Content-aware Video Encoding for Cloud Gaming

Mohamed Hegazy
Simon Fraser University
mehagzy@sfu.ca

Khaled Diab
Simon Fraser University
kdiab@sfu.ca

Mehdi Saeedi
Advanced Micro Devices, Inc.
Mehdi.Saeedi@amd.com

Boris Ivanovic
Advanced Micro Devices, Inc.
Boris.Ivanovic@amd.com

Ihab Amer
Advanced Micro Devices, Inc.
Ihab.Amer@amd.com

Yang Liu
Advanced Micro Devices, Inc.
Yang.Liu1@amd.com

Gabor Sines
Advanced Micro Devices, Inc.
Gabor.Sines@amd.com

Mohamed Hefeeda
Simon Fraser University
mhefeeda@sfu.ca

ABSTRACT

Cloud gaming allows users with thin-clients to play complex games on their end devices as the bulk of processing is offloaded to remote servers. A thin-client is only required to have basic decoding capabilities which exist on most modern devices. The result of the remote processing is an encoded video that gets streamed to the client. As modern games are complex in terms of graphics and motion, the encoded video requires high bandwidth to provide acceptable Quality of Experience (QoE) to end users. The cost incurred by the cloud gaming service provider to stream the encoded video at such high bandwidth grows rapidly with the increase in the number of users. In this paper, we present a content-aware video encoding method for cloud gaming (referred to as CAVE) to improve the perceptual quality of the streamed video frames with comparable bandwidth requirements. This is a challenging task because of the stringent requirements on latency in cloud gaming, which impose additional restrictions on frame sizes as well as processing time to limit the total latency perceived by clients. Unlike many of the previous works, the proposed method is suitable for the state-of-the-art High Efficiency Video Coding (HEVC) encoder, which by itself offers substantial bitrate savings compared to prior encoders. The proposed method leverages information from the game such as the Regions-of-Interest (ROIs), and optimizes the quality by allocating different amounts of bits to various areas in the video frames. Through actual implementation in an open-source cloud gaming platform, we show that the proposed method achieves quality gains in ROIs that can be translated to bitrate savings between 21% and 46% against the baseline HEVC encoder and between 12% and 89% against the closest work in the literature.

CCS CONCEPTS

• **Information systems → Multimedia streaming**.

KEYWORDS

Cloud Gaming, Content-based Encoding, Video Streaming

ACM Reference Format:
Mohamed Hegazy, Khaled Diab, Mehdi Saeedi, Boris Ivanovic, Ihab Amer, Yang Liu, Gabor Sines, and Mohamed Hefeeda. 2019. Content-aware Video Encoding for Cloud Gaming. In *10th ACM Multimedia Systems Conference (MMSys '19), June 18–21, 2019, Amherst, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3304109.3306222

1 INTRODUCTION

Video games nowadays contain complex graphical scenes, light and shader effects as well as interactions based on physics and complex calculations. Playing such games at high quality requires installing expensive high-end graphics cards at end user devices. Moreover, the heterogeneity of user devices has pushed the gaming industry towards developing customized versions of the same game for each family of devices; thus, increasing the cost and time to market for these complex games [5]. Cloud gaming has emerged to alleviate these costs for end users and the gaming industry.

There is currently a substantial interest in cloud gaming from industry and academia. The size of the cloud gaming market is projected to be US$4 billion dollars in 2023, up from US$1 billion dollars in 2017 with a CAGR of 26.12% [4]. Many major companies offer cloud gaming services, including Sony's PlayStation Now [36], NVIDIA's GeForce Now [11], LiquidSky [29], Google's Project Stream [16], and Microsoft's Project xCloud [7]. Furthermore, these cloud gaming service providers (CGSPs) are supported by numerous other companies offering hardware and software products, such as the AMD's ReLive gaming streaming system [1], and the hardware encoding capability of AMD's GPUs in Steam Link [12].

At its essence, cloud gaming moves the sophisticated game logic and rendering from devices at end users to servers deployed in cloud data centers. This means that servers need to encode the rendered game frames and stream them to clients. This, however, imposes significant bandwidth requirements on the CGSP, especially for popular, graphics-rich video games with thousands of concurrent users. For example, the minimum bandwidth that CGSPs require from each client ranges from 5 Mbps [29, 36] to 15 Mbps [11], while the recommended bandwidth is between 20 and 25 Mbps [11, 29].

The goal of this paper is to improve the perceptual quality of the streamed video frames, given a target bandwidth requirement. This is a challenging task in cloud gaming, because such an environment imposes strict requirements especially on latency and quality fluctuations. For latency, games require high responsiveness, since high latencies make players become out of sync with the server and can cause them to lose in the game which can drive them away from cloud gaming. Previous studies, e.g., [10], showed that some latency-sensitive games require latency as low as 50 milliseconds, while most games cannot tolerate a latency above 100 milliseconds. This low latency requirement also restricts the amount of buffering a client can have [18]. The maximum amount of data to be buffered is at most one frame for low latency applications [42], as opposed to regular video streaming, e.g., services offered by YouTube and Netflix, which can buffer multiple frames or even seconds [33]. In addition, since games have frequent scene changes, encoders will generate more bits at these scene changes. This may result in frames of significantly different sizes, introducing rate fluctuations and possible stalls in the video streams. Furthermore, the low latency requirement and the scale of current clients do not allow utilizing complex content analysis tools to optimize the encoding process.

In this paper we propose a Content-Aware Video Encoding (CAVE) method for cloud gaming, which optimizes the quality by allocating different amounts of bits to various areas in each video frame. CAVE has two main steps. First, it assigns weights to blocks in video frames based on the importance of these blocks from the perspective of players. It, then, allocates bits to blocks based on their weights, while meeting the low-latency requirement of cloud gaming and achieving consistent quality of the encoded frames.

Unlike most previous works, e.g., [30, 38], we have designed CAVE for the state-of-the-art HEVC encoder [37], which by itself achieves much higher compression ratios compared to the previous encoders. We have implemented and integrated CAVE in the open-source GamingAnywhere cloud gaming platform [17]. We conducted an extensive empirical study with various video gaming segments, and measured multiple quality metrics, bitrate savings, and processing overheads imposed by CAVE. We compare CAVE to the base HEVC encoder and the closest work in the literature [38], referred to as RQ (Rate-Quantization) method, which we implemented on top of the HEVC encoder. Our results show that CAVE consistently outperforms both the base HEVC encoder and RQ by wide margins. For example, CAVE can reduce the bitrate needed to achieve the same quality in the most important areas (regions of interest, ROIs) in the frames by up to 46% compared to the base HEVC encoder running with the most recent rate control model (i.e., the $\lambda-$domain model [25]). Compared to RQ, the gain is even higher, because the proposed content-aware rate control algorithm in CAVE is more accurate than the one used in RQ. Also, CAVE is able to run in real-time without affecting the latency requirements of cloud gaming.

The remainder of the paper is organized as follows. In Section 2, we discuss the related works in the literature. In Section 3, we present the details of the proposed method. In Section 4, we describe our implementation and experimental study. We conclude the paper in Section 5. In the Appendix, we describe the structure of our source code (publicly available) and the steps needed to reproduce the results in this paper.

2 RELATED WORK

At a high level, optimization of cloud gaming systems can be divided into two main areas [5]: (i) Cloud Infrastructure and (ii) Content and Communications. The first area includes problems such as allocation of server resources to clients to maximize the user's QoE and minimize the cost incurred by cloud gaming providers [14]. It also includes proposing new architectures for cloud gaming systems such as introducing edge servers to reduce the latency perceived by clients [8]. The second area includes various optimizations for the compression methods of gaming content as well as adaptive transmission methods to cope with the network dynamics [15]. The work in this paper belongs to the second area.

The second optimization area can further be divided into two categories. The first category tries to reduce the amount of bits needed to transmit and render the graphical structures of games. For example, the work in [28] simplifies the 3D models of the game on the server and sends the simplified models to be rendered on the client device. The work in [9] constructs a base layer of the graphical structures, which is sent to the client device. It then encodes the difference between the full quality and base layer as an enhancement layer, which is sent to clients if there is enough bandwidth. The second category in this area does not manipulate the graphical structures of the content. It rather optimizes the encoding of the resulting frames to be transmitted to the clients. Methods in this second category are the most commonly used by cloud providers, as they do not require changes to the internals of the games. CAVE belongs to this specific category and we will thus describe it in more detail.

Multiple works in the literature have been proposed to optimize the quality of the encoded game video streams. In [30], an average of the importance and the depth of pixels are used to distribute bits in the frame and enhance their quality for an H.264/AVC encoder. A disadvantage of this approach is that the ROI might end up with a lower quality than a non-ROI, if the ROI happens to be far from the virtual camera. In [38], an ROI-based rate control method relying on a Rate-Quantization model is devised for H.264/AVC. We improve upon their work by using the latest HEVC encoder and proposing a better weight assignment which is configurable based on the desired discrepancy in quality between ROIs and non-ROIs.

A game attention model is introduced in [2] by combining a top-down approach based on the current activity in the game and a bottom-up approach based on a saliency map to reduce the bitrate by giving less important areas a higher quantization parameter (QP). However, this work does not perform rate control in the sense that it tries to use a lower bitrate than the target bitrate. This is done by dividing the frame into different levels of importance and assigning different QPs to them. As a result, the produced bitrate will be lower than the bitrate resulting from assigning all the areas a QP value corresponding to the highest level of importance. In our work, we try to improve the quality of ROIs under the same bitrate. Also, the saliency models rely on expensive computations and do not always capture the actual user's attention. A control-theoretic algorithm is proposed in [22] to provide ROI-based rate control for an H.264/AVC encoder. Unlike our proposed method, this work assumes having only a single ROI in the middle of the screen and varies its size based on the drift between the target and actual bitrates.

