Dissecting Games Engines: the Case of Unity3D

Farouk Messaoudi
IRT B<>Com
farouk.messaoudi@b-com.com

Gwendal Simon
Telecom Bretagne
gwendal.simon@telecom-bretagne.eu

Adlen Ksentini
University Rennes 1 – IRISA
adlen.ksentini@irisa.fr

***Abstract*—Recent trends on how video games are played have pushed for the need to revise the game engine architecture. Indeed, game players are more mobile, using smartphones and tablets that lack CPU resources compared to PC and dedicated box. Two emerging solutions, cloud gaming and computing offload, would represent the next steps toward improving game player experience. By consequence, dissecting and analyzing game engines performances would help to better understand how to move to these new directions, which is so far missing in the literature. In this paper, we fill this gap by analyzing and evaluating one of the most popular game engine, namely Unity3D. First, we dissected the Unity3D architecture and modules. A benchmark was then used to evaluate the CPU and GPU performances of the different modules constituting Unity3D, for five representative games.**

I. Introduction

Today's game creators rely on *game engines* to develop the main pieces of software for their games. A game engine simplifies the task of the programmers by offering convenient abstractions for the hardware and operating systems on top of which the game runs. The purpose of a game engine is also to fully exploit the capabilities of the host machines so that the gamer can get the most immersive gaming experience possible. But today's game engines have to evolve to address two trends: cloud and mobility.

First, the concept of cloud gaming receives a growing attention because running games in controlled data-centers brings multiple opportunities. However, despite progresses in virtualization of multimedia tasks, the number of games that can simultaneously run on a given machine is still low [1]. The lack of consolidation limits the scalability and increases the cost of cloud gaming services. To serve more gamers from a given data-center, the game engines have to (i) allow resource sharing among distinct game instances, and (ii) be distributed so that the data-center resources can be better exploited, especially by distinguishing the cutting-edge modules that require specific Graphics Processing Unit (GPU) hardware to the simpler ones that can accommodate any Central Processing Unit (CPU) configuration. Unfortunately, state-of-the-art cloud gaming platforms [2], [3] still consider each running game as an individual black box.

Second, gamers play on mobile devices, for example smartphones, tablets, and Virtual Reality (VR) headsets. These devices have two main limitations: the low computing capabilities, especially GPU, with respect to the expectations, and the battery drains due to the high utilization of hardware resources. A solution is to offload the game computation on close-by powerful, plugged machines (for example desktop and set-top-boxes) [4]. Remote graphic rendering is not a new topic [5] but none of the existing solutions matches the expectations for fast rendering of high-definition, complex, 3D images. Moreover, to our knowledge, no previous work has addressed offloading non-graphic components of a game engine, for example physics module and object behavior scripts.

While the need for revisiting game engines is high, the scientific community misses papers that dissect and evaluate the state-of-the-art, *i.e.*, the current game engines. The body of literature related to game engines is surprinsingly small with regard to the significance of the gaming industries. Despite two position papers that called for research on game engine architecture [6], [7], the architecture and the performance of modern game engines have never been formally analyzed. To our knowledge, the related work focuses on applying game engines to specific areas such as serious games [8], research [9], and VR serious applications [8], [10], [11].

In this paper, we make a first step toward the understanding of game engines by analyzing the most popular game engine in the market, namely Unity3D [12]. We conduct an evaluation of performance of five representative games with different levels of quality. We analyze the performances for both CPU and GPU times to generate frames. We then analyze the processing consumption by modules, according to the types and the functions of the modules in the game engines. Finally, we extract the calls between the different modules and functions of Unity3D game engine for the generation of one frame.

Our main findings are summarized as follows:

- The time to generate a frame varies significantly, even in a given game. This high variability is a drawback regarding cloud consolidation. Moreover, the consumption time also varies for two different quality levels of a similar game, so the resource consumption is hard to predict.
- The majority of the CPU consumption and most of the GPU consumption are related to Rendering modules. These modules are tightly coupled by series of calls, which make the opportunity of offloading parts of the code less appealing in current game engines.

We first give some background information about game engine in Section II. Then, we describe in Section III the experimental platform that we set up to measure the performances of Unity3D-based games. Then, we present our performance evaluation in Section IV. Finally, we discuss the results and open perspectives regarding this first step toward next-generation game engines in Section V.

