An Architecture for 2D Game Streaming Using Multi-Layer Object Coding

Diego Cordeiro Barboza, Débora C. Muchaluat-Saade, Esteban Walter Gonzales Clua, Diego Gimenez Passos
Computing Institute
Universidade Federal Fluminense (UFF)
Niterói, Brazil
{dbarboza, debora, esteban, dpassos}@ic.uff.br

***Abstract*— Advances in cloud computing have enabled systems where the game logic and rendering are processed on a server and streamed to a thin client. Meanwhile, there are still many challenges to provide games through the cloud without latency issues. This work proposes a novel technique to reduce encoding times and bandwidth usage on cloud gaming systems. We propose LACES (LAyer Caching gamE Streaming), an API that separates 2D game objects into multiple layers and allows the cloud server to encode and stream only modified parts of the frame. Using our technique, we managed to reduce the encoding size on the server by 99,79% and the average streaming size to the client by 96%, with no changes to the video quality or resolution and with no noticeable user input delay.**

Keywords - cloud computing; streaming; game streaming; games;

I. INTRODUCTION

In recent years, a variety of services are becoming available through remote access due to advances in cloud computing [1]. As computer networks become more reliable and provide faster response rates, combined with larger bandwidth, multimedia applications started benefiting from this environment [2]. Streaming services started providing content directly to the user without the need for downloading and installing the content before it is consumed. Currently, many different media types are provided through streaming, such as movies, music and games [3].

However, there are still some challenges to provide these services to a wide range of users with different client configurations, operating systems, devices and network capabilities. For example, Netflix (www.netflix.com) stores 120 versions of the same movie to accommodate different screen sizes, bitrates and codecs on every device supported by the platform [4]. Netflix's approach works because it is streaming pre-recorded video. For live video or real-time interactive applications, any necessary adaptation on the content has to be done on the fly. The cost for transcoding a video in real time can be very high. Twitch (www.twitch.tv), for example, only uses Adaptive Bitrate (ABR) to transcode the raw video received from the streamer into multiple live video streams for the top channels in popularity. For other less popular channels, it just prepares the raw video stream and delivers it to users, avoiding transcoding tasks costs [5].

Video games are resource-demanding applications, which often require fast processors for handling its logic, physics and systems, and powerful graphics processors for rendering. Furthermore, games are always evolving on the technological side, which usually makes new games more and more hardware demanding. On upgradeable platforms, such as personal computers, users have to handle higher requirements by upgrading their systems. On other platforms with shorter life cycles, such as mobile phones, developers may choose to drop support for older hardware that cannot handle all the game's requirements. In addition, on game consoles that usually have longer life cycles, developers have to limit their games to the current hardware until a new platform is launched.

Compatibility between multiple platforms is also a great concern in game development. On consoles, a new platform release usually means dropping support for the entire legacy library. This is a problem for user satisfaction but mainly from game preservation [6] [7]. As older platforms are harder to find in the market, games for those platforms become inaccessible.

Backwards compatibility is also a great issue on console and PC games. Microsoft has added an Xbox 360 emulator for Xbox One [8] and Sony is currently providing PS3 games through the cloud to PS4 users with the PlayStation Now service [9]. On PC, support for older games is not always available, even though solutions like DOSBox [10] try to solve this issue.

In all those cases, most problems rely on the client-side. Console manufacturers cannot release a new upgraded platform every year, since the user base is built along many years during the platform life. On mobile and PC clients, higher requirements can reduce the amount of players that are able to access the game. Most of those issues could be solved by offloading to a remote server part of the game's processing. This way, if a game requires more processing power, the game provider could update its servers, but users would still access the platform with a standard device.

Cloud computing can bring solutions to those issues. The process of delivering games through cloud computing is known as cloud gaming. These systems allow users to remotely access and play games without specific or dedicated hardware. Mobile phones, low-powered laptop or desktop computers, smart TVs and other devices can be used to access games through the network and consume them as a video stream that was generated in the cloud. Since most of the workload with this model is on the server-side, developers can create games with lower concerns about performance and compatibility on the client-side. This allows users to run the same game on devices with different processing power and operating systems with little, if any, changes on the server-side. The client device just needs to be able to connect to a server, receive the video stream that is displayed to the user, and stream the user input back to the server.

Cloud gaming also allows new business models for providing games to users based on a service model, instead of a product. Just like music and video streaming services,

games can be provided on a subscription basis, where users pay a monthly fee to have access to a library of streaming games. Other non-conventional ways to deliver streaming content are also possible, for example, using a local server to stream games to airplane or ship passengers, or hotel guests.

Even though cloud gaming provides ways to reduce compatibility issues between different platforms, allows users to access games on multiple devices, helps to solve game preservation issues and opens new business models, there are still some challenges to make this kind of service accessible and functional for all users. Games are a real-time interactive media and, as described by Huang et al. [11], users expect both high-quality video and low interaction delay. For a good user experience, some requirements such as a tolerable latency, related to a short round-trip time for the client input to be processed by the server and the result be displayed to the user, and high-quality low-delay video streaming must be met.

On cloud gaming systems, most of the processing workload is on the server-side. This means that for every game frame that is generated and streamed to the client, the server needs to process the game's logic, render its graphics output, encode this output using a video codec and stream it. Maintaining a system that is able to perform all of these tasks, while keeping a low response delay, is challenging.

Many works have proposed improvements in the streaming process to try to reduce delays and server workload [11] [12] [13] [14] [15]. There is a great focus in the literature on accelerating stages of the video encoding process, such as motion estimation, or applying techniques to reduce the video bitrate. Using those techniques, authors are trying to reduce the server encoding complexity, which leads to lower processing delays and amount of data sent through the network, reducing transmission delays.

In this work, we propose LACES (LAyer Caching gamE Streaming), an API that introduces a novel technique applied directly on the application level that uses knowledge about the game to separate and group together data that needs to be encoded and transmitted to the client. Our approach focuses on using layer-caching techniques that can reduce both server workload and network usage. Using knowledge about how the game is structured, we propose combining game elements into different game layers that can be encoded and streamed separately. By caching each encoded layer, we can reuse this data for the following video frames and if a layer has not been modified, no new data needs to be sent to the client nor be encoded by the server.

