Performance Analysis of Game Engines on Mobile and Fixed Devices

FAROUK MESSAOUDI, IRT b◇com
ADLEN KSENTINI, Eurecom
GWENDAL SIMON, Telecom Bretagne
PHILIPPE BERTIN, IRT b◇com, Orange Labs

Mobile Gaming is an emerging concept, wherein gamers are using mobile devices, like smart phones and tablets, to play best seller games. Compared to dedicated gaming boxes or PCs, these devices still fall short of executing newly complex 3D-video games with a rich immersion. Three novel solutions, relying on cloud computing infrastructure, namely Computation Offloading, Cloud Gaming, and Client-Server architecture will represent the next generation of game engine architecture aiming at improving the gaming experience. The basis of these above-mentioned solutions is the distribution of the game code over different devices (including set-top-boxes, PCs, and servers). In order to know how the game code should be distributed, advanced knowledge of game engines is required. By consequence, dissecting and analyzing game engine performances will surely help to better understand how to move in these new directions (*i.e.*, distribute game code), which is so far missing in the literature. Aiming at filling this gap, we propose in this paper to analyze and evaluate one of the famous engines in the market, *i.e.*, "Unity 3D". We begin by detailing the architecture and the game logic of game engines. Then, we propose a test-bed to evaluate the CPU and GPU consumption per frame and per module for nine representative games on three platforms, namely a stand-alone computer, embedded systems and web players. Based on the obtained results and observations, we build a valued graph of each module, composing the Unity 3D architecture, which reflects the internal flow and CPU consumption. Finally, we made a comparison in term of CPU consumption between these architectures.

CCS Concepts: • **General and reference** → **Measurement**; **Evaluation**; **Experimentation**; **Performance**;

Additional Key Words and Phrases: Computation Offloading, Cloud Gaming, Games, Unity 3D, Rendering

ACM Reference format:
Farouk MESSAOUDI, Adlen KSENTINI, Gwendal SIMON, and Philippe BERTIN. 2016. Performance Analysis of Game Engines on Mobile and Fixed Devices. *ACM Trans. Web* 9, 4, Article 39 (March 2016), 28 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

Game engines, like Unity 3D [1], are well-established tools to generate interactive, fully-multimedia environments that range from games to serious health applications. Game engines are complex software, which consume a lot of resources (including Central Processing Unit (CPU), Graphics Processing Unit (GPU), energy, and memory) to meet the demand of gamers. Indeed, the Quality of Experience (QoE) depends on the interactivity (especially the delay between the command of an action and the visible result of this action on the scene) and on the multimedia quality (resolution,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2009 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
1559-1131/2016/3-ART39 $15.00
https://doi.org/0000001.0000001

frame rate, and also the level of details in the scene). Therefore, game developers design their best-seller games so that the game engine exploits the dedicated hardware that is typically present in “boxes" (e.g. Xbox, PS4), however resource-limited devices such as mobile devices (smartphone and tablet) cannot run these games, or at a much lower quality. Game engine developers are looking for solutions to address this issue and to embrace the mobile market, which represents billions of dollars of revenues.

To get rid of the limitations of end-user devices, one of the envisioned solutions is to leverage cloud computing infrastructure. Developers of games have three options:

- *Computation offloading* consists in offloading some modules of the game engine to a nearby server. It requires interactions and data exchanges between distant modules.
- *Cloud gaming* consists in offloading all the modules to a remote server and streaming back an encoded video to the client.
- *Client-server architecture*, wherein the game engine runs almost entirely on the client side and the server updates positions of players in a multi-player context and/or updates the position of the Non-Player Characters (NPCs).

Whilst the third approach (client/server) is widely implemented and mastered (research work mostly addresses network management [16, 38, 47]), the two first approaches need more investigations regarding the software aspect of the game engines. Computation offloading requires low latency communication with the remote servers, which might be solved using Mobile Edge Computing (MEC) [9, 21] and Fog Computing [5, 24]. However, deciding which module or function should be offloaded remains an open issue. The latency issues of cloud gaming have also been frequently studied [13, 20, 37, 39]. But, the management of virtualized resources (known as *server consolidation*) to prevent performance degradation and resource congestion has been rarely addressed [17, 54].

To identify the best implementation option, developers should take into account the nature of the game and the considered platform. However, the body of literature related to performance evaluation for game engines is surprisingly small with regard to the significance of the gaming industries. In this regard, we may mention two position papers that called for research on game engine architecture [3, 48], and some other work that focused on applying game engines to specific areas such as serious games [14], research [26], and Virtual Reality (VR) serious applications [14, 23, 30]. The literature is missing work on the architecture and the performance analysis of modern game engines.

To fill this gap, we analyze the performance of a well-known engine, Unity 3D. In this paper, we report our experimentation, where we have run nine representative games with two game quality encoding over several platforms: (i) Dell M4800 laptop running two Operating Systems (OSs) (Windows 7 and Linux Ubuntu 14.04 LTS); (ii) Microsoft Surface Pro running Windows 10; (iii) two web players using Mozilla Firefox and Opera hosted by the laptop Dell M4800 running Windows 7; and (iv) two smart phones HTC One (M8) and Samsung Galaxy S6 Edge running Android OSs. We analyze the performances of these games considering both CPU and GPU, in term of needed time to generate frames. We study the time consumption per modules, according to the type and the function of these modules in the game engine. Then we extract different valued graph, which represent the internal flow of the game engine. Finally, we propose a proof-of-concept of the cloud gaming and computation offloading architectures, and we compare the performance results with the client-server one. This study, which extends our previous work [31], is a primary reference for researchers who explore computation offloading and server consolidation.

One of the main novel contributions of the present paper in regards of our previous work [31] is a proposal to classify games with respect to their behaviours and to the different architectures: *cloud gaming*, *computation offloading*, and *client-server*. We distinguish two parameters that characterize

a given game and enable its classification into one of these three architectures: *playability*, which represents the requirement in term of CPU and GPU time to generate one frame with respect to a given genre-dependent threshold and *resource variability*, which gives the difference in CPU and GPU time to generate two frames.

The paper is organized as follows: Section 2 gives some background about game engines and rendering activities. Section 3 presents the experimental platform we used to measure the performance of Unity 3D-based games. The performance analysis is given in Section 4. Section 5 studies the feasibility of computation offloading. Section 6 details experiment results that validate both our proposed game classification and our conclusion on offloading feasibility. Finally, we discuss the results and open perspectives regarding this first step toward next-generation game engines in Section 7.

2 BACKGROUND: GAME ENGINES

Game engine can be defined either as a framework for game creators or as a piece of code for gamers. Accordingly, two definitions arise:

- The game engine is the set of tools (*i.e.*, including low-level libraries, User-Interface (UI) editors, and game multimedia management tools) that facilitate the work of a game developer in the process of creating a new game. The community of game developers considers thus a game engine as a framework or a platform. The framework provides an abstraction layer between the game content (*i.e.*, multimedia content and main scripts) and the underlying hardware.
- The game engine is the set of software and data that runs on a device to provide the game to an end-user. The community of gamers considers a game engine as a piece of code. The border between the game and its framework is often confused. Some frameworks offer a clear separation, while others do not. All games created by Unity 3D share similarities, making them consistent Unity 3D game engines. We focus in this paper on typical Unity 3D software, which we refer to as a game engine.

The delay for the result of an action to appear in the screen is central in gaming since the QoE is linked to interactivity [20]. We distinguish different *genres* of games with different requirement. An action game can be a First Person Shooter (FPS) or a Third Person Shooter (TPS) depending on whether the player is immersed in the scene or the avatar representing the player is visible. Studies [12, 25, 36] 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 Game (RPG) and Massively Multiplayer Online Games (MMOG). Given their specific constraints regarding short delays, the management of FPS has received scientific efforts [4, 27]. In this paper, we tested several game genres and considered these requirements.

Another key criteria for high QoE is the frame rate. Gamers get the feeling of immersion in an animated world when the game engine generates a high number of frames per second (fps). Less than 30 fps is widely seen as non-tolerable [11] and the recent trends in video and interactive multimedia is to deliver frame rate up to 90 fps [33]. The *rendering pipeline* (see Section 2.3) is the part of the game engine that generate one frame every x ms (x ranges from 33 to 10). A large part of our measurement study is on the time needed to generate one frame.

2.1 Game Architectures

We focus now on the game architectures. Figure 1 depicts the characteristics of the discussed architectures by indicating the location of the various modules of the game engine and the used mechanisms to communicate and exchange data or commands.

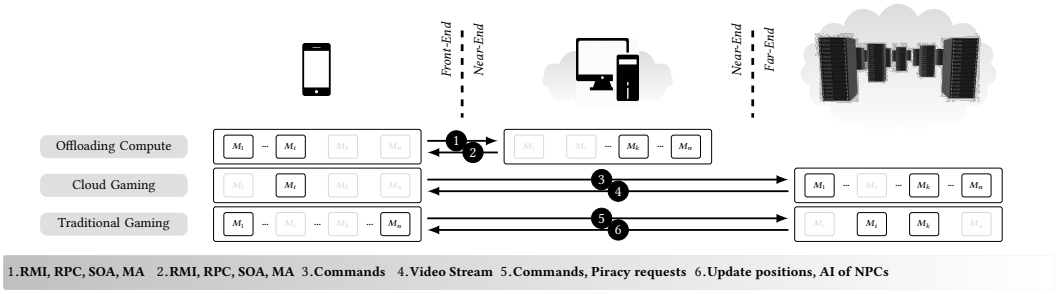


Fig. 1. **Comparison of gaming architectures**

Traditional client-server. The game engine runs almost entirely on the client side and the server assists the game engine, either as a secured session manager or as a multi-player gaming enabler. When the server is a session manager, it manages authentication, updates game assets (for example new levels and new graphical environments) and matchs fighting players [49]. As a multi-player gaming enabler, the server takes a stronger role: **(i)** it gets all the gamer input commands and forwards them to the other players, **(ii)** it implements anti-cheaters policy and acts as referee [52], and **(iii)** it manages NPC. Several architectures exist to deal with scalability. (1) *client-server* architecture with a single server [46], (2) *mirrored client-server* [35, 50, 51], and (3) *peer-to-peer* architecture [7, 53].

Cloud gaming. The entire game engine runs in a remote server. The client includes only UI command and a video decoder. This architecture is similar as an interactive video delivery system. The player sends the commands to the server. The engine first converts the client commands into appropriate in-game actions, then computes the game logic, and renders the game scene. This scene is then compressed by a video encoder and forwarded to a video streaming module, which finally streams it to the client. The client decodes the video and displays the frames [10, 18, 29].

Cloud gaming solution relies on the concept of *resource consolidation*, which consists of using the virtualization technology: each game engine is encapsulated within a virtual machine (or a container) and dynamically mapped onto a pool of physical resources including CPU, GPU, memory, and Inputs/Outputs (I/O). Hence, these game engines concurrently share the server resources through the hypervisor. Studies have shown that virtualization techniques barely degrade the performance compared to a configuration where the game engine is the only program running in a bare-metal [44].

Two main challenges arise with the cloud gaming architecture: the delay and the resource use. In comparison to the delay issue in the traditional client-server architecture, cloud gaming adds extra-delay due to video encoding and decoding. Despite various proposals [28, 43], both processes take around 30 ms [45]. With respect to the network communication delays (around 50 ms in average [8]), the overall delay is close to the threshold at which QoE is reported to degrade [13, 20]. Regarding resource use, the Cloud Gaming Providers (CGPs) have to find a trade-off. The smaller is the number of games concurrently running on a server, the higher can be the quality setting, but the higher are also the cost. Moreover, the CGPs have to reserve some free spare resources in servers, otherwise some games may interrupt due to concurrent workload peaks. One of our motivations is to get a better understanding of the load variability and the internal modular structure of the game engine so as to make server consolidation more efficient and less risky.

