An API-Level Frame Workload Model for Real-Time Rendering Applications

Iman Soltani Mohammadi [1], Mohammad Ghanbari [1], and Mahmoud-Reza Hashemi [1]

[1]Affiliation not available

October 30, 2023

Abstract

This work proposes a workload model for modern graphics APIs, named GAMORRA, to estimate each frame's workload and possibly predict rendering times on a target hardware.

Also, a suite of benchmarks is described that evaluates the performance of the target hardware and the overhead of different graphical operations to be used in the proposed model for workload estimation.

Posted on 11 May 2020 — CC-BY-NC-SA 4 — https://doi.org/10.36227/techrxiv.12278666.v1 — e-Prints posted on TechRxiv are preliminary reports that are not peer reviewed. They should not b...

An API-Level Frame Workload Model for Real-Time Rendering Applications

Iman Soltani Mohammadi

School of Electrical and Computer Engineering, College of Engineering, University of Tehran, soltani.m@ut.ac.ir

Mohammad Ghanbari

School of Electrical and Computer Engineering, College of Engineering, University of Tehran, ghan@ut.ac.ir as well as School of Computer Science and Electronic Engineering, University of Essex, UK, ghan@essex.ac.uk

Mahmoud-Reza Hashemi

School of Electrical and Computer Engineering, College of Engineering, University of Tehran rhashemi@ut.ac.ir

ABSTRACT

This paper introduces a mathematical workload model of a modern graphics API pipeline, named as GAMORRA which works based on the load and complexity of each stage, either fixed-function or programmable. GAMORRA models each stage of the pipeline based on the stage's operation complexity and the size of the data that these operations are performed on. To showcase GAMORRA's capabilities, it is used to estimate real-time applications' frame times in practice. A suite of benchmarks is also utilized to determine the processing time of each stage of the pipeline on the user's device based on its mathematical model to be fed to GAMORRA. These benchmarks focus the overall processing load of the pipeline on a specific stage for each benchmark to determine the performance of that stage. Our experiments on direct3d11 show a 92.1% accuracy in estimating frame times.

CCS CONCEPTS

• Software and its engineering → Software organization and properties → Contextual software domains → Software infrastructure → Middleware • Software and its engineering → Software organization and properties → Software system structures → Real-time systems software • Software and its engineering → Software organization and properties → Software system structures → Abstraction, modeling and modularity • Computing methodologies → Computer graphics → Rendering → Rasterization •Mathematics of computing → Mathematical analysis → Functional analysis → Approximation

KEYWORDS

Graphics API, Workload modeling, Frame time estimation, Real-time rendering

1 Introduction

Graphics pipelines have evolved extremely during the past two decades, especially real-time rendering engines which are mostly used in computer games. Computer games produce heavily variant workloads in different scenes. Graphics streaming [1] in cloud gaming, as opposed to traditional CG [2], is gaining more attention in studies and estimating the rendering time of frames in a computer game is mandatory in such systems to ensure that game frames are rendered in time on the user's device. Also, DVFS-based power management systems for mobile games need to take into account the amount of workload of each frame [3, 4]. These studies usually rely on simple workload models [5] or mostly focus on predicting the upcoming frames' workload. Instead, by utilizing an efficient model which can give a pretty good estimate of the workload of each frame without the need to render the frame, corresponding applications, e.g. power management of mobile GPUs or frame time estimation in graphics streaming systems, wouldn’t need to rely on inaccurate prediction methods. Current frame workload estimation methods are scarce and usually utilize a linear prediction model, which contradicts the true nature of workloads even in consecutive frames.

Due to the high variety of GPU architectures among vendors, a hardware level model effectively limits the applicability of such a model to a specific hardware device. Although each graphics driver covers a certain range of hardware models, they also differ significantly from each other due to their hardware dependent nature and the frequent updates they receive. But graphics APIs usually follow a certain model for their rendering pipeline with minor differences between different APIs’ pipeline architecture. So, designing a mathematical model to estimate frame times on an API-level is more effective and covers a much wider range of real-time applications, compared to a hardware architecture or driver level model. Additionally, utilizing an API-level model avoids the need for game engine modifications which are not possible to be applied to commercial off-the-shelf software. Therefore, a Graphics API-level Model of Rendering workload for Real-time Applications, *GAMORRA*, is proposed in this paper.

GAMORRA is a performance model that estimates the worst-case rendering time of a frame for real time applications like computer games. Graphics streaming [1, 6] or joint video-graphics streaming cloud gaming systems [7] which aim to improve the overall gaming experience of users that use low end hardware, require to estimate the rendering time of each frame practically to make sure that the client’s device can handle the workload of the game for every frame. Also, real time collaborative rendering platforms like Kahawai [8] and other similar platforms [9] that adjust the workload of each frame of the game based on the rendering capability of the client’s device, need to set the graphical level of details of every frame based on the thin client’s computational power.

