Distributed Rendering for Video Games via Object Streaming

Giacomo Parolini*, Dario Maggiorini†, Davide Gadia‡, and Laura Anna Ripamonti§
Department of Computer Science
University of Milan
Milan, Italy
Emails: *giacomo.parolini@studenti.unimi.it, †dario@di.unimi.it, ‡gadia@di.unimi.it, §ripamonti@di.unimi.it

***Abstract*—Propelled by the growing availability of broadband connection in recent years, the gaming industry is now devoting a considerable amount of resources and investments in online and cloud gaming. In legacy online gaming, the gaming experience is usually provided with the support of remote servers, and online players rely on their local PCs or consoles, which hold a local copy of all the content (assets) and must follow some minimum requirements in hardware and software specifications. Whereas, on cloud gaming, intensive computational tasks are almost completely offloaded to dedicated servers: video frames are rendered on the remote machine, encoded and sent to the players as a video stream. This approach soften the need for updated and powerful devices, but it suffers from all the limitations and problems inherent to multimedia real-time streaming.**

In this paper we explore an hybrid approach between a (video) streaming-based cloud gaming and the traditional approach where all assets are local to the player. We propose a solution where the rendering pipeline is split between server and client. In this distributed architecture, the server manages most of the game scene description, runs the game simulation, performs the first segment of the graphics pipeline's application stage, and finally sends a stream of pre-processed graphical objects to the client, which performs the final rendering steps. The proposed approach reduces the computational burden on the server, which is not required to perform rendering, improving scalability when compared with cloud gaming solutions based on video streaming.

***Index Terms*—Video Games, Distributed Rendering, Cloud gaming, Online gaming, Game streaming**

I. Introduction

The ever-increasing demand of high-end graphics, sophisticate AI, realistic animations, and content richness in modern video games today requires consumers to buy more and more powerful (and expensive) hardware in order to unlock the full potential games have to offer. Moreover, the increasing graphics fidelity and sheer amount of content of modern games means they are getting bigger and bigger in size very rapidly. Since the vast majority of modern video games are delivered through the Internet rather than on physical media, the situation above translates in longer and longer waiting times between the acquisition of a game and its fruition. This problem is made worse by the fact that most video games, both single and multiplayer, are patched and updated on a regular basis. Updates for high-end video games can easily reach hundreds, if not thousands, of megabytes in size.

The increasing availability of broadband connections we can witness worldwide allows for brand new approaches to solve the above-mentioned problems. As a standard feature, modern gaming platforms adopts a *progressive downloading* approach. This approach consists in downloading not the entire game at once, but just a small part of it. This allows the player to start playing immediately, while the rest of the game is fetched in the background. A progressive downloading reduces the inconvenience of long waiting times, but it comes with some downsides. First of all, several game features may not be available until the download has reached a certain completion threshold. Moreover, the game still takes up all the storage space it would without file streaming, and periodic updates are essentially untouched by this approach. Last but not least, an initial waiting time is still required.

In recent years, the gaming industry has devoted resources and investments in *online* and *cloud multiplayer gaming*. Legacy online gaming exploits remote servers to provide centralized operations required by a game (e.g., interaction between different players and synchronization of the game state), and relies to each player's local device for the actual rendering. The rendering is performed upon receiving all the required data in order to replicate the current shared state of the game. It is important to note that all assets are already present at client side; data sent over the network include only status updates.

In online multiplayer games, many clients are connected to the same server. This server is in charge to manage the shared virtual environment for all clients, collect inputs from the network and then update the virtual world. This update includes, among other things, physics simulation, collisions detection, and AI procedures for all non-player characters in the virtual world. The updates to the virtual environment are then sent back to all clients. Each client collects its own updates and renders a new scene to the player. This approach requires from the gaming device used by the player to fulfill minimum requirements in hardware and software capabilities for the rendering operations.

As an alternative approach, the idea behind cloud gaming is to never deliver the full game to the players. Instead, the game is executed and rendered completely on a remote server. The rendered video frames are then encoded and sent to the player as a video stream. This approach has several advantages: the player's gaming device does not need to be a powerful machine or even be compatible with the game's requirements:

it only needs the capability to render a video stream [1]. By implementing a cloud gaming architecture, the game can be fully treated as a service: extending system scalability, allowing innovative multiplayer game design patterns, and enabling new revenue models such as monthly subscription or pay-as-you-play [2].

