A Workload Prediction Model for 3D Textured Meshes in Webgl Context

Gabriel Meynet, Jean Claude Iehl, Johanna Delanoy, Guillaume Lavoué, Florent Dupont

A Workload Prediction Model for 3D Textured Meshes in Webgl Context

Gabriel Meynet
Speedernet, Universite Claude Bernard Lyon 1, CNRS, INSA Lyon, LIRIS, UMR5205
69009 Lyon, France
gabriel.meynet@liris.cnrs.fr

Jean-Claude Iehl
Universite Claude Bernard Lyon 1, CNRS, INSA Lyon, LIRIS, UMR5205
F-69622 Villeurbanne, France
jean-claude.iehl@liris.cnrs.fr

Johanna Delanoy
INSA Lyon, CNRS, Universite Claude Bernard Lyon 1, LIRIS, UMR5205
F-69621 Villeurbanne, France
johanna.delanoy@liris.cnrs.fr

Guillaume Lavoué
Centrale Lyon, CNRS, INSA Lyon, LIRIS, UMR5205, ENISE
F-42023 Saint Etienne, France
guillaume.lavoue@liris.cnrs.fr

Florent Dupont
Universite Claude Bernard Lyon 1, CNRS, INSA Lyon, LIRIS, UMR5205
F-69622 Villeurbanne, France
florent.dupont@liris.cnrs.fr

ABSTRACT

The vast majority of simplification algorithms are based roughly on the assumption that rendering time is related to the number of primitives, with the aim of reducing memory impact and rendering complexity. In this paper, we define more precisely the links between 3D object intrinsic characteristics and rendering time in order to provide a new tool for prediction and to guide these simplification methods. We conduct a large-scale experiment in a WebGL environment on multiple devices to measure the rendering time of a set of photo-reconstructed and textured 3D meshes. The results showed us the influence of features on rendering time and that the number of vertices is not the most relevant characteristic. We then trained a predictor capable of predicting the rendering performance of a 3D mesh. This predictor takes as input various characteristics of 3D objects as well as a set of device rendering performance features that we have introduced and achieves a prediction accuracy of 1.16 ± 0.09 ms on average (19.70 ± 2.44 % relative error). We also provide an analysis of the most important characteristics for the task of prediction.

CCS CONCEPTS

• **Computing methodologies** → **Classification and regression trees**; Rendering; *Graphics processors*; **Mesh models**.

KEYWORDS

Workload Prediction, Rendering Time Estimation, Triangular Mesh, Random Forest, Mesh Analysis, Mesh Features, WebGL

1 INTRODUCTION

In recent years, advancements in technology have significantly improved raw computing power, rendering techniques like geometry virtualization, and data acquisition through photogrammetry. These improvements, along with better modeling tools, have made 3D graphics content much more detailed and rich, opening up possibilities for creating diverse 3D environments. Technological improvements have also been made in the area of browser-based graphics rendering using WebGL. This growth is due to its ease of use and compatibility with a wide range of devices. WebGL is used in various research fields, including medicine [Min et al. 2018], cultural heritage [Gaspari et al. 2023], and urban planning [Gautier et al. 2023; Vincent Jaillot and Gesquière 2020], among others. It is also a subject of great interest to the contributors involved in the development of the metaverse, as this technology is promising and used in several projects [Choi et al. 2022; Gadea et al. 2016]. WebGL plays a crucial role in the metaverse by delivering immersive 3D graphics directly to web browsers. Its accessibility and cost-effectiveness make it a catalyst for mass adoption.

The task of displaying such complex data in real-time is still a challenge due to the size of the input data and the limited memory on most portable devices and low-power machines. A preprocessing step is often required and it consists of reducing the complexity of the shape and/or the texture information through simplification while preserving an overall shape [DGG 2024; Kim et al. 2002; Lee and Kyung 2016; Potamias et al. 2022]. When dealing with very complex objects, this task is often carried out manually by an artist [Autodesk 2024; EpicGames 2024] as most algorithms give poor results. Another approach to visualizing massive 3D meshes is to use spatial data structures to speed up culling operations [Stein et al. 2014], but these are limited to static scenes or come with a non-negligible update cost.

One flaw of the simplification pipeline is the lack of tools to precisely predict at which level a 3D object becomes displayable for a given target device in a web context. The number of primitives (faces or vertices) is often used for guiding the simplification task but, as illustrated in Figure 1, we show that in some cases this information is not enough to predict the displayability i.e. if the number of Frames Per Second is above a given threshold. We use the standard 30 FPS established for gaming applications as the lower bound for displayability. For immersive or XR applications, this threshold can be raised to 60, 120, or even 240 with certain devices.

In this context, we propose a new offline model of "Mesh-Workload" capable of predicting if a 3D object, for a given target device, will be displayable in real-time using a set of geometrical, texture layout, and data ordering features coupled with descriptors related to the devices' rendering performances. Initially, we have chosen to establish our method using dedicated graphics cards, and we reserve the study of portable devices for future work.

This tool paves the way for smarter simplification algorithms whose objective will not simply be to reduce the number of primitives but to optimize a set of parameters to render the object above an FPS threshold while preserving its visual quality. It could also be included in mesh editing software to provide feedback about the rendering complexity for a given hardware target.

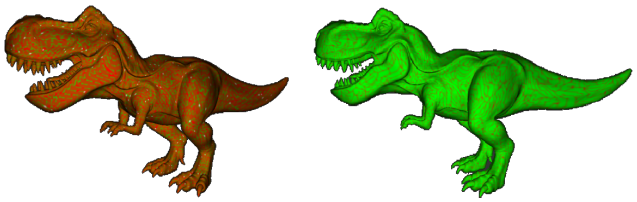

Figure 1: Impact of vertex ordering on the rendering time: Two 3D meshes with the same number of primitives organized in different orders. The color shows the number of transformations per vertex, green is one transformation, and red is six. The left object takes 400 μs to render, the right object 250 μs and the worst case with a fully random order of vertices is even slower: 600 μs.

In summary, our contributions are the following:

(1) We designed a measurement protocol to collect rendering times of 3D objects in a WebGL framework.
(2) We propose a new way to extract characteristics of the rendering capabilities of a physical device.
(3) We train and provide an optimized predictive random forest model and we show that our model can generalize to new hardware.

The rest of this paper is organized as follows. The next Section reviews the related works in the domain of performance prediction. Section 3 presents an overview of our method. Section 4 describes the dataset we use, how we augment it, and the set of features that we compute on 3D meshes and devices. Details about the rendering environment and the measurement protocol are presented in Section 5. Our model is presented in Section 6 along with our feature selection process and hyperparameters. We present and evaluate our results in Section 7. Concluding remarks and perspectives are given in Section 8.

2 RELATED WORK

The field of performance prediction is vast and is highly coveted in areas such as task parallelization or memory consumption prediction in high-performance computing centers, but also in the field of computer rendering, where the main interest is in predicting rendering time for a wide range of applications.