The primary goals of this work are as follows:

- *Presenting a reliable model for graphics APIs rendering pipeline in real-time applications*: In GAMORRA, workload modeling is performed on an API level so that no modification in the game engine and no drastic change in the model, due to hardware upgrade or software update, are required.
- *Practicality and supporting commercially available off-the-shelf software*: GAMORRA aims at presenting a general framework that takes the graphics data of each frame as input after they are being produced by the rendering engine. Taking this approach removes the need for accessing the graphics engine itself.
- *Simplicity and real-time functionality*: It is very important for GAMORRA to not impose too much overhead on the system because that would defeat the purpose of such a model.
- *Designing a benchmark suite to evaluate the target system’s performance*: Since the underlying graphics driver and hardware are responsible for carrying out the graphical workload, a benchmark suite is required to be performed on the target device to evaluate the performance of hardware and software involved in rendering the application. Direct3d 11 is chosen as a testcase of GAMORRA’s reliability in modeling a modern graphics API. Benchmarks are also performed on an API level so that they can capture the underlying software and hardware behavior and performance as well as the behavior of the API’s software itself.

The primary contribution of GAMORRA is the mathematical workload model of a modern graphics API and the benchmark suite designed to work alongside this model for frame time estimation. Proper benchmarking has always been a focus in the field of computer graphics and GAMORRA, with its detailed model of a rendering pipeline, provides a blueprint for benchmarking of computer games for developers as well as a general graphics benchmarking framework for already released games. The rest of this paper is organized as follows: next section discusses notable works in this field and how GAMORRA differs from these methods. Section 3 explains the core design and functionality of GAMORRA in detail. Section 4 focuses on the implementation details of the proposed method and the experimental results and finally section 5 concludes the paper.

2 Related works

A limited number of studies have focused on estimating frame times targeting different applications for their proposed methods. In [10], a behavior-aware power management system is proposed for mobile games which

estimates each frame's workload based on game application's system calls and texture processing load. But there are much more contributing factors to rendering time of frames than only the number of API calls and the size of texture data like the size of 3D data models.
Song et al [11] proposed a fine-grained GPU power management for closed source mobile games which works based on the number of vertices, the number of commands and the size of textures. Such an approach doesn't take into account the effects of other contributing factors like the structure and performance of the graphics API that processes all the aforementioned parameters. This causes such a model to produce the same results for different APIs.
Also, Zhang-Jian et al [3] model the workload of a computer game simply based on the number of triangles and pixels. Since the main focus of such studies is to predict the upcoming frames' workload [12], rather than precisely modeling the workload of each frame based on available detailed frame data, they use controllers such as PID controllers along with linear models for workload prediction.
Some of the studies in this field require hardware and software changes to perform properly [13, 14] which is not desired in the case of closed source software and already available hardware. Also, in [5], a simple model is proposed to determine the workload of graphics frames which is based on the number of triangles and might not suffice in a realistic scenario. So, for GAMORRA to become fully needless of any modifications in the rendering engine, it is implemented at an API level without the need to modify the API itself as well.

3 GAMORRA

GAMORRA acts as a middleware that resides between game engine and the graphics API software, capturing the output API commands produced by the engine. Figure 1 shows the placement of GAMORRA in a computer system and its relationship with the rendering engine and the graphics API layer. GAMORRA analyzes the graphics data stream on the fly and feeds each command to the graphics API as soon as it is done analyzing it. These analyses provide the value of the contributing factors to the workload so that they are fed to the mathematical model. To estimate frame times, a performance evaluation of the target device is mandatory before GAMORRA is ready to be used. After the performance signature of the target device is acquired, the pipeline model is tuned by the performance results and the setup is complete.

3.1 Workload model

Today's graphics pipelines are composed of multiple programmable stages as opposed to the fixed function stages of old devices. A modern graphics API pipeline (Direct3d 11 in this case) is shown in figure 2.