Eye tracking data from clients are used in [19] to enhance the quality of ROIs. This approach assumes that clients have eye tracking devices at their disposal, which is not realistic as these devices are expensive and are not relevant to the game. The impact of video encoding parameters such as the frame rate and bitrate on the user's QoE is studied in [35]. This study concluded that different encoding configurations should be employed with different game genres. ROI-based encoding techniques were proposed in [26, 31] for HEVC. However, these works target video conferencing applications, thus their methodology for weight calculation may not be suitable for cloud gaming as the content is significantly different.

Finally, an important component of the optimization of game video streams is controlling the bitrate. The relationship between the distortion and the resulting bitrate can be modeled using one of the following models:

- Q-domain R-D [6]: creates a relationship between the bitrate and the quantization parameter.
- ρ-domain R-D [39]: creates a relationship between the bitrate and the percentage of transformed coefficients with a value of zero after quantization.
- λ-domain R-D [25]: creates a relationship between the bitrate and the slope of the R-D curve. This model was adopted in HEVC [24] as it has shown higher accuracy in rate control over the older pixel-wise unified R-Q model [6].

In this work, we utilize and extend the λ-domain model to support optimizing the quality of game video streams while meeting the strict latency requirements of cloud gaming.

3 PROPOSED CAVE METHOD

3.1 Overview

We consider a cloud gaming system consisting of server(s) and clients. Servers are deployed in data centers of public or private clouds. Servers are instantiated as virtual machines (VMs), where each instance runs the gaming engine to render and encode the game actions generated by clients. Specifically, upon receiving a client's input, the server processes this input and renders the output of the game as raw video frames. These frames are then encoded and streamed to the client. Clients, on the other hand, run simple functions on end user devices such as tablets, smartphones, and PCs. In particular, clients capture the players' actions from the control devices, e.g., keyboards, touch screens, and game-pads, and send them to the server. And upon receiving the encoded video frames from the server, clients decode and display them to the players. No computation-intensive tasks, e.g., graphics rendering, are performed on clients' devices. The high-level architecture of cloud gaming is shown in Figure 1.

The proposed content-aware video encoding method, CAVE, is to be integrated with cloud gaming servers to optimize the quality of video streams delivered to gaming clients and reduce the bandwidth consumption of such streams. Achieving these goals is quite challenging in cloud gaming, because of the following requirements:

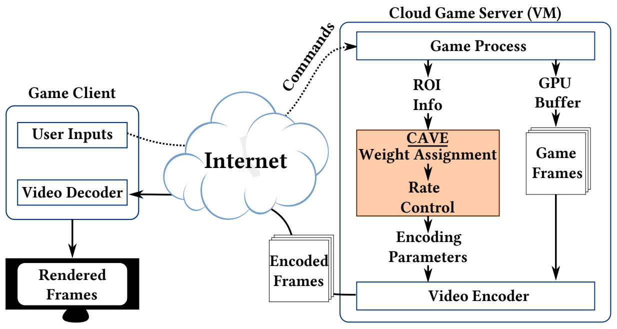


Figure 1: High-level architecture of cloud gaming platforms incorporating CAVE.

- *Low Latency.* Cloud video games are highly interactive applications, where timing of events is critical. This requirement not only restricts the time allowed for the server to render and encode video frames, but also restricts the time needed to transmit such frames. In other words, the resulting bitrate should not vary significantly, which puts additional constraints on the rate control model used to allocate bits to frames.
- *Scalability.* Cloud gaming servers are designed to serve thousands of users concurrently, where video streams could be customized for individual users. Thus, the additional resources (memory and CPU) needed by any optimization of the video streams should be minimal. For example, complex image analysis tasks cannot be performed on frames in real time.
- *Modular Design of Cloud Gaming Platforms.* These platforms are complex systems with many software and hardware components. In real deployments, these components come from different sources and are integrated through various interfaces and APIs. That is, in many cases the cloud gaming service provider may not have access to the source code of individual components. For example, the video encoder, whether it is implemented in software or hardware, is typically sourced from a third-party company as a black-box with various APIs to configure and run it. This means that for any encoding optimization method to be practically viable, it needs to interact with the encoder through its exposed APIs and should not assume direct access to the encoder source code.

We design CAVE to satisfy the above practical requirements. At a high level, CAVE is implemented as a software component between the Game Process and Video Encoder in Figure 1. It does not require changing the encoder, nor does it impose high processing overhead on the server. It controls the encoding bitrate while running in real time to meet the low-latency requirements. In addition, CAVE does not maintain or use state to encode successive video frames. This is a desirable feature especially for cloud platforms, in which servers run on VMs that can and do fail. In case of a VM failure, CAVE does not require any state to be transferred to the new VM instance.

The cloud gaming engine performs various operations to produce video frames to be sent to the client. It reads the inputs from the client and feeds them to a running process of the game. The cloud gaming engine captures the game frames from the GPU frame buffer. CAVE uses information about the game's ROIs and computes various encoding parameters to optimize the quality. It then passes these parameters to the Video Encoder, which produces the encoded frames sent the client.

CAVE optimizes the quality by allocating different amounts of bits to various areas (blocks) in each video frame, based on the importance of these blocks to players. Assigning importance to blocks is not easy, especially in video games where frames typically contain rich and complex graphics covering most of the blocks. This complexity makes using saliency-based approaches, e.g., [2], to estimate the importance of various blocks inaccurate, because they rely on visual cues such as color and intensity, which do not necessarily correlate with the player's attention and interaction with the game [32]. In Section 3.2, we present our proposed approach for assigning weights to different blocks, which considers the characteristics of video games and how players interact with them.

CAVE, then, allocates bits to blocks based on their weights. This is a crucial step as the bitrate needs to be carefully controlled to meet the low-latency requirement of cloud gaming, while achieving high and consistent quality of the encoded video frames, given a target bitrate set by the cloud gaming service provider. In Section 3.4, we present our approach for controlling the bitrate, which is designed for the state-of-the-art HEVC video encoder and extends its basic $\lambda-$domain rate control algorithm to meet the requirements of content-aware cloud gaming.

3.2 Weight Assignment

Prior works, e.g., [2], considered a bottom-up approach in assigning weights to blocks. A bottom-up approach relies on the saliency and low-level features, e.g., color, texture complexity, and spatial frequency, of different areas to estimate their importance. As mentioned above, these visual cues do not capture the importance of various objects to players. For example, in a first-person shooter game, the target/enemy is the most important object for the player even if its texture/color is not complex, whereas static and background objects are less important even if they have complex graphics. In addition, generating saliency maps and analyzing visual cues require extensive image processing operations and may not be suitable for latency-restricted cloud gaming.

In contrast, we propose a top-down approach to assign weights to different blocks, which considers the attention of players. That is, our approach assigns higher weights to the most relevant blocks to tasks being accomplished by players. Our approach is inspired by previous studies [13] that show in active search tasks, top-down cues are the most relevant from the user's attention perspective. Identifying relevant blocks is straightforward for game developers, because they know the logic and semantics of the game. Thus, they can expose this information as metadata with the game that can be accessed via APIs. CAVE assumes that the game developer exposes simple information about different objects in each frame. Using this information, one or more regions of interest (ROIs) are defined as bounding boxes containing objects of importance to the task being achieved by the player. For example, in a first-person shooter game, ROIs can include objects such as the enemy characters, health bar and world map of the game. We note that ROIs depend on the semantics of the game and the situation of players at different time instances. In the next section, we proved an example illustrating how the ROI information can be exported.

After defining ROIs, CAVE calculates different weights for blocks inside and outside of ROIs based on a foveated imaging and retinal eccentricity model. Specifically, the intuition behind using foveated imaging is that the spatial resolution of the eye is at its peak at the center of gaze. As we move further away from the center of gaze, the resolution of information delivered by the eye decreases logarithmically. The angle between any perceivable detail and the visual axis of the eye is referred to as the retinal eccentricity angle, which we denote by θ. The fovea is only capable of covering a small visual angle of 2°[23]. An early work [40] developed a model to approximate the sensitivity of the human visual system (HVS) to different areas in an image. This model is, however, fairly complex as it depends on the spatial frequency as well the eccentricity angle. Computing spatial frequency requires applying various filters to the image and is costly as it needs to process every pixel in a frame which affects the end-to-end latency. And as we discussed before, the spatial frequency may not necessarily reflect the importance of various objects to players. We simplify the model in [40] by considering only the retinal eccentricity angle θ. The sensitivity of the HVS in CAVE is approximated as:

$$e^{\tan^{-1}(\theta)} = e^{\tan^{-1}(\frac{-d}{D})} \approx e^{\frac{-d}{D}}, \tag{1}$$

where d is the Euclidean distance between a pixel in the frame and the center of gaze, and D is the diagonal of the frame to eliminate any dependency on the frame resolution. Notice that the value of d/D is between 0 and 1 and the tan^{-1} function on that range can be approximated by the identity function.