One area of research is specialized in the prediction of workload for mobile devices (mostly smartphones). These predictions are used by a power saving method called DVFS [Dietrich et al. 2013; Macken et al. 1990; Pathania et al. 2015a,b; Zhang-Jian et al. 2009] for Dynamic Voltage Frequency Scaling. Estimating the workload beforehand allows proper management of the power and resources to improve battery life. As an example, a hybrid workload prediction scheme with a switch between a PID-based and a frame structure-based predictor is proposed [Gu and Chakraborty 2008]. The authors found a strong linear correlation between rasterization and total workload, hence the emphasis on this prediction.

Another domain of application where the performance prediction is used is in High-Performance Computing (HPC) architectures. A genetic algorithm approach [Tikir et al. 2007] is derived from an analytical formula as a way to predict the bandwidth from cache hit rates in memory-bound applications. In the same field of application, a regression-based approach to predict parallel program scalability is proposed [Barnes et al. 2008].

For real-time rendering applications, a frame processing time estimation model using the frequency of integrated GPUs is proposed [Gupta et al. 2016] but this is not applicable in a web context with limited access to hardware measurements. A signature-based predictor for estimating the rendering time of a 3D graphic scene is proposed [Mochocki et al. 2006]. This method performs at runtime and uses triangle information such as the average area, count, and height as well as vertex count as a signature at each frame. A distance is computed with each signature recorded thus allowing the model to give a precise estimation of the rendering time. This method is simple and compact but requires modification of the rendering pipeline which is not desired in our case.

An analytical model [Wimmer and Wonka 2003] is proposed for predicting the rendering time using the number of transformed vertices coupled with the number of projected pixels given a viewpoint and a list of potentially visible objects. The authors claim their method can only provide an upper bound due to time-stamping limitations and proposed two hardware modifications to solve this issue.

As for more recent work, an API-level workload model for modern video games is proposed (Gamorra) [Mohammadi et al. 2022]. They used a multi-linear regression model with a hybrid training scheme (online/offline) as a method for computing weights. Benchmarking each step of the rendering pipeline improved greatly their accuracy. Unfortunately, this method requires access to the commands sent at the Application Programming Interface level (API) which is not an easy task in the case of a WebGL pipeline. They need to train their model beforehand with frames from the game to find a proper weight set and this is not applicable in our context as our model performs offline.

Our work differs from most of these methods in that it can be described as non-intrusive as we do not perform any pipeline alteration. This also fits perfectly into a web context where low-level GPU information cannot be queried easily. Our method differs from most rendering time predictors by being completely offline and not using any information from the renderer or the graphical API for the prediction. Our context of experimentation is also different because we are working in a rather classical rendering context (static photo-reconstructed 3D models and simple materials) which differs from models working on video game scenes, for example.

Overall, to the best of our knowledge, there are no deep studies about the impact of a 3D object intrinsic's features on its rendering time in a WebGL context, the subject that we tackle in this paper.

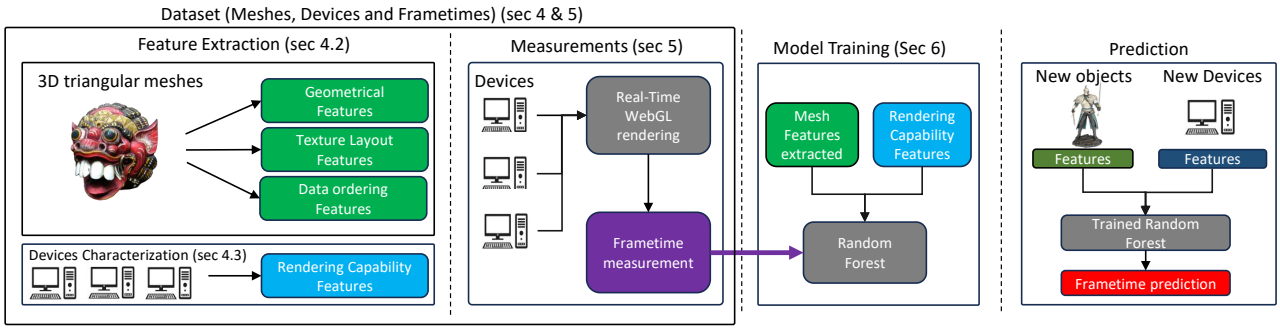

Figure 2: Overview: A set of features is computed for all meshes and all devices from our dataset, We render in a WebGL environment every mesh for every device and measure the corresponding frametime. The resulting data is then used to train a machine-learning model, that can be used to predict the rendering time for any new object and device.

3 METHOD OVERVIEW

Our goal is to predict the rendering time of a 3D textured mesh on various devices. To provide a rendering time, our predictor will take as input the characteristics of the mesh and the device. To train it, we need to gather these characteristics and ground truth rendering times. Our proposed method is composed of three main steps: 3D mesh and device feature extraction, frametime measurements, and the training of predictor (see Figure 2).

Features extraction. The first challenge is to characterize 3D objects and devices' rendering performances. Triangular meshes are a well-known and used representation of a 3D object and there is a wealth of information that can be gathered from analyzing surface properties (such as mean face surface area), the parametrization (such as the length of the boundary edges in the UV space), or even looking at the ordering of vertices in the final rendering buffer (such the estimated number of transformed vertices). We extract a large feature vector from the mesh geometry, parametrization (texture layout), and data ordering presented in table 1. We use a dataset [Maggiordomo et al. 2020] of 568 meshes reconstructed from real photos containing a large variety of shapes and defects that we augment with four simplified versions of each object using state-of-the-art simplification algorithms, resulting in a dataset of more than 3000 objects.

Characterizing a device's rendering capability is an even more complex task due to the fact there is a copious amount of components, brands, and rendering architectures that can have an impact on the rendering performances. We propose a tool to evaluate the rendering capability of devices by measuring the mean rendering speed for two specific tasks (transformation and rasterization) derived into three performance descriptors (see Section 4.3).

Measurements. To accurately predict the frametime we need a large amount of data from a diverse set of devices. We conducted an experiment (described in Section 5) in which we rendered multiple mesh instances in real-time on a user's web browser and extracted frametime values for 14 physical devices.

Frametime Predictors. We use the features and frametime measurements of the 3D models coupled with devices' rendering performance indicators to train a features-based random forest predictor (FTRF for FrameTime Random Forest), that estimates a 3D model's rendering time on a specific device from its intrinsic features. We demonstrate that our model generalizes well on new objects and new devices.

4 FEATURE EXTRACTION

In this section we present the dataset we use for our experimentation and how we augment it to increase diversity. We also provide details about the different sets of features computed on the 3D mesh and the performance descriptors we extract from the rendering device.

Figure 3: Subset of 3D meshes used for the experimentation from the dataset [Maggiordomo et al. 2020]. The numbers associated represent the number of vertices.

