Toward a mobile gaming based-computation offloading

Farouk MESSAOUDI
IRT b◇com
Rennes, France
farouk.messaoudi@b-com.com

Adlen KSENTINI
Eurecom
SophiaTech, Biot, France
adlen.ksentini@eurecom.fr

Philippe BERTIN
IRT b◇com, Orange Labs
Rennes, France
philippe.bertin@b-com.com

***Abstract*—3D Video games are considered as one of the most complex applications in the market, due to their real-time constraint and high computational requirements. Compared to dedicated gaming boxes (e.g., Xbox and PlayStation), mobile devices, including smartphones and tablets, still fail to achieve interactive rendering rates, even with low gaming requirements. Two solutions are offered to mobile gamers, leveraging resourceful platforms. The first one is the *cloud gaming*; wherein the entire game engine is hosted on dedicated servers in the cloud. The servers render the frames and stream back an encoded video to the player. The latter interacts with the servers through Human Interface Device (HID), and uses a decoder to display the streamed video. The second solution is known as *computation offloading* that consists in migrating parts of tasks of the game engine (the most resource consuming) to a remote powerful computer hosted in the cloud or at the network edge (using the concept of Edge computing). On the successful execution, the results are sent back to the mobile device for integration with the rest of the application inside the mobile device. In this paper, we unveil an offloading solution to improve the performance of one of the most popular game engines in the market, namely "Unity 3D". Using a testbed composed of a smartphone and a server, we evaluate and compare the performance of the proposed solution by report to the classical solution (i.e., running all the game on the smartphone); particularly focusing on the ability to improve the capacity of the smartphone to run complex games.**

I. Introduction

Video Games are complex, intensive, and real-time applications that need powerful devices to run smoothly. To provide a gamer an illusion of inside an animated world, the rendering engine should perform all the rendering activities in real-time. For a gamer, a good Quality of Experience (QoE) is obtained when the game engine displays frames at a higher rate, more than 30 frames per second (fps) [1]. Thus, the rendering engine has at most $33\,ms$ to generate one frame. Usually, much less time is available, since the bandwidth is also consumed by the other engines such physics and scripting. While dedicated hardware, such as "boxes" (e.g., Xbox and PS4), are able to achieve this rate for best-seller games, running these high-resources consuming games on powerless devices (such as smartphones and tablets) is still a challenge. To overcome the limited mobile device capabilities, one of the envisioned solutions is to use Cloud gaming systems such as *Onlive*[1] and *Gaikai*[2]. Although Cloud gaming offers good performance, it is high bandwidth consuming, latency dependent and limited in the number of games.

An alternative to cloud gaming is computation offloading of part of the gaming tasks to remote server. Computation offloading is gaining ground with the emergence of Mobile Edge Computing (MEC), which locates servers at the network edge allowing to drastically reduce the end-to-end latency. Several works have addressed the problem of computation offloading in the context of mobile Cloud. In *MAUI* [2], the authors presented a dynamic offloading framework operating at the method (i.e. application component) granularity. Similarly, *ThinkAir* [3] introduced a mobile cloud computing framework, with dynamic offloading. The proposed framework clones the smart phone platform in Virtual Machine (VM). It offers a library and a compiler to make easy the adaptation of games. A code generator creates the wrappers and utility functions. A customized native development kit (NDK) is used to convert the ARM-based instructions of the remote methods into x86 instructions. ThinkAir uses *Java reflection* to offload methods based on past invocations. ThinkAir defines four objective functions that combine execution time, energy, and money cost. In [4], [5] the authors propose *DPartner* framework, an automatic partitioning system that rewrites the Java bytecode of monolithic application into a distributed one. The framework operates in three steps; first, it classifies the Java bytecode classes into anchored or movable based on the Java lexicon. Then, the framework clusters the classes regarding the call frequency into different groups representing the game modules. Finally, the framework rewrites the clusters bytecode and packages them into OSGi bundles. These bundles are classified into anchored or movable modules according to classes. The framework defines a proxy, which rewrites the classes to create new interfaces, and duplicates classes on both sides (client and server). The framework offloads all the bundles that improve the performance and reduce the energy consumption. Another framework proposing to offload Graphics Processing Unit (GPU) computation to remote servers has been introduced in *Kahawai* [6]. It uses a collaborative rendering, which combines both server GPU and mobile device GPU outputs to render frames. Kahawai uses two techniques for collaborative rendering; the delta encoding and client-side I-frame rendering. In delta encoding, the mobile device renders frames with low quality encoding,