CAVE assumes that the center of the bounding box encompassing an ROI is the center of gaze. Based on this assumption, CAVE will assign more bits to an ROI to enhance its quality since an ROI is the most likely area that will attract the user's attention. However, there may be multiple ROIs in a single frame [30], and these ROIs might have different types which implies that they should be assigned different importance factors relative to each other. Therefore, we extend Eq. (1) to support the potential presence of multiple ROIs, as follows:

$$\frac{1}{M}\sum_{i=1}^{M} e^{(-Kd_i)/(F_i D)}, \tag{2}$$

where M is the number of ROIs in the frame, K is a constant scaling factor controlling the desired discrepancy in bit allocation and quality between ROI and non-ROI areas, d_i is the Euclidean distance between a pixel in the frame and the center of the i^{th} ROI and F_i is the importance factor of the i^{th} ROI which is in the range of $0 < F_i \leq 1$. The importance factors F_i can be defined by the game developers based on prior knowledge of the current task or mission in the game and the importance of each ROI in achieving that task. The scaling factor K is a tunable parameter; in our experiments it ranges from 2 to 10.

We calculate the weight of each $N \times N$ block not belonging to an ROI using the sensitivity value in Eq. (2) between its center, i.e., at position $(N/2, N/2)$ relative to its top left corner, and the center of each bounding box encompassing an ROI. For a block belonging to an ROI we assign to it a weight proportional to the scaling factor K and its relative importance F_i to maintain the same quality for the blocks inside the ROI. Based on our experiments the sensitivity values of individual pixels inside a single block do not have a large variance and thus the sensitivity value of the pixel in the center of a block is considered as a suitable representative of the sensitivity value for the whole block in CAVE. We follow this approach to avoid processing every pixel in the frame and to reduce the overhead of the weights' calculation.

In summary, the weight w of a block whose center position is at (x, y) denoted by $b[x, y]$ is determined as:

$$w[x,y] = \begin{cases} e^{-1/(KF_i)} & \text{if } b[x,y] \in ROI_i \\ \frac{1}{M}\sum_{i=1}^{M} e^{(-Kd_i)/(F_i D)} & \text{if } b[x,y] \notin ROI_i \end{cases}, \forall i \qquad (3)$$

3.3 Exporting ROI Information

The ROI information is essential for any content-aware encoding method. One way to obtain ROI information is to use some deep learning tools, e.g., [34], to estimate it, after the game is deployed and used by many players. Although this is possible for popular games with thousands of users from which large datasets can be collected, the time cost can be very high and may not be suitable for real-time cloud gaming.

Another way to obtain ROI information is to export such information during game development. This is not difficult as game developers already expose various other information like error logs to debug game crashes. For example, the ROI information can be stored in the Stencil buffer so that it can be intercepted during the rendering process. The Stencil buffer contains information that can be used for post-processing the rendered pixels in a frame. It is considered as an auxiliary buffer in addition to the essential ones such as the frame (color) and depth buffers. It is used in [30] to hold the importance of every pixel. Processing every pixel in the Stencil buffer to get the ROI information is, however, expensive. We propose a more efficient approach, in which the game developer assigns simple tags to game objects and these tags are used later in run time to extract the ROI information in order to optimize the video coding. The tags can be accessed via an API that the game developer exposes.

To illustrate our approach, we have developed a simple game, based on [21], using the popular Unity game engine. The game consists of a controllable player character that can move around in a bounded arena and collects items from the floor to increase their score. The game is shown in Figure 2 where the player is represented as a black sphere and the collectable items are shown as white cubes. To support CAVE, the game developer would need to create a few tags representing the priority of different objects in the game. For this simple example, two tags are sufficient: player and item. Player has higher priority; therefore, it has a different tag than the item object. Then, the game developer would attach tags to the various objects in different frames. In addition, a simple API to export the objects' coordinates given their tags would be needed. In the Unity framework[1], this API can have the following signature: `List<Rect> getBoundingBoxes(string tag);`.

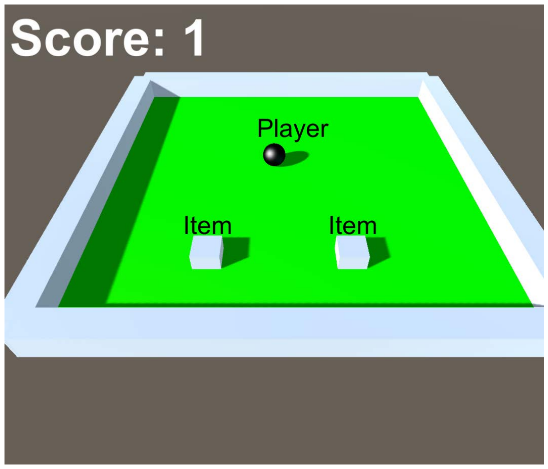


Figure 2: An example of simple game in Unity to illustrate how the ROI information can be exported. The tags above the objects are printed directly by the Unity game engine given the knowledge of the assigned tag to each object.

This API can be implemented by the game developer as follows. The player character in this game is given the tag `"Player"`. Calling the following function in Unity: `FindGameObjectWithTag("Player")` returns a Unity `GameObject` which is the parent of any object in a Unity game. `GameObjects` in Unity, such as the player object, hold as one of their properties the coordinates and bounds in 3D space of that object. By projecting the coordinates of the `GameObject` from 3D space onto the screen's 2D space, the game developer can easily retrieve the bounding box around the object in the screen's 2D space by getting the two farthest points that were projected in the 2D space. This projection is easy as the game developer already has the needed transformation matrices to perform this operation. The game developer can then return a list of bounding boxes (`Rect` objects in C#) around the objects of the corresponding tag when the above API is called by CAVE.

Assuming that the game is instrumented during the development using the simple tags and the availability of the API described above, CAVE can perform its optimization as follows. Given that CAVE has the names of the different tags in the game and their corresponding importance factors (F_i in Eq. (3)), it can then retrieve the bounding boxes and assign weights to various objects in the frame using Eq. (3).

3.4 Rate Control

Cloud gaming service providers strive to achieve good quality of the delivered video streams to players, while minimizing the delivery cost. Thus, they specify an encoding bitrate that achieves a desired target video quality. The rate control component of the encoder allocates bits to blocks and frames within the bit budget of the specified encoding bitrate. The allocation of bits to meet the average encoding bitrate can be done on three levels: GOP (Group

[1]Using the C# version.

of Pictures), frame, and block levels. Content-awareness adds another complexity to the rate control process, as bits are supposed to be allocated to also reflect the importance of blocks, while still meeting the target bitrate and not introducing significant spatial and temporal variations.

As discussed in Section 2, multiple rate control models have been proposed in the literature. We focus on the most recent λ-domain model and extend it to support content-aware encoding for cloud gaming. In the following, we start by presenting the basics of λ-domain model. Then, we describe our extensions to it.

Basic λ-domain Model. The relationship between the bitrate and the λ value of the R-D curve is modeled as [25]:

$$\lambda = \alpha b^{\beta}, \tag{4}$$

where b denotes the number of bits per pixel, and α and β are the model parameters. b represents the target bitrate R independent of the resolution and frame rate, and is calculated as $b = R/(f \times w \times h)$, where f is the frame rate, w and h are the width and height of the frame or the block respectively. α and β are updated after encoding a single frame or single block, depending on the granularity of the rate control, according to the following equations:

$$\lambda_a = \alpha_{old} b_a^{\beta_{old}} \tag{5}$$

$$\alpha_{new} = \alpha_{old} + \delta_\alpha \times (\ln \lambda_t - \ln \lambda_a) \times \alpha_{old} \tag{6}$$

$$\beta_{new} = \beta_{old} + \delta_\beta \times (\ln \lambda_t - \ln \lambda_a) \times \ln b_a, \tag{7}$$

where λ_a is the computed λ value using the actual generated number of bits b_a after encoding, λ_t is the target λ value before encoding using the target number of bits, and δ_α and δ_β are constants.

For bit allocation at the GOP level, the number of bits allocated to a GOP is adjusted based on the actual number of bits used in the previous GOP to account for the fact that some GOPs may use more/less bits than were allocated to them. The number of bits allocated to the i^{th} GOP after encoding $j-1$ frames in the GOP is updated as follows:

$$T_{GOP_i}(j) = \begin{cases} \frac{\frac{R}{f}\times(N_c+S_w)-R_c}{S_w} \times N_{GOP}, & \text{if j = 1} \\ T_{GOP_i}(j-1) - R_i(j-1), & \text{if j} = 2,\ldots,N_{GOP} \end{cases} \tag{8}$$

where N_c is the total number of encoded frames, S_w is a constant which is used to smooth out the quality and rate changes between GOPs (set to 120 in CAVE)[2], R_c is the actual number of bits generated by the N_c frames, N_{GOP} is the number of frames in a GOP and $R_i(j-1)$ is the number of bits generated by the $(j-1)^{th}$ frame in the i^{th} GOP.

After the GOP bit allocation, the number of bits at the frame level is adjusted based on the number of bits used by the previous frames in the same GOP and the number of remaining bits as follows:

$$T_i(j) = \frac{T_{GOP_i}(j) - R_{GOP_i}}{\sum_{k=j}^{N_{GOP}} w_k} \times w_j, \tag{9}$$

where j refers to the index of the current frame in the i^{th} GOP, R_{GOP_i} is the number of bits generated by already encoded frames in the i^{th} GOP, and w_j is the weight of a frame that depends on its level if a hierarchical GOP structure is used.

[2]As discussed in [25], large values of S_w lead to smoother bitrate and quality changes.