II. Background: Game Engines

We distinguish the game engine as a framework for game creators, and the game engine as a piece of code for gamers.

On the one hand, the game engine is the set of tools (including low-level libraries, User-Interface 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. We will use "framework" throughout the paper. The framework provides an abstraction layer between the game content (multimedia content and main scripts) and the underlying hardware. The most popular frameworks like Unity3D are cross-platform, *i.e.*, a given game engine can run on various operating systems and various hardware configuration.

On the other hand, the game engine is the set of software and data that eventually run on a device to provide the game to an end-user. The community of gamers considers a game engine as a piece of code. All the games that have been created by the framework Unity3D share similarities that make them consistent Unity3D game engines. We focus in this paper on typical Unity3D software, which we refer to as game engine.

A. Main Modules

A game engine consists of various modules. For a given game, some of them are mainly written by the game creator:

Artificial Intelligence (AI) – the management of Non-Player Character (NPC) is achieved by some specific modules, which mix pre-computed scenario with behaviors that are generated on-the-fly to give the illusion of intelligence.

Physics engine – these modules aim to make the game world as realistic as possible regardless of the events that are decided by the gamer (for the main character) and by the AI modules (for the NPCs). These modules also simulate the behaviors of movable elements of the world according to physics laws.

Scripting – these modules contain the gameplay itself. From the captured inputs (given by specific modules), the game developer details the series of game content and events in a scripting langage, which is specific to every framework.

Some other modules are mostly common between every game created with a given framework. These modules represent the abstraction layer and prevent game creators to spend time on low-level issues. In particular:

Input – the capture of the commands from the gamer (*e.g.*, joystick, keyboard, and now VR sensors) are all concentrated on a set of modules. These modules require minimum re-coding from the game creator.

Multimedia rendering – these modules are in charge of generating the graphical and audio elements of the game. Regarding graphics, the rendering modules have to generate one new frame every x milliseconds (ms) where x is typically between 10 and 30. Note also that the output of these modules is usually raw videos, which need to be encoded for a distant display output [13].

Networking – these modules implement communication routines and protocols for multiplayer games and server-based games. With respect to the requirements of gamers, the networking modules should be fast and cheat-proof.

B. Rendering Pipeline

We now describe the rendering pipeline of game engines, and most specifically of Unity 3D. We start with a short description of the pipeline of the low-level DirectX library, on top of which Unity 3D builds the graphical elements on Windows OS, for Mac and Linux, Unity uses the OpenGL library, and in the field of embedded like Android and iOS, and the web, Unity uses respectively OPenGl ES and WebGL. For more details about these libraries please refer to this link[1]. Then, we show how Unity 3D implements this pipeline. More information can be found in [14].

The rendering pipeline is a combination of multiple stages or steps needed to generate a 2D image based on a given geometric description of a 3D scene and an oriented virtual camera. This DirectX pipeline is a series of actions, which occur in consecutive steps as follows.

First, the input assembler reads data (vertices and indices) of 3D objects from memory and approximates it by triangle meshes. The more triangles, the better approximation but the more processing power. Each object is then put into a local coordinate system with its own orientation and size. Finally, all objects are brought together in a global coordinate system by applying geometric transformations.

The second step is related to the camera. The *View Space* is a coordinate system wherein the virtual camera is translated to the origin of the world space. It is then possible to distinguish the front side and the back side of objects. Since the back sides are not seen by the camera, all the back-face polygons are *culled*.

The *lighting* step is 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. This step also include a repetition of absorbing and reflecting light processes depending on various parameters (*e.g.*, smoothness and material of the surface and incidence angle). The *culling* step is then implemented to decide which objects should be discarded from the scene according to the computation of *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 culled.

Finally, the scene is rendered based on perspective projection of the 3D vertices onto a 2D projection window inside the frustum. The vertices coordinates are transformed to place the 2D scene into rectangle window on the screen, which is called the *viewport* by scaling and translating. The outputs of this stage are pixels. The *rasterization* transforms these pixels into screen coordinates forming a list of triangles, which should be checked and colored.

[1] https://www.khronos.org/opengl/

Unity 3D uses the DirectX as a default rendering pipeline, but the engine uses four additional rendering activities that occur in conjunction with the main pipeline. We shortly describe them hereafter.