4.1 3D triangular mesh dataset

We used the dataset "Real-World textured things" [Maggiordomo et al. 2020] composed of 568 photo-reconstructed 3D meshes (see Figure 3). This dataset contains meshes with a number of vertices ranging from 1K to 4.7M, with the vast majority around 100k. In terms of parametrization, the objects have a number of textures ranging from one up to five for most of them, with a few exceptions at 10. Texture resolutions vary from 1K for the smallest to 8K for

Table 1: Feature list of 3D meshes

Geometrical (12)	Texture Layout (13)	Data Ordering (7)
Face Number, Vertice Number, Edge Number, Bdry Edge Number, Bdry Edge Lgth, Bdry Edge Lgth Normalized, Area, Area Normalized, Nonmanifold Edge Number, Nonmanifold Vertice Number, Unref Vertice Number, Number Connected Components	Texture Number, Tex Pixel Count, Area UV, Edge UV Number, Bdry Edge UV Number, Bdry Lgth UV, Null Charts Number, Fold Number, Charts Number, Occupancy, Mapped Fraction, Crumbliness	Vertices Transformed, Overdraw, Pixels Shaded, Covered Pixels, Overfetch, Bytes Fetched, ATVR

the largest. To enrich and diversify the content of the dataset, we extended it by simplifying each object using two simplification methods, one that only seeks to reduce the number of primitives and one that also optimizes cache access by reordering the list of vertices. These simplification methods are described below.

4.1.1 Quadric Error Metric. QEM [Garland and Heckbert 1997] consists of decimating the mesh using edge collapses while minimizing the error computed and propagated using quadrics. We simplify our meshes at four levels of detail: −50% and −80% of the original face count and we also explicitly set two other decimation targets on the vertice count (150000 and 300000) meaning only meshes above those thresholds are simplified. We use the implemented version from Meshlab-2022.02 [Cignoni et al. 2008].

4.1.2 MeshOptimizer. MeshOptimizer [Kapoulkine 2023] is a tool for the simplification of 3D content used for lightweight hardware. As well as reducing the number of primitives while preserving global appearance (deviation from the original mesh using normalized distance), it can also remove any duplicated vertex which does improve performances. Furthermore, its main feature consists of reorganizing a 3D mesh triangle list to reduce cache operations for the GPU while minimizing vertex transformations. We use MeshOptimizer as a second simplification method and we simplify at two levels of details: −50% and −80% of the original face count coupled with vertex cache optimizations.

4.2 3D Mesh related Features

We selected 32 features that are computed for each 3D mesh using TexMetro [A and P 2019] and MeshOptimizer [Kapoulkine 2023]. In order to better assess their influence on the rendering time we categorize these features into three categories: Geometrical, Texture Layout and Data Ordering features as in Table 1.

4.2.1 Geometry-Based Features. In this category, we gather features related to the geometric information of the 3D meshes. For clarification, features described with a normalization attribute mean they are computed on objects normalized beforehand in a unit cube. Our ensemble starts with the number of primitives describing the object (*Face Number*, *Vertice Number* and *Edge Number*). As our dataset contains nonmanifold meshes, we keep these three features although they are strongly linked for manifold meshes. We also report features that characterize boundary edges (*Bdry Edge Number*, their length *Bdry Edge Lgth*, and a normalized length *Bdry Edge Lgth Normalized*). It is known a large number of subpixel triangles will impact the rasterizer's task thus we sum the area of triangles and we provide a normalized value (*Area* and a *Area Normalized*). Non-manifold elements may require more memory to store and process which can affect rendering performances (*Nonmanifold Edge Number* and *Nonmanifold Vertice Number*). Some models in our dataset also contain non-referenced elements (*Unref Vertices Number*), they are not displayed but they are always loaded, thus increasing the memory required to process the mesh and more specifically in the case where a large quantity of these elements are present. Finally, we report the number of connected components as it could have an impact on the underlying culling procedure (*Number Connected Components*).

4.2.2 Texture layout features. In this category, we selected characteristics related to the parametrization of a mesh that can have an impact on rendering time. We report the number of textures (*Textures Number*) associated with a mesh and the sum of pixels of all textures from a mesh (*Tex Pixel Count*). We also collect information about the parametrization in the UV space such as the area of triangles (*Area UV*). We report the number of primitives (*Edge UV Number* and *Bdry Edge UV Number*) with their length (*Bdry Lgth UV*) as it is known in the literature it could have an impact on the rendering time due to duplication of vertices along a texture seam [Maggiordomo et al. 2020]. The layout of the texture components is another factor that can also impact the rendering time, the number of texture charts, those folded and with a zero surface area are also reported (*Null Charts Number*, *Fold Number Charts Number*). Their presence in large quantities means the texture might contain more boundaries and increase the sampling workload. We also provide a descriptor measuring the ratio of every texel that falls inside UV triangles over the total number of texels (*Occupancy*) and the percentage of surface area that is mapped to a non-zero texture area (*Mapped Fraction*). Finally, we selected two statistical descriptors that are invariant to the object size, they measure how fragmented a texture is (*Crumbliness* and *Solidity*). The reader can refer to the work of Maggiordomo et al. [Maggiordomo et al. 2020] for more details about these features and how they are computed.

4.2.3 Data Ordering features. In this category, we include all descriptors provided by MeshOptimizer, that relate to the ordering of the data and the efficiency of the task of rendering.

Vertices Transformed is the number of transformations operated on the vertices buffer. It ranges from the number of vertices (for a ratio of one transformation per vertex in the best case) to up to six times the number of vertices.

Overdraw is a metric computed from a set of orthographic cameras placed around the object. It measures the average over the views of a ratio between the number of effective pixels shaded (*Pixels Shaded*) to the total number of pixel shader invocations (*Covered Pixels*).

Overfetch is the ratio between the number of bytes fetched in memory (*Bytes Fetched*) over the total number of bytes, a large number of memory accesses could result in a lower frametime.

ATVR(Average Transformed Vertex Ratio) is the number of transformed vertices over the total vertices. This descriptor allows us to better assess the rendering efficiency of a mesh.

4.3 Devices related Features

With the tremendous number of different rendering-related hardware such as brands with Nvidia, AMD, Qualcomm or Intel and their specific hardware optimizations, characterizing rendering performance becomes a real challenge. We considered using 3Dmark [Solutions 2023] score as a measure of performance but we faced two problems: First, score values are tied to specific screen resolutions and hardware (CPU/RAM, etc), which may not match the specification of the device; Second, there is no available open database for us to acquire.

We thus designed a tool to characterize the performance of a given machine. Inspired by Gamorra [Mohammadi et al. 2022], we perform several rendering tests, each measuring the performance of a stage of the rendering pipeline. The first test measures the vertex transformation rate and the second one stresses the rasterizer stage and fragment shading stage of the device. Both tests transform the same number of vertices, rasterize, and then shade the same number of fragments for a fixed screen resolution. These tests draw several full screen tessellated quads with a constant total number of vertices. To obtain our final measure, we apply a temporal median filter to reduce the noise and keep the median value over 10,000 rendered frames.

Transformation test. We draw four configurations with 65536, 262144, 1048576 and 4194304 total vertices. We report in Table 2 the distribution of the mean rendering time of 14 devices computed on our dataset along with the performance indicator *Transform 65536*. We can see that this indicator accurately discriminates the rendering speed of slower devices and characterizes well the underlying rendering speed.