However, the cloud gaming paradigm is not a panacæa. The streaming of video game frames incurs all the limitations and problems inherent to multimedia real-time streaming, such as image degradation under congested network conditions. Studies conducted on cloud gaming services show that the achieved frame rate drops linearly with the network packet loss, while graphics quality can degrade sensibly for bandwidths smaller than 4 Mbps [3]. Even on optimal conditions, streaming video games at higher resolutions require a proportionally higher bandwidth and, in a market where FullHD and even 4K gaming is becoming the standard, providing a resolution capped at 720p is likely to leave a big share of customers dissatisfied from the service.

Furthermore, the *interaction delay*, i.e., the round-trip time (*RTT*) from the moment an input is generated at client-side and when the updated frame is delivered back from the server is not negligible [4]. This means that very fast-paced games, such as first person shooters, may offer a substantially degraded experience when played via cloud gaming and streamed over a path spanning multiple ISPs. This degradation poses a serious limitation to the user experience if the game being played is a real time online multiplayer game, especially a competitive one.

Time measurements on popular cloud gaming platforms show that Round Trip Time (RTT) values consistently remain below 25 ms, with average values between 10 and 15 ms [5]. Nevertheless, this may be not enough for games where interaction time is fundamental, such as online beat'em ups or highly competitive eSport games such as League of Legends.

In this paper, we explore an hybrid approach between video-streaming-based cloud gaming and the traditional "local gaming" model. We split the rendering pipeline in two parts and provide a solution where some stages of the rendering are devolved to the server and some to the client. This distributed architecture has the goal to maximize data transfer efficiency and video quality robustness. In the proposed approach, we create a distributed system where the server contains most of the game world description and sends a pre-processed stream of data to the client, which performs a limited set of operations required to create the final rendered frame. The goal is to to unburden the client from downloading the game assets, thus saving storage and waiting time for the player, and the server from fully rendering the video game, thus saving computational power and improving scalability for the cloud gaming service provider.

The remainder of this paper is organized as follows. Section II presents a review of some related contributions to the problem in literature. In Sec. III we briefly present the graphics rendering pipeline. In Sec. IV the proposed solution is described. In Sec. V, we provide a performance analysis of the proposed system. Finally, Sec. VI draws conclusions and suggests future work.

II. Related work

Cloud gaming has been subject for many scientific studies over the last few years [1]. One relevant question is if the game engines currently in use are effective and flexible enough to manage large-scale cloud gaming development [6]. This has lead to the proposal of solutions for *distributed* game engines. This family of game engines is characterized by the decomposition of the architecture of a game engine in independent modules interacting with each other by means of a dedicated messaging protocol.

An example of game engine following this approach is *SMASH* (*Stackless Microkernel Architecture for SHared environments*) [7]. The dynamic and independent modules of *SMASH* use a microkernel-like message bus for internal intercommunication. With this approach, game modules can be inserted, debugged, and removed from a running engine once an internal messaging protocol is clearly defined. Moreover, modules can also be dynamically dislocated on multiple machines in order to achieve a more scalable and fault-resilient system.

Authors of [8] proposed a distributed game engine characterized by a loose-coupling between the graphical renderer module and the rest of the game engine. The proposed approach is based on the use of generic graphics commands, which are sent to a cloud component and then converted to actual graphics API calls. The architecture allows to use and dynamically change graphics modules with different hardware configurations and heterogeneous graphics APIs, in order to adapt to changing network conditions.

Considering now the streaming of geometry information over a network, which is the main focus of this paper, in [9] it was proposed a method for robust progressive 3D geometry streaming over lossy communication channels, based on redundant and order-independent information. The authors decompose a 3D model into ellipsoids which are efficiently encoded and sent through the network to form a coarse approximation of the model, interleaving them with sampled points used to refine the surface details. The technique allows for fast delivery of a rough (but still high-fidelity) version of the model even under poor network conditions. This rough model can then be refined later. Following a similar approach, authors of [10] used a "gaze-guided" method to progressively send increasing Level Of Details (LODs) of a 3D model with a point-based streaming technique.