Computation offloading. A subset of the game engine modules run on nearby server(s) [2, 22, 41]. Depending on specific criteria, including network bandwidth, latency, and processing time, the

engine is spread into client and server partitions. The server hosts the modules that improve the responsiveness of the game, and the client hosts the remaining modules, typically those interacting with the device components or with the gamer (like UI). The two sides exchange data during the execution. Whenever needed, the server can call a module, or download it using mechanisms such as Remote Procedure Calls (RPC), Remote Methods Invocation (RMI), Service Oriented Architecture (SOA), and Mobile Agents (MA). The partitioning requires profiling the whole game.

The computation offloading solution presents two constraints. First, as in Cloud gaming, computation offloading requires low latency. Second, the game engines should tolerate module partitioning. One of the advantages is that module partitioning allows a better exploitation of resources, especially by distinguishing the cutting-edge modules that require specific GPU hardware from the standard ones that can accommodate any CPU configuration. Remote graphic rendering is not a new topic [19], but none of the existing solutions match the expectations for fast rendering high-definition, complex, 3D images. Moreover, to the best of our knowledge, no previous work has addressed the case of offloading non-graphic components of a game engine, like physics module, audio, and object behaviour scripts. One of our motivations is to explore offloading different non-graphic components of a game engine.

2.2 Game Engine Main Modules

A game engine consists of various modules depending on the game genre. An engine designed for a two-person fighting game is different from an FPS engine, an MMOG engine, a Real-Time Strategy (RTS) engine, or racing engine. Nevertheless, we identify some families of module that are common to most of the game engines. Figure 2 shows the architecture of a generic game engine with the main module families.

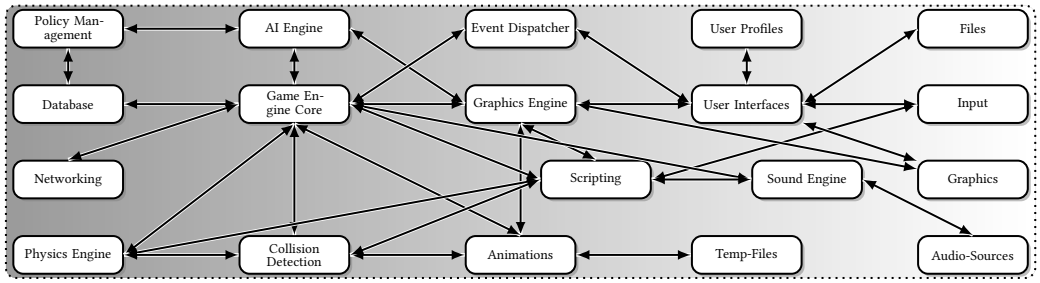


Fig. 2. **Game Engine Architecture**

Some of the module families of the engine are written by the game creator including:

Artificial Intelligence (AI) – these modules emulate an artificial and intelligent behaviour of NPCs, for example, learn and interact socially, exhibit emotions, and even the ability to hunt and the instinct to survive. Different features are used to implement the AI such as: **(i)** *decision-making*, to affect the NPC; **(ii)** *basic perception*; the AI needs some way of perceiving its environment using human-like sensors (including sight, hear, smell, and touch), and navigating using basic mechanisms like *Crash and Turn*, and *pathfinding*; **(iii)** *prediction*; the

ability to effectively anticipate the player behaviour through historical decisions, random guess or backtracking. ALive[1], GAIGE[2], and kismet[3] are three advanced AI modules.

Scripting – these modules contain the gameplay itself. From the captured inputs obtained from the Human Interface Devices (HIDs), the game developer details the series of game content and events in a scripting language, which is specific to every framework. The scripts can be used to create graphical effects, control the physical behaviour of objects or implement an AI system for NPCs.

Animation – any character in a game (whether it is a human, an animal, or a robot) needs animations (*e.g.* move, jump, stand up, and set down). Five basic animations exist; **(i)** *rigid body hierarchy animation*, **(ii)** *skeletal animation*, **(iii)** *sprite/texture animation*, **(iv)** *vertex animation*, and **(v)** *morph targets*. In particular, skeletal animation generates a pose for every bone in the skeleton and passes them to the rendering engine as a set of matrices that are transformed into a final blended vertex position. This process is called *skinning*. It also interacts with the physics engine to simulate *rag dolls*, which are dead animation characters, whose bodily motion is simulated by the physics.

Some module families are common between most games created within a given framework. These modules represent the abstraction layer and prevent game creators from spending time on low-level issues. In particular:

Physics Engine – it includes collision detection and rigid body dynamics. This module aims to make the game world as realistic as possible using the physics laws. Without this module, objects would interpenetrate, leading to block interactions with the virtual world. This module is integrated as a third-party Software Development kit (SDK). Existing physics engines in the market include *Havok*[4], *PhysX*[5], *Open Dynamics Engine (ODE)*[6], and *I-Collide*[7].

Input – a player interacts with the game using HIDs, *e.g.*, joystick, keyboard, mouse, steering wheels, dance pads, and VR sensors. The input module provides a mechanism to customize the mapping between the physical control and logical game functions. It may include a system for detecting chords, sequences, and gestures. These commands are encapsulated as objects and forwarded to the system.

Networking – this set of routines and protocols enables interactions with a server to set up multiple players and share a sense of space, time, and presence through avatars. In MMOGs, the players join the game dynamically through *game sessions* and interact by exchanging commands [32]. The architecture of most of the MMOGs is based on client/server.

Multimedia Rendering – these modules generate the graphical and audio elements of the game. Graphics are complex aspects of a game. 3D games are built over 3D assets created by a design application like *Maya*, *XNA*, *Blender*, and *WPF*. These assets are mixed with other objects like materials, shadows, lights, and animations to create a realistic scene. This engine is designed as a layered architecture, including: **(i)** *Low-Level Renderer*, where the focus is on rendering as quickly and richly as possible a set of geometric primitives; **(ii)** *Visual Effects*, like particle systems, light mapping, dynamic shadows, and colour correction; **(iii)** *Front End* where 2D graphics overlaid on 3D scene such as in-game menu, a console, and in-game Graphical

[1] http://alive.sourceforge.net/
[2] http://game-ai.gatech.edu/
[3] http://www.ai.mit.edu/projects/humanoid-robotics-group/kismet/
[4] http://www.havok.com/physics/
[5] https://developer.nvidia.com/gameworks-physx-overview
[6] httph//www.ode.org
[7] http://www.cs.unc.edu/~geom/I_COLLIDE/index.html

User Interface (GUI). Regarding sounds, the audio engines vary in sophistication. The audio clips are exported in different formats like mono, stereo, wave files (.wav) or ADPCM files (.vag). Existing commercial audio engines include *Quake* audio engine, *XACT*, *SoundR!OT*, and *Scream*.

2.3 Rendering Pipelines

We describe now the rendering pipeline under Unity 3D. We first describe the pipeline that achieves a 3D rendering, and then, we show how Unity 3D implements it.

A rendering pipeline is a combination of successive stages to generate a 2D image for a geometric description of a 3D scene and a virtual camera. Unity 3D is cross-platform: a given game can run on various operating systems and hardware. For each configuration, the engine uses the default rendering pipeline developed for these targets.

- *Direct 3D* is a Microsoft's 3D graphics low-level Application Programming Interface (API). It is used to draw lines, triangles, and points or to launch parallel processing on the GPU. It is the primary competitor of OpenGL. It is used on Microsoft platforms (Xbox 360, Xbox One, and the Windows operating system platforms).
- *OpenGL* is a *Khronos*-developed graphic library. It is the most used API in industry. It is developed to launch a large number of applications on different computer platforms, such as gaming, manufacturing, medical, VR, content creation, and Dynamic Audio-Visual Communication (DAC).
- *OpenGL ES* is a fork of OpenGL. It has been developed for embedded systems, including smart phones, consoles, appliances, and vehicles.
- *WebGL* is a free low-level 3D graphics API based on OpenGL ES 2.0. It is integrated to HTML5 as Document Object Model interfaces. Major browsers (such as Safari, Chrome, Firefox, and Opera) implement this pipeline as a 3D plugin.

Figure 14 in the appendix, describes the graphic pipelines used by Unity 3D (*i.e.*, OpenGL, OpenGL-ES 2.0, WebGL, and DirectX). Each graphic pipelines is represented by different stages along with their interactions. Each pipeline differs by the number of stages and internal flow, but the general idea of rendering remains the same.

Firstly, a series of geometric operations are done to render a collection of geometric primitives as fast as possible. The data (vertices and indices) of 3D objects are approximated by triangle meshes, forming the basic building blocks. The more triangles are used to approximate an object, the better is the approximation but the more processing power is needed. Each object is then centered at the origin of a local coordinate system with local orientation and size. Thereafter, all objects are brought together in a global coordinate system by applying geometric transformations. At the end, the virtual camera in the scene is translated to the origin of the world space.

Secondly, a set of aesthetic effects (lights, materials, shaders, and textures) is applied to the scene. The *lighting* is a key step to produce a realistic scene. The light sources are simple objects defined in the world space with a combination of colour, intensity, direction, focus, and position. This step also includes a repetition of absorbing and reflecting light processes depending on various parameters (*e.g.*, smoothness and material of the surface and incidence angle).

Thirdly, a set of culling operations (*scene graph/culling optimization*) is done. Each object has two sides with respect to camera position: a front and a back side. The front (respect. back) sides are polygons with vertexes ordered in a clockwise (respect. counter clockwise). All the back-face polygons are culled. The culling step is triggered to also decide which object should be discarded from the scene, according to the computation of the *view frustum*. An object inside (respect. outside) the frustum is kept in (respect. completely culled from) the scene, while the object that is between the inside and outside of the frustum is partially clipped.

Finally, a perspective projection is done to render the 3D vertices into a 2D projection window inside the frustum. The vertex coordinates are transformed to place the 2D scene into a rectangular window on the screen, called the *viewport*. The outputs of this stage are pixels. The *rasterization* transforms these pixels into screen coordinates forming a list of triangles that should be checked and coloured.

Unity 3D uses four additional rendering activities, which occur in conjunction with the main pipeline. We summarize these activities in the appendix Section A.1.

3 METHODOLOGY AND TESTBED

We describe now the configuration of the devices that we used in our experiment, the process of running games on these devices, a description of the used games, and our measurement.

3.1 Platforms

To evaluate the performance of the game engines generated by the framework Unity 3D, we installed Unity 3D v5.03 on top of three devices: a Dell Precision M4800 laptop, a Microsoft Surface Pro tablet, and a Dell PC tower. The latter is used as a server in our second experiment (implementation of cloud gaming and computation offloading architectures). We used different configurations as shown in Table 1. We then used the installed frameworks to compile the games and generate the adequate files for a range of OSs: an ".exe" file in Windows 7/8/10, an ".deb" in Linux, an HTML page with javascript files for web players, and an ".apk" for Android devices. Accordingly, we were able to run the games generated by Unity 3D on a wider range of configurations: not only the said Dell M4800 and Surface Pro, but also a smartphone HTC One (M8) and a Samsung galaxy S6 Edge. Moreover, we were able to implement cloud gaming and computation offloading architectures.