The *Forward rendering* pipeline separates Ambient Pass for those objects that are not affected by the lights, the Light Pass for opaque objects, and the Transparency Pass where the lights and the colors of the transparent objects are combined to the colors of other opaque objects.

The *Deferred shading* pipeline is based on a smart management of geometry buffers [15] to first compute geometry and then apply lighting.

The *Pre-pass rendering* pipeline addresses the restricted usage of different material shaders in deferred shading. The lighting is stored in a new buffer after a light pre-pass rendering, which improves shading computation.

Finally, the *Vertex lit rendering* is the fastest one. 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. However, this solution does not support per-pixel effects, such as shadows, light cookies, normal mapping and highly detailed specular highlights.

III. Experimental Platform

To evaluate the performances of different game engines generated by the framework Unity3D, we ran the Unity3D version 5.02 on top of a laptop running a Windows OS (win 7 professional) using d3d11 as DirectX library for rendering pipeline. The laptop is a Dell Precision M4800 with an Intel Core i7 processor clocked at 2.8 Ghz. The laptop was improved with a Nvidia Quadro K2100M (with 2 GB and 16 GB) GPU card.

We selected five different games from the "Asset store" of Unity3D, which have different characteristics aiming at covering the most representative games in the market. We summarize the main characteristics of each game in Table I. Whilst type, dimension and game player give an idea on the game type, the other parameters are more related to the CPU consumption of rendering, physic and unity script. For more details about these games, please refer to the Asset store.

For each game we generated more than 10,000 frames for two quality types : the good quality with a reasonable game pace, here 30 frames per second (fps), and the fast quality so that the game pace is maximal. The engine achieves these two qualities depending on several parameters such as Pixel Light Count, Texture Quality, Anti Aliasing, Soft Particles and Realtime Reflection Probe for rendering, Shadow Resolution, Shadow Distance, Shadow Projection, Shadow Cascades for shadows, and other parameters like Blend Weights, Maximum LOD Level, V Sync Count.

[2]https://www.assetstore.unity3d.com/en/#!/content/29140
[3]https://www.assetstore.unity3d.com/en/#!/content/11228
[4]https://www.assetstore.unity3d.com/en/#!/content/7677
[5]https://www.assetstore.unity3d.com/en/#!/content/40756
[6]https://www.assetstore.unity3d.com/en/#!/content/20749

These frames are generated by testing, for each quality, 34 times the same scene in different positions with a different player behavior. From each test, we obtained 298 frames (due to Unity Profiler we could not get more). Hence, we made 34×2 tests for each game.

We use the profiler of Unity to get the CPU and GPU consumption statistics. The profiler saves these statistics in a log file with binary format. Unfortunately, to our knowledge, only the unity engine can read the file. Due to the non readability of the log file, we developed a script to get access to the unity serializer, so the results can be directly read from it. Thus we launched the games in a windowed mode, which automatically launches the unity profiler, aiming at writing the data in the log file. Since we obtained at least 298 frames (seen on the profiler window), we stopped the game and relaunched it on the unity editor mode, in order to load the log file to the serializer, then our script takes place, writes these readable data inside a simple text file.

To get the internal flow of Unity 3D, we dumped the memory using the Visual Studio 2015 profiler[7] in combination with Dependency Walker.[8]

IV. Performances Analysis

We provide in the following our analysis from these experiments. We first deal with the consumption of CPU and GPU and examine this consumption based on the main families of modules in Unity3D. Such an analysis is especially useful for studies related to cloud consolidation. Then, we go deeper into the calls between modules, which is the starting point for studies related to application offloading.

A. CPU and GPU Consumption

We show in Figure 1 the time spent by the CPU (respectively GPU) to generate one frame for each of the games. The results are given in Figure 1a (resp. Figure 1b). We use boxplots to present the results as we want to focus on the *variability* of both CPU and GPU consumption per frame. Indeed, the *consolidation* of resources in a data-center is easier when the consumption of processing resources can be predicted. The more stable are the CPU and GPU consumptions, the more games can concurrently run in a cluster.

We distinguish three categories of games. Some games are ideal for 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 Survival Shooter. Some other games exhibit a very high variability, for example the good-quality version of Tower Bridge and Into The Space. These are the worst cases for cloud provider, which has to reserve resources to accommodate the peaks (more than 30 ms CPU for Tower Bridge) although the median frame requires almost half the time (18 ms here) and more than a quarter of frames are done after 10 ms of CPU utilization. It means resource waste.