[1]http://onlive.com
[2]http://gaikai.com

while in the server, the same frames are rendered with high quality. The per-frame difference is streamed to the mobile device to transform frames into high quality frames. In client-side I-frame, both the mobile device and server are rendering frames at high quality encoding, however, the mobile device generates the frames at a low rate compared with the server. The server compresses the frames into a video, replaces the I-frames with empty place-holders, and streams the remaining P-frames to the mobile device, which fills in the missing I-frames and renders the video. Di et al. [7] have proposed *Dust*, a real-time code offloading system for device-to-device. Dust uses a network evaluator component to find the stable linked offloadees, and a task scheduler component, which takes a decision regarding each task of a game. The tasks are annotated by programmers as "@offloadable".

However, the common drawback of these frameworks is the non consideration of 3D First Person Shooter (FPS) games, which are very high resource consuming and real-time interacting. Indeed, in all these frameworks, authors have considered only strategic 2D games like Sudoku, Chess, N-Queens, and Gomuko, which renders one frame for each player movement. In this paper, we introduce a first offloading 3D FPS game system, using one of the most popular game engines in the market namely *"Unity 3D"*. We split the game scene into different Game Objects (GOs) including Non-Player Character (NPC), Player Character (PC), environment, and particles. To select the GOs to offload, we use a heuristic relaying on three main criteria, namely, resource consumption, code dependency, and network latency as detailed later. Furthermore, we introduce a network manager component, which orchestrates the offloading mechanism. This solution offers a promising performance. It is easy to configure, as the GOs to offload are added to the server through a network manager, and scalable as all the games could be modified easily to fill in this architecture. Lastly, the proposed solution is not bandwidth consuming, as only command packets are exchanged over the network.

The rest of the paper is organized as follows. In section II, we present some foundations of the game's world. We describe our methodology in section III. Section IV aims to highlight our proposed solution. Section V presents the performance of our framework. Finally, Section VI draws a conclusion around the contribution in the paper.

II. Game Engines Background

To better understand our contribution in this paper, we introduce in the following some concepts and keywords that we will use through the paper.

A. Interactivity and Framerate

The *interaction delay*, defined as *the elapsed time between a user action is captured by a HID, and the moment that the result of this action appears on the screen*, is central in gaming. Studies [8]–[10] have shown that the acceptable delay depends on the game *genre* and varies from 100 to $200\,ms$ and even up to $500\,ms$ for Role-Playing Games (RPGs) and Massively Multiplayer Online Games (MMOGs). FPS games require low delays (less than $100\,ms$), since the gamer is immersed in the scene, and a high interaction delay will degrade the QoE [11]. Regarding this constraint, the management of FPS games has received scientific efforts [12], [13]; therefore the interaction delay will be considered as a metric in this work.

The *framerate* is another key criteria to ensure high QoE for the gamers. Indeed, the latter are immersed in an animated world when the game engine generates a high number of fps. Less than $30\,fps$ is seen as non-tolerable by players [1]. For the interactive multimedia, the High Frame Rate (HFR) combined with the High Dynamic Range (HDR) technology can deliver up to $240\,fps$ [14]. The *rendering pipeline* [15] represents the engine responsible for generating frames; every $x\,ms$ the graphic pipeline displays one frame, with x ranging from 33 to $10\,ms$. The framerate will be also considered as a metric in this work.

B. Main Modules

A game engine is a combination of different modules depending on the game *genre*. Messaoudi et al. [16] have identified some modules that are common to most game engines, and classified them into different families. Some of these module families are written by game developers that include: (i) the *Artificial Intelligence (AI)*, which emulates an artificial and intelligent behaviour of the NPC to learn, to interact, to fight, and to survive; (ii) the *scripts* represent the game scenario. Game developers detail, in a scripting language, the control flow of the game, from the instant wherein the gamer command is captured by HID until displaying a frame on the screen. (iii) *Animations* are used to make objects, dynamic in the game. They emulate movement or reshape objects.