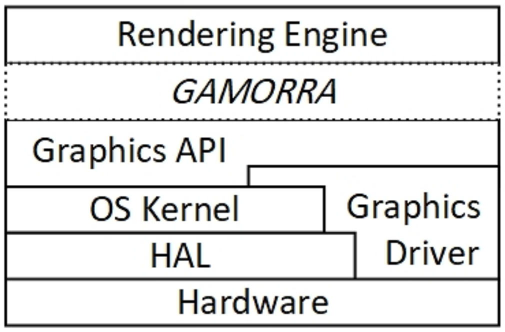


Figure 1: A rendering system hierarchy in presence of GAMORRA

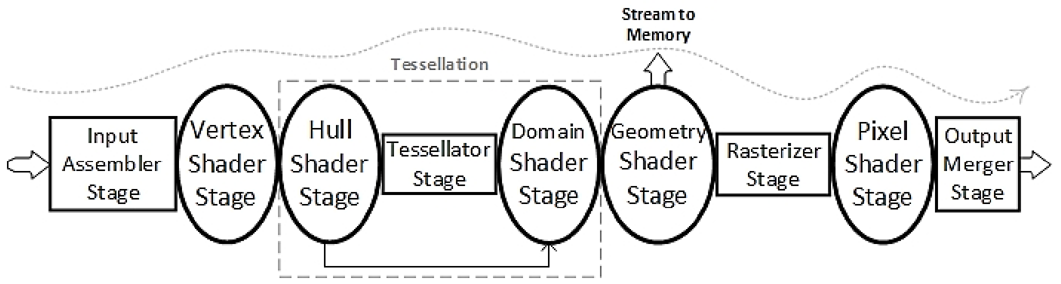


Figure 2: Direct3d 11 graphics rendering pipeline

Direct3d 11's graphics pipeline consists of 4 fixed function stages (depicted by rectangles) and 5 programmable stages (depicted by oval-shaped containers), a total of 9 stages. The programable stages of a rendering pipeline are called *Shaders*. To model the workload of each stage, the following generic formula is proposed which might differ in some details for different stages due to their unique characteristics:

$$W_{X,L_x} = Perf_X(L_X)/\eta \qquad (1)$$

where $Perf_x$ represents the performance function of xth stage in time unit and L_x stands for the computational load of each stage. Also, η determines the number of cores of the target device. $Perf_x$ is determined by custom graphics benchmarks that focus on executing a specific operation for a certain number of times to map each amount of load to the corresponding processing time. The final form of the function depends upon multiple factors like the performance and quantity of GPU cores residing in user's device. L_x stands for the computational load of the xth stage. This load depends on the input size of each stage as well as the stage's code complexity for shaders. L_x is also different for each stage and its value depends on the content of the intercepted graphics data.

For a fixed function stage, L_x mainly depends on the number of inputs or outputs of that stage. But, the overall processing load of shaders is strongly affected by their shader program as well. The complexity of the shader program is obtained through analyzing its assembly code which is comprised of a series of instructions each of which performs a specific operation based on its opcode. The estimated time of each assembly operator on user's device is obtained through multiple benchmarks that cover all the operations and is needed to be performed before starting a rendering session. So, the complexity of a programmable stage, C_X, and consequently, L_X is calculated as follows:

$$C_X = \sum_{m=0}^{N_{OP}-1} op_m.n_m \qquad (2)$$

$$L_X^P = N_X.C_X \qquad (3)$$

In the above equations, x determines the stage, N_X stands for the number of elements that the stage operates on, op_m stands for the mth operator's weight which is derived from the benchmark results, n_m is the number of times that the mth operator is used in the corresponding shader and N_{OP} is the total number of operators. op_m is actually the ratio of the mth operator's processing time for a certain load to the simple assignment (=) operator's time for that load. Our experiments indicate that texture sampling operators cost more than mathematical operators due to the texture bandwidth limitation.

To precisely model a stage, which means to find its choke point along with the main parameters that triggers it, the functionality of each stage should be discussed. A vertex is a data structure that holds the attributes of a point such as its position, color, normal and texture coordinates in a graphics model [15, 16]. A scene in a game is comprised of multiple models and to render a frame of a scene, the frame is broken down to multiple batches. Each batch has a different pipeline state along with possibly different resources that results in a *drawcall* which draws the pixels prepared by the current batch [17].

After estimating the required time to complete each stage by means of equations (1), (2) and (3), the overall processing time of each batch can be obtained as:

$$Batch_b = \omega + \sum_{n=1}^{\mathrm{N_{Stage}}} W_{n,L_n} \qquad (4)$$

where *n* determines one of the $\mathrm{N_{Stage}}$ stages of the pipeline for the b^{th} batch, and ω is the minimum workload of the simplest pipeline state and shaders to render one batch. Since the main application for GAMORRA in this paper is to estimate frame times, the value of ω is considered to be in time unit.

Overestimating frame times leads to inefficiency while underestimating them would lead to a laggy application experience for users. Batches can be processed in parallel, but there is no way of accessing the user's internal hardware affairs before they are done, so the proposed frame time model is based on the worst-case scenario in which all the batches are rendered sequentially where *b* is the batch number in the i^{th} frame:

$$Frametime_i = \sum_{b=0}^{B_i-1} Batch_b \qquad (5)$$

3.1.1 IA stage

The vertex data needs to be loaded to GPU memory to process a batch. The Input Assembler (IA) stage reads and prepares the vertex data that are required for the current batch by determining their attributes and topology. For the IA stage, since the data is read from resource buffers, the available memory bandwidth becomes the potential bottleneck, thus, the load of the IA stage (L_{IA}) is chosen to be the size of the input vertex data which might vary based on the number of vertices and their attributes.