[7]http://blogs.msdn.com/b/visualstudioalm/archive/2015/07/20/performance-and-diagnostic-tools-in-visual-studio-2015.aspx
[8]http://www.dependencywalker.com/

Game	Type	Dimension	Game player	Rendering	Physic	Unity Script
Viking Village[2]	Video	3D	1st	+++	+	+
Tower Bridge Defense[3]	Video and Audio	2D	3rd	++	+	++
Stealth[4]	Video and Audio	3D	3rd	+++	++	++
Survival Shooter[5]	Video and Audio	3D and 2D	3rd	++	+	+
Into the space[6]	Video and Audio	2D	3rd	++	+	++

TABLE I: Main characteristics of the tested game engines

Finally, the games that are the most demanding have less variability, typically Stealth and Viking Village in our set of games. These games are good candidates for offloading due to their high resource consumption.

We also observe that a given game can be in one category for one quality level and in another one for another quality level. It is typically the case for Tower Bridge, where the good-quality game has a very large variability while the low-quality game has no variability. This observation can make the work of cloud providers even harder.

To better understand the resource consumption, we looked at the origin of this consumption. The modules of the game engines are grouped into families, which correspond to the aforementioned main components of game engines. We show the percentage of time each module family contributed to the overall consumption of resources. The results are given in Figure 2 (respectively Figure 3) for CPU (respectively GPU).

We observe that the modules that are the most CPU-consuming are the *rendering* modules. This is not necessarily intuitive since GPU is expected to be in charge of many rendering calls. The rendering modules are nonetheless responsible for up to 92% of CPU consumption for Viking Village game, and more generally more than 50% for other games. The appetite for CPU from rendering modules is a concern for code offloading. Indeed, one of the intuitive ideas for running a game engine in separate computers is to cut the game engines into families of modules, for example audio and physics in a computer and rendering in another. It appears from our results that, in general, the modules that are not related to rendering represent less than half of the CPU consumption for the demanding games.

Since the GPU is dedicated to graphic rendering actions, we show in Figure 3 the sub-families of modules related to rendering. The most important observation we make from our measurement is that each game has its own sharing of GPU resources. For example Viking Village spends most of its GPU in post processing while Stealth GPU focuses on shadows management. We let for future works a deeper analysis of the reasons behind such different GPU exploitation strategies.

Another observation is that the GPU is not excusively used for rendering processing. For example the game Into the Space entirely consumes the available GPU for other types of modules.We also observe that, for both the highest quality of Tower Bridge and the highest quality of Into the Space, the GPU consumption per frame (see Figure 1b) shares the same characteristics, where more than 50% of the frames are generated in a time between 17 and 19 ms, which is stable, but more than 25% of the frames require less than 5 ms of GPU per frame. We let for future works the analysis of GPU consumption by modules that are not related to rendering jobs.

B. Calls between Modules

We now pay attention to the offloading of modules of the game engines. The main idea is to run different modules of the same game on different computers according to the needs of these modules and to the resources provided by the computers. In the following, we restrict our studies to CPU consumption with a particular focus on rendering modules since they represent the majority of the CPU consumption.

We depict in Figure 4 the main calls between different modules. Since we analyze only the rendering modules, the other modules (for example physics, audio) are grouped into only one item to make more room for the Figure.