Some other families of modules are leveraged as a third-party Software Development Kits (SDK) and middleware accessible through Application Programming Interfaces (APIs). These families of module represent an abstraction layer common to all games created within a given framework, aiming at preventing the game developers from spending time in low-level programming. These modules include the following: (iv) *physics*, which simulates the physics laws to make the game as realistic as possible. Physics uses collisions and rigid body dynamics[3]. Without physics module, objects would interpenetrate, leading to block interactions with the virtual world. (v) *Multimedia rendering* modules are responsible for generating the graphical and audio elements of the game. Rendering is a resource-consuming module in game engines, since the 3D-scene undergoes several transformations through the rendering pipeline before getting displayed on the screen [17]. (vi) *Inputs* convert the physical commands applied by the gamer on his HID (including gamepad, joystick or keyboard) into logical game functions, and forward them to the engine system. Finally, (vii) *networking* modules define a set of routines and protocols that enable interactions with a remote server to share a game instance between multiple players.

[3]https://gafferongames.com/post/physics_in_3d/

C. Scene Representation

A real-world scene is a projection of dynamic foreground (the *dynamic GOs*) on a layout of a static background or *static GOs*. The static background layout is crucial in video games, as it brings the player inside an immersive world. The game's world populates different types of GOs, through which the gamer explores the virtual world. The game world as a whole presents perceptual stimuli to the player, which experiences a degree of presence over the objects of this world that he can manipulate. These objects include: (i) a *PC*, which is a fictional character, controlled by the gamer. Generally, these characters are based on real persons, such as sportive and historical persons. FPS games use black characters without any characteristic. (ii) A *NPC* is controlled by the computer through an AI and triggered by specific actions. A NPC may define an *enemy*, a *partner* or a *support* character, depending on whether the NPC opposes the PC in duels, helps the PC in its adventure or assists the storyline of the game. (iii) *Environment* represents the virtual static and realistic area where the game takes place. (iv) *Lights* are a key step to produce a realistic scene. The light sources are simple objects, defined in the world space, which are a combination of color, intensity, direction, focus, and position. (v) *Particles* are amorphous objects such as smoke clouds and sparks. They are animated in a rich variety of ways that vary in position, orientation, and size from frame to another. (vi) *Sound sources* are in charge of reproducing what the player would like to hear such as a car engine sound or a background music. (vii) *Camera* is a GO that displays what it currently *seen* on the screen. The camera can move and rotate around, hence the displayed view moves and rotates accordingly. The area seen by the camera defines a truncated pyramid known as a *frustum*.

III. Methodology

We describe now the game, platform, and qualities encoding that have been used in our experiment, and the methodology undertaken to offload modules to a remote server.

A. Game

We modified the multiplayer FPS game[4] to make player characters fighting together against a NPC inside an arena. The player character is a robot with blasters flying inside the arena. The NPC is a humanoid avatar triggered by the player characters when they are near to its position. The game scene is depicted in Figure 1. We summarize the main characteristics of this game in Table I.

TABLE I: **Game characteristics**

# of players	Dimension	Type	Rendering	Physics	Scripts
multiplayer	3D	FPS	++++	+++	+++

[4]https://www.youtube.com/watch?vŪK57qdq_lak

B. platform

To evaluate the performance of the proposed solution, we installed Unity 3D engine v5.4 on top of a Dell PC tower. The installed engine is used to compile the tested game and generate two different instances; the first one runs on the server (Dell PC tower), while the second one runs on the smartphone HTC One M8. The configuration of these devices is given in Table II.

TABLE II: **Platforms characteristics**

Platform	CPU	GPU	RAM	OS
HTC one (M8)	Quad-Core 801 Snapdragon, 2.3GHz	Adreno 330	2GB	Android 4.4.2
Dell PC tower	Intel Core i7, 3.4 GHz	3x NVIDIA GeForce GTX 780 Ti, 3GB	16GB	Windows 8.1 Pro

C. Quality Encoding

We generated 10,000 frames for two encoding qualities; a *good* and *fast* quality. The good quality is encoded with high parameter settings, which generate a reasonable framerate, i.e., around $30\,fps$. The fast quality, configured with reduced requirements, produces inferior visualization results, hence obtain a maximum framerate. Unity 3D achieves these two qualities through different parameters as described in [17].