Finally, in [11] it is presented an alternative cloud gaming system called *LiveRender* that implements "graphics streaming". Graphics streaming, rather than streaming rendered frames to the client, streams graphics commands (along with their geometry data) which are interpreted by the client and rendered on its GPU. This is in general more bandwidth-costly than video streaming, but *LiveRender* accomplishes to reach lower traffic by compressing the stream in sophisticate ways closely tailored to graphics commands and geometry. Their

approach can be applied to any game thanks to a software layer that intercepts and wraps graphics API calls on the server.

III. Overview of the graphics rendering pipeline

Graphics rendering [12] is probably the most important task of a game engine, because it is responsible of generating the visual information provided to the player. Currently, an acceptable frame rate for optimal player experience in a video game is around 60 frames per second. Thus, it is evident how the graphics rendering pipeline need to be highly optimized.

The real-time rendering pipeline is composed by different stages: some of the preliminary operations (like e.g., loading of the game assets) are performed on the CPU, while the rest of the operations are performed on the GPU. The GPU is specifically designed to accelerate computationally expensive tasks as vertex processing and per-pixel calculations, which are operations that involve a high level of data parallelism. Modern GPUs allow the developers to control these operations through the use of *shaders*.

IV. The proposed solution

As already mentioned, our solution splits the rendering pipeline among the server and the client.

The server performs the first part of the pipeline: it owns and manages all the game assets (3D models, textures, etc.), and its job is to run the game simulation, update the information about the game scene content, and stream to the client the raw data assets it needs with the due priority. In a game engine, the game simulation involves not only the graphics rendering, but also a certain number of other operations, whose computation is performed on the CPU, and can affect the final rendering of the scene [13]. Common examples of non-graphical operations are Artificial Intelligence and physics simulation. In this paper, we focus only on the rendering process, and thus the game loop we will consider in the experiments will focus its operations in dynamically change positions and characteristics of models and lights in the virtual scene. By excluding the intervention of external modules, we are trying to get a better assessment of the performance improvement.

The client processes the raw data streamed by the server and render the game world, performing on the GPU of the player's device the final stages of the rendering pipeline. The client handles also the player input, forwarding it to the server. A diagram reporting client's and server's respective tasks is depicted in Fig. 1.

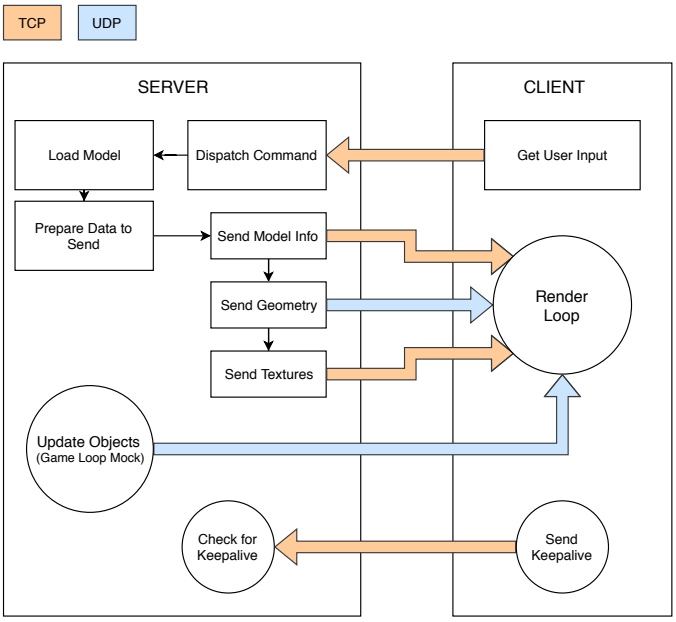


Fig. 1. Scheme of client's and server's tasks and their relation

A. Implementation details

The main development language for the project is C++14. The project was developed using a cross-platform approach, as it has been tested in both Windows and Linux environments.

We have adopted Vulkan [14] as graphics library for our rendering engine. Vulkan is a graphics API specification released in 2016 as the logical *successor* to OpenGL. Vulkan aims to be much more low-level than OpenGL, and more suited to exploit the multi-core and multi-threading characteristics of modern computational architectures. Indeed, it allows for greater fine-tuning and control over almost every aspect of the application, like e.g., synchronization, and memory management.

