A Cloud Gaming Framework for Dynamic Graphical Rendering Towards Achieving Distributed Game Engines

James Bulman
Lancaster University

Peter Garraghan
Lancaster University

Abstract

Cloud gaming in recent years has gained growing success in delivering games-as-a-service by leveraging cloud resources. Existing cloud gaming frameworks deploy the entire game engine within Virtual Machines (VMs) due to the tight-coupling of game engine subsystems (graphics, physics, AI). The effectiveness of such an approach is heavily dependant on the cloud VM providing consistently high levels of performance, availability, and reliability. However this assumption is difficult to guarantee due to QoS degradation within, and outside of, the cloud - from system failure, network connectivity, to consumer datacaps - all of which may result in game service outage. We present a cloud gaming framework that creates a distributed game engine via loose-coupling the graphical renderer from the game engine, allowing for its execution across cloud VMs and client devices dynamically. Our framework allows games to operate during performance degradation and cloud service failure, enabling game developers to exploit heterogeneous graphical APIs unrestricted from Operating System and hardware constraints. Our initial experiments show that our framework improves game frame rates by up to 33% via frame interlacing between cloud and client systems.

1 Introduction

Video games are now the largest entertainment industry in the world, growing to $143.5b revenue in 2020 [1] with the PC distribution platform; Steam, reaching over 22 million concurrent players alone [2]. Cloud gaming - a paradigm whereby gaming is delivered as a service by leveraging cloud resources [3] - has begun to gain increasing popularity in society, providing advantages over traditional desktop and console gaming including lower installation times, reduced hardware cost, greater device portability, and the ability to leverage cloud resources for higher graphical quality [3].

A design principle shared across all existing cloud gaming frameworks [3–7] is that service is provisioned by deploying a *game instance* (an instantiation of a game program) within a Virtual Machine (VM), whereby game state is manipulated and resulting frames are encoded and streamed to a consumer client device based on user input [8]. This approach is necessary primarily due to the *game engine* architecture: Monolithic systems with multiple *subsystems* tightly coupled with each other and the underlying operating system to facilitate aspects of graphics, physics, audio, and AI [9].

However such a design results in limitations within cloud gaming due to a strong dependence on performance and dependability of the cloud game instance and its connection to client devices. In the cloud, game instances deployed within VMs are exposed to a plethora of detrimental datacenter behaviors, spanning interference [10], resource contention [11], and failures [12]. Such behaviors - in isolation or in tandem - result in Quality of Service (QoS) degradation in terms of lower interactivity and frames per second (FPS), lower graphical quality (lower resolution, bitrate), as well as reduced availability and reliability [13]. While various approaches have been proposed to alleviate such issues [4, 14, 15], they all assume a stable network connection established between the client device and the cloud game instance. Thus, such approaches are unable to tolerate cloud outages, prolonged network disconnection, or consumer data caps, resulting in consumers unable to access games entirely.

One approach to overcome these limitations would be to *dynamically distribute all game engine subsystems across the cloud* - and is the overarching research vision for our work. A distributed game engine would allow for dynamic deployment and reconfiguration of all subsystems across both cloud and client systems in response to the specified QoS requirements. However this is currently not possible in existing game engine architectures due to tight-coupling with their underlying platforms [16]. Taking graphical rendering (a core game subsystem) as an example, game systems are developed to use specific graphics APIs (OpenGL, Vulkan, etc.), and require specialized developer knowledge to take advantage of advanced hardware features (e.g. real-time ray-tracing). Thus it is not possible to transition to other graphics APIs and platforms without significant development effort and time.