In light of what is stated in the background section, the following tackles the questions: *how we can improve the performance of a game through modules offloading?* or *what is the optimal location (on the mobile device or on the server) of each module of the game engine?* To answer efficiently this question, we introduced the following criteria:

1) ***Resource consumption.*** Usually, the gameplay is concentrated within dynamic objects, which are high resource consuming. Rendering these GOs is complex in video games [17]. Each object in the scene is approximated by triangle meshes. The more triangles are used to approximate an object, the better is the approximation, but more is the processing.
2) ***Code Dependency.*** Games depend on hardware (e.g., sensors) and software known as *libraries* and *SDKs*, but also interact with players via *User-Interfaces (UIs)*, which manage the HIDs. According to this, we distinguish three classes of *non-transferable modules*; modules involving UIs [2], [18]; modules interacting device sensors [19]; and modules depending on local APIs [20], [21].
3) ***Bandwidth consumption and Network latency.*** Some GOs, if they are offloaded to the server, need high network communication with the mobile device, which increases the bandwidth consumption and the interaction delay. Particles are an example, they are 2D images generated and animated in large number. Several modules are interacting together to make their behaviour, which leads to high communication between modules of the game engine.

Fig. 1: **Game screenshot**

IV. Proposed Framework

Figure 2 presents a global view of the proposed architecture. At the beginning, a connection is established between the mobile device and the server via a network manager. This latter is a set of scripts responsible for remotely instantiating GOs, and orchestrating the offloading process.

When the connection is established, a *game manager* script is executed on the mobile device. It starts the execution of the local GOs, while it requests the network manager to start computation on the server. Therefore, remote GOs are rendered on the server at default coordinates. Start playing the game, input modules capture the gamer inputs and send commands to the server. On the server side, both the GOs and the outputs of the involved modules (modules used by each GO to compute its behaviour) are updated with the gamer commands. Thus the network manager captures these results and injects them on modules, located in the mobile device, interacting with the remote GOs. At the end, a frame composed of the local and the remote GOs is rendered on the mobile device. This process is repeated for each frame until a disconnection of the gamer. In this case, the GOs are destroyed on both client and server.

The network manager uses both *Remote Procedure Call (RPC)* and *GigE Vision Streaming Protocol (GVSP)* [22] carried over User Datagram Protocol (UDP) for communication between the mobile device and the remote server, with a multi-channel design supporting a variety of levels of Quality of Service (QoS), and a flexible network topology supporting peer-to-peer and client-server architectures. RPCs are used to update some module entries and variables such as (N)PC *health level* or *weapon state*. GVSP is used to stream *OpenGL ES* commands from the remote server to the mobile device, then the latter will prefetch these commands and inject them inside the rendering pipeline to draw a frame on the mobile device.

A. Proposed Heuristic

Each GO involves a number of modules to compute its behaviour and to draw its shape. These modules may differ in number and family between the GOs. We propose to enclose for each GO, the requested modules inside clusters. Our heuristic, computed by the network manager, dissects the possibility to offload or not a cluster, with respect to the code dependency constraint. This solution is a cluster decision-making; that is, it focuses on each GO independently from

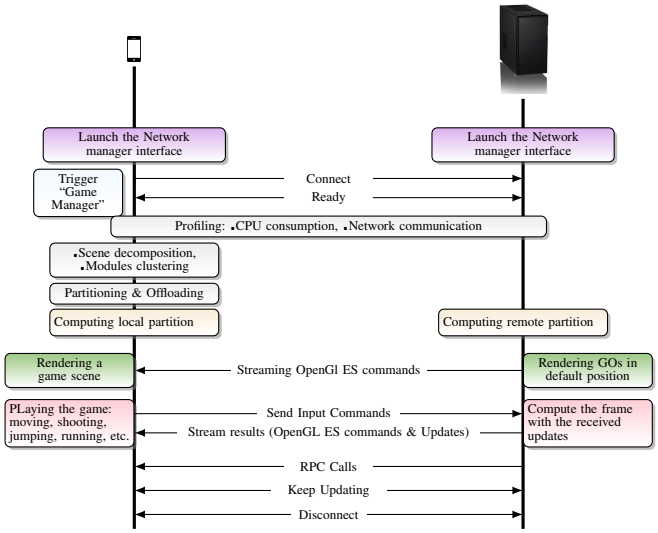


Fig. 2: **Global overview of the architecture**

the others, since we need to decide rapidly to offload a cluster or not, to avoid any additional delay induced by more sophisticated algorithms like Integer Linear Programming (ILP) or graph resolution. The proposed algorithm accepts as input a cluster, j (the set of modules requested by the object GO_j) and returns a binary decision x_j (i.e., to offload or not the cluster j). The concept of this algorithm is simple; if either the network latency or the time to send/receive data is higher than the local execution time of a cluster, then offloading the cluster will not improve the performance, thus $x_j = 0$. Otherwise, if both latency and time to exchange data are less than the cluster execution time, the algorithm checks if a gain is achieved when offloading the cluster. The *offloading gain* is the difference between the local cost and the offload cost, this latter includes the communication cost and the remote execution. It is given by:

$$TG_{GO_j} = \left(\frac{s-1}{s}\right) \times \sum_{i=1}^{k_j} T_i - \left(\frac{d_i}{UL} + \frac{r_i}{DL} + RTT\right) \quad (1)$$

where s is the speed ratio between the mobile device and the remote server, T_i is the execution time of the module i that belongs to cluster j, k_j is the number of modules enclosing the cluster j, and d_i and r_i represent respectively, the data to send (receive, respectively) on the uplink, UL (downlink DL, respectively) bandwidth.

If offloading a cluster will achieve a gain, then it will be offloaded (i.e., $x_i = 1$).

Algorithm 1 Offloading Decision

Inputs: $GO_j : \{T_i, i = 1, 2, ..., k_i\}$
Outputs: $Decision\, x_j$
if $((RTT < \sum_{i=1}^{k_j} T_i)\ \&\ (d_i/B_s + r_i/B_r < \sum_{i=1}^{k_j} T_i))$ **then**
  if $(TG_{GO_j} > 0)$ **then**
    $x_j = 1$
  else
    $x_j = 0$
else
  $x_j = 0$
return x_j

V. Performance

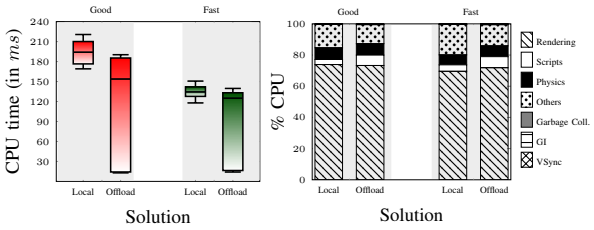
Now, we present the results of our measurement campaign regarding the CPU consumption and network communication needed to generate one game frame.

A. CPU Consumption

This section discusses the CPU consumption per frame and per module, for local and remote execution, under the two encoding qualities. Figure 3a presents the time (in ms) needed to generate one frame. We used box-plots as we want to focus on the *variability* of the CPU consumption per frame. The aim is to quantify the *stability* of our framework. The box plot includes the 10^{th}, 25^{th}, median, 75^{th}, and 90^{th} percentiles of these times. On the other hand, Figure 3b shows (in %) the time spent by the aforementioned modules to contribute to the frame generation. We observe two things:

- *Performance improvement*. As seen in Figure 3a, our framework improves the performance by up to 21%. Indeed, more than 50% of frames are generated in less than $154\,ms$ ($125\,ms$, respectively) for the good (fast, respectively) quality. However, the framework is not enough stable as the *IRQ*[5] and the *range*[6] are high values ($143.97\,ms$ and $151.52\,ms$, respectively).
- *Rendering consumption*. As Figure 3b is showing, rendering is the main consuming module. Indeed, this module is responsible for up to 70% of the CPU consumption for both executions (i.e., whether or not the game is offloaded) under the two encoding qualities. This high CPU consumption represents a concern in mobile gaming. For the other modules that are not related to rendering, they represent less than 30% of the CPU consumption of the demanding games.

[5]Interquartile Range (IQR) corresponds to the range of half of the scores around the median (the difference between the 75^{th} and the 25^{th} percentiles)
[6]The difference between the highest (90^{th}) and the lowest (10^{th}) score



(a) Consumption per frame (b) Consumption per Module

Fig. 3: CPU-Time consumption per frame and module

B. Network Communication

To test the network performance of our framework, we captured the network load incurred by offloading the game modules to the remote server. Packets were captured using *"Wireshark"*. We limited the captures to $200\,s$.

We plot in Figure 4 the bitrate (in $bytes/s$) for the network traffic between the mobile device and the remote server for the two qualities encoding.