The client and server portions of the engine share a common set of resources they handle. These resources include 3D models, textures, shaders, cameras and lights. While, for the most part, the client's internal representations differ from the server's (since they usually operate in different ways on each resource), some of them are shared between the two. In particular, they share resources in a serialized state. These are data structures encoding the binary format of the TCP data transmitted through the network. Upon serialization, the sender only needs to cast the data structure to a byte array, whereas the receiver simply casts back the received bytes to the correct data structure, to be passed on to the rendering subsystem.

B. Network Communication

The client-server communication logic can be schematized as follows:

1) The server listens on a TCP socket for incoming connection requests. On connection, an handshake is performed to initiate the application-level protocol.
2) Non-critical data transfer is performed out-of-band by opening two UDP flows.
3) The server can start pushing data to the client immediately. In our demo we initiate the transmission with all lights in the scene.
4) In any moment, the client can request a specific asset, or a set of assets. We use a *send on demand* service model to better test the architecture. In this preliminary work, the client will request only 3D assets and the server will provide them in a FIFO fashion to better mimic a local rendering pipeline. Nevertheless, it is possible

to implement a server-side policy for asset priority and scheduling.

5) The server will send a model as follows: a description summary including materials will be send in-band (via the TCP channel). Then, geometry data (vertices and indices) are streamed out-of-band using the UDP channel. Finally, after geometry has been sent, textures are sent in-band via TCP. This approach is intended to encompass perceptual requirements for game experience.
6) The server also push the model's transform data via UDP every time it changes.
7) After sending a model, the server can mark it as *active* and start simulating it in the game loop.

Let's now see briefly when UDP and TCP transmissions are used.

UDP transmission is used for messages that can be received out of order, or later, and for messages that can be lost without hampering the system. For the first message type we also use acknowledgements (*ACK*) packets on the reverse UDP flow. The sender will periodically resend all packets that were not acknowledged after a timeout. Via the UDP channels we send geometry data, transforms updates, and light updates. Currently, we request ACKs only for geometry data, since transforms and lights update are occurring periodically and re-transmission will be redundant. Sequence numbers are used to allow message reordering and re-transmission.

Messages requiring delivery reliability are sent over the TCP in-band socket. All connection-control messages belong to this category: *handshakes*, *keep-alive*, and *disconnection* requests. Another message type transmitted over TCP is the *resource exchange* coordination message. whenever the server wants to send one or more assets to the client (e.g., a texture, a material, or a model), it first sends a resource exchange message. This gives the client the opportunity to prepare a memory buffer to store incoming data. When the client is ready to receive the resources, it sends an acknowledgement back to the server, which then proceeds to send the actual resources out-of-band via the UDP channel. Last but not least, the TCP channel is also bringing *client requests* to the server. In a client request, the client explicitly asks the server to send models on demand.

C. Multi-threaded architecture

As already mentioned, in order to accomplish its tasks, three sockets are in use: an inbound UDP socket, an outbound UDP socket and a bidirectional TCP socket. These network endpoints are used by various threads in different ways. More specifically, the server uses a total of six threads, while the client uses five.

On server side, the *main thread* is responsible for starting the server and running the main *game loop*. Upon starting, the main thread allocates the TCP socket endpoint and spawns a *primary TCP thread* waiting for incoming clients. On client connection, the primary TCP thread will spawn a *secondary TCP thread* to manage all in-band data for the newly connected client. Together with the secondary TCP thread, the primary TCP thread will also spawn a *UDP incoming thread* and a *UDP outgoing thread* to manage out-of-band messages. Out-of-band messages, as already mentioned, are ACKs for the UDP incoming thread and assets plus updates for the UDP outgoing thread. Finally, one last thread will be responsible to keep-alive messages and clean data structures on server disconnection.

On the client side, we have a specular architecture with one less socket. The client has no use for the secondary TCP thread since there will be no incoming connections requests. Inside the client, the primary (and only) TCP thread will take care of the in-band messages. As a matter of fact the UDP outgoing thread will generate ACKs and the UDP incoming thread will receive assets, while the TCP thread will perform connection management and send asset requests to the sever. Of course we will also have a last thread responsible for the keep-alive messages.