In this paper we present a cloud gaming framework that enables loose-coupling between the game engine and the graphical renderer, representing the first distributed game engine designed for the cloud. Our approach allows for dynamic run-time rendering across cloud and client systems using heterogeneous graphics APIs (Vulkan, OpenGL) in response to changing performance and network conditions. Our specific contributions are as follows:

- Creation of a distributed cloud gaming framework whereby graphical rendering is performed by the game instance via submitting generic graphics commands to the framework which automatically converts commands into specific graphics API function calls in cloud and client systems based on QoS and hardware constraints. Our design reduces game engine graphics development complexity across heterogeneous platforms.
- Empirical demonstration that our framework enhances game instance performance and fault-tolerance via renderer hot-swapping and frame interlacing. Our results show that cooperative client and cloud-based rendering can improve game frame rates by 33% against client-side or cloud-side rendering whilst using 1.1-4.0 Mbps

2 Background

Game Engines: A term popularized in the 1990's [17, 18], game engines are a composition of subsystems working in tandem to facilitate real-time interactive simulation for entertainment purposes [19]. Game engines are used to instantiate a *game instance*, which provide coded gameplay logic that utilizes generic data-driven systems provided by the game engine to perform a *game loop* [19]. As shown in Figure 1, the game loop includes a set of distinct stages, each executed once per frame, and will interact with the operating system and hardware in different ways. *Handle Input* polls input events from user devices, *Update Game State* advances virtual world state to the next frame and prepares for rendering loading for any required assets, *Render Game State* interacts with the graphics hardware via the graphics API to generate an image to be displayed on the screen, ending with *Swap Buffers* displaying the computed image. Modern game engines are designed as fully-fledged frameworks, such as Unity [20] and Unreal Engine 4 [21], using different architecture paradigms (event-driven, data-driven) [22, 23]. Notably, most modern game engines comprise four fundamental subsystems - physics, sound, input handling and graphics [16] [24].

Renderer: The graphical renderer[1] is the game engine subsystem responsible for performing graphical operations to display geometry, lighting and texture information [8]. A renderer exploits *graphic APIs* such as OpenGL [25], Vulkan [26], DirectX [27] and WebGPU [28] to interface with the

[1] Which we refer to as the *renderer*

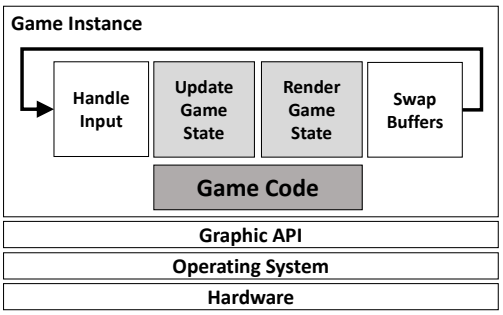


Figure 1: Depiction of standard game loop.

graphics hardware. Graphics hardware (such as GPUs) follow a strict computation pipeline, as shown in Figure 2, with each graphics API using a different method to manipulate or interface graphics pipeline stages. If the graphics API cannot modify a specific stage (i.e. proprietary), the hardware is responsible for performing the specific stage. Operating System support for a given graphics API is determined by the organization that defines it, leading to reduced portability between platforms as a graphics API may only work for a subset of hardware vendors, and is especially true for *Game Consoles APIs*, DirectX [27] and Metal [29]. Thus, it takes considerable developer experience and effort to build games to be Operating System agnostic. Specific graphics features (such as real-time raytracing) are currently only accessible on specific hardware architectures [30].

Cloud gaming: Cloud gaming is a paradigm whereby cloud-resources are provisioned as a service for gaming purposes, interacted by users via a thin client [31]. Cloud gaming enables game engines to be deployed and executed entirely within the cloud, running as a *cloud game instance* within an isolated environment such as a VM [3, 5–7]. As depicted in Figure 3, user inputs are streamed from a device to the cloud game instance, which are used to update and render the game state. Once the frame has been rendered in the cloud, it is compressed via a video codec algorithm and streamed to the client to be displayed on the screen. Cloud gaming provides several key advantages over traditional gaming on dedicated consumer hardware, including greater device and game instance portability [32], reduced location restrictions and power consumption for consumer devices, and access to cutting-edge graphics features such as ray tracing [3].