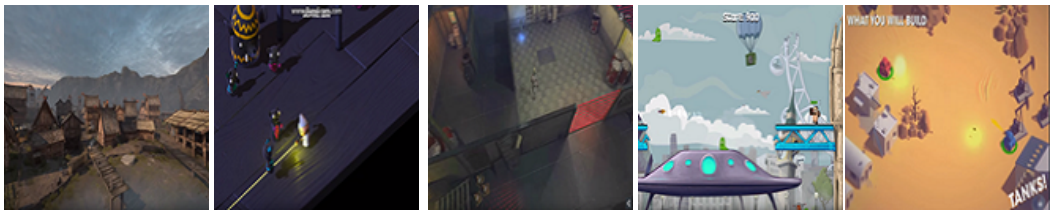
Platform	Target	CPU	GPU	RAM	Pipeline
Windows 7 Pro Ubuntu 14.04 Mozilla Firefox 44.0.2 Opera 35.0.2066.37	Dell Precision M4800	Intel Core i7, 2.8GHz	Nvidia Quadro k2100M, 2GB	16GB	D3D11 OpenGL WebGL WebGL
Windows 10	Microsoft Surface Pro	Intel Core i5 - 3317U, 1.7GHz	Intel HD 4000	4GB	D3D11
Android 4.4.2	HTC one (M8)	Quad-Core 801 Snapdragon, 2.3GHz	Adreno 330	2GB	OpenGL ES 2.0
Android 5.1.1 Lollipop	Samsung Galaxy S6 edge	Octa-Core Exynos 7420 - 2.1 GHz	ARM Mali T760	3GB	OpenGL ES 2.0
Windows 8.1 Pro	Dell PC tower	Intel Core i7, 3.4 GHz	3x NVIDIA GeForce GTX 780 Ti, 3GB	16GB	D3D11

Table 1. **Main characteristics of the used platforms**

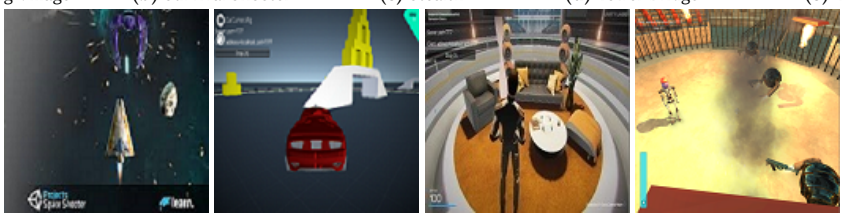
3.2 Games

We selected nine games from the "Asset Store" of Unity 3D for our evaluation. These games have different characteristics, which cover the most representative games in the market. A description of the different games is presented in the appendix. Figure 3 represents a screen-shot of the different games. We summarize the main characteristics of each game in Table 2. For more details about these games, the reader may refer to the Unity 3D asset store.

[8]https://www.youtube.com/watch?v=iV-224nMwN8
[9]https://www.youtube.com/watch?v=iTwtoOO7wXc
[10]https://www.youtube.com/watch?v=LBbPwMmpqQI
[11]https://www.youtube.com/watch?v=uRsspkum8LI



(a) Viking Village (b) Survival Shooter (c) Stealth (d) Tower Bridge (e) Tanks



(f) Space Shooter (g) Car (h) Unity Lab (i) Multiplayer FPS

Fig. 3. **Screen-shot of the different games**

Games	# of players	Dimension	Type	Rendering	Physics	Scripts
Viking Village[8]	1 player	3D	FPS	++++	+	+++
Tower Bridge[9]	1 player	2D	TPS	+	+++	+
Stealth[10]	1 player	3D	TPS	+++	+++	+++
Survival Shooter[11]	1 player	3D	TPS	++	++	++
Tanks[12]	2 players	2D	MMOG	+	++	++
Space Shooter[13]	1 player	2D	TPS	++	+	++
Car[14]	1 player	3D	Racing	++	++	+++
Unity Lab[15]	1 player	3D	TPS	++++	+++	+++
Multiplayer Shooter[16]	multiplayer	3D	MMOG-FPS	+++	+++	+++

Table 2. **Main characteristics of the tested games**

3.3 Game qualities

For each game, we generated more than 10,000 frames for two quality types: good quality with a reasonable framerate, *i.e.*, around 30 fps, and fast quality, where the game pace is maximum. The engine achieves these two qualities depending on many parameters, which are outlined in Table 3.

[12]https://www.youtube.com/watch?v=paLLfWd2k5A
[13]https://www.youtube.com/watch?v=kX0hnOS1QQQ&list=PLX2vGYjWbI0RibPF7vixmr4x8ONJX-mNd
[14]https://www.youtube.com/watch?v=-Lkbo9ZyYbo
[15]https://www.youtube.com/watch?v=XjiBDKrCsVI
[16]https://www.youtube.com/watch?v=UK57qdq_lak

	Quality Settings	Good Quality	Fast Quality
Rendering	Resolution	Full-screen	Full-screen
	Pixel Light Count	4	0
	Texture Quality	Full Res	Eighth Res
	Anisotropic Textures	Per Texture	Disabled
	Anti Aliasing	4x Multi Sampling	Disabled
	Soft Particles	Activated	Disabled
	Realtime Reflection	Activated	Disabled
	Billboards Face Camera Position Probes	Activated	Disabled
Shadows	Shadows	Hard Shadows Only	Disable Shadows
	Shadow Resolution	Medium Resolution	Low Resolution
	Shadow Projection	Stable Fit	Stable Fit
	Shadow Distance	50	15
	Shadow Near Plane Offset	2	2
	Shadow Cascades	Two Cascades	No Cascades
	Cascade Splits	(33,3%, 66.7%)	0
Others	Blend Weights	2 Bones	1 Bone
	V Sync Count	Every V Blank	Don't Sync
	Lod Bias	1	0.3
	Maximum LOD Level	0	0
	Particle Raycast Budget	256	4

Table 3. **Good Quality vs. Fast Quality**

3.4 Measurement

We describe now how we obtain the internal flow and the execution time per frame and per module for the different games over the different platforms and targets, including cloud gaming and computation offloading architectures.

Execution Time. The CPU/GPU execution time per frame, for the aforementioned modules, is obtained for 10,000 frames for each quality encoding. We used the Unity Profiler and script to obtain these values.

Internal Flow. We used two tools with a script to identify the Unity 3D internal flow. We dumped the memory using the script that identifies all the classes/objects and methods that are called per frame. We used Visual Studio 2015 profiler and Dependency Walker to check the validity of the obtained results.

4 PERFORMANCE ANALYSIS AND GAME CLASSIFICATION

Now, we present the results of our measurement campaign regarding the time needed to generate one game frame. We then propose a classification of the games with respect to the best approach to adopt for efficient implementation (i.e., whether the game engine should run on the device, in a cloud gaming system, or with computation offloading). In Section 5, we will study more precisely the case of computation offloading by a thorough analysis at the modular scale.

4.1 Time Needed to Generate One frame

Figure 4 shows the time (in *ms*) spent by the CPU to generate one frame for five games on five platforms. The results in Figure 4a (respectively Figure 4b) correspond to the good (respectively fast) quality. Figure 5 represents also the time spent by the CPU to generate one frame for Stealth, Space Shooter, Car, Unity Lab, and Multiplayer FPS games on the Dell and mobile devices for the good (Figure 5a) and fast (Figure 5b) qualities. These results have been split into two figures for readability because the time needed to generate one frame for the Stealth and the Multiplayer FPS games, on the mobile devices, is one order of magnitude larger than for the other games. Moreover, Car, Unity Lab, and Multiplayer FPS have not been tested on all devices and platforms. The same remark applies for Figure 6, which represents the time spent by the CPU to generate one frame

for the two web players on the Dell laptop. These figures reveal a wide range of performance

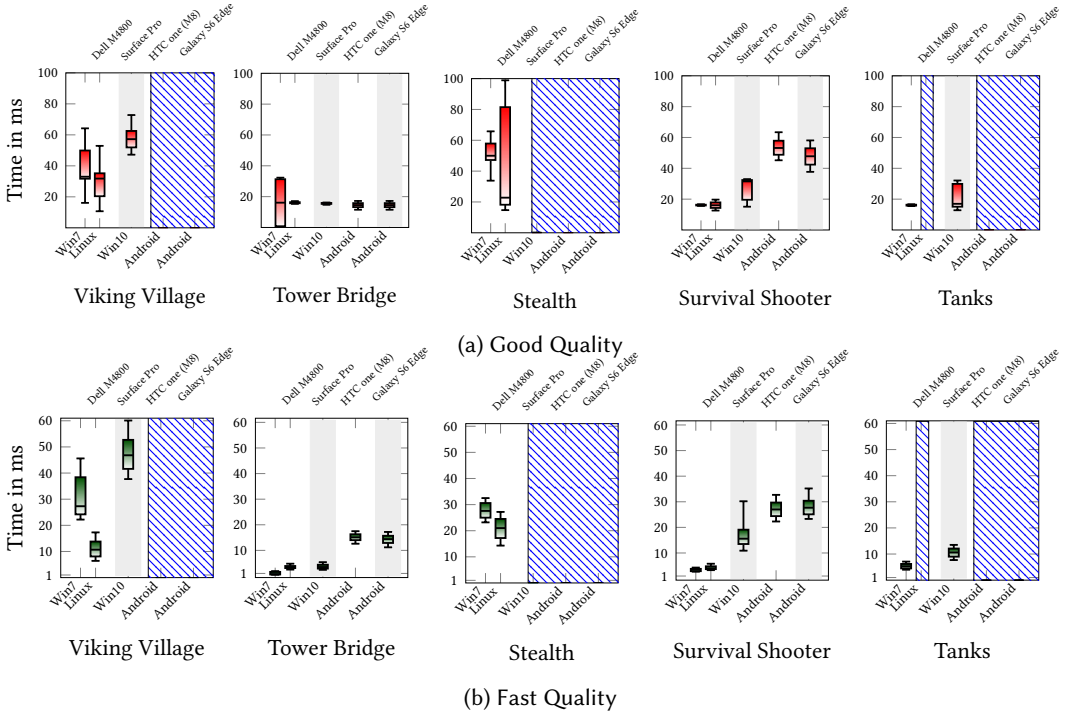


Fig. 4. **Time required by the according processing unit to generate one frame. The box plot includes the 10th, 25th, median, 75th, and 90th percentiles of these times.**

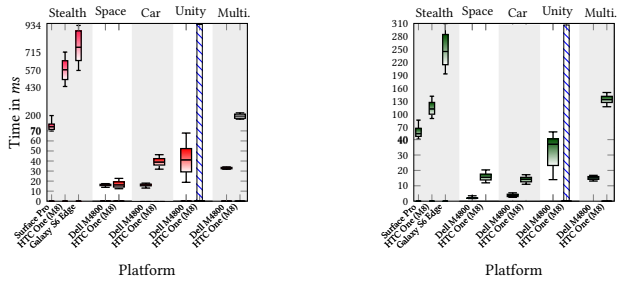
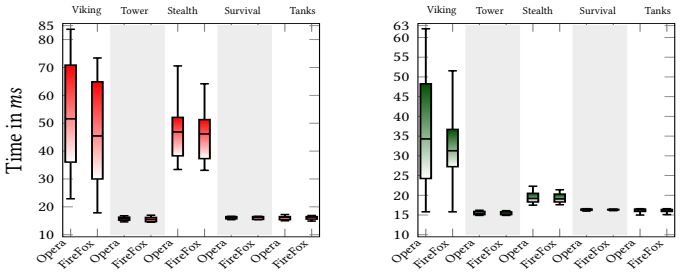


Fig. 5. **CPU-Time required to generate one frame on the standalone and wearable devices.**

regarding the time needed to generate one frame. The obtained results confirm that predicting the frame generation time of a game depends on three main factors: the quality settings, the platform that runs the game engine, and the type of the game.

Impact of the quality settings. It is epitomized by the Stealth game, for which the time needed to generate a good quality frame is excessive on both Dell M4800 and Surface Pro (up to 100 and 200 ms respectively), while it stays in more acceptable ranges for fast quality (up to 32 and 90 ms resp.). We observe the same impact on the other games: *e.g.*, the time consumption per frame is reduced



(a) Good Quality (b) Fast Quality

Fig. 6. **Time required to generate one frame for the Web Players on the Dell M4800.**

by a factor ranging from two to three for the Survival Shooter game. To understand the reasons behind this impact, readers are referred to Section 2.3 and Annex part A.1. A pixel resolution of 640×480 has typically shorter processing time than a 1920×1200 resolution, in particular because the additional pixels in the viewport requires more processing at the rasterization step. Another impacting parameter in high quality settings is the pixel light count, where the process of drawing geometry with lighting is repeated during the light pass step for all the pixel lights. Both texture quality and shadow also increase the processing time.