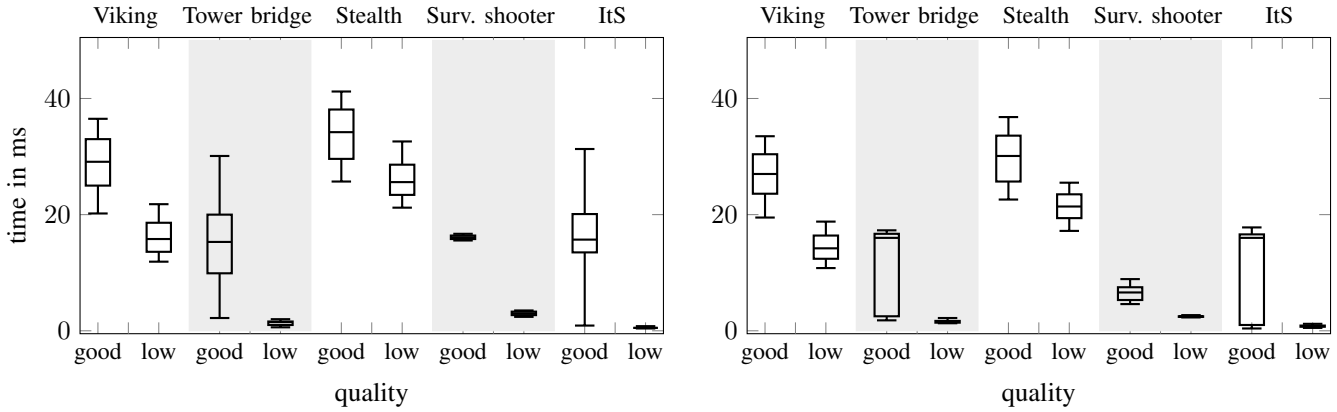
We provide two main observations from this "modular call graph". First, the modules related to rendering are rarely independent. On the contrary, calls come from multiple other modules, which are not necessarily in their own sub-families. These inter-calls among modules from different sub-families also make the code offloading harder. Indeed, frequent calls mean intensive communication between the different computers and ultra-low-delay data mirroring, which are two factors that make the implementation of offloading strategies more difficult.

Our second observation is that some modules are much more called than the other modules. For example, SharedSetPass, RenderTecture, and MeshVBOModule. They represent the modules that are eventually called to generate the image. Please note also the importance of the WaitingForJobs module. This module is used to synchronize several modules that have different running time. The time spent into the WaitingForJob module can be seen as a waste of time and CPU consumption. In the context of cloud computing, other implementations are possible to synchronize modules.

V. Discussion

In this paper, we have presented a first dissection of Unity3D game engine. Our choice of Unity3D is justified by the popularity of this engine,[9] where more than 47% of game developers are using this engine which is constantly growing, with 45% share of the full feature game engine market, and 600 million gamers spending more than $US\$110B$. We have given some details about performance evaluation with a focus on two research topics that are especially significant: consolidation in the cloud, and offloading into separate computers. Our findings are relatively negative regarding these topics,

[9] http://unity3d.com/public-relations



(a) CPU (b) GPU

Fig. 1: Time required by the according processing unit to generate one frame. The box plot includes the 10^{th}, 25^{th}, median, 75^{th}, and 90^{th} percentiles of these times.

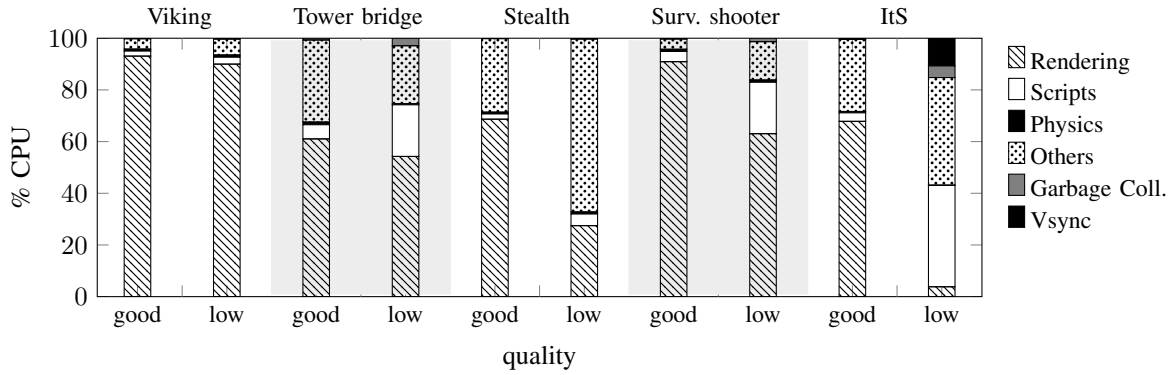


Fig. 2: How the CPU time is divided among modules

but we have also some reasons to hope for interesting future works.

The negative results are twofold. Firstly, the CPU and GPU consumption times to generate one frame exhibit both high variability during the game play and unpredictability before it starts running. This first negative observation makes the consolidation in the cloud much harder because the risks that two game engines concurrently running on one physical server have activity peaks at the same time are too high. Secondly, the modules related to rendering, which represent the majority of the CPU consumption and most of the GPU consumption are all mixed by series of calls. This second negative observation is a challenge for the implementation of code offloading since it reduces the gains in terms of performances, and it imposes to generate calls between distant machines.