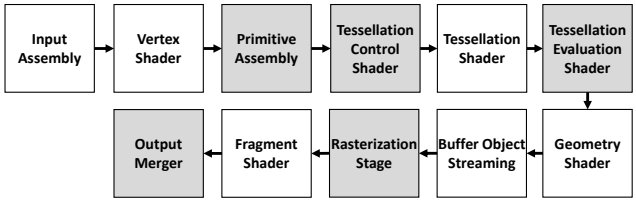


Figure 2: Graphics Pipeline (Grey denotes limited/no developer control).

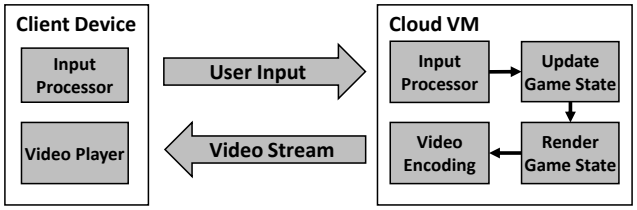


Figure 3: Depiction of Cloud Gaming Architecture

Cloud gaming limitations: Delivering gaming service via existing frameworks [3, 5, 7] are strongly dependant on the operational performance and dependability of the cloud game instance and its connection to the consumer. Game instances deployed within virtualized datacenters will encounter detrimental phenomena including co-location performance interference [10], high contention [11][2], and frequent software and hardware failure [12]. All of these behaviors result in reduced QoS, measured in terms of interactivity, FPS, reduced graphics quality, as well as reduced availability and reliability.

3 Related Work

Various cloud gaming approaches have been proposed to enhance game instance operation [33–37]. Of relevance to this work are all empirically evaluated gaming and streaming systems which deploy game instances in cloud VMs. Huang et al. [3] proposed a streaming system that executes the game instance in a cloud VM, with Audio-Visual data streamed to user thin-clients. Polled user input is encoded into a custom protocol sent to the VM to update game state and frame rendering which is transmitted, decoded, and displayed on the client. Shi et al. [38] proposed a streaming framework which allows post-processing effects to be computed on the client device with the majority of the game computation and rendering performed on the cloud. The work uses rendering viewpoint, pixel depth and camera motion to perform image warping to improve the quality and encoded efficiency of the streamed video. Commercial cloud gaming frameworks such as Google Stadia [7] and Microsoft Project xCloud [6] also utilize cloud VMs, however due to the closed-source nature of projects it is currently unknown how game engines are modified to execute within their frameworks using the Vulkan graphics API [26]. Lee et al. [4] proposed a system which actively speculates user thin-client input, and predicts a subset of next possible frames for streaming upon receiving input. Such work helps tolerate network latency due to reduced computational delay between receiving input and frame streaming, however incurs substantial cloud VM computation. Command-based execution for rendering and input handling over the network has been used in the X11 protocol [39], however is limited to sending display commands to client machines as opposed to graphics rendering.

[2]Outages are common for Day 1 major game releases expansions.

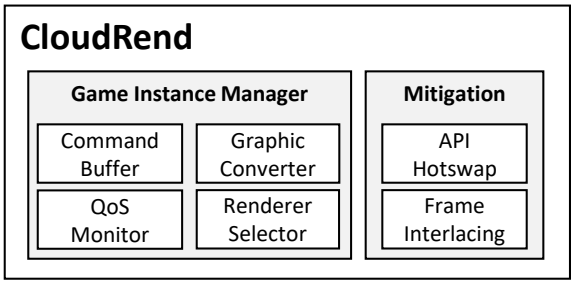


Figure 4: Overview of CloudRend component.

