Improving Resource and Energy Efficiency for Cloud 3D through Excessive Rendering Reduction

Tianyi Liu
tianyi.liu@utsa.edu
University of Texas at San Antonio

Jerry Lucas
jlucas8427@gmail.com
University of Texas at San Antonio

Sen He
senhe@arizona.edu
University of Arizona

Tongping Liu
tongping@umass.edu
University of Massachusetts Amherst

Xiaoyin Wang
xiaoyin.wang@utsa.edu
University of Texas at San Antonio

Wei Wang
wei.wang@utsa.edu
University of Texas at San Antonio

Abstract

The rise of cloud gaming makes interactive 3D applications an emerging type of data center workload. However, the excessive rendering in current cloud 3D systems leads to large gaps between the cloud and client frame rates (FPS, frames per second), thus wasting resources and power. Although FPS regulation can remove excessive rendering, due to the highly-varying frame processing time and the use of rendering delays, existing cloud FPS regulation solutions have low FPS and slow motion-to-photon (MtP) latency, causing violations of Quality-of-Service (QoS) requirements.

In this paper, we present a novel cloud FPS regulation solution, called OnDemand Rendering (ODR). ODR employs multi-buffering, dynamic rendering delay/acceleration, and input processing prioritization to reduce excessive rendering and ensure QoS satisfaction. ODR was evaluated in our private cloud and Google cloud. Evaluation results showed that ODR effectively removed excessive rendering, thus improving DRAM performance by 19% and reducing power usage by 16% over no FPS regulation. Better memory efficiency also allowed ODR to increase client FPS by 5.5%. Moreover, ODR reduced average MtP latency by more than 92% and outperformed existing FPS regulations. More importantly, ODR's high FPS and low latency make it feasible to deploy 3D applications to conventional public clouds.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; **Real-time system architecture**; • **Computing methodologies** → **Graphics processors**.

Keywords: Cloud Graphics Rendering, Resource and Energy Efficiency, FPS gaps, OnDemand Rendering, Priority Frames

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04.
https://doi.org/10.1145/3627703.3650064

ACM Reference Format:
Tianyi Liu, Jerry Lucas, Sen He, Tongping Liu, Xiaoyin Wang, and Wei Wang. 2024. Improving Resource and Energy Efficiency for Cloud 3D through Excessive Rendering Reduction. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3627703.3650064

1 Introduction

With cloud 3D systems, interactive real-time 3D applications, such as computer games and Virtual-Reality (VR) applications, are becoming a major type of workload for cloud computing [38, 39, 66]. In cloud 3D, 3D applications are rendered in the cloud, and the rendered frames are encoded and sent to the users through the network, allowing the users to enjoy graphics-intensive 3D applications without expensive and/or power-hungry high-end GPUs.

Similar to non-cloud 3D systems, Quality-of-Service (QoS), including the frame rate (FPS/frames-per-second) and motion-to-photon (MtP) latency, is the primary design metric for cloud 3D.[1] Besides QoS, system efficiency, including the energy and resource efficiency, is also an important design metric for cloud 3D, similar to other data center use cases [27, 80]. However, current cloud 3D systems usually emphasize QoS, and thus, may have low system efficiency.

A main cause of the low system efficiency in cloud 3D is **excessive rendering**. Figure 1 gives an example of excessive rendering with the cloud rendering and client decoding frame rates of two cloud 3D benchmarks, *Red Eclipse* and *InMind*, from the Pictor benchmark suite [50]. Figure 1 shows that there were **large FPS gaps** between the cloud rendering and client decoding for both benchmarks, indicating that the cloud servers were rendering at excessive FPS. When frames are rendered at a higher rate than they can be encoded, transmitted, or decoded, excessive frames are discarded. Energy and computing cycles are wasted on rendering these discarded frames. Excessive rendering also increases memory contention, further degrading resource efficiency.

[1] We will use FPS and frame rate interchangeably in this paper. We will also use MtP latency and latency interchangeably.

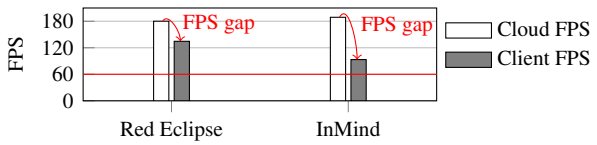


Figure 1. Excessive frame rendering causes large FPS gaps.

Excessive rendering also exists in non-cloud 3D and is usually addressed by **FPS regulation**, where frame rendering is delayed to reduce the FPS gap [60, 65]. Prior research also proposed to port existing FPS regulation solutions to the cloud [49]. However, when ported to the cloud, existing FPS regulation solutions usually have difficulties simultaneously reducing the excessive rendering while meeting the QoS requirements, due to the following challenges,

The first challenge comes from the variations of the frame processing time in cloud 3D. Compared to its non-cloud counterpart, cloud 3D requires additional frame processing steps than rendering, including frame copying, encoding, network transmission, and decoding. The processing time of each step varies from frame to frame. As shown in Section 4, this variation is one of the main causes of FPS gaps. Existing FPS regulation solutions either overlook these variations or incur high-overhead by collecting the varying timings as feedback. Therefore, existing solutions usually fail to meet the FPS requirements (e.g., 30 or 60FPS).

The second challenge comes from the extra rendering delays injected by existing FPS regulation solutions [60, 65]. These delays are required to synchronize the speed of different frame processing steps. These delays usually do not cause MtP latency issues in non-cloud 3D, as the latency before adding the delay is low. However, as the MtP latency is already high for cloud 3D, this extra delay can lead to violations of the maximum allowed MtP latency.

In this paper, we address the research problem of *how to regulate frame rates in cloud 3D to reduce excessive rendering for better energy and resource efficiency without sacrificing the QoS (i.e., client FPS and latency)*. We also seek an open-source solution for a wide audience to support those who cannot adopt proprietary cloud 3D services.

Here, we present a novel cloud FPS regulation solution, called *OnDemand Rendering* (*ODR*), that solves the above research problem. ODR has three components:

The first component is *multi-buffering*, which quickly synchronizes frame processing steps to reduce FPS gaps without collecting timing information. Multi-buffering has been traditionally applied in synchronizing rendering for non-cloud 3D [74]. Traditional multi-buffering synchronizes in 3D applications and GPU, which do not know the timings of encoding and network. Therefore, traditional multi-buffering does not work for cloud 3D. ODR, however, synchronizes FPS through multi-buffer swapping in the server proxy, which handles frame encoding and connects to the 3D application and network. Hence, the server proxy knows the varying frame processing time in every step, making it a better place to adjust frame rates to reduce FPS gaps.

The second component is the *FPS regulator* to ensure that FPS targets are always met. Unlike existing FPS regulations, ODR's FPS regulator does not only delay but also accelerates frame processing, so that the FPS target can be met even when the processing time suddenly increases.

The last component is called *PriorityFrame*, which addresses the increased MtP latency issue with the following observation: the majority of the rendered frames are triggered by 3D application's internal refreshes instead of user inputs. Consequently, we can prioritize the rendering of input-triggered frames to retain the benefits of FPS regulation without significantly increasing the MtP latency.

We evaluated ODR with six cloud 3D benchmarks in our private cloud and Google cloud, reflecting edge and public cloud deployments. The evaluation results showed that ODR effectively reduced excessive rendering and improved system efficiency without scarifying QoS: 1) ODR reduced the average FPS gap from 99.1 to 2.6 frames, which leads to an average power usage reduction of 16.0% over no FPS regulation. ODR also reduced DRAM read access time by 19% and improved Instruction-per-Cycle (IPC) by 14.4%. 2) When the QoS goal was to maximize FPS, ODR increased the average client FPS by 5.5% over no regulation, due to its better memory efficiency/performance. 3) When the QoS goals were 30 or 60 FPS, ODR met these FPS targets; 4) ODR reduced the average MtP latency by more than 92% over no regulation, due to PriorityFrame and reduced FPS gaps.

ODR also significantly outperform two state-of-the-art FPS regulation solutions. More specifically, when compared with interval-based regulation and Remote VSync [49], ODR increased the average client FPS by 62.4% and 34.7%, and decrease the MtP latency by 30.7% and 27.3%, respectively.

More importantly, the results on Google cloud had showed that ODR made it feasible to deploy 3D applications to conventional public clouds by satisfying the QoS requirements of 60FPS and 100ms latency.

The contributions of this paper include,

- A performance analysis of existing FPS regulation solutions that provides the insights required to design effective cloud FPS regulation solutions.
- ODR's multi-buffering and FPS regulator that effectively reduce FPS gaps while reaching the target FPS.
- The PriorityFrame of ODR that retains the benefits of FPS regulation without significantly increasing (and in many cases reducing) the "Motion-to-Photon" latency.
- The experimental evaluation of ODR to show the effectiveness and benefits of FPS regulation in cloud 3D, especially with the current public cloud and Internet.

The rest of this paper is organized as follows: Section 2 presents related work; Section 3 presents the background of cloud 3D; Section 4 analyzes the challenges faced by FPS

regulation in the clouds; Section 5 presents the design of ODR; Section 6 reports the experimental evaluation of ODR; and Section 7 concludes the paper.

2 Related Work

Frame Rate Regulation Non-cloud 3D applications also have excessive rendering, which causes a large gap between the frame rendering rate and the display's refreshing rate. This gap may lead to "screen tearing," which hurts user experience [29]. High FPS may also be purposely reduced to save energy, especially on mobile devices [36]. FPS regulation is extensively used to address excessive rendering on non-cloud 3D. Software and hardware FPS regulations typically delay frame rendering to ensure all frames are rendered at regular intervals. The length of this interval is determined by FPS targets or refreshing rates. For example, if the targeted FPS is 60, then one frame should be rendered at every 16.6ms interval ($\frac{1sec}{60} \approx 16.6ms$).

Software-based frame rate regulation solutions apply the delay in the main loops of 3D applications [24, 36, 60] to ensure a frame's rendering starts at the beginning of a regular interval. This "interval-based" FPS regulation assumes that most frames can be rendered within the interval. However, as shown later in Section 4.1, the processing time in cloud 3D has large variations, making this assumption invalid in cloud 3D, resulting in low QoS.

Vertical Synchronization (VSync) is a hardware-based FPS regulation solution to address the FPS gap between the rendering rate and display refreshing rate [65]. Current displays have vertical blanking (vblank) signals between the visibility of two consecutively-displayed frames [61]. These vblank signals happen regularly based on the display's refreshing frequency (e.g., every 16.6ms for 60Hz). With VSync, the rendering of a 3D application is synchronized to the blank signals. That is, the rendering is paused until the next vblank signal, allowing the rendering to synchronize with the display's refreshing rate [10]. For newer displays, VSync is extended to G-Sync and FreeSync to allow reversely adjusting of the display's refreshing rate [5, 64]. These hardware FPS regulations cannot be directly applied to cloud 3D, as the GPU and the client display are separated by the network.