Finally, the number of bits at the block level is adjusted based on the number of bits used to encode previous blocks in the same frame and the number of remaining bits as follows:

$$T_{block}[x,y] = \frac{T_i(j) - R_h - B}{\sum_{l=x}^{N_h} \sum_{k=y}^{N_v} w[l,k]} \times w[x,y], \tag{10}$$

where (x,y) is the position of the current block in the frame, N_h and N_v are the number of blocks in the horizontal and vertical directions respectively, B is the number of bits generated by already encoded blocks in the current frame, R_h is the estimated number of header bits, and $w[x,y]$ is the weight of the block that depends on the Mean Absolute Difference (MAD) between the original and predicted signal values of the pixels inside the block.

Using the λ value calculated by equation (4), the QP of the frame or the block can be calculated as:

$$QP = C_1 \times \ln \lambda + C_2, \tag{11}$$

where C_1 and C_2 are constants equal to 4.2005 and 13.7122, respectively [25].

Assigning Unequal QPs. Using equations (9) and (10) we can calculate the target bits per pixel, b, for a frame or a block, and from there the λ value can be calculated using equation (4). After that, through equation (11), the QP values can be calculated on the frame and block levels.

The granularity of rate control in CAVE is at the frame-level only in order to reduce the complexity and overhead of CAVE on the overall performance of the cloud gaming platform. CAVE assigns bits and QPs to blocks unequally using equation (10) to achieve our content-aware goals. Therefore, after encoding a whole frame, we update the α and β values on the frame-level and use them as estimates while calculating the λ values for the blocks of the next frame. The weight $w[x,y]$ of a block in equation (10) is replaced by the weights calculated by equation (3). Since we perform frame-level rate control, the R_h and B values in (10) are assumed to be zeros while allocating bits for individual blocks using our weights since it is done once only for all blocks before encoding the frame. Thus in CAVE, we modify equation (10) to become:

$$T_{block}[x,y] = T_i(j) \times \overline{w}[x,y], \tag{12}$$

where $\overline{w}[x,y]$ is the relative weight of the block at position (x,y) to the total weights of all blocks and is calculated as:

$$\overline{w}[x,y] = \frac{w[x,y]}{\sum_{i=1}^{N_h} \sum_{j=1}^{N_v} w[i,j]}. \tag{13}$$

In this way, the summation of $\overline{w}[x,y]$ over all blocks is equal to 1 to ensure that the summation of bits assigned to each block, $T_{block}[x,y]$, does not exceed the target number of bits for the frame $T_i(j)$. A mean filter is applied on the weights calculated by equation (13) to avoid blocking artifacts.

As an additional step to avoid overshooting the target number of bits for a frame, we apply a zero-accumulated ΔQP approach after assigning unequal QPs to blocks. This is done by calculating a reference QP for the frame using equation (11) and summing the difference between the frame's QP and each of the blocks' QPs. If this sum is less than 0, it means that there might be an overshoot over the target number of bits for the frame as many blocks have a

QP lower than the frame's reference QP. In this case, we raise the QPs evenly in blocks outside the ROIs to make sure that the average QP of the blocks is the same as the frame's reference QP. This is a heuristic to avoid rate spikes that might occur due to lowering the QPs of the blocks inside the ROIs.

Minimizing Rate Fluctuations Across Frames. To reduce rate fluctuations, we use a flat GOP structure containing I and P frames only with no bi-directional B-frames. This means that the weights of the frames in equation (9) are all equal to 1 as this setting is the most suitable for low latency applications.

Since the original λ-domain model did not include a buffer model and to further reduce the rate fluctuations, we modify the virtual buffer model [42] and integrate it with the rate control algorithm. Specifically, equation (8) is modified as follows:

$$T_{GOP_i}(j) = \begin{cases} (\frac{R}{f} - \tau \times \frac{V_i(j)}{f}) \times N_{GOP}, & \text{if } j = 1 \\ T_{GOP_i}(j-1) - R_i(j-1), & \text{if } j = 2, \ldots, N_{GOP} \end{cases}, \tag{14}$$

where $V_i(j)$ is the virtual buffer occupancy after encoding $j - 1$ frames in the i^{th} GOP, and τ is a constant controlling the convergence speed [42], set to 0.5 in CAVE based on our experiments. The virtual buffer occupancy $V_i(j)$ is updated as follows:

$$V_i(j) = \begin{cases} 0.5 \times Q, & \text{if } i = 1; j = 1 \\ V_{i-1}(N_{GOP}), & \text{if } j = 1 \\ V_i(j-1) + R_i(j-1) - \frac{R}{f}, & \text{if } j = 2, \ldots, N_{GOP} \end{cases}, \tag{15}$$

where Q represents the virtual buffer size which is usually equal to the size of one frame (R/f) in low-latency applications.

Equation (9) is modified such that the total number of target bits for the next frame will depend on two factors: the number of remaining bits in the GOP ($\tilde{T}$) and the number of bits based on the feedback from the virtual buffer ($\hat{T}$) which are defined as follows:

$$\tilde{T}_i(j) = \min(\frac{R}{f}, \frac{T_{GOP_i}(j)}{N_{GOP} - j}), \tag{16}$$

and

$$\hat{T}_i(j) = \frac{R}{f} + \min(0, L - V_i(j)), \tag{17}$$

where L represents the target buffer level and is equal to $0.5 \times Q$. Therefore the target number of bits for the j^{th} frame in the i^{th} GOP is calculated as follows:

$$T_i(j) = \eta \times \tilde{T}_i(j) + (1 - \eta) \times \hat{T}_i(j), \tag{18}$$

where η is a constant equal to 0.9 for low-latency use cases [42]. Opposed to [42], we ensure in equations (16) and (17) that the target number of bits for the next frame is always upper bounded by R/f to avoid rate spikes and latencies.

Summary of all Steps. We perform rate control and allocate bits to different areas within the game frames and across frames as follows:

- Given a target number of bits for a frame, we calculate the target bits of each block using equation (12).
- Using the target bits of the block we calculate the λ and QP values for a block using equations (4) and (11), respectively.
- After encoding a single frame we update the virtual buffer occupancy and the number of bits in the GOP through equations (15) and (14), respectively.

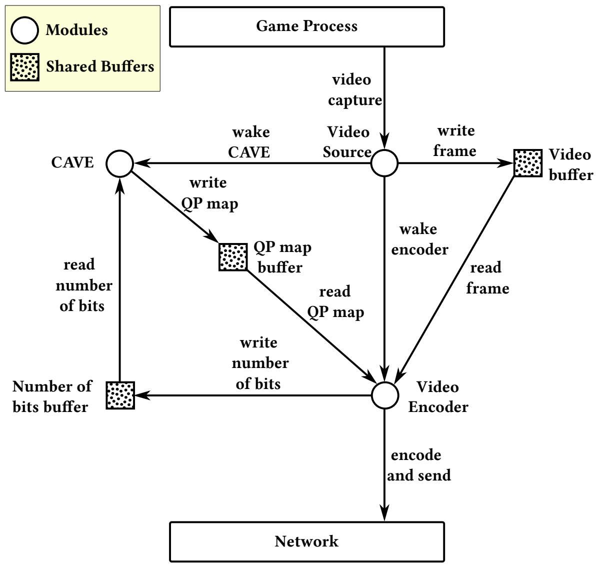


Figure 3: Implementation of CAVE as a GamingAnywhere server-side module.

- Also, the α and β values are updated using the actual number of bits generated after encoding the frame using equations (5), (6), and (7).
- The target number of bits for the next frame is calculated using equation (18).

4 EXPERIMENTAL EVALUATION

In this section, we describe our implementation of CAVE in GamingAnywhere, an open-source cloud gaming system. We also present our experimental setup and show the performance gains achieved by CAVE as well as analyze the overhead imposed by it.

4.1 Implementation in GamingAnywhere

GamingAnywhere is an open-source system designed to enable researchers to evaluate their ideas in a realistic cloud gaming platform. The server side of the system is composed of 6 modules to handle the communications with the client as well as encoding the audio/video content of the game. The client consists of two modules to decode the audio/video from the server and capture the player's actions and send them to the server.

CAVE is implemented as an additional module at the server side of GamingAnywhere; CAVE does not require any changes to the client side. We illustrate the interaction between CAVE and the various modules of the GamingAnywhere server in Figure 3, which is mainly done through buffers. Notice that we do not show all modules of the server, only the ones that CAVE interacts with. Notice also that GamingAnywhere did not have support for the recent HEVC encoder; we added this support by integrating the open-source x265 HEVC encoder [20] into GamingAnywhere.

As shown in Figure 3, the Video Source module is responsible for starting the Video Encoder and CAVE modules. The Video Encoder module interacts with the Video Source module through the `Video`

Abbreviation	Resolution	Perspective	Motion	Brightness	ROI: Importance Factor (F)
Game1	1280x720	3rd Person	High	High	Player's avatar:1 Textual information:0.9 Map:0.9
Game2	640x480	1st Person	Low	Medium	Middle of screen:1
Game3	800x600	1st Person	Medium	High	Middle of screen:1 Textual information:1
Game4	640x480	1st Person	High	Medium	Character's face:1

Table 1: Characteristics of the gaming video segments used in our experiments.

`Buffer` to encode raw frames after converting them from the RGB color space to the YUV color space, which is the format required by the encoder. After encoding a frame by the encoder, the actual number of bits returned by the encoder's API is written into the `Number of bits buffer` so that it can be read by the rate control component of CAVE to update the remaining number of bits in the GOP and the virtual buffer occupancy. This buffer is created by using the existing GamingAnywhere APIs that allow the creation of buffers to communicate information among modules as needed and is created and managed by the Video Encoder module. Similarly, the rate control component in CAVE communicates QPs of the blocks to the encoder through the `QP map buffer`, such that the QPs are applied while encoding the next frame. This buffer is created and managed by CAVE.