Rasterizing test. We draw nine configurations with tessellation factors 4, 8, 16, 32, 64, 128, 256, 512 and 1024, while adjusting the number of tessellated quads to transform a constant total number of vertices. Each configuration draws each full screen quad as a grid of smaller quads, eg the first test draws a full screen quad as a 4x4 grid of smaller quads. We also adjust the number of grids to draw to transform a constant number of vertices. This test varies the on-screen size of the quads but transforms the same number of vertices and shades the same number of fragments.

Shading test. Our objects use the same simple textured shading model. The shading cost is thus constant in this case and we chose to not measure fragment shading performance.

In addition to these measures, information such as the Operating System and the screen resolution (in pixels) used for the experimentation are gathered and then fed to our predictor as explanatory variables. See Table 2 for the specification of the devices used in this experimentation.

5 FRAMETIME MEASUREMENT

To train our model we asked 12 participants to conduct an experiment that consisted in rendering 3D objects in a 3D WebGL environment and measuring the frametime on multiple sets of devices and configurations. In Section 5.1 we present our rendering context, the object loading procedure of the experimentation, and how we measure the frametime in Section 5.2, and technical details are given in Appendix A.

5.1 Web Context

We opted to conduct this experiment utilizing a WebGL rendering engine due to several compelling reasons. The need for additional analysis tools for applications based on WebGL technologies stems from two primary considerations. Firstly, there are works from multiple research fields that are constructed using WebGL technology, highlighting the practical relevance and demand for advancements in this domain. Secondly, this project has the potential to stimulate the creation of new solutions designed for devices with limited performance. By addressing these imperatives, our research aims to not only enhance existing applications but also pave the way for more efficient and adaptable solutions, particularly in the field of WebGL technology.

As most of the latest versions of web browsers (Chrome, Firefox) support WebGL rendering, we conducted preliminary experiments to assess their stability, robustness, and lightness. We chose Google Chrome as it provides the most consistency in our measurements. For the choice of the WebGL framework, we chose to use the *Three.js* [Cabello 2024] framework due to its stability and flexibility.

Figure 4: Screenshot of the experiment and visualization of the grid layout - This state corresponds to the loading procedure represented by a yellow cross in Figure 5

5.2 Loading procedure

To save power, most GPUs adjust their frequency depending on the difficulty of the rendering task, but in our case, we need to push the GPU to its full capacity to get accurate measurements. To achieve

Table 2: Overview of the devices used in our experiment - Screen resolutions are given in the standard format

Device ID	GPU	VRAM	CPU	OS	Resolution	Mean Frametime (ms)	Transform 65536[†]
device_14	RTX 3090	24 GB	Xeon W-2245	Windows	WQHD	3.9	10.24
device_6	RTX 3090	24 GB	Ryzen 9 5950X	Windows	UW4K	4.1	10.24
device_10	GTX 1080 Ti	11 GB	Core i9-12900K	Windows	WQHD	3.9	11.26
device_13	RTX 3080	10 GB	Ryzen 9 5950X	Windows	FHD	3.5	12.29
device_12	RTX 3080	10 GB	Ryzen 9 5950X	Windows	WXGA HD	3.8	12.29
device_1	RTX 3080	10 GB	Xeon Silver 4208	Linux	WQHD	4.8	12.29
device_5	RTX 2080 Ti	11 GB	Xeon E5-1660 v2	Windows	UWQHD	4.2	12.58
device_4	TITAN RTX	24 GB	Core i9-10900K	Windows	FHD	4.3	13.95
device_8	RTX 3070	8 GB	Core i9-10900K	Windows	WQHD	4.6	15.36
device_3	RTX 2080	8 GB	Ryzen 7 1700X	Linux	WSXGA+	5.7	19.90
device_7	RTX 2060	6 GB	Core i9-10900K	Windows	WQHD	6.7	24.58
device_9	GTX 1080 Ti	11 GB	Core i7-6700K	Windows	WQHD	7.9	34.82
device_2	GTX 1660	6 GB	Ryzen 5 3600	Linux	FHD	10.5	39.04
device_11	GTX 1050 Ti	4 GB	Core i5-9400	Windows	FHD	12.6	82.94

[†] This table is sorted by the transform indicator, lower is better, device ID corresponds to the i-th device tested

maximum GPU usage, we load a high number of instances on the screen to simulate a complex scene. The object and their respective instances are loaded consecutively on a 3D regular grid layout of size 4*4*4 accounting for 64 instances of the object in total (see Figure 4). They are pre-normalized into a unit cube of unit size to ensure consistency between objects, their center is recomputed at the center of the unit cube, and their position is fixed with a distance of one unit plus a small epsilon to avoid any overlap. Their orientations are uniformly sampled to avoid any bias caused by the fact that the camera is looking at them from a fixed point of view. We used a subset of the previously introduced dataset so as not to exceed six hours of experimentation per user. For the object selection, our preliminary tests revealed that rendering time is strongly linked to vertex transformation time, so we sampled the initial dataset on this dimension for 600 meshes. For each mesh, we load and try to display it then we check every 200 ms for the reception of the callback from the loader as certain objects are over one gigabyte in size. Then we create instances every 500 ms until the grid is full. Once all the instances have been loaded, we wait for 10 seconds during which we measure the rendering time. After that time, we clean the rendering environment and wait six seconds before loading the next object (see Fig.5).

Collecting the frametime precisely is a particularly complex task because, for security reasons, Web browsers do not allow any direct communication between the client and the GPU. We considered using the exposed "EXT_disjoint_timer" extension but this is not supported on all operating systems. Instead, we query Frames Per Second (FPS) at an interval of one second and we collect the frametime value as the inverse of the FPS. Frametime is logged every second for the whole duration of the experiment, we process raw data afterward using timestamps generated at the beginning and the end of the 10-second measurement window. You can observe the distribution of the Frames per Second measured for 14 devices in Figure 6.

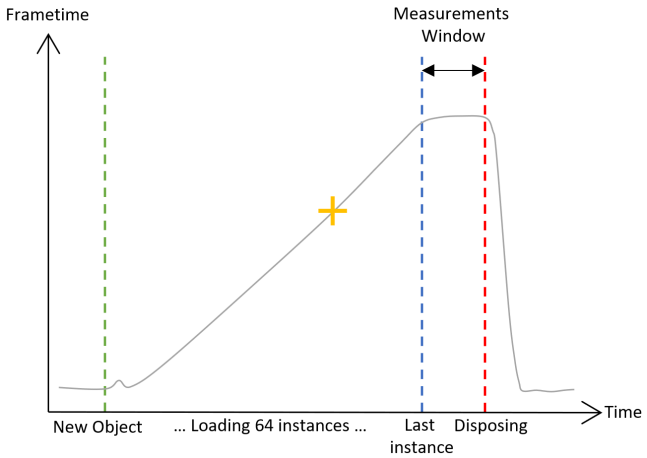

Figure 5: Frametime measurement - We first load the object, then its instances as we observe the frametime increasing. Only frametimes measured within the measurement window are considered stable. Following the dispose call, the environment is cleaned and ready for the next object

6 FRAMETIME PREDICTOR

To predict the rendering time of a 3D object based on its intrinsic characteristics, we propose a regression model (*FTRF*) based on an ensemble of decision trees (Random Forest) [Breiman 2001] that we detail in Section 6.2. We perform a few upstream operations on the data collected that we discuss in Section 6.1.