This multi-threading architecture is intended to make the system more modular and pave the way to implement a completely distributed game engine in the future.

V. Performance Analysis

To test the performance of the proposed architecture, we have set up a demo scene, and performed some preliminary tests. We employed a Linux desktop workstation as server, and a Windows 10 laptop (equipped with a NVIDIA GeForce 960M graphic board) as client. Both the machines were connected to the same LAN. We benchmarked the transmission and rendering of two freely available models: the *Nanosuit* and the *Sponza palace*, both originally created by Crytek. Both models have color textures, and use *specular* and *normal* maps. The *Nanosuit* model consists of 14262 vertices, and its memory allocation is approximately $18\,\mathrm{MB}$ ($1\,\mathrm{MB}$ for geometry, $17\,\mathrm{MB}$ for textures). The *Sponza palace* model consists of 239670 vertices with a memory allocation of approximately $96\,\mathrm{MB}$ ($16\,\mathrm{MB}$ for geometry, $80\,\mathrm{MB}$ for textures). Figure 2 shows the models rendered by our prototype.

A. Client framerate

As explained in Sec. IV-B, the server initially sends the lights present in the scene, and then proceeds to send the geometry and textures of an asset. In the test, we measured the client framerate by sending an increasing number of dynamic (i.e., moving) lights before each of the test models. All the measurements refer to a fullscreen scene rendered at 1920x1080 (FullHD) resolution, with framerate locking disabled. For both *Nanosuit* and *Sponza palace*, we found that the client is able to achieve 60 frames per second when the number of moving lights in the scene is below 200.

In this experiment, we measure the frame rate on stable regime, i.e. while no heavyweight TCP data is being transferred. That is because, during TCP resource updating, both the client and the server's frame rates jitter due to spikes in either I/O or computation, and this effect pollutes the measurements. While this is an hill-desirable effect, it is also a strong indication that a priority policy must be established between modules and kernel operations, and that protocol

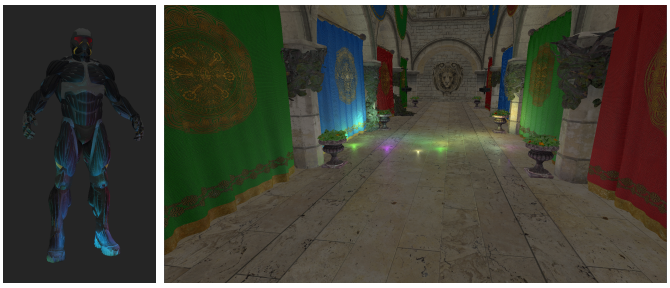


Fig. 2. A sample of the *Nanosuit* and *Sponza palace* models rendered by our engine (original models courtesy of Crytek.)

processing delay is critical for real-time even on the small scale.

B. Response Delay

To measure Response Delay, we initially measured the network bandwidth between the server and the client, and found it to be about $68\,\mathrm{Mbps}$. This is taking system and transport-level overheads out of the picture. We took this as our "baseline" bandwidth and measured the response delay under these conditions. From the moment the server starts to send the mesh data, we measured the response delay in two different intervals: the first, called *geometry delay*, reports when all the geometry information is arrived on the client, while the second, called *complete transmission delay*, reports when all the assets linked to the geometry (e.g., textures) are also received.

We artificially reduced the network capacity by implementing a token bucket in the network middleware of our prototype. Then, we tested the response delay under other 3 different bandwidths levels. The results of the measurements are reported in Tab. I. The largest contribution to the response delay is provided by textures exchange. This is expected, as on average textures have a much bigger size than the models' geometry.

Unfortunately, as we can see in the table, timing is not at all to the level of a fast-paced game. Nevertheless, a number of lessons can be learned from this experiment. First of all, compression is mandatory. Sending uncompressed assets even on a large network pipe is unsustainable in term of user experience. From what we observe, the network is such a bottleneck that the time saved on the client (due to the fact data processing is not required) is not enough to compensate the transmission delay. Performance gain achieved by placing packets payload directly in the GPU seems to be way smaller than the additional transmission time required by a larger data plus the operating system management overhead. On the other hand, compression – as we know it – is optimized for sequence of raster images. There is a whole new world to explore to identify and reduce redundancy from streaming 3D assets. This includes, but is not limited to, evaluate the perceptual redundancy of a 3D mesh.