In order to send QPs to the x265 encoder, we operate it under the constant QP mode (CQP) by giving it an arbitrary QP (e.g., 22) and enabling the adaptive quantization mode. The adaptive quantization mode allows sending a map of QP offsets for blocks in the frame. These offsets are added on top of the decisions taken by the encoder. Since we have set the operation mode of the encoder to CQP, the encoder's decision is to apply the same QP (i.e., 22) to all blocks in the frame. We leverage this mode to achieve our content-aware rate control purposes by sending offsets from the constant QP that are calculated by CAVE. The x265 encoder's interface is extended in order to enable the adaptive quantization mode with the CQP mode. It is important to notice that CAVE does not require modifying the source code of the video encoder; it only uses the exposed encoder's APIs to optimize the video encoding process based on the game content.

We use the following configurations of the x265 encoder, which are recommended to achieve low latency for cloud gaming [17]:

```
--preset ultrafast --tune zerolatency --ref 1 --me dia
--intra-refresh --merange 16 --bframes 0
```

4.2 Experimental Setup

We conduct multiple experiments with different video games in various scenarios. We assess the performance in terms of several practical metrics and compare our method against the baseline HEVC encoder and the closest work in the literature.

Video Games: We select segments of four diverse games to test the performance of CAVE; we refer [3] to them as Game1, Game2, Game3, and Game4. Each video segment is 20 seconds long and contains 600 frames. Game1 is a popular vehicular combat game with a large user base on different platforms. Game2, Game3, and Game4 are commonly-available samples with game development tools, which are representative of real games in terms of complexity, motion and lighting. They also contain all necessary elements of real games in terms of rendering graphical components. The three samples were chosen with different characteristics in terms of motion and brightness. They have different ROIs as well. Table 1 summarizes the characteristics of the four gaming segments used in our experiments.

Methods Compared Against: We compare CAVE against the following:

- *Base*: this is a baseline HEVC encoder with the recent λ-domain rate control model, which assigns the same QP to all blocks.
- *RQ* [38]: this is the closest work in the literature that aims at achieving content-aware rate control in cloud gaming. The content-aware method in [38] also uses ROIs. The authors of this model stated that it can be extended to support the HEVC encoder by using the pixel-wise unified RQ model for rate control. We implemented this method as in the last version of the HEVC Test Model (HM-8.0), which contains the RQ model.

We did not include other works, e.g., [30], in our comparison as they were designed for H.264/AVC and did not claim portability or support for HEVC.

Performance Metrics: We consider the following metrics:

- *Structural Similarity Index (SSIM) [41]:* This metric measures the quality of a distorted frame relative to a reference frame by using perceptual models that simulate the Human Visual System. We measure SSIM for the ROIs and the whole frame.
- *Video Multimethod Assessment Fusion (VMAF) [27]:* This metric, developed and used by Netflix, is based on a machine learning model (SVM) to predict the subjective score of a distorted video. It also captures the temporal degradation among frames, since one of the features used in training the SVM model is a temporal quality metric.
- *BD-rate [3]:* This metric computes the average bitrate savings between two competing methods by calculating the average area between their Rate-Distortion (R-D) curves. Negative values denote bitrate savings achieved by the first method compared to the second one.

[3]Although the games we used in the experiments are open source, we could not reveal their names due to legal restrictions and precautions of our industrial collaborators.

- *Quality Fluctuations:* This metric measures the variation of quality between frames and is calculated as the standard deviation of the SSIM values of the frames. Lower values of this metric are better and indicate that the encoded segment has consistent quality.
- *Rate Control Accuracy:* This metric measures the mismatch between the target and actual bitrates. It shows the accuracy of the rate control in various methods. It is calculated as the absolute difference between the target and actual bitrates divided by the target bitrate.
- *CPU Time:* This metric measures the time (in milliseconds) needed to perform the required steps in CAVE and is used to compute the overhead of CAVE, which is important for meeting the low-latency requirements of cloud gaming.

Experiments: We set up the GamingAnywhere system on a Windows 10 desktop equipped with 2.40 GHz processors and 12 GB of memory. We conducted end-to-end experiments to ensure the whole system (server and client) works, and to assess the processing overheads of our method. To analyze the quality of multiple video segments under different bitrates and many encoding options, we encode the video segments offline using our method, after collecting the raw frames from GamingAnywhere. Specifically, we instrument the code of the `filter-rgb2yuv` module to save the raw frames into a file after their conversion to the YUV color space. The code responsible for offline encoding loads the raw frames from the saved file and sends them to the x265 encoder along with the QPs that we calculate in CAVE. We define the ROIs manually in order to show the quality gains that can be achieved if the ROI information were to be exported from the game. To minimize the manual assignment of ROIs, we define ROIs for every other frame as the content does not change significantly between two frames.

Since each video segment has different characteristics of motion and resolution, we determine the target bitrates suitable for each segment by encoding it under 4 different QPs (22, 27, 32, 37) and using the resulting bitrates as targets for CAVE. We use the HM reference decoder (HM-16.18) to decode the bitstreams generated by the x265 encoder so that we can evaluate its quality using SSIM/VMAF in MATLAB.

In CAVE, K in equation (3) is determined by varying its value such that the overall quality is not degraded compared to the baseline for the lowest bitrate. The optimal K value of the lowest bitrate is increased by a step of one as we go from one bitrate to the next one. The intuition behind this approach is that we can allow a higher discrepancy in bit allocation between blocks in ROIs and blocks in non-ROIs if we have a larger number of bits to assign. A summary of the target bitrates for each segment and the value of K for the lowest bitrate, K_{min}, are shown in Table 2.

4.3 Results

To make the presentation clearer and due to space limitations, we only present a representative sample of our figures and results in the following.

Quality Gains and Bandwidth Savings. We start by showing the quality improvements in the ROIs that can be achieved by CAVE in comparison to Base and RQ. In Figure 4, we plot the SSIM in the ROIs achieved by CAVE, Base, and RQ for Game1 under different bitrates. The figure shows that CAVE consistently outperforms Base and RQ for all bitrates. The quality gains in ROIs can be translated to bitrate savings. For example, for a target SSIM quality level of 0.90, CAVE uses around 1 Mbps while the Base and RQ methods require around 2 Mbps to achieve the same quality in the most important areas in the frames. This is equivalent to a bitrate saving of ≈50%. Similar results were obtained for the other games as shown in Figure 5.

Segment/QP	22	27	32	37	K_{min}
Game1	7760.37	4232.44	1911.93	716.38	7
Game2	421.96	224.17	124.51	75.82	4
Game3	7430.28	3923.28	1923.74	923.47	2
Game4	4969.67	2796.85	1388.36	627.11	2

Table 2: QPs and resulting bitrates in kbps of game segments used in the experiments. The rightmost column shows K_{min}; the K value of the lowest bitrate.

Segment/Comparison	CAVE– Base	CAVE– RQ
Game1	-46%	-33%
Game2	-25%	-12%
Game3	-22%	-68%
Game4	-21%	-89%

Table 3: Potential bitrate savings by CAVE; computed using the BD-Rate metric on the SSIM R-D curves of ROIs.

We next quantify the bitrate savings achieved by CAVE using the BD-rate metric. Table 3 lists the potential savings for each game's segment, when the BD-rate metric is computed on the ROI SSIM curves. The table shows that CAVE can achieve bitrate savings from 20% to up to 45% compared to Base, and from 12% to up to 88% compared to RQ. These are substantial savings especially for high-quality games with thousands of users playing for extended periods of time.

In terms of overall quality measured by SSIM, Figure 6 shows that CAVE is able to maintain the overall quality at a level very close to the baseline encoder. This means that our weights and the unequal QP assignment to blocks do not introduce instabilities to the encoding process and the overall quality. Since the SSIM metric does not capture temporal degradations, we refer to the results of VMAF in Figure 7. The results of the VMAF metric confirm that CAVE does not introduce temporal artifacts between successive frames.

While the quality in ROIs is perceived by players the most, it is still important to maintain a good quality in the other non-ROI areas. We show in Figure 6 and Figure 7 that overall quality (in terms of SSIM and VMAF) across all areas did not drop because of the unequal bit allocation of CAVE.

Quality Fluctuations and Rate Control Accuracy. The quality fluctuations metric, measured as the standard deviation of the SSIM value of the frames, is plotted in Figure 8 for CAVE, Base, and RQ for three different video games. The figure shows that CAVE maintains consistent quality and does not introduce any additional fluctuations because of the unequal bit allocation.

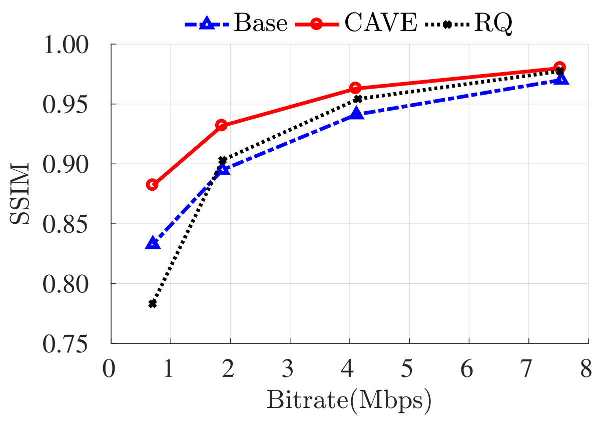


Figure 4: Performance of CAVE against Base and RQ in terms of the SSIM in ROIs, for Game1.

