Decoupling Video Upscaling from Rendering for Cloud Gaming

Deniz Ugur
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada

Ihah Amer
Advanced Micro Devices, Inc.
Markham, ON, Canada

Mohamed Hefeeda
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada

ABSTRACT

Many recent video games require powerful hardware to render them. To reduce such high hardware requirements, upscalers have been proposed in the literature and industry. Upscalers save computing resources by first rendering games at lower resolutions and frame rates and then upscaling them to improve players' quality of experience. Current upscalers, however, are tightly coupled with the rendering logic of video games, which requires updating the source code of each game for every upscaler. This increases the development cost and limits the use of upscalers. The tight coupling also stifles the deployment of upscalers in cloud gaming platforms to reduce the required computing resources. We propose decoupling upscalers from game renderers, which allows utilizing various upscalers with games without changing their source code. It also accelerates deploying upscalers in cloud gaming. Decoupling upscalers from renderers is, however, challenging because of the diversity of upscalers, their dependency on information at different rendering stages, and the strict timing requirements of video games. We present an efficient solution that addresses these challenges. We implement the proposed solution and demonstrate its effectiveness with two popular upscalers. We also develop a cloud gaming system in the emerging Media-over-QUIC (MoQ) protocol and implement the proposed approach with it. Our experiments show the potential savings in computing resources while meeting the strict timing constraints of video games.

CCS CONCEPTS

• **Information systems** → *Multimedia information systems.*

KEYWORDS

Super Resolution, Video Games, Cloud Gaming

ACM Reference Format:
Deniz Ugur, Ihah Amer, and Mohamed Hefeeda. 2025. Decoupling Video Upscaling from Rendering for Cloud Gaming. In *The 16th ACM Multimedia Systems Conference (MMSys '25), March 31–April 4, 2025, Stellenbosch, South Africa.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3712676.3714439

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MMSys '25, March 31–April 4, 2025, Stellenbosch, South Africa
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM
ACM ISBN 979-8-4007-1467-2/25/03...$15.00
https://doi.org/10.1145/3712676.3714439

1 INTRODUCTION

Recent video games strive to offer a rich and engaging experience to players. This requires video games to have high frame rates to support fast motions, complex lighting and shadowing effects to increase realism, and high resolutions to represent detailed textures and graphical elements. All of these features, however, require extensive computing resources. For example, the popular Star Wars Outlaws game supports up to 144 frames per second (fps) and 4K resolution, which requires a GPU with 10-24 GB of memory [23]. This kind of GPU costs hundreds of dollars and requires powerful/expensive gaming consoles or workstations, which may not be available to many players, limiting the potential reach of the game. To partially address this problem, *upscalers* have been proposed in the literature, e.g., RenderSR [11], ExtraSS [32], Mob-FGSR, [34], and Neural Supersampling [33], and developed by industry, including AMD FidelityFX Super Resolution (FSR) [7], NVIDIA Deep Learning Super Sampling (DLSS) [10], and Intel Xe Super Sampling (XeSS) [9]. Upscalers enable running video games with less computing resources. This is done by first rendering the game at lower resolutions and/or frame rates and then upscaling them across the spatial and/or temporal domains.

Upscalers, however, are quite different in their software designs and algorithms. For example, some upscalers use computational image processing methods, while others use deep learning models. Moreover, the computational methods and deep learning models vary across versions of the same upscaler. The myriad of upscalers, with multiple versions each, pose a significant challenge for game developers, as they need to customize the source code of their games for each upscaler and often versions of the same upscaler. This not only increases the development time and cost but also practically limits the possible number of upscalers that can be supported.

One of the critical problems behind increasing the development time/cost of utilizing upscalers and limiting their wide deployment is the *tight coupling* of the upscalers with game renderers. That is, the upscaler code currently must be integrated within the game code itself. Although common upscalers from the industry provide instructions and libraries to facilitate the integration process with various game development and rendering frameworks, the process has to be repeated for every single upscaler and often for the different versions of the same upscaler. In addition, the tight coupling of upscalers and game renderers complicates the deployment of upscalers in cloud gaming environments, which are gaining popularity but require extensive computing and bandwidth resources.

To address this problem, we propose decoupling upscalers from game renderers, which allows them to be developed separately with minimal dependence on each other. It also allows various upscalers to be easily integrated with video games without significant changes in the games' source codes, accelerating the wide adoption

of upscalers in stand-alone and cloud gaming environments. Realizing this decoupling, however, is challenging because of the diversity of upscalers, their dependency on information at different stages of the rendering process, and the strict timing requirements of video games. We address these challenges and present mechanisms to achieve this decoupling efficiently. In particular, the contributions of this paper are:

- We propose the idea of decoupling upscalers from game renderers, providing flexibility and cost-effectiveness.
- We analyze common upscalers in practice and identify the graphics resources and camera parameters required to enable upscalers and renderers to run independently.
- We design efficient synchronization schemes to support distributing upscalers and renderers on different GPUs and machines.
- We implement the proposed decoupled approach in an open-source game rendering engine to demonstrate its practicality.
- We implement an end-to-end cloud gaming system and show the potential resource saving achievable by upscalers. Our results also show that the decoupled upscaling approach does not increase latency and provides high-quality streams in real time.

2 BACKGROUND AND RELATED WORK

2.1 Background

Video Games. The logic, structure, and graphics of video games are written using frameworks such as Unity [29], Unreal [13], and O3DE [12], which are called rendering engines. Such frameworks offer high-level constructs for developing games. These constructs abstract and utilize APIs from common 3D graphics libraries such as Microsoft DirectX, OpenGL, and Vulkan.

Game Rendering. When a player issues a command, e.g., pressing a button, the game engine determines what needs to happen, e.g., firing a weapon. It then updates the game state, such as object positions, animations, and lighting. Most modern video games, e.g., Cyberpunk 2077, Call of Duty, and Battlefield, use *ray tracing* [24] to render realistic scenes. In ray tracing, light rays are cast into the scene, where each pixel corresponds to a ray traveling through the scene. Rays are then intersected with the scene objects to determine their appearance, which also depends on the material properties of objects, secondary reflections, and shadows. Rendering is *computationally expensive* and typically done on GPUs.

We provide a high-level illustration of rendering in Figure 1. The Initialization step pre-loads objects, textures, and other assets that will be used in rendering each frame. This is done at the start of the game and when significant changes occur, e.g., selecting a different game level. After initialization, multiple steps are performed for each frame, constituting the *rendering loop*. The Scene Setup step collects player actions and game decisions to create the scene, including objects, cameras, and lights. Culling techniques [21] determine which objects will be in the camera's view and need to be rendered. Then, the Geometry Processing step transforms the 3D models into a form that can be rendered on the GPU. This includes applying transformations like rotation, scaling, and translations. It also applies materials and textures to objects. The Lighting and Shading step determines how light interacts with various surfaces, accounting for multiple light sources, reflections, and subtle effects like ambient occlusion. Then, Shadow Mapping determines which

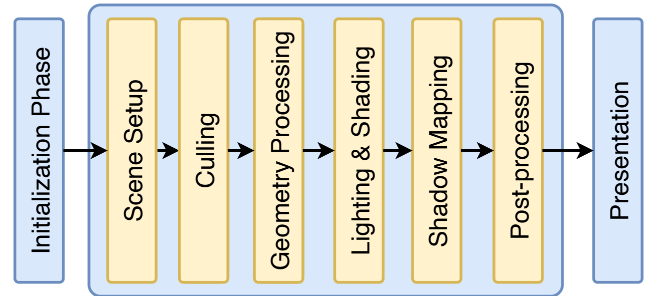


Figure 1: Main rendering steps in video games (simplified).

areas are shadowed from the perspective of each light source. Finally, post-processing effects, e.g., color correction and anti-aliasing, are applied, and the frame is presented on the display.