The application of data-agnostic compression algorithms such as *DEFLATE* or *LZMA* applied to the *Nanosuit* model in the demo is summarized in Tab. II. As it can be observed, compression rate and timing are not fitting the bill for a fast-paced game. Delay is not going to be reduced in a substantial way. The result here is that either we design a mesh-specific compression algorithm or, yet again, we bet on reducing perceptual redundancy.

TABLE I
RESPONSE DELAY (SECONDS) WITH VARYING BANDWIDTHS

Bandwidth	*Nanosuit*		*Sponza palace*	
	geometry	*complete*	*geometry*	*complete*
68 Mbps	< 0.5	(1.8 ± 0.2)	(1.3 ± 0.2)	(5.0 ± 0.2)
40 Mbps	< 0.5	(4.2 ± 0.3)	(5.4 ± 0.2)	(32.0 ± 0.3)
24 Mbps	< 0.5	(7.9 ± 0.3)	(8.4 ± 0.2)	(52.0 ± 0.3)
16 Mbps	< 0.5	(11.5 ± 0.2)	(8.7 ± 0.2)	(52.0 ± 0.3)

TABLE II
COMPRESSING NANOSUIT DATA (ORIGINAL SIZE 18 MB)

Compression Algorithm	*Compressed Size (MB)*	*Compression Ratio*	*Time taken (seconds)*
DEFLATE	11	1.63	1.5
Burrows–Wheeler	11	1.63	1.7
LZMA	6.6	2.72	6.4

C. Network Traffic

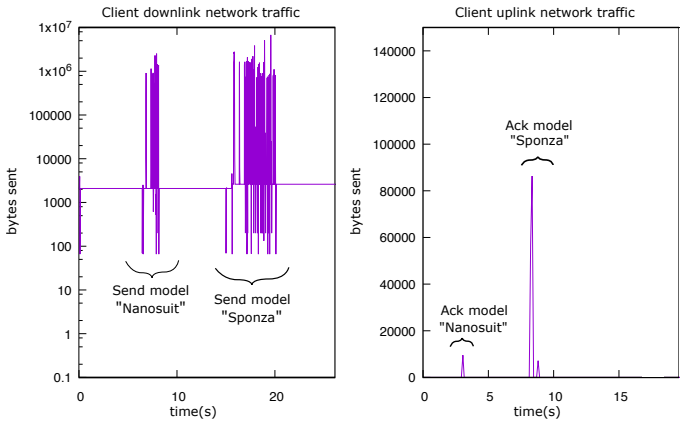
In order to measure the network usage, we used *tcpdump* to capture all packets sent by the server to the client. Unsurprisingly, the downlink traffic is much higher than the uplink one, as it includes full model geometry and textures, while the uplink traffic pretty much only includes UDP *ACKs* (client outbound TCP messages are negligible in size).

As it can be observed in Fig. 3a, we experience traffic bursts with spikes corresponding to sub-elements of each 3D mesh. With the progressive reduction of network capacity, as reported in Tab. I, spikes are spread over time in a *Constant Bit Rate* (CBR) fashion until two consecutive bursts will compete for medium access and the network is saturated.

Beside having an additional hint to compression, there is a clear link between time dependency and complexity. Even if assets are transmitted in advance to populate a local buffer, no constant prefetch time can be assumed and the anticipation time must depend on content complexity (i.e., memory size) and the scene context. Each single 3D object should have its own priority related to scene context and its functionality for the gameplay. As an example, a purely decorative object can be sent later to the client if resources are limited; that is not going to hamper the player experience since there will be no interaction with the player.

VI. CONCLUSION AND FUTURE WORK

In this work, we designed and implemented a real-time rendering system where the assets are retrieved in real-time from a remote server. The client is able to render a complex scene with a high number of polygons and dynamic lights at a satisfying frame rate. The engine has been built with



(a) (b)

Fig. 3. Network traffic over time with unconstrained bandwidth.

modularity and expandability in mind, so it can effectively work as a basis to expand upon, or as a module to integrate into a bigger project. In particular, our demo "game loop" on the server may be replaced by a more sophisticate loop updating not only the objects' transformations and the dynamic lights' parameters, but also several subsystems such as an animation system, a physics simulation, etc. Moreover a priority-based scheduling middleware can be easily inserted in order to verify resource-optimization policies.

The proposed approach reduces the computational burden on the server, which is not required to perform rendering, while allowing a thin client because all the receiver needs to do is transferring packets' payload to the GPU. This approach can enhance scalability for standard cloud gaming solutions based on video streaming.

While a constant or sudden throttling in bandwidth does not affect perceived video quality (because we are not streaming video frames), on the other hand a larger bandwith usage is required due to the uncompressed nature of the GPU-ready data structures. It turned out that the time saved on data manipulation is not compensating enough transmission delay, and the adoption of compression techniques seems to be mandatory as well as a pre-fetching strategies, that might be context- and complexity-dependent.

We envision different approach to improve this research in the future. An optimization strategy may consist in using different level of details for models and textures: the client can receive (and start rendering) in a very short time a low-fidelity version of each object. This object can then be improved step by step in the following seconds or minutes, when more detailed versions are received.

Another opportunity could be to mix video streaming, for elements not fundamental to the gameplay, with object streaming, for which hi-fidelity is required. Streamed objects will always be rendered at full quality overimposed to a background scene with lower quality but using much less resources.

REFERENCES

[1] W. Cai, R. Shea, C.-Y. Huang, K.-T. Chen, J. Liu, V. C. M. Leung, and H. C.-H., "A survey on cloud gaming: Future of computer games," *IEEE Access*, 2016.

[2] A. Di Domenico, G. Perna, M. Trevisan, L. Vassio, and D. Giordano, "A network analysis on cloud gaming: Stadia, GeForce Now and PSNow," *Network*, 2021.

[3] K.-T. Chen, Y.-C. Chang, H.-J. Hsu, D.-Y. Chen, C.-Y. Huang, and C.-H. Hsu, "On the quality of service of cloud gaming systems," *IEEE Transactions on Multimedia*, vol. 16, no. 2, pp. 480–495, 2014.

[4] J. Saldana and M. Suznjevic, "QoE and latency issues in networked games," *Handbook of Digital Games and Entertainment Technologies, eSpringer*, 2015.

[5] M. Carrascosa and B. Bellalta, "Cloud-gaming: Analysis of google stadia traffic," *Computer Communications*, vol. 188, pp. 99–116, 2022.

[6] D. Maggiorini, L. A. Ripamonti, and G. Cappellini, "About game engines and their future," in *EAI GOODTECHS*, 2015.

[7] D. Maggiorini, L. A. Ripamonti, E. Zanon, A. Bujari, and C. E. Palazzi, "SMASH: A distributed game engine architecture," in *IEEE ISCC*, 2016.

[8] J. Bulman and P. Garraghan, "A cloud gaming framework for dynamic graphical rendering towards achieving distributed game engines," in *USENIX HotCloud*, Jul. 2020.

[9] S. Bischoff and L. Kobbelt, "Streaming 3d geometry data over lossy communication channels," in *Proceedings. IEEE International Conference on Multimedia and Expo*, vol. 1, 2002, pp. 361–364.

[10] F. Meng and H. Zha, "Streaming transmission of point-sampled geometry based on view-dependent level-of-detail," in *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.*, 2003, pp. 466–473.

[11] X. Liao, L. Lin, G. Tan, H. Jin, X. Yang, W. Zhang, and B. Li, "Liverender: A cloud gaming system based on compressed graphics streaming," *IEEE/ACM Transactions on Networking*, vol. 24, no. 4, pp. 2128–2139, 2016.

[12] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, *Real-Time Rendering*. A K Peters/CRC Press, 2018.

[13] J. Gregory, *Game engine architecture*. CRC Press, 2019.

[14] G. Sellers and J. a. Kessenich, *Vulkan® Programming Guide: The Official Guide to Learning Vulkan®*. Addison-Wesley Professional, 2016.