The rate control accuracy of CAVE is assessed by calculating the average of absolute differences between the actual and target bitrates using all the actual bitrates for each segment. The results are summarized in Table 4. The results show that CAVE is accurate in staying within the target bitrate. It achieves a low error between the actual and target bitrates of up to 0.7%.

Segment/Method	CAVE	Base	RQ
Game1	0.62%	0.43%	0.50%
Game2	0.01%	0.40%	1.97%
Game3	0.70%	0.81%	1.91%
Game4	0.29%	0.36%	2.24%

Table 4: Accuracy of the rate control of CAVE, Base, and RQ, in terms of the error percentage between the target and actual bitrates.

Overhead Imposed by CAVE. Finally, we analyze the overhead imposed by CAVE. We run a live session using one of the graphics samples, Game3, to study the impact of CAVE on the overall performance of GamingAnywhere. The gaming client is connected to the gaming server over a local area network to isolate the effect of network queuing and delays and analyze only the impact of CAVE. The ROIs in this experiment are pre-defined and loaded by CAVE during its initialization in GamingAnywhere. CAVE computes the block weights every other frame. The existing transport protocol in GamingAnywhere (RTP over UDP) is used in this experiment. We note that since CAVE provides accurate rate control, it will not have any significant impact on the packet transmission rate in the cloud gaming system.

We measure the CPU time consumed by the two components of CAVE. First, we measure the CPU time needed to calculate the weights. Then we measure the CPU time needed to perform rate control. We run the gaming session until 4500 frames are encoded, and compute the average time consumed by CAVE over 900-frame periods. Table 5 lists the time used by each component of CAVE in milliseconds. In this table, we also compute the percentage of the total overhead relative to a typical allowed latency for most games, which is 100 milliseconds [10]. As shown in the table, CAVE can easily run in real-time, as it adds an overhead of only 1.21% on average to the total latency time.

We note that the times in Table 5 were measured without any parallelization of CAVE's code; parallelization is usually done for video codecs. Since there is no dependency between the weights of individual blocks, the weight calculation in CAVE can be parallelized. The same concept also applies to the rate control algorithm. Furthermore, the processing overhead of CAVE can be reduced if we implemented its components inside the code of the video encoder itself. However, to preserve the modularity of the cloud gaming system, CAVE is implemented as a separate module that communicates with the encoder as a black-box through its APIs.

Period	Weights(ms)	Rate Control(ms)	Overhead(%)
1	0.33	1.87	2.20
2	0.54	0.59	1.13
3	0.44	0.71	1.15
4	0.20	0.43	0.59
5	0.42	0.57	0.99
Average	0.39	0.83	1.21

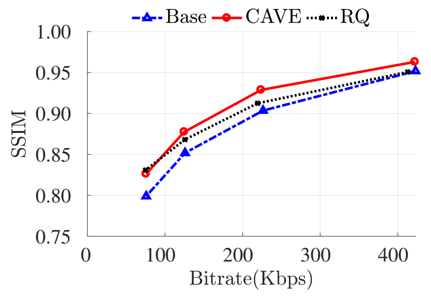
Table 5: Processing overhead of the two components of CAVE, weight calculation and rate control, and the total overhead as a percentage of a typical latency of 100 ms. Results are for Game3.

Summary of the results. The results in this section show that:

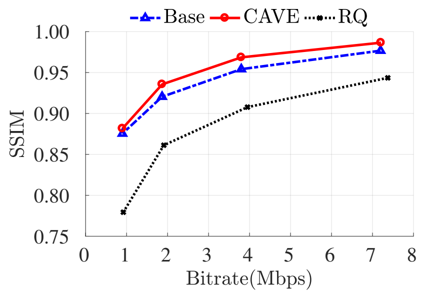
- CAVE achieves consistent quality gains in ROIs compared to other methods.
- CAVE maintains the overall quality at a level that is no worse than the baseline encoder which does not perform content-aware encoding.
- CAVE does not introduce significant quality fluctuations despite its unequal allocation of bits and QPs to blocks within each frame.
- CAVE is an accurate rate control method that meets the target bitrate while achieving content-aware quality gains.
- CAVE is parallelizable, runs in real-time, and does not impose significant processing overhead on the cloud gaming system.

5 CONCLUSIONS

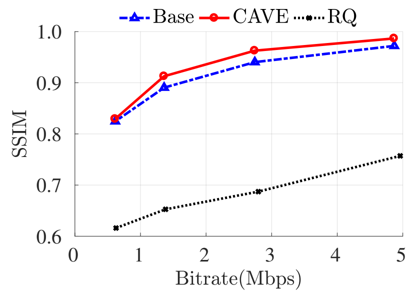
Cloud gaming platforms are getting popular because they allow clients with diverse devices to participate in the game without requiring special hardware. Encoding and transmitting the game actions as video streams to clients is expensive, especially for high-quality games with complex graphics. We designed and implemented a new method to efficiently encode video streams for cloud gaming platforms. The proposed method, referred to as CAVE, strives to improve the quality of content delivered to clients, given a target bitrate budget. It does so by allocating the bit budget to different areas in the video frames such that the quality in the most important areas (i.e., regions of interest) is maximized, leading to an



(a) Game2

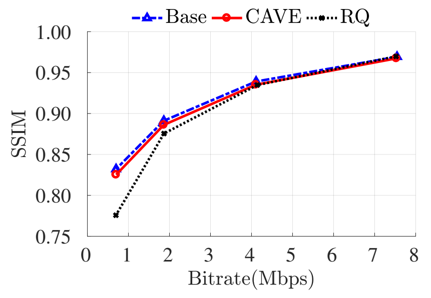


(b) Game3

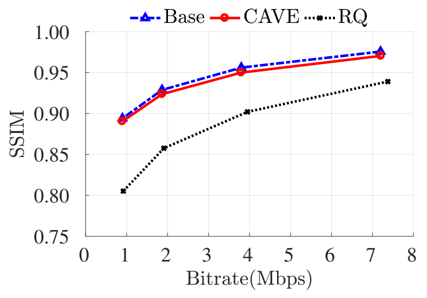


(c) Game4

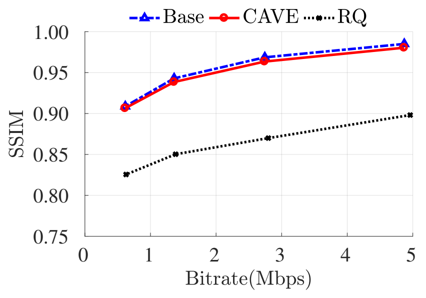
Figure 5: Performance of CAVE against Base and RQ in terms of the SSIM in ROIs, for three different games.



(a) Game1

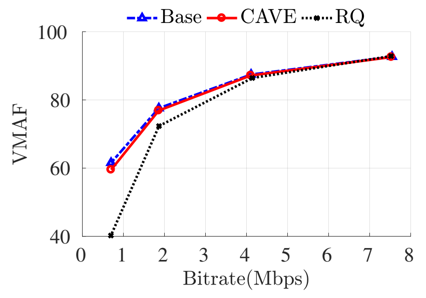


(b) Game3

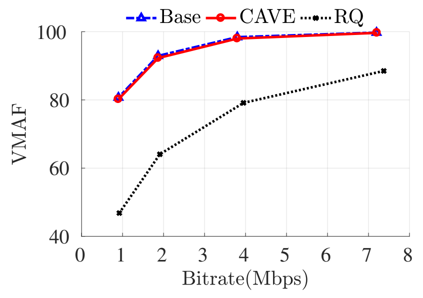


(c) Game4

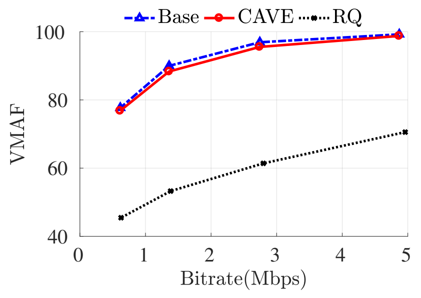
Figure 6: Performance of CAVE against Base and RQ in terms of the SSIM in all areas, for three different games.



(a) Game1



(b) Game3



(c) Game4

Figure 7: Performance of CAVE against Base and RQ in terms of VMAF, for three different games.

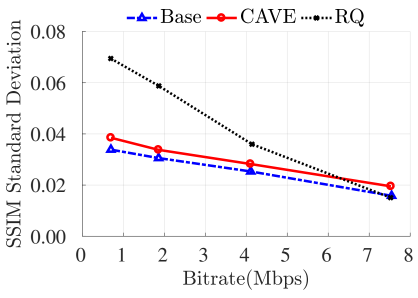
overall improvement of perceptual quality. We have implemented CAVE in GamingAnywhere, which is an open-source cloud gaming platform. We conducted extensive experiments with multiple real video games and measured several performance metrics, including SSIM, VMAF, quality fluctuations, and bitrate savings. Our experiments show that CAVE consistently outperforms the closest work in the literature.