4 System Design

Overview: We propose a cloud gaming framework that creates a distributed game engine that loosely-couples the renderer subsystems from other subsystems. The renderer is loosely-coupled by removing the game code's ability to perform direct function call access to specific graphics APIs. Instead, the game code generates generic graphics commands enqueued to a *CloudRend* component that interprets and converts commands into specific graphics API function calls to render across cloud or client systems, driven by QoS and hardware constraints.

Rationale: The advantage of using general graphics commands is two-fold: First, our framework only requires to map specific graphics API calls once, after which game code can now leverage multiple graphics APIs, thus reducing and simplifying development effort to implement specific graphics function calls for every new game engine. Second, it allows for ready transmission and replication of the renderer across client devices and cloud VMs comprising heterogeneous Operating Systems and hardware constraints.

CloudRend: The CloudRend component is responsible for determining and managing game instance rendering for game code. CloudRend can execute as a process in a single game instance, or be deployed in its own VM to facilitate multiple game instances. CloudRend comprises two main sub-components as shown in Figure 4. The *Game Instance Manager* interprets, transmits, and converts generic graphical commands from the game code into a specific graphics API for rendering across the system. *Mitigation* is responsible for selecting and executing mechanisms to improve game instance QoS due to performance degradation and failure. We have focused on developing three core generic graphical commands for CloudRend as shown in Table 1. These commands were chosen as they are fundamental across all graphics APIs: *RenderClear* for screen clearing at frame start, required for more advance functions *RenderModel* for drawing geometry and *RenderLightSet* for large computation (per pixel lighting).

Phases: CloudRend uses four phases shown in Figure 5.

Buffering: At the beginning of the game instance, the game code requests CloudRend to create a *Command Buffer*. The Command Buffer is a small block of memory used to

Description	CloudRend	OpenGL	Vulkan
Clear screen with color; optionally clear depth buffer	*RenderClear()*	*glClearColor()*, *glClear()*	*vkCmdBeginRenderPass()*
Draws referenced model to screen with model matrix	*RenderModel()*	*glBindBuffer()*, *glDrawElements()*, *glUniformMatrix4fv()*	*vkCmdBindVertexBuffers()*, *vkCmdDrawIndexed()*, *vkCmdPushConstants()*, *vkCmdBindIndexBuffer()*
Set position and color of global lighting setup	*RenderLightSet()*	*glUniform3fv()*, *glUniform4fv()*	*vkCmdPushConstants()*

Table 1: CloudRend generic graphics commands and graphics API calls.

store command-information tuples responsible for encoding generic graphics commands along with the relevant information required for processing. For example, during a given game loop iteration, the game code executes command *RenderModel(x,y)* to render a model reference *x* with matrix *y*.

Discovery: Upon game instance instantiation and execution, the *QoS Monitor* will determine hardware constraints and periodically measure game instance performance for rendering within client devices or the cloud VM. If failure or performance degradation is detected, CloudRend will activate *Mitigation* to enhance game instance QoS.

Conversion: After game code has completed state updates and render commands required for the current game loop, all graphics commands in *Command Buffer* are transmitted by CloudRend to the *Graphic Converter* in the client device or cloud VM. Commands are automatically translated via a jump table that corresponds to one or more graphics API-specific procedure calls that are emitted with required data paired with command type. *RenderModel()* called in Vulkan is converted to *vkCmdBindVertexBuffers()* and *vkCmdDraw()* as shown in Table 1. The jump table allows command execution to scale if new commands are required they can just be appended to this table with a corresponding command type for lookup.

Drawing: Our framework has two methods for image drawing: *Co-location* (game code and rendering performed in the same system) iterates over command-information tuples, executes graphics API function calls, and displays the produced image. *Distributed* (rendering performed in a different instances from game code) encodes and streams the image from the remote renderer to game code and drawn to client window.