With LACES we expand our initial tests presented in [16] that were based on GamingAnywhere [11] and served as a validation for the multilayer streaming proposal. In this paper, we present a full game streaming architecture and an API to support multilayer game development and streaming. Also, we apply caching on the server-side and the possibility to stream translation updates for frames that were just moved on the screen.

Because unchanged layers do not need to be encoded, this approach results in a workload reduction on the server-side. In addition, the layers that have not been modified are also cached on the client-side to be displayed to the user, so they do not need to be streamed until they are changed, which also leads to lower network bandwidth usage.

In this work, we also propose an API that game developers should use to adapt the game to benefit from our cache-based streaming proposal. The API core allows the developer to specify which game elements belong to which layer. Even though this API requires some work from the developer, his knowledge of the game is very important to provide a better user experience. For instance, the developer knows which elements are just background decoration and could have their encoding quality reduced if needed, without causing much impact to the user and the game. This means that the developer should specify how objects should be separated and prioritized, but no further game logic adjustments are needed. In practice, this can be easily implemented into any game engine, using tags to mark and classify these elements.

The remainder of this paper is structured as follows. In Section 2, we present a related work survey and compare the proposed techniques found in the literature with our approach. In Section 3, we present an in-depth view of our proposed technique and API. This approach is evaluated in Section 4, where we present our experimental results. In Section 5, we present our conclusions and suggestions for future work.

II. RELATED WORK

Improvements on video streaming for cloud gaming have been proposed by different works in the literature, but the focus is usually on solutions for accelerating video encoding stages [14] [15], especially motion estimation, or to reduce the video bitrate by reducing the scene complexity [12]. Some older works also focus on streaming geometry (the game's 3D models) [17] and graphics commands [13], splitting the graphics processing between client and server. On most cases, the goal is to reduce the server load, which is a major concern for cloud-based systems with multiple users, and bandwidth usage.

To the best of our knowledge, the idea of splitting the game rendering into multiple layers that are completely independent, using contextual information, has not been explored in any of previous works.

Video games usually present rich graphics output with a lot of visual content to the user. When an image with more detail is encoded, the result is a frame with larger bitrate. Since not all objects presented on screen at a single time are crucial to the user's experience, Hemmati et al. [12] propose a content adaptation method that selects and remove objects that are less important to the player's current activity from the current frame. This results in an image with less detail, which is faster to encode and has lower bitrate.

Hemmati et al. [12] propose to remove objects based on player's activity within the game and Rahimi et al. [18] discuss how the object selection is done and the impact it has on player's experience. The technique results in a significant bitrate and encoding time reduction, which can be even higher when considering the results for cloud servers with a massive number of concurrent users. However, there is an important setback, which consists that the objects selected for removal are completely deleted from the scene. The idea is that the developer may specify which elements are important for each activity. However,

even if an object is not important from the gameplay point-of-view, visual elements are still important for immersion and having objects popping on the screen while game activities change can lead to odd experiences to the user. For instance, removing an object that is not an obstacle when the player is shooting will not affect the gameplay when the shooting activity is performed, but will result in a noticeable object pop in and out to the user.

Even though most cloud gaming systems are based on video streaming, some works approach this problem in a different way. Eisert et al. [13] tries to solve the video encoding cost on the server by completely removing the video encoding from the process. Instead, graphics commands issued to the graphics hardware on the server are intercepted before they are sent to the video card and encoded and streamed to the client. The client application must decode and process these graphics commands locally to render the game and present the output to the user.

That solution greatly reduces the workload on the server-side and can help to achieve lower interaction delays, because graphics commands can be streamed as soon as they are issued, instead of waiting for a full frame to be ready. With lower workload, servers could provide services for more concurrent users and still keep a good quality of service. The server can also translate graphics commands between APIs before sending them to the client. For example, if a client only supports the OpenGL API [19] and the game runs on Windows machines, the server may intercept the Windows DirectX commands and stream the equivalent OpenGL to the client. This helps solving compatibility issues between multiple platforms.

On the other hand, Eisert et al.'s approach [13] offloads a large portion of the processing to the client. The game's logic still runs in the server, but the client is responsible for processing the graphics commands and, ultimately, rendering the frame that is displayed to the user. This increases the hardware requirements in the client and makes this solution not feasible on low-powered devices. In addition, all the game's geometry, game objects and textures need to be transferred to the client before the rendering starts. This can result in high bitrates while these objects are being loaded and bitrate peaks during the game while new objects need to be loaded or the number of graphics commands varies. With the video streaming approach, the bitrate variation depends only on the frame complexity.

Li et al. [17] present a similar geometry streaming approach for online multiplayer games. The engine proposed by the authors enables progressive game download where the game content is downloaded as necessary, based on player's location within the game's world. This approach also allows game objects to be prioritized based on their importance and available bandwidth.

Noimark et al. [14] propose a technique to reduce the complexity of streaming remote walkthroughs to handheld devices. This technique is based on segmenting objects into background and foreground layers and varying quantization levels for each layer. To represent the background information, Noimark et al. use an MPEG-4 feature that allows representing an image as a mosaic that is reconstructed at the client. Modified parts are updated using global motion compensation data sent by the server. When these transformations are not enough to represent the necessary camera operations, such as zoom, a new full image is encoded and streamed from the server to the client.

The motion estimation stage of MPEG-4's encoding is very costly. To avoid this, Cheng et al. [15] propose an algorithm to replace the usual motion estimation process and calculate motion vectors directly from data collected from the scene. Using pixel position, depth buffer information and camera projection matrix, the algorithm is able to acquire the motion vectors by un-projecting a pixel in 2D space back to the 3D space and re-projecting this 3D point to the 2D space using the camera configurations from the previous frame. This enables the acquisition of a motion vector for a pixel between two consecutive frames.

Cheng et al.'s method can present a great speed up on motion vector acquisition and fails only at some specific cases, such as pixel occlusion. The algorithm can only be used on static scenes, where interactivity comes only from camera movement. If the camera is static and objects are moved, the algorithm will not be able to detect the pixel movement. While this technique can present good results for visualization applications, it is not suitable for most games, due the massive presence of dynamic objects with constant changes on shape, color, position, scale and rotation.