6.1 Pre-processing

Our dataset contains only one categorical feature *Operating System* with two outcomes: "Windows or Linux". As the random forest cannot process textual data we use a One Hot Encoding on these labels.

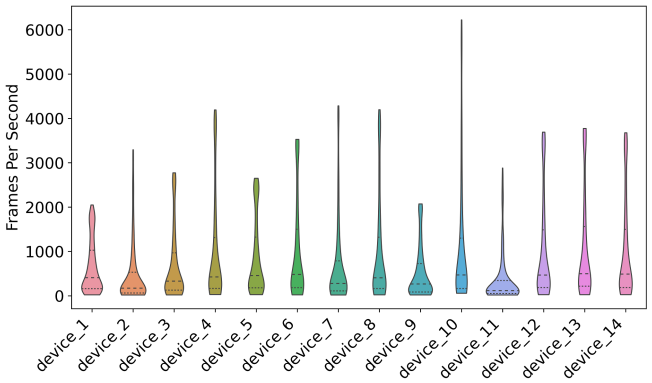

Figure 6: Frames per Second distribution for each device - On the fastest devices, some objects are rendered at more than 6000 FPS while most of the values are under 1000 FPS

Despite collecting the median frametime over more than 10 seconds, for a few sets of massive 3D meshes with multiple textures we observe discrepancies in the frametime measured caused by web-browser memory limitation. After a histogram analysis of the frametime distribution, we observe that approximately 2% of the measurements are outliers and consequently removed by a quantile filter.

6.2 Random Forest Regressor

We choose to use a random forest regressor rather than deep-learning methods for the task of prediction because the latter is less effective when working on tabular data with a relatively small number of samples as in our case (<10000) [Borisov et al. 2022; Grinsztajn et al. 2022]. With its feature importance mechanism and the interpretability of decision trees, the random forest is considered to be more explainable than deep learning models. We then propose a cross-validation scheme (see Section 6.2.1) where neither a tested 3D mesh nor a tested rendering device is seen in the training fold, this allows us to demonstrate our model capabilities to generalize. To compare our model we also train two models for comparison with only unseen meshes and devices respectively FTRF_meshs and FTRF_dev.

6.2.1 Cross-validation objects and devices splits - FTRF. We train our model FTRF using a three-fold cross-validation scheme where we split our data along two dimensions at the same time, one along the objects and one along the devices. Each fold contains one-third of meshes rendered on one-third of our devices for our test set while our training set contains the two-thirds remaining for both dimensions, leading approximately to 80%:20% train-test split (see Figure 7). We report the results as the average over the three folds. We fine-tune our model using a grid search method and we report the best parameter for the model that minimizes the L1 norm - Mean Absolute Error (MAE) $\sum(y_i - \hat{y_i})$ over the folds where y_i are the ground truths and $\hat{y_i}$ the predictions. We set the number of estimators to 30, and the number of features per split is set to 50% of our total dataset feature number. This parameter value allows us to take full advantage of the random mechanism provided by the random forest and reduces overfitting. Finally, the maximum depth of estimators is set to 16.

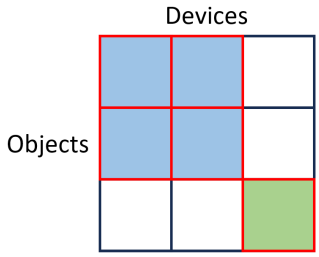

Figure 7: Cross-validation: We represent in blue the training split and in green the test split over one fold in red. For a total of five blocks, the training split accounts for 80% of the fold data and the test split for 20%.

6.2.2 Model variations for analysis - FTRF_meshs and FTRF_dev. For comparison and to illustrate the generalization capabilities of our method, we also train two models using three-fold 80%:20% train-test cross-validation schemes where we split our data along specific dimensions. For FTRF_meshes, neither a 3D object nor its simplified versions seen during training are used for prediction. For FTRF_dev, we make sure that every measurement for each device belongs to either a training or a testing set. We fine-tune our models using a grid search method and we report the best parameter for the model that minimizes the MAE over the folds. We set the number of estimators to 90, and the number of features per split is set to 60% of our total dataset feature number. Finally, the maximum depth of estimators is set to 16.

7 IMPLEMENTATION, RESULTS AND EVALUATION

Our random forest regressor was implemented in *Python* with the *SciKitLearn* framework. Our model training was performed on a desktop computer with an AMD Ryzen 9 5950X 16-core Processor and 32GB RAM.

7.1 Metrics

We attach particular importance to the choice of metrics for evaluating our model as our predictions cover multiple orders of magnitude.

The *Pearson* and *Spearman* correlations are calculated between the predictions and the ground truth. The Pearson correlation is the simplest tool for measuring the strength and direction of the linear relationship, while the Spearman correlation is used to assess the extent to which the rank of our predictions is consistent with the rank of ground truth measurements.

The coefficient of determination R^2 is another indicator used to determine the prediction performance of the model. The R^2 coefficient is calculated as follows considering $\hat{y}$ the predictions, y the corresponding ground truths and $\bar{y}$ the mean over the predictions :

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y_i})^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \quad (1)$$

Table 3: Performance of different models

model	$Corr_p$	$Corr_s$	R^2	MAE (ms)	RMSE (ms)	Emax (ms)	MAPE (%)
Median (cst)	n/a[1]	n/a[1]	0.12 ± 0.05	4.2 ± 0.12	7.24 ± 0.26	46.05 ± 1.01	204.8 ± 54.14
LM	0.76 ± 0.10	0.88 ± 0.01	0.43 ± 0.25	2.5 ± 0.3	4.93 ± 0.87	37.5 ± 6.4	188.1 ± 67.22
SVR	0.90 ± 0.08	0.96 ± 0.01	0.79 ± 0.16	**1.11 ± 0.23**	2.79 ± 1.17	37.3 ± 24.6	26.49 ± 5.25
FTRF (ours)	**0.93 ± 0.009**	**0.98 ± 0.002**	**0.85 ± 0.003**	1.16 ± 0.09	**2.61 ± 0.18**	**22.12 ± 2.31**	**19.70 ± 2.44**
FTRF_VN†	0.80 ± 0.03	0.90 ± 0.01	0.63 ± 0.06	2.08 ± 0.04	4.12 ± 0.26	34.33 ± 6.42	50.32 ± 4.30

[1] Due to the constant nature of the predictions of the Median regressor model, $Corr_p$ and $Corr_s$ could not be reported
† FTRF_VN is a Random Forest model trained with devices features and Vertices Number as explanatory variable

We report performances of an L1 norm as the Mean Absolute Error (*MAE*), an L2 norm as the Root Mean Square Error (*RMSE*) along with Emax standing for the l-infinity norm. We also compute our metrics on specific FPS ranges, by splitting our dataset according to the measured frametime (See columns 3 to 7 in Table 4). These Frames Per Second (FPS) intervals correspond to different quality targets in a real-time environment : [1:30],]30:60],]60:120],]120:1000],]1000:inf[respectively: Poor, Adequate, Smooth, Fluent, and Exceptional.