We captured both the uplink and downlink traffic between the client and the server. In the downlink direction, the packet size varies between 54 and $394\,B$, and the median is $88\,B$ for both the fast and the good quality. On the uplink direction, the packet sizes vary between 60 and $162\,B$ (60 and $228\,B$, respectively) with a median about $86.5\,B$ ($83.9\,B$, respectively) for the fast (good, respectively) quality. As stated above, only commands are streamed to the mobile device. Therefore, the variation in the bit rate depends on the number of offloaded GOs. Indeed, higher are the GOs offloaded to the remote server, the greater is the number of streamed commands, hence, the higher is the bit rate. Between the two qualities, there is also a variation in the bit rate. We believe that it is due to the difference in the time needed by the server to compute the GOs before streaming the commands. This time depends on the number and type of GOs, which *may* differ between the two qualities encoding.

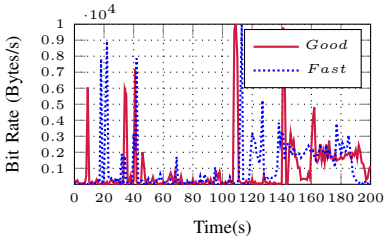


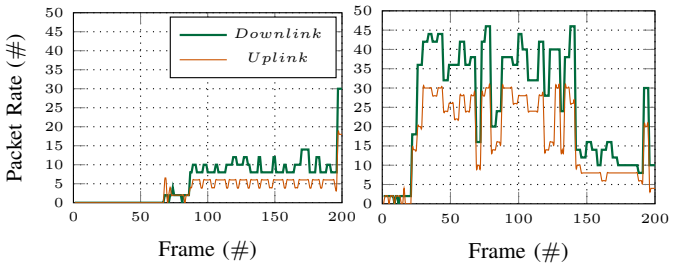
Fig. 4: Packets load per a tick of 1 second interval

Figure 5a (5b, respectively) illustrates the number of packets captured on the uplink and downlink directions for each frame encoded with the good (fast, respectively) quality. We make three main observations:

- *Downlink rate is higher than the uplink rate*. The average rate for the fast (good, respectively) on the downlink direction is around 17.13 (3.94, respectively)

$packets/frame$, while on the uplink direction, this average is about 11.51 (2.39, respectively) $packets/frame$. This is somehow obvious as our framework relies on the remote server to stream back rendering commands and update various modules. The *uplink* traffic represents only input commands and (N)PC variables.

- *The server follows the client pace*. Despite the powerful capabilities of the remote server, it has to follow the client pace to stream back the results, since the network manager synchronizes between them. Indeed, the server is triggered only when it receives an event (input commands or RPC) from the mobile device.
- *Server requested only on performance enhancement*. Both Figures 5a and 5b exhibit two behaviors; less and high network communication. When the heuristic estimates that no offloading gain can be achieved, then the whole game is computed on the mobile device, hence the server is in an idle state, that is to say, only few control packets are sent by the server. However, when a gain can be achieved, the network manager establishes a communication between the mobile device and the remote server to collaborate in the frame rendering.



(a) Good Quality (b) Fast Qualiy

Fig. 5: Uplink and Downlink packets rate per frame

C. Responsiveness

To better understand how the client and server contributed in the frame generation, it is necessary to determine the interaction delay, which is consumed by several tasks: (i) capture of gamer input actions, (ii) transfer of command(s) to the remote server, (iii) execution of the m offloaded modules (om) on the server and the $n-m$ non-offloaded modules (nom) on the client (where n is total number of modules), (v) stream OpenGL ES commands, and finally (vi) inject the commands in the graphic pipeline and render a frame. This overall interaction delay IDL is divided into three parts:

1) *Processing Delay*, PD, is the maximum time between local and remote execution. Local (remote, respectively) execution time is the sum of non-offloaded (offloaded, respectively) modules execution delays as given in Equation (2). We used the *visual studio* profiler to extract these delays.

$$PT = max\left(\sum_{i=1}^{n-m} t_i^{(nom)}, \sum_{i=1}^{m} t_i^{(om)}\right) \quad (2)$$

2) *Updating Delay*, UD, is the time spent to update *locally* modules inputs (such as the scripting module). We instrumented the code of the game to identify these delays. The UD is then, equal to the sum of all the networked update times $t_i^{(uom)}$ for om_i, given by Equation (3).

$$UT = \sum_{i=1}^{m} t_i^{(uom)} \quad (3)$$

3) *Communication Delay*, CD, corresponds to the command streaming delay, input commands forwarding delay, and other control communication delay. It is the sum of the RTT and the time to send an amount of Q data as shown in Equation (4).

$$CT = \frac{(Packet\,Rate) \times (Packet\,Size)}{Bandwidth} + RTT \quad (4)$$

The ID is given by Equation (5). It corresponds to the average time obtained in Figure 3a.

$$RT = PT + UT + CT \quad (5)$$

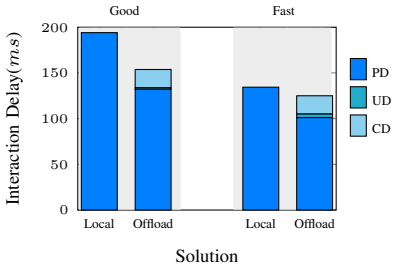


Fig. 6: Average of interaction Delay

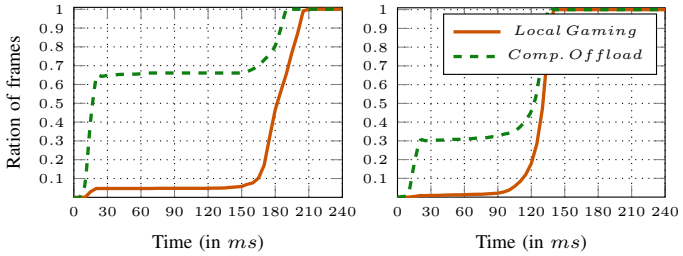
Figure 6 illustrates the average ID achieved under both the local and remote execution. We observe that: (1) our framework achieves a small UD, at most $3.91\,ms$ ($1.62\,ms$, respect.) for fast (good, respect.) quality. This time represents $3\,\%$ ($0.93\,\%$, respect.) of the ID. (2) The PD is $3\times$ longer than what we expected, $105.79\,ms$ and $151.48\,ms$ for respectively fast and good quality. Since we leverage powerful remote server, we hope closer performance to cloud gaming solutions, as the consuming modules are computed on the remote server and only command packets transit over the network. We believe that this drawback is due to the injection of the OpenGL ES commands in the graphic pipeline.

D. Framerate

We conclude the performance section by summarizing the different results via the framerate performance.

Figures 7a and 7b depict the ratio of frames in a population of 1000 frames that are generated in less than $x\,ms$ for the good and fast quality, respectively. When using our heuristic, the game engine generates more than 65% (30%, respectively) of frames in less than $33\,ms$ for the good (fast, respectively) quality in comparison to the local execution, where the engine generates less than 10% of frames for the two qualities. Some frames are composed of several GOs that highly communicate

through various modules which increase the interaction delay. Indeed, as the figures are showing, up to 35%, 70% of frames for the good, respectively fast quality need until $210\,ms$ and $150\,ms$ to be rendered. Our heuristic, in this case, offloads only a few clusters, as the communication cost is higher than the computational cost, or because no offloading gain is obtained for these frames.



(a) Good Quality (b) Fast Qualiy

Fig. 7: CDF for frame generation

VI. Conclusion

In this paper, we proposed a solution to play best-seller 3D games with high quality encoding, on powerless devices via offloading computation to a remote server. The concept is to identify modules involved by each GO in the game scene. Then, decide to offload them (as a whole) or not, depending on three main criteria: resource consumption, network communication, and code dependency. The mobile device and the server are synchronized through network manager, which orchestrates the offloading. The solution is scalable and adaptable to the network latency as only modules improving performance are offloaded. It supports mobility since network packets are automatically routed to the mobile device.

Our framework is still a work in progress, with several performance optimizations still possible. In the future we will deal with two main issues; the network latency and the rendering activity. One of our motivations is to leverage MEC architecture as an enabler for low-latency computation offloading based-games. To this aim we want to rely on our framework proposed in [23] to orchestrate the offloading process.

References

[1] M. Claypool and K. Claypool, "Perspectives, frame rates and resolutions: it's all in the game," in *Proceedings of the 4th ACM International Conference on Foundations of Digital Games*, 2009.

[2] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys 2010), San Francisco, California, USA, June 15-18, 2010*, 2010, pp. 49–62.

[3] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proceedings of the IEEE INFOCOM 2012, Orlando, FL, USA, March 25-30, 2012*, 2012, pp. 945–953.

[4] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring android java code for on-demand computation offloading," in *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, 2012, pp. 233–248.

[5] Y. Zhang, G. Huang, W. Zhang, X. Liu, and H. Mei, "Towards module-based automatic partitioning of java applications," *Frontiers of Computer Science*, vol. 6, no. 6, pp. 725–740, 2012.

[6] E. Cuervo, A. Wolman, L. P. Cox, K. Lebeck, A. Razeen, S. Saroiu, and M. Musuvathi, "Kahawai: High-quality mobile gaming using GPU offload," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2015, Florence, Italy, May 19-22, 2015*, 2015, pp. 121–135.

[7] D. Huang, L. Yang, and S. Zhang, "Dust: Real-time code offloading system for wearable computing," in *2015 IEEE Global Communications Conference, GLOBECOM 2015, San Diego, CA, USA, December 6-10, 2015*, 2015, pp. 1–7.

[8] M. Claypool and K. T. Claypool, "Latency and player actions in online games," *Commun. ACM*, vol. 49, no. 11, pp. 40–45, 2006.

[9] Y. Lee, K. Chen, H. Su, and C. Lei, "Are all games equally cloud-gaming-friendly? an electromyographic approach," in *Proceedings of the 11th ACM Netgames Workshop*, 2012.

[10] P. Quax, A. Beznosyk, W. Vanmontfort, and R. Marx, "An evaluation of the impact of game genre on user experience in cloud gaming," in *Proc. of the IEEE Int. Games Innov. Conf. (IGIC)*, 2013, pp. 216–221.

[11] M. Jarschel and D. Schlosser, "An evaluation of qoe in cloud gaming based on subjective tests," in *Proceedings of the 5th Conf. on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, 2011.

[12] G. J. Armitage and A. Heyde, "REED: optimizing first person shooter game server discovery using network coordinates," *TOMCCAP*, vol. 8, no. 2, p. 20, 2012.

[13] Y. Li, X. Tang, and W. Cai, "Play request dispatching for efficient virtual machine usage in cloud gaming," *IEEE Trans. Circuits Syst. Video Techn.*, vol. 25, no. 12, pp. 2052–2063, 2015.

[14] D. Q. M. Lantin and A. G. S. Arden, "High frame rate (hfr), a white paper."

[15] H. Pfister, M. Zwicker, J. Van Baar, and M. Gross, "Surfels: Surface elements as rendering primitives," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 2000, pp. 335–342.

[16] F. Messaoudi, G. Simon, and A. Ksentini, "Dissecting games engines: The case of unity3d," in *Network and Systems Support for Games (NetGames), 2015 International Workshop on*, Dec 2015, pp. 1–6.

[17] F. Messaoudi, A. Ksentini, G. Simon, and P. Bertin, "Performance analysis of game engines on mobile and fixed devices," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 13, no. 4, pp. 57:1–57:28, Sep. 2017.

[18] S. Ou, K. Yang, and A. Liotta, "An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems," in *4th IEEE International Conference on Pervasive Computing and Communications (PerCom 2006), 13-17 March 2006, Pisa, Italy*, 2006, pp. 116–125.

[19] M. Othman and S. Hailes, "Power conservation strategy for mobile computers using load sharing," *Mobile Computing and Communications Review*, vol. 2, no. 1, pp. 44–51, 1998.

[20] X. Gu, A. Messer, I. Greenberg, D. S. Milojicic, and K. Nahrstedt, "Adaptive offloading for pervasive computing," *IEEE Pervasive Computing*, vol. 3, no. 3, pp. 66–73, 2004.

[21] S. Ou, K. Yang, and Q. Zhang, "An efficient runtime offloading approach for pervasive services," in *IEEE Wireless Communications and Networking Conference, WCNC 2006, 3-6 April 2006, Las Vegas, Nevada, USA*, 2006, pp. 2229–2234.

[22] W. He, K. Yuan, H. Xiao, and Z. Xu, "A high speed robot vision system with gige vision extension," in *2011 IEEE International Conference on Mechatronics and Automation*. IEEE, 2011, pp. 452–457.

[23] F. Messaoudi, A. Ksentini, and P. Bertin, "On using edge computing for computation offloading in mobile network," in *2017 IEEE Global Communications Conference, GLOBECOM 2017, Singapore, December 4-8, 2017*, 2017, pp. 1–7.