Spatial and Temporal Upscaling. To save computational resources, rendering can be done on a subset of pixels. Then, an *upscaling* method is used to estimate the remaining pixels. This is called spatial or resolution upscaling. Estimating pixels typically requires less computational resources than rendering them. To save even more resources, the game can be rendered at a lower frame rate. Upscaling can then be used to interpolate additional frames between the rendered ones. This is called temporal upscaling or frame generation. Example commercial upscalers include FSR [7] and DLSS [10]; both provide spatial and temporal upscaling.

Limitations of the Current Approach. In the current approach for utilizing upscalers in video games, developers need to tightly integrate and customize their code for each upscaler and even for different versions of the upscalers. As mentioned before, this process increases development costs and limits the number of upscalers that can be implemented. In addition, the tight integration of upscalers with renderers prevents upscalers from being utilized to reduce the required computational resources in cloud gaming. This is because current upscalers are designed for end users: they assume direct access to various renderers' data structures and display buffers.

Cloud Gaming. Players in cloud gaming run thin clients on virtually any device. The computationally expensive rendering process runs on cloud servers. The thin client captures the player's actions and sends them to a cloud server. The server runs the game logic, renders scenes, encodes frames using a video encoder, and streams them to the client. The client decodes and plays the received frames using any of the widely available video decoders. The entire processing pipeline must be completed within a strict deadline (around 30 msec, depending on the game), which poses a major challenge for any optimization method designed for cloud gaming.

2.2 Related Work

Upscalers. Upscalers are used to reduce the required hardware to play recent video games, which have rich graphics and fast motions [1]. There are several upscalers developed by the industry, including DLSS [10], FSR [7], and XeSS [9]. Each upscaler has its guidelines for integrating with video games. This requires developers to integrate each upscaler separately, increasing development time and cost.

In this paper, we do not present new upscalers. Instead, we propose a method to decouple renderers from upscalers.

Cloud Gaming Optimizations. Multiple works strive to address the challenges of cloud gaming, including reducing latency, bandwidth, and hardware requirements [14]. Latency is critical as even minor delays can severely impact the gaming experience, especially in fast-paced games where responsiveness is critical [5]. Bandwidth requirements of cloud gaming pose a challenge due to the high data transmission demands needed to stream high-quality graphics and maintain a smooth experience [14]. On the hardware side, several studies highlight the complexity of providing sufficient GPU resources to support diverse gaming workloads across different devices [15, 25, 27].

The proposed decoupling approach enables cloud gaming platforms to utilize upscalers to reduce computational resources.

Distributed Rendering. Some prior approaches focus on reducing the computational complexity of game renderers by modularizing and distributing them over multiple computing nodes [17, 19, 22, 28, 35]. For example, the authors of [17, 28, 35] propose distributing rendering tasks to edge servers, which can deliver lower latency to end users. The approach in [3] utilizes edge servers to support cloud gaming servers when they cannot meet the required quality. The work in [28] reduces bandwidth costs on the backbone network by sending lower-resolution game streams to edge servers, which are then upscaled using a game-specific model. Further, multiple works, e.g., [2, 19, 22], propose dividing the rendering workload between the gaming server and client.

This body of research is orthogonal to our work and can be combined with it to achieve further resource savings.

3 PROPOSED SOLUTION

3.1 Overview and Challenges

Overview. In contrast to the current approach of implementing upscalers within the game code, we propose decoupling the upscalers from renderers as illustrated in Figure 2. That is, the game renderer and upscaler run as independent processes, either on the same machine or different machines. Further, the upscaler and renderer can run on different GPUs within the same machine. This is enabled by identifying the critical graphics elements and structures required by upscalers and efficiently sharing them across distributed processes.

The current approach requires changing the source code of each game and customizing it for each upscaler version separately, limiting its scalability. Whereas the proposed approach does not require changing the source code of the game. It only makes minor modifications to the rendering framework and adds thin wrappers around various upscalers without changing their source codes. This enables upscalers to be easily and transparently used with many games.

Challenges. Realizing the proposed decoupling approach faces multiple challenges. First, upscalers are quite diverse in their internal designs and requirements. Some, e.g., DLSS and XeSS, are implemented using deep learning models, while others, e.g., FSR, are implemented using computational image processing methods. Computational upscalers tend to require accessing more graphical structures than deep learning ones, as they execute detailed mathematical equations on various aspects of the pixels and frames. Deep learning upscalers rely on the mapping power of neural networks.

In addition, most upscalers are designed assuming tight integration with the renderer, which allows them to access the needed structures and information at different stages of the rendering process, not necessarily at the end of the pipeline. For example, FSR may require accessing color textures pre- and post-translucency passes [8]. In §3.2, we analyze the most commonly used upscalers in practice, FSR and DLSS, and identify the general requirements to support various upscalers.

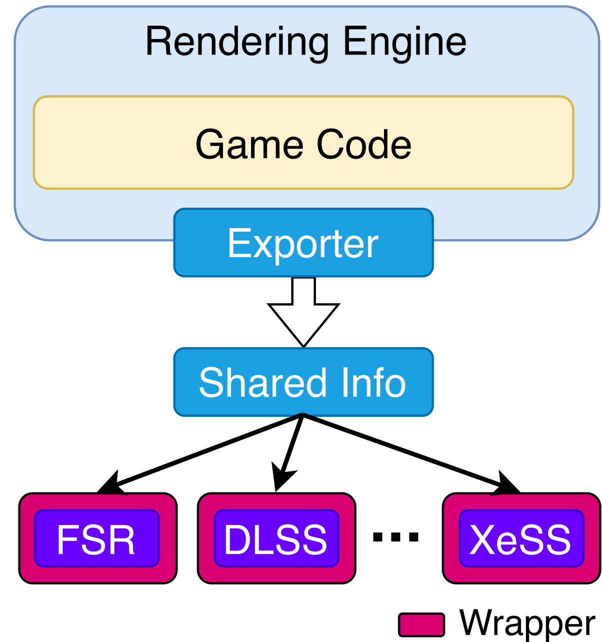


Figure 2: Overview of the proposed approach of decoupling upscalers and renderers.

The second main challenge stems from running the upscaler and renderer as independent processes instead of one as in the current approach. This is because they run at different speeds. Thus, they need synchronization to access the shared data structures. However, current video games can render up to 144 fps at 4K, generating large amounts of data and imposing high computational demands that must be performed by strict deadlines. Further, the upscaler and renderer processes can run on different GPUs or even different machines, making it harder to meet the deadlines over the network. In §3.4, we present an efficient solution using ring buffers and *fences*. Fences are synchronization primitives for sharing resources within GPUs as well as between GPUs and CPUs, and they support remote direct memory access (RDMA). Fences or their equivalent are available in common 3D graphics libraries, including DirectX, OpenGL, Vulkan, and Metal.

Finally, rendering engines and upscalers have large and complex code bases. Thus, changes to them must be clearly defined and minimized for the proposed approach to be practical. In §3.3, we present a simple wrapper around upscalers that does not change their code. We also identify and localize the required minor modifications to rendering engines.

Benefits of Decoupling Upscalers and Renderers. The proposed approach offers multiple advantages to game developers, cloud providers, and players, as summarized below.

Benefits for Game Developers: The proposed approach does not require integrating upscalers inside the game code, which reduces the development time. Upscalers can be used transparently and on different machines, increasing the game's availability to players.

Benefits for Cloud Gaming Providers: Decoupling upscalers from renderers allows utilizing upscalers in cloud gaming, which is currently not possible. Upscalers save substantial computing resources, as discussed in §2.1. Decoupling also allows running upscalers and renderers on different machines, i.e., offloading upscalers to other machines. Given the different computational requirements of renderers and upscalers, this enables the optimization of their performance separately by customizing the machines that run each. Further, upscalers are themselves quite diverse regarding the ideal platforms to run each. For example, DLSS runs better on NVIDIA GPUs because it relies on native NVIDIA APIs, while FSR may yield similar performance on various GPUs as it uses standard computational image processing functions. The proposed decoupling approach allows large cloud gaming providers to deploy multiple upscalers on different machines.

Finally, the proposed approach enables cloud providers to dynamically manage the computing resources allocated to different gaming sessions. This can be achieved by controlling the spatial and temporal scaling factors of the upscaler, which can easily be accessible since the upscaler is running as an independent process.

Benefits for Players: Players benefit from the ability to deploy various upscalers that best suit their hardware and the games they are playing, improving their gaming experience.

3.2 Identifying Upscalers Requirements

Simple upscaling methods, e.g., bilinear and bicubic interpolation, only require a color input to function. The results from these methods significantly lack quality compared to methods like FSR and DLSS. The underlying reason is that the upscaler lacks the context around each pixel and, therefore, must apply generic upscaling for each pixel. Modern upscaling methods require additional textures from the rendering engine to understand the scene in greater detail.

We analyzed the most popular upscalers currently deployed: AMD FSR [7], NVIDIA DLSS [10], and Intel XeSS [9]. We inspected their code integration guidelines and steps. We also experimented by implementing them with different games. Based on our analysis, we define three categories of resources for decoupling upscalers from renderers: (i) required resources, (ii) optional resources, and (iii) camera parameters.

The required resources are essential for the operation of upscalers. They determine which regions of the scene need to be upscaled and how. We identified three common resources needed by all upscalers: Depth Map, Motion Vectors, and Final Color. Optional resources assist in improving the quality produced by upscalers but are not necessary for the operation. For example, the UI Color and Alpha resource helps ensure that UI elements like health bars, menus, and text do not become blurry during the upscaling process. Optional resources vary across upscalers. Thus, we identified the optional resources for each upscaler. Then, we export the union of all optional resources to support a wide variety of upscalers. Based on our analysis, the most common optional resources are: Final Color Subrect, Hudless, UI Color and Alpha, Bidirectional

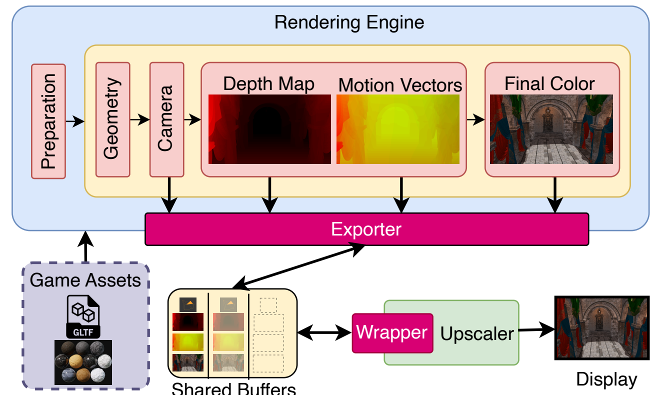


Figure 3: Collecting graphics resources and camera parameters at different rendering stages.

Distortion Field, T&C Mask, Reactive Mask, and Opaque Color. All of the required and optional resources are handled by the GPU.

The third category of resources for decoupling upscalers from renderers is the camera parameters. There are two types of parameters: intrinsic and extrinsic. Both are used in game rendering to accurately capture and display 3D scenes. Intrinsic parameters include: Focal Length (determines the zoom level of the camera), Principal Point (where the optical axis intersects the image plane), Skew Coefficient (describes the angle between the x and y pixel axes), and Lens Distortion Coefficients (corrects distortions caused by the camera lens). Extrinsic parameters define the position and orientation of the camera in the 3D world, and they include: Rotation Matrix (specifies the camera's orientation) and Translation Vector (specifies the camera's position). Camera parameters are essential for upscalers. For example, the Focal Length and Field of View parameters help the upscaler understand the depth and perspective and the scene. Similarly, the extrinsic position and orientation parameters ensure that the upscaled frames align with the game's world and have consistent motions.

In addition, camera parameters are essential for upscalers to utilize *sub-pixel jittering* in collecting more information and correctly upscaling frames. Sub-pixel jittering means slightly shifting the camera's position by a fraction of a pixel, capturing more detail in each frame. Sub-pixel jittering allows upscalers to sample different points within a pixel across multiple frames, giving it more information to upscale lower-resolution frames to higher quality. The intrinsic camera parameters, e.g., focal length, principal point, and lens distortion coefficients, affect how sub-pixel jittering is applied and how frames are upscaled. Similarly, the extrinsic parameters define the camera's position and orientation in the world, and sub-pixel jittering requires these parameters to ensure that the small movements are accurately captured and aligned.

Camera parameters are managed by the CPU, unlike the other two categories of resources, which are stored on the GPU.

3.3 Modifying Renderers and Upscalers

We propose a general approach for decoupling upscalers from renderers, illustrated in Figure 3. We identify all required resources and make them available to different upscalers. This approach allows

Algorithm 1 Renderer Process

```
1:  InitializeResources()
2:  while RendererRunning() do
3:  |  WaitOnFence(ResourceEvicted, bufferIndex)
4:  |  ProcessUserInputs()
5:  |  for all render module rm do
6:  |  |  GenerateCommandList(rm)
7:  |  |  if RequiredForUpscaling(rm) then
8:  |  |  |  CopyTextureRegion(texture, buffer)
9:  |  ExecuteCommandList()
10: |  SignalFence(ResourceReady, bufferIndex)
```

Algorithm 2 Wrapper Process

```
1:  InitializeResources()
2:  while WrapperRunning() do
3:  |  WaitOnFence(ResourceReady, bufferIndex)
4:  |  for all bufferTexture in buffer do
5:  |  |  texture = GetReference(bufferTexture)
6:  |  EvaluateUpscaler()
7:  |  ExecuteCommandList()
8:  |  PresentFrame()
9:  |  SignalFence(ResourceEvicted, bufferIndex)
```

switching upscalers on demand and easily utilizing future upscalers. As discussed above, the required and optional graphics resources are stored in the GPU memory, whereas the camera parameters are maintained in the main CPU memory. Thus, we create two shared buffers: one in the GPU and another in the CPU. In addition, our approach allows running multiple upscalers independent of the game renderer. A small wrapping code enables the upscalers to transparently access the needed graphics resources and camera parameters. The wrapping code also handles synchronization of the shared buffers with renderers, even if both run on different machines, as explained in §3.4.

Implementation in Rendering Engines. We first note that game rendering engines, such as Unity and Unreal Engine, provide all the required textures in their own way. Internally, the process needed to generate these textures is largely the same. Existing upscaler plugins for these engines modify the rendering pipeline to generate these textures or obtain a reference to them if they are already being used by the developer. These textures are fundamental building blocks of any rendering engine, and game engine developers ensure they are easily accessible.

A typical rendering pipeline consists of multiple render modules, each responsible for a specific aspect of the rendering process. For example, geometry processing transforms 3D scene data into screen-space coordinates, while lighting modules compute how light interacts with objects, and texturing modules apply image textures to surfaces. Afterward, post-processing modules may add visual effects such as bloom or depth of field.

Each render module generates *commands* that describe actions like drawing objects, applying shaders, or switching textures. These commands are collected into a master command list, which is then submitted to the GPU for execution. Since this process is performed for every frame and these steps may occasionally take longer than the duration of a single frame, the renderer uses a **swap chain** to avoid blocking the main thread. A swap chain is a sequence of render targets (buffers) that are used in rotation. While the GPU executes commands to render the current frame to one buffer (called the back buffer), the renderer processes the next frame in another buffer. When it is time to present the next frame, the back buffer and front buffer are swapped, displaying the newly rendered frame while the next one is prepared.

We summarize the proposed rendering engine modifications to support decoupling renderers and upscalers in Algorithm 1. Mainly, we added new commands to copy various resources at different rendering stages (lines 7–8) and synchronize shared memory buffers (lines 3 and 10). Everything else in the rendering engine is not changed. Conventional upscaling methods are tightly integrated with the game's rendering pipeline, allowing them to reference the texture directly rather than copying it. However, if a texture is only valid at a particular stage (i.e., it may be released before being presented), it is copied to an intermediary texture to preserve its state. Upscalers need this process to reference all necessary resources to upscale the frame. When the upscaler is decoupled, we must always preserve the texture's state, so we copy the textures whenever the renderer finishes using them.

Additionally, when a frame completes execution on the GPU, it signals the shared resource buffer to indicate that the frame data is ready for upscaling. This is shown in Algorithm 1 at line 10 for the renderer and in Algorithm 2 at line 3 for the wrapper. Both processes use fences to synchronize their state and determine when to act on a buffer. This is crucial because the wrapper acquires references to individual textures in the buffer slot as if they were its own, allowing the upscaler to use them as usual. The synchronization process is explained in §3.4.

Wrappers for Upscalers. To support decoupling upscalers from renderers, we have developed a wrapper that emulates the actual game environment. The wrapper supplies the necessary resources and textures to upscalers, just as the game would if the upscalers were tightly integrated. We summarize the wrapper design in Algorithm 2. It waits until resources are ready in the shared resource ring buffer. Upon receiving the signal, the wrapper continues to acquire references to textures within the buffer. Then wrapper calls the upscaler (line 6), which generates the necessary commands to upscale the frame on the GPU. These commands are appended to the main command list before sending it to GPU (line 7). Once the commands are executed, the upscaled frame is presented on the display (line 8). Finally, the wrapper evicts the buffer slot that was used by the frame by signaling the synchronization fence.

TransparentSR Library. We abstracted the creation and management of shared resources into an open-source library called TransparentSR [30]. This library simplifies the integration of detached upscaling into custom rendering engines. Our repository provides examples demonstrating how this library can be used effectively. For instance, the renderer can generate a low-resolution 3D scene and leverage the `TSROps::TransferToSharedBuffer` method to transfer the necessary resources to the upscaler. Additionally, the library offers a utility function, `tsr_map_udta`, which facilitates

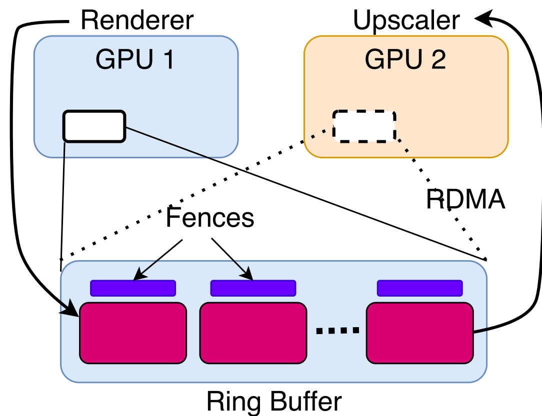


Figure 4: Synchronization of shared resources between renderer and upscaler deployed on different machines using fences and RDMA (remote direct memory access).

seamless management of user data (e.g., camera parameters) across frames, ensuring consistency and ease of use.

3.4 Synchronizing Shared Buffers

The proposed approach runs the game renderer and upscaler as independent processes, which provides flexibility and ease of utilizing various upscalers. However, both processes execute at different speeds and can be allocated on different machines. Thus, efficiently sharing the required graphics resources and camera parameters among them is critical for ensuring the quality and timeliness of rendered frames.

We propose a general approach that allows renderers and upscalers to run on: (i) the same GPU, (ii) different GPUs within the same machine, and (iii) GPUs running on distributed machines. Figure 4 provides a high-level illustration of the proposed approach. Specifically, we create a shared buffer on the renderer machine. Then, we virtually map this buffer to the upscaler machine to avoid duplicating the resources. The upscaler process can access the shared buffer using DMA (Direct Memory Access) if it resides on the same machine as the renderer. If the upscaler is on a different GPU or machine, it uses RDMA (Remote DMA). Current technologies, such as AMD Radeon Open Compute (ROCm), NVIDIA GPUDirect, and Intel oneAPI, support RDMA. Distributing renderers and upscalers to different machine is particularly beneficial for cloud gaming platforms, where the renderer may run on GPUs optimized for ray tracing and graphics operations and upscalers may be deployed on GPUs with tensor cores optimized for neural network operations and/or computational upsampling.

To enable this generality of deploying renderers and upscalers, we propose synchronizing access to the shared buffer through fences. Fences are synchronization primitives designed for inter-CPU and inter-GPU communications. They are supported in widely-used 3D graphics libraries, such as DirectX, OpenGL, Vulkan, and Metal. Fences also function in distributed environments and support RDMA. As described in our implementation (§4), we create fences and share their handlers between renderers and upscalers.

We design the shared buffer as a ring buffer with a fixed size of three slots. Having more slots wastes memory and could introduce undesirable delay between when the frames are rendered and when they are displayed (i.e., after the upscaler finishes processing them). Video gaming is an interactive application, and players expect to see the effect of their actions on the display within a strict deadline. On the other hand, having less than three slots in the ring buffer can lead to wasting computing resources, as one of the processes may have to wait to access the shared buffer: The renderer may have to wait for a free slot to write the output frame in, and the upscaler may wait until for a frame to be available to upscale it.

We note that in most cases, the upscaling process will run faster than the renderer. Thus, at least two slots in the shared buffer are needed so that while the renderer is writing frame data into one, the upscaler is reading from the other. The third slot in our design serves as a relief buffer. If upscaling takes longer than expected due to complex operations or OS-level scheduling, for example, the renderer can continue without stalling.

Implementation wise, we place fences at the end of the command list to signal when the resources are ready for upscaling. When the command list is executed on the GPU, the data will be copied to the next available slot in the shared ring buffer. Simultaneously, we copy the camera data to the shared buffer region located in the system memory. When the fence is signaled, our wrapper immediately upscales the frame using the frame data generated by the renderer. Once the upscaling is complete, the upscaled frame is presented, and the shared buffer region is signaled to release the frame data.

3.5 Overheads and Practical Considerations

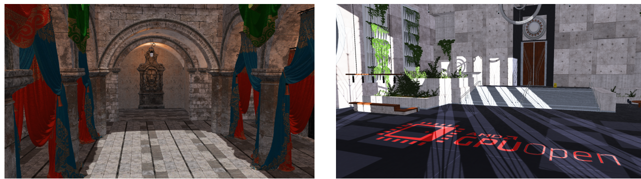
The proposed decoupling solution requires minor modifications to the rendering engine, and it imposes small overheads. The modifications can be easily added to existing rendering engines. Importantly, these modifications are applied once to the rendering engine rather than to each individual game. This significantly reduces the development effort and streamlines the integration process across multiple games.

Regarding overheads, the primary considerations are the memory and time associated with copying resources. In our experiments, copying resources to the shared ring buffer adds approximately 10.6 MB for the entire frame data, a small amount for modern systems. Also, it takes, on average, 0.16 ms for the exporter to copy the needed resources to the shared buffer. This is insignificant, considering the typical rendering time per frame is 33.3 ms (for a frame rate of 30 fps). Even for a high frame rate of 100 fps (i.e., rendering time of 10 ms), the added overhead time is 1.6%.

We believe these are small overheads considering the potential gains of easily and transparently integrating various upscalers with many video games.

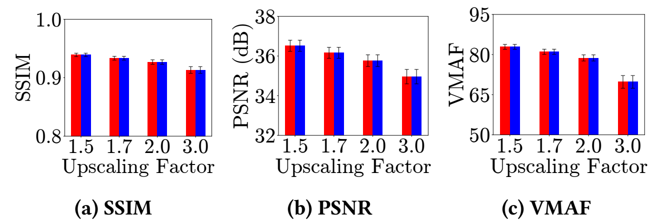
4 EVALUATION

First, we describe our proof-of-concept implementation in an open-source rendering platform. Then, we demonstrate the effectiveness of the proposed decoupled approach with two common upscalers (FSR and DLSS) with different upscaling configurations and game scenes. Then, we present our implementation in a cloud gaming



(a) Sponza (b) Brutalism

Figure 5: Game scenes used in our experiments. They are open-source samples with AMD FSR upscaler [8].



(a) SSIM (b) PSNR (c) VMAF

Figure 6: Comparison of the quality achieved by the decoupled (Red) and current (Blue) approaches. Results for DLSS with no frame generation processing Sponza game.

platform using Media-over-QUIC, and we conduct end-to-end analysis showing the feasibility of utilizing upscalers to save computing resources in cloud gaming.

4.1 Prototype Implementation

We implement our proposed solution in the open-source rendering engine Cauldron [6], which is used to showcase various rendering technologies, including FSR [8]. To modify this engine to support our decoupled approach, we had to understand how it prepares the rendering command list, updates the camera matrices, and allows us to access the graphics resources. We summarize our modifications in the following.

Modifying the Renderer. The renderer modifications primarily involved ensuring that textures and related upscaling data are consistently copied for each frame at the appropriate times in the rendering pipeline. To achieve this, we issue copy commands once the renderer finishes that particular piece of data. For GPU-bound data, we append the copy command to the command list when a resource is not modified further. The necessary operation is similar for CPU-bound data; we copy it when the renderer has completely finished with that frame. Before copying textures to their destination, we initialize the shared ring buffers.

To achieve the synchronization described in §3.4, we create two shared ring buffers, one on the GPU and one on the CPU. Additionally, we establish shared fences to facilitate synchronization between the GPU and CPU. This requires creating textures with the `D3D12_HEAP_FLAG_SHARED` flag and fences with the `D3D12_FENCE_FLAG_SHARED` flag. These flags allow the renderer or upscaler processes to obtain a handle to the resources using DirectX 12 device methods `OpenSharedHandleByName` and `OpenSharedHandle`. Each buffer stores the textures associated with a frame in a contiguous memory region, and both processes access the buffer using offsets. When both processes are active, they wait for an available slot in the ring buffer to operate on. The renderer requires an empty slot to write to, while the upscaler needs a slot with all the frame data written and ready for upscaling. Each process signals the other about the status of a particular slot, allowing coordination for further work. By keeping both processes synchronized and aware of the shared ring buffer's status, we effectively eliminate any race conditions that may occur in multi-process designs.

Wrappers for FSR and DLSS. Wrappers create a renderer-like environment for the upscalers. They supply the frame data in the same manner as a typical renderer would. From the upscalers' perspective, there is no difference in the input data compared to when they are tightly integrated with the actual renderer. When the wrapper is signaled that a slot on the shared ring buffer is available, it copies the textures into its own address space, along with the camera properties stored in the CPU-side shared ring buffer. This information is continuously supplied to the upscalers, allowing them to track the temporal data as needed. The wrapper itself does not alter the upscalers' implementation; it simply enables them to function by redirecting the necessary data from the actual renderer.

4.2 Experimental Setup

Upscalers and Game Scenes. We consider two popular upscalers: FSR and DLSS. FSR uses computational image processing methods and is open source. DLSS uses deep learning models and is partially open source. Both support various configurations, including frame generation and multiple spatial upscaling ratios.

We utilize two game scenes that are open source and diverse, as shown in Figure 5. We summarize all parameters used in the experiments in the following.

- **Upscalers:** None, FSR 3, DLSS 3
- **Modes:** Decoupled, Conventional
- **Frame Rate (FPS):** 20, 30
- **Frame Generation:** Enabled, Disabled
- **Upscaling Ratios:** 1.5, 1.7, 2.0, 3.0
- **Scenes:** Sponza, Brutalism

Performance Metrics. For each test run, we collect several performance and quality metrics, which are summarized below.

- **System Memory:** Total, Renderer, Upscaler
- **CPU Utilization:** Overall , Renderer, Upscaler
- **GPU**: Power Usage, Load Percent, Memory Usage
- **Processing Time:**[1] Render time, Upscale time
- **Quality:** SSIM, PSNR, VMAF

The system resource usage metrics are gathered from three separate tools. Metrics related to the overall utilization of individual resources are collected by the **HWiNFO** tool [20]. Process-specific CPU and memory utilization is collected by the Python library **psutil** [26]. In-process timings for different stages of rendering and upscaling are collected by the sample applications and reported as aggregated values at the end of each test run.

Quality comparisons are performed using SSIM (Structural Similarity Index) [31], PSNR (Peak Signal-to-Noise Ratio), and VMAF (Video Multi-Method Assessment Fusion) [36] metrics. SSIM and PSNR provide visual similarity to human perception and objective quality, respectively. VMAF leverages multiple metrics to assess video quality more comprehensively.

[1] These are collected inside the Renderer/Upscaler processes

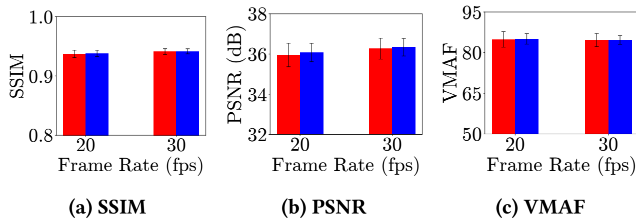


Figure 7: Comparison of the quality achieved by the decoupled (Red) and current (Blue) approaches. Results for FSR with frame generation processing Sponza game.

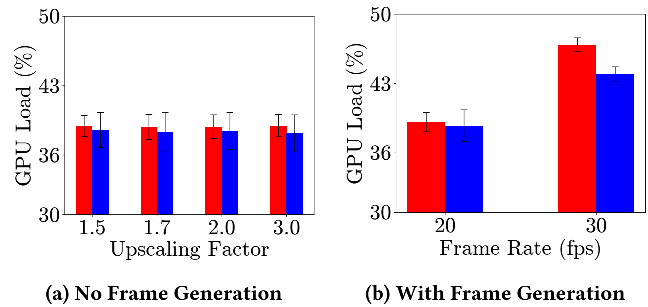


Figure 8: GPU utilization for the decoupled (Red) and current approaches (Blue). Results for DLSS processing Sponza.

Workstation Specifications. For all our tests, we use a workstation with i9-14900K CPU, 64 GB DDR5 Memory, and NVIDIA RTX 4060 GPU with 8 GB memory. This workstation provided sufficient performance to run all tests smoothly.

4.3 Evaluation on Individual Gaming Stations

We analyze the quality of upscaling produced by the proposed decoupled approach and compare it against the current, fully integrated, upscaling approach, which requires changing the source code of each game. We also analyze the overheads imposed by the proposed approach.