Mitigation: CloudRend exploits renderer and game engine distribution to enhance game instance QoS via two features:

Interlacing: Enables multiple renderers distributed across the client device and cloud VM to collaboratively render frames based on game instance performance. The proportion of frames per second rendered by client or cloud systems can be user-defined, or dynamically calculated based on game instance performance (FPS), network performance (latency, bandwidth), constraints (client hardware architecture). For example, if the CloudRend *QoS Monitor* detects good network performance between the client and cloud, an increasing proportion of frames will be rendered within the cloud, and vice versa for degraded network conditions (and entirely client-side in the event of no network access).

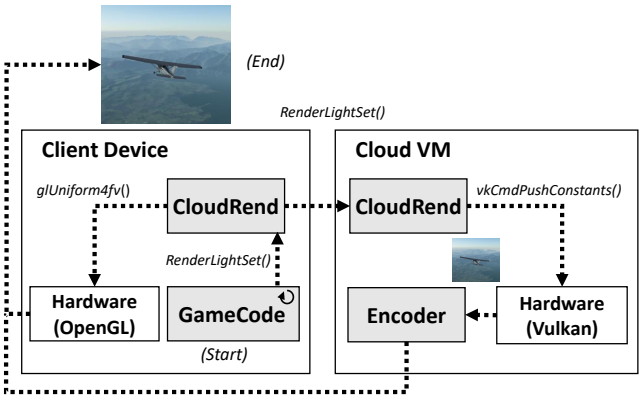


Figure 5: Cloud gaming framework in operation.

Hot-Swapping: Allows dynamic run-time renderer reconfiguration with no impact on game code execution. In the event a graphics API error, we redefine enabled extensions or alternatively swap to another graphics API. A game instance using a cloud renderer can have the client renderer on warm-standby to activate upon failure detection or network connection loss. A benefit is not needing to restart the game instance upon major graphics reconfiguration (i.e. Anti-aliasing, Tessellation etc.) currently required for the majority of current games. While we have implemented this feature (we observed no observable frame rate difference when swapping graphics APIs during game instance execution), it has been omitted from evaluation due to space limitations. Even with such mitigation in place, severe scenarios (cloud outage, network disconnection) will still reduce graphical quality due to resource and/or hardware constraints on client systems. We believe that tolerating such scenarios is an acceptable alternative compared to service loss.

5 Preliminary Evaluation

For our experiments we have measured the performance and resource-efficiency of our proposed cloud gaming framework within various operational scenarios.

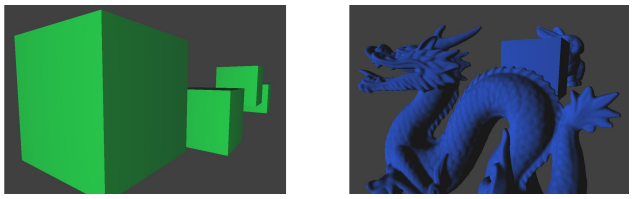


Figure 6: Rendered scenes: Low (100 vertices), High (200,000 vertices).

Setup: The cloud VM was created via deploying a VM in a desktop machine (i7-7700HQ, 16GB RAM, Intel HD 630), and using a Raspberry Pi 4 as the client device. The selected experiment scenarios entails our framework executing a game instance where the client device contains both the game code and CloudRend, while the cloud VM contains CloudRend. Network communication between the client and cloud was performed over a wireless network to introduce greater latency variability between 3ms - 100ms. The Cloud VM was configured to only use Vulkan, and the client device OpenGL (Vulkan is currently not supported on the Raspberry Pi). The zlib library was used to compress streamed images from the cloud VM, reducing its size by up to 80%. The framework was configure at different frame interlacing ratios, ranging between 100% client (rendering performed on client device), 100% cloud (rendering on cloud VM), and client-cloud ratios of 25-75%, 50-50%, and 75-25%. Each experiment entailed the game instance executing two 1280x720 scenes for 5,000 frames at 60FPS for different levels of graphical quality as depicted in Figure 6: *low* (100 vertices) and *high* (200,000 vertices) both using per pixel diffuse lighting in order to create additional computation within the system.

Frame Rate: Figure 7 shows the game instance frame rate using different framework interlacing ratios. We observed that the client device is capable of achieving a frame rate of 60 FPS for low vertices scenes, and degrades to 41 FPS with increased cloud interlacing. This is because the client device is sufficiently powerful to render the geometry of low vertices scenes faster than the latency of transmitting the rendered and encoded images from the cloud VM. However, this pattern changes for high vertices scenes, where we found 50/50% frame interlacing gave a 33% improvement over cloud or client-only rendering. Such improvement is due to the cloud renderer providing faster rendering for complex geometry scenes resultant of greater compute power, however still incurs latency issues for Cloud VM transmission. However, we believe that cloud rendering performance will increase proportionally to scene complexity, as we identified through experiments that latency stemming from the cloud VM was not due to rendering or CloudRend, but due to the transport system that transfers the image into GPU memory creating a performance bottleneck. Hence, our initial results indicate the existence of a trade-off between between game instance rendering on a slower, yet direct client device rendering against a faster, yet latency induced cloud VM renderer.

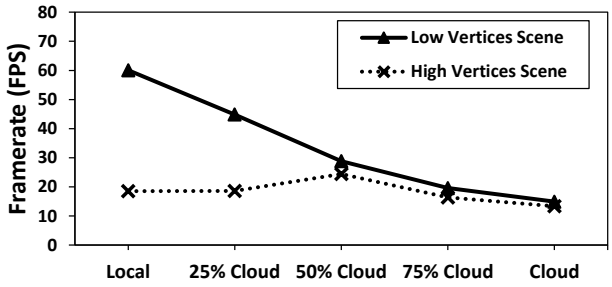
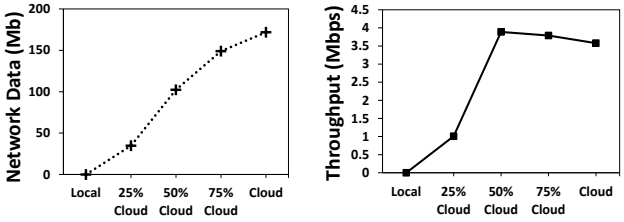


Figure 7: Comparison of frame rate during frame interlacting.



(a) Total Network Usage (b) Throughput

Figure 8: Cloud gaming network characteristics

Network: We found that outbound network traffic (i.e. generic graphics commands to the Cloud VM) sent 12-32kbs. For inbound network traffic of images to the client device, we observed that when leveraging the cloud VM, our approach attained network throughput between 1.1Mpbs-4Mbps as shown in 8b. We observed that 50-50% interlacing achieved the highest network throughput compared to 100% cloud rendering at 3.6Mbps. This is because 50-50% interlacing achieves the highest frame rate as shown in Figure 7, and thus capable of streaming more images per second. Increased cloud rendering results in a linear increase in network data as shown in Figure 8a. As compared to other cloud gaming approaches, network usage will be higher as command buffer size increases, especially for games at higher frame rates.

6 Conclusions

All existing Cloud gaming frameworks deploy and stream game engine instances using VMs. However such design encounters issues with performance degradation and failure in the cloud and between cloud and consumers. In this paper we presented a cloud gaming framework that dynamically distributes the game engine via loose-coupling of the graphical renderer from the game engine, allowing for cloud-client rendering. We have empirically demonstrated that our framework improves frame rates against solely cloud or client rendering. Results indicate that future cloud gaming engines (desktop, mobile, VR) may benefit from further distribution, allowing deployment in cloud-client, cloud-cloud, or fog-cloud based on performance and graphical quality trade offs.

7 Discussion

While have successfully distributed the game engine within the cloud, there are some clear improvements to the design and implementation of our cloud gaming framework.