7.2 Results

We first compare our proposed model with other classic machine learning models and show that it performs better by analyzing correlations and errors. We also report results from another frametime prediction model for comparison. Finally, we demonstrate FTRF's capability to generalize to new objects and devices and we also report results from two variations of our model that were trained with other configurations described in 6.2.2.

7.2.1 Comparison against others models. We tested the performances of a constant regressor predicting the median, a Linear Model (LM), and a non-linear model: Support Vector Regressor (SVR) all trained with the same procedure as our predictor FTRF (See Table 3). There is a pre-processing step for the SVR training, the data and the target frametime have been normalized and we transform back the predictions to compute our metrics. We quantify this with the metrics we described previously (see Section 7.1) averaged over three folds. Results show that our model performs better in terms of Pearson and Spearman coefficients and R2 respectively $Corr_p \approx 0.93$, $Corr_s \approx 0.98$ and $R^2 \approx 0.85$ with relatively small deviations demonstrating the stability of our model. It achieves a strong linear correlation between predictions and ground-truth while preserving the ranking order of frametimes. Results show that on average, our model produces the smallest Emax (≈ 22.12 ms), L1 and L2 errors for our model are four times smaller than a constant predictor and twice as small as a linear model. Results from the SVR model show a relatively close performance to our model but with higher deviation indicating a lack of stability over the folds. In terms of relative error MAPE, FTRF achieves the lowest score with a value of ≈ 19.70%. These performances comfort us in the choice of a random forest model for predicting the rendering time.

To demonstrate that the number of vertices alone might not be the best feature for the task of predicting the rendering time, we compare our model to one (FTRF_VN) trained with device performance-related features and only vertex number as an explanatory variable. The result shows that using all mesh descriptors allows our model to increase the Pearson correlation by ≈ 0.13 and the L1 MAE is approximately divided by a factor of two.

Finally, our work operates in a different experimental context to that of other methods that work at API level or on the content of a rendered frame, which makes comparison more complicated. Nevertheless, as a qualitative comparison, and although there is no open-source code available, we report the performance of the Gamorra model [Mohammadi et al. 2022]. We compare the results of their offline (no adjustments in real-time) model over their dataset to FTRF_meshs as it falls closer to this training scheme where we predict only on a new set of meshes. Their model has on average an estimation error of 1.55 ms with 13.8% relative error and FTRF_meshs still fall closely with a value of 0.73 ms with 11.7% relative error thus showing that our model can perform as well as recent predictors.

7.2.2 Frametime predictor FTRF. Along with the results presented previously, we also report quantitative results in Table 4 as the average performance of the cross-validated predictors on their respective test sets (prediction on new objects for FTRF_meshs; new devices for FTRF_dev and on both for FTRF). Our variation model FTRF_meshs achieves sub-millisecond error with an MAE value of 0.73 ms on average demonstrating great accuracy, with the highest Pearson correlation coefficient of 0.97. FTRF_dev has lower scores than the other variant as there is an increased complexity of the task of predicting on another device, but also because the number of devices in our dataset is lower compared to the number of objects. Despite having a complexified training process, FTRF manages to obtain relatively good scores in terms of correlation and error, thus showing a great generalization performance.

By looking at the FPS range [1:30], we see that the three models have the lowest correlation scores and the highest error. This is mostly due to the low number of measurements used for training but when we look at the maximum error Emax for FTRF, the prediction of an object measured at 15 FPS will fall between 11 FPS and 22 FPS in the worst case. In the FPS range]120:1000], for an object measured above 120+ FPS, the prediction falls at 98 FPS in the worst case. This is still enough to determine if an object will be displayable (rendered above 30 FPS) or not on a specific device.

Table 4: FTRF performances: Ranges in FPS - Numbers in parenthesis correspond to the number of measurements

	metric	global	[1:30] (102)	]30:60] (446)	]60:120] (955)	]120:1000] (3614)	]1000:inf[(1860)
FTRF	$Corr_p$	0.93 ± 0.01	0.46 ± 0.1	0.54 ± 0.04	0.54 ± 0.09	0.91 ± 0.02	0.81 ± 0.02
	MAE (ms)	1.16 ± 0.09	15.68 ± 3.23	5.70 ± 0.29	2.35 ± 0.48	0.70 ± 0.09	0.10 ± 0.01
	RMSE (ms)	2.61 ± 0.18	16.42 ± 2.78	6.77 ± 0.38	3.07 ± 0.69	1.05 ± 0.16	0.16 ± 0.01
	Emax (ms)	**22.12 ± 2.31**	22.12 ± 2.31	16.21 ± 4.04	9.87 ± 2.58	5.17 ± 1.66	0.83 ± 0.38
	MAPE (%)	19.70 ± 2.44	37.16 ± 7.35	24.79 ± 0.40	20.24 ± 4.17	19.08 ± 3.18	19.28 ± 2.50
FTRF_meshs	$Corr_p$	**0.97 ± 0.002**	0.51 ± 0.07	0.70 ± 0.08	0.70 ± 0.1	0.92 ± 0.01	0.92 ± 0.03
	MAE (ms)	**0.73 ± 0.04**	6.69 ± 0.86	2.82 ± 0.21	1.44 ± 0.17	0.45 ± 0.04	0.06 ± 0.005
	RMSE (ms)	**1.81 ± 0.10**	8.66 ± 0.91	3.98 ± 0.55	2.19 ± 0.48	0.93 ± 0.09	0.11 ± 0.03
	Emax (ms)	25.06 ± 4.32	20.66 ± 2.62	16.54 ± 3.95	14.71 ± 11.39	10.79 ± 0.79	0.89 ± 0.50
	MAPE (%)	**11.67 ± 0.51**	16.04 ± 2.01	12.60 ± 1.03	12.49 ± 1.56	11.97 ± 1.20	10.42 ± 0.89
FTRF_dev	$Corr_p$	0.95 ± 0.011	0.11 ± 0.29	0.58 ± 0.04	0.56 ± 0.11	0.92 ± 0.02	0.90 ± 0.03
	MAE (ms)	1.05 ± 0.22	12.38 ± 2.76	5.39 ± 0.81	1.97 ± 0.26	0.62 ± 0.11	0.08 ± 0.02
	RMSE (ms)	2.43 ± 0.43	14.25 ± 2.90	6.54 ± 0.74	2.61 ± 0.29	0.97 ± 0.15	0.10 ± 0.02
	Emax (ms)	24.85 ± 3.17	24.44 ± 3.75	18.83 ± 1.71	10.05 ± 0.52	6.15 ± 1.30	0.42 ± 0.04
	MAPE (%)	16.72 ± 2.59	29.26 ± 5.96	23.77 ± 3.58	16.82 ± 2.10	16.68 ± 2.60	14.85 ± 3.07