Some works instead of tackling a specific game streaming issue, present a complete streaming system that could be used by developers and researchers to experiment with cloud-based game streaming systems. The most prominent example of this kind of system is GamingAnywhere, an open source cloud gaming platform proposed by Huang et al. [11]. GamingAnywhere provides a communication infrastructure between client and server using network standards and is designed for extensibility, where other researchers could easily plug in new techniques or replace parts of the original streaming process. As a cloud-based game streaming system, all of the game's processing is executed at GamingAnywhere's servers and the video output is captured and streamed to the client that needs to decode and display the video stream to the user. Even though GamingAnywhere presents some good results, it does not implement methods for content adaptation based on system capabilities, game content information or ways to reduce server workload and network usage.

In this work, we propose a cloud gaming streaming architecture based on layer caching. The core of this method consists on separate game elements into different variable-size layers that are encoded and streamed separately. Therefore, instead of encoding a full resolution video frame, the server checks which layers were changed since the last update and encodes only those ones. This allows the system to prioritize elements that are more important in the game, such as the main character. It also enables reuse of previously encoded data between multiple users. Moreover, it is possible to vary quality settings for each layer separately when it is necessary, instead of changing settings for the whole image. While the constraint of developer dependence exists, since he/she has to mark or tag layers and elements, the optimization practice is very common in game development.

Programmers and designers are always trying to optimize models, illumination processes and draw calls.

Techniques such as those proposed by Eisert et al. [13] and Cheng et al. [15] have been proved to work under specific circumstances but do not really fit for a cloud gaming system providing multiple and different games as a service. Cheng et al.'s method, even with great speed up results, only works with static scenes where the only movement comes from the camera. Any other movement in this context would invalidate the results. Eisert et al.'s way relies too much on offloading the rendering stages to the client. This could work on some more powerful systems that are able to process the necessary graphics commands, but would also create some compatibility issues (the server would have to stream different commands for different architectures) and could be too demanding for low-powered devices. In our work, we use a similar approach to Eisert et al.'s for updating cache on the client, when it is not completely invalidated. In these scenarios, when a layer is moved, rotated or scaled, for example, but aside from this the original image remains unchanged, we stream commands to the client to apply a transformation (which may be a simple translation or a complex image warping) locally. However, different from the Eisert et al.'s graphics commands, these transformations are much less frequent and are processed directly from the client application, instead of issuing graphics commands for the graphics hardware.

Our approach allows the server to prioritize content that is more important as proposed by Hemmati et al. [12], but without removing the lower priority objects from the final rendering. Instead, these objects could be encoded with lower quality and spatial/temporal resolution.

Instead of relying on the games' depth buffer to segment the scene into separate layers, as proposed by Noimark et a.l [14], we introduce an API that allows the developer to specify and control which objects belong to each layer, how many layers should be in the game (instead of Noimark's background/foreground proposal), how the objects should be combined together and which layers should be prioritized. We also extend Noimark's proposal by allowing dynamic adjustments of spatial and temporal resolution, instead of only controlling quantization levels.

III. LAYER CACHING GAME STREAMING API

A typical cloud gaming system is usually composed by two main components, which are the server and client modules, as presented by Shea et al. [20]. Most of the overall workload is on the server-side. This component is responsible for processing user input, handling game logic, rendering video output, encoding this output and streaming video to the client. The client component needs to decode and present the video received through the network, and capture user input to send it to the server. Client tasks are less demanding, which allow low-powered devices to run client applications and receive the result of complex high-end games from the network.

Both client and server applications need to perform some tasks to make the cloud gaming system work. These tasks form a loop that organizes and connects all the sub-systems to be perceived by the user as a single system that is working to provide the game that is being consumed at runtime. These tasks are outlined in Fig. 1. At the client-side, user input and network feedback data are gathered and the received video decoded and presented to the user. At the server-side, user input updates the game's logic and network feedback is used to make necessary quality adjustments. The game is then rendered and its output is encoded and streamed to a client.

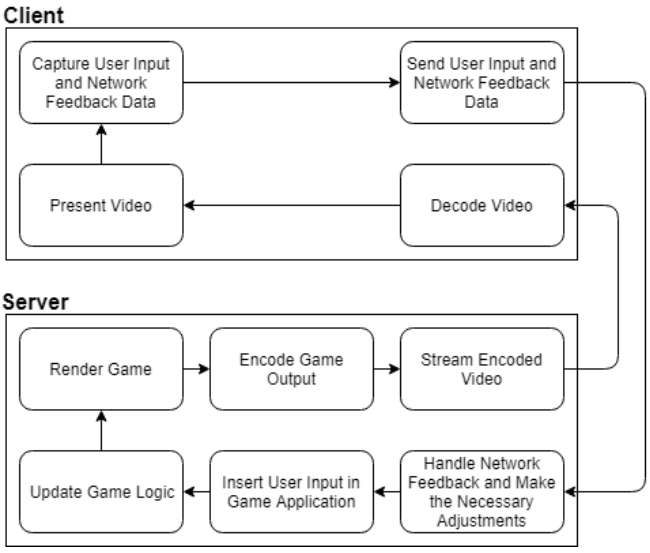

Figure 1. Tasks processed on client and server to provide a cloud-based game.

Based on this common architecture, our work proposes using the concept of layers to handle the game content separately, instead of always streaming a full video frame. We use knowledge about the game to combine different objects into these layers and handle them as needed. Less important layers may use a lower bitrate or be updated less frequently, while other layers containing relevant game elements can be prioritized. This is specified by the game developer.

The core idea about using game layers is that not all content is updated at the same time within a game and not all content has the same importance. A game background could be static for several frames while the player character may change its animation at every update. If the background has not changed and the game has means to detect it, we could employ some technique to avoid this image from being encoded and sent to the client. This greatly reduces the encoding tasks at the server and the streaming size sent to the user.

This way, instead of transmitting a single video stream with fixed size and quality parameters, we apply multiple streams, each with its own spatial/temporal resolution and quality parameters. This allows the video streamer to prioritize objects that are more important and handle them separately.

For example, in some games, the main character controlled by the player occupies only a fraction of the screen. However, when using the conventional streaming method, the character's region on the video cannot have better quality or be updated more frequently than the rest of the video. Using layers, a video stream with the size just large enough to enclosure the player character can be set

up and allow the player to be updated on the client without streaming the full resolution video.

Fig. 2 presents an overview of how a game scene could be split into multiple separate layers. It is up to the developer to specify which objects should be combined together using its knowledge about the game. In this example, we have identified four layers: background (A), foreground (B), player (C), and enemies and objects (D). Background and foreground represent large static elements that will be updated less frequently, but can not be placed on the same layer because some other elements appear between them. The enemies and objects will be updated more frequently but they use less screen space. And since this layer is composed by small elements, its quality parameters could be reduced without causing much impact on player experience, if needed. Finally, the player layer is where the main character is. This layer usually has the user focus all the time and should keep the highest possible quality. On the other hand, the player character is only a small element on the screen, which means that, even if at a high frame rate, updating this layer should be a lot cheaper than updating the whole screen at every frame.

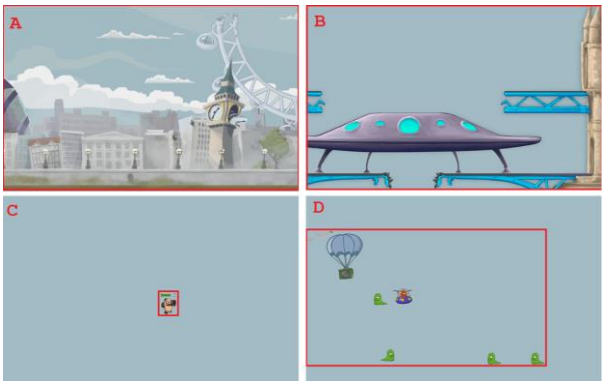

Figure 2. The game objects are separated into different layers that are handled separetely and can be updated and streamed with no dependency of the others (this example is based on Unity Tower Bridge game).

Another advantage of this layer technique is removing motion estimation from the process. As seen before, motion estimation is a very costly stage on video encoding [15]. However, instead of relying on the generic motion estimation algorithms that are designed to work with any kind of video, we use the streaming API to detect when objects within the scene have changed and need to be updated. This usually happens when objects have been added or removed from a layer, have moved, rotated or scaled, or have changed any visual property, such as an animation frame or its color. Using our proposed API, all these changes are detected when the commands to change any of these properties are issued and the streaming server is able to react accordingly, by re-encoding and updating the cache for that specific layer. Then again, even if we are on a complex scene with many elements on the screen, only the layers containing the changed objects will need to be updated.

In Fig. 3 we outline how a typical cloud gaming architecture could be modified to work with the proposed layer technique. At the cloud server, changes are made starting from the rendering stage. Instead of the regular graphical output, this component outputs multiple separate layers with the game elements. The video encoder then processes each layer separately as needed (layers that have not been updated do not need to be encoded again) and the video streamer sends the encoded layers to the client. At the client side, the video decoder needs the handle of each layer on its own and combine the data received from the network with the previous layers that have been already cached to create the final image that is displayed to the user.

Instead of just decoding and presenting the video, in some cases, the client has to decode multiple videos and store at least the last frame for each layer on a local cache. Therefore, client system processing and memory requirements are slightly increased on this architecture. On the other hand, streamed videos within this architecture will usually have a lower resolution, since they represent only part of the whole game.

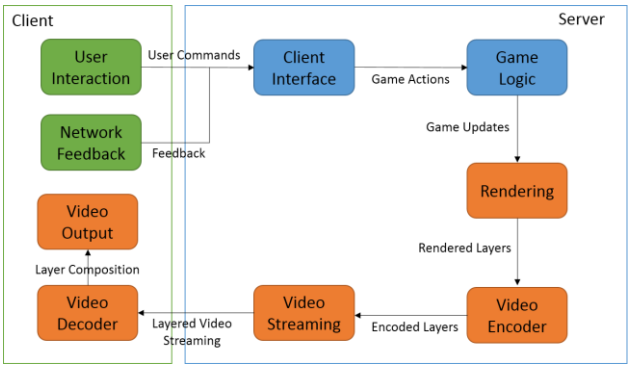

Figure 3. Typical cloud gaming architecture adapted for layered streaming.

We also propose a simple API that works as a basic scene graph and allows the developer to add objects and specify game layers. By using this, any updates to the game objects must go through the API, which allows the streaming application to detect changes that have happened and any new layer that needs encoding.

The proposed LACES API works as follows: an Application component is responsible for managing the whole game. The developer should extend this class and create his own application (MyApplication), which will load game resources, create game objects and specify its layers. Each application has one or more Layers that combine a group of game objects. When a game object is changed, its layer is marked and will be encoded and streamed. Game Objects typically contain the game logic. These entities are usually added to a game to encapsulate a specific behavior, but in our context, game objects should always be attached to elements that have a visual representation. At this point, our API focus only on 2D games, so these Game Objects represent sprites in the screen. Elements that are behavior-only, such as a timer, do not need to belong to a layer, since they will not be streamed to the client.

At the rendering stage, each layer renders its game objects into an individual buffer. The result of a render pass are *N* image buffers ready for cache check, encoding and streaming, where n is the number of layers in the game. Before encoding and streaming an image buffer to a client, the server first checks if changes were made to that layer. If no changes happened, the current image buffer on

the server should be the same as the local cache at the client and no retransmission is needed. If anything changed within a layer, it needs to be updated at the client. Two issues are important for this process: how to check if the layer cache has changed and how to update it at the client level.

Within LACES, every drawable object belongs to a layer, as illustrated in Fig. 2. An object is invalidated and needs to be redrawn in two specific scenarios: it has suffered a transformation (translation, rotation or scale) or it has changed its image (updated its animation frame, for instance). Whenever these situations happen, the game object informs to its layer through the API which type of modification happened. After all objects of a layer have been updated, the layer has enough knowledge to decide whether it needs a complete redraw or it is enough to just send some transformation operations to the client to update its local cache.

If the combined changes for all objects within a layer could be expressed as a series of transformation operations, then the server does not need to encode and stream a new image to the client. Instead, it just sends the transformation operations that will be applied at the client. For example, if all objects within a layer have moved, the server may send a translation command to the client instead of encoding and streaming a new video frame. If these objects have also been scaled, the server will send a scale command and so on.

After determining which layers need encoding, the server must send the updated data to the client. For layers that were invalidated, new frames will be encoded and streamed. The client will receive these frames, store them on a local cache and display them to the user. For the layers that were updated but not redrawn, the server will send the transformation operation to the client, which will process and apply these transformations to its local cache and update the final image displayed to the user.

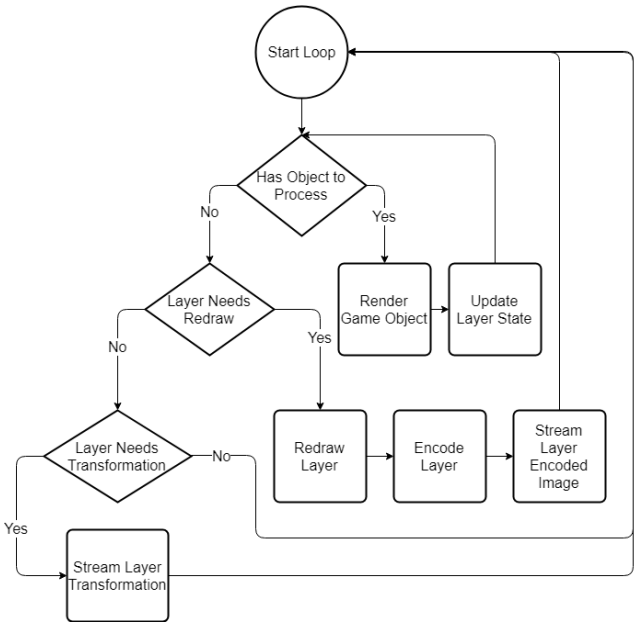

Figure 4. Overall update and render flow for a single layer within the game using LACES.

The flow for updating a single layer is demonstrated in Fig. 4. The same process is repeated for each layer in a game using our API. For every update, we iterate on all game objects within a layer and render them. The render result may update the layer state, if the image has changed (and the layer needs a complete redraw) or if an object has been transformed. If nothing happened, the layer does not requires an update. After all objects are rendered, the current layer state will determine if it needs to be redrawn or not. If the layer needs to be redrawn, the server will use the layer output, encode it and send it to the client. Otherwise, the server will check if the layer needs to be transformed and send the necessary transformation operations to the client. After that, a new loop starts.

IV. EVALUATION

Our proposed API and test application were developed in C++ using FFmpeg [21], Allegro [22] and Simple and Fast Multimedia Library (SFML) [23]. Allegro is used to handle multimedia assets, to detect user input and display graphics output to the client. SFML is used to handle network communications and FFmpeg provides functionality for video encoding at server-side and decoding at client-side.

A core element for LACES is how game layers are defined and used. Grouping together a large amount of objects in a single layer would probably cause this layer to be updated more frequently. Then it would be streamed more frequently, therefore, reducing overall cache usage. On the other hand, having too many layers could also impact the performance, because in this case, the server would process and encode multiple separate layers that the client needs to receive from the network, decode and present to the user.

We have run our tests in different scenarios to demonstrate how the system behaves with different setups, ranging from a single layer – a regular video stream – to each object within its own layer in a game with a few dozens of visual objects.

The test application is a Breakout game (Fig. 5) with the following elements: background image, ball, player's paddle and target bricks, with a resolution of 800x600. In this game, the player controls a paddle in the bottom of the screen and has to hit the ball and destroy the bricks in the top. Each of these elements can be included in a separate layer or combined in the same layer. For the tests presented in this paper, four different scenarios were used:

(a) one layer: all objects in the same layer (without using cache, this represents a regular video stream analogous to other game streaming solutions)

(b) three layers: (1) background and bricks, (2) ball, (3) player

(c) four layers: (1) background, (2) bricks, (3) ball, (4) player

(d) n layers: (1) background, (2) ball, (3) player, and (27) each brick on its own layer

The first scenario (a) will update the image layer every time an object is changed, which in this case should be every frame the ball moves, for instance. Scenario (b) will update layer 2, if the ball is changed, layer 3, if the player is changed, and layer 1, if a brick is changed. Scenario (c) will update layer 3, if the ball is changed, layer 4, if the

player is changed, and layer 2, if a brick is changed. Layer 1 (background) is static in this game and will not be updated. Finally, scenario (d) allocates a separate layer for each game object, including each separate brick. This means that when a brick is changed, only that small portion of the screen needs to be updated. However, this also means that client and server are now handling multiple low-resolution videos instead of a few larger ones.

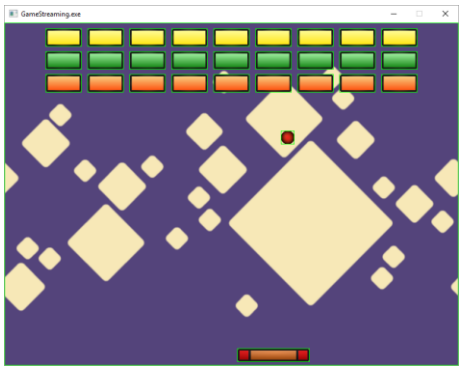

Figure 5. The Breakout game used as test application.

We ran the application on these different scenarios for one minute of gameplay and collected data to compare how the application behaves with several layers and using the cache technique or not. Our metrics for these tests are: total video encoded size on the server and number of encoded frames, total streamed size and number of streamed frames, number of translation frames, and encoding time.

Using LACES cache technique, we expect a great reduction in encoding and transmission. Each layer frame is stored after its initial encoding and, on the next time it is needed, we first check if it exists in the cache to avoid encoding it again. If a layer has an animated object (a character walking, for example), each animation frame is encoded the first time it should be presented and, on the next time, the cache is used. If caching is ignored, each time the layer changes, it needs to be encoded again.

When cache for a layer exists and it has just been moved on screen, we can send a translation frame to update that layer's position in the client, instead of sending the entire encoded layer frame. This packet contains just a few bytes identifying the layer and specifying the new layer position. We measure this information to show how many times the cache technique allows the application to skip frame encoding and streaming.

The number of layers and the size of each layer have an impact on how much time the server application spends encoding each frame. In the single layer scenario (a), this is the total time to encode each frame that is sent to the client. In the other scenarios (b), (c) and (d), this time is the accumulated time for encoding all layers. If cache for a specific frame exits, it is not necessary to encode it, therefore, the total encoding time is greatly reduced.

We run the Breakout game for one minute for each test simulating a gameplay session. To avoid differences caused by user input, the player's paddle has been programmed to follow the ball position instead of waiting the player's input to move.

A. Test scenarios with static game objects

Table I presets the results for test case (a) where all objects are placed in the same layer. This is the worst case for this proposal, since it does not provide the opportunity to reuse previous frame cache. This way, the results for using or not using cache are very close and we could consider them equal within an error margin (less than 3% encoded bitrate difference).

TABLE I. TEST CASE (A): ALL OBJECTS IN THE SAME LAYER

	Using Cache	Not Using Cache
Encoded Frames	4694	4850
Encoded and Streamed Size	749243 Kb	769891 Kb
Average Encoded bitrate	12486.6 Kbps	12829.5 Kbps
Streamed Frames	4694	4850
Encoding Time	38.6 s	39.8 s

Better results with our LACES proposal start to appear when we separate objects into multiple layers, each with its own video stream. In scenario (b), we create three layers: (1) background and bricks, (2) ball and (3) player. Background and bricks are static elements, so they are placed in the same layer. Layer 1 will change only when a brick is destroyed. The ball and the player's paddle will move around separately, so each gets its own layer.

In the scenario presented in Table II, we are actually handling three separate video streams, one for each layer. But each video is large enough only to cover all objects within the layer, which means the video for the ball object, for example, is only 24x24 pixels. In addition, these video layers are not being encoded and streamed all the time. If the objects in a layer have not changed their visual state between frames, that layer does not need to be updated and, if the objects have just moved, we can issue a translation command to update the layer position in the client.

Using cache in scenario (b), the server needed to encode only 17 frames (one for the ball, another for the player and the remaining 15 for each time a brick is destroyed). This greatly reduces the encoding and streaming size and bitrate. Notice that, since most of the time the player's paddle and the ball are just moving on screen, without actually changing their images, the server is able to send a translation update to the client without encoding or streaming a new frame.

TABLE II. TEST CASE (B): THREE SEPARATE LAYERS FOR BACKGROUND AND BRICKS, BALL, AND PLAYER PADDLE

	Using Cache	Not Using Cache
Encoded Frames	17	11604
Encoded Size	2279 Kb	674303 Kb
Average Encoded bitrate	37.9 Kbps	11236 Kbps
Streamed Frames	17	11604
Stream Size	2279 Kb	674303 Kb
Average Stream bitrate	37.9 Kbps	11236 Kbps
Streamed Translations	13884	0
Encoding Time	0.12 s	33.4 s

Of course, not using cache in scenario (b) has a huge impact on performance, because the server would encode and stream three separate video layers for every game frame. This is not a recommended scenario for our technique. Results when "Not Using Cache" are presented in the tables just for comparison reasons.

A better comparison though would be scenario (b) using cache and scenario (a) not using cache. This way, we can see how our technique compares to a standard video stream. Using three layers and storing separate caches for them, we managed to reduce the encoding size on the server 99.7% (from 769891 Kb to 2279 Kb) and the average stream size 99.7% (from 12829.5 Kbps to 37.9 Kbps). This has a large impact not only on the network usage and necessary bandwidth, but also on the server load. Encoding so many frames reduced the total encoding time 99.6% (from around 39.8 seconds to less than 1 second).

A key element of our proposal is fine-grained streaming where only the necessary parts of the image are encoded and sent to the client. If all game objects are placed in the same layer, this means that any change makes the whole frame invalid and the server needs to encode a new frame. If objects are separated into multiple layers, only the layers that have modified objects will need re-encoding. In Table III, we show the effects of adding more layers to the game. Scenario (b) has three layers and every time a brick is removed, a new frame of the remaining bricks and the background is encoded. In scenario (c), bricks are combined in a different layer from the background and when a brick is changed, the background does not need to be re-encoded. Finally, in scenario (d), each brick is independent from the others and when brick is changed, nothing happens to the other bricks' layers.

TABLE III. COMPARISON BETWEEN THE SCENARIOS WITH 3 (B), 4 (C) AND N (D) LAYERS

	(b)	(c)	(d)
Encoded Frames	17	16	39
Encoded Size	2279 Kb	2007.2 Kb	330.7 Kb
Average Encoded bitrate	37.9 Kbps	33.4 Kbps	5.5 Kbps
Streamed Frames	17	16	39
Stream Size	2279 Kb	2007.2 Kb	330.7 Kb
Average Stream Size	37.9 Kbps	33.4 Kbps	5.5 Kbps
Streamed Translations	13884	12460	10578
Encoding Time	0.12 s	0.05 s	0.01 s

The number of frames in scenario (d) is larger because each brick is encoded separately at least once and there are 27 bricks in the scene. Whenever a brick is destroyed, a new frame is created to represent this new state. The other frames correspond to the background, ball and player, each encoded one time and updated with a translation command when necessary.

Separating bricks in different layers makes it possible to modify a brick without any effect to the other bricks or any other element in the scene. These results in a bitrate as low as 5.5 Kbps, because the only thing that is streamed after the initial state is the removal of each brick's layer when they are destroyed.

B. Test scenarios with animated game objects

The previous scenarios work well as a benchmark for our technique and allows us to test how it behaves compared to a standard video stream with different number of layers, but they do not reproduce the behavior of an actual game. We also developed another scenario that is closer to the expected behavior of a typical 2D game. Usually, each game object in a 2D game is composed of some discrete frames that are looped to create the illusion of animation and movement, as illustrated in Fig. 6. We modified the Breakout game from the previous tests and added animations to the player's paddle, the ball and the bricks. These animations are played at different speeds and updated separately. Using our cache technique, the goal is to encode each frame just once and when it is needed again, the stored cache is used.

Figure 6. A character in a 2D game is made of multiple animation frames to create the illution of animation and movement.

In Table IV, we compare the standard single-layer video stream with no cache (a), with the cached video stream with three (b), four (c) and several layers (d) as we made in the previous tests, but now using animated game objects.

TABLE IV. COMPARISON BETWEEN STANDARD VIDEO (A) AND THE SCENARIOS WITH 3 (B), 4 (C) AND N (D) LAYERS WITH ANIMATED OBJECTS

	(a)	(b)	(c)	(d)
Encoded Frames	4554	96	95	245
Encoded Size	726964 Kb	13871.9 Kb	13009.6 Kb	1503.4 Kb
Average Encoded bitrate	12115.1 Kbps	231.1 Kbps	216.8 Kbps	25 Kbps
Streamed Frames	4554	1070	1072	7264
Stream Size	726964 Kb	44165.7 Kb	43079.4 Kb	28849.8 Kb
Average Stream bitrate	12115.1 Kbps	736 Kbps	717.9 Kbps	480.7 Kbps
Streamed Translations	0	12885	12239	9566
Cache Hit/Miss	0/4554	974/96	977/95	7019/245
Encoding Time	38 s	0.72 s	0.3 s	0.05 s

In these scenarios, each layer is updated whenever a child game object changes its current animation frame. Here we introduce a new metric: cache hit and miss. When the server needs to send a new frame to the client, first it checks if that frame was previously encoded and is stored on the server cache. If the cache exists, it is a cache hit. This means that the server does not need to encode it again. The data retrieved from the cache is sent directly to the client.

This has the potential for greatly reducing encoding time on the server. For example, in these tests, the player's paddle has four animation frames. After the fourth frame, the server would not need to encode anything else for this game object, it just reuses the cache. In addition, if the server is providing a service for multiple users, the cache created for the first user could be reused for new players, which can bring huge performance gains in cloud-based systems that usually work with massive amount of users.

In our tests, we observed that using more layers leads to better results. With each object separated in its own layer (d), we managed to reduce the average encoded size 99,79% (from 12115.1 Kbps to 25 Kbps) and the average streamed size 96% (from 12115.1 Kbps to 480 Kbps), compared with the regular video stream (a). We streamed a total of 7264 frames (the sum of all layers during the test) but only 245 of those were actually encoded. The remaining 7019 frames were successfully fetched from the cache.

Also, notice that for moving objects, such as the ball and the player's paddle, sometimes their positions change, but the animation frame stays the same. In these cases, the server sends just a translation frame with the new position for that layer. This reduces the amount of data that is sent to the client. On our tests, this happened 9566 times.

In Fig. 7 and Fig. 8, we present a comparison of average bitrate for scenarios (b), (c) and (d). With static game objects, the server is encoding every frame that needs to be sent to the client, so encoded and stream bitrate are the same. With animated objects, when a frame that was previously encoded is needed, the cache data is used and that frame is not encoded again and the server workload is greatly reduced. Fig. 8 shows in more detail how a large amount of data that is sent to the client is reused and do not need to be encoded more than once.

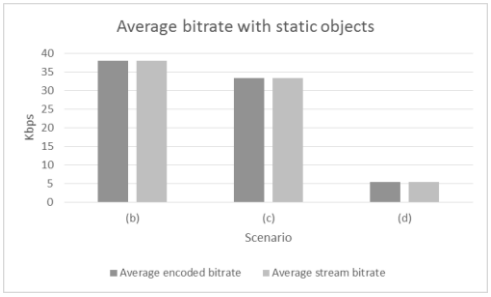

Figure 7. In the scenarios with static objects, everytime a frame is sent to the client, it is also encoded. So, there is no cache reuse from previous frames.

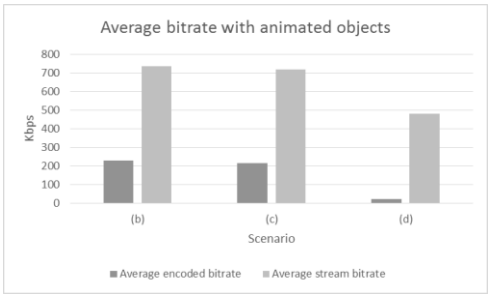

Figure 8. In the scenarios with animated objects, when a frame is sent to the client, the cache will be used if it exits and no new encoding will be necessary. This way, the server is able to encode less data than what is actually streamed.

C. *Input delay*

Using multiple layers and the cache technique proposed in our work might introduce some new complexity in a game streaming system that could result in a larger input delay that could ruin the user experience. The total input delay is the total response time between a key press at the client-side, its processing at the server-side, network round-trip time and the result presented to the player. Since we are testing the input delay added by our technique, we made the tests locally on a same machine. This way, network delays, which would apply to any streaming service, were disregarded.

In our tests, we have stored a timestamp for each user input on the client and sent an id with the input message to the server. The server then adds the latest id whenever a new frame is sent to the client. When the client receives a new frame, which will update the game's state from the player's point-of-view, it uses that id to compare the temporal difference between when the input was sent and a frame generated by that input was received.

In Table V we compared average, minimum and maximum input delay for scenarios with no cache (a), with the cached video stream with three (b), four (c) and n layers (d) using animated game objects. We tested some intermediary cases using n layers (with 12 and 21, in addition to 30), to measure how the delay grows compared to a larger amount of layers. Since in our previous tests each line of blocks has 9 blocks, we removed one line at a time for this test. We also present the standard deviation as a measure of how spread these values are. We ran the application in each scenario and collected 1000 inputs that were entered in the client, sent to the server and received back in the client to compare the timestamp difference.

TABLE V. INPUT DELAY COMPARISON BETWEEN STANDARD VIDEO WITH NO CACHE (A) AND THE SCENARIOS WITH 3 (B), 4 (C) AND N (D) LAYERS WITH ANIMATED OBJECTS

	(a)	(b)	(c)	(d)		
				12	21	30
Average delay	39 ms	10 ms	11 ms	10 ms	13 ms	15 ms
Min delay	29 ms	5 ms	6 ms	6 ms	7 ms	8 ms
Max delay	46 ms	55 ms	46 ms	36 ms	56 ms	88 ms
Standard deviation	2,6	4,9	3,7	2,5	4,6	8,8

The average input delay for the scenarios using cache is around 10-15 milliseconds with peaks around 90 milliseconds in some cases. Compared with the standard video with no cache, which in average presented a 39 milliseconds delay, we believe the multi-layer and cache processing should not add a significant input delay to the overall streaming system, in fact, the delay average delay is much lower using cache, as seen in Fig. 9.

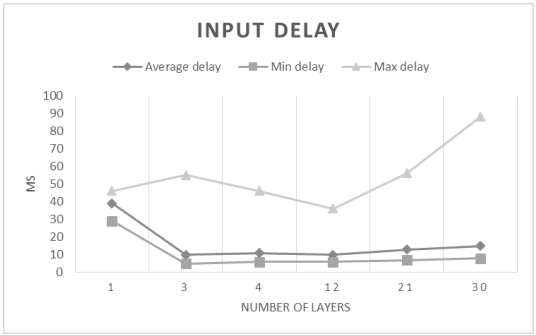

Figure 9. Input delay for the tested scenarios. Notice that the average delay is much lower in all the cases that are using our cache techinique (with 3 or more layers) compared to the standard video (with a single layer).

V. CONCLUSIONS

In this work we presented LACES, an API that allows a cloud game server to split game objects into multiple video layers and handle them separately, resulting in great performance improvements when compared to a standard video stream.

With our technique, we were able to reduce the average encoding size on the server 99.79% (from 12115 Kbps to

just 25 Kbps). This reduction comes from the reuse of previously encoded frames that were stored on the server cache. In cloud gaming systems that usually have a large amount of concurrent users, the combination with a much lower encoding time and the reuse of encoded frames between different users can result in a major performance boost.

We also applied our technique to identify when elements within a layer were just moved but were not redrawn. When this happens, that layer can be moved in the client with a simple translation message, instead of encoding and sending a new frame. This allows a great reduction on network bandwidth usage between server and client. With a standard video stream, everything that is encoded is sent to the client. With our layer-cached stream, we send new frames only when needed. This way, we managed to reduce the average streaming size to the client 96% (from 12115 Kbps to 480 Kbps).

As future work, we intend to reuse data generated for one user to others. This would gradually increase the cache size and cache hits would become more constant, reducing even more the amount of data the server needs to encode to provide its service to a specific user.

We also intend to propose dynamic video quality encoding adjustments to keep a good user experience. Since game elements are already separated into layers, our idea is to prioritize layers with more important objects (such as the player character) and adjust the quality of the other layers as needed. This way, if we need to use less network bandwidth, for example, we could reduce the background resolution or the frame rate of enemy animations.

Moving LACES to a game engine, such as Unity, Unreal or GODOT is an approach that would greatly improve its usage, since it would reduce the amount of work need to adapt games intended to run on this proposed cloud environment.

It would also be possible to apply these techniques to stream layered outputs for emulated older consoles that already work with multiple object layers for their sprites, backgrounds and other elements.

Finally, using machine learning, we believe it would be possible to estimate which layers and frames a user will most likely need given the current game state and stream them using the available bandwidth. Using this strategy, it would be possible to further improve input delay, reduce network peaks and increase client-side cache hits.

ACKNOWLEDGMENT

This work was supported by CAPES, CNPq and FAPERJ.

REFERENCES

[1] A. Fox et al., “Above the clouds: A Berkeley view of cloud computing,” Dept Electr. Eng Comput Sci. Univ. Calif. Berkeley Rep UCBEECS, vol. 28, p. 13, 2009.

[2] M. Jarschel, D. Schlosser, S. Scheuring, and T. Hoßfeld, “Gaming in the clouds: QoE and the users’ perspective,” Math. Comput. Model., vol. 57, no. 11–12, pp. 2883–2894, Jun. 2013.

[3] W. Zhu, C. Luo, J. Wang, and S. Li, “Multimedia Cloud Computing,” IEEE Signal Process. Mag., vol. 28, no. 3, pp. 59–69, May 2011.

[4] Netflix, “Complexity In The Digital Supply Chain,” 2012. [Online]. Available: http://techblog.netflix.com/2012/12/complexity-in-digital-supply-chain.html. [Accessed: 13-Jul-2019].

[5] K. Pires and G. Simon, “DASH in Twitch: Adaptive Bitrate Streaming in Live Game Streaming Platforms,” 2014, pp. 13–18.

[6] P. Gooding and M. Terras, “‘Grand Theft Archive’: A Quantitative Analysis of the State of Computer Game Preservation,” Int. J. Digit. Curation, vol. 3, no. 2, pp. 19–41, Dec. 2008.

[7] National Library of Australia, Guidelines for the preservation of digital heritage. 2003.

[8] Microsoft, “Xbox One Backward Compatibility,” 2019. [Online]. Available: http://www.xbox.com/en-us/xbox-one/backward-compatibility. [Accessed: 13-Jul-2019].

[9] Sony, “Playstation Now,” 2019. [Online]. Available: https://www.playstation.com/en-us/explore/playstation-now/. [Accessed: 13-Jul-2019].

[10] DOSBox, “DOSBox,” 2019. [Online]. Available: http://www.dosbox.com. [Accessed: 13-Jul-2019].

[11] 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 (MMSys), 2013, pp. 36–47.

[12] M. Hemmati, A. Javadtalab, A. A. Nazari Shirehjini, S. Shirmohammadi, and T. Arici, “Game as video: bit rate reduction through adaptive object encoding,” in Proceeding of the 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), 2013, pp. 7–12.

[13] P. Eisert and P. Fechteler, “Low delay streaming of computer graphics,” in 15th IEEE International Conference on Image Processing (ICIP), 2008, pp. 2704–2707.

[14] Y. Noimark and D. Cohen-Or, “Streaming scenes to MPEG-4 video-enabled devices,” IEEE Comput. Graph. Appl., vol. 23, no. 1, pp. 58–64, Jan. 2003.

[15] L. Cheng, A. Bhushan, R. Pajarola, and M. El Zarki, “Real-time 3D graphics streaming using MPEG-4,” in Proceedings of the IEEE/ACM Workshop on Broadband Wireless Services and Applications (BroadWise’04), 2004, pp. 1–16.

[16] D. C. Barboza, D. C. Muchaluat-Saade, and E. W. G. Clua, “A real-time game streaming optimization technique based on layer caching,” in Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE, Las Vegas, NV, 2015, pp. 714–719.

[17] F. W. B. Li, R. W. H. Lau, D. Kilis, and L. W. F. Li, “Game-on-demand:: An online game engine based on geometry streaming,” ACM Trans. Multimed. Comput. Commun. Appl., vol. 7, no. 3, pp. 1–22, Aug. 2011.

[18] H. Rahimi, A. A. N. Shirehjini, and S. Shirmohammadi, “Context-aware prioritized game streaming,” in IEEE International Conference on Multimedia and Expo (ICME), 2011, pp. 1–6.

[19] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane, OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3, 8th ed. Addison-Wesley Professional, 2013.

[20] R. Shea, Jiangchuan Liu, E. C.-H. Ngai, and Yong Cui, “Cloud gaming: architecture and performance,” IEEE Netw., vol. 27, no. 4, pp. 16–21, 2013.

[21] FFmpeg, “FFmpeg Libraries for developers,” 2019. [Online]. Available: https://www.ffmpeg.org/. [Accessed: 13-Jul-2019].

[22] Allegro, “Allegro - A game programming library,” 2019. [Online]. Available: http://liballeg.org/. [Accessed: 13-Jul-2019].

[23] L. Gomila, “Simple and Fast Multimedia Library (SFML),” 2019. [Online]. Available: http://www.sfml-dev.org/. [Accessed: 13-Jul-2019].