Performance: An improved frame display system can be created for better compression size and speed of streaming frames from the cloud VM using a video codec such as H.265/HEVC. The network implementation can also be optimized; during experiments we discovered that performance bottlenecks from the cloud VM were not due to framework rendering, but from our networking architecture using TCP streams that blocked the game loop while the client waits for the VM to send a computed frame. This could be improved by multi-threading the network system so the renderer does not block the main game loop while waiting to receive cloud frames, allowing more command buffers to be generated, thus increasing the cloud rendering framerate.

Graphics APIs: While we have currently implemented generic graphic commands in OpenGL and Vulkan, we aim to include additional commands and graphics APIs in future work. A long-standing issue in the games industry is the secretive nature of in-house game engine development which have gotten increasingly larger and complex. We hope this research would allow development of game engines across platforms to be considerably simpler and easier.

Multi-instance: Although we have focused on a single client game instance connecting to dedicated cloud VM similar to existing cloud gaming frameworks, we believe that the CloudRend component can be designed to facilitate multiple clients, allowing for multi-tenant rendering across entirely different game codes currently not explored. We aim to explore alternative game instance deployment scenarios (e.g. code deployed on a cloud VM with client-side rendering). Multiple Cloud VM instances could be deployed for a single client to maximize game instance performance and fault tolerance in the event of instance failure.

Mitigation: While we have demonstrated that frame interlacing can help tolerate performance degradation and failure, we plan to evaluate hotswapping and have mechanisms dynamically adapt their operation in response changing system and network conditions. Calculating the optimal ratio for frame interlacing between the cloud and client based on monitoring and modelling game instance and network performance is likely a sensible direction to investigate.

Vision: We intend to expand our cloud gaming framework to encompass other game engine subsystems such as the physics, animation and artificial intelligence. We are interested in soliciting feedback on interesting challenges posed with run-time discovery and orchestration of highly loosely-coupled and distributed game engine subsystems. This challenge is difficult as subsystems such as physics require strict timing requirements within the game code, and latency between networked subsystems between the cloud instances.

Acknowledgements

We would like to thank anonymous HotCloud reviewers and our shepherd Ymir Vigfusson for their constructive comments and valuable feedback to improve our paper.

References

[1] B. Newzoo, "The global games market report," *Amsterdam: gamesindustry. com*, 2017.

[2] V. Inc. (2020) Steam: Game and player statistics. [Online]. Available: https://store.steampowered.com/stats/

[3] C.-Y. Huang, C.-H. Hsu, Y.-C. Chang, and K.-T. Chen, "Gaminganywhere: an open cloud gaming system," in *Proceedings of the 4th ACM multimedia systems conference*, 2013, pp. 36–47.

[4] 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 *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, 2015, pp. 151–165.

[5] D. De Winter, P. Simoens, L. Deboosere, F. De Turck, J. Moreau, B. Dhoedt, and P. Demeester, "A hybrid thin-client protocol for multimedia streaming and interactive gaming applications," in *Proceedings of the international workshop on Network and operating systems support for digital audio and video*, 2006, pp. 1–6.

[6] Microsoft. (2020) Microsoft project xcloud. [Online]. Available: https://www.xbox.com/en-US/xbox-game-streaming/project-xcloud

[7] Google. (2019) Google stadia. [Online]. Available: https://stadia.google.com/

[8] T. Koehler, A. Dieckmann, and P. Russell, "An evaluation of contemporary game engines," in *Proceedings of the 26th eCAADe Conference*, 2008, pp. 743–750.

[9] E. F. Anderson, S. Engel, P. Comninos, and L. McLoughlin, "The case for research in game engine architecture," in *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, 2008, pp. 228–231.

[10] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–12.

[11] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu, "Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters," *IEEE Transactions on Services Computing*, vol. 12, pp. 91–104, Jan 2019.

[12] X. Chen, C. Lu, and K. Pattabiraman, "Failure analysis of jobs in compute clouds: A google cluster case study," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, Nov 2014, pp. 167–177.

[13] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan 2004.