But, as previously said, we also have some hopes, which are based on observations that are not enough frequent in our results. However, these observations are interesting starting points for future works.

During our experiments, we sometimes observed some patterns for the CPU consumption, with peaks occurring at regular intervals. While this phenomena is not reproducible on every game, we would like to explore further this idea and to revisit the "priority" of some modules in the task scheduler. Our hope is that some tasks can be slightly delayed so that the processing consumption of a given game fits with the other games running on the same physical device.

Another observation that we made during some of our experiments is that some modules related to rendering are mostly in a waiting mode for the CPU. It means that the CPU consumption related to these modules is not a significant consumption. Note that these waiting times do not represent 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 time represent opportunities to free some resources and to better exploit the processing units. Our future works include the validation that this observation actually holds for many games. Then we would like to explore strategies to exploit these waiting times.

Finally, even if the modules that are not related to rendering are not the most demanding for the CPU, they sometimes represent a significant part of the CPU consumption. This is especially true in games where the AI and the physics can be intensive. One of our motivations regarding the code offloading is to study the gains when the display device only embeds

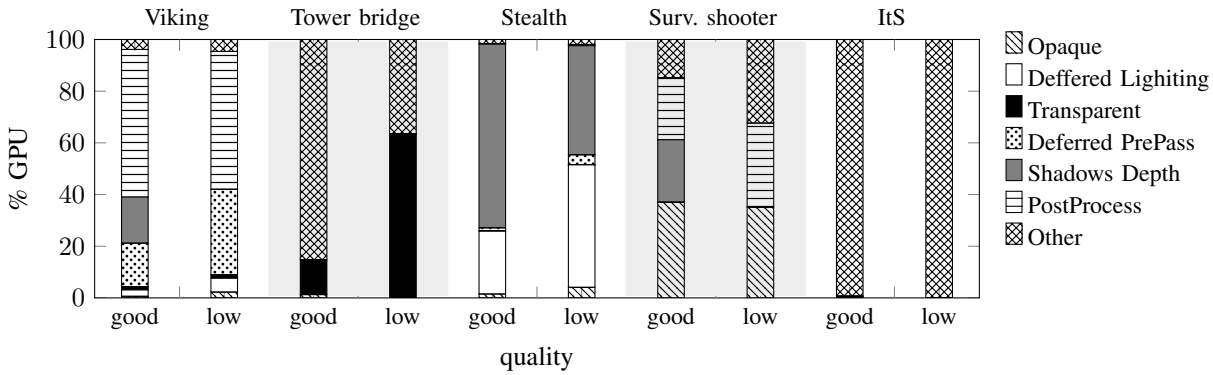


Fig. 3: How the GPU time is divided among modules

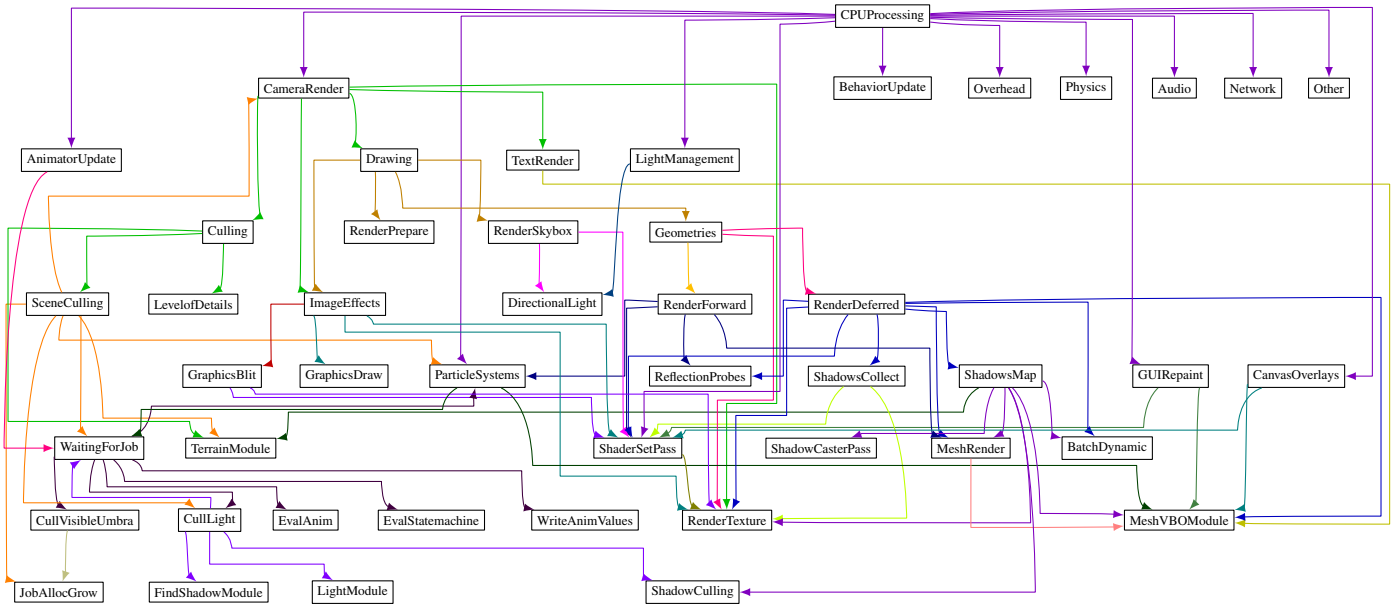


Fig. 4: Description of the calls between the main modules related to Rendering on the CPU

dedicated GPU resources and powerful servers are available nearby. This configuration is the case in the new generation of game centers with VR headsets.

References

[1] R. Shea, D. Fu, and J. Liu, "Cloud gaming: A reality check towards public cloud deployment," *IEEE Transactions on Circuits and Systems for Video Technology*, 2015.