Quality Analysis. Accurately computing the SSIM [31], PSNR, and VMAF [36] quality metrics requires alignment of camera movement across different test runs; otherwise, the produced frames will have different pixel values and will not be comparable. While it is possible to keep the camera static, doing so would hinder the proper evaluation of the upscalers. To consider the dynamic aspects of game scenes, we move the camera in a torus curve to create a relatively complex camera movement. During the test run, we save the front buffer (presented frame) as a JPEG file without any compression at every animation step. Then, we combine the frames into a MOV container and save the video file without re-encoding the frame data.

To ensure our experiments are conducted as consistently as possible, we automate the entirety of our experiments and quality evaluation, including the camera movement of the renderer. We also fix the animation step time to achieve pixel-perfect frame alignment between test runs. This means that regardless of how long it takes to render a particular frame, we always capture the same scene setup. We run each test for 30 seconds, with a 30-second cool-down period, and we repeat the process three times. We run *every* permutation of the configuration parameters in §4.2.

We present representative *samples* of our results in Figure 6 and Figure 7; other results are similar. In these figures, we plot average results across all runs as well as the 95% confidence intervals as error bars. The results in Figure 6 for the DLSS upscaler show that the quality achieved by the proposed decoupled approach is indistinguishable from the quality of the current approach for all spatial upscaling ratios considered and all three quality metrics. Figure 7 demonstrates similar results for the FSR upscaler with frame generation enabled. That is, the proposed approach of decoupling upscalers from renderers does not negatively impact the upscaling quality, while it facilitates utilizing upscalers with different games. In addition to analyzing the quality using various objective metrics, we have also visually inspected the produced frames with and without our transparent upscaling method to ensure that there is no distinguishable quality difference.

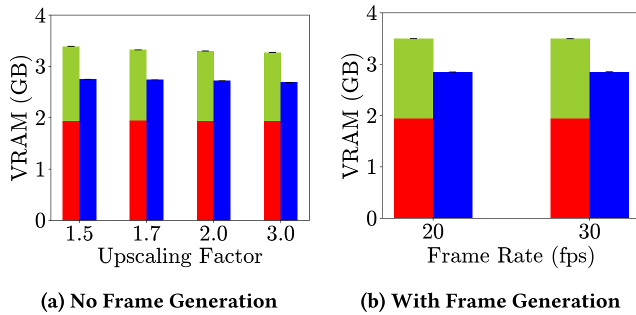
Overhead Analysis. To assess the overheads of the proposed approach, we terminate all applications running on the workstation and keep only our renderer and upscaler processes. We measure the various system utilization metrics mentioned in §4.2 using HWiNFO [20] and psutil [26] tools.

We present a sample of our results in Figure 8 and Figure 9, showing the GPU utilization and CPU memory usage, respectively. The proposed decoupling approach runs the renderer and upscaler as separate processes, which requires synchronization and copying of some data structures. This marginally increases the GPU utilization as shown in Figure 8. For example, with no frame generation (Figure 8a), the average increase in GPU utilization is less than 1%. Even when the upscaler generates additional frames (Figure 8b), which requires more processing, the increase in GPU utilization because of decoupling is up to 2.54%. We note similar increases in the CPU load have been observed in our experiments; figures are omitted due to space limitations.

In Figure 9, we analyze the memory requirements. We measure and plot the memory of the renderer and upscaler of the decoupled approach separately. The current approach, however, integrates the renderer and upscaler, and thus, its memory is not separated. As the figure shows, the decoupled approach requires less than 1 GB of additional memory. We also analyze the GPU memory requirements (figure not shown). Our results indicate that the decoupled approach requires up to 927 MB of additional GPU memory. The additional memory is because the decoupled approach creates two independent processes for the renderer and upscaler instead of one. Each process allocates its memory and resources. This is in addition to the shared buffers for exchanging required information between the renderer and upscaler. For practical deployments, multiple code optimizations can be performed to reduce the memory footprint of decoupling; our proof-of-concept prototype focused more on demonstrating the feasibility of decoupling.

4.4 Evaluation in Cloud Gaming

Implementation in Cloud Gaming using Media-over-QUIC. We implement the proposed decoupled upscaling approach in a cloud gaming platform to demonstrate its practicality. We are not aware of other works in the literature that utilize upscaling to reduce the extensive computing resources required by cloud gaming.



(a) No Frame Generation

(b) With Frame Generation

Figure 9: CPU Memory usage for the decoupled approach and current approach (Blue). Memory of the decoupled approach is divided into two parts: Renderer (Red) and Upscaler (Green). Results for FSR processing Sponza.

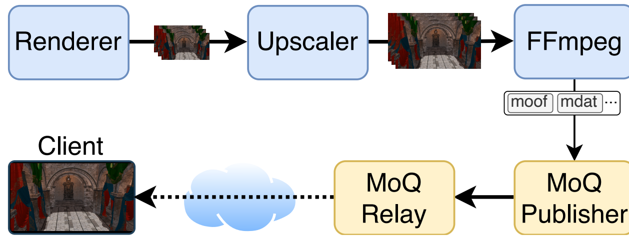


Figure 10: Overview of our cloud gaming testbed.

We consider the emerging Media-over-QUIC (MoQ) platform, which is being actively developed by the Internet Engineering Task Force (IETF) for efficient and low-latency streaming environments [16]. It is built on top of the widely deployed QUIC transport protocol, which is used in popular streaming services such as YouTube. Compared to WebRTC and other existing frameworks, MoQ offers several advantages, including better scalability and efficient congestion control. It also offers the ability to prioritize packets, such as player inputs or critical game frames, ensuring that essential data is delivered promptly, which is crucial for cloud gaming, where latency directly affects gameplay.

Our MoQ cloud gaming testbed is illustrated in Figure 10. The game renderer produces frames at low resolution and frame rate, which consume less computing resources. The upscaler, running as a separate process, improves the spatial resolution and generates new frames. The result of the upscaler is frames in RGBA32 pixel format. We pipe this raw pixel data into an FFmpeg module, which encodes and packages it into a fragmented MP4 container format. The encoded video stream is then piped through two other modules: MoQ Publisher and MoQ Relay. Both are based on the open-source implementation of MoQ [18]. MoQ Publisher publishes the stream using HTTP/3 (on top of QUIC). MoQ Relay takes the stream from the MoQ Publisher and serves it to clients, while performing functions such as adaptation, metadata processing, and caching. Clients connect to the MoQ Relay using a web browser that supports WebTransport, which provides low-latency and bidirectional communication over QUIC. The browser demultiplexes the stream, extracts video segments, and displays them.

Quality Analysis. We first subjectively validated the visual quality of the rendered frames on the receiver side after going through the entire pipeline in Figure 10. We observed no glitches, freezes, or visual distortions. This served as a sanity check for our implementation.

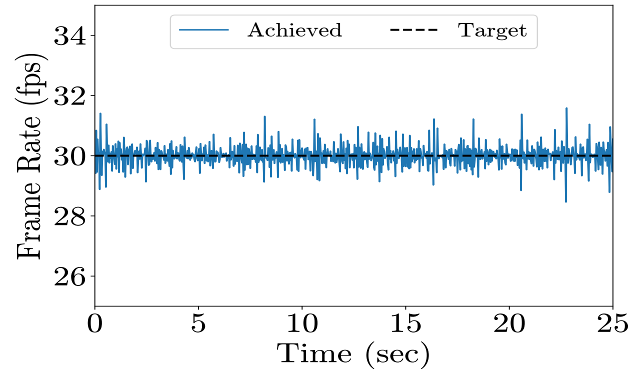


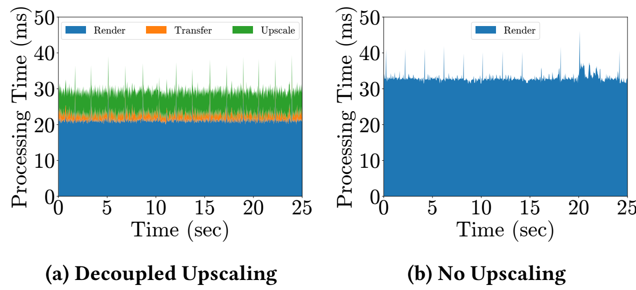
Figure 11: Performance of the FSR upscaler in the cloud gaming testbed implemented using the decoupled approach. Results show that our approach achieves the target frame rate.

To objectively analyze performance, we measure the frame rate achieved at the receiver. We set a target frame rate of 30 fps and configure the game engine to render at this rate with low resolution. Then, we upscale the resolution by 3.0x using FSR. We chose a 3.0x upscaling ratio because the FSR documentation recommends it as the maximum resource-saving option. There is no limit imposed on the upscaling ratio otherwise.

Following upscaling, frames are then encoded and transmitted over the MoQ testbed to the receiver. We record the arrival time of each frame at the receiver and compute the achieved frame rate over an extended period of time. The results, shown in Figure 11, confirm that our decoupled approach implemented in a cloud gaming testbed can achieve the target frame rate.

Timing Analysis. We analyze the time used by the renderer and upscaler for each frame. We record the timestamps as a frame passes through key stages of the pipeline. These stages include the start and end times for rendering and the start and end times for upscaling. Since our solution performs rendering and upscaling in separate processes, we also record the texture transfer time from the renderer to the upscaler. The results are shown in Figure 12a, where each stage is stacked on top of the previous stage to show the accumulated processing time for individual frames. The majority of the processing is done in the rendering stage, which accounts for more than 65% of the total processing time. Upscaling is much faster compared to rendering but still takes about 6-7 ms to perform. Transferring shared data between the renderer and upscaler takes about 1-2 ms.

To put these numbers in perspective, we plot, in Figure 12b, the time taken by the renderer for each frame without using any upscaling. In this case, the renderer produces the full resolution. As the figure indicates, the renderer takes, on average, slightly more time than the total time in Figure 12a, by about 2-3 ms per frame. That is, upscaling can actually reduce the total latency, which is especially important in cloud gaming where latency is critical. In addition, Figure 12b shows more occasional spikes in frame processing time



(a) Decoupled Upscaling (b) No Upscaling

Figure 12: Timing analysis from our cloud gaming testbed.

than the case with upscaling. This means the decoupled upscaling can provide a smoother performance.

Benefits of Upscaling in Reducing Computing Resources. We study the potential benefits of using upscalers to reduce computing resources. Specifically, we analyze two video games: Forza Horizon 5 and F1 23. Both are graphically rich car racing games. We first play each game without any upscaler, and we measure the fps achieved by the GPU on our workstation. We use CapFrameX [4] to record frame timings. Then, we run the game with different upscaler configurations. Then, we repeat the whole experiment for the other game.

A sample of our results is shown in Figure 13 for the F1 23 game using the FSR upscaler, where we plot the overall average frame rate achieved by the renderer across the whole experiment time. The figure shows that significant savings in computing resources could be realized by using upscalers. For example, without using any upscalers, our machine could render, on average, about 40 fps. Whereas, when we render at low resolution and then use FSR to upscale the resolution by 2.0x or 3.0x, the machine could almost double or triple the average number of rendered frames. This is achieved with almost no impact on the visual quality in the case of 2.0x and minimal impact in the case of 3.0x, based on our subjective analysis of the displayed frames during the experiments.

In Figure 14, we plot the achieved frame rate by the renderer across time with and without using upscalers. The results in the figure are for DLSS with frame generation enabled. We also plot the average across 1-sec periods as thicker lines. Two observations can be made on this figure. First, significant gains, in terms of the number of rendered frames per second, can be achieved by using upscalers. This aligns with the aggregate results in Figure 13, which were obtained with different upscaler and video games. Second, the achieved frame rate, with and without upscaler, fluctuates with time. This is because both the visual complexity of frames and the temporal motion across frames change during the game. This fluctuation indicates that the required computing resources in cloud gaming vary. Since our proposed approach decouples the renderer from upscaler, it offers a flexible and easy method to control the configuration parameters of the upscaler. This, in turn, enables cloud providers to dynamically manage these parameters across different gaming sessions to maximize the utilization of their computing resources and achieve the target quality of experience.

In summary, our cloud gaming testbed demonstrated that the proposed decoupled upscaling approach can successfully process and deliver high-quality gaming streams in real time, and it does not introduce additional latency. In addition, our experiments show using upscalers in cloud gaming can save computing resources and/or serve more gaming sessions with the same resources.

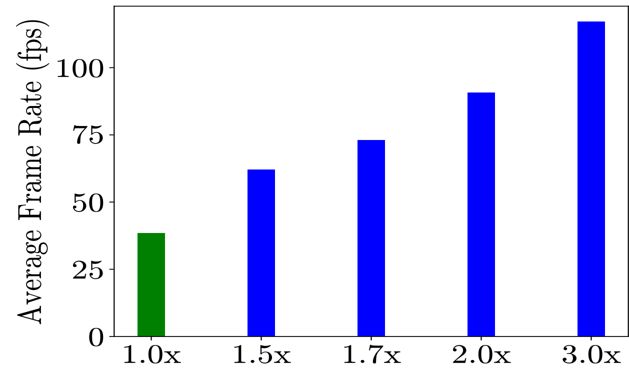


Figure 13: Performance gains from using upscalers. Results for FSR processing F1 game.

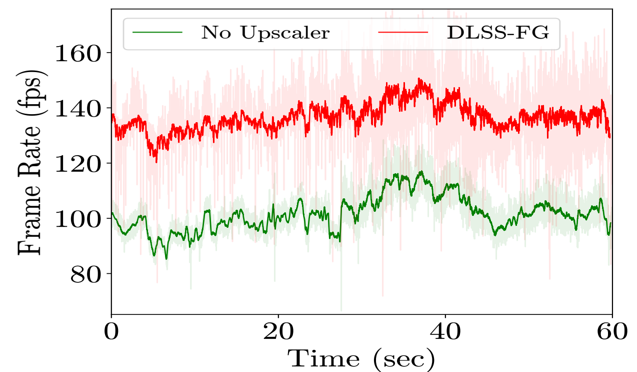


Figure 14: Performance gains from using upscalers. Results for DLSS with frame generation processing Forza game.

5 CONCLUSION

Current upscalers require tight integration with the source code of video games, increasing the development cost and limiting their wide deployment. We presented a method to decouple upscalers from game renderers, which facilitates transparently using various upscalers without the need to change the source code of games. It also enables the utilization of upscalers in cloud gaming to save computing resources, which has not been done before. We demonstrated the practicality of the proposed method by implementing it in an open-source rendering game engine and showed its effectiveness with two popular upscalers. In addition, we developed a cloud gaming testbed using the Media-over-QUIC (MoQ) framework. The results from this testbed demonstrated that the proposed decoupled upscaling approach can successfully process and deliver high-quality gaming streams in real time. Our experiments also showed that using upscalers in cloud gaming can save substantial computing resources and/or serve more gaming sessions with the same resources.

REFERENCES

[1] Saeed Anwar, Salman Khan, and Nick Barnes. 2021. A Deep Journey into Super-resolution. *Comput. Surveys* 53, 3 (5 2021), 1–34. https://doi.org/10.1145/3390462

[2] Keith Bugeja, Kurt Debattista, and Sandro Spina. 2019. An asynchronous method for cloud-based rendering. *The Visual Computer* 35 (12 2019), 1827–1840.

[3] James Bulman and Peter Garraghan. 2020. A cloud gaming framework for dynamic graphical rendering towards achieving distributed game engines. In *Proceedings of the 12th USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*. 1.

[4] CapFrameX. 2024. CapFrameX: Frametimes Capture and Analysis Tool. https://www.capframex.com/. Accessed 2024-08-23.

[5] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. 2012. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games (NetGames)*. 1–6.

[6] AMD Corporation. 2024. AMD FidelityFX™ Cauldron Framework. https://gpuopen.com/fidelityfx-cauldron-framework/.

[7] AMD Corporation. 2024. AMD FidelityFX™ Super Resolution (FSR). https://www.amd.com/en/products/graphics/technologies/fidelityfx/super-resolution.html.

[8] AMD Corporation. 2024. AMD FSR GitHub Repository. https://github.com/GPUOpen-LibrariesAndSDKs/FidelityFX-SDK. GitHub repository, Accessed 2024-08-23.

[9] Intel Corporation. 2024. Intel Xe Super Sampling (XeSS). https://www.intel.com/content/www/us/en/products/docs/discrete-gpus/arc/technology/xess.html.

[10] NVIDIA Corporation. 2024. NVIDIA Deep Learning Super Sampling (DLSS). https://www.nvidia.com/en-us/geforce/technologies/dlss/.

[11] TingxingTim Dong, Hao Yan, Mayank Parasar, and Raun Krisch. 2022. RenderSR: A Lightweight Super-Resolution Model for Mobile Gaming Upscaling. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR'22) Workshops*. New Orleans, LA, 3086–3094. https://doi.org/10.1109/CVPRW56347.2022.00348

[12] Open 3D Foundation. 2024. Open 3D Engine. https://o3de.org/.

[13] Epic Games. 2024. Unreal Engine. https://www.unrealengine.com/.

[14] Philippe Graff, Xavier Marchal, Thibault Cholez, Stephane Tuffin, Bertrand Mathieu, and Olivier Festor. 2021. An Analysis of Cloud Gaming Platforms Behavior under Different Network Constraints. In *Proceedings of the 17th International Conference on Network and Service Management (CNSM)*. 551–557. https://doi.org/10.23919/CNSM52442.2021.9615562

[15] Hua-Jun Hong, Fan-Chiang Tao-Ya, Cheng-Hsin Hsu, Kuan-Ta Chen, Chun-Ying Huang, and Cheng-Hsin Hsu. 2014. GPU consolidation for cloud games: Are we there yet?. In *Proceedings of the 13th Annual Workshop on Network and Systems Support for Games (NetGames)*. 1–6. https://doi.org/10.1109/NetGames.2014.7008969

[16] IETF. 2024. Media over QUIC (MoQ). https://datatracker.ietf.org/group/moq/about/.

[17] Youngjin Kim, Yubin Choi, Young Choon Lee, Hyuck Han, and Sooyong Kang. 2022. E-Render: Enabling UHD-Quality Cloud Gaming Through Edge Rendering. *IEEE Access* 10 (7 2022), 72107–72119.

[18] kixelated. 2023. moq-rs: Rust library for Media over QUIC. https://github.com/kixelated/moq-rs. GitHub repository, accessed 2024-08-23.

[19] Li Lin, Xiaofei Liao, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang, and Bo Li. 2014. LiveRender: A Cloud Gaming System Based on Compressed Graphics Streaming. In *Proceedings of the 22nd ACM International Conference on Multimedia (MM)*. 347–356.

[20] Martin Malik. 2024. HWiNFO: System Information and Diagnostics Tool. https://www.hwinfo.com/.

[21] Ioannis Pantazopoulos and Spyros Tzafestas. 2002. Occlusion culling algorithms: A comprehensive survey. *Journal of Intelligent and Robotic Systems: Theory and Applications* 35, 2 (2002). https://doi.org/10.1023/A:1021175220384

[22] Giacomo Parolini, Dario Maggiorini, Davide Gadia, and Laura Anna Ripamonti. 2022. Distributed Rendering for Video Games via Object Streaming. In *Proceedings of the IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*. Bologna, 209–214.

[23] PCGuide. 2024. Our graphics settings guide for Star Wars Outlaws on PC. https://www.pcguide.com/software/guide/best-graphics-settings-for-star-wars-outlaws/. Accessed 2024-08-23.

[24] Jon Peddie. 2019. *Ray Tracing: A Tool for All*. Springer.

[25] Zhengwei Qi, Jianguo Yao, Chao Zhang, Miao Yu, Zhizhou Yang, and Haibing Guan. 2014. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming. *ACM Transactions on Architecture and Code Optimization* 11, 2 (6 2014), 1–25. https://doi.org/10.1145/2632216

[26] Giampaolo Rodola. 2024. Psutil. https://github.com/giampaolo/psutil. GitHub repository, accessed 2024-08-23.

[27] Ryan Shea, Di Fu, and Jiangchuan Liu. 2015. Cloud Gaming: Understanding the Support From Advanced Virtualization and Hardware. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 12 (12 2015), 2026–2037. https://doi.org/10.1109/TCSVT.2015.2450172

[28] Xinkun Tang, Ying Xu, Feng Ouyang, Ligu Zhu, and Bo Peng. 2023. A Cloud-Edge Collaborative Gaming Framework Using AI-Powered Foveated Rendering and Super Resolution. *International Journal on Semantic Web and Information Systems (IJSWIS)* 19, 1 (4 2023), 1–19.

[29] Unity Technologies. 2024. Unity Engine. https://unity.com/products/unity-engine.

[30] Deniz Ugur. 2024. TransparentSR. https://github.com/DenizUgur/TransparentSR. GitHub repository, accessed 2024-08-23.

[31] Z. Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. 2004. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing* 13, 4 (4 2004), 600–612. https://doi.org/10.1109/TIP.2003.819861

[32] Songyin Wu, Sungye Kim, Zheng Zeng, Deepak Vembar, Sangeeta Jha, Anton Kaplanyan, and Ling-Qi Yan. 2023. ExtraSS: A Framework for Joint Spatial Super Sampling and Frame Extrapolation. In *Proceedings of ACM SIGGRAPH*. New York, NY, USA, 1–11. https://doi.org/10.1145/3610548.3618224

[33] Lei Xiao, Salah Nouri, Matt Chapman, Alexander Fix, Douglas Lanman, and Anton Kaplanyan. 2020. Neural supersampling for real-time rendering. *ACM Transactions on Graphics* 39, 4 (7 2020). https://doi.org/10.1145/3386569.3392376

[34] Sipeng Yang, Qingchuan Zhu, Junhao Zhuge, Qiang Qiu, Chen Li, Yuzhong Yan, Huihui Xu, Ling-Qi Yan, and Xiaogang Jin. 2024. Mob-FGSR: Frame Generation and Super Resolution for Mobile Real-Time Rendering. In *Proceedings of the ACM SIGGRAPH 2024 Conference Papers*. 1–11.

[35] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. 2019. Improving Cloud Gaming Experience through Mobile Edge Computing. *IEEE Wireless Communications* 26, 4 (4 2019), 178–183.

[36] Zhi Li, Anne Aaron, Ioannis Katsavounidis, Anush Moorthy, and Megha Manohara. 2016. Toward A Practical Perceptual Video Quality Metric.