Impact of the used platform. The hardware configuration and the used OS matter. Regarding the hardware, it is obvious that: (i) using a GPU enables faster frame rendering; (ii) large memory prevents page fault exception and paging process; (iii) multi-core CPU better exploits the parallel executions. Considering the OS, each one has a different architecture with different interruptions, scheduling policies, and algorithms for page replacement in cache, central memory management, and swap. In addition, the device drivers and the used compiler may impact game performance.

For all games, the time needed to generate one frame is smaller on the Dell than on tablet, which in turn is faster than smartphones. Regarding OS, we observe a difference in the frame duration, *e.g.* Viking Village game at fast quality is three to four time faster in Linux than in Windows. For other games (*e.g.* Tower Bridge), the frame duration has different variations depending on the executions in Linux and Windows. We also notice that the frame generation time on the web players is (to our surprise) in the same range as when the frame is directly generated on Windows and Linux. We explain this by the screen resolution, which is automatically reduced to almost a half of the screen window. Some parameters of the quality setting are independent of the screen size, but multiple parameters directly depend on the viewport size.

Impact of the game genre. Some games are more demanding than others with respect to scene complexity, AI, and scripting. The various game genres impact the frame duration. For the same platform and the same quality settings, two different games exhibit different profiles of resource consumption and variability of resources. In particular, we observe that the variability of the frame generation times differs among the games: for some games, all the frames take approximately the same time to be generated, while the generation of a frame can be eight times longer than another frame for some other games. Each game has a behavior signature.

4.2 Game Classification

Game providers are interested in determining the best option to host the game engine. From our previous results and analysis, we identify two *criteria* to characterize games: the variability of resource demand, and the relation between resource demand and quality settings, which we call *playability*. We set *thresholds* for both criteria and we define *predicates*, which can be either verified

or not by each game. Finally, we classify games based on the set of validations. In the following, we first detail the predicates (criteria and threshold), and then we explain the classification process.

Playability. It is a mix of framerate and quality settings. To evaluate a framerate for a game, we take the 90^{th} percentile of the longest frame duration.This percentile is a basis for the setting of the achieved framerate of the game. The higher is the framerate, the better is the QoE. The latter also depends on the quality setting. The higher is the quality setting, the better is the immersion, so the higher is the QoE. We combine both by setting that a given game is *playable* if the framerate is greater than 30 fps (respectively 60 fps) for good quality (resp. fast quality) settings. That is, the 90^{th} percentile should be lower than 33 ms (resp. 16 ms) to validate the playability of a given game. This criteria depends on the platform on which the game engine runs. For example, Tower Bridge at good quality on the Dell M4800 is playable on Linux but not on Windows. When a game cannot verify the playability criteria on a device, the game engine cannot run on the device in the traditional client-server mode. Other options must be considered, either cloud gaming platform or computation offloading.

Resource Variability. It describes the variability in frame generation time. We consider two values to differentiate high and low variabilities: the *range* and the *Interquartile Range (IQR)*. The range is the difference between the highest and the lowest score, while IQR corresponds to the range of half of the scores around the median (the difference between the 75^{th} and the 25^{th}). We say that the variability is low when the range (resp. IQR) is less than 20 ms (resp. 10 ms). For example, Survival Shooter, Tanks, and Tower Bridge games exhibit a low resource variability.

The resource variability matters because it impacts the efficiency of the implementation on a virtualized hardware. A high resource variability means that the game provider must reserve a vast amount of resources to absorb the peak, at the expense that the game engine only uses a fraction of these resources during most of the execution time. High resource variability, thus, means an inefficient resource reservation and a lower benefits for the game providers. On the contrary, a game characterized by a low resource variability is easy to pack into a well-sized Virtual Machine (VM) (or container), which enables a better consolidation and low infrastructure cost.

Based on both predicates, we classify the games as follows:

- *Client-server*, where all functions of the game engine (except session and multi-player management) run on the device. A game falls into this category if and only if it verifies the playability criteria.
- *Cloud gaming*, which means that the game as a whole runs in a VM in a data-center, while the client only runs a video decoder. A game falls into this category if both it is not playable on the device and it exhibits a low resource variability.
- *Computation offloading*, which means that the game engine is distributed among a client and a server, both of them being linked with a high-bandwidth low-latency network connection. A game falls into this category if the playability criteria is not verified and the game exhibits a high resource variability.

Table 4 summarizes the classification of the games based on the aforementioned devices (more details in the appendix). We observe that, for these representative games and devices, cloud gaming is rarely the best option (only for two games on the mobile devices, and one game on Dell, and browsers). On the contrary, solutions based on computation offloading are the best option for four games in all devices except the fourth game on the Dell and browsers. This result calls for a deeper exploration of computation offloading solution, which has not been studied for gaming so far.

Games	Dell M4800	Surface Pro	HTC M8	Galaxy S6 Edge	Browsers
Viking Village	Offload	Offload	Offload	Offload	Offload
Tower Bridge	Client-server	Client-server	Client-server	Client-server	Client-server
Stealth	Offload	Offload	Offload	Offload	Offload
Survival Shooter	Client-server	Client-server	Cloud	Cloud	Client-server
Tanks	Client-server	Client-server	Client-server	Client-server	Client-server
Space Shooter	Client-server	Client-server	Client-server	Client-server	Client-server
Car	Client-server	Client-server	Cloud	Cloud	Client-server
Unity Lab	Offload	Offload	Offload	Offload	Offload
Multiplayer	Cloud	Offload	Offload	Offload	Cloud

Table 4. **Best option for architecture implementation per game and device**

5 IS COMPUTATION OFFLOADING POSSIBLE IN GAME ENGINES?

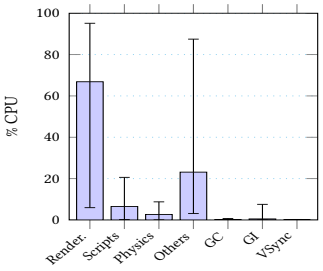
We address now a second part of our measurement analysis with a focus on modular aspects of game engines. Our main motivation is the study of solutions based on computation offloading. In this approach, the resource requirements of each module and the interactions between the modules are two needed key information for an efficient implementation. A secondary motivation is the study of solutions based on cloud gaming. In existing platforms [18, 29], the game engine is seen as a “black box". One of the improvements we would like to study in the future is to distribute a game engine into several physical machines in a data-center to enable parallel computation. Similarly, one needs to understand the resource requirements at the module scale as well as the interactions between the modules.

5.1 CPU Consumption per Module

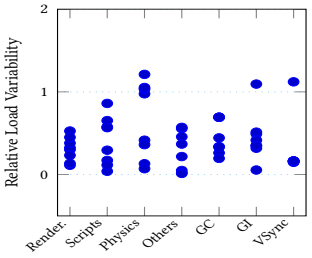
To understand the time consumption of each module per frame, we look at the CPU usage percentage at the modular level. We gather modules into seven families: Rendering, Scripts, Physics, Garbage Collector (GC), Global Illumination (GI), Vsync, and Others. We abusively use the term "modules" hereafter to refer to a family of modules.

Figure 7 shows the consumption (in % of CPU) of modules. We considered for each module, the percentage of time it takes to compute in relation to the overall consumption needed to generate one frame. We aggregated all the data corresponding to the nine games, the two qualities on the different devices (except the Dell tower server) into Figure 7a. We plotted the three keys values: average, minimum, and maximum consumption. For more details, readers may refer to Figures 15 and 16 in the appendix.

For every game, the frame duration is limited by one of three threads: the CPU game thread, the CPU render thread, or the GPU thread. In Figure 7a, we identify the rendering engine as the most consuming module. It is responsible of up to 70% of CPU usage in average. Since the rendering is done by the GPU, it means that the game performance is limited by the GPU thread. Indeed, once the CPU launches the rendering process in the GPU, it can run in parallel some other modules, including AI, physics and scripting. As soon as the process of these modules terminates, the CPU waits for the GPU process termination. For some highly-graphical games such as Viking Village, the rendering modules is responsible of 95% of the CPU consumption on the Dell Win7 platform at high-quality.



(a) Module Consumption (min, avg, max)



(b) Relative load variability: $\frac{r_{pq}-R}{\mu}$

Fig. 7. **Module resource requirement and relative load variability.**

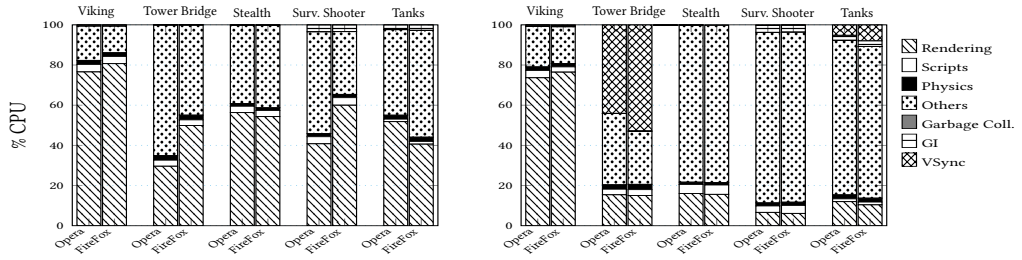
We are also interested in understanding the differences in the module consumption *per platform and game quality*. For a given platform and a given user quality setting, we compute the average time consumption for all frames and all games to obtain a global average platform-dependent quality-dependent module requirement. We thus get eight measures of the average module requirement (four platforms and two game qualities), noted r_{pq} for each platform p and each quality q. We compute the average requirement, noted R for the eight measures. Then, we compute, for one given platform-quality pq, the difference to the average measure, $r_{pq} - R$. A platform-quality pq with a high difference means that, for this given module, the requirement significantly differs from the other platform-quality, and thus it would justify the game provider to pay a specific attention. Our idea is also to check if some modules exhibit much wider different behaviors than others. To enable comparison across modules, we thus compute a *relative* measure of the difference by computing the ratio of this difference $r_{pq} - R$ to the standard deviation μ. We call this measure the *relative load variability*. The higher is the relative load variability of a platform-quality pq, the more different is the platform-quality pq from the other platforms.

We represent, for each module, the eight relative load variability measures in Figure 7b. We have two observations. First, the resource requirement is sensitive to platform-quality. Some platform-quality pq have a difference to the average that is nearly 1.5 larger than the standard deviation. It calls for paying attention to platform-quality when implementing a computation offloading because these platform-quality exhibit specific and unusual resource consumption. Second, all modules have similar relative load variability. This is counter-intuitive since we expected that rendering modules would be more sensitive to platform-quality than other modules.

In Figure 8, we represent the consumption per module for the web players. The CPU spends a lot of time computing other type of modules, like scheduling the tasks over the different layers. The rendering portion of time is lower than for the case of standalone platforms, which is due to the default resolution used in the web players. The script and physics take more portion of CPU than in the standalone cases.