Remote-VSync (RVS) was an FPS regulation solution that extended VSync to the cloud [49]. RVS was designed to remove the FPS gap by estimating the time difference between the end of a frame's decoding and the next vblank signal. This time difference is then sent back to the cloud to delay rendering the next frame. However, as shown in Section 4.1, due to the slow client-to-cloud feedback, RVS cannot quickly respond to frame-to-frame processing time variation, causing low client FPS and QoS violations.

Other Related Work. Several studies have proposed virtual desktop infrastructure and cloud gaming systems [34, 42, 62, 84]. ODR is designed to be generic and can be applied to these systems. Besides FPS regulation, prior research also investigated the proper FPS target or bit rate for cloud 3D to reduce bandwidth usage [1, 31, 32, 36, 75, 88]. These studies are orthogonal to our work, as we studied how to regulate FPS, whereas they provide the FPS target for the regulation. There were also studies on the cloud 3D systems and graphics systems, such as cloud 3D system designs [19, 39, 43, 44, 72], CPU-GPU coordination [7], GPU virtualization and sharing [3, 47, 53, 58, 89, 90], server provisioning [20], multi-GPU rendering [71, 82], network protocols [1, 25], edge computing [2, 11, 12, 18, 48, 85, 87], image compression [6, 9, 22, 41, 51, 55, 86], VR/360° video processing [33, 46, 56, 57, 91], and graphics hardware [23, 26, 35, 45, 54, 70, 81, 83]. These studies are also orthogonal to our work as they focus on other aspects of cloud 3D than excessive rendering and FPS regulation. Moreover, our work focuses on enabling a more efficient cloud 3D on public clouds with conventional hardware.

There are also commercial cloud/remote gaming services, such as NVIDIA GeForce Now [66], Microsoft XCloud [59], Shadow [73], Steam Link [76], and Parsec [68]. However, these services are proprietary, at least for their server-side implementations. In this work, we aim to seek an open-source solution for a wider audience and to support use cases that cannot adopt proprietary services.

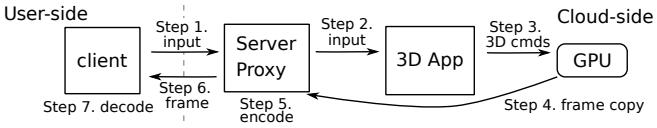


Figure 2. The architecture of a typical cloud 3D system.

3 Background on Cloud 3D

Cloud 3D Systems. There are several open-source implementations for cloud 3D systems, such as Furion [43], GamingAnyWhere [34], Sunshine [52] with Moonlight [28], and TurboVNC [17] with VirtualGL [16]. These systems typically follow the architecture illustrated in Figure 2 and operate in the following steps. In the first step (step 1 in Figure 2), the client captures the user input (e.g., a mouse click or VR headset motion) and sends this input to the cloud through the network. In the cloud, the server proxy captures the input and forwards it to the 3D application (step 2). The 3D application processes this input and generates the corresponding 3D rendering commands, which are sent to the GPU to render (step 3). After the rendering finishes, the rendered frame is copied to the server proxy (step 4). The server proxy then encodes the frame (step 5) into a specific format (usually a video frame) and transmits the encoded frame to the client through the network (step 6). Finally, the frame is decoded by the client and displayed on the screen (step 7). Note that frame rendering can also be triggered by 3D application's internal refreshes instead of user inputs.

In this paper, we define the steps of frame rendering (step 3), copying (step 4), encoding (step 5), network transmission (step 6), and decoding (step 7), as *frame processing steps*. These steps are usually carried out in parallel similar to CPU pipelines to maximize the FPS. Step1 and step2 are not included since not all the frames are triggered by user inputs. Figure 5a illustrates a specific implementation of the key steps of this software pipeline (more discussions in Section 4).

QoS Requirements. The QoS requirements for cloud 3D primarily focus on frame rates (FPS) and the MtP latency. For frame rates, there are two levels of QoS requirements: 1) recreation and education usually only demand that the frame rate is higher than a minimal *FPS target*, usually 30 or 60FPS [8, 69, 88, 89], whereas 2) competitive gaming and VR may require as high FPS as possible [4, 13, 21, 63].

MtP latency refers to the time between a user issues an input and the responding frame displayed on the screen. MtP latency requirements vary depending on the type of 3D applications. For example, for action-intensive VR, the latency should be less than 25ms [40, 43]. For action games, the latency should be less than 100ms, whereas other games' maximum latency can be 500ms or 1 second [14]. Nonetheless, latency is usually the lower the better [15].

In this paper, we consider the FPS requirements of maximizing FPS or 30/60 FPS. We also aim to reduce the latency as much as possible, as lower latency is always preferable.

4 Challenges for Cloud FPS Regulation

This section presents the design challenges and system efficiency benefits of FPS regulation in cloud 3D by experimentally analyzing existing FPS regulation solutions.

This analysis was conducted using the benchmark *InMind* (IM) from the Pictor benchmark suite [50]. The benchmark was executed with resolution 1280×720 in our private cloud, and the hardware information is provided in Section 6.1.

4.1 Impact of Processing Time Variation

As stated in the introduction, processing time variation is a main cause of FPS gaps and makes it challenging to reduce FPS gap while maintaining satisfying QoS. The following paragraphs illustrate this challenge by analyzing the behaviors of no FPS regulation and two existing FPS regulation solutions – Interval-based regulation and Remote VSync.

NoReg: No FPS regulation. Figure 3 gives the frame rendering and encoding FPS in the cloud, as well as client decoding FPS, under *NoReg*. As Figure 3 shows, there was a large gap of 96 frames between the rendering and encoding FPS, indicating serious excessive rendering. Moreover, the client (decoding) FPS was 93, indicating that *InMind* should be able to meet the 60FPS target.

Note that, one of the main causes of FPS gap is the processing speed differences between different steps – in this case (i.e., *InMind*), the speed difference between rendering and decoding. Indeed, current FPS regulation solutions focus primarily on reducing the FPS gaps caused by this speed difference. However, as shown next, the FPS gap caused by processing time variation is more difficult to mitigate.

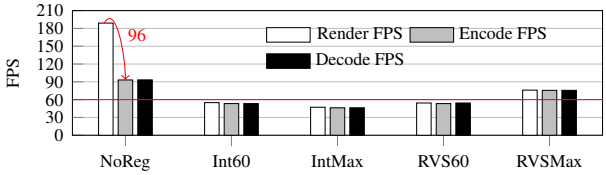
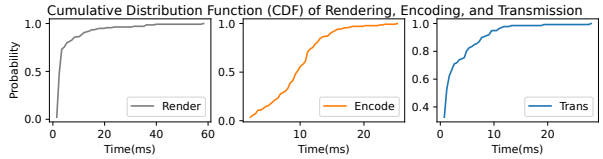
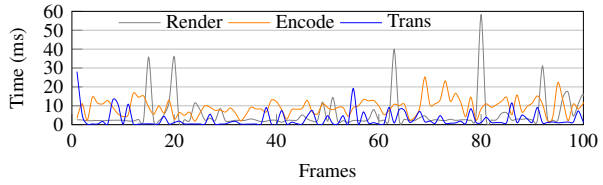


Figure 3. InMind's frame rendering, encoding and decoding FPS under different FPS regulations.



(a) CDF of the processing time of *InMind*'s frame rendering, encoding, and network transmission.



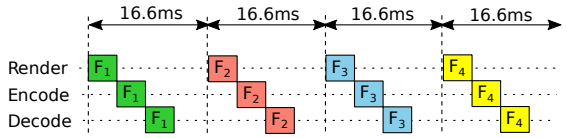
(b) A trace of the processing time of *InMind*'s frame rendering, encoding, and network transmission.

Figure 4. Processing time variation of *InMind*'s frame rendering, encoding, and network transmission (decoding time is relatively lower and hence omitted).

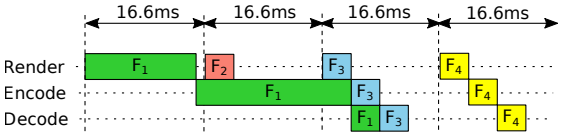
Int60: Interval-based regulation with a QoS goal of 60FPS. The interval-based FPS regulation solution was introduced at the beginning of Section 2. To meet the 60FPS target, this FPS regulation renders a frame at an interval of 16.6ms. Figure 3 gives *InMind*'s FPS under this *Int60* regulation, which missed the 60FPS target and still had FPS gap, i.e., its rendering FPS was only 55, and its encoding and decoding FPS was only 53.

These results for *Int60* were quite surprising, as when *InMind* was executed under *NoReg*, its client FPS was well above 60. Therefore, the below-60FPS of *Int60* was not because the hardware was incapable of maintaining 60FPS. Further analysis revealed that *Int60*'s low FPS and FPS gap were caused by the high variations of rendering, encoding, and transmission time. Figure 4 displays the processing time variation of *InMind*'s rendering, encoding, and transmission time. The CDF of these key steps in Figure 4a shows that about 80% - 90% of the frames' processing time is less than 16.6 ms, and about 10% - 20% could increase to well above that. These changes in processing time could be caused by the

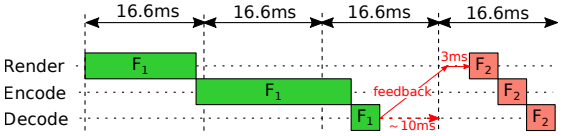
changes in frame complexity (e.g., changes in the number of objects and lighting) and the performance variation in cloud systems [30, 79]. Figure 4b gives a snapshot of *InMind*'s rendering, encoding, and transmission time for about 100 frames, which shows the frame processing time varied significantly.



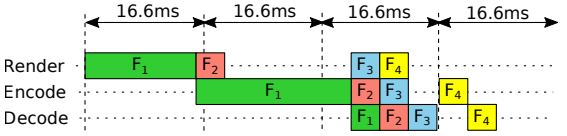
(a) Ideal pipeline with interval-based FPS regulation where 4 frames rendered, encoded, transmitted, and decoded during the span of 4 intervals.



(b) Actual pipeline with Int60 FPS regulation where 4 frames were rendered, and 3 frames were encoded, during the span of 4 intervals, due to the slow rendering and encoding of frame F_1.



(c) To reduce FPS gap, RVS60 delays frame rendering based on the time differences between frame decoding and next vblank interval.



(d) ODR synchronizes the frame processing with multi-buffering and increases encoding rate if the FPS drops below the FPS target.

Figure 5. One main cause of QoS violation and FPS gap – the sudden increase of frame processing time, and how it is handled by RVS and ODR. Copying and transmission are omitted for clarity. These figures are based on real rendering trace, but modified for better readability.

The suddenly-increased processing time leads to *Int60*'s low rendering FPS and its FPS gap. Figure 5b sketches how a frame with extremely slow rendering/encoding lowers FPS and creates FPS gap. In Figure 5b, the rendering and encoding of frame F_1 are extremely slow. Therefore, when F_1 finishes encoding, frame F_3 is already rendered, causing F_2 to be dropped. The dropping of F_2 leads to only three frames sent to the client during four intervals, causing low client FPS and FPS gap. However, in an ideal pipeline with stable processing time (shown in Figure 5a), there is no frame drop.

In short, frame processing time variation is also a main cause of FPS gap. The interval-based regulation overlooks this variation, which makes it unable to simultaneously reduce FPS gaps and satisfy the FPS requirement.

IntMax: Interval-based regulation with a QoS goal of maximizing FPS. The interval-based regulation can be adapted to maximize FPS while removing FPS gaps by letting the cloud reduce its rendering FPS to match the client's decoding FPS. Figure 3 gives the frame rates of *InMind* under this *IntMax* regulation. As Figure 3 shows, although the QoS goal was to maximize FPS, the client (decoding) FPS was only 46, which was significantly lower than the 93FPS under *NoReg*.

Further analysis showed that the low client FPS was also caused by the varying frame processing time. Figure 4b shows that *InMind*'s rendering and encoding time could suddenly increase to more than 20ms. Due to these sudden increases, there were still large FPS gaps even when the rendering FPS was initially reduced to match the client's FPS. Therefore, observing the still-existing gap, *IntMax* regulation further delayed the rendering to match the client FPS. Eventually, when the FPS gap was removed, the rendering and client FPS were too low. The fundamental issue was that *IntMax* cannot re-adjust its rendering rate when a sudden increase of processing time passes and the time returns to normal. That is, *IntMax* cannot respond fast enough to the frame-to-frame processing time variation to constantly meet FPS goals.

RVS60: Remote VSync regulation with a QoS goal of 60FPS. As described in Section 2, Remote VSync (*RVS*) [49] reduces the FPS gap by having the client compute the time difference between the end of frame decoding and the next vblank interval. This time difference is then sent to the cloud to delay rendering the next frame. Because vblank is generated by the display (based on its refreshing frequency), in this analysis, *RVS* was configured to use a client with a 60Hz refreshing frequency.

Figure 3 shows that *InMind* had only 54FPS under *RVS60*, missing the 60FPS target. The low client FPS was because *RVS* needs to collect time feedback. Figure 5c gives an example of how RVS works under a 60Hz display. In Figure 5c, *RVS* estimates that the difference between frame F_1's decoding and the next vblank is 10ms. This 10ms is then sent back to the cloud as feedback to delay the rendering of F_2. Upon receiving this 10ms, *RVS* scales it down to 3ms using a parameter *cc*. Hence, in Figure 5c, F_2 is delayed by about 3ms. Delaying F_2 ensures no frame drop and no FPS gap. However, only 2 frames are rendered within 4 intervals.

The main issue of *RVS* was the long feedback path – the sending of the time difference over the network causes the feedback of a frame to be received long after it is rendered. This slow feedback may unduly increase the rendering delay. Hence, the parameter *cc* was added as a "low-pass filter" to compensate for the slow feedback [49]. Nonetheless, because

RVS needs to collect timing feedback, the FPS of *RVS* is always lower than the refreshing rate, which was 60 in this case. Note that this behavior is expected, as *RVS* was designed to reduce FPS gaps, not meeting a QoS target.

RVSMax: Remote VSync regulation with a QoS goal of maximizing FPS. *RVS* regulates the FPS based on the display's refreshing frequency. Therefore, to maximize the client FPS, we configured *RVS* to use a client with 240Hz refreshing frequency (current high-end display). Figure 3 shows the *RVSMax* removed FPS gaps. However, its client decoding FPS was only 76, which was considerably lower than *NoReg*'s 93FPS. This low client FPS was also caused by the overhead from *RVS*'s long feedback path.

Moreover, the use of *cc* also limits *RVS*' ability to quickly respond to processing time variation. The value of *cc* was empirically determined in *RVS*, which had to be manually tuned for each hardware setup. However, *cc* is still a constant and cannot be adjusted for each frame to fit frame-to-frame processing time variation, which further reduces *RVSMax*'s FPS when removing FPS gaps.

Summary. First, the analysis of *Int60* showed that processing time variation is a main cause of FPS gap. Properly managing this variation is crucial to reduce FPS gap while ensuring satisfying QoS. Second, the analysis of *IntMax* (and *RVSMax*) showed that FPS regulation needs to respond quickly to frame-to-frame processing time variation to adjust for both processing time increases and decreases. Third, the analyses of *RVS60*/*RVSMax* showed that collecting timing feedback is too slow for cloud FPS regulation, causing low FPS.

4.2 Impact of FPS Regulation on MtP Latency

Figure 6 gives the MtP latency of *InMind* under different FPS regulations (measured by the Pictor benchmarking framework [50]), which shows that FPS regulations significantly increased MtP latency. For instance, compared to *NoReg*, the latency was increased by 24.7ms or 59% under *IntMax* and by 26.4ms or 63% under *RVS60*.

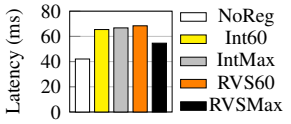
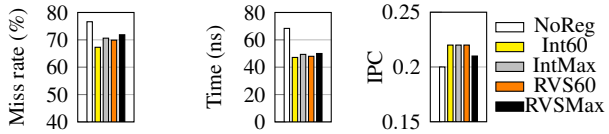


Figure 6. MtP latencies for InMind under different FPS regulations.

The MtP latency increase was caused by the delays injected by the FPS regulations. As stated previously, another cause of FPS gaps was the speed difference between different processing steps. Therefore, to reduce FPS gap, it is necessary to delay faster steps, which inevitably increases the overall processing latency. This increased latency negatively impacts user experience and makes existing FPS regulations impractical for users with slow networks or 3D applications with low latency requirements.

Note that, this observation that *RVS* increases MtP latency was different than what was reported by *RVS* [49]. To reduce latency, *RVS* combined VR headset poses to only render for the last pose. However, combined processing of pending inputs was already employed by our benchmarks by default. Therefore, there were no extra latency benefits for *RVS* from input combining.

4.3 Impact of FPS Regulation on Efficiency



(a) DRAM row buffer miss rates. (b) DRAM read access time. (c) IPC.

Figure 7. FPS regulation and DRAM efficiency.

Memory Efficiency. Excessive rendering can also degrade DRAM efficiency. Using hardware performance monitoring units (PMU), we also obtained the memory performance of *InMind*, which is reported in Figure 7.[2] Figure 7a shows that FPS regulations could reduce DRAM row buffer miss rates. For example, in Figure 7a, *Int60* reduced the miss rate by 9% over *NoReg*. This reduction in DRAM miss rate was mainly because reduced excessive rendering lead to reduced DRAM contention among application logic, frame rendering, copying, and encoding. These operations are memory-intensive, requiring reading and writing megabytes per frame.

The reduced row buffer miss rates, in turn, reduce the DRAM read access time and improve the instructions per cycle (IPC). For example, in Figure 7b, *Int60* reduced DRAM read time from *NoReg*'s 68ms to 47ms, which lead to a 10% increase in IPC for *Int60* in Figure 7c.

In summary, FPS regulation can improve DRAM efficiency as cloud 3D tends to have intensive DRAM operations. It is also worth noting that, we did not observe a major impact of FPS regulations on last-level cache, which is likely because of the streaming nature of frame processing and large frames do not always fit in caches. These results also illustrate the need for additional architecture optimizations for memory operations involved in cloud 3D.

Power and Resource Efficiency. By reducing the FPS gaps, FPS regulations can release CPU/GPU computing cycles, and thus, reduce CPU/GPU utilization and power consumption. However, due to space limitation, power usage results are reported in Section 6.

[2]The row buffer miss rates include the percentage of misses caused by both empty or conflict row buffers [78]. The DRAM read access time measures the time between a request was issued to the memory controller and the result returned to the controller, using PMU UNC_M_RPQ_OCCUPANCY and UNC_M_RPQ_INSERTS [37].

5 The Design of OnDemand Rendering

Figure 8 illustrates the architecture of ODR. ODR has three main components: 1) *multi-buffering*, reduces FPS gaps; 2) *FPS regulator*, ensures an FPS QoS target is always met; 3) *PriorityFrame*, ensures the delays injected from the other two components do not increase MtP latency. Moreover, ODR is designed to work with any 3D applications without accessing their source code. The rest of this section provides the detailed design of each component.

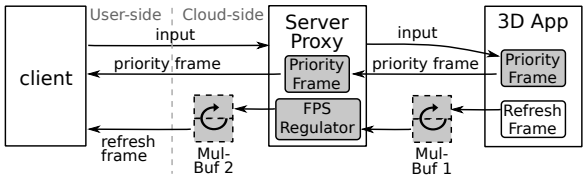


Figure 8. Overview of ODR. ODR's components are shaded.

5.1 ODR Multi-Buffering

ODR's *Multi-buffering* reduces FPS gaps by synchronizing frame processing steps and quickly responding to frame-to-frame processing time variation. Unlike existing FPS regulations, which synchronize the FPS inside the 3D application, ODR's multi-buffering synchronizes FPS through the server proxy's management (swapping). The server proxy handles frame copying and encoding and connects to the 3D application and network. Hence, the server proxy knows the time of most frame processing steps and is a better place to synchronize FPS across different steps to remove FPS gaps.

ODR has two multi-buffers: one between the 3D application and the server proxy ("Mul-Buf1" in Figure 8), and one between the server proxy and network ("Mul-Buf2"). The first multi-buffer ("Mul-Buf1") synchronizes the rendering (in 3D application) and encoding (in the server proxy) frame rates through the following procedure. Mul-Buf1 has two buffers – a front and a back buffer. The front buffer stores the current frame, which will be encoded by the server proxy. The back buffer is used to store the next (newer) frame that is being rendered by the 3D application. When the server proxy finishes encoding the frame from the front buffer, it swaps the front and back buffers. In other words, the server proxy treats the old back buffer as the new front buffer to encode the newer frame. The synchronization between the rendering and encoding happens at the buffer swapping. The server proxy only swaps the buffers after it finishes encoding, and after the back buffer has a new frame. If the back buffer is empty, the server proxy pauses swapping to wait for it to be populated. Similarly, the 3D application pauses its rendering until the buffers are swapped (i.e., an empty back buffer is available). By pausing themselves based on the buffer swapping event, the server proxy and 3D application naturally synchronize their frame rates – the faster one will pause itself to wait for the slower one to clear or populate the buffers.

Figure 5d gives an example of how ODR's multi-buffer synchronizes the rendering and encoding. In Figure 5d, after frames F_1 and F_2 are rendered, F_1 occupies the front buffer, and F_2 occupies the back buffer. The buffer swap does not happen until F_1 is encoded. Therefore, the 3D application delays rendering F_3 until F_1 is encoded and no obsolete frames are generated and dropped. Hence, the 3D application naturally delays or accelerates the rendering in synchronous with the server proxy through the multi-buffer.

The second multi-buffer ("Mul-Buf2") works similarly to sync under varying encoding/network time. The server proxy stores the encoded frame in the back buffers, while the network sends the encoded frame in the front buffer to the client. Both the server proxy and network pause themselves to wait for the slower one to clear or populate the buffers.

5.2 ODR FPS Regulator

Design. ODR's FPS regulator is designed to ensure that the encoding FPS in the server proxy matches an FPS target. With the help of multi-buffering, once the encoding FPS matches the target, the rendering FPS and transmission FPS are naturally regulated to match the FPS target as well.

To meet the FPS target under suddenly-increased processing time, ODR's FPS regulator accelerates the frame processing if it detects a processing time increase. This design is different from existing FPS regulation solutions that only delay the rendering. When ODR's FPS regulator detects that the frame encoding rate is below the FPS target, it increases the encoding FPS to ensure that the FPS target is still met. The increased encoding FPS also increases the FPS of rendering and network transmission with the help of multi-buffering, allowing the whole 3D system to accelerate to meet the FPS target. For example, in Figure 5d, when ODR detects that only one frame (F_1) is encoded within the first two intervals, it accelerates the encoding to output two more frames (F_2 and F_3) back to back. The faster encoding also accelerates the rendering through the multi-buffer. Hence, three frames are rendered/encoded throughout three intervals, meeting the FPS target. For frame F_4, as the FPS target has been met, its encoding is delayed to the beginning of the fourth interval to avoid the FPS exceeding the target.

Algorithm 1 gives the algorithm of ODR's FPS regulator that delays or accelerates encoding to meet an FPS target. At the beginning of execution, ODR computes the expected encoding interval length based on the FPS target (line2), similar to the interval-based FPS regulation. The variable, *acc_delay*, declared on line 3 is the key to aid ODR to delay or accelerate encoding. *acc_delay* stores the time that needs to be delayed to match the FPS target. The value of *acc_delay* is accumulated based on past frames' encoding time. More specifically, during execution, ODR encodes a frame from the front buffer of Mul-Buf1 (line6) and stores the encoded

Algorithm 1: ODR FPS Regulator

```
   Input: targetFPS;                                  // the FPS target
1  Function main(targetFPS):
2      interval ← 1000msec / targetFPS;                          // in msec
3      acc_delay ← 0;                  // accumulated delay length
4      while app is running do
           // encode frame from Mul-Buf1
5          start_time ← getCurrentTime();
6          f ← encode_frame_in_Mul-Buf1();
           // saved encoded frame to Mul-Buf2
7          wait_to_swap_Mul-Buf2();
8          store_frame_to_Mul-Buf2(f);
9          end_time ← getCurrentTime();
           // compute the accumulated delay
10         encode_time ← end_time − start_time;
11         time_diff ← interval − encode_time;
12         acc_delay ← acc_delay + time_diff;
           // sleep only if acc_delay > 0, otherwise
              immediately move on to encode next frame to
              meet QoS
13         if acc_delay > 0 then
14             sleep(acc_delay);
15             acc_delay ← 0;
16         end
           // swap Mul-Buf1
17         wait_for_Mul-Buf1_back_buf_full();
18         swap_Mul-Buf1();
19     end
20     return
```

frame in the back buffer of Mul-Buf2 (line8). The encoding time for this frame is computed at line 10. Then the difference ($time\_diff$) between this encoding time and the expected encoding interval is computed (line11), which accumulates to *acc_delay* (line12). If $time\_diff$ is positive, then the encoding time is shorter than the expected interval. If $time\_diff$ is negative, then the encoding time is longer than the expected interval. Similarly, if *acc_delay* is positive, then the encoding rate of the past frames is higher than FPS target, and the encoding should slow down (line 13 to 16). However, if *acc_delay* is negative, then the current encoding FPS is lower than the target. In that case, the encoding should continue without delay until the FPS target is met and the *acc_delay* returns to positive. At last, ODR swaps Mul-Buf1 to process the next frame (line17-18).

Notes on FPS Regulation Goal. ODR's FPS regulator does not aim at guaranteeing frames arriving at the client at regular intervals. Indeed, the varying rendering, encoding, and network time make this guarantee impossible in cloud 3D. Instead, ODR aims at ensuring the FPS target is met for each small period (e.g., 200ms).

This design choice is based on three reasons. First, as corroborated by user study in Section 6.7, accelerating rendering to generate enough frames at targeted rates improves user experience and reduces stutters/lags in cloud 3D. The sudden delays in encoding/network may affect several consecutive frames. Accelerating rendering prevents these delays from further degrading FPS, and thus, reduces stutters/lags. Second, as also corroborated by our user study, as long as the client FPS meets the target for every small period, it can still provide a satisfying user experience. In our evaluation, ODR could ensure 30 or 60FPS for every 200ms interval at least. Third, generating enough frames at targeted rates in the cloud may further improve user experience by allowing more client-side optimizations. For example, high frequency (90-240hz) displays with FreeSync/GSync are designed to reduce lag by allowing frames to arrive at high but varying rates [5, 64]. We will explore client optimizations in the future.

5.3 ODR PriorityFrame

Design. PriorityFrame is used to reduce the MtP latency of ODR by prioritizing the processing of the frames that are generated based on user inputs. PriorityFrame is designed based on the observation that the majority of the frames rendered by an interactive 3D application are due to the application's internal updates/refreshes instead of user inputs. A normal user typically only produces fewer than 250 actions/inputs per minute (APM) [77]. Even professional game players usually have an APM of only 300. That is, on average, there are usually fewer than 5 inputs per second, and thus, no more than 5 input-generated frames per second. As there are only a small number of input-generated frames, it is possible to prioritize these frames to reduce the MtP latency.

PriorityFrame has two parts (Figure 8). One part is located inside the 3D application which detects if there is user input. Once detected, the rendering delay (i.e., the buffer swapping wait) in the 3D application is canceled, allowing the input-generated frame to be rendered immediately. After rendering, the frame is sent to the PriorityFrame inside the server proxy for encoding and network transmission without any delay. To ensure a correct frame sequence, any unsent frames rendered before the input-generated frame become obsolete frames and are dropped. This frame dropping will not significantly increase the FPS gaps (as shown in Section 6.2), because there are usually fewer than 5 priority frames per second, and not every priority frame causes frame drop. Note that, PriorityFrame is only engaged for frames generated based on user inputs, as MtP latency is only applicable to those frames.

Impact of Position/Posture Polling. Mouse and VR headsets can poll cursor position or user posture at high frequency, e.g., VR headsets may poll at 1000hz [67]. A large number of polling events (i.e., user inputs) may be generated due to this frequent polling. However, when rendering, multiple pending polling events are typically combined so that only the most recent position/posture is used. This combination is employed by all our open-source benchmarks at least. With this combination, users always see the frame with the last position and perceive low latency [49]. That is, this combination already ensures low latency for polling events and also eliminates the need to render for every polling event. Therefore, ODR does

Benchmark	Desicription
SuperTuxKart (STK)	Racing Game
0 A.D. (0AD)	Real-time Strategy Game
Red Eclipse (RE)	First-person Shoot Game
DoTA2 (D2)	Battele Arena Game
InMind (IM)	VR Game
IMHOTEP (ITP)	Health Training VR

Table 1. Cloud 3D benchmarks used in this evaluation.

not prioritize polling event inputs. After excluding the polling events, there were 2 to 5 (average 3.6) priority frames per second observed in our evaluation.

5.4 ODR Implementation

We implemented ODR on Linux for 3D applications built with OpenGL and X Window. The main implementation difficulty is to handle proprietary and closed-source computer games and VR applications, as the FPS regulator and PriorityFrame require adding delays to the rendering operations and reading user inputs inside 3D applications.

To meet these two requirements without using 3D application's source code, we employed API hooks. More specifically, ODR intercepts the *glXSwapBuffers* [74] API from OpenGL to allow delaying the rendering when the Mul-Buf1 is yet-swapped. *glXSwapBuffers* is called at the end of every frame rendering to allow the GPU to finish processing 3D commands. Therefore, the delay of rendering can be inserted directly after *glXSwapBuffers*. The PriorityFrame inside the application requires detecting user inputs, which is achieved by intercepting the *XNextEvent* [74] API from X Window. If the intercepted *XNextEvent* returns a user input, then PriorityFrame cancels the rendering delay.

Additionally, our system and all comparison experiments are implemented on a state-of-the-art open-source cloud 3D system, TurboVNC with VirtualGL [16, 17]. Following the common practice, we modified the TurboVNC to use video streaming to transmit rendered frames [31, 75]. We also tuned the "low-pass filter" parameters for *RVS* for each benchamrk and configuration to optimize *RVS*' performance.

6 Experimental Evaluation

6.1 Evaluation Setup

Benchmarks. We used all six benchmarks from the Pictor benchmark suite [50], which is designed to evaluate cloud 3D systems. As listed in Table 1, these benchmarks include gaming and VR applications from different genres. Each benchmark was executed with two resolutions at 1280×720 (720p) and 1920×1080 (1080p).

Platforms. We evaluated ODR using two platforms, our private cloud and Google cloud with Compute Engine (GCE), which represent edge and public cloud deployments. In the private cloud, the benchmarks were executed on a server with an Intel i7-7820x CPU and NVIDIA GTX 1080Ti GPU. The server and client were connected using 1Gbps network. In GCE, a VM of type *n1-highcpu-16* was used, which had a 16-core Intel Xeon CPU and NVIDIA Tesla P4 GPU. The GCE VM was allocated in region *us-central1*, which was well beyond 500 miles from our lab. The server-to-client ping latency for our private cloud is about 2ms and for GCE is about 25ms. No specialized network devices/links were used. We did not alter or control network connections in our experiments.

The client for all our experiments was a desktop computer with an Intel i5-7400 CPU. The servers (in private cloud and GCE) and client had Ubuntu 16.04 as the OS.

Configurations. We considered two QoS goals, including maximizing FPS and maintaining stable 60FPS (for 720p) or 30FPS (for 1080p). We evaluated no regulation and three FPS regulations, including ODR, interval-based regulation (*Int*) and Remote VSync (*RVS*). As there are 2 resolutions, 2 cloud platforms, 2 QoS goals, 3 regulations, and no regulation, each benchmark was executed with 28 configurations.

When reporting results, no regulation configuration is labeled as **NoReg**. ODR, Int, and RVS configuration results are labeled as **ODRMax**, **IntMax**, **RVSMax**, when the QoS goal was maximizing FPS. They are labeled as **ODR30**, **Int30**, and **RVS30** when the QoS goal was 30FPS. They are labeled as **ODR60**, **Int60**, and **RVS60**, when the goal was 60FPS. *Due to space limitation, detailed results for the 1080p private cloud were omitted.* These results were consistent with other configurations, and their averages are reported.

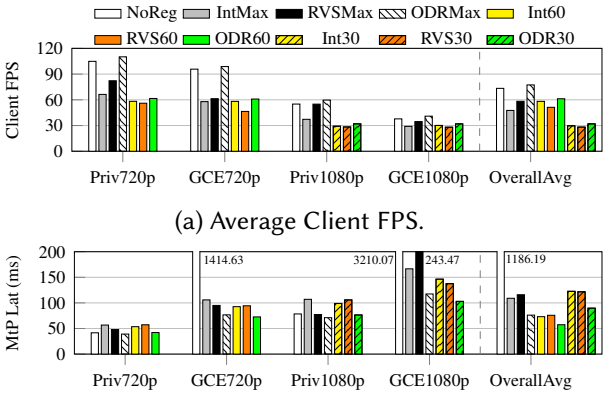
	720p Priv Cloud	720p GCE	1080p GCE
NoReg	60.7/218.8 (ITP)	154.7/671.0 (ITP)	140.6/465.6 (ITP)
IntMax	0.4/0.9 (IM)	0.4/0.8 (IM)	0.3/0.7 (D2)
RVSMax	0.1/0.3 (IM)	0.7/1.1 (IM)	0.1/0.13 (D2)
ODRMax-noPri	0.1/0.4 (IM)	0.6/1.1 (IM)	0.5/1.1 (0AD)
ODRMax	1.3/3.2 (IM)	1.8/3.3 (IM)	2.7/3.7 (RE)
Int60 or Int30	0.0/0.07 (IM)	0.0/0.01 (IM)	0.1/0.5 (IM)
RVS60 or RVS30	0.0/0.04 (0AD)	0.0/0.05 (IM)	0.0/0.02 (ITP)
ODR60 or ODR30	2.6/3.9 (IM)	2.8/3.7(IM)	2.9/4.2 (RE)

Table 2. The Average/Max FPS gaps for each configuration. The benchmark in the parentheses was the one having the largest gap for this each configuration.

6.2 Effectiveness of FPS Gap Reduction

FPS Gaps of ODR. Table 2 reports the average and maximum FPS gaps for ODR under each configuration. As Table 2 shows, ODR could effectively reduce the FPS gap. For example, for the 1080p GCE evaluation, *ODRMax* reduced the average FPS gap from *NoReg*'s 140.6 frames to only 2.7 frames. The same large reductions of FPS gaps by ODR can be observed in every configuration in Table 2. Even the largest FPS gap for ODR was only 4.2 frames.

Furthermore, to evaluate the impact of PriorityFrame, we also measured ODR's FPS gaps without PriorityFrame, which are reported in the *ODRMax-noPri* row in Table 2. Table 2 shows that the difference between the average FPS gaps of



(a) Average Client FPS.

(b) Average MtP Latency.

Figure 9. Average QoS results over six benchmarks for all 28 configurations, as well as overall averages.

ODRMax-noPri and *ODRMax* was only 1.2 frames, indicating that PriorityFrame does not significantly increase FPS gaps. Table 2 also shows that *ODRMax-noPri*'s average FPS gap was always below one frame, suggesting ODR's multi-buffering nearly eliminates FPS gap.

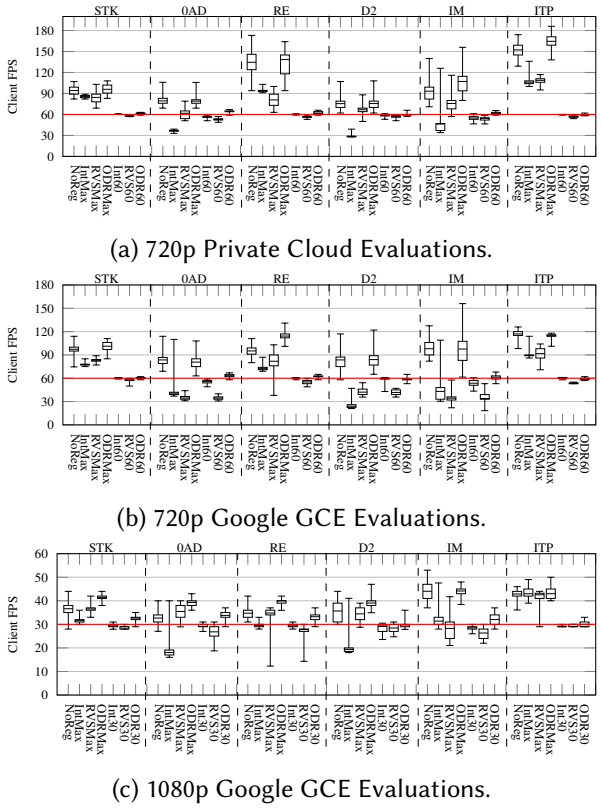
Comparison with Int/RVS. Table 2 also shows that both *Int* and *RVS* regulations could effectively reduce FPS gaps. However, as shown later with the FPS and latency results, their reduction came with a price of low QoS.

6.3 QoS Evaluation Results – Client FPS

Average Client FPS of ODR. Figure 9a gives the average client FPS under each configuration. As Figure 9a shows, when the QoS goal was to maximize the client FPS, *ODRMax*'s average client FPS was consistently higher than *NoReg*. For example, in 720p private cloud evaluation (i.e, "Priv720p" in Figure 9a), *ODRMax*'s average client FPS was 110, which was 6% higher than *NoReg*'s 104FPS. We observed that this increase in FPS is due to the reduced memory contention, as discussed in Section 4. Along with the FPS gaps reported previously, these results suggested *ODRMax* could simultaneously reduce FPS gap and meet the goal of maximizing FPS (by increasing client FPS over *NoReg*).

When the QoS goal was 30 or 60FPS, Figure 9a also shows that *ODR30* or *ODR60* had an average client FPS slightly higher than 30 and 60, meeting the QoS goal. For example, in 720p private cloud evaluation (i.e, "Priv720p" in Figure 9a), *ODR60*'s average client FPS was 61.6, meeting the 60FPS target. The slightly higher FPS was mainly because of the occasional priority frames.

Comparison with Int/RVS. Figure 9a shows that both *IntMax* and *RVSMax* usually had lower client FPS than *ODRMax*. For example, in the 720p private cloud evaluation (i.e, "Priv720p" in Figure 9a), the average FPS of *IntMax* and *RVSMax* was 66.3 and 82.2, which were both significantly lower than *ODRMax*'s 110.



(a) 720p Private Cloud Evaluations.

(b) 720p Google GCE Evaluations.

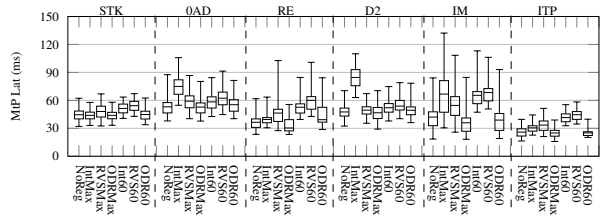
(c) 1080p Google GCE Evaluations.

Figure 10. Detailed client FPS results. Box plots shows the 1%ile, 25%ile, mean, 75%ile, and 99%ile.

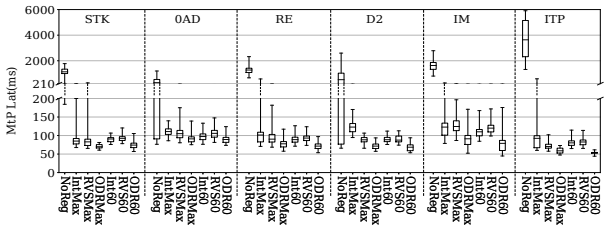
Moreover, when the QoS goal was meeting 30/60FPS targets, both *Int* and *RVS* usually could not meet these targets. For example, in the 720p private cloud evaluation (i.e, "Priv720p" in Figure 9a), *Int60*'s average client FPS was only 58, and *RVS60*'s average client FPS was only 56, which were both lower than 60. These results suggested that neither *Int* nor *RVS* regulations could meet the QoS goals of maximizing FPS or 30/60FPS (as analyzed in Section 4).

Individual benchmark results and tail (1%ile) FPS. Figure 10 gives the detailed client FPS results for each benchmark with the tail FPS. These detailed results are consistent with the averages reported in Figure 9a. That is, Figure 10 show that, when the QoS goal was to maximize FPS, *ODRMax* had the same or higher client FPS than *NoReg* for nearly all benchmarks. For example, for *InMind* in the 720p private cloud evaluation (i.e., "IM" in Figure 10a), *ODRMax* increased the average FPS from *NoReg*'s 93 to 107.

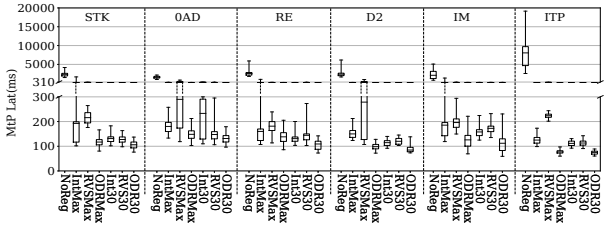
Figure 10 also shows that *ODRMax* had similar or better tail (1%ile) FPS than *NoReg* in most cases. It is also interesting that the FPS of both *NoReg* and *ODRMax* had similar fluctuation ranges in Figure 10a, suggesting that a main cause of the fluctuations of *ODRMax* 's FPS was the fluctuation inside the 3D benchmarks (i.e., rendering time).



(a) 720p Private Cloud Evaluations.



(b) 720p Google GCE Evaluations.



(c) 1080p Google GCE Evaluations.

Figure 11. Detailed MtP Latency results. Box plots shows the 1%ile, 25%ile, mean, 75%ile, and 99%ile.

Figure 10 also shows that, in most cases, *ODR30* or *ODR60* could ensure their average FPS was consistently above 30 or 60, satisfying the QoS goal of 30/60FPS. The only two exceptions were "D2" and "ITP" in Figure 10b, where *ODR60*'s average client FPS was 59, only 1 frame fewer than the 60FPS target. Figure 10 also shows that the tail (1%ile) FPS of *ODR30* or *ODR60* was also usually very close to the 30/60FPS targets.

For *IntMax* and *RVSMax*, Figure 10 also shows that their client FPS was lower than *ODRMax* for all benchmarks. For *Int30/60* and *RVS30/60*, Figure 10 also shows that their client FPS usually missed the 30/60FPS targets. Moreover, *Int* and *RVS* regulations also usually had much lower tail FPS than ODR. These results are consistent with the averages from Figure 9a, showing that *Int* and *RVS* regulation had difficulties meeting the QoS goals.

6.4 QoS Evaluation Results – MtP Latency

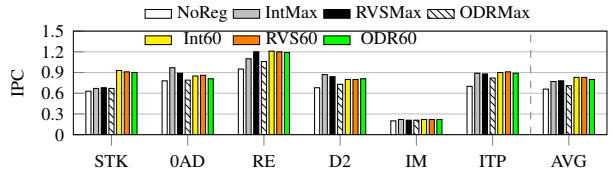
Average MtP latency of ODR. Figure 9b gives the average MtP latency of each configuration. As Figure 9b shows, when the QoS goal was to maximize FPS, *ODRMax* always had lower average latency than *NoReg*. For example, for the 720p private cloud evaluation ("Priv720p") in Figure 9b, *ODRMax*'s average latency was 39.1ms, which was 6% lower than *NoReg*'s 41.6ms. Similarly, *ODRMax*'s average latency was 9% lower than *NoReg* for "Priv1080p" in Figure 9b. ODR's lower latency was because of PriorityFrame. In *NoReg*, input-generated frames must wait for the previous frames to be processed, which introduces queuing delay. However, with ODR's PriorityFrame, input-generated frames are prioritized over other frames, removing this queuing delay and causing ODR to have lower latency than *NoReg*.

For 720p and 1080p GCE evaluations, Figure 9b surprisingly shows that the average MtP latency for *NoReg* was extremely high (up to 3.2 seconds). Further analysis showed that this high latency was due to the combined impact of FPS gap and slow network. The large FPS gap under *NoReg* caused network congestion, which significantly increased network latency, leading to high latency. By removing the FPS gap, *ODRMax* did not have this network congestion issue. Therefore, *ODRMax*'s average latency was more than 95% faster than no-regulation. Moreover, when the QoS goal was 30/60FPS, Figure 9b shows that *ODR30* or *ODR60* also had lower average latency than *NoReg*.

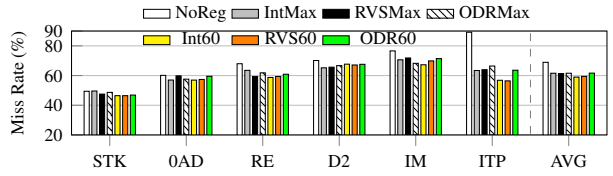
More important, Figure 9b shows that the average latency of *ODRMax* and *ODR30* was less than 77ms in the 720p GCE evaluation, which met the most stringent 100ms latency requirement for computer games [14]. The average latency of *ODRMax* and *ODR30* was also less than 120ms in the 1080p GCE evaluation. This low latency is important as it shows that ODR makes it feasible for 3D applications to be deployed to conventional public clouds without specialized hardware. The user study in Section 6.7 also corroborates the feasibility of public cloud deployments using ODR.

Comparison with Int/RVS. Figure 9b shows that the average MtP latency of *Int* and *RVS* regulations were always higher than ODR. For example, in the 720p private cloud evaluation (i.e, "Priv720p" in Figure 9b), the average latency of *IntMax* and *RVSMax* was 56.7ms and 47.9ms, higher than *ODRMax*'s 39.1ms. And the average latency of *Int60* and *RVS60* was 53.5ms and 57.3ms, higher than *ODR60*'s 42.1ms. As discussed in Section 4, the delays injected by the *Int* and *RVS* regulations increase the latency, whereas ODR's PriorityFrame avoids the penalty of these delays.

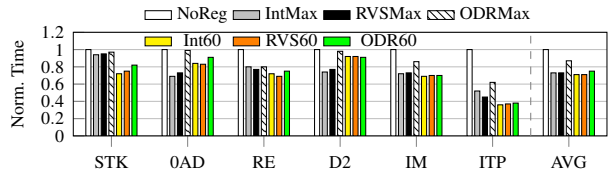
Individual benchmark results and tail (99%ile) Latency. Figure 11 gives the detailed MtP latency results for each benchmark with tail latency. These detailed results are consistent with the averages in Figure 9b. That is, ODR's average and tail latency were better than *NoReg*, *Int*, and *RVS* for most configurations. Moreover, for all benchmarks, ODR's latency was lower than 92ms in the 720p GCE evaluation and lower than 150ms in the 1080p GCE evaluation. Particularly, *ODRMax*'s and *ODR30*'s latency was only 77.1ms and 74.2ms for *IMHOTEP* in the 1080p GCE evaluation ("ITP" in Figure 11c). This low latency in GCE further shows that ODR enables cloud 3D with conventional public clouds.



(a) Instructions per Cycle (IPC).



(b) DRAM row buffer miss rates.



(c) DRAM read access time. Normalized to *NoReg* for legibility.

Figure 12. Memory efficiency (720p private cloud).

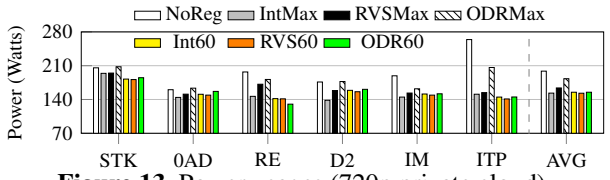


Figure 13. Power usages (720p private cloud).

6.5 System Efficiency Evaluation Results

Figure 12 provides the memory efficiency results, and Figure 13 provides the power usage results, for the 720p private cloud evaluation. Power usages were measured using a Klein Tools CL110 meter. This meter has a current sensor that reads current in ampere from wall outlets. The current readings were then converted to wattage by multiplying the household voltage. Note that, due to the lack of physical access to GCE cloud servers, memory, and energy efficiency results were not available for GCE evaluations.

Memory Efficiency of ODR. When the QoS goal was maximizing FPS, Figure 12a shows that *ODRMax* improved the IPC for each benchmark over *NoReg*. On average, *ODRMax* improved the IPC from *NoReg*'s 0.66 to 0.71 (by 7.6%). The improved IPC was mainly due to the improved DRAM performance. Figure 12b shows that *ODRMax* improved the average DRAM row buffer miss rates by 10.7% over *NoReg*. Figure 12c also shows that *ODRMax* reduced the average DRAM access time by 13% over *NoReg*. The reason is that the frame rendering, copying, and encoding operations are all pipelined (for better performance) and executed in their own threads/processes. Hence, frequent rendering will increase the probability that these tasks execute simultaneously. Simultaneous execution leads to simultaneous DRAM access and thus DRAM row buffer contention, and in turn, leads to slower memory operations and lower IPC.

Similarly, when the QoS goal was 60FPS, Figure 12 shows that, on average, *ODR60* improved IPC by 21.2%, reduced DRAM row buffer miss rates by 10.5%, and reduced DRAM access time by 25%, over *NoReg*. With higher QoS, These results illustrate ODR's improvement on DRAM efficiency and the need for additional architecture optimizations for memory operations involved in cloud 3D.

Power Consumption of ODR. When the QoS goal was maximizing FPS, Figure 13 shows that *ODRMax* reduced the average power usage from *NoReg*'s 198.7 watts to 183.1 watts or by 7.9%. The highest power reduction of *ODRMax* was observed with *IMHOTEP*, where the power usage from 264.1 watts to 206.3 watts, or by 22%. This power usage reduction was because of the reduced excessive rendering.

It is also worth noting that *ODRMax*'s power usage reduction was achieved with an increase in client FPS (as shown in Figure 9a), indicating that the power reduction was not only from the reduced FPS gap, but also from the improved resource efficiency – *ODRMax*'s higher memory resource efficiency allows better QoS with lower power usages.

When the QoS goal was 60FPS, Figure 13 shows that *ODR60* reduced average power usage by 22% (from 198.7 watts to 155.1 watts), with a maximum reduction of 45% for *IMHOTEP* (from 264.1 watts to 145.2 watts), over *NoReg*.

Comparison with Int/RVS. Figure 12 and Figure 13 show that the *Int* and *RVS* regulations had similar or lower memory contention and power usages than ODR. However, the lower resource and energy results do not mean that these two regulations were more efficient. Rather, it was the direct result of their lower client FPS and QoS.

6.6 Evaluation Summary for ODR

To facilitate the understanding of the large amount of data in our evaluation, Figure 12, and Figure 13 also provide overall average performance results for all configurations, which are summarized in the following paragraphs.

Overall Average - FPS Gap. As Table 2 shows, for all benchmarks and all 28 configurations, the overall average FPS gap of ODR was only 2.6 frames with a maximum gap of 4.2 frames. For *NoReg*, the average FPS gap was at least 60.7 frames, which was significantly worse than ODR. *Int* and *RVS* could also reduce the FPS gap, however, they had a lower QoS than ODR.

Overall Average - Client FPS. Figure 9a gives the overall average FPS of all benchmarks and configurations. Figure 9a shows that when the QoS goal was maximizing FPS, *ODRMax*'s overall average client FPS was 77.4, which was

5.5% higher than *NoReg*, 62.5% higher than *IntMax*, and 32.8% higher than *RVSMax*. When the QoS goal was 30 or 60FPS, ODR's average FPS was 31.9 or 61.2, meeting the 30/60FPS targets. When considering all Max/60/30 configurations, ODR increased the average client FPS by 62.4% and 34.7% over *Int* and *RVS*, respectively.

Overall Average - MtP Latency. Figure 9b gives the overall average MtP latency for all benchmarks and configurations. Figure 9b shows that the overall average latency of *ODRMax* was 76.1ms, which was 93.6% faster than *NoReg*, 30.2% faster than *IntMax*, and 34.4% faster than *RVSMax*. The average latency of *ODR60* was 57.4ms, which was 95.2%, 21.5%, and 24.3% faster than *NoReg*, *Int60*, and *RVS60*, respectively. The average latency of *ODR30* was 89.7ms, which was 92.4%, 26.8%, and 26.3% faster than *NoReg*, *Int30*, and *RVS30*, respectively. When considering all Max/60/30 configurations, ODR reduced the MtP latency by 30.7% and 27.3% over *Int* and *RVS*, respectively.

Overall Average - Resource Efficiency. Figure 12 shows that for all 720p evaluations in our private cloud, ODR improved the overall average IPC by 14.4% over *NoReg*. ODR also reduced DRAM row buffer miss rate by 11% and reduced DRAM access time by 19%. Figure 13 shows that ODR reduced the overall average power usage by 16.0% over *NoReg*. Moreover, *Int* and *RVS* had slightly lower resource usage than ODR. However, this lower usage does not mean that *Int* and *RVS* were more efficient. Rather, it was the direct result of their lower client FPS and QoS.

Moreover, although network bandwidth was not a performance limiting factor in our evaluation, we also measured the network bandwidth usages of ODR, which ranged from 15 Mbps to 60Mbps depending on the benchmarks and configurations. This low bandwidth usage is primarily because the frames were encoded as video streams.

6.7 User Experience Study

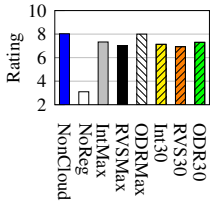


Figure 14. Average user ratings.

Setup. Besides FPS and latency, other factors may also affect user experience, such as frame stuttering and screen tearing. Therefore, we also conducted a user study (IRB-approved) with 30 participants to evaluate user experience. Each participant played with a randomly-picked benchmark with same amount of time with no other restrictions, under *NoReg* and all three regulations in GCE, along with a local execution (i.e., NonCloud). All benchmarks were executed with a resolution of 1920×1080. The client had an ordinary 60hz display.

Results. Figure 14 gives overall user ratings of each configuration. Here, the participants were asked to rate each configuration using a scale of 1 to 10, with 10 being the best.

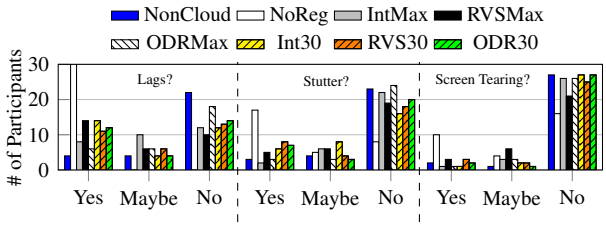


Figure 15. The number of participants responding to lagging, stuttering, and screen tearing in each setting.

Figure 14 shows that *ODRMax* had a rating of 8.0, which was comparable to the 8.03 of the local (NonCloud) execution. Interestingly, when asked which configuration was better, exact half (i.e., 15) of the participants thought *ODRMax* was better than the local execution. Figure 14 also shows that *NoReg*, *IntMax*, and *RVSMax* had lower ratings than *ODRMax*, suggesting ODR provided better user experience. Most users also rated *NoReg* with a score of only 3.1, showing no FPS regulation is unacceptable for cloud 3D.

Figure 15 reported the numbers of participants who had experienced lags, stutters, and tearing in each setting. For *ODRMax*, 24 and 26 participants reported no stutters and no tearing, which were similar to NonCloud. However, 18 participants reported no lags under *ODRMax*, fewer than the 22 of NonCloud. These results also showed that *ODRMax* had a comparable (or slightly worse) user experience than NonCloud. Furthermore, fewer participants reported no lags, stutters and tearing in *IntMax* and *RVSMax* than *ODRMax*, indicating ODR had better user experience. Particularly, considerably more participants reported no lags in *ODRMax* than *IntMax* and *RVSMax*, suggesting PriorityFrame does reduce the MtP latency.

Figure 14 and Figure 15 also provide the user study results when the QoS goal was 30FPS. As expected, the user experience was lower under 30FPS than the case of maximizing FPS. Nonetheless, compared to *Int30* and *RVS30*, *ODR30* still achieved higher user ratings in Figure 14 and had fewer participants reporting lags/stutters/tearing in Figure 15.

This user study also corroborates the notes on FPS regulation goals in Section 5.2. The fact that more users reporting no stutters/lags in *ODR30* (than *Int30* and *RVS30*), along with *ODR30*'s better user rating, shows that accelerating to render at the targeted rate can improve user experience even if the frames do not arrive at the client at regular intervals. These results also suggest that ensuring FPS target is met for each small period can provide a good user experience in ODR.

7 Conclusion

To reduce excessive rendering, regulation of frame rates (FPS) is crucial for cloud 3D system's energy and resource efficiency. Here, we presented a novel FPS regulation solution, ODR, to reduce excessive rendering. Evaluation results showed that ODR significantly reduced FPS gaps, improved

memory/energy efficiency, and ensured QoS, enabling deploying 3D applications to conventional public clouds.

Acknowledgment

This work was partially supported by the National Science Foundation grants, 2221843, 2155096, 2215359, 2215193, and 2007718. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of NSF. The authors would like to thank the anonymous reviewers for their insightful comments.

References

[1] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. 2017. Enabling High-Quality Untethered Virtual Reality. In *USENIX Symp. on Networked Systems Design and Implementation*.

[2] Yoshihisa Abe, Roxana Geambasu, Kaustubh Joshi, H. Andrés Lagar-Cavilla, and Mahadev Satyanarayanan. 2013. VTube: Efficient Streaming of Virtual Appliances over Last-Mile Networks. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*.

[3] Seyed Javad Seyed Aboutorabi and Mohammad Hossein Rezvani. 2020. An Optimized Meta-heuristic Bees Algorithm for Players' Frame Rate Allocation Problem in Cloud Gaming Environments. *The Computer Games Journal* (2020), 1–24.

[4] Gae Ae Ryu and Kwan-Hee Yoo. 2020. Key Factors for Reducing Motion Sickness in 360 Virtual Reality Scene: Extended Abstract. In *The 25th International Conference on 3D Web Technology*.

[5] AMD. [n. d.]. AMD FreeSync Technology. https://www.amd.com/en/technologies/free-sync. [Online; accessed 11-Dec-2020].

[6] E. Badry, K. Inoue, and M. S. Sayed. 2020. Decision Tree Models and Early Splitting Termination in Screen Content Extension of High Efficiency Video Coding. *IEEE Access* (2020).

[7] T. Baruah, Y. Sun, S. Dong, D. Kaeli, and N. Rubin. 2018. Airavat: Improving energy efficiency of heterogeneous applications. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*.

[8] Marc Carrascosa and Boris Bellalta. 2020. Cloud-gaming: Analysis of Google Stadia traffic. arXiv:2009.09786 [cs.NI]

[9] Yu-Hsin Chen and Vivienne Sze. 2015. A Deeply Pipelined CABAC Decoder for HEVC Supporting Level 6.2 High-Tier Applications. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 5 (2015), 856–868. https://doi.org/10.1109/TCSVT.2014.2363748

[10] H. J. Chi, Y. H. Choi, S. M. Lee, J. Y. Sim, H. J. Park, J. J. Lim, P. S. Kang, B. Y. Lee, J. C. Hong, and H. S. Lee. 2011. A 2-Gb/s Intrapanel Interface for TFT-LCD With a VSYNC-Embedded Subpixel Clock and a Cascaded Deskew and Multiphase DLL. *IEEE Transactions on Circuits and Systems II: Express Briefs* 58, 10 (2011), 687–691.

[11] S. Choy, B. Wong, G. Simon, and C. Rosenberg. 2012. The Brewing Storm in Cloud Gaming: A Measurement Study on Cloud to End-user Latency. In *Annual Workshop on Network and Systems Support for Games*.

[12] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. 2014. A Hybrid Edge-cloud Architecture for Reducing On-demand Gaming Latency. *Multimedia Systems* 20, 5 (2014), 503–519.

[13] Kajal T Claypool and Mark Claypool. 2007. On Frame Rate and Player Performance in First Person Shooter Games. *Multimedia Systems* 13, 1 (2007), 3–17.

[14] Mark Claypool and Kajal Claypool. 2006. Latency and Player Actions in Online Games. *Commun. ACM* 49, 11 (Nov. 2006), 40–45.

[15] Mark Claypool and Kajal Claypool. 2010. Latency Can Kill: Precision and Deadline in Online Games. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*.

[16] DR Commander. 2007. VirtualGL: 3D without boundaries–the VirtualGL project.

[17] D. R. Commander. 2009. User's guide for TurboVNC 0.6.

[18] Landon P. Cox and Lixiang Ao. 2020. LevelUp: A Thin-cloud Approach to Game Livestreaming. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. 246–256. https://doi.org/10.1109/SEC50012.2020.00037

[19] Eduardo Cuervo, Alec Wolman, Landon P. Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. 2015. Kahawai: High-Quality Mobile Gaming Using GPU Offload. In *Proc. of Int'l Conf. on Mobile Systems, Applications, and Services*.

[20] Yunhua Deng, Yusen Li, Xueyan Tang, and Wentong Cai. 2016. Server Allocation for Multiplayer Cloud Gaming. In *Proc. of ACM Int'l Conf. on Multimedia*.

[21] R. Ellerweg. 2018. Make Frame Rate Studies Useful for System Designers. In *Int'l Conf. on Graphics and Interaction*.

[22] Domenic Forte and Ankur Srivastava. 2013. Energy- and Thermal-Aware Video Coding via Encoder/Decoder Workload Balancing. *ACM Trans. Embed. Comput. Syst.* 12, 2s, Article 96 (May 2013).

[23] J. Gaur, R. Srinivasan, S. Subramoney, and M. Chaudhuri. 2013. Efficient Management of Last-level Caches in Graphics Processors for 3D Scene Rendering Workloads. In *IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*.

[24] Dan Ginsburg, Budirijanto Purnomo, Dave Shreiner, and Aaftab Munshi. 2014. *OpenGL ES 3.0 Programming Guide*. Addison-Wesley Professional.

[25] Ashvin Goel, Charles Krasic, and Jonathan Walpole. 2008. Low-Latency Adaptive Streaming Over Tcp. *ACM Trans. Multimedia Comput. Commun. Appl.* 4 (sep 2008).

[26] Ayub A. Gubran and Tor M. Aamodt. 2019. Emerald: Graphics Modeling for SoC Systems. In *Proc. of Int'l Sym. on Computer Architecture (ISCA '19)*.

[27] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S. Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. 2021. Chasing Carbon: The Elusive Environmental Footprint of Computing. In *IEEE Int'l Symp. on High Performance Computer Architecture (HPCA)*.

[28] Cameron Gutman, Diego Waxemberg, Aaron Neyer, Michelle Bergeron, Andrew Hennessy, and Aidan Campbell. 2024. Moonlight Stream. https://moonlight-stream.org/. [Online; accessed 18-Feb-2024].

[29] MyungJoo Ham, Inki Dae, and Chanwoo Choi. 2015. LPD: Low Power Display Mechanism for Mobile and Wearable Devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*.

[30] Sen He, Glenna Manns, John Saunders, Wei Wang, Lori Pollock, and Mary Lou Soffa. 2019. A Statistics-based Performance Testing Methodology for Cloud Applications. In *Proc. of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Accepted*.

[31] Mohamed Hegazy, Khaled Diab, Mehdi Saeedi, Boris Ivanovic, Ihab Amer, Yang Liu, Gabor Sines, and Mohamed Hefeeda. 2019. Content-Aware Video Encoding for Cloud Gaming. In *Proc. of ACM Multimedia Systems Conference*.

[32] H. Hong, C. Hsu, T. Tsai, C. Huang, K. Chen, and C. Hsu. 2015. Enabling Adaptive Cloud Gaming in an Open-Source Cloud Gaming Platform. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 12 (2015), 2078–2091.

[33] Luke Hsiao, Brooke Krajancich, Philip Levis, Gordon Wetzstein, and Keith Winstein. 2021. Towards Retina-Quality VR Video Streaming: 15ms Could Save You 80% of Your Bandwidth. *arXiv preprint arXiv:2108.12720* (2021).

[34] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. 2013. GamingAnywhere: An Open Cloud Gaming System. In *Proc. of ACM Multimedia Systems Conference*.

[35] Muhammad Huzaifa, Rishi Desai, Samuel Grayson, Xutao Jiang, Ying Jing, Jae Lee, Fang Lu, Yihan Pang, Joseph Ravichandran, Finn Sinclair, Boyuan Tian, Hengzhi Yuan, Jeffrey Zhang, and Sarita V. Adve. 2021. Exploring Extended Reality with ILLIXR: A new Playground for Architecture Research. arXiv:2004.04643

[36] Chanyou Hwang, Saumay Pushp, Changyoung Koh, Jungpil Yoon, Yunxin Liu, Seungpyo Choi, and Junehwa Song. 2017. RAVEN: Perception-Aware Optimization of Power Consumption for Mobile Games. In *Proc. of Int'l Conference on Mobile Computing and Networking*.

[37] Intel. 2021. Intel Microarchitecture Code Named Skylake-X Events. https://download.01.org/perfmon/index/skylake_server.html. [Online; accessed 10-Aug-2021].

[38] Gareth R James. 2010. *Citrix XenDesktop Implementation: A Practical Guide for IT Professionals*. Elsevier.

[39] Teemu Kämäräinen, Matti Siekkinen, Jukka Eerikäinen, and Antti Ylä-Jääski. 2018. CloudVR: Cloud Accelerated Interactive Mobile Virtual Reality. In *Proceedings of the 26th ACM International Conference on Multimedia*.

[40] David Kanter. 2015. Graphics processing requirements for enabling immersive vr. *AMD White Paper* (2015).

[41] Kyungah Kim and Won Woo Ro. 2019. Fast CU Depth Decision for HEVC Using Neural Networks. *IEEE Transactions on Circuits and Systems for Video Technology* 29, 5 (2019), 1462–1473. https://doi.org/10.1109/TCSVT.2018.2839113

[42] Albert M. Lai and Jason Nieh. 2006. On the Performance of Wide-area Thin-client Computing. *ACM Transactions on Computer Systems* 24, 2 (May 2006).

[43] Zeqi Lai, Y. Charlie Hu, Yong Cui, Linhui Sun, and Ningwei Dai. 2017. Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices. In *Proc. of Int'l Conf. on Mobile Computing and Networking*.

[44] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. 2015. Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming. In *Proc. of Int'l Conf. on Mobile Systems, Applications, and Services*.

[45] Yue Leng, Chi-Chun Chen, Qiuyue Sun, Jian Huang, and Yuhao Zhu. 2019. Energy-Efficient Video Processing for Virtual Reality. In *Proc. of Int'l Symp. on Computer Architecture*.

[46] David Li, Ruofei Du, Adharsh Babu, Camelia D. Brumar, and Amitabh Varshney. 2021. A Log-Rectilinear Transformation for Foveated 360-degree Video Streaming. *IEEE Transactions on Visualization and Computer Graphics* (2021), 1–10.

[47] Yusen Li, Chuxu Shan, Ruobing Chen, Xueyan Tang, Wentong Cai, Shanjiang Tang, Xiaoguang Liu, Gang Wang, Xiaoli Gong, and Ying Zhang. 2019. GAugur: Quantifying Performance Interference of Colocated Games for Improving Resource Utilization in Cloud Gaming. In *Proc. of Int'l Symp. on High-Performance Parallel and Distributed Computing*.

[48] Y. Lin and H. Shen. 2015. Cloud Fog: Towards High Quality of Experience in Cloud Gaming. In *Int'l Conf. on Parallel Processing*. 500–509.

[49] Luyang Liu, Ruiguang Zhong, Wuyang Zhang, Yunxin Liu, Jiansong Zhang, Lintao Zhang, and Marco Gruteser. 2018. Cutting the Cord: Designing a High-Quality Untethered VR System with Low Latency Remote Rendering. In *Proc. of Int'l Conf. on Mobile Systems, Applications, and Services*.

[50] Tianyi Liu, Sen He, Sunzhou Huang, Danny Tsang, Lingjia Tang, Jason Mars, and Wei Wang. 2020. A Benchmarking Framework for Interactive 3D Applications in the Cloud. In *Int'l. Symp. on Microarchitecture (MICRO)*.

[51] Tianyi Liu, Sen He, Vinodh Kumaran Jayakumar, and Wei Wang. 2022. A Cloud 3D Dataset and Application-Specific Learned Image Compression in Cloud 3D. In *Computer Vision – ECCV 2022*. Springer Nature Switzerland, Cham, 268–284.

[52] LizardByte. 2024. Sunshine. https://github.com/LizardByte/Sunshine. [Online; accessed 18-Feb-2024].

[53] Q. Lu, J. Yao, H. Guan, and P. Gao. 2020. gQoS: A QoS-Oriented GPU Virtualization with Adaptive Capacity Sharing. *IEEE Transactions on Parallel and Distributed Systems* 31, 4 (2020), 843–855.

[54] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel. 2008. Tradeoffs in Designing Accelerator Architectures for Visual Computing. In *IEEE/ACM Int'l Symp. on Microarchitecture*.

[55] P. Malani, Y. Tan, and Q. Qiu. 2007. Resource-aware High Performance Scheduling for Embedded MPSoCs With the Application of MPEG Decoding. In *IEEE Int'l Conf. on Multimedia and Expo*.

[56] Amrita Mazumdar, Armin Alaghi, Jonathan T. Barron, David Gallup, Luis Ceze, Mark Oskin, and Steven M. Seitz. 2017. A Hardware-Friendly Bilateral Solver for Real-Time Virtual Reality Video.

[57] Amrita Mazumdar, Brandon Haynes, Magda Balazinska, Luis Ceze, Alvin Cheung, and Mark Oskin. 2019. Perceptual Compression for Video Storage and Processing Systems. In *Proceedings of the ACM Symposium on Cloud Computing*.

[58] Jiayi Meng, Sibendu Paul, and Y. Charlie Hu. 2020. Coterie: Exploiting Frame Similarity to Enable High-Quality Multiplayer VR on Commodity Mobile Devices. In *Proc. of Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*.

[59] Microsoft. 2023. Project xCloud: Gaming with you at the Center. https://www.xbox.com/en-US/cloud-gaming. [Online; accessed 18-Oct-2023].

[60] Andrew Mulholland and Glenn Murphy. 2003. *Java 1.4 Game Programming*. Wordware Publishing, Inc.

[61] H. Nam and S. Lee. 2010. Low-power Liquid Crystal Display Television Panel with Reduced Motion Blur. *IEEE Transactions on Consumer Electronics* 56, 2 (2010), 307–311.

[62] Jason Nieh, S. Jae Yang, and Naomi Novik. 2003. Measuring Thin-client Performance Using Slow-motion Benchmarking. *ACM Transactions on Computer Systems* 21, 1 (Feb. 2003).

[63] NVIDIA. 2019. Why Does High FPS Matter For Esports? https://www.nvidia.com/en-us/geforce/news/what-is-fps-and-how-it-helps-you-win-games/. [Online; accessed 11-Dec-2020].

[64] NVidia. 2020. G-SYNC. https://developer.nvidia.com/g-sync. [Online; accessed 11-Dec-2020].

[65] NVidia. 2023. Adaptive VSync. https://www.geforce.com/hardware/technology/adaptive-vsync/technology.

[66] NVIDIA. 2023. Geforce Now. https://www.nvidia.com/en-us/geforce-now/. [Online; accessed 18-Oct-2023].

[67] Oculus. 2013. Building a Sensor for Low Latency VR. https://www.oculus.com/blog/building-a-sensor-for-low-latency-vr/. [Online; accessed 9-Dec-2021].

[68] Parsec. 2024. Parsec. https://parsec.app/. [Online; accessed 18-Feb-2024].

[69] Anuj Pathania, Alexandru Eugen Irimiea, Alok Prakash, and Tulika Mitra. 2015. Power-Performance Modelling of Mobile Gaming Workloads on Heterogeneous MPSoCs. In *Proceedings of the 52nd Annual Design Automation Conference*.

[70] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler. 2016. A case for toggle-aware compression for GPU systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[71] Xiaowei Ren and Mieszko Lis. 2021. CHOPIN: Scalable Graphics Rendering inMulti-GPU Systems via Parallel Image Composition. In *IEEE Int'l Symp. on High Performance Computer Architecture (HPCA)*.

[72] Alec Rohloff, Zackary Allen, Kung-Min Lin, Joshua Okrend, Chengyi Nie, Yu-Chia Liu, and Hung-Wei Tseng. 2021. OpenUVR: an Open-Source System Framework for Untethered Virtual Reality Applications.

[73] Shadow. 2024. Shadow PC. https://shadow.tech/. [Online; accessed 18-Feb-2024].

[74] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. 2013. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3* (8th ed.). Addison-Wesley Professional.

[75] Ivan Slivar, Lea Skorin-Kapov, and Mirko Suznjevic. 2016. Cloud Gaming QoE Models for Deriving Video Encoding Adaptation Strategies. In *Proc. of Int'l Conf. on Multimedia Systems*.

[76] Steam. 2024. Steam Link. https://en.wikipedia.org/wiki/Steam_Link. [Online; accessed 18-Feb-2024].

[77] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C. Lawrence Zitnick. 2017. ELF: An Extensive, Lightweight and Flexible Research Platform for Real-time Strategy Games. In *Advances in Neural Information Processing Systems 30*.

[78] Wei Wang, Tanima Dey, Jack W. Davidson, and Mary Lou Soffa. 2014. DraMon: Predicting Memory Bandwidth Usage of Multi-threaded Programs with High Accuracy and Low Overhead. In *Int'l Symp. on High Performance Computer Architecture*.

[79] W. Wang, N. Tian, S. Huang, S. He, A. Srivastava, M. L. Soffa, and L. Pollock. 2018. Testing Cloud Applications under Cloud-Uncertainty Performance Effect. In *11th IEEE Conference on Software Testing, Validation and Verification*.

[80] Josh Whitney and Pierre Delforge. 2014. Data Center Efficiency Assessment. *Issue paper on NRDC (The Natural Resource Defense Council)* (2014).

[81] Chenhao Xie, Xie Li, Yang Hu, Huwan Peng, Michael B. Taylor, and Shuaiwen Leon Song. 2021. Q-VR: System-Level Design for Future Collaborative Virtual Reality Rendering. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[82] C. Xie, F. Xin, M. Chen, and S. L. Song. 2019. OO-VR: NUMA Friendly Object-Oriented VR Rendering Framework For Future NUMA-Based Multi-GPU Systems. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*.

[83] C. Xie, X. Zhang, A. Li, X. Fu, and S. Song. 2019. PIM-VR: Erasing Motion Anomalies In Highly-Interactive Virtual Reality World with Customized Memory Cube. In *IEEE Int'l Symp. on High Performance Computer Architecture (HPCA)*.

[84] S. Jae Yang, Jason Nieh, Matt Selsky, and Nikhil Tiwari. 2002. The Performance of Remote Display Mechanisms for Thin-Client Computing. In *Proc. of USENIX Annual Technical Conference*.

[85] R. D. Yates, M. Tavan, Y. Hu, and D. Raychaudhuri. 2017. Timely cloud gaming. In *IEEE INFOCOM - IEEE Conf. on Computer Communications*. 1–9.

[86] Inchoon Yeo and Eun Jung Kim. 2008. Hybrid Dynamic Thermal Management Based on Statistical Characteristics of Multimedia Applications. In *Proc. of Int'l Symp. on Low Power Electronics and Design*.

[87] T. Yoshihara and S. Fujita. 2019. Fog-Assisted Virtual Reality MMOG with Ultra Low Latency. In *Int'l Symp. on Computing and Networking (CANDAR)*. 121–129.

[88] S. Zadtootaghaj, S. Schmidt, and S. Möller. 2018. Modeling Gaming QoE: Towards the Impact of Frame Rate and Bit Rate on Cloud Gaming. In *2018 Tenth International Conference on Quality of Multimedia Experience (QoMEX)*.

[89] C. Zhang, J. Yao, Z. Qi, M. Yu, and H. Guan. 2014. vGASA: Adaptive Scheduling Algorithm of Virtualized GPU Resource in Cloud Gaming. *IEEE Transactions on Parallel and Distributed Systems* 25, 11 (2014).

[90] Wei Zhang, Xiaofei Liao, Peng Li, Hai Jin, and Li Lin. 2017. ShareRender: Bypassing GPU Virtualization to Enable Fine-Grained Resource Sharing for Cloud Gaming. In *Proc. of ACM Int'l Conf. on Multimedia*.

[91] S. Zhao, H. Zhang, S. Bhuyan, C. S. Mishra, Z. Ying, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das. 2020. Déjà View: Spatio-Temporal Compute Reuse for‘ Energy-Efficient 360° VR Video Streaming. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*.