[14] W. Zhang, J. Chen, Y. Zhang, and D. Raychaudhuri, "Towards efficient edge cloud augmentation for virtual reality mmogs," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–14.

[15] J. R. Lange, P. A. Dinda, and S. Rossoff, "Experiences with client-based speculative remote display." in *USENIX Annual Technical Conference*, 2008, pp. 419–432.

[16] D. Maggiorini, L. A. Ripamonti, E. Zanon, A. Bujari, and C. E. Palazzi, "Smash: A distributed game engine architecture," in *2016 IEEE Symposium on Computers and Communication (ISCC)*, 2016, pp. 196–201.

[17] idSoftware. (1996) Quake. [Online]. Available: https://github.com/id-Software/Quake

[18] idSoftware. (1993) Doom. [Online]. Available: https://github.com/id-Software/DOOM

[19] J. Gregory, *Game engine architecture*. crc Press, 2018.

[20] U. Technologies. (2005) Unity3d game engine. [Online]. Available: https://unity.com/

[21] I. Epic Games. (2004) Unreal 4 engine. [Online]. Available: https://www.unrealengine.com/

[22] S. Marks, J. Windsor, and B. Wünsche, "Evaluation of game engines for simulated surgical training," in *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, 2007, pp. 273–280.

[23] S. Bilas, "A data-driven game object system," in *Game Developers Conference Proceedings*, 2002.

[24] M. Lewis and J. Jacobson, "Game engines," *Communications of the ACM*, vol. 45, no. 1, p. 27, 2002.

[25] K. Group. (2017) Khronos opengl 4.6 specification. [Online]. Available: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf

[26] Khronos Group. (2020) Khronos vulkan specification. [Online]. Available: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html

[27] Microsoft. (2019) Direct3d 12 programming guide. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide

[28] W3C, "WebGPU," May 2020. [Online]. Available: https://gpuweb.github.io/gpuweb/

[29] A. Inc. (2020) Metal - apple developer. [Online]. Available: https://developer.apple.com/metal/

[30] NVIDIA. (2018) Nvidia turing gpu architecture. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[31] K. Chen *et al.*, "On the quality of service of cloud gaming systems," *IEEE Transactions on Multimedia*, vol. 16, no. 2, pp. 480–495, 2013.

[32] W. Cai, R. Shea, C.-Y. Huang, K.-T. Chen, J. Liu, V. C. Leung, and C.-H. Hsu, "A survey on cloud gaming: Future of computer games," *IEEE Access*, vol. 4, pp. 7605–7620, 2016.

[33] H. Hong, D. Chen, C. Huang, K. Chen, and C. Hsu, "Placing virtual machines to optimize cloud gaming experience," *IEEE Transactions on Cloud Computing*, vol. 3, no. 1, pp. 42–53, 2015.

[34] Ubitus. (2019) Ubitus gamecloud. [Online]. Available: http://www.ubitus.net

[35] H. Hong, C. Hsu, T. Tsai, C. Huang, K. Chen, and C. Hsu, "Enabling adaptive cloud gaming in an open-source cloud gaming platform," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 12, pp. 2078–2091, 2015.

[36] Y. Xu, Q. Shen, X. Li, and Z. Ma, "A cost-efficient cloud gaming system at scale," *IEEE Network*, vol. 32, no. 1, pp. 42–47, 2018.

[37] W. Cai, Y. Chi, C. Zhou, C. Zhu, and V. C. M. Leung, "Ubcgaming: Ubiquitous cloud gaming system," *IEEE Systems Journal*, vol. 12, no. 3, pp. 2483–2494, 2018.

[38] S. Shi, C. Hsu, K. Nahrstedt, and R. Campbell, "Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming," in *ACM international conference on Multimedia*, 2011, pp. 103–112.

[39] J. G. Steiner, D. E. Geer, and Jr., "Network services in the athena environment," in *Proceedings of the Winter 1988 Usenix Conference*, 1988.