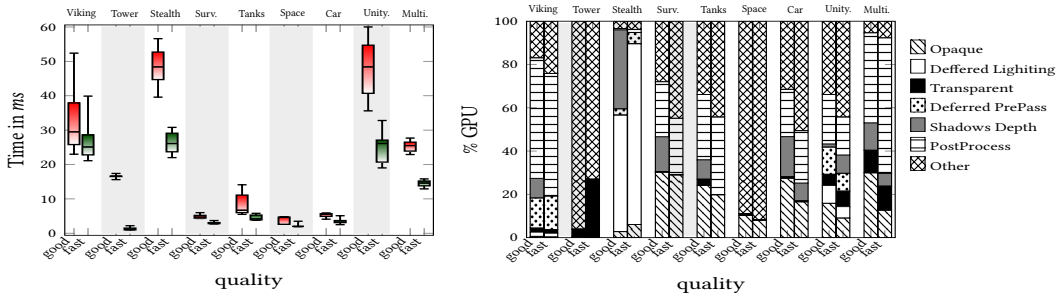
5.2 GPU Consumption per Module

We concluded from Figure 7 that the Rendering modules are the most consuming modules regarding CPU use. One of our previous explanations is that CPU is bounded by the performance of GPU to render the scene. In the following, we look at the GPU consumption to validate this statement. We plot in Figure 9a the time spent by the GPU to generate one frame for each of the games on the Dell laptop. In Figure 9b, we show the percentage of time the GPU spends in each subfamily of the rendering module. We derive from Figure 9a that the time taken by the GPU is approximately the same as the CPU in Figures 7a. It confirms our intuition that, since the CPU finishes processing tasks faster than the GPU, the CPU waits for the GPU to finish the frame generation. As we will see



(a) Good Quality (b) Fast Quality

Fig. 8. **CPU consumption per modules on Dell M4800 for Web Players**



(a) Consumption per frame (b) Consumption per module

Fig. 9. **GPU consumption per frame and per module on Dell M4800 for Windows 7**

later, the “waiting mode" is considered as a module to schedule and synchronize the two processors. Moreover, in Figure 9b, the consumption per rendering activity is different for each game. Indeed, the nine game worlds are different. We also observe that the GPU is not entirely used for rendering processing. Typically a game like Tower Bridge consumes the entirety of available GPU for other types of module.

5.3 Calls between Modules

We pay now attention to the modular architecture and the interactions between these modules. This study is key for the implementation of computation offloading, and it is also useful for new implementation of cloud gaming systems.

We depict in Figures 10 and 11 the main calls inside each family of modules. Specifically, Figure 10 (resp. 11a, 11b, 11c, 11d, and 12) corresponds to the Physics module family (resp. Audio, AI, Animator, Script, and Rendering module families). In each graph, the vertices represent each module in the family, while the oriented edges are a combination of two parameters: the first one is the number of time the source vertex calls the destination vertex (the calls frequency), the second value is the execution time of the destination vertex. These flowcharts correspond to the case of Survival Shooter game on the HTC One (M8) for the good quality encoding.

Each module family can be offloaded as a separate service. Indeed, we observe in the figures that these families exhibit few interactions and data sharing *each other*, which is expected since these families of modules are often designed and implemented by different teams. The main coordination and synchronization task is done through the modules *Update* and *Fixed Update*, which are shared between all the families and represent the main game thread. Since the frequency calls between

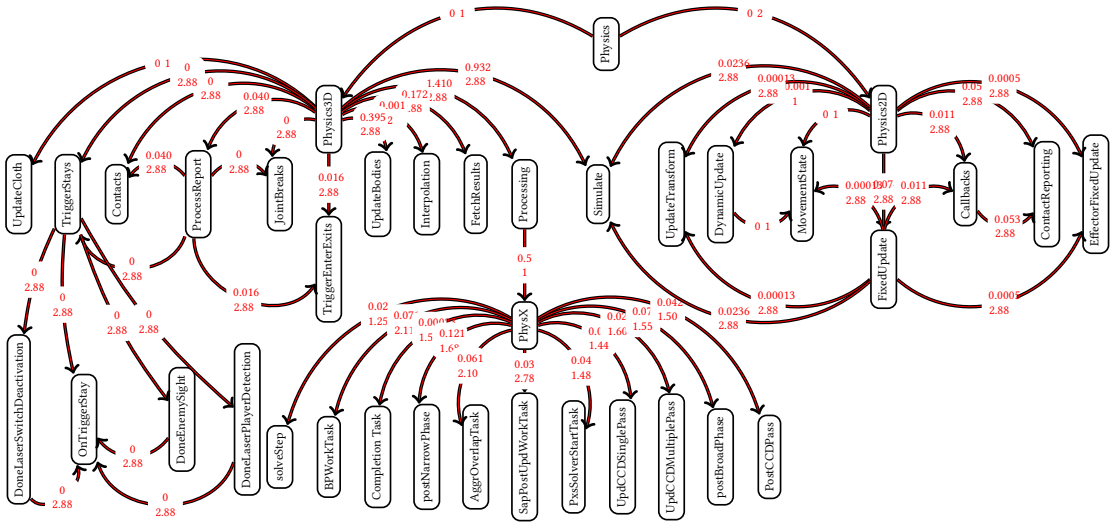


Fig. 10. **Internal calls inside the Physics module**

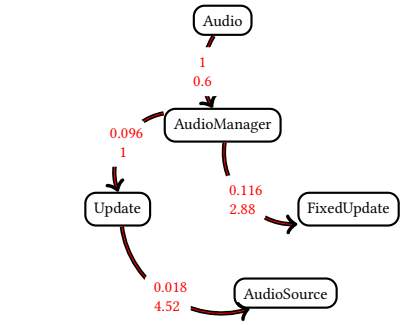
families are low, the offloading per module families makes sense. Inside each module family, here are our main observations:

- The Physics family (see Figure 10) can be spread into three sub-families: *Physics3D*, *Physics2D*, and *PhysX*. PhysX interacts only with Physics3D over the *Processing* class, and Physics3D shares with Physics2D only one class (the *Simulate* class). It would thus be relatively simple to distribute the computation of these three sub-families over distinct computers.
- The Scripting family (see Figure 11d) is the only module that interacts directly with the Physics engine. It would thus make sense to run this family of modules on the same computer as the one that hosts the Physics modules.
- The Audio family (see Figure 11a) has no interaction with the other main modules and only deals with the game thread. This module can thus be offloaded as an API to a remote machine. The communication between the client and the remote machine can be done either by RPC or by streaming the audio data.
- The AI family can be computationally intensive. It is difficult to simulate the game on constrained-resource devices without reducing the complexity of the AI. Since this family is also mostly independent, the module family can be offloaded.

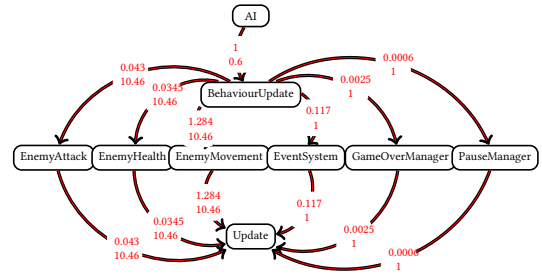
Finally, we address in Figure 12 the Rendering modules. We revealed in our previous results that this family is the most resource consuming, and thus is a candidate to migrate from the lightweight client devices to nearby servers in a computation offloading solutions (or from one standard server running the game engine to a specialized GPU-enabled server in the same cluster of servers for the case of a cloud gaming solution).

The offloading mechanism can be instruction-based or image-based. In instruction-based systems, the client renders the graphics by itself using the commands received from the server. This system consumes less bandwidth as only graphics drawing commands transit over the network. Image-based system streams the rendered game as a real-time video. The clients are platform- and implementation-independent, and demand fewer resources. However, it is harder to distribute this module. Indeed, we identify some “sub-families" within the Rendering family that are rarely

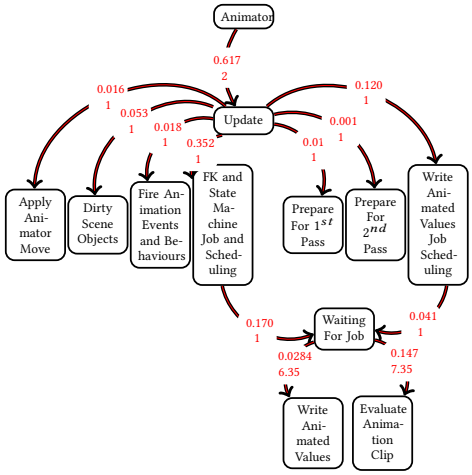
independent. We observe in Figure 12 that the calls come from multiple other modules, which are not necessarily in their own sub-family. These inter-calls between modules, from different sub-families, make the code offloading harder. Indeed, the calls frequency is high, which implies intensive communication between the different computers and ultra-low-delay data mirroring in the case of offloading. Moreover, some sub-families are more called than the others. For example, *SharedSet-Pass*, *RenderTexture*, and *MeshVBOModule* are called to generate the frames. Finally, the *WaitingForJobs* module is used to synchronize the modules that have different running time. The time spent into the WaitingForJob module is a waste of time and CPU consumption.



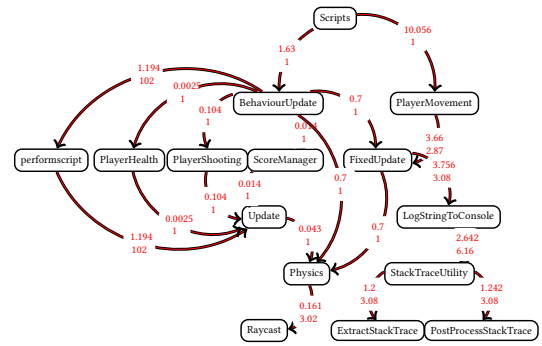
(a) **Internal calls inside the Audio module**



(b) **Internal calls inside the AI module**



(c) **Internal calls inside the Animator module**



(d) **Internal calls inside the Scripting module**

Fig. 11. **Internal flow concerning the main modules: Audio, AI, Animations and Scripting.**

6 GAME PERFORMANCE WHEN ASSISTED BY SERVER.

In Section 4, we tested the different games in the traditional client-server architecture, from which we derived a game classification for the different platforms. In Section 5, we studied the possibility to offload a game engine and concluded with promising solution regarding offloading the game engine modules. Now, we present the performance of the games when we offload a part and the whole game engine to a server as in the two architectures (cloud gaming and computation offloading). We present the needed CPU-time to generate one game frame on the client device in both architectures.

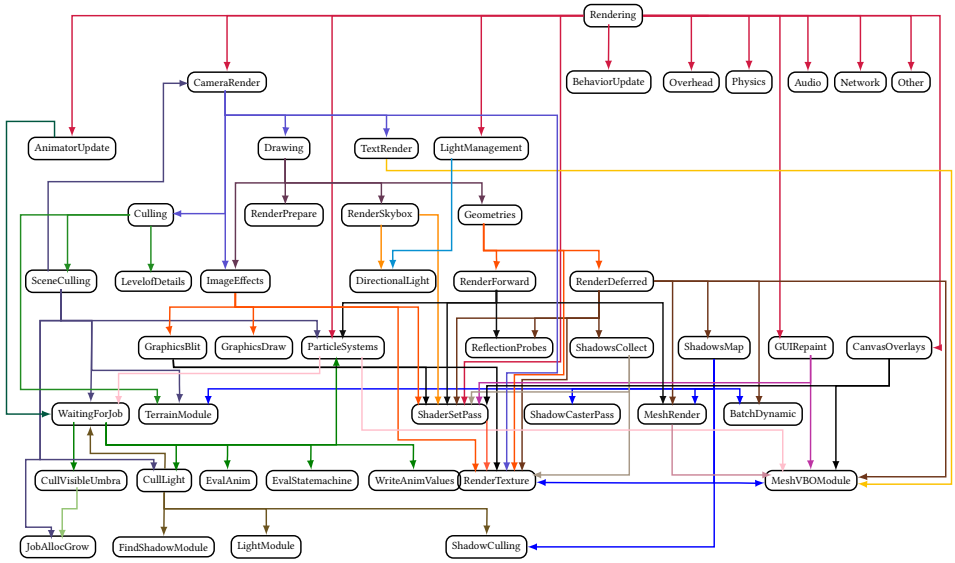


Fig. 12. **Internal calls inside the Rendering module**

For a fair comparison between the games, we offload the same game objects namely NPC, Player Character (PC), environment, and lights. Table 5 summarizes the game module distribution. Table 8 presents the used devices for each architecture.