[2] C. Huang, K. Chen, D. Chen, H. Hsu, and C. Hsu, "Gaminganywhere: The first open source cloud gaming system," *ACM Trans. on Multimedia Comp., Comm., and App. (TOMM)*, vol. 10, no. 1s, p. 10, 2014.

[3] M. Luo and M. Claypool, "Uniquitous: Implementation and evaluation of a cloud-based game system in unity," in *Proc. of IEEE GEM*, 2015.

[4] M. Satyanarayanan, Z. Chen, K. Ha, W. Hu, W. Richter, and P. Pillai, "Cloudlets: at the leading edge of mobile-cloud convergence," in *Proc. of MobiCASE*, 2014.

[5] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski, "Chromium: a stream-processing framework for interactive rendering on clusters," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 693–702, 2002.

[6] E. F. Anderson, S. Engel, P. Comninos, and L. McLoughlin, "The case for research in game engine architecture," in *Proc of the 2008 Conf. on Future Play*, 2008.

[7] J. Tulip, J. Bekkema, and K. Nesbitt, "Multi-threaded game engine design," in *Proc. of the Interactive Entertainment conf (IE)*, 2006.

[8] B. Cowan and B. Kapralos, "A survey of frameworks and game engines for serious game development," in *IEEE Int. Conf. on Advanced Learning Technologies (ICALT)*, 2014.

[9] M. Lewis and J. Jacobson, "Game engines," *Communications of the ACM*, vol. 45, no. 1, p. 27, 2002.

[10] B. J. Kot, B. Wuensche, J. C. Grundy, and J. G. Hosking, "Information visualisation utilising 3d computer game engines case study: a source code comprehension tool," in *Proc. of the 6th ACM Conf. on Computer-Human Interaction (CHI)*, 2005.

[11] S. Marks, J. A. Windsor, and B. Wünsche, "Evaluation of game engines for simulated surgical training," in *Proc of the 5th ACM Int. Conf. on Computer Graphics and Interactive Techniques (Graphite)*, 2007.

[12] "Unity: The leading global game industry software," accessed in Aug. 2015, http://unity3d.com/public-relations.

[13] R. Shea, D. Fu, and J. Liu, "Towards bridging online game playing and live broadcasting: Design and optimization," in *Proc. of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2015.

[14] M. V. de Hoef and B. Zalmstra, "Comparison of multiple rendering techniques," University of Utrecht, Tech. Rep., 2010.

[15] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The triangle processor and normal vector shader: a VLSI system for high performance graphics," in *Proc of the SIGGRAPH Conf*, 1988.