resolution (**FSR**), released in 2021, offered an open, hardware-agnostic alternative that used sophisticated edge reconstruction and spatial upscaling algorithms without relying on dedicated AI cores. In 2022, Intel introduced Xe Super Sampling (**XeSS**), combining machine learning with cross-vendor compatibility, further expanding the accessibility of intelligent upscaling methods.

Collectively, these innovations transformed GPU upscaling from a basic resampling operation into a sophisticated, real-time reconstruction process that leverages both traditional signal processing and modern artificial intelligence. Today, upscaling has become an integral part of GPU rendering pipelines, balancing visual fidelity and computational efficiency — a critical capability for modern applications in gaming, simulation, and real-time graphics rendering.

Table 1
Comparison of well known techniques

Method	Type	Performance Impact	Image Quality	Temporal Stability	Typical use cases
Nearest-neighbor	Spatial	Very low (fastest)	Poor (pixelated)	High (static)	Retro / pixel-art rendering, low-power systems
Bilinear interpolation	Spatial	Low	Fair (slightly blurry)	High	Simple real-time upscaling in 2D/3D applications
Bicubic interpolation	Spatial	Moderate	Good (smooth gradients)	High	Image viewers, offline rendering
Lanczos resampling	Spatial	Moderate to high	Very good (sharp edges)	High	Video upscaling, offline graphics processing
TAAU (Temporal Anti-Aliasing Upscale)	Temporal	Moderate	Very good	Medium (can cause ghosting)	Modern games with temporal anti-aliasing
AMD FSR 2/3	Temporal + Spatial	Low to moderate	Excellent	High	Real-time rendering, gaming (vendor-neutral)
NVIDIA DLSS 2/3	Deep Learning (AI-based)	Moderate (requires tensor cores)	Excellent (AI reconstruction)	Very high	High-end gaming, ray tracing workloads
Intel XeSS	Deep Learning (AI-based)	Moderate	Excellent	High	Gaming and hybrid rendering environments

4. Software rendering

The name of software rasterization originates from the imaging process when the entire image rasterization process, the whole pipeline is carried out by the CPU instead of a target hardware (e.g. 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 main memory. The main memory also holds the shape of 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 sends the completed image to the video controller.

Framebuffer is an area in memory which is being streamed by 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. Firstly, 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 the OpenGL's platform-independent (e.g. GLDrawPixels or Texture) [6] or the DirectX (e.g. DirectDraw surface or Texture) solutions.

4.1. Benefits of software rendering

Despite software rendering is applied rarely in practice, it has many advantages over the GPU-based technology. The first and most emphasized point is, there is less need to worry about compatibility issues, because the pipeline stages are processed entirely by the CPU. In contrast the GPU, CPUs structure changes less rapidly, thus the need of adapting to any special hardware/instruction set (e.g. MMX, SSE, AVX, etc.) is much smaller. In addition, these architectures are open and well-documented instead of 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 to several platforms causes fewer problems. Today's two leading GPU manufacturer publish their drivers only in closed form, where there are significant problems in performance at the Linux platforms. The 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.

4.2. 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 frequently modify these segmented data, this can cause significant loss of speed.

The second major problem, which originates also from the bus (PCIe) bandwidth, is the movement of large amounts of datasets between the main and the video memory. During one second the screen should be redrawn at least 50-60 times, which results in a significant amount of dataflow between the two memories.

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

5. System Design and Implementation

The proposed rendering system is based on a hybrid architecture that separates the pixel computation and display rendering processes between the CPU and the GPU. The CPU performs all low-level rendering operations into a software buffer, while the GPU is responsible for the final upscaling and presentation of the image on screen. This design efficiently utilizes hardware resources and minimizes data transfer overhead.

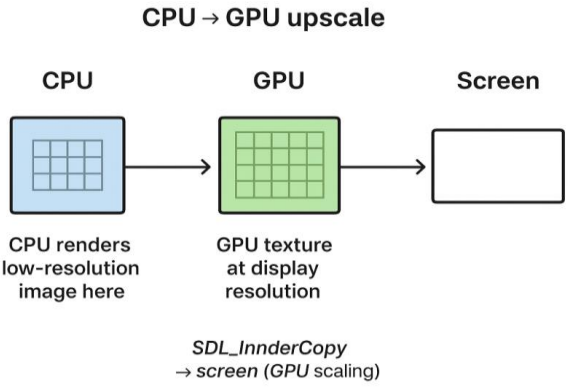

Figure 1. Upscale process pipeline

5.1. The method of drawing pixels

GPU is designed to work efficiently with vertex and texture data. It is therefore worth choosing a solution that uses these processing processes 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

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 that 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:

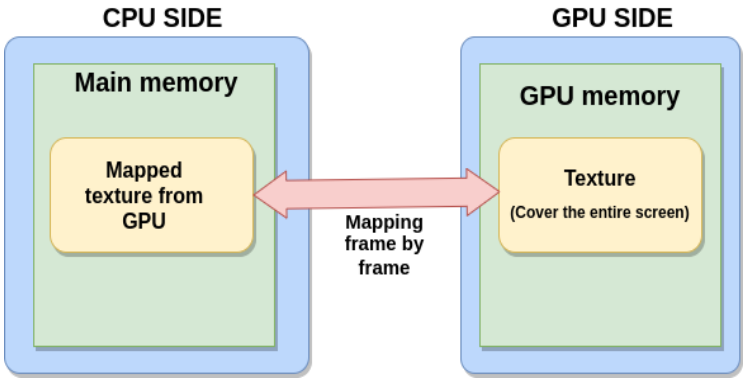

Figure 2. The process of software rendering on today GPU's

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.

5.2. CPU Rendering Buffer

The rendering pipeline begins with a fixed-size pixel buffer allocated in main memory, typically defined as a one-dimensional array of 32-bit color values (RGBA format). An efficient approach is to handle the components of the pixels together. In the case of RGBA, each component requires 1 byte, so 4 bytes = 32 bits is required 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.

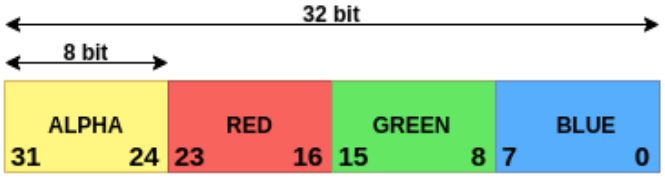

Figure 3. Pixel component in an 32 bit integer

So there are two buffers on the CPU side in the main memory. One *int32* type array which holds the screen pixel data and one for *z* buffer if needed. All rendering operations, such as line drawing, shading, or rasterization, are performed directly on this buffer using standard CPU instructions. When the CPU is finished for drawing, the *int32* buffer is copied into the mapped GPU texture data.

5.3. The virtual canvas

The purpose of this study is to investigate the efficiency of a hybrid rendering process that combines the distinct roles of the CPU and GPU to optimize image generation. The core concept is that all drawing operations are performed on the CPU side, but only within a lower-resolution framebuffer, while the final image upscaling and display are handled by the GPU. The main advantage of this approach is that the computational load on the CPU can be significantly reduced, since fewer pixels are processed, while the GPU—optimized for parallel operations—can upscale the image to the display's native resolution in real time without noticeable visual degradation.

To achieve this, a software-defined framebuffer is created, which can be dynamically configured to any desired lower resolution. This memory area serves as the rendering target for all CPU-side drawing operations, including rasterization, coloring, and shading. Once the image is completed, it is transferred to the GPU via data copying or texture updating (e.g., using SDL_UpdateTexture), where hardware-based upscaling takes place, leveraging the efficiency of the GPU's graphics pipeline. The entire procedure operates in real time as a continuously repeating rendering loop, in which each frame generated by the CPU is upscaled to full display resolution by the GPU.

This approach is particularly suitable for improving the performance of software renderers aiming for real-time visualization under limited CPU resources. The hybrid model allows the CPU to retain full control over the rendering process—such as implementing custom rasterization algorithms—while using the GPU solely for upscaling and presentation. This minimizes synchronization overhead and implementation complexity, providing a simple yet effective bridge between software and hardware rendering paradigms.

6. Performance Evaluation

To evaluate the performance of the developed hybrid rendering system, a prototype

application was implemented in C++, utilizing the SDL library to manage CPU–GPU cooperation. The primary goal of the measurement was to assess rendering performance, specifically the variation in frame rate (frames per second, FPS) under different drawing complexities and resolutions.

During the experiments, rendering was performed on the CPU side into a software framebuffer with an adjustable resolution. The resulting image was then copied each frame into a GPU texture, where upscaling and final on-screen presentation were handled. The purpose of this evaluation was to determine the extent to which lower CPU-side rendering resolution contributes to increased rendering speed, and how this performance gain changes depending on the number of rendered objects.

6.1. Measurement Methodology

Rendering performance was characterized in all cases by the average FPS (frames per second) value, which was computed by the program using timing cycles based on the SDL_GetTicks() function. Each measurement was conducted over a 60-second runtime, with the initial few seconds excluded to eliminate the warm-up phase and ensure stable results. The experiments were performed at several different rendering resolutions (640 × 480, 800 × 600, and 1024 × 768), while the output texture always matched the display's native resolution of 1920 × 1080. CPU and GPU utilization were monitored separately using the *htop* and *nvidia-smi* tools to provide an accurate overview of processing load distribution between the two components.

6.2. Test Scenarios and results

The measurements were conducted through four distinct test scenarios, each designed to evaluate the system's performance under increasing rendering complexity. In these tests, a series of uniformly sized images (textures) were rendered on the screen, each with a resolution of 256 × 256 pixels. The number of these rendered objects was gradually increased across the tests to observe how the growing CPU workload affects the overall frame rate and the benefits of GPU-based upscaling.

The four test configurations were as follows:

- **Test 1:** Rendering 5 objects of 256×256 pixels
- **Test 2:** Rendering 50 objects of 256×256 pixels
- **Test 3:** Rendering 100 objects of 256×256 pixels
- **Test 4:** Rendering 200 objects of 256×256 pixels

This experimental setup allows for systematic observation of how the performance gain from reduced CPU-side rendering resolution changes as the number of rendered elements increases, helping to identify the point where the CPU becomes the primary bottleneck in the hybrid rendering pipeline. Table 2 shows the

performance results:

Table 2
Test cases with results (FPS)

Case (256x256 texture)	No upscalse (original)	Upscale from 1024x768	Upscale from 800x600	Upscale from 640x480
5 texture	325	581	765	975
50 texture	180	240	280	371
100 texture	120	150	165	222
200 texture	72	81	92	128

6.3. Observations and Expected Trends

Based on the initial measurements, it can be concluded that lower-resolution CPU-side rendering provides a significant performance gain, especially when the number of rendered objects is low or moderate. However, as the scene complexity increases — for instance, when rendering 100 or 200 objects — the CPU-side processing becomes the primary bottleneck, since the rendering process runs on a single thread. As a result, the performance benefits gained from GPU upscaling gradually decrease, and the difference between various rendering resolutions becomes less pronounced.
This phenomenon clearly illustrates that although the GPU efficiently handles the upscaling process, the overall rendering speed is ultimately determined by the CPU's drawing capacity. Further optimization could be achieved by parallelizing the CPU rendering across multiple threads or by introducing partial GPU acceleration into the drawing phase.

6.4. Improving Image Quality on the GPU Side

The quality of the upscaled image depends on the interpolation technique used during GPU scaling. While simple methods such as nearest-neighbor or bilinear filtering are fast, they often produce visible pixelation or blur. Higher-quality results can be achieved using bicubic or Lanczos filtering, or through custom GPU shaders that perform edge-aware or adaptive interpolation. Modern solutions like NVIDIA DLSS and AMD FSR further enhance visual fidelity by applying AI-based reconstruction, which restores fine details from low-resolution inputs with minimal performance loss.

7. Improving Image Quality on the GPU Side

While the main focus of this work is on improving rendering performance through CPU–GPU cooperation, visual quality remains an important factor in evaluating the overall effectiveness of the proposed method. The quality of the upscaled image depends largely on the interpolation and filtering techniques applied by the GPU during texture magnification.

In the simplest implementation, **nearest-neighbor** filtering is used, which directly maps pixels from the low-resolution buffer to the target resolution. Although this approach is computationally inexpensive and fast, it tends to produce visible pixelation and blocky edges. A commonly preferred alternative is **bilinear interpolation**, which blends adjacent pixel values to produce smoother transitions. However, bilinear filtering may lead to a slight blurring of fine details.

For higher visual fidelity, the GPU can employ more advanced sampling methods such as bicubic or **Lanczos filtering**, which consider a wider pixel neighborhood to reconstruct image details more accurately. These techniques can be implemented efficiently in modern graphics APIs using fragment shaders. By writing custom shaders, developers can also create **edge-aware** or **adaptive interpolation** methods, which preserve contours and textures more effectively than standard linear approaches.

Furthermore, recent developments in GPU technology have introduced **AI-assisted upscaling** solutions, such as NVIDIA's DLSS (Deep Learning Super Sampling) or AMD's FSR (FidelityFX Super Resolution). These techniques use trained neural networks to reconstruct high-frequency image details from low-resolution inputs, achieving visual quality close to native rendering with significantly lower computational cost. Although these systems require dedicated hardware or proprietary software frameworks, they demonstrate the potential of combining machine learning and real-time graphics for optimal performance–quality balance.

While the current implementation focuses on performance and architectural efficiency, even a moderate improvement in the upscaling stage could lead to a more visually appealing and professional output without increasing CPU workload

8. Conclusion

This work presented a hybrid CPU–GPU rendering approach where the CPU performs low-resolution software rendering, and the GPU upscales the image to the display resolution. The results show that this method can notably improve performance when the number of rendered objects is moderate. As rendering complexity increases, however, the CPU becomes the main bottleneck, limiting the benefits of GPU upscaling. Overall, the approach offers a simple and efficient way to accelerate software rendering without sacrificing visual quality. Future work may include multi-threaded CPU rendering or the integration of advanced, AI-based upscaling methods to further enhance performance and image fidelity.

References

[1] Abrash, M. (1997). *Michael Abrash's Graphics Programming Black Book*. Coriolis Group Books.

[2] Carmack, J. (1996). Software Rendering in Quake. *Game Developers Conference Proceedings*.

[3] Chopp, D. (2018). *TinyRenderer Project*. https://github.com/ssloy/tinyrenderer

[4] Mesa3D Project (2022). LLVMpipe Software Rasterizer. https://docs.mesa3d. org/drivers/llvmpipe.html

[5] *Scratchapixel 2.0: Computer Graphics Tutorials*. 2023. https://scratchapixel.com

[6] Wald, I., Parker, S., Knoll, A., Martin, W., Benthin, C. (2014). Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)*, vol. 33, no. 4. pp. 1-8 https://doi.org/10.1145/2601097.2601199

[7] Pharr, M., Humphreys, G. (2023). *Physically Based Rendering: From Theory to Implementation*. 4th edition, Morgan Kaufmann.

[8] Owens, J. D. et al. (2008). GPU Computing. *Proceedings of the IEEE*, Vol. 96, No. 5, 879–899. https://doi.org/10.1109/JPROC.2008.917757

[9] Navarro, C., Suarez, J., Lopez, R., Aguilera, E. (2021). Heterogeneous CPU–GPU Rendering Frameworks. *Computer Graphics Forum*, Vol. 40, No. 2.

[10] Lantinga, S. (2024). Simple DirectMedia Layer 2 (SDL2) Documentation. https://wiki.libsdl.org/SDL2/FrontPage

[11] Holten-Lund, H., Bangsgaard, H. (1997). A Case Study of a Hybrid Parallel 3D Surface Rendering Graphics Architecture. *Proceedings of Graphics Interface '97*. pp. 149–154

[12] Kenzel, M., Meister, D., Steinberger, M. et al. (2018). A High-Performance Software Graphics Pipeline Architecture for the GPU." *Proceedings of High-Performance Graphics 2018.* pp. 1–15, https://doi.org/10.1145/3197517.3201374

[13] Juliachs, M., Coma, I., Chover, M. (2007). Hybrid CPU-GPU Unstructured Meshes Parallel Volume Rendering on PC Clusters. *Eurographics Symposium on Parallel Graphics and Visualization*, pp. 33–40. https://doi.org/10.2312/EGPGV/EGPGV07/085-092

[14] Tan, Z., Wang, J., Li, P. (2022). Cloud-Assisted Hybrid Rendering for Thin-Client Games and VR Applications. *Bohrium Research Reports on Cloud Graphics*.

[15] Christensen, P. H. et al. (2025). RenderMan XPU: A Hybrid CPU+GPU Renderer for Interactive and Final-frame Rendering. *ACM SIGGRAPH Talks 2025*. Vol 15, no 8, pp 1-17.

[16] Patney, A., Tzeng, S., Fatahalian, K., Owens, J. (2014). *Piko: A Design Framework for Programmable Graphics Pipelines*. arXiv preprint arXiv:1404.6293.

[17] Peng, Z., Du, J., Qiao, Y. (2023). Design of GPU Network-on-Chip for Real-Time Video Super-Resolution Reconstruction. *Micromachines*, Vol. 14, No. 5. https://doi.org/10.3390/mi14051055

[18] Zhong, S., Yang, Y., Zhu, Q. et al. (2023). *FuseSR: Super Resolution for Real-time Rendering through Efficient Multi-resolution Fusion*. arXiv preprint arXiv:2310.09726, 2023.

[19] Lee, H., Kim, J., Park, S. (2018). GPU-based Real-Time Super-Resolution System for High-Quality UHD Video Up-Conversion. *IEEE Trans. Consumer Electronics*. vol. 64, no. 4, pp. 519–528, 2018

[20] Rodrigues, A., Celes, W. (2018). A Hybrid CPU-GPU Scalable Strategy for Multi-resolution Rendering of Large Digital Elevation Models with Borders and Holes. *Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2018) - GRAPP* pp. 240–247, https://doi.org/10.5220/0006621902400247