ACKNOWLEDGMENTS

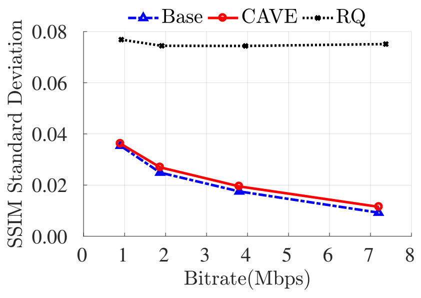
This work is supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

REFERENCES

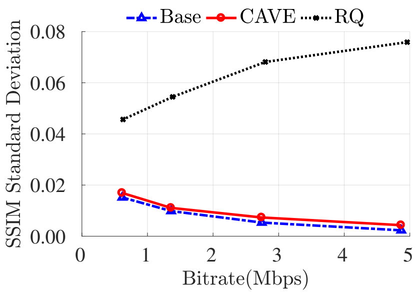
[1] Advanced Micro Devices, Inc. 2018. How to Capture Your Gameplay Using Radeon ReLive. Retrieved November 05, 2018 from https://support.amd.com/en-us/kb-articles/Pages/DH-023.aspx#Streaming
[2] Hamed Ahmadi, Saman Zad Tootaghaj, Mahmoud Reza Hashemi, and Shervin Shirmohammadi. 2014. A game attention model for efficient bit rate allocation in cloud gaming. *Multimedia Systems* 20, 5 (2014), 485–501.
[3] Gisle Bjontegaard. 2001. Calculation of average PSNR differences between RD-curves. *Video Coding Experts Group-M33* (2001).
[4] Business Wire, Inc. 2018. Cloud Gaming Market Analysis By Platform, Service Type & Geography, With Forecasts to 2023. Retrieved October 08, 2018 from https://www.businesswire.com/news/home/20180424005697/en/Cloud-Gaming-Market-Analysis-Platform-Service-Type
[5] Wei Cai, Ryan Shea, Chun-Ying Huang, Kuan-Ta Chen, Jiangchuan Liu, Victor C.M. Leung, and Cheng-Hsin Hsu. 2016. A Survey on Cloud Gaming: Future



(a) Game1



(b) Game3



(c) Game4

Figure 8: Performance of CAVE against Base and RQ in terms of quality fluctuations, for three different games.

of Computer Games. *IEEE Access* 4 (2016), 7605–7620.

[6] Hyomin Choi, Junghak Nam, Jonghun Yoo, D. Sim, and Ivan Bajic. 2012. Rate control based on unified RQ model for HEVC. *ITU-T SG16 Contribution, JCTVC-H0213* (2012), 1–13.

[7] Kareem Choudhry. 2018. Project xCloud: Gaming with you at the center. Retrieved October 08, 2018 from https://blogs.microsoft.com/blog/2018/10/08/project-xcloud-gaming-with-you-at-the-center/

[8] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. 2012. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *Proc. of the ACM International Workshop on Network and Systems Support for Games.* 2:1–2:6.

[9] Seong-Ping Chuah and Ngai-Man Cheung. 2014. Layered Coding for Mobile Cloud Gaming. In *Proc. of the ACM International Workshop on Massively Multiuser Virtual Environments.* 1–6.

[10] Mark Claypool and Kajal Claypool. 2010. Latency Can Kill: Precision and Deadline in Online Games. In *Proc. of the ACM International Conference on Multimedia Systems.* 215–222.

[11] NVIDIA Corporation. 2018. NVIDIA GeForce. Retrieved October 08, 2018 from https://www.nvidia.com/en-us/geforce/products/geforce-now/

[12] Valve Corporation. 2018. Extend your Steam gaming experience to your phone, tablet, or TV over your local network. Retrieved November 05, 2018 from https://store.steampowered.com/steamlink/about

[13] John M. Henderson, James R. Brockmole, Monica S. Castelhano, and Michael Mack. 2007. Chapter 25 - Visual saliency does not account for eye movements during visual search in real-world scenes. In *Eye Movements.* Elsevier, 537 – III.

[14] Hua-Jun Hong, De-Yu Chen, Chun-Ying Huang, Kuan-Ta Chen, and Cheng-Hsin Hsu. 2015. Placing virtual machines to optimize cloud gaming experience. *IEEE Transactions on Cloud Computing* 3, 1 (2015), 42–53.

[15] Hua-Jun Hong, Chih-Fan Hsu, Tsung-Han Tsai, Chun-Ying Huang, Kuan-Ta Chen, and Cheng-Hsin Hsu. 2015. Enabling Adaptive Cloud Gaming in an Open-Source Cloud Gaming Platform. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 12 (2015), 2078–2091.

[16] Catherine Hsiao. 2018. Pushing the limits of streaming technology. Retrieved October 08, 2018 from https://blog.google/technology/developers/pushing-limits-streaming-technology/

[17] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. 2013. GamingAnywhere: An Open Cloud Gaming System. In *Proc. of the ACM International Conference on Multimedia Systems.* 36–47.

[18] Chun-Ying Huang, Yu-Ling Huang, Yu-Hsuan Chi, Kuan-Ta Chen, and Cheng-Hsin Hsu. 2015. To Cloud or Not to Cloud: Measuring the Performance of Mobile Gaming. In *Proc. of the ACM International Workshop on Mobile Gaming.* 19–24.

[19] Gazi Illahi, Matti Siekkinen, and Enrico Masala. 2017. Foveated video streaming for cloud gaming. In *Proc. of the IEEE International Workshop on Multimedia Signal Processing.* 1–6.

[20] MulticoreWare Inc. 2018. About x265. Retrieved October 19, 2018 from http://x265.org/about/

[21] Instructables. 2017. How to Make a Simple Game in Unity 3D. Retrieved Januaury 19, 2019 from https://www.instructables.com/id/How-to-make-a-simple-game-in-Unity-3D/

[22] Yihao Ke, Guoqiao Ye, Di Wu, Yipeng Zhou, Edith Ngai, and Han Hu. 2017. GECKO: Gamer Experience-Centric Bitrate Control Algorithm for Cloud Gaming. In *Proc. of the ACM International Conference on Image and Graphics.* 325–335.

[23] Jong-Seok Lee and Touradj Ebrahimi. 2012. Perceptual video compression: A survey. *IEEE Journal of Selected Topics in Signal Processing* 6, 6 (2012), 684–697.

[24] Bin Li, Houqiang Li, Li Li, and Jinlei Zhang. 2012. Rate control by R-lambda model for HEVC. *ITU-T SG16 Contribution, JCTVC-K0103* (2012), 1–5.

[25] Bin Li, Houqiang Li, Li Li, and Jinlei Zhang. 2014. λ-Domain Rate Control Algorithm for High Efficiency Video Coding. *IEEE Transactions on Image Processing* 23, 9 (2014), 3841–3854.

[26] Shengxi Li, Mai Xu, Xin Deng, and Zulin Wang. 2015. Weight-based R-λ rate control for perceptual HEVC coding on conversational videos. *Signal Processing: Image Communication* 38 (2015), 127–140.

[27] Zhi Li, Anne Aaron, Ioannis Katsavounidis, Anush Moorthy, and Megha Manohara. 2018. Toward A Practical Perceptual Video Quality Metric. Retrieved October 18, 2018 from https://medium.com/netflix-techblog/toward-a-practical-perceptual-video-quality-metric-653f208b9652

[28] Xiaofei Liao, Li Lin, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang, Bo Li, Xiaofei Liao, Li Lin, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang, and Bo Li. 2016. LiveRender: A Cloud Gaming System Based on Compressed Graphics Streaming. *IEEE/ACM Transactions on Networking* 24, 4 (2016), 2128–2139.

[29] LiquidSky Software, Inc. 2018. LiquidSky. Retrieved October 08, 2018 from https://liquidsky.com/

[30] Yao Liu, Sujit Dey, and Yao Lu. 2015. Enhancing video encoding for cloud gaming using rendering information. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 12 (2015), 1960–1974.

[31] Marwa Meddeb, Marco Cagnazzo, and Béatrice Pesquet-Popescu. 2014. Region-of-interest-based rate control scheme for high-efficiency video coding. *APSIPA Transactions on Signal and Information Processing* 3 (2014), e16.

[32] Robert J. Peters and Laurent Itti. 2008. Applying Computational Tools to Predict Gaze Direction in Interactive Visual Environments. *ACM Transactions on Applied Perception* 5, 2, Article 9 (2008), 19 pages.

[33] Ashwin Rao, Arnaud Legout, Yeon-sup Lim, Don Towsley, Chadi Barakat, and Walid Dabbous. 2011. Network Characteristics of Video Streaming Traffic. In *Proc. of the ACM International Conference on Emerging Networking Experiments and Technologies.* 25:1–25:12.

[34] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *Proc. of the IEEE International Conference on Computer Vision and Pattern Recognition.* 779–788.

[35] Ivan Slivar, Lea Skorin-Kapov, and Mirko Suznjevic. 2016. Cloud Gaming QoE Models for Deriving Video Encoding Adaptation Strategies. In *Proc. of the ACM International Conference on Multimedia Systems.* 18:1–18:12.

[36] Sony Interactive Entertainment LLC. 2016. Playstation Now. Retrieved October 08, 2018 from https://www.playstation.com/en-ca/explore/playstationnow/

[37] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, Thomas Wiegand, et al. 2012. Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Transactions on Circuits and Systems for Video Technology* 22, 12 (2012), 1649–1668.

[38] Kairan Sun and Dapeng Wu. 2015. Video rate control strategies for cloud gaming. *Journal of Visual Communication and Image Representation* 30 (2015), 234–241.

[39] Shanshe Wang, Siwei Ma, Shiqi Wang, Debin Zhao, and Wen Gao. 2013. Quadratic ρ-domain based rate control algorithm for HEVC. In *Proc. of the IEEE International Conference on Acoustics, Speech and Signal Processing.* 1695–1699.

[40] Zhou Wang and Alan C. Bovik. 2001. Embedded foveation image coding. *IEEE Transactions on Image Processing* 10, 10 (2001), 1397–1410.

[41] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. 2004. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612.

[42] Zhongzhu Yang, Li Song, Zhengyi Luo, and Xiangwen Wang. 2014. Low delay rate control for HEVC. In *Proc. of the IEEE International Symposium on Broadband Multimedia Systems and Broadcasting.* 1–5.

A APPENDIX[*]

This appendix explains the steps needed to reproduce the results reported by our proposed method (CAVE). CAVE is a content-aware rate control method devised for HEVC encoders for use in cloud gaming. We evaluated CAVE in terms of 2 main aspects: quality and overhead. The quality evaluation requires having the same video sequences to be evaluated and compared under different methods. This is done by implementing CAVE in a stand-alone project to evaluate those video sequences. The overhead evaluation of CAVE is done through an implementation in an existing open-source cloud gaming system (GamingAnywhere). The implementation of CAVE in GamingAnywhere includes a separate module for our method at the server-side.

The source code for CAVE and its evaluation can be found at https://github.com/mohamedhegazy/CAVE.git.

In the following subsections, we explain the structure of our datasets and source code as well as the steps needed to run and evaluate CAVE.

A.1 Dataset Structure

This subsection explains the dataset structure for evaluating CAVE in terms of quality (e.g., SSIM, VMAF, etc.). Each video sequence to be evaluated in the dataset should be in a separate folder with the following special structure to run with the provided evaluation code:

- *Folder Name:* A video sequence should be placed in a folder called "`ga<sequence number>`" (the sequence number concatenated to `ga`).
- *Sequence Name:* The video sequence file should have the following name: `raw_<width>_<height>.yuv`
- *Folder Contents:* The folder should contain a sub-directory called `QP` to hold the encoded versions of the video sequence under 4 different QPs. In addition, a folder called `temp` should be present to hold the evaluation results of the encoded sequences. The folder should contain a text file called `conf.txt` which contains key-value pairs as follows:
  - width=<video width>
  - height=<video height>
  - fps=<frames per second>

An example of the folder structure for a video sequence should look as follows:

```
ga1
 |__QP
 |__temp
 |__raw_1280_720.yuv
 |__conf.txt
```

A.2 Source Code Structure

This subsection explains the details of the source code and the required modifications to run on other environments. The following dependencies are needed to run the source code for the quality and overhead evaluation:

- Windows 7
- Linux CentOS
- Visual Studio 2017 & Visual Studio 2010 (for GamingAnywhere)
- Matlab
- OpenCL-enabled GPU[1]
- Python
- Java

The source code directory contains the following folders:

- `YUVPlayerROI`: contains a Java project responsible for defining ROIs manually. The YUV conversion is taken from here[2]. The program opens a GUI to select a video file with YUV format and allows specifying ROIs using the format defined here[3]. By default, the ROIs are defined for every other frame in the video. There are multiple ROI types with different importance factors which can be edited through the source file `YUVPlayer.java`.
  By default, the ROIs will be written in a file besides the location of the video sequence (i.e., inside the `ga<sequence number>` folder). The ROI files should not be moved outside of this directory.
- `Encoder`: contains the source code of the stand-alone project containing CAVE which is responsible for encoding a raw video file under different methods (i.e., Base, CAVE, RQ). The folder contains a Visual Studio 2017 solution. Inside the debug directory, there exists a python script called `encode.py` which drives the encoding process of the various video sequences. This script contains variables that should be changed to evaluate new sequences. These variables are:
  - `base_path` at line 4: This path should be changed to point at the parent directory containing the folders of the video sequences (i.e., the folder containing the `ga<sequence number>` folders).
  - `K` at line 6: This is a 2D array that contains the K value at each bitrate for each video sequence. The rows of this 2D array represent the video sequence and the columns represent the bitrate. For example, at the highest bitrate for the first video sequence we can have a K value of 7, therefore the entry (0,0) of this 2D array would be 7.
  - `length` at line 7: This is the length of the video sequence in seconds. This value should be changed to the length of the video sequence and assuming that all video sequences have the same length.
  - `width` at line 11: This is a 1D array holding the width of each video sequence.
  - `height` at line 12: This is a 1D array holding the height of each video sequence.
  - `range` at line 14: This is the range of the video sequences numbers. For example, if the video sequences are as follows: `ga4, ga5, ga6`, then the range should be changed to become (4,7).

  The `encode.py` script calls another python script called `qp.py` to encode a sequence under 4 different QPs at line

[*]Subject to the open source code identified, the code in this package was written solely by the SFU researchers.

[1]OpenCL is needed as the RQ method is implemented on a GPU to make its evaluation faster.

[2]https://github.com/luuvish/java-yuv-viewer

[3]https://github.com/AlexeyAB/darknet#how-to-train-to-detect-your-custom-objects

16. Then the `encode.py` script calls the `Encoder.exe` which takes the following options in order:
– `path`: This is the path to the video sequence folder (i.e., `ga<sequence number>`).
– `bitrate`: This is the target number of bits per second.
– `method`: This is the method to use for encoding which takes the following values: 0 for CAVE, 2 for RQ, and 4 for Base.
– `sequence number`: This is sent as 1 by default.
– `K`: This is the K value used for CAVE and is valid when the method to encode with is set to CAVE only and should be sent as 0 otherwise.
– `encoder`: This is the index of encoder to use and should be sent as 0 by default to use the x265 encoder.

- `Evaluation`: contains the Matlab scripts required to evaluate the encoded videos. Before running the evaluation, VMAF should be downloaded and compiled (C++) for Linux as noted here[4]. The evaluation code should run on a Linux machine, by running the `run.sh` script. This script will decode the encoded sequences in their directory and evaluate their VMAF score. Then the script will call the Matlab script called `evaluationsVariations.m` to evaluate the sequences under different methods. The evaluation scripts should be placed inside the folder containing the video sequences to be evaluated. The following variables should be changed in `runs.sh` to evaluate new sequences:
– `games` at line 2: This is a 1D array containing the name of the folders of the video sequences (e.g., `ga1, ga2, . . .`).
– `width` at line 5: This is a 1D array holding the width of each video sequence.
– `height` at line 6: This is a 1D array holding the height of each video sequence.
The following variables should be changed in `evaluationsVariations.m` to evaluate new sequences:
– `width_` at line 19: This is a 1D array holding the width of each video sequence.
– `height_` at line 20: This is a 1D array holding the height of each video sequence.
– `base_path_` at line 26: This is a 1D array containing the name of the folders of the video sequences (e.g., `ga1, ga2, . . .`).
Lines 5, 6, 7 in the Matlab script can be commented out if multiple cores do not exist on the machine.
- `gaminganywhere`: this folder contains the implementation of CAVE in GamingAnywhere. The implementation was done under Windows 7 and was not tested on other operating systems. The run time overhead of CAVE will be printed in the log file maintained by GamingAnywhere. Therefore, when running a specific game logging should be enabled by specifying a path to the log file. A sample of a configuration file containing the added configuration needed by CAVE is found under `gaminganywhere/bin.win32/config/server.d3dex-rc.conf` which has 4 new added variables at the end of the file:
– `mode`: This is the mode to run GamingAnywhere under which can take a value of 0 for CAVE and 2 for RQ and 3 for Base.
– `K`: This is the K value used for CAVE and is valid only when the mode is set to CAVE.
– `recording`: This is a boolean variable used to store raw frames in a file. The raw file will be stored with the Game executable if the mode of GamingAnywhere is event-driven, otherwise it will be stored with the executable of the server.
Two common configuration files are placed inside the `common` directory for the configuration of the x265 encoder and are called `video-x265-param-rc.conf` and `video-x265-rc.conf`. The directory containing the executable of the server and the directory of the executable of the game should contain a file called `roi0.txt` to hold default ROI information for CAVE.

A.3 Steps For Evaluation

This subsection explains the steps needed to evaluate new video sequences. The steps for quality evaluation of CAVE are as follows [5]:

(1) A video sequence should be acquired either through recording using GamingAnywhere [6] or externally. This video sequence should be renamed and placed following the dataset structure convention discussed above.
(2) Using the provided Java program, ROIs should be defined by opening the video sequence from the Java program.
(3) The provided Visual Studio solution (in `Encoder` folder) should be used to encode the recorded video sequence under different methods by calling the driver python script `encode.py`.
(4) Using the code provided in the `Evaluation` folder, the newly encoded sequences are evaluated using the bash script `run.sh`.

The steps needed to evaluate CAVE in terms of overhead are:

(1) The code should be compiled in Windows 7 by following the steps described by GamingAnywhere developers.
(2) The provided configuration file (`server.d3dex-rc.conf`) should be adjusted to enable logging as described in GamingAnywhere documentation.
(3) In the log file, the average time in terms of milliseconds needed to run CAVE will be printed at periods of 900 frames.

A.4 Miscellaneous

The source code directory contains a folder called `Unity-ROI` which contains the example of the game developed in Unity to show the ability to extract ROIs from a real game engine. The code is instrumented to take a screenshot of each frame and store it besides the solution. The requirements to run this code are Unity 5.5 and Visual Studio 2017.

[4] https://github.com/Netflix/vmaf

[5] The first four steps should be done under Windows 7 OS and the last one under Linux CentOS.

[6] Using the periodic server is preferable.