8 CONCLUSION, LIMITATIONS AND FUTURE WORKS

We used a 3D photo-reconstructed mesh dataset that we extended with different levels of detail using simplification algorithms from the literature. We defined a protocol for measuring the frametime of 3D triangular meshes in a WebGL real-time rendering context. We developed a tool to measure the rendering capabilities of physical rendering devices and we derived these values into descriptors. From all these descriptors and measurements we created a dataset that we used to train an offline random forest regression model FTRF for predicting the rendering time of 3D photo-reconstructed meshes in a WebGL context. FTRF is fine-tuned to predict the frametime of new objects on a set of unknown devices using our new performance descriptors with an average estimation error of 1.16 ms and 20 % of relative error. this last value might appear relatively high. However, it's worth considering that most of our measurements are conducted in a WebGL environment with less timestamp precision than other methods used in the literature. In addition, our predictions are performed on devices never observed beforehand by the model (cross-validated) which is a complex task not tackled previously using only a few performance descriptors and considering the renderer as a black box. Despite this, our analysis model FTRF_meshes can reach a sub-millisecond accuracy of 0.73 *ms* and 12% of relative error which is relatively close to results from models in the literature performing real-time predictions along with autotuning features and real-time corrections (online models). We observed that data ordering features, along with performance descriptors, are predominant in the prediction of frametime, while texture layout features have the least impact but are necessary to achieve the best prediction performance (see Appendix B for more details).

Limitations and future works. The dataset used to train the predictor is composed of 3D models reconstructed from real objects using photogrammetry methods. This could differ from typically handcrafted or CAD generated models. Our predictor could greatly benefit from extending its training on other datasets containing more diverse 3D mesh types and rendering methods.

Disabling the vertical synchronization was mandatory to have a free FPS upper bound instead of being capped to the monitor refresh rate. Due to limited browser memory, we instantiated meshes instead of loading them multiple times, especially for objects with multiple 8K textures that could otherwise not be loaded in a web browser. The growing importance of web rendering technologies, especially WebGPU, could potentially remove certain limitations associated with measurements.

For our future work, we plan to expand our experimentation to include more configuration types running WebGL content e.g., VR headsets and smartphones. Although this method is designed for WebGL, it could easily be extended to Vulkan or DirectX.

We are exploring the potential applications that this model could have. For example, we can derive our predictor into a new approach for guiding simplification and filtering algorithms while preserving quality attributes, leading to less energy consumption. Our predictor could be used in artist modeling tools to predict if they can add more details to their scene for a target device with instant feedback. Finally, in the context of streaming 3D assets to a client for rendering, our model could predict an adequate level of detail by just using the object's precomputed descriptors sent over the network.

ACKNOWLEDGMENTS

The authors acknowledge support from Speedernet, and Association Nationale de la Recherche et de la Technologie, PhD grant n°2021/0473.

REFERENCES

Maggiordomo A and Cignoni P. 2019. TexMetro. https://github.com/cnr-isti-vclab/texmetro/.

Autodesk. 2024. Maya. http://www.autodesk.com/maya.

Bradley J Barnes, Barry Rountree, David K Lowenthal, Jaxk Reeves, Bronis De Supinski, and Martin Schulz. 2008. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*. 368–377.

Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, and Gjergji Kasneci. 2022. Deep neural networks and tabular data: A survey. *IEEE*

Transactions on Neural Networks and Learning Systems (2022).
Leo Breiman. 2001. Random forests. *Machine learning* 45 (2001), 5–32.
Ricardo Cabello. 2024. three.js - A JavaScript 3D library. https://threejs.org/.
Sangsu Choi, Kajoong Yoon, Miae Kim, Jintak Yoo, Bonghyeon Lee, Inho Song, and Jungyub Woo. 2022. Building Korean DMZ metaverse using a web-based metaverse platform. *Applied Sciences* 12, 15 (2022), 7908.
Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. 2008. MeshLab: an Open-Source Mesh Processing Tool. In *Eurographics Italian Chapter Conference*, Vittorio Scarano, Rosario De Chiara, and Ugo Erra (Eds.). The Eurographics Association, 129–136. https://doi.org/10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136
DGG. 2024. RapidCompact. https://www.rapidcompact.com/.
Benedikt Dietrich, Dip Goswami, Samarjit Chakraborty, Apratim Guha, and Matthias Gries. 2013. Time series characterization of gaming workload for runtime power management. *IEEE Trans. Comput.* 64, 1 (2013), 260–273.
EpicGames. 2024. Unreal Engine - Nanite. https://www.epicgames.com/.
Cristian Gadea, Daniel Hong, Dan Ionescu, and Bogdan Ionescu. 2016. An architecture for web-based collaborative 3D virtual spaces using DOM synchronization. In *2016 IEEE International Conference on Computational Intelligence and Virtual Environments for Measurement Systems and Applications (CIVEMSA)*. IEEE, 1–6.
Michael Garland and Paul S Heckbert. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 209–216.
Federica Gaspari, F Ioli, F Barbieri, C Rivieri, M Dondi, and L Pinto. 2023. Rediscovering cultural heritage sites by interactive 3D exploration: A practical review of open-source WebGL tools. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 48 (2023), 661–668.
C Gautier, J Delanoy, and Gilles Gesquière. 2023. Representation of urban geometry evolution through space-time cube. In *2023 27th International Conference Information Visualisation (IV)*. IEEE, 414–419.
Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. 2022. Why do tree-based models still outperform deep learning on typical tabular data? *Advances in neural information processing systems* 35 (2022), 507–520.
Yan Gu and Samarjit Chakraborty. 2008. A hybrid DVS scheme for interactive 3D games. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*. 3–12.
Ujjwal Gupta, Joseph Campbell, Umit Y Ogras, Raid Ayoub, Michael Kishinevsky, Francesco Paterna, and Suat Gumussoy. 2016. Adaptive performance prediction for integrated GPUs. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8.
Alexander Hapfelmeier and Kurt Ulm. 2013. A new variable selection approach using random forests. *Computational Statistics & Data Analysis* 60 (2013), 50–69.
A. Kapoulkine. 2023. Meshoptimizer. https://github.com/zeux/meshoptimizer.
Sun-Jeong Kim, Chang-Hun Kim, and David Levin. 2002. Surface simplification using a discrete curvature norm. *Computers & Graphics* 26, 5 (2002), 657–663.
Hyunho Lee and Min-Ho Kyung. 2016. Parallel mesh simplification using embedded tree collapsing. *The Visual Computer* 32 (2016), 967–976.
Peter Macken, Marc Degrauwe, Mark Van Paemel, and Henri Oguey. 1990. A voltage reduction technique for digital systems. In *1990 37th IEEE International Conference on Solid-State Circuits*. IEEE, 238–239.
Andrea Maggiordomo, Federico Ponchio, Paolo Cignoni, and Marco Tarini. 2020. Real-World Textured Things: A repository of textured models generated with modern photo-reconstruction tools. *Computer Aided Geometric Design* 83 (2020), 101943.
Qiusha Min, Zhifeng Wang, Neng Liu, et al. 2018. An evaluation of HTML5 and WebGL for medical imaging applications. *Journal of healthcare engineering* 2018 (2018).
Bren C Mochocki, Kanishka Lahiri, Srihari Cadambi, and X Sharon Hu. 2006. Signature-based workload estimation for mobile 3D graphics. In *Proceedings of the 43rd annual design automation conference*. 592–597.
Iman Soltani Mohammadi, Mohammad Ghanbari, and Mahmoud Reza Hashemi. 2022. GAMORRA: An API-level workload model for rasterization-based graphics pipeline architecture. *Computers & Graphics* 106 (2022), 9–19.
Anuj Pathania, Alexandru Eugen Irimiea, Alok Prakash, and Tulika Mitra. 2015a. Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs. In *Proceedings of the 52nd Annual Design Automation Conference*. 1–6.
Anuj Pathania, Santiago Pagani, Muhammad Shafique, and Jörg Henkel. 2015b. Power management for mobile games on asymmetric multi-cores. In *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 243–248.
Rolandos Alexandros Potamias, Stylianos Ploumpis, and Stefanos Zafeiriou. 2022. Neural mesh simplification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 18583–18592.
UL Solutions. 2023. 3DMark. https://benchmarks.ul.com/.
Christian Stein, Max Limper, and Arjan Kuijper. 2014. Spatial data structures for accelerated 3D visibility computation to enable large model visualization on the web. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies*. 53–61.
Mustafa M Tikir, Laura Carrington, Erich Strohmaier, and Allan Snavely. 2007. A genetic algorithms approach to modeling the performance of memory-bound computations. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. 1–12.
Sylvie Servigne Vincent Jaillot and Gilles Gesquière. 2020. Delivering time-evolving 3D city models for web visualization. *International Journal of Geographical Information Science* 34, 10 (2020), 2030–2052. https://doi.org/10.1080/13658816.2020.1749637 arXiv:https://doi.org/10.1080/13658816.2020.1749637
Michael Wimmer and Peter Wonka. 2003. Rendering time estimation for real-time rendering. In *Rendering Techniques*. 118–129.
Da-Jing Zhang-Jian, Chung-Nan Lee, Ing-Jer Huang, and Shiann-Rong Kuang. 2009. Power estimation for interactive 3D Game using an efficient hierarchical-based frame workload prediction. In *Proceedings: APSIPA ASC 2009: Asia-Pacific Signal and Information Processing Association, 2009 Annual Summit and Conference*. Asia-Pacific Signal and Information Processing Association, 2009 Annual ..., 208–215.