$$L_{IA} = \mathrm{N_{Vertex}}.\sum_{m=0}^{N_{Attr}} Size(Attr_m) \qquad (6)$$

$$Time_{IA} = Perf_{IA}(L_{IA}) \qquad (7)$$

Where N_{Vertex} stands for the number of input vertices and $Size(Attr_m)$ represents the size of the m^{th} attribute of a vertex. The benchmark that obtains $Perf_{IA}$ should focus the overall workload on the IA stage. To this end, a certain number of vertices are fed to the pipeline with all the stages either set to off for optional stages or set to pass through for other shaders. The value of N_{Vertex} at which the vertices overburden the data bus and the GPU is unable to read them in time (e.g. a maximum of 33.3 ms for rendering at 30 fps), is the maximum value to be tested for this parameter in the benchmark. Also, there is a need to make sure that none of the vertices are rasterized, otherwise the rasterization delay would also be added to $Perf_{IA}$ which is undesirable. This means that the vertices need to be outside the camera's view so that they are clipped by the clipper. Although the clipping and culling operations do affect $Perf_{IA}$ but it is much subtler than the rasterization overhead. Since all the vertices are outside the camera view, they will all be clipped, so the performance of the clipper directly affects the final results and might even be the real bottleneck which should be taken into account as well for this stage and all the other stages that don't need the rasterization to be performed on the vertices. If the clipper becomes the bottleneck, then the number of primitives becomes the contributing factor to the IA stage's delay which is already taken into account in equation (6).

3.1.2 VS stage

The Vertex Shader (VS) stage, which is the second unit in the pipeline, manipulates the attributes of a vertex based on the functionalities defined in the VS program that is set for the current batch on the pipeline. Since VS is a programmable stage, its performance heavily depends upon the complexity of the shader's code and should be reflected in L_{VS}. VS program is invoked individually for each vertex, so L_{VS} is also affected by the number of vertices and is defined as follows:

$$L_{VS} = C_{VS}.N_{Vertex} \qquad (8)$$

C_{vs} is the assembly code complexity and N_{Vertex} is the number of input vertices. For programmable stages, the assembly operators used in shader assembly code are profiled separately and a performance function is derived for each operator. Some operators are used exclusively in a specific stage (e.g. sampling operator for PS). To benchmark the VS stage, like the IA stage, all the vertices need to be outside the camera view to avoid rasterization. VS passes the processed vertices to the Tessellation stages.

3.1.3 Tessellation stages

Hull Shader (HS), the Tessellator and Domain Shader (DS) stages are optional stages that allow hardware-based tessellation for smoother model rendering and they manipulate the vertices in a *patch*. A patch is a primitive data type used only for tessellation and is comprised of up to 32 arbitrary vertices (in d3d11) with no implicit topology. HS consists of a main shader program and a patch constant function that are executed once per *output* control point and once per patch respectively and can generate or remove control points statically. So, the load of this stage can also be simply considered as the number of vertices (N_{Vertex}) along with the number of patches (N_{PCF}) as the input to the patch constant function which is treated like a complete shader stage and analyzed by its assembly code. The complexity of HS's main shader and patch constant function are shown by C_{HS} and C_{CF} respectively while the load for each one is represented by L_{HS} and L_{CF}. $Perf_{HS}$ and $Perf_{CF}$ are also obtained through custom benchmarks.

$$L_{HS} = C_{HS}.N_{Vertex} \quad (9)$$
$$L_{CF} = C_{CF}.N_{PCF} \quad (10)$$
$$Time_{HS} = Perf_{HS}(L_{HS}) + Perf_{CF}(L_{CF}) \quad (11)$$

The Tessellator stage is also a fixed function unit and its inputs are the tessellation factors and patch constant data that are produced by HS's patch constant function. This stage generates the coordinates of new control points in a primitive based on the inside and edge tessellation factors for a generic domain shape. For this stage, L_{Tess} mainly depends on the total number of newly generated points in each patch where the total number of patches is shown by P and the number of tessellations in the patch by N_{Tess}:

$$L_{Tess} = \sum_{n=0}^{P-1} N_{Tess} \quad (12)$$

Domain Shader (DS) stage is fed with the output of the Tessellator and HS stages, namely UVW coordinates of every point in a patch from the Tessellator along with control points and patch constants from the HS stage. The DS stage produces a single tessellated vertex per input vertex, So L_{DS} is equivalent to the number of vertices:

$$L_{DS} = \mathrm{N_{DS}} \quad (13)$$

For benchmarking, tessellation shaders (HS and DS) are treated as normal shaders, like VS. The tessellator stage on the other hand, is totally different from a programmable stage but easier to benchmark due to its fixed functionality. The tessellator produces smooth texture mapping coordinates out of a rough low-detail model based on the tessellation factor. So, varying the number of input data and the tessellation factor while all the other shaders are off or set to pass-through, results in the performance function of the tessellator stage.

3.1.4 GS stage

Geometry Shader (GS) is an optional stage which handles complete primitives instead of a single vertex and can add or remove geometry. So L_{GS} is also dependent upon the number of vertices which might be different from the input of VS due to being processed in tessellation stages (if tessellation is on):

$$L_{GS} = \mathrm{N_{GS}} \quad (14)$$

The complexity and the number of geometries that go through GS contributes to the overall delay of this stage of the pipeline. It can be benchmarked by varying these parameters for each iteration and taking into account the results of the previous stages as well while the Tessellation is off and other stages are set as pass through.

3.1.5 Rasterizer

Rasterizer is also a fixed function stage that interpolates vertex attributes and performs vertex clipping and back-face or front-face culling to map the vertices to the viewport. This stage generates fragments that might end up on screen as pixels. The number of primitives and the target resolution directly affect the performance of this stage, so, L_{Ras} would be as:

$$L_{Ras} = N_{Prim}.N_{Attr}.N_{Width}.N_{Height} \quad (15)$$

Resolution can make rasterizer a bottleneck in the pipeline and reducing it strongly improves the performance of a graphical application in such situations. Also, an increase in the number of triangles results to an increase in the total number of computations that are needed for rasterization because it increases the number of fragments that are produced. So, we need to benchmark rasterizer based on resolution, the number of attributes associated with vertices and the number of triangles that comprise the scene.

3.1.6 PS stage

Pixel Shader (PS or fragment shader) is the last programmable stage that manipulates each input fragment's color [18]. Fragments are mainly candidates for the final pixel values and as the number of fragments that are produced by the Rasterizer increases, the number of times that a PS is invocated increases as well. Also, the shader code complexity should be considered in the model:

$$L_{PS} = C_{PS}.N_{Fragment} \qquad (16)$$

In this case, fragments should be rasterized so that PS can be applied to them. The rasterization delay should also be accounted for to obtain the proper performance function for the PS stage.

3.1.7 OM stage

Output Merger (OM) is the last stage of the pipeline and the fragments that are processed by the PS are fed to this stage. OM calculates the final color of a pixel based on fragments and depth information which is also received from PS or the Rasterizer. Reading and writing to the render targets are the main cause of the performance issues related to the OM stage when blending is utilized. So, this stage is mostly bandwidth limited and is affected by the number of fragments along with the render target resolution and is defined as:

$$L_{OM} = N_{Width}.N_{Height}.N_{Fragments} \qquad (17)$$

As the number of the overlapping fragments for a specific pixel increase, the overhead that the OM stage introduces to the final rendering time becomes more significant while blending. Since the rasterization performance function is already known at this stage and other stages are set to pass through or are turned off, the performance of the OM stage can be obtained through varying the number of overlapping fragments along with the target resolution which also agrees with equation (17).

3.1.8 Compute Shader

Compute Shader (CS) is also a programmable shader with an independent logical pipeline dedicated to general computations. Although it is not part of the main pipeline, this shader should be considered in the performance model, following the same rule for input size and cost (code execution time) estimation.

3.2 Benchmark notes

CPU also affects frame times immensely and can potentially become the bottleneck in a rendering session. But CPUs' performance model is already discussed in literature [19] from as late as 1977 and are much simpler to obtain. So, the estimation of the CPU's time is not discussed in this work but is considered in the experiments.

Although the underlying hardware (a single GPU core) is the same for all the stages, the programmatical differences result in different contributing parameters and performance as defined in the previous section. Since actual rendering results are obtained and used as performance functions, a set of extensive benchmarks that covers the basic functionalities of the rendering pipeline for a certain number of parameters are required. Each benchmark focuses on a single operation in a single stage of the pipeline while isolating other stages and omitting other operators as much as possible.

In addition to each stage's functionality, other pipeline states, e.g. the presentation model or blend mode, directly affect the final performance and need to be addressed in the benchmarks. For example, in direct3d 11, DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL and DXGI_SWAP_EFFECT_DISCARD presentation models would give two very different performance results based on the rendering resolution, which need to be taken into account while designing the benchmarks.

Also, the benchmarks should account for the post transform cache [20] implemented in GPUs. This cache causes the VS to be invocated less frequently for indexed draw calls. So, to properly have a 1:1 relation between the vertices and the number of times that a VS is invoked, indexed draws should be avoided unless benchmarking actual post transform cache performance.

4 Implementation and results

API Trace [21] is used to intercept API commands that are produced by the rendering engine. Since computer games are the main applications that use graphics APIs, like Direct3d, to their maximum capacity, all the tests are done on modern AAA games. Also, to cover applications with lower frame rate demands, workloads that lead to a maximum of 100 ms of delay (10 fps) are performed by the benchmarks.

4.1 Obtaining performance functions

The results of benchmarking a device are shown in Figures (x-y). The tested device is packed with a Core i7 7500U working at a 2.70 GHz frequency, a single module of 8 GBs of DDR4 RAM working at 1200 MHz along with a dedicated GTX950m GPU with 2 GBs of GDDR5 VRAM and runs the latest x64 version of Windows 10.

Since the Tessellation and GS stages are optional, they are disabled for the current test. Also, blending is not used and since the OM stage is not likely to become a bottleneck, the performance function for this stage is ignored as well.

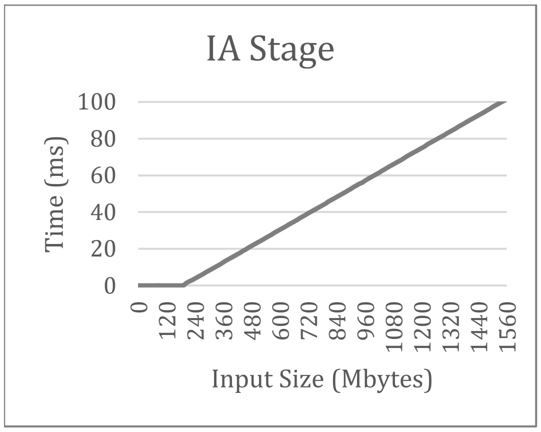


Figure 3. The IA Stage's performance chart

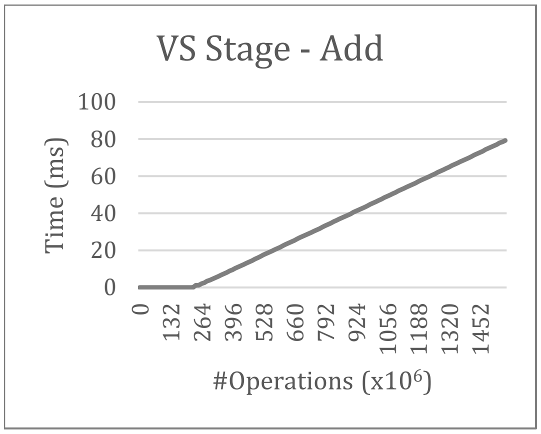


Figure 4. The VS Stage's performance chart for *Add*

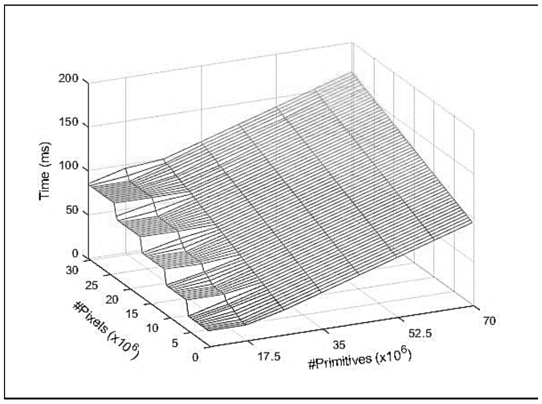


Figure 5. The Rasterization stage's performance chart

With a θ of 6.966 ms, Figure 3 shows the result of the performance benchmark of the IA stage. It is evident that if the size of loaded vertices into the pipeline surpasses 1500 MBs, GTX950m would struggle to load them. After being prepared by the IA stage, the vertices go through the VS stage. Tens of opcodes are available to be used by developers at this stage but, for the sake of an example, only the *Add* operator is considered here. The performance chart of this operator is shown in Figure 4. This benchmark shows that a VS can perform of up to 130 million *Add* operations under a 100-ms time interval. Normally, the performance function of this stage would be a 3d chart, but, for simplicity and more comprehensible output, the number of attributes and the number of vertices is fixed and set to 1. The only attribute considered in this test is *position* which is a 3-component floating point variable.
Figure 5 shows the benchmark results for the rasterizer. As the number of primitives increase over 18 million, the rasterizer starts to impose larger overhead on the pipeline. Also, as the number of pixels start to surpass 4K resolution (almost 9 million pixels), the GTX950m starts to struggle with the rasterization process. When the rasterization benchmark is 3 million pixels, short of the 8K resolution, the rendering time gets closer enough to the unacceptable 100 ms especially for 10 million primitives and more.

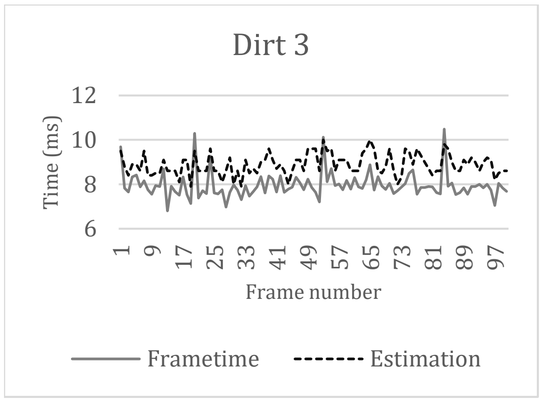
4.2 Estimating frame times

After establishing the performance functions of the target graphics card, GAMORRA is ready to be used to estimate frame times. Six AAA computer games were chosen for the tests, namely, Dirt 3 (Racing), Splinter Cell: Blacklist (third person, stealth), Battlefield Bad Company 2 (first person shooter), Far Cry 3 (open world first person shooter), Rocket League (sports, racing) and Trine 4 (arcade side scroller). If the corresponding game utilizes any of the optional stages, they should be also taken into account.
Table (1) shows the miss rate of frame time estimation for each game. Higher miss rates indicate more frame time under estimation. The results show that the performance of GAMORRA strongly depends upon the standard deviation of frame times and doesn't depend much on factors like genre or average frames per second. Usually, consecutive frames tend to have similar structure and consequently, similar frame times. Our experiments show three main reasons for high deviation in consecutive frame times: 1- software inefficiency or bugs, e.g. graphics driver, OS or game engine 2- weak hardware or incompatibility between hardware and software, e.g. not meeting the minimum system requirements and 3- excessive use of GPU demanding visual effects e.g. motion blur. The more stable a rendering session is, the more efficient and precise becomes the process of frame time estimation by GAMORRA.

Figures (6) and (7) show the results of frame time estimation for Dirt 3 and Splinter Cell Blacklist in comparison to the actual frame times in a 100-frame sequence of gameplay for each game. These two games are chosen as the best-case and worst-case scenarios since they have the lowest and highest miss rates respectively. Dirt 3 utilizes Ego engine [22] which is specifically designed and used for racing games, thus, performs efficiently on a mid-range GPU. On the other hand, Splinter Cell uses a pretty much outdated unreal engine 2.5 [23] which is pushed to its limits for the best visual output leading to a somewhat unstable rendering performance, hence the drop in GAMORRA's accuracy.

Table 1. Results of frame time estimation using GAMORRA

Game	Miss Rate (%)	Average Frame Time (ms)	Standard Deviation	Genre
Battlefield BC2	6.30	12.22	0.943	FPS
Dirt 3	5.89	7.97	0.588	Racing
Far Cry 3	6.98	24.51	1.213	FPS
Rocket League	7.14	16.13	1.247	Sports
Splinter Cell	11.35	14.74	3.093	TP Stealth
Trine 4	9.76	16.42	2.582	Side Scroll



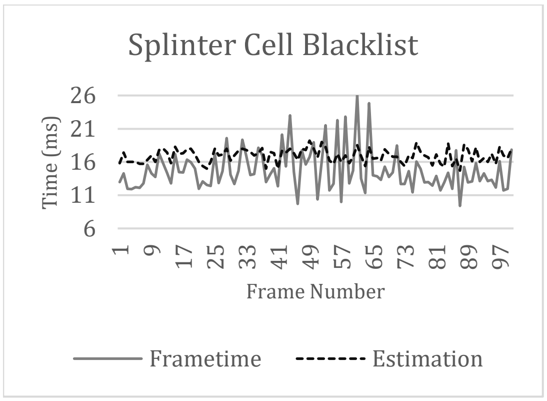


Figure 6. The Rasterization stage's performance chart

Figure 7. The Rasterization stage's performance chart

5 Concluding Remarks

This paper proposes GAMORRA, an API-level workload model for graphics-based applications (mainly computer and mobile games). Modeling the workload of a game's frames proves useful in different applications like DVFS-based power management schemes in smartphones or estimation of performance measures like frame times in a graphics streaming-based cloud gaming system. GAMORRA takes into account the overall structure of a graphics rendering pipeline along with the size of input data, i.e. vertex and texture data. Since modern graphics APIs take advantage of multiple programmable stages called Shaders, in their rendering pipeline, the complexity of each stage is defined as the complexity of the stage's code which is accessible by intercepting each frame that is generated by the corresponding application's engine. Also, to account for the processing power of the rendering hardware and the performance of the software involved in the rendering session (e.g. the graphics driver or the Graphics API's software), a thorough benchmark suite is designed which evaluates the rendering system's processing power to estimate frame times.

REFERENCES

[1] Itay Nave, Haggai David, Alex Shani, Yoav Tzruya, Arto Laikari, Peter Eisert and Philipp Fechteler. 2008. Games@large graphics streaming architecture. In *Proceedings of the 2008 IEEE International Symposium on Consumer Electronics*, April, 2008, Vilamoura, Portugal. IEEE. 1-4. https://doi.org/10.1109/ISCE.2008.4559473

[2] Chun-Ying Huang, Kuan-Ta Chen, De-Yu Chen, Hwai-Jung Hsu and Cheng-Hsin Hsu. 2014. GamingAnywhere: The first open source cloud gaming system. *ACM Trans. Multimedia Comput. Commun. Appl.* 10, 1s (January 2014), 25. https://doi.org/10.1145/2537855

[3] Da-Jing Zhang-Jian, Chung-Nan Lee, Chun-Ying Huang and Shiann-Rong Kuang. 2009. Power Estimation for Interactive 3D Game Using an Efficient Hierarchical-Based Frame Workload Prediction. In *Proceedings of the Proceedings of 2009 APSIPA Annual Summit and Conference*, 2009, Sapporo, Japan. 208-215

[4] Benedikt Dietrich, Dip Goswami, Samarjit Chakraborty, Apratim Guha and Matthias Gries. 2015. Time Series Characterization of Gaming Workload for Runtime Power Management. *IEEE Transactions on Computers* 64, 1 (January 2015), 260-273. https://doi.org/10.1109/TC.2013.198

[5] H. Li, M. Li and B. Prabhakaran. 2006. Middleware for Streaming 3D Progressive Meshes over Lossy Networks. *ACM Trans. Multimedia Comput. Commun. Appl.* 2, 4 (November 2006). https://doi.org/10.1145/1201730.1201733

[6] Xiaofei Liao, Li Lin, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang and Bo Li. 2016. LiveRender: A Cloud Gaming System Based on Compressed Graphics Streaming. *IEEE/ACM Transactions on Networking* 24, 4 (August 2016), 2128-2139. https://doi.org/10.1109/TNET.2015.2450254

[7] Xiaoming Nan, Xun Guo, Yan Lu, Yifeng He, Ling Guan, Shipeng Li and Baining Guo. 2014. A novel cloud gaming framework using joint video and graphics streaming. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, July, 2014, Chengdu, China IEEE. https://doi.org/10.1109/ICME.2014.6890204
[8] Eduardo Cuervo, Alec Wolman, Landon P. Cox, Kiron Lebeck, Ali Razeen, Madan Musuvathi and Stefan Saroiu. 2015. Kahawai: High-Quality Mobile Gaming Using GPU Offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, 2015. ACM. 121–135
[9] De-Yu Chen and Magda El Zarki. 2019. A Framework for Adaptive Residual Streaming for Single-Player Cloud Gaming. *ACM Trans. Multimedia Comput. Commun. Appl.* 15, 2s (July 2019), 23. https://doi.org/10.1145/3336498
[10] Zhinan Cheng, Xi Li, Beilei Sun, Jiachen Song, Chao Wang and Xuehai Zhou. 2016. Behavior-Aware Integrated CPU-GPU Power Management For Mobile Games. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, September, 2016, London, UK IEEE. 439-444. https://doi.org/10.1109/MASCOTS.2016.18
[11] Jiachen Song, Xi Li, Beilei Sun, Zhinan Cheng, Chao Wang and Xuehai Zhou. 2016. FCM: Towards Fine-Grained GPU Power Management for Closed Source Mobile Games. In *Proceedings of the International Great Lakes Symposium on VLSI*, May, 2016, Boston, MA, USA. IEEE. 353-356. https://doi.org/10.1145/2902961.2902989
[12] Benedikt Dietrich, Swaroop Nunna, Dip Goswami, Samarjit Chakraborty and Matthias Gries. 2010. LMS-based Low-Complexity Game Workload Prediction for DVFS. In *Proceedings of the IEEE International Conference on Computer Design*, 2010, Amsterdam, Netherlands. 417-424. https://doi.org/10.1109/ICCD.2010.5647675
[13] Michael Wimmer and Peter Wonka. 2003. Rendering Time Estimation for Real-Time Rendering. In *Proceedings of the Eurographics Symposium on Rendering*, 2003, Leuven, Belgium. Eurographics Association. 118-129
[14] Pietro Mercati, Raid Ayoub, Michael Kishinevsky, Eric Samson, Marc Beuchat, Francesco Paterna and Tajana Šimunić Rosing. 2017. Multi-variable dynamic power management for the GPU subsystem. In *Proceedings of the Design Automation Conference*, June, 2017, Austin, TX, USA IEEE. 1-6. https://doi.org/10.1145/3061639.3062288
[15] Paul Varcholik *Real-Time 3D Rendering with DirectX and HLSL*. Addison-Wesley Professional, 2014.
[16] Jason Zink, Matt Pettineo and Jack Hoxley *Practical Rendering and Computation with Direct3D 11*. CRC Press, 2016.
[17] Matthias Wloka. 2003. Batch, Batch, Batch: What Does It Really Mean? In *Proceedings of the Game Developers Conference*, 2003. Nvidia
[18] *DirectX Graphics and Gaming | Microsoft Docs*. 2018.
[19] Bernard L. Peuto and Leonard J. Shustek. 1977. An Instruction Timing Model of CPU Performance. In *Proceedings of the International Symposium on Computer Architecture*, 1977. ACM
[20] Bernhard Kerbl, Michael Kenzel, Elena Ivanchenko, Dieter Schmalstieg and Markus Steinberger. 2018. Revisiting The Vertex Cache: Understanding and Optimizing Vertex Processing on the modern GPU. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2 (August 2018), 1-16. https://doi.org/10.1145/3233302
[21] *APITrace*. 2019.
[22] *Ego (game engine)*. Codemasters.
[23] *Unreal Engine*. Epic Games.