Element	Module	Client	Server
Player character	Rendering	☒	☑
	Audio	☒	☑
	Animations	☒	☑
	Scripting	☒	☑
	Physics	☒	☑
	Inputs	☑	☒
Non-Player character	Rendering	☒	☑
	Audio	☒	☑
	Animations	☒	☑
	AI	☒	☑
	Physics	☒	☑
Particle Effects	Rendering	☑	☒
	Animation	☑	☒
	Physics	☑	☒
Arena	Rendering	☒	☑
	Physics	☒	☑
Sound	Audio	☑	☒
	Scripting	☑	☒
Sun Light	Rendering	☑	☒
Arena Light	Rendering	☒	☑
Main Camera	Rendering	☑	☒
Weapon Camera	Rendering	☒	☑
Game Manager	Scripting	☑	☒

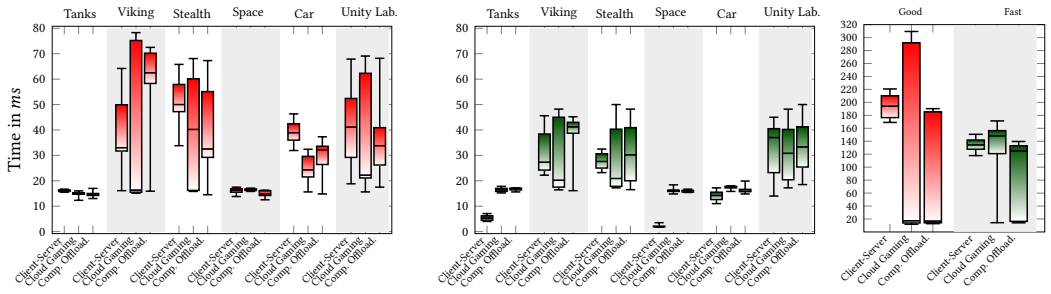
Table 5. **Location of the game objects in case of computation offloading**

Figure 13 depicts the time needed to generate one frame on the three architectures (i.e., client-server, cloud gaming, and computation offloading) for each of the used games for the good (Figure 13a) and fast qualities (Figure 13b). We use box-plots to present the results since we focus on the variability of CPU consumption per frame. Indeed, the *consolidation* of resources in a data-center is easier when the consumption of processing resources is accurately predicted. The more stable are the CPU and GPU consumption, the more games can concurrently run in a cluster.

We distinguish three categories of game. Some games are ideal for client-server architecture and/or consolidation because all frames take approximately the same CPU and GPU time to be generated. The 10^{th} and 90^{th} percentiles are so close that they nearly overlap. It is notably the case of Tanks and Space Shooter. A game offering a small variability features scenes that are not complex and are thus easily rendered by the GPU. For the good quality, a frame is generated in less then 33 *ms* and in less then 16 *ms* for the fast quality. These games are device-friendly.

Some other games exhibit a high variability, notably Viking Village, Stealth, and Unity Lab. These are the worst cases for cloud provider, which has to reserve resources to accommodate the peaks (more than 75 *ms* CPU for Viking Village, 65 *ms* for Stealth, and 67 *ms* for Unity Lab), although the median frame requires almost quarter the time (16 *ms* here). The reserved resources are wasted. These games in general are the most consuming for the CPU and GPU resources due to the scene complexity. Since these games are not desirable for cloud gaming architectures, and cannot be played locally on the wearable devices, then these games should be offloaded.

Finally, the games that are the less demanding and have less variability, typically Car in our set of games, are good candidates for cloud gaming systems.



(a) Good Quality (b) Fast Quality (c) Multiplayer Game

Fig. 13. CPU-Time required to generate one game frame on the three architectures

7 DISCUSSION AND MAIN FINDINGS

To sum up our findings in this paper: Firstly, the GPU is the main source of performance limitation in the different cases. The CPU game and CPU render threads are frequently blocked waiting for the results of the GPU due to a *synchronization* problem. To improve GPU performance, an idea would be to free the latter from some jobs. For instance, recent GPU cards improves the game performance by offloading the PhysX from GPU to CPU, especially for powerful CPU. Unfortunately, this solution works only for games using PhysX as physics engines (like Batman [6], Assassin's Creed [34], and some others[17]) but it does not work for other engines like Havok, ODE, or I-Collide. We also identified that both CPU and GPU consumptions have a high correlation, which is useful for the

[17] http://www.geforce.com/hardware/technology/physx/games

cloud gaming providers to estimate from the CPU demand the amount of GPU resources that need to be reserved, as well as the game variability.

Second, the frame rate of device-friendly games is generally higher than 60 fps. This is a waste of resources, especially in terms of energy. Here, we envision a solution where we first calculate the number of generated and saved frames in the frame buffer, by monitoring the synchronization, then the GPU breaks the game main loop when the number of 60 fps is reached. The idle GPU saves a signfiicant part of the energy consumption. The provider can use the idle GPU to serve more games using a preemptive scheduling. NVIDIA proposed the mechanism *G-Synch*, which improves the *V-Sync* mechanism, where the monitor refresh is conducted by the GPU frequency. It finds trade-off between the off- and on-mode of V-Sync. Indeed, when V-Sync is deactivated, the GPU sends the frames following its own pace. At each screen refresh cycle, multiple frames are displayed because the frame rate is maximal causing the phenomenon of *tearing*. Now, when the V-Sync is activated, the GPU follows the screen tempon so tearing no longer occurs. But, when the time to render a frame is longer than a refresh cycle, the phenomenon of *stutter* and *display delay (lag)* occurs because the monitor has to display again the last frame.

Third, some modules related to rendering are mostly in waiting mode, meaning that the CPU consumption associated with these modules is not significant. These waiting times are not necessarily a waste of resources when only one game runs on a computer, since the rendering pipeline is at a given step and no further actions can be taken. However, in the context of cloud computing, these waiting times represent opportunities to free some resources and to better exploit the processing units.

Fourth, a non-graphic component of the game engine can represent a significant part of the CPU consumption, typically in games where the AI, the physics, and the scripts are complex. A motivation behind code offloading is to study the gains when the display device embeds dedicated GPU resources, and powerful servers are available nearby. This configuration matches the new generation of game centers with VR headsets.

Finally, the game classification of the games into the three architectures matches our predictions based on the criteria: *playability*, and *resource variability*.

8 CONCLUSION

This paper deals with the topic of game engines. We presented a general architecture and conducted a set of experiments on three architectures: client-server, cloud gaming, and computation offloading. We used nine representative games, including FPS, TPS, Racing, and MMOG. We adapted these games to different platforms and tested their performances on each platform for two quality encodings. Based on our results, we classified the games into the three architectures depending on two parameters: playability and resource variability. We also identified that the quality settings, the used platforms, and the game genre exhibit varying behaviors. Next, we provided a detailed view of the game engine by representing the internal flow that constitutes each module, which is a basis for solutions exploring computation offloading. Finally, we validated our finding regarding the game classification according to the aforementioned criteria and the feasibility of game engine offloading. In our experiments, we used static offloading of the sub-families of modules. Our work opens new exciting perspectives and research directions to improve gaming experience. Indeed, we may mention particularly two directions to follows, one based on computation offloading and another one on cloud gaming consolidation. Regarding computation offloading, it is interesting to explore not only partitioning algorithms to find the best execution location of each module sub-family, but also use systems for polygon rendering to decompose an object, or an entire image to offload some part aiming at improving the rendering process between the client and the server. About

game consolidation, it will be useful to define a regression model for the resource consumption to maximize the consolidation and minimize the risks, and find somehow to serve different gamers in Single Online Shared Game Instance (SOSGI) (i.e., throw only one instance of game engine).

REFERENCES

[1] accessed in Aug. 2015. Unity: The Leading Global Game Industry Software. (accessed in Aug. 2015).

[2] Saeid Abolfazli, Zohreh Sanaei, and Ejaz Ahmed. 2014. Cloud-Based Augmentation for Mobile Devices: Motivation, Taxonomies, and Open Challenges. *IEEE Communications Surveys and Tutorials* 16, 1 (2014), 337–368.

[3] Eike Falk Anderson, Steffen Engel, Peter Comninos, and Leigh McLoughlin. 2008. The Case for Research in Game Engine Architecture. In *Proc of the 2008 Conf. on Future Play*.

[4] Grenville J. Armitage and Amiel Heyde. 2012. REED: Optimizing first person shooter game server discovery using network coordinates. *TOMCCAP* 8, 2 (2012), 20.

[5] Paul B Beskow, Andreas Petlund, and Geir A Erikstad. 2010. Reducing Game Latency by Migration, Core-selection and TCP Modifications. *International Journal of Advanced Media and Communication* 4, 4 (2010), 343–363.

[6] W. Brooker. 2012. *Hunting the Dark Knight: Twenty-First Century Batman*.

[7] Eliya Buyukkaya, Maha Abdallah, and Gwendal Simon. 2015. A survey of peer-to-peer overlay approaches for networked virtual environments. *Peer-to-Peer Networking and Applications* 8, 2 (2015), 276–300.

[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 *Proceedings of the 11th ACM/IEEE Netgames workshop*.

[9] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. 2014. A Hybrid Edge-Cloud Architecture for Reducing On-demand Gaming Latency. *Multimedia Systems* 20, 5 (2014), 503–519.

[10] Seong-Ping Chuah, Chau Yuen, and Ngai-Man Cheung. 2014. Cloud gaming: a green solution to massive multiplayer online games. *IEEE Wireless Commun.* 21, 4 (2014), 78–87.

[11] Mark Claypool and Kajal Claypool. 2009. Perspectives, frame rates and resolutions: it's all in the game. In *Proceedings of the 4th ACM International Conference on Foundations of Digital Games*.

[12] Mark Claypool and Kajal T. Claypool. 2006. Latency and player actions in online games. *Commun. ACM* 49, 11 (2006), 40–45.

[13] Mark Claypool and Kajal T. Claypool. 2010. Latency can kill: precision and deadline in online games. In *Proceedings of the First Annual ACM MMSys Conference*.

[14] Brent Cowan and Bill Kapralos. 2014. A Survey of Frameworks and Game Engines for Serious Game Development. In *Proceedings of the 14th IEEE International Conference on Advanced Learning Technologies (ICALT)*.

[15] Michael Deering, Stephanie Winner, and Bic Schediwy. 1988. The triangle processor and normal vector shader: a VLSI system for high performance graphics. In *Proc. of 15th Conf. on Comp. Graphics and Interactive Techniques, SIGGRAPH*. 21–30.

[16] Wu-chang Feng and Wu-chi Feng. 2003. On the geographic distribution of on-line game servers and players. In *Proceedings of the 2nd ACM Netgames workshop*.

[17] Hua-Jun Hong, De-Yu Chen, Chun-Ying Huang, and Kuan-Ta Chen. 2015. Placing Virtual Machines to Optimize Cloud Gaming Experience. *IEEE Transactions on Cloud Computing* 3, 1 (2015), 42–53.

[18] Chun-Ying Huang, Kuan-Ta Chen, De-Yu Chen, and Hwai-Jung Hsu. 2014. GamingAnywhere: The first open source cloud gaming system. *ACM Trans. on Multimedia Comp., Comm., and App. (TOMM)* 10, 1s (2014), 10.

[19] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, and Sean Ahern. 2002. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph.* 21, 3 (2002), 693–702.

[20] Michael Jarschel and Daniel Schlosser. 2011. 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)*.

[21] Teemu Kämäräinen, Matti Siekkinen, Yu Xiao, and Antti Ylä-Jääski. 2014. Towards pervasive and mobile gaming with distributed cloud infrastructure. In *Proceedings of the 13th ACM Netgames Workshop*.

[22] Atta Rehman Khan, Mazliza Othman, and Sajjad Ahmad Madani. 2014. A Survey of Mobile Cloud Computing Application Models. *IEEE Communications Surveys and Tutorials* 16, 1 (2014), 393–413.

[23] Blazej J. Kot, Burkhard Wuensche, John C. Grundy, and John G. Hosking. 2005. Information visualisation utilising 3D computer game engines case study: a source code comprehension tool. In *Proceedings of the 6th ACM Conf. on Computer-Human Interaction (CHI)*.

[24] Kyungmin Lee, David Chu, and Eduardo Cuervo. 2014. Demo: DeLorean: using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 12th ACM MobiSys Conference*.

[25] Yeng-Ting Lee, Kuan-Ta Chen, Han-I Su, and Chin-Laung Lei. 2012. Are all games equally cloud-gaming-friendly? An electromyographic approach. In *Proceedings of the 11th ACM Netgames Workshop*.

[26] Michael Lewis and Jeffrey Jacobson. 2002. Game engines. *Commun. ACM* 45, 1 (2002), 27.

[27] Yusen Li, Xueyan Tang, and Wentong Cai. 2015. Play Request Dispatching for Efficient Virtual Machine Usage in Cloud Gaming. *IEEE Trans. Circuits Syst. Video Techn.* 25, 12 (2015), 2052–2063.
[28] Yao Liu, Sujit Dey, and Yao Lu. 2015. Enhancing Video Encoding for Cloud Gaming Using Rendering Information. *IEEE Trans. Circuits Syst. Video Techn.* 25, 12 (2015), 1960–1974.
[29] Meng Luo and Mark Claypool. 2015. Uniquitous: Implementation and Evaluation of a Cloud-based Game System in Unity. In *Proceedings of IEEE GEM Conf.*
[30] Stefan Marks, John A. Windsor, and Burkhard Wünsche. 2007. Evaluation of game engines for simulated surgical training. In *Proc of the 5th ACM Int. Conf. on Computer Graphics and Interactive Techniques (Graphite).*
[31] Farouk Messaoudi, Gwendal Simon, and Adlen Ksentini. 2015. Dissecting games engines: The case of Unity3D. In *Proc. of ACM Workshop on Network and Systems Support for Games, NetGames.* 1–6.
[32] Vlad Nae, Alexandru Iosup, and Radu Prodan. 2011. Dynamic Resource Provisioning in Massively Multiplayer Online Games. *IEEE Trans. Parallel Distrib. Syst.* 22, 3 (2011), 380–395.
[33] R. M. Nasiri, J. Wang, A. Rehman, and S. Wang. 2015. Perceptual quality assessment of high frame rate video. In *Proceedings of the 17th IEEE International Workshop on Multimedia Signal Processing (MMSP).*
[34] Magy El Nasr, Maha Al-Saati, and Simon Niedenthal. 2008. Assassin's Creed: A Multi-Cultural Read. (2008).
[35] Beatrice Ng, Antonio Si, Rynson W. H. Lau, and Frederick W. B. Li. 2002. A multi-server architecture for distributed virtual walkthrough. In *Proc. of the ACM Symp. on Virtual Reality Software and Tech. VRST.* 163–170.
[36] Peter Quax, Anastasiia Beznosyk, Wouter Vanmontfort, and Robin Marx. 2013. An evaluation of the impact of game genre on user experience in cloud gaming. In *Proc. of the IEEE Int. Games Innov. Conf. (IGIC).* 216–221.
[37] Kjetil Raaen, Ragnhild Eg, and Carsten Griwodz. 2014. Can Gamers Detect Cloud Delay?. In *Proceedings of the 13th ACM/IEEE Netgames Workshop.*
[38] Kjetil Raaen and Andreas Petlund. 2015. How Much Delay is There Really in Current Games?. In *Proceedings of the 6th ACM Multimedia Systems (MMSys) Conference.*
[39] Andreas Sackl, Raimund Schatz, and Tobias Hossfeld. 2016. QoE Management made uneasy: The case of Cloud Gaming. In *Proceedings of the IEEE International Conference on Communications Workshops (ICC).*
[40] Takafumi Saito and Tokiichiro Takahashi. 1990. Comprehensible rendering of 3-D shapes. In *Proceedings of the 17th ACM Conference on Computer Graphics and Interactive Techniques (SIGGRAPH).* 197–206.
[41] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. 2014. Cloudlets: at the leading edge of mobile-cloud convergence. In *Proceedings of MobiCASE.*
[42] Simon Schneegans, Felix Lauer, Andreas-C. Bernstein, and Andre Schollmeyer. 2014. guacamole - An extensible scene graph and rendering framework based on deferred shading. In *Proceedings of the 7th IEEE Workshop on Software Engineering and Architectures for Realtime Interactive Systems, SEARIS.*
[43] Mehdi Semsarzadeh, Abdulsalam Yassine, and Shervin Shirmohammadi. 2015. Video Encoding Acceleration in Cloud Gaming. *IEEE Trans. Circuits Syst. Video Techn.* 25, 12 (2015), 1975–1987.
[44] Ryan Shea, Di Fu, and Jiangchuan Liu. 2015. Cloud Gaming: Understanding the Support From Advanced Virtualization and Hardware. *IEEE Trans. on Circuits and Systems for Video Tech.* 25, 12 (2015), 2026–2037.
[45] Ryan Shea and Jiangchuan Liu. 2013. On GPU pass-through performance for cloud gaming: Experiments and analysis. In *Proceedings of 12th IEEE/ACM Netgames Workshop.*
[46] Jouni Smed, Timo Kaukoranta, and Harri Hakonen. 2002. Aspects of networking in multiplayer computer games. *The Electronic Library* 20, 2 (2002), 87–97.
[47] Mirko Suznjevic, Jose Saldana, and Maja Matijasevic. 2014. Analyzing the Effect of TCP and Server Population on Massively Multiplayer Games. *International Journal of Computer Games Technology* 2014 (2014), 2.
[48] James Tulip, James Bekkema, and Keith Nesbitt. 2006. Multi-threaded Game Engine Design. In *Proceedings of the Interactive Entertainment conf (IE).*
[49] Maxime Véron, Olivier Marin, and Sébastien Monnet. 2014. Matchmaking in multi-player on-line games: studying user traces to improve the user experience. In *Proceedings of the 13th ACM/IEEE Netgames Workshop.*
[50] Steven Daniel Webb, Sieteng Soh, and William Lau. 2007. Enhanced Mirrored Servers for Network Games. In *Proceedings of the 6th ACM Netgames Workshop.*
[51] Lars C. Wolf (Ed.). 2002. *Proceedings of the 1st Workshop on Network and System Support for Games, NETGAMES 2002, Braunschweig, Germany, April 16-17, 2002, 2003.* ACM.
[52] A. Yahyavi, K. Huguenin, J. Gascon-Samson, J. Kienzle, and B. Kemme. 2013. Watchmen: Scalable Cheat-Resistant Support for Distributed Multi-player Online Games. In *Proc. of the 33rd IEEE Int. Conf. on Dist. Comp. Sys. (ICDCS).*
[53] Amir Yahyavi and Bettina Kemme. 2013. Peer-to-peer architectures for massively multiplayer online games: A Survey. *ACM Comput. Surv.* 46, 1 (2013), 9.
[54] Y. Zhang, P. Qu, J. Cihang, and W. Zheng. 2016. A Cloud Gaming System Based on User-Level Virtualization and Its Resource Scheduling. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2016), 1239–1252.

A APPENDIX

A.1 Rendering Pipelines

Unity 3D uses four additional rendering activities, which occur in conjunction with the main pipeline. We summarize these rendering activities hereafter.

(1) *Forward rendering* – It has been introduced for dynamic lighting in 3D environments. It consists of three passes: **(i)** *Ambient Pass* applies a global low lighting to the overall 3D scene. The results of this pass are saved in the frame buffer; **(ii)** *Light Pass* repeats the drawing process for each opaque object, affected by one of the light sources regardless of others in the scene. The lighting is accumulated in the frame buffer; **(iii)** *Transparency Pass* draws all the transparent objects in the same way as for opaque objects, but with a drawing order of transparent objects from back to front. The transparent geometry is added to the frame buffer and combined with the ambient and opaque results [15].

(2) *Deferred shading* – The idea is to defer the lighting calculation to the second pass until all the geometry has been rendered. The deferred shader algorithm uses smart management of different buffers [40], defined through three passes as follows: **(i)** *Geometry Pass* (Opaque Pass) applies an ambient lighting, saves the results in the frame buffer, and fills the Geometry buffer; **(ii)** *Light Pass* processes the lighting through the buffers transition calculations, starting with a *depth buffer* to rebuild origin position of pixels. The results are added to a *normal buffer* data to calculate the diffuse light. Then, a specular lighting is calculated using the *position*, the *normal*, and *specular data buffers*. Finally, the specular light is applied to *colour* and *shading data buffer* and accumulated with all the other light sources. The global colour of these buffers is accumulated in the frame buffer; **(iii)** *Transparency Pass* renders the transparent geometry using the forward rendering (Ambient Pass and Light Pass) [15, 42].

(3) *Pre-pass rendering* – Similarly as deferred shading, it addresses the restricted usage of different material shaders. The lighting is stored in the new buffer with light pre-pass rendering, instead of applying it to the colour and shading data buffer. This process follows four passes: **(i)** *Geometry Pass* applies the ambient lighting, draws the opaque geometry, and fills the G-buffer; **(ii)** *Light Pass* performs the lighting calculations in the pixel shader using the G-buffer for all light sources. The results are accumulated with additive primitive in lighting buffer; **(iii)** *Material Pass* draws again the geometry using the lighting buffer as lighting input for the material-specific shader; **(iv)** *Transparency Pass* uses the forward rendering (Ambient Pass & Light Pass) to process the lighting. The results are saved in the frame buffer.

(4) *Vertex lit* – This operation is the fastest one and is supported by a large number of hardware. This process is done in one pass, wherein each object is rendered with lighting calculated on the vertices of the object from all the light sources.

A.2 Game Description

Viking Village – is a FPS 3D-game. The sequence offers a look at a medieval Viking village. The scene is characterized by high design quality with many details and various effects like water vibration, fire and smoke particles, sunlight, firelight, shadows, and reflecting lights, shadows, and colours by surfaces.

Tower Bridge Defense – is a TPS 2D platformer game. It depicts a player character fighting against NPC in a physics-driven 2D sample level. The avatars jump between suspended platforms, over obstacles, to advance the game. It features many objects subject to the 2D physics such as gravity, velocity, and other forces.

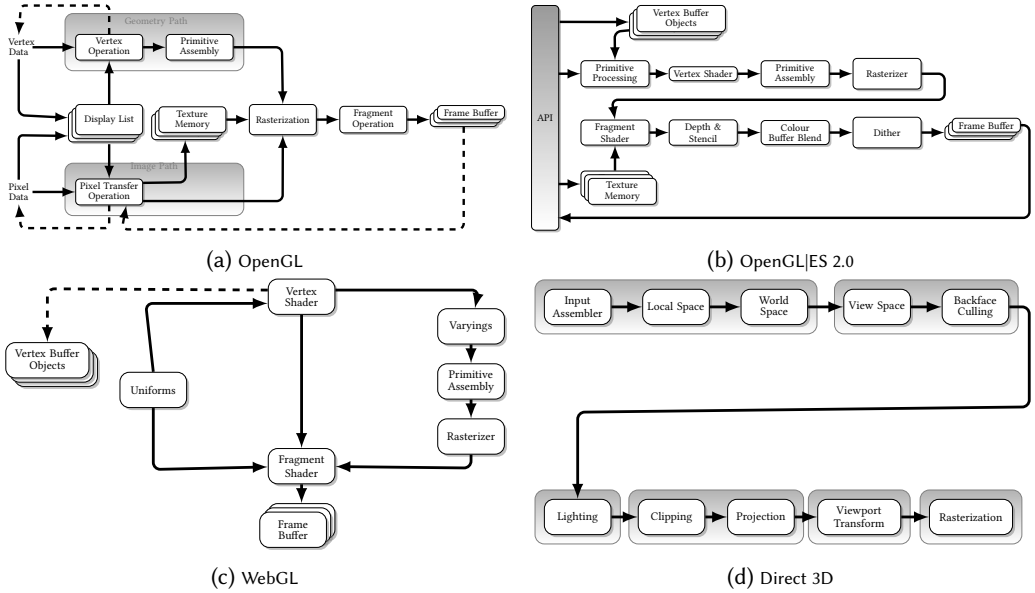


Fig. 14. **Different Graphic Pipelines**

Stealth – is a TPS 3D-game that describes a hostile environment characterized with guards (NPC) and security cameras, laser gates, key-card, and elevator. The game objects have a high rendering quality. The PC and NPC have an advanced animators, and the AI that manages NPC, laser gates, key-cards, and elevator is complex.

Survival Shooter – is a TPS 3D action-game, wherein the gamer fights against an NPC. The objective is to shoot them, eliminate as many as possible, stay alive and try to get a high score. The game is characterized by 3D physics, character animations, and many NPC.

Tanks – is a 2D MMOG where two players fight in a tank shooting game in a hostile desert environment. It is characterized by few graphical add-ons (such as rocks, palm trees, and rocky mountains). The tanks are subject to physics (explosion particles, velocity, and friction), and directional light to simulate the sun.

Space Shooter – is a 2D TPS game that takes place in space. The game is described by a large number of spacial-ships (enemies) and asteroids objects that are subject to physics: explosion particles, velocity and friction and, AI management.

Car – is a 3D racing game, with platforms. It is a standard scene, representing a car with a whole engine system (including speedup, stop, red light, turning left and right, and engine sound). The scene includes some obstacles (static objects) and has platforms like bridges and pings. The terrain is a car road with obstacles and platforms around.

Unity Lab – is a 3D TPS game with a simulation about the daily life of Dr. Charles Francis, a research scientist at Unity Lab. The Unity lab includes different rooms reachable through hallways, with dynamic doors and elevators. The environment has an improved graphics, including shading, cinematic image effects, particles systems, and lighting. The NPC is represented by a flying robot.

Multiplayer FPS – is a 3D MMOG FPS game describing a player fighting against a NPC inside an arena. The player character is a flying robot with blasters. The NPC is a humanoid character. The game is characterized with various effects for the arena such as fires, smokes, lights, sun, and storm.

A.3 Game Classification

- (i) Classification per Playability

We start by collecting the data from Figures 4, 5, and 6, which enable to classify the nine games on each platform with the two encoding qualities as *playable* or *non-playable*. Next, we use the system 1 to compact the classification through Table 6.

$$\begin{cases} Playability \wedge \neg(Playability) \Rightarrow \neg(Playability) \\ Playability \wedge Playability \Rightarrow Playability \\ \neg(Playability) \wedge \neg(Playability) \Rightarrow \neg(Playability) \end{cases} \quad (1)$$

	Dell M4800	Surface Pro	HTC M8	Galaxy S6 Edge	Browsers
Viking Village	Non-Playable	Non-Playable	Non-Playable	Non-Playable	Non-Playable
Tower Bridge	Playable	Playable	Playable	Playable	Playable
Stealth	Non-Playable	Non-Playable	Non-Playable	Non-Playable	Non-Playable
Survival Shooter	Playable	Playable	Non-Playable	Non-Playable	Playable
Tanks	Playable	Playable	Playable	Playable	Playable
Space Shooter	Playable	Playable	Playable	Playable	Playable
Car	Playable	Playable	Non-Playable	Non-Playable	Playable
Unity Lab	Non-Playable	Non-Playable	Non-Playable	Non-Playable	Non-Playable
Multiplayer	Non-Playable	Non-Playable	Non-Playable	Non-Playable	Non-Playable

Table 6. **Games classification per playability**

- (ii) Classification per Resource variability

We attribute the predicate *high variability* or *low variability* for each game on each platform and for each quality to the results obtained in Figures 4, 5, and 6. Then, we use the system 2, to compact the classification through Table 7.

$$\begin{cases} HighVariability \wedge HighVariability \Rightarrow HighVariability \\ HighVariability \wedge LowVariability \Rightarrow HighVariability \\ LowVariability \wedge LowVariability \Rightarrow LowVariability \end{cases} \quad (2)$$

- (iii) Classification per Playability and Resource variability

Joining Table 6 and Table 7, by applying the system 3, we obtain the classification of the different games (*i.e.,* Table IV : "Best option for architecture implementation per game and device").

$$\begin{cases} Playable \Rightarrow Traditional \\ \neg(Playability) \wedge LowVariability \Rightarrow Cloud \\ \neg(Playability) \wedge HighVariability \Rightarrow Offload \end{cases} \quad (3)$$

A.4 Client and Server Devices

Table 8 presents the devices used for each architecture to do our experiments.

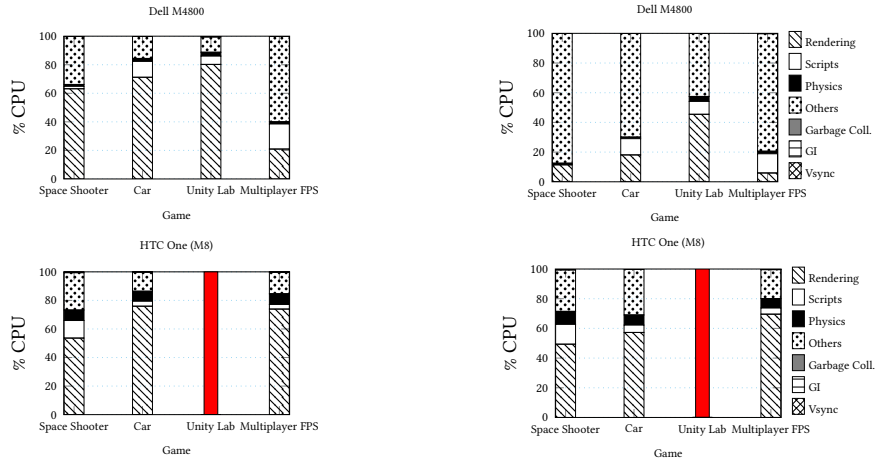
A.5 Modular results

	Dell M4800	Surface Pro	HTC M8	Galaxy S6 Edge	Browsers
Viking Village	High Variability	High Variability	High Variability	High Variability	High Variability
Tower Bridge	Low Variability	Low Variability	Low Variability	Low Variability	Low Variability
Stealth	High Variability	High Variability	High Variability	High Variability	High Variability
Survival Shooter	Low Variability	High Variability	Low Variability	Low Variability	Low Variability
Tanks	Low Variability	Low Variability	Low Variability	Low Variability	Low Variability
Space Shooter	Low Variability	Low Variability	Low Variability	Low Variability	Low Variability
Car	Low Variability	Low Variability	Low Variability	Low Variability	Low Variability
Unity Lab	High Variability	High Variability	High Variability	High Variability	High Variability
Multiplayer	Low Variability	High Variability	High Variability	High Variability	Low Variability

Table 7. **Games classification per variability**

Games	Traditional Architecture	Cloud Gaming	Computation Offloading
Tanks	Dell Precision M4800	Dell Precision M4800† Dell Tower Machine*	Dell Precision M4800† Dell Tower Machine*
Viking Village	Dell Precision M4800	Dell Precision M4800† Dell Tower Machine*	Dell Precision M4800† Dell Tower Machine*
Stealth	Dell Precision M4800	Dell Precision M4800† Dell Tower Machine*	Dell Precision M4800† Dell Tower Machine*
Space Shooter	HTC One (M8)	HTC One (M8)† Dell Tower Machine*	HTC One (M8)† Dell Tower Machine*
Car	HTC One (M8)	HTC One (M8)† Dell Tower Machine*	HTC One (M8)† Dell Tower Machine*
Unity Lab	Dell Precision M4800	Dell Precision M4800† Dell Tower Machine*	Dell Precision M4800† Dell Tower Machine*
Multiplayer	HTC One (M8)	HTC One (M8)† Dell Tower Machine*	HTC One (M8)† Dell Tower Machine*

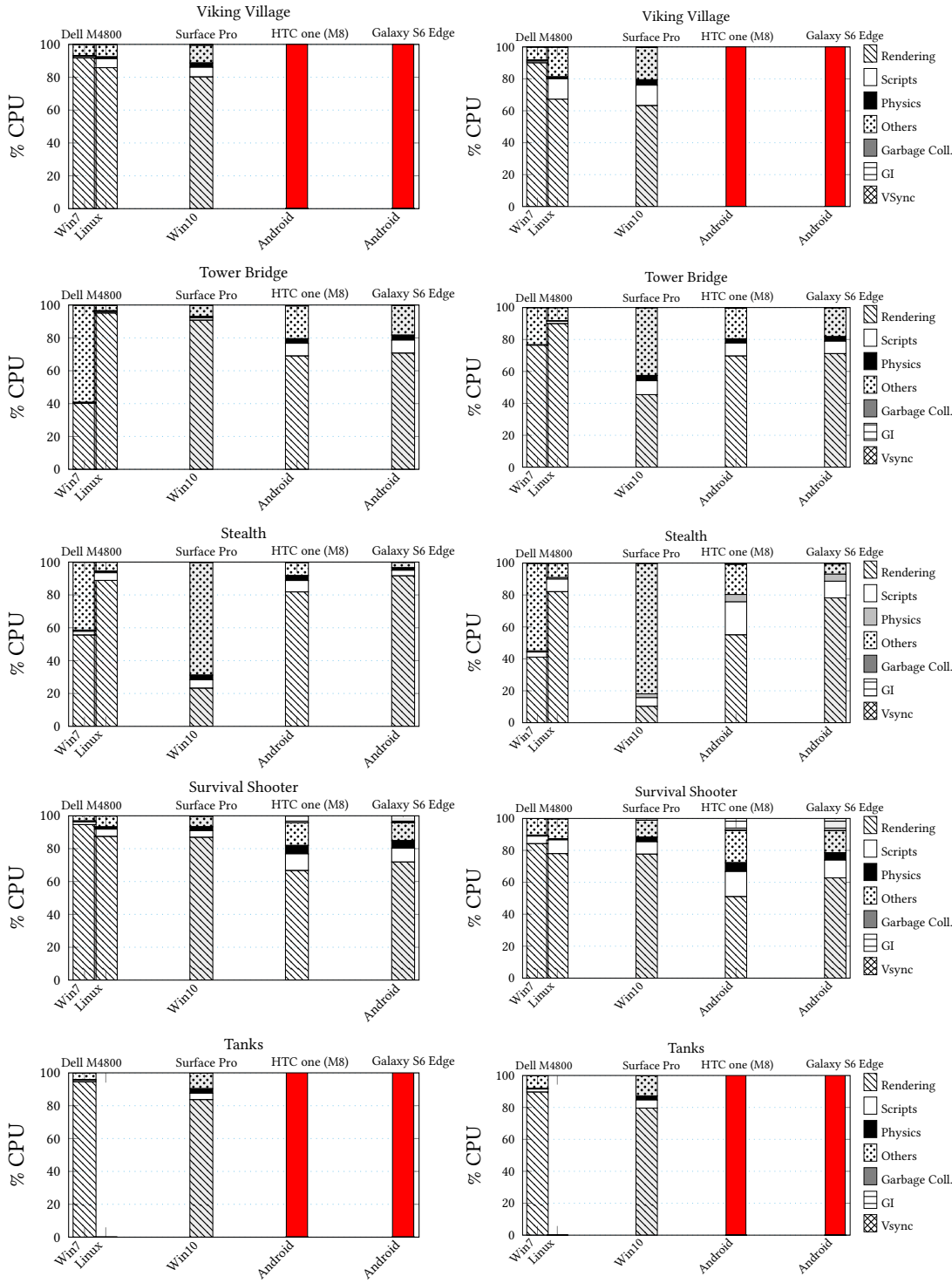
Table 8. **The chosen device for each game under the three architectures. The symbol (†) represents the client, and (*) represents the server.**



(a) Good quality

(b) Fast quality

Fig. 15. **How the CPU time is divided among modules on Dell M4800 and HTC One (M8).**



(a) Good Quality (b) Fast Quality

Fig. 16. **How the CPU time is divided among modules on different targets**