A TECHNICAL DETAILS

In this kind of performance-sensitive experiment, any other application running on their device could alter our measurements, thus we defined a strict protocol and asked all participants to ensure they closed every unnecessary and GPU-related application on their device before the start of the experiment. We disable the vertical synchronization that is activated by default in the browser allowing us to measure frametime above the screen refresh rate. The experiment has to run in full screen to match the screen resolution. On multi-GPU configuration and for Windows users only, we specified a procedure to force the main GPU to be used with Chrome. Our process is now fully scripted and the user only has to press one button to start the experiment.

The scene is composed of one ambient light and a perspective camera with position (0,0,10) in XYZ axes. As for the renderer parameters, we use sRGBEncoding for the renderer output, and ACESFIlmic for the tone mapping with the exposure coefficient set to 1. The pixel ratio used is the one provided by the window as well as the width and height parameters of the participant screen.

Our experiment is implemented in Javascript and some precautions have been taken regarding memory management. The Garbage Collector (GC) is an automatic process that is activated when the resource is disposed but we have no certitudes about the completion of the task of cleaning thus we fix a timer of six seconds to avoid any memory leak between the objects. Chrome has to be started with custom flags: '–disable-frame-rate-limit', '–disable-gpu-vsync', and '–start-fullscreen'.

B FEATURE ANALYSIS

The random forest model has a built-in feature importance mechanism allowing us to gain deeper theoretical insight. Specifically, we were interested in the impact of 3D mesh features on the predictability of the rendering time. We compute the permutation importances [Hapfelmeier and Ulm 2013] on our model *FTRF* and we extracted the scores reported in Figure 8.

From these importance values, we can deduce that the number of transformed vertices *Vertices Transformed* is the most important feature for predicting rendering times on new objects and devices. Next come the descriptors relating to device performance (*Transform 262144* and *Rasteriser 32*), which also play a predominant role in the predictions.

The number of textures (*Texture Number*) and the number of pixels (*Tex Pixel Count*) are the least important features probably

Table 5: Ablation study: Device performance features are always used, MAE and Emax in milliseconds

	Features	$Corr_p$	R^2	MAE(ms)	Emax(ms)	MAPE
CV_meshs	G	0.908	0.819	1.348	32.951	21.427
	UV	0.885	0.782	1.620	33.192	34.395
	DO	0.968	0.935	0.791	24.800	13.507
	All	0.971	0.941	0.726	25.063	11.675
FTRF	G	0.853	0.718	1.628	30.676	27.629
	UV	0.833	0.685	1.853	34.100	40.535
	DO	0.935	0.834	1.254	21.961	21.626
	All	0.935	0.854	1.161	22.117	19.701

G stands for geometric, UV for texture layout and DO for data ordering features

because they are too redundant with other parametrization-related descriptors but removing them increases the error slightly.

We also conducted an ablation study (see Table 5) that consists of training four predictors in the same way that we present in Section 6 meaning the cross-validation splits with constraints are used here with four sets of descriptors: G for geometrical descriptors, UV for texture layout descriptor, DO for data ordering and All for every features. Rendering performance descriptors are always used in each predictor. From these results we can first deduce an overall trend: features in the Texture Layout (UV) category have a lower predictive power than those in the Geometric and Data Ordering categories but they are still required to achieve the best L1 error MAE. There is a strong relation between the data ordering feature and the frametime as this set alone achieves a relatively close prediction performance but it is only when we use the three sets of features all together that we can achieve the best performance for every metric used in this evaluation. Overall, Data Ordering, Geometric, and Device Performance descriptors are the most important features in predicting the rendering time. Texture layout descriptors fall below others because they seem to have a lesser influence factor.

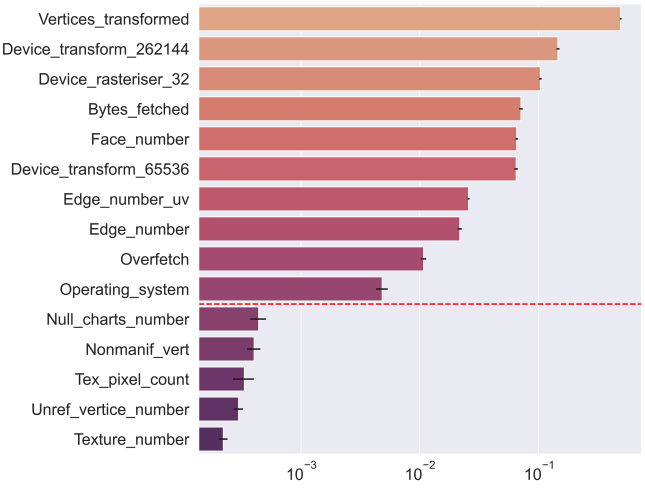

Figure 8: Permutation importