

Received 23 May 2022, accepted 24 June 2022, date of publication 1 July 2022, date of current version 13 July 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3187696

RESEARCH ARTICLE

E-Render: Enabling UHD-Quality Cloud Gaming Through Edge Rendering

YOUNGJIN KIM¹, YUBIN CHOI¹, YOUNG CHOON LEE², HYUCK HAN³,
AND SOOYONG KANG¹, (Member, IEEE)

¹Department of Computer Science, Hanyang University, Seoul 04763, South Korea

²School of Computing, Macquarie University, Sydney, NSW 2109, Australia

³Department of Computer Science, Dongduk Women's University, Seoul 02748, South Korea

Corresponding author: Sooyong Kang (sykang@hanyang.ac.kr)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant by the Korean Government through MSIT (Decentralized High Performance Consensus for Large-Scale Blockchains, 50%) under Grant 2021-0-00590, and in part by the National Research Foundation of Korea (NRF) Grant by the Korean Government through MSIP (Collaborative IoT-Mobile-Edge Computing for Intelligent IoT Service Optimization, 50%) under Grant 2019R1A2C1011009.

ABSTRACT Cloud gaming services enable users to enjoy various high-quality computer games without installing them in their devices which may have poor resources. However, it consumes high bandwidth in backbone networks as game videos have to be constantly streamed to client devices, which can be a hindering factor for providing Ultra High Definition (UHD)-quality cloud gaming services whose demands are ever increasing. In this paper, we present a novel cloud gaming architecture, *E-Render*, in which the graphics engine is separated from the cloud game server and is deployed in the edge for rendering. To realize it, however, we need to effectively reduce the amount and latency of graphics commands transfer from the game engine in the cloud to the graphics engine in the edge server. E-render solves the problem by using novel techniques including multi-granular command deduplication and asynchronous buffer binding. We implement a prototype cloud gaming platform based on the E-Render architecture, and evaluate its performance using four real world games. The evaluation results confirm that E-Render can provide UHD-quality cloud gaming services.

INDEX TERMS Cloud gaming, edge service, graphics commands streaming, command deduplication.

I. INTRODUCTION

The rapid progress of cloud computing and network technologies have enabled users to enjoy high-quality computer (or mobile) games even with low-end devices through the cloud gaming service. Both game users and developers benefit from the cloud gaming service. Users can enjoy various brand-new high-quality games on demand, using a lightweight application called 'thin client', without game installation and hardware upgrades. For game developers, they are enough to implement and maintain only a single version of games for the cloud platform, which eliminates the burden of implementing and maintaining various versions of games for different platforms (i.e., hardware and OS) of user devices. Owing to such benefits, the world market in cloud gaming service already

reached \$1 billion in 2021, and is expected to steadily grow to \$7.24 billion in 2027 [1].

Figure 1 illustrates a typical software architecture of cloud gaming service [2]. The thin client program accepts and sends user input to a game instance in the cloud server. The game instance delivers user input received from the thin client to its game engine. In the game engine, the physics engine updates the game state according to the user input, and the rendering engine, to generate updated video frames, issues and executes graphics commands by calling library functions in the graphics engine. Such video frames, together with an audio stream generated by the audio engine, are encoded and streamed to the thin client to be decoded and played on the user device.

Of importance in the cloud gaming is the latency between the user input generation and the display of updated video frames by the input [3], [4]. The maximum tolerable latency

The associate editor coordinating the review of this manuscript and approving it for publication was Mehdi Sookhak¹.

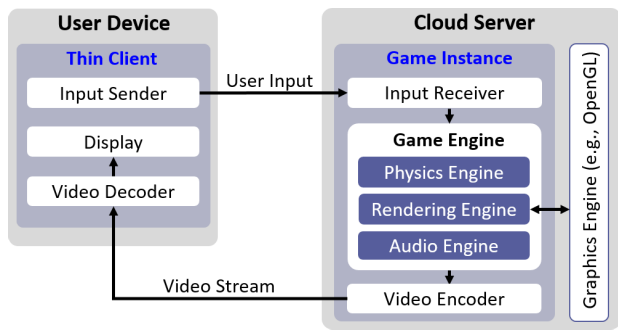


FIGURE 1. Typical architecture of the cloud gaming service.

for a highly latency-sensitive game play (e.g., racing game) is known to be 100 ms [5]. Another important quality metric for cloud gaming service is the quality of video, which is determined by the resolution and frame rate of video streams [2]. For a smooth display of game scenes, 720p (HD) resolution and 30 frames per second (fps) are known to be the minimum requirements. However, in the current local gaming environment where advanced graphics cards and high resolution display devices are available, high-end games that support 4K ($3,840 \times 2,160$) resolution and ≥ 60 fps are prevalent. To satisfy game users who are accustomed to such local gaming environment, cloud gaming service is evolving to provide UHD-quality (4K, 60 fps) high-end gaming environment to users [6]. For example, in Google Stadia [7] and Microsoft xCloud [8], users can setup 4K resolution and 60 fps of cloud gaming environment.

However, the current cloud gaming service architecture has difficulty in providing UHD-quality gaming service due to the large bandwidth requirement for video streams in the backbone network. Google Stadia requires at least 35 Mbps of backbone network bandwidth for a single UHD-quality video stream [7]. Considering ever increasing network traffic for various cloud services including multimedia streaming services, it is challenging for cloud gaming service providers to guarantee such amount of backbone network bandwidth to each of their users. The limited network bandwidth degrades the overall quality of cloud gaming services due to the increased latency and decreased video stream quality.

In this work, we incorporate edge computing to the cloud gaming service architecture and use remote rendering (via graphics streaming [9], [10]) to realize the UHD-quality cloud gaming service. We separate the graphics engine from the cloud game server and locate it to servers in the edge of the backbone network (i.e., edge servers). Graphics commands issued by the cloud game engine are transferred to the graphics engine in the edge server for frame rendering. Then, video frames generated in the edge server are directly streamed to the client device through high-bandwidth (≥ 35 Mbps) subscriber networks, such as 5G mobile network and FTTH (fiber-to-the-home) network. This approach gives us an opportunity to effectively reduce the backbone network bandwidth consumption and so enable UHD-quality cloud gaming service.

TABLE 1. Bandwidth & latency problems in remote rendering.

	CMDs	Bandwidth	Rendering	FPS
Truck town [13]	2,450	45.3 Mbps	16.5 ms	60
Voxel [14]	618	11.9 Mbps	20.8 ms	48
cf. Legacy Archi.		28 Mbps	≤ 16.6 ms	60

* 'Rendering' time includes the graphics commands transfer time and the rendering time for a frame in the edge server. Commands ('CMDs') and 'Rendering' are average per-frame values, while 'Bandwidth' and 'FPS' are average per-second values.

There have been a few recent studies that examined a similar 'remote rendering' approach, but for a different objective, i.e., relieving resource contention in the cloud game server [11], [12]. While they achieved their goal using the approach, they did not present and address its two possible side effects, *bandwidth and latency inflation problems* (shown below), resolving of which is essential to benefit from the approach for enabling UHD-quality cloud gaming services.

Table 1 shows the average per-frame rendering time and the consumed bandwidth for graphics commands transfer when we use naive remote (edge) rendering architecture for two example 4K, 60 fps 3D games in Godot game engine. For this experiment, we captured every graphics command (including data which amounts up to hundreds of KB) in the cloud game server (an Amazon EC2 instance) and transferred each of them to our own edge server in which the OpenGL graphics engine is running. The Truck town game issued average 2,450 commands per frame (148K commands per second) and they consumed 45.3 Mbps of bandwidth to be transferred to the edge server, which is larger than the consumed bandwidth for video streaming (28 Mbps) in the legacy cloud gaming architecture (*bandwidth inflation problem*). In the Voxel game, although it issued far less commands than Truck town game and so used less bandwidth than a video streaming, its rendering time (average 20.8 ms) exceeds the tolerable limit ($\leq 1/60$ sec. ≈ 16.6 ms per frame)¹ for providing 60 fps of frame rate, and therefore, it shows only 48 fps of frame rate. Such large rendering time is due to the dynamically generated objects, depending on user viewpoint and location, which incur time-consuming synchronous communications between game and edge servers while rendering the frame (*latency inflation problem*). (We will elaborate on this issue in Section III-C.)

The above preliminary experimental results quantitatively show the importance of addressing the aforementioned two problems. To that end, we set up two goals for our approach; 1) the amount of network traffic for transmitting graphics commands from the game server to the edge server should be noticeably less than the video stream traffic, and 2) the possibly increased latency due to the separation of graphics engine from the game server should be within the tolerable margin. To achieve the goals, we propose a novel

¹The tolerable limit is a theoretical bound, and the client-side video buffering can relax this limit for ordinary video streaming services such as Video-on-Demand service. However, it is not recommended in the cloud gaming since it incurs additional latency in game play.

cloud gaming service architecture, *E-Render*, which incorporates edge servers to the cloud gaming service architecture. E-Render effectively reduces the amount of network traffic for graphics commands transfer by *multi-granular commands deduplication*, and suppresses the latency increase by *asynchronous buffer binding*. The evaluation results show that E-Render enables us to provide UHD-quality cloud gaming services by achieving the goals. The contributions of this work is three fold:

- We design a novel cloud gaming service architecture, E-Render, which decouples the graphics engine from the cloud game server and locates it to the edge servers, to enable UHD-quality cloud gaming services.
- We devise two core schemes, multi-granular commands deduplication and asynchronous buffer binding, that enable us to practically benefit from the edge rendering.
- We implement a prototype E-Render which can be used transparently with legacy game engines without requiring any modification to them, and evaluate its performance using real systems and UHD-quality games.

We assume that edge servers equipping GPU-enabled high-performance graphics cards are prevalent along the edge of the backbone network, e.g., in each 5G base station.

The rest of this paper is organized as follows. Section II discusses related work. Sections III and IV present E-Render focusing on its core functionalities and implementation details, respectively. In Section V, we evaluate the performance of E-Render, and in Section VI, we present the limitations of E-Render and our plans to address them. Finally, in Section VII, we conclude this paper.

II. RELATED WORK

During the past decade, lots of efforts have been made to improve the quality of cloud gaming services in two directions; performance enhancement in the legacy service architecture and exploring new (mostly, distributed) service architectures.

A. IMPROVEMENTS IN THE LEGACY SERVICE ARCHITECTURE

Efforts in this category deal with either the system resource management or video frame generation in the game server. In [15]–[17], they focused on the efficient management of computation resources across game instances (virtual machines) in the cloud game server. In [18]–[21], they proposed various algorithms to improve the quality of user experiences (QoE). Specifically, a prediction-based frame generation algorithm is proposed in [18] which speculates on the user input and generates next frames in advance. If the speculation fails, it generates the frames again using the actual input. In [20], they proposed an adaptive frame dropping scheme that drops some non-critical frames to save the network bandwidth under the congested state, thus sacrificing video quality in exchange for the tolerable latency. In [21], they developed a rendering system that interpolates other

frames based on the key frames using a three-dimensional (3D) image warping algorithm. The system improved the video encoding performance by reducing bit rates of non-key frames.

The above works either are orthogonal to this work [15]–[18] or have clear limitation in reducing the backbone network bandwidth consumption [20], [21].

B. EXPLORING NEW SERVICE ARCHITECTURES

Efforts in this category proposed new cloud gaming service architectures that also exploit computing resources near the client device for QoE enhancement.

1) REMOTE HOSTING

In [22]–[24], they proposed a service architecture in which edge servers, instead of cloud servers, host game instances to reduce the network latency and increase QoE. While the approach greatly reduces both the backbone network bandwidth consumption and latency, it has three important problems desired to be solved. First (scalability problem), an edge server has limited amount of resources to host a sufficiently large number of game instances for users in its coverage. Second (migration problem), when a mobile user moves out of the coverage of an edge server, the entire game instance for the user also needs to migrate to other edge server, which is harmful to latency-sensitive games. Finally (initialization problem), since a single edge server cannot host all games that cloud servers host, we may need a time-consuming on-demand game deployment and initialization in the edge server.

To address the scalability problem, a resource sharing scheme among game instances in the edge server [25] and a hybrid cloud-edge gaming service architecture in which a game instance is hosted in either the cloud or edge server depending on the user's latency requirement [26] have been proposed. A fog computing approach [27] that deploys a game instance to other GPU-enabled user device (not the edge server) near the client device can also be a solution to the problem, while it cannot guarantee the service availability. In [24], they proposed a novel game instance migration scheme, as a solution to the migration problem, which duplicates an active game instance in both the original and new edge servers, temporarily during the migration time. In [25], they also proposed a solution to the initialization problem in which only the minimal set of game assets necessary to start a game is downloaded to the edge server first, for the rapid start of the game play. While such enhancement schemes alleviate aforementioned problems in a certain degree, they could not clearly remove the problems from this service architecture.

2) REMOTE RENDERING

In contrast to the coarse-grained service deployment (i.e., deploying the entire game instance) to other servers/devices in remote hosting architecture, other efforts tried to offload only a portion (i.e., rendering and video encoding/streaming) of the entire game engine functionalities to

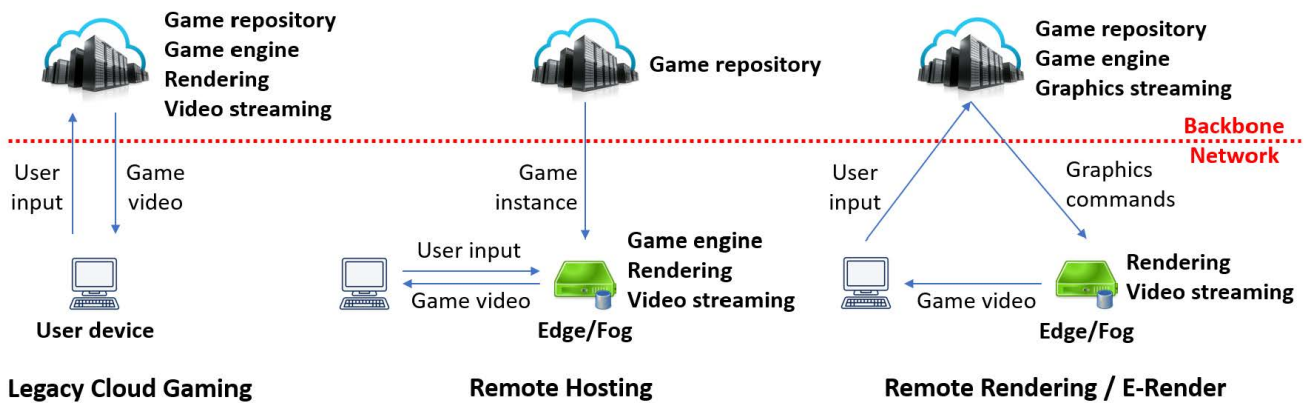


FIGURE 2. Comparison of the cloud gaming service architectures: E-Render is a remote rendering architecture equipped with novel schemes that address bandwidth and latency inflation problems.

other servers/devices, to lower the resource contention in cloud game servers. In [11], they proposed a selective rendering architecture in which the rendering subsystem is located in both the cloud server and the client device to be selectively used for frame rendering. In [12], they proposed a fog computing-based service architecture that exploits unused resources in other computing devices (e.g., idle desktops) near the client device, for rendering and video streaming on behalf of the cloud server. In [28], they proposed a resource allocation scheme for multiplayer cloud gaming, which selects edge servers and communication paths for each game user while minimizing inter-player delay. Although the above works alleviated the resource contention in game servers and thus saved the cost for maintaining cloud servers, they did not address the bandwidth and latency inflation problems in graphics commands transfer which are critical to enable UHD-quality cloud gaming service as presented in Section I. A client rendering approach, in which the cloud game server transfers graphics commands directly to the client device for local rendering, has been proposed in [29] and [30]. However, since it cannot support low-end client devices, which is one of primary goals of cloud gaming services, we do not further discuss on them in this paper.

3) COMPARATIVE ANALYSIS

Figure 2 compares the above cloud gaming service architectures in terms of the roles and actions of each participating entity in the cloud gaming. For their comparison in the performance's viewpoint, in this section, we compare them in terms of four metrics; backbone traffic amount, latency, scalability and operation cost. Among them, the amount of backbone network traffic and latency determines the QoE, and the scalability and operation cost determines the overall service overhead. The operation cost includes game engine deployment and instance initialization cost, and game instance migration cost.

The legacy architecture consumes a large amount of backbone network bandwidth and shows a moderate latency, while it is highly scalable and has no operation cost. The remote

hosting architecture achieves better QoE than other architectures since the gaming service is provided near the client device. However, it has larger service overhead than others due to the scalability, migration and initialization problems, although enhancement schemes improved scalability and operation cost in some degree. Also, in multi-user games, user events need to be transferred to other users (possibly in the coverage of other edge servers) for global game state updates, which requires game engine modification.

The remote rendering architecture has far less service overhead than the remote hosting architecture since it uses fine-grained offloading (i.e., better scalability) and has neither migration nor initialization problem. Selective rendering architecture [11] further improves its scalability by also using cloud rendering. However, its operation cost is larger than the legacy architecture due to the cost of managing fog devices [12] or rendering subsystems in both cloud servers and client devices [11]. Also, the bandwidth and latency inflation problems in transferring graphics commands to remote devices makes it achieve worse QoE than others.

The proposed architecture, E-Render, in this paper targets to achieve better QoE than both the legacy and prior remote rendering architectures by effectively addressing the bandwidth and latency inflation problems, without additional operation cost and any game engine modification.

III. DESIGN OF E-RENDER

E-Render is a novel cloud gaming architecture which incorporates edge computing for remote rendering. What makes E-Render different from the prior fog-based remote rendering [12] is the manageability of the remote rendering device. While a large number of (GPU-enabled) fog devices (i.e., user devices near the client device) may improve the scalability of the remote rendering architecture, we can neither guarantee the availability of such devices nor manage (modify) their software to support cloud gaming. On the other hand, edge servers are fully manageable by service providers, which enables us to optimize their software for effective cloud gaming services. E-Render exploits their manageability to address the bandwidth and latency inflation problems.

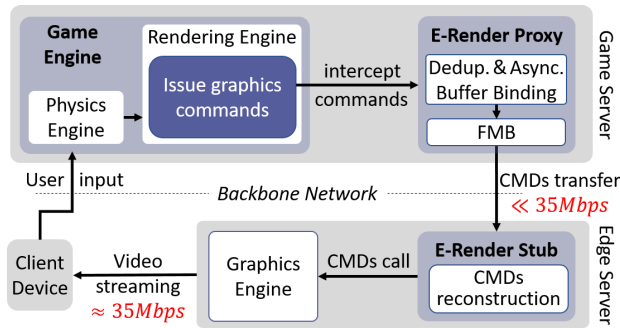


FIGURE 3. Cloud gaming service architecture in E-Render.

A. OVERALL SERVICE ARCHITECTURE

Figure 3 shows the overall service architecture of E-Render. E-Render decouples the graphics engine from legacy game engines and locates it to edge servers. It consists of two core components: E-Render Proxy in the cloud game server and E-Render Stub in the edge server. On receiving a user input from the client device, the physics engine in the game server updates the game state and the rendering engine issues graphics commands to generate updated game frames. The E-Render proxy intercepts such graphics commands (CMDs) and transfers them to the E-Render stub, *transparently*, so that we do not need to modify legacy game engines. An issued command by the rendering engine may contain data (e.g., geometry and texture data for game frames) in its parameters [31]. In this paper, we collectively call them (command + data) as a *command*. In Section IV, we will elaborate how to transfer command arguments (i.e., data), especially pointer type arguments, to the E-Render stub.

The essence of the E-Render proxy and stub are their solutions to the bandwidth and latency inflation problems, *multi-granular command deduplication* and *asynchronous buffer binding*, that practically enable UHD-quality cloud gaming services. They collaborate each other to reduce both the amount of network traffic and latency for graphics commands transfer. Specifically, the E-Render proxy deduplicates graphics commands for a frame using multi-granular command deduplication scheme, and removes the command transfer latency inflation by using asynchronous buffer binding scheme. Deduplicated graphics commands for a frame are temporarily stored in the *frame message buffer* (FMB) until the last command for the frame arrives, and then transferred as a whole to the E-Render stub which reconstructs the original graphics commands from the received ones. Such frame-unit commands transfer (i.e., transferring the entire commands for a frame, at once) further reduces the amount of network traffic. Finally, the E-Render stub executes reconstructed commands by calling them one after another, to generate a video frame. As in the legacy architecture, generated video frames are encoded and streamed to the client device for playback. In the following sections, we design two core schemes of E-Render in detail.

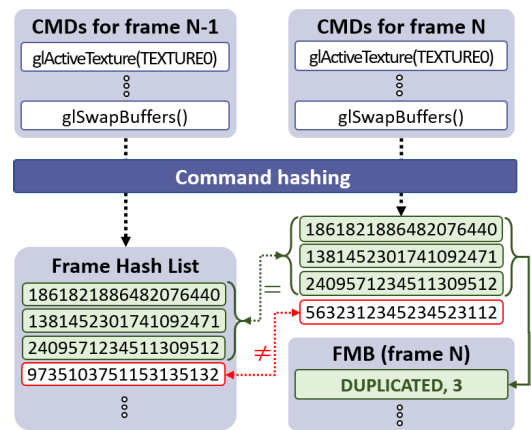


FIGURE 4. Deduplication w/ inter-frame commands similarity.

B. MULTI-GRANULAR COMMAND DEDUPLICATION

The rendering engine iterates a loop (i.e., graphics pipeline) in which a series of graphics commands (command sequence) are issued for a frame. We pay attention to two favorable properties observed from command sequences for different frames. In a high frame rate video, the rendering engine issues similar command sequences if there is no scene change between two successive frames (*inter-frame commands similarity*). Also, an object may appear in multiple different frames, either successive or not, in which case the rendering engine issues similar graphics commands to draw the object (*inter-object commands similarity*). Our multi-granular commands deduplication scheme exploits such characteristics of the rendering engine to effectively reduce the amount of network traffic for graphics commands transfer. Specifically, graphics commands for a frame pass through three steps; 1) InterFrame-dedup, 2) InterObject-dedup, and 3) command sequence transfer.

1) INTERFRAME-DEDUP: DEDUPLICATION WITH INTER-FRAME COMMANDS SIMILARITY

The graphics pipeline in an ordinary rendering engine has a *fixed* command generation pattern; applying the camera viewpoint → applying the environmental condition (e.g., lighting and shading) → drawing background models (e.g., sky and mountains) → drawing foreground models (e.g., characters). Such fixed pattern gives us an opportunity to exploit the inter-frame commands similarity for command deduplication. Without rapid scene changes caused by user input, successive frames may have identical viewpoint, environment and background models which are all static. Then the rendering engine generates identical sequences of commands for those frames until issuing foreground model drawing commands. Such identical, initial subsequence of commands can be effectively deduplicated, by our InterFrame-dedup, to save the backbone network bandwidth.

To that end, the E-Render proxy maintains a *frame hash list* which contains a sequence of hash values for each graphics command of the *previous* frame, as shown in Figure 4. From

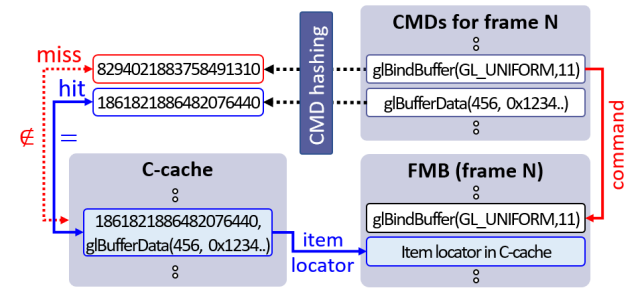


FIGURE 5. Deduplication w/ inter-object commands similarity.

the first command issued for a new frame, the E-Render proxy hashes each command and compares its hash value with the corresponding one in the frame hash list. Such comparison continues until it finds a different hash value in the frame hash list. Then, the proxy stores a message, ‘DUPLICATED, n ’, in the frame message buffer, where n is the number of initial commands until finding a different hash value, and proceeds to the next step (InterObject-dedup) of the multi-granular command deduplication. The frame hash list is emptied and refilled by hash values for the new frame before transferring commands in the frame message buffer to the E-Render stub (Section III-B3).

When there is no dynamic (or autonomously moving) objects in the current game scene, the game engine generates the same frames until a user input arrives. During that quiescent period, the rendering engine issues exactly the same sequences of graphics commands for successive frames. For those frames, the InterFrame-dedup finds that all hash values for a new frame are identical with those in the frame hash list. Then our multi-granular command deduplication scheme skips the next step and transfers the sole message (‘DUPLICATED, n ’) in the frame message buffer to the E-Render stub. Such case maximizes the effectiveness of the multi-granular command deduplication scheme.

2) INTEROBJECT-DEDUP: DEDUPLICATION WITH INTER-OBJECT COMMANDS SIMILARITY

InterObject-dedup prevents the E-Render proxy from re-transmitting frequently issued commands that have been transferred to the E-Render stub from the beginning of the game. Differently from the InterFrame-dedup which exploits the inter-frame commands similarity, InterObject-dedup exploits the inter-object commands similarity; graphics commands for rendering the same object in different frames can be identical and thus be reused. In particular, as a game has either a smaller number of different objects or a larger number of static (or still) objects, such identical (thus deduplicable) commands are expected to appear more frequently, increasing the effectiveness of the command deduplication.

For the InterObject-dedup, the E-Render proxy maintains a command cache (*C-cache*) that contains (hash value, command)-pairs of frequently issued individual commands, as shown in Figure 5. For each graphics command issued, the

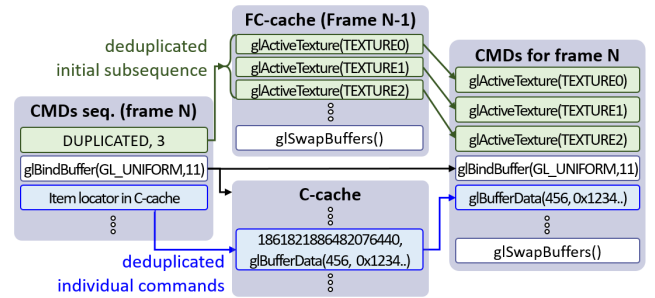


FIGURE 6. Command sequence reconstruction in the stub.

E-Render proxy hashes it to find the identical command from the command cache. If it exists, the proxy stores its in-cache location information (detailed in Section III-B4), instead of the command itself, in the frame message buffer. Otherwise, the proxy stores the command as it is, and also stores its (hash value, command)-pair in the command cache. If the command cache has no room for the new pair, it is handled based on a least recently used (LRU) replacement algorithm.

The size difference between a graphics command and its in-cache locator makes the InterObject-dedup effective. While the size of a graphics command varies from tens of Bytes to hundreds of KB, depending on both the type of command and the complexity of an object the command draws, the size of a locator is far smaller than that. Hence, the InterObject-dedup can effectively save the backbone network bandwidth.

3) COMMAND SEQUENCE TRANSFER

Deduplicated commands in the frame message buffer are transferred to the E-Render stub after all graphics commands for a frame have been issued by the rendering engine and our multi-granular command deduplication scheme has stored every message (DUPLICATED, item locator or command itself) for each of them in the frame message buffer. When the rendering engine finishes generating all drawing commands for a frame, it finally issues a *SwapBuffer* command. The E-Render proxy, upon intercepting it, recognizes that all graphics commands have been issued for a frame, and thus transfers the command sequence in the frame message buffer to the E-Render stub.

4) COMMAND SEQUENCE RECONSTRUCTION

Upon receiving the command sequence for a new frame, the E-Render stub builds a full sequence of graphics commands for the frame by reconstructing deduplicated commands in the sequence. To that end, it maintains a *frame command cache* (FC-cache) and a command cache. The frame command cache contains all graphics commands for the just rendered preceding frame.

Figure 6 shows the process of the command sequence reconstruction. On receiving the command sequence for a new frame, the E-Render stub first reconstructs initial subsequence of commands deduplicated by the InterFrame-dedup,

if '(DUPLICATED, n)' message appears in the beginning of the sequence, by copying them from the frame command cache. In Figure 6, three initial commands have been deduplicated and thus copied from the frame command cache. Then the stub reads subsequent messages in the sequence, one by one, to reconstruct remaining commands. If a message is an item locator, the stub directly goes to the location in its command cache and copies the command in that location. Otherwise, since the message is a command itself, the stub just copies the command for reconstruction, and again inserts it into the command cache with its hash value. Upon processing the last message, the stub appends a `SwapBuffer` command and finishes the command sequence reconstruction.

Of importance here is the use of in-cache location information, instead of the hash value, for transferring an individually deduplicated command by the InterObject-dedup. Note that any hash function may generate identical hash values for different keys. When two different commands happen to have identical hash values, the E-Render stub cannot distinguish them each other in its command cache using their hash values. Hence, when we transfer the hash value of a deduplicated command, it can be reconstructed to a different command in the command cache that has the same hash value.

Such problem can be resolved by transferring in-cache locator instead of the hash value. If we maintain a hash table with chaining for collision resolution, items with the same hash value can be distinguished using their different locations in the hash table. Provided that command caches in both the proxy and stub are always kept identical, in terms of the item set they store and the in-cache location of each item, an item locator can identify the same command in both command caches and therefore, the stub can reconstruct individually deduplicated commands correctly. To that end, E-Render uses the same hash structure, cache size and replacement policy in the proxy and stub. Then, we can always keep two command caches identical, since the proxy and stub access to their command caches for exactly the same sequence of commands.

The above hash collision problem may also occur in the InterFrame-dedup. However, differently from the InterObject-dedup, a hash value is compared with only one hash value in a specific location in the frame hash list. In that case, the probability of hash collision is extremely low, provided that we use a hash function with sufficiently large hash space. Hence, we do not consider the problem in the InterFrame-dedup.

C. ASYNCHRONOUS BUFFER BINDING

As a game play progresses, lots of objects newly appear and disappear dynamically. For a new object in a frame, the rendering engine first issues a buffer generation command to allocate a buffer object in GPU memory, and the address (or name)² of the buffer object is returned as an output of the

²OpenGL and Vulkan engines use different references to the buffer object, name and address, respectively. In this paper, we collectively call them 'buffer address' for the sake of convenience.

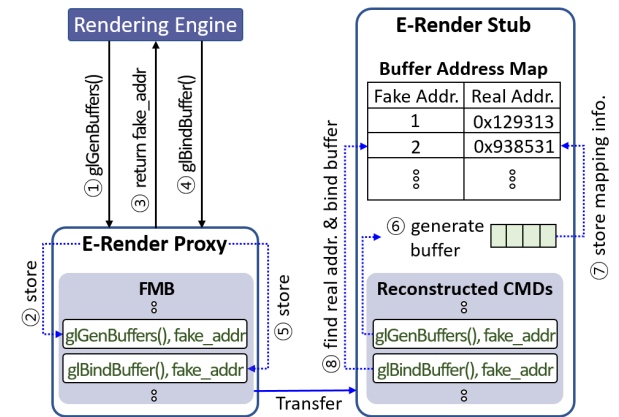


FIGURE 7. Asynchronous buffer binding.

command. Then the rendering engine binds the buffer object to one of predefined targets (i.e., context), to set up its internal state. To that end, it issues a buffer binding command which uses the buffer address returned by the buffer generation command for the binding. Hence, the buffer generation command must be *executed* before issuing the buffer binding command.

In the remote rendering architecture, such buffer binding process necessitates a synchronous communication between the cloud game server who only issues commands and the remote rendering server who executes them. Specifically, the game server can issue the buffer binding command only after the preceding buffer generation command has been executed in the remote rendering server and its output (buffer address) has arrived to the game server. Such synchronous communication for buffer binding increases the overall rendering latency in the remote rendering architecture, as presented in Section I.

E-Render addresses the latency inflation problem using our asynchronous buffer binding scheme, as shown in Figure 7. On intercepting a buffer generation command (e.g., `glGenBuffers`), the E-Render proxy immediately generates a fake buffer address and delivers the command with the fake address to our multi-granular command deduplication module which stores the command in the frame message buffer (1 and 2). Then the command is returned with the fake address so that the succeeding buffer binding command can be issued immediately using the fake address (3). This removes the time-consuming synchronous communication between game and edge servers. The issued buffer binding command (e.g., `glBindBuffer`) and the fake address is also delivered to the deduplication module to be stored in the frame message buffer (4 and 5). Then the proxy waits for the succeeding graphics commands.

The real buffer generation and binding is done by the E-Render stub. When it finds a buffer generation command in the reconstructed command sequence, it executes the command and gets the real buffer address returned (6). Then it stores the (fake address, real address) pair in its buffer address mapping table which maintains mapping information between fake and real addresses for future accesses to

buffer objects (⑦). When the E-Render stub finds a buffer binding command in the reconstructed command sequence, it first translates the given fake address to the real address by looking up the table, and then executes the command using the real address (⑧). An entry in the buffer address map is removed upon receiving a corresponding delete command (e.g., `glDeleteBuffers`).

IV. PROTOTYPE IMPLEMENTATION

The prototype E-Render [32] is written in C++ language targeting to support OpenGL version 3.0 or later. We implemented the E-Render proxy as a dynamic library. For the graphics commands interception in the E-Render proxy, we implemented wrapper functions in the proxy for each of OpenGL graphics commands (total 262 commands in our current prototype implementation) defined in the OpenGL loader (`glad`), and loaded the proxy using `LD_PRELOAD` when the game engine starts so that our wrapper functions can override original OpenGL functions. Such function overriding enables us to intercept graphics commands without modifying the game engine itself. A wrapper function assigns arguments of the intercepted command to its own (corresponding) parameters. A pointer argument is not copied directly but dereferenced so that its *value* is assigned to the corresponding (non-pointer) parameter of the wrapper function. The E-Render stub, upon receiving such dereferenced arguments, assigns their memory addresses to corresponding pointer parameters of the reconstructed graphics command. As such, E-Render enables the remote execution of graphics commands that have pointer type parameters. The wrapper function, not the original graphics command, is the target of our multi-granular command deduplication and is stored in the frame message buffer after deduplication.

E-Render uses ZeroMQ [33] for communications between the proxy and stub. A message (`message_t` structure in ZeroMQ) for a non-deduplicated command consists of three fields; command name (2 bytes), arguments for non-pointer parameters (variable size) and optionally arguments for pointer parameters (variable size). In the command name field, we store a deduplication flag (2 bits, set to 00) and a unique command number (14 bits); we enumerated all OpenGL commands to assign a unique number to each command. A message (2 bytes) for deduplicated command sequence by the InterFrame-dedup consists of a deduplication flag (set to 10) and the number of deduplicated commands (14 bits). Finally, a message (4 bytes) for a deduplicated command by the InterObject-dedup contains a deduplication flag (set to 11), reserved bits (6 bits) and an in-cache locator (20 bits bucket ID + 4 bits chain index). The `multipart_t` structure is used to materialize the frame message buffer which contains a sequence of `message_t` structures.

We use `std::hash` in C++ with a non-cryptographic 64-bits hash function (`MurmurHashUnaligned2`) for command hashing in the multi-granular command deduplication. The frame hash list and frame command cache, used for the InterFrame-dedup and reconstruction, are implemented as

singly linked lists, since they are always accessed sequentially from the first item. The command cache is materialized by `std::unordered_map` which has average $O(1)$ complexity in searching, inserting or removing an element [34]. To enable item accesses using in-cache locator, we added two member functions, `get_index(key)` and `direct_find(locator)`, which returns the location (bucket ID and chain index) of an input key and an item in the input locator, respectively. To implement the LRU replacement policy in the command cache, we maintain a separate LRU list (`std::list`) of hash values for cached commands. An element, once selected as a victim, is removed from the list and then the corresponding element in the command cache is also removed. The cache size is controlled by limiting the total number of elements in it. When the number reaches the limit, replacement occurs whenever a new element is to be inserted to the cache.

Multi-granular command deduplication incurs additional overhead in game engines. During the InterFrame-dedup, it takes $O(n)$ time for each frame to hash commands and compare hash values in the frame hash list, where n is the number of commands in a frame. During the InterObject-dedup, it takes $O(n)$ amortized time for command hashing and command cache lookup. Overall, the E-Render proxy incurs $O(n)$ overhead for each frame in the cloud game server. During the command sequence reconstruction in the edge server, the E-Render stub requires $O(n)$ amortized time to look up both frame command cache and command cache for each frame. In the middle of the command sequence reconstruction, the stub needs to look up the buffer address mapping table for asynchronous buffer binding, when it encounters a buffer binding command. However, noting that both the number of buffer binding commands in a frame and the number of entries in the mapping table are small, such lookup overhead is negligible.

V. PERFORMANCE EVALUATION

In this section, we compare E-Render with prior work in terms of the backbone network bandwidth consumption and latency. As we presented in Section II, prior work can be categorized into ‘Legacy Cloud Gaming’, ‘Remote Hosting’ and ‘Remote Rendering’, based on their service architectures. In terms of the above performance metrics which we focus on, efforts in the same category naturally show almost identical performance due to their identical service architectures. Hence, instead of comparing individual schemes, we collectively compare them with E-Render in the names of their architectural categories. Among them, schemes in the ‘remote hosting’ category show the best performance with no doubt since they do not use backbone network. However, they have clear limitations to become practical due to the scalability, migration and initialization problems, and thus we do not include them in our comparative study.

A. EXPERIMENTAL ENVIRONMENT

We used three Linux machines, a cloud game server, an edge server and a client device. For the cloud game server, we used

TABLE 2. Characteristics of the example games.

	Dodge	Hexa	Voxel	Truck
Dimension	2D	2D	3D	3D
View	3rd-person	3rd-person	1st-person	1st-person
Camera movement	×	✓	✓	✓
Dynamic object generation	×	×	✓	×
Light	×	×	×	✓
Gravity	×	×	✓	✓
Avg. # of CMDs per frame	285	592	618	2,450

a g4dn.2xlarge Amazon EC2 instance in Asia Pacific (Seoul) region (Intel Xeon Platinum 8259CL CPU of 2.5 GHz, 32 GB RAM and NVidia T4 graphics card) with 25 Gbps network bandwidth. For an edge server, we used our own PC (located in Seoul) which equips Intel Core i7-8700 (3.2GHz) CPU, 16 GB RAM, NVIDIA Geforce RTX 2070 Super 8 GB model and an 1 Gbps Ethernet adaptor. The client device is a laptop computer connected to the internet through 802.11ac WiFi AP to which the edge server is also connected via 1 Gbps ethernet port.

We installed the Godot game engine [35] and E-Render proxy in the cloud server, and the E-Render stub and OpenGL in our edge server. Multiple instances of E-Render stub may run in parallel, to support concurrent users in the edge's coverage, and thus we containerized the stub using Docker. We implemented and installed a simple thin client application to the client device. The application maintains a TCP connection for user input transfer to the cloud game server. We used *steam* [36] as a video streaming server in both the cloud and edge servers, and *Steam Link* [37] as a streaming client for the thin client application. The *Steam* and *Steam Link* keep a UDP connection for video streaming.

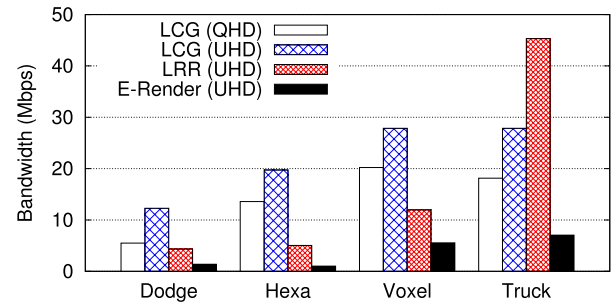
Four example games in Godot game engine, Dodge the Creeps (shortly, 'Dodge') [38], Hexagonal Map (shortly, 'Hexa') [39], Voxel and Truck Town (shortly, 'Truck'), are used for experiments, and their characteristics are shown in Table 2. We played each game for five minutes with active interactions, and averaged measured values.

B. FEASIBILITY OF THE UHD-QUALITY CLOUD GAMING SERVICES

We first evaluate the performance of E-Render, in terms of the backbone network bandwidth consumption, displayed frame rate and latency, to show the feasibility of the UHD-quality cloud gaming service using E-Render.

1) BACKBONE NETWORK BANDWIDTH CONSUMPTION AND DISPLAYED FRAME RATE

Figure 8 compares the backbone network bandwidth consumption among three architectures, legacy cloud gaming (LCG), legacy remote rendering (LRR) and E-Render, with the command cache size set to maximum 1,000 elements. The consumed bandwidth in this figure is the average of 300 samples (60 sec. $\times$ 5 min.) of data each of which is the amount of bandwidth consumed for a second. In LCG, about 12 Mbps, 20 Mbps, 28 Mbps and 28 Mbps of bandwidth is consumed for streaming UHD-quality videos of Dodge, Hexa, Voxel and

**FIGURE 8.** Backbone network bandwidth consumption. (Command cache size: 1,000 and Quad-HD (QHD) resolution: 2,560 $\times$ 1,440.)**TABLE 3.** Average number of dropped frames per second. (Values in the parenthesis are the displayed frame rates on the client device.)

	LCG (UHD)	LRR	E-Render
Dodge	5.54 (54.46)	1.89 (58.11)	0.08 (59.92)
Hexa	6.11 (53.89)	1.27 (58.73)	1.32 (58.68)
Voxel	6.72 (53.28)	12.80 (47.20)	1.41 (58.59)
Truck	6.75 (53.25)	1.99 (58.01)	1.63 (58.37)

Truck games, respectively, depending on the complexity of their video frames. As anticipated, 3D games consume more bandwidth than 2D games given the same video quality. The LRR consumes less bandwidth than LCG in all games except the Truck town game (about 45 Mbps) which generates a huge amount of graphics commands for each frame and thus has the bandwidth inflation problem. E-Render consumes only about 1.4 Mbps, 1.0 Mbps, 5.5 Mbps and 7.0 Mbps of bandwidth, which amount to about 11%, 5%, 20% and 25% of the LCG's consumption for Dodge, Hexa, Voxel and Truck games, respectively. In particular, its bandwidth consumption is far less than even that for QHD-quality gaming in LCG architecture. Noteworthy is the effectiveness of the command deduplication in E-Render for the Truck town game, in which E-Render saved 84% of bandwidth compared to LRR, solving the bandwidth inflation problem.

In terms of the video frame rate displayed on the client device, the LCG showed average 53.7 fps for all games due to (presumably) the unstable backbone network bandwidth availability. The LRR achieved about 58.3 fps for all games except the Voxel game in which it showed average 47.2 fps (minimum 30 fps) due to the latency inflation problem. It is interesting that the frame rate of LRR did not decrease in the Truck game, despite its bandwidth inflation problem. It is due to the 'idle time exploitation effect' in commands transfer, which will be elaborated in the next section. E-Render, which not only consumes far less backbone network bandwidth than others but also solves the latency inflation problem, achieved approximately 58.4 ~ 59.9 fps for all games. Table 3 shows the average number of dropped frames per second, due either to the network or to the rendering latency, which caused the decreased frame rates in LCG and LRR.

2) LATENCY

Figure 9 compares the latency among three architectures. For this experiment, we measured the average latency from the

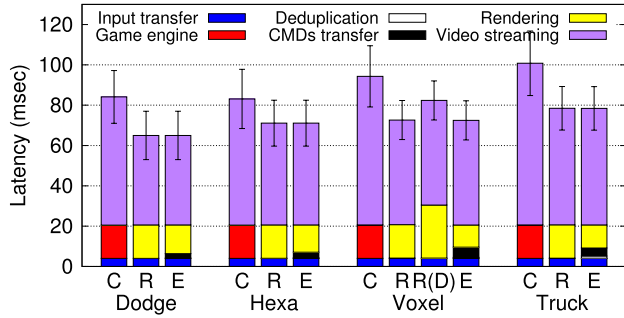


FIGURE 9. Latency breakdown in LCG (C), LRR (R) and E-Render (E): R and R(D) in the Voxel game represents frames w/o and w/ dynamic object generation, respectively.

user input generation to the display of the updated video frame by the input. In E-Render, the ‘Deduplication’ time includes command generation, deduplication and buffering times in the game server, and the ‘Rendering’ time includes commands reconstruction and execution times in the edge server. The ‘Video streaming’ time includes the streaming latency and the decoding time in the client. In the LRR architecture, since commands are not buffered but transferred to the edge upon generation, commands transfer and execution are performed in parallel. Hence, we combined them into the ‘Rendering’ time. We issued 899 (Dodge), 2,076 (Hexa), 600 (Voxel) and 2,576 (Truck) user inputs during the game play, and show their average latency in the figure. Since the latency itself directly affects the quality of user experiences, infrequent but large latency can seriously degrade the overall quality of user experiences. Hence, in addition to the average latency, we show the standard deviation of the measured latency using error bars.

Interestingly, the results are quite the opposite to our anticipation that ‘decoupling graphics engine from the game server would inevitably increase the gaming latency’. Detailed investigation revealed that the game engine, when it finishes rendering a frame before its deadline (i.e., 16.6 ms for providing 60 fps), it does not proceed to process the next frame but becomes idle until the deadline. The ‘Game engine’ portions in LCG include such idle times, and hence they are identical across all games regardless of the possibly different times for doing ‘real’ work in each game. E-Render effectively exploited such idle times for command deduplication, transfer and reconstruction, and could start video streaming without additional delay in all games. The ‘Rendering’ portions in LRR and E-Render also include such idle times, although they are less than those in LCG. The only factor that differentiates the overall latency is the video streaming latency. Since E-Render exploits sufficient bandwidth and low latency of the subscriber network for streaming, it shows less latency than LCG which uses competitive backbone network. The LRR, in R(D) case, could not finish Rendering within the deadline due to the latency inflation problem, and thus shows larger latency than others.

The experimental results in Figures 8 and 9 confirm that E-Render makes it feasible to provide UHD-quality cloud

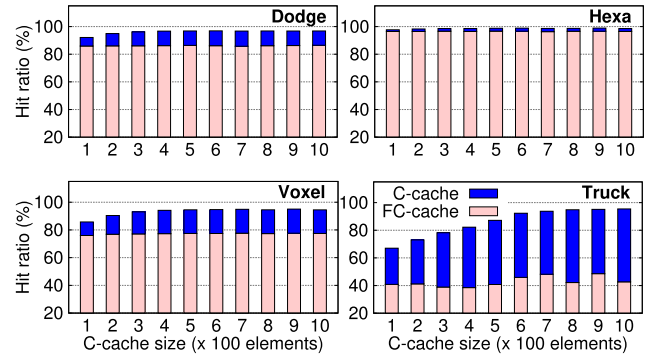


FIGURE 10. Cache hit ratios with varying command cache size.

gaming service owing to its multi-granular command deduplication and asynchronous buffer binding schemes.

C. DETAILED ANALYSIS

We now examine the effects of multi-granular command deduplication and asynchronous buffer binding, in detail.

1) CACHE HIT RATIO

Figure 10 shows cache hit ratios, varying the command cache size, which determine the effectiveness of the multi-granular command deduplication in E-Render. Since the frame command cache stores all commands for the very previous frame, its size changes depending on the complexity of the frame and we did not explicitly control its size. We averaged hit ratios of total 18,000 frames (60 frames $\times$ 300 sec.).

An interesting result, which is common across all games, is high frame command cache hit ratio that makes the InterFrame-dedup very effective. As stated in Section III-B1, the fixed command generation pattern in ordinary rendering engines produces identical initial commands (for applying camera viewpoint and environmental condition, and drawing background objects) in successive frames, which can be safely deduplicated by the InterFrame-dedup. Such effect is maximized in the Hexa game where the background image rarely changes and thus only those commands for drawing a moving foreground object (i.e., player character) are different between successive frames. The Truck game, which changes an environmental condition (i.e., light) as the player’s truck moves, shows the lowest frame command cache hit ratio. The notable differences of the frame command cache hit ratio among different command cache sizes in Truck game are due to the non-identical user actions in each experiment.

The effectiveness of the InterObject-dedup depends on the command cache size; the larger its size is, the higher hit ratio we obtain and thus the more effective the InterObject-dedup becomes. Except in the Hexa game, in which most commands are hit in the frame command cache, the command cache hit ratio increases until its size becomes 400, 500 and 800 elements in Dodge, Voxel and Truck games, respectively. As a result, with only 800 elements of command cache size, we can achieve about 96.8%, 98.6%, 94.5% and 94.8% of total cache hit ratios in Dodge, Hexa, Voxel and Truck games,

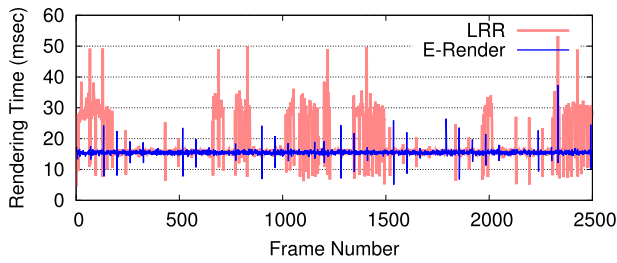


FIGURE 11. Individual frame rendering time in the Voxel game.

TABLE 4. Average CPU and host memory utilization.

Resource		LCG	LRR	E-Render
Cloud	CPU	0.8%	2.3%	1.3%
	Memory	117 MB	93 MB	105 MB
Edge	CPU	—	1.8%	1.2%
	Memory	—	67 MB	83 MB

respectively. Such high cache hit ratios contribute to the large backbone network bandwidth savings, as shown in Figure 8.

2) INDIVIDUAL FRAME RENDERING TIME

To investigate the effectiveness of the asynchronous buffer binding scheme, we measured individual frame rendering times for the initial 2,500 frames in the Voxel game in both LRR and E-Render. In LRR, as shown in Figure 11, the rendering time for a frame rapidly increases whenever one or more new objects are dynamically created in the frame, due to the latency inflation problem. Such large rendering time results in not only the delayed display of the frame but also the decreased frame rate (cf. the large overall latency in the LCG due to the large streaming latency, shown in Figure 9, also delays the display of frames but does not affect the frame rate). The asynchronous buffer binding scheme solves the latency inflation problem and enables E-Render to show stable rendering times. Note that peaks in E-Render is due not to the latency inflation problem but to the delayed arrival of graphics commands when the backbone network is instantaneously congested.

3) RESOURCE CONSUMPTION

To investigate the overhead of E-Render due to its command deduplication and asynchronous buffer binding schemes, we measured average CPU utilization and host memory footprint in the cloud and edge servers, when we play the Truck game which consumes more resources than others.

Table 4 shows the results. Although E-Render consumes slightly more CPU than LCG in the cloud server due to its command deduplication and transfer, its overall CPU utilization is not noticeable in both servers. Interestingly, E-Render consumes less CPU power than LRR which does not perform command deduplication. It owes to the E-Render's command buffering; all commands for a frame are aggregated in the buffer and transferred as a whole, which greatly reduces the number of network packet transmissions compared to the individual command transfer used in the LRR.

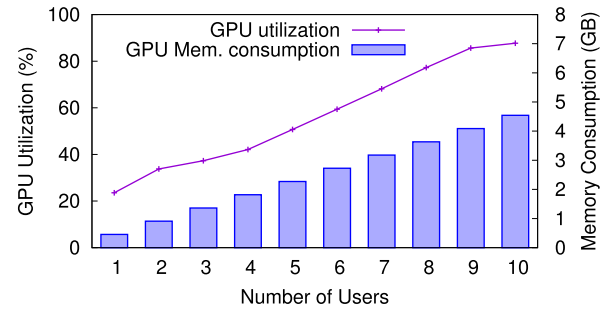


FIGURE 12. GPU resource utilization in the edge server.

The game engine and the Truck game itself consumes about 77 MB of cloud memory in all architectures. In addition, the LCG consumes about 40 MB of memory for graphics command processing (OpenGL) in the cloud, while the LRR and E-Render consumes the same amount of memory in the edge server. E-Render consumes about 12 MB and 16 MB more memory in the cloud and edge servers, respectively, than LRR for command deduplication (mainly for the cache management). The E-Render proxy and stub themselves, excluding their caches, consume about 16 MB and 28 MB of memory in the cloud and edge servers, respectively.

GPU is the most scarce resource in cloud gaming services. Since the GPU workloads in each architecture are not different each other, LCG, LRR and E-Render consume the same amount of GPU resources in either the cloud (LCG) or edge (LRR and E-Render) servers. However, the limited amount of GPU resources in an edge server poses a restriction on the server's service capacity in LRR and E-Render. To investigate the per edge service capacity of E-Render, we measured the GPU utilization and GPU memory consumption of the Truck game, increasing the number of concurrent game users (i.e., containers running the E-Render stub), in an edge server (Figure 12). In this experiment, we also used the Truck game which consumes the most amount of GPU resources.

The GPU memory consumption for a single Truck game was approximately 465 MB and increased in proportion to the number of users. The GPU utilization increased to 85.6% for nine users without noticeable delay in rendering video frames. However, with ten concurrent users, the GPU became saturated and number of frames have been dropped due to the delayed rendering, which limited the per edge service capacity in E-Render to nine for the Truck game. Such capacity depends on both the specification and the number of GPU devices being used in the edge server. As more powerful GPU devices are being introduced in the market, we believe that the rendering capacity of an edge node will gradually increase making the edge rendering more feasible.

The above results confirm that, using a reasonable amount of edge resources, E-Render enables UHD-quality gaming services, in terms of the bandwidth consumption and latency.

VI. LIMITATIONS AND FUTURE WORK

As stated in Section V-B2, E-Render effectively exploits the idle time, after it finishes rendering a frame until the

affordable rendering deadline, for its own work (i.e., command deduplication, transfer and reconstruction). However, for those frames whose rendering times are almost the same as the tolerable limit, E-Render cannot obtain such overhead hiding effect and therefore its overhead can be exposed to the user, which is one of limitations of the current E-Render design. To address this problem, we plan to devise a pipelining scheme that transfers commands for the next frame while the edge server is rendering the previous frame, by which we can obtain the overhead hiding effect without such idle time. However, current game engines have been designed to start generating commands for the next frame only after finishing the rendering of the current frame, which hinders such pipelining. Overcoming such unfavorable behavior of game engines is the major technical challenge in devising the pipelining scheme.

In terms of the service architecture in E-Render, there is a clear limitation in the service scalability. Since the frame rendering is performed in the edge servers with limited GPU resources, the rendering capacity in an edge server limits the gaming service capacity within the edge's coverage. To make E-Render fully scalable like the legacy cloud gaming architecture, the collaborative rendering between the cloud and edge servers is indispensable. In this regard, we plan to devise a selective remote rendering architecture in which graphically undemanding, latency-sensitive games are prioritized to be rendered in an edge server taking the server's rendering capacity into account, while other games are rendered in the cloud.

VII. CONCLUSION

While the demand for high-quality cloud gaming services is ever increasing, the legacy cloud gaming architecture has difficulty in providing satisfactory QoE for UHD-quality games due to the large backbone network bandwidth consumption of video streams. In this work, we propose a novel cloud gaming service architecture, E-Render, in which graphics commands are streamed to an edge server, and the edge server renders and streams video frames to client device through high bandwidth subscriber network. E-Render effectively saves the backbone network bandwidth by deduplicating graphics commands and prevents latency inflation due to the separation of graphics engine from the game server by asynchronously binding dynamically generated buffers. We implemented a prototype E-Render system and evaluated its performance on the real world environment. Evaluation results show that E-Render saves 75% ~ 95% of backbone network bandwidth consumption compared with the legacy architecture without increasing the overall gaming latency, and thus enables UHD-quality cloud gaming services.

REFERENCES

- [1] P. Bajpai. (2021). *How Cloud Computing is Changing the World of Games and Gaming*. [Online]. Available: <https://www.nasdaq.com/articles/how-cloud-computing-is-changing-the-world-of-games-and-gaming-2021-02-24>
- [2] R. Shea, J. Liu, E. C.-H. Ngai, and Y. Cui, "Cloud gaming: Architecture and performance," *IEEE Netw.*, vol. 27, no. 4, pp. 16–21, Jul./Aug. 2013.
- [3] R. D. Yates, M. Tavan, Y. Hu, and D. Raychaudhuri, "Timely cloud gaming," in *Proc. IEEE Conf. Comput. Commun.*, May 2017, pp. 1–9.
- [4] O. S. Peñaherrera-Pulla, C. Baena, S. Fortes, E. Baena, and R. Barco, "Measuring key quality indicators in cloud gaming: Framework and assessment over wireless networks," *Sensors*, vol. 21, no. 4, p. 1387, Feb. 2021.
- [5] M. Claypool and K. Claypool, "Latency and player actions in online games," *Commun. ACM*, vol. 49, no. 11, pp. 40–45, Nov. 2006.
- [6] R. Shea, D. Fu, and J. Liu, "Cloud gaming: Understanding the support from advanced virtualization and hardware," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 12, pp. 2026–2037, Dec. 2015.
- [7] Google Stadia. Accessed: Jul. 7, 2022. [Online]. Available: <https://stadia.google.com/>
- [8] Microsoft XCloud. Accessed: Jul. 7, 2022. [Online]. Available: <https://www.xbox.com/en-US/xbox-game-streaming/project-xcloud/>
- [9] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J. P. Laulajainen, R. Carmichael, V. Pouloupoulos, A. Laikari, P. Perälä, A. De Gloria, and C. Bouras, "Platform for distributed 3D gaming," *Int. J. Comput. Games Technol.*, vol. 2009, pp. 1–15, Jan. 2009.
- [10] A. Jurgelionis, F. Bellotti, A. D. Gloria, P. Eisert, J. Laulajainen, and A. Shani, "Distributed video game streaming system for pervasive gaming," in *Proc. 6th Streaming Day*, 2009, pp. 1–6.
- [11] J. Bulman and P. Garrahan, "A cloud gaming framework for dynamic graphical rendering towards achieving distributed game engines," in *Proc. USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2020, pp. 1–7.
- [12] Y. Lin and H. Shen, "CloudFog: Leveraging fog to extend cloud gaming for thin-client MMOG with high quality of service," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 431–445, Feb. 2017.
- [13] 3D Truck Town Demo. Accessed: Jul. 7, 2022. [Online]. Available: <https://godotengine.org/asset-library/asset/524>
- [14] 3D Voxel Demo. Accessed: Jul. 7, 2022. [Online]. Available: <https://godotengine.org/asset-library/asset/676>
- [15] C.-Y. Huang, C.-H. Hsu, Y.-C. Chang, and K.-T. Chen, "GamingAnywhere: An open cloud gaming system," in *Proc. 4th ACM Multimedia Syst. Conf. (MMSys)*, 2013, pp. 36–47.
- [16] H.-J. Hong, D.-Y. Chen, C.-Y. Huang, K.-T. Chen, and C.-H. Hsu, "Placing virtual machines to optimize cloud gaming experience," *IEEE Trans. Cloud Comput.*, vol. 3, no. 1, pp. 42–53, Jan./Mar. 2015.
- [17] Y. Zhang, P. Qu, J. Cihang, and W. Zheng, "A cloud gaming system based on user-level virtualization and its resource scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1239–1252, May 2016.
- [18] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn, "Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming," in *Proc. 13th Annu. Int. Conf. Mobile Syst., Appl., Services*, May 2015, pp. 151–165.
- [19] J. R. Lange, P. A. Dinda, and S. Rossoff, "Experiences with client-based speculative remote display," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 419–432.
- [20] J. Wu, C. Yuen, N.-M. Cheung, J. Chen, and C. W. Chen, "Enabling adaptive high-frame-rate video streaming in mobile cloud gaming applications," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 12, pp. 1988–2001, Dec. 2015.
- [21] S. Shi, C.-H. Hsu, K. Nahrstedt, and R. Campbell, "Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming," in *Proc. 19th ACM Int. Conf. Multimedia*, 2011, pp. 103–112.
- [22] X. Zhang, H. Chen, Y. Zhao, Z. Ma, Y. Xu, H. Huang, H. Yin, and D. O. Wu, "Improving cloud gaming experience through mobile edge computing," *IEEE Wireless Commun.*, vol. 26, no. 4, pp. 178–183, Aug. 2019.
- [23] M. Viitanen, J. Vanne, T. D. Hamalainen, and A. Kulmala, "Low latency edge rendering scheme for interactive 360 degree virtual reality gaming," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2018, pp. 1557–1560.
- [24] W. Zhang, J. Chen, Y. Zhang, and D. Raychaudhuri, "Towards efficient edge cloud augmentation for virtual reality MMOGs," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, Oct. 2017, pp. 1–14.
- [25] A. Bhojan, S. P. Ng, J. Ng, and W. T. Ooi, "CloudyGame: Enabling cloud gaming on the edge with dynamic asset streaming and shared game instances," *Multimedia Tools Appl.*, vol. 79, no. 43, pp. 32503–32523, 2020.
- [26] S. Choy, B. Wong, G. Simon, and C. Rosenberg, "A hybrid edge-cloud architecture for reducing on-demand gaming latency," *Multimedia Syst.*, vol. 20, no. 5, pp. 503–519, Oct. 2014.

- [27] B.-I. Oros and V. I. Bacu, "RenderLink remote rendering platform for computer games: A WebRTC solution for streaming computer games," in *Proc. IEEE 16th Int. Conf. Intell. Comput. Commun. Process. (ICCP)*, Sep. 2020, pp. 555–561.
- [28] P. Parastar, F. Pakdaman, and M. R. Hashemi, "FRAME-SDN: A fair resource allocation for multiplayer edge-based cloud gaming in SDN," in *Proc. 25th ACM Workshop Packet Video (PV)*, 2020, pp. 21–27.
- [29] X. Liao, L. Lin, G. Tan, H. Jin, X. Yang, W. Zhang, and B. Li, "LiveRender: A cloud gaming system based on compressed graphics streaming," *IEEE/ACM Trans. Netw.*, vol. 24, no. 4, pp. 2128–2139, Aug. 2016.
- [30] P. Eisert and P. Fechteler, "Low delay streaming of computer graphics," in *Proc. 15th IEEE Int. Conf. Image Process.*, Oct. 2008, pp. 2704–2707.
- [31] M. Segal and K. Akeley, *The OpenGL Graphics System: A Specification, Version 4.5 (Core Profile)*. Accessed: Jul. 7, 2022. [Online]. Available: <https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf>
- [32] *E-Render*. Accessed: Jul. 7, 2022. [Online]. Available: <https://github.com/hyu-splab/edge-render>
- [33] *ZeroMQ: An Open-Source Universal Messaging Library*. Accessed: Jul. 7, 2022. [Online]. Available: <https://zeromq.org/>
- [34] *Cppreference.Com (Std: Unordered_Map)*. Accessed: Jul. 7, 2022. [Online]. Available: https://en.cppreference.com/w/cpp/container/unordered_map
- [35] *Godot Engine*. Accessed: Jul. 7, 2022. [Online]. Available: <https://godotengine.org/>
- [36] *Steam*. Accessed: Jul. 7, 2022. [Online]. Available: <https://store.steampowered.com>
- [37] *Steam Link*. Accessed: Jul. 7, 2022. [Online]. Available: <https://store.steampowered.com/remoteplay#anywhere>
- [38] *2D Dodge The Creeps Demo*. Accessed: Jul. 7, 2022. [Online]. Available: <https://godotengine.org/asset-library/asset/515>
- [39] *2D Hexagonal Map Demo*. Accessed: Jul. 7, 2022. [Online]. Available: <https://godotengine.org/asset-library/asset/111>



laborative distributed computing), and data-intensive computing.

YOUNG CHOON LEE received the B.Sc. (Hons.) and Ph.D. degrees from The University of Sydney, Australia, in 2004 and 2008, respectively. He is currently a Senior Lecturer with the School of Computing, Macquarie University, Sydney, Australia. His research focus is on scheduling and resource management in distributed systems. His current research interests include cloud data center efficiency and sustainability, edge computing and cloud computing (collaborative distributed computing), and data-intensive computing.



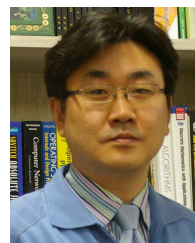
HYUCK HAN received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Seoul National University, Seoul, South Korea, in 2003, 2006, and 2011, respectively. From 2011 to 2013, he has worked as a Senior Researcher at Samsung. Since March 2014, he has been with the Department of Computer Science, Dongduk Women's University, Seoul, as an Assistant Professor. His research interests include operating systems, database systems, and distributed systems.



YOUNGJIN KIM received the B.S., M.S., and Ph.D. degrees in computer science from Hanyang University, Seoul, South Korea, in 2013, 2015, and 2021, respectively. Since September 2021, he has been with Samsung Electronics, Suwon, South Korea, as a Senior Researcher. His research interests include distributed computing systems, storage systems for cloud computing, big data processing, edge computing, and cloud computing.



YUBIN CHOI received the B.S. degree in business administration and computer science from Gachon University, Seoul, South Korea, in 2021. She is currently pursuing the master's degree in computer science with Hanyang University, Seoul. Her research interests include software platforms for edge computing and cloud computing.



SOOYONG KANG (Member, IEEE) received the B.S. degree in mathematics and the M.S. and Ph.D. degrees in computer science from Seoul National University (SNU), Seoul, South Korea, in 1996, 1998, and 2002, respectively. He was a Postdoctoral Researcher at the School of Computer Science and Engineering, SNU. Since March 2003, he has been with the Department of Computer Science, Hanyang University, Seoul, as a Professor. From January 2017 to February 2018, he was a Visiting Professor with the Department of Computing, Macquarie University, Sydney, Australia. His research interests include storage systems, distributed computing systems, mobile cloud computing, edge computing, and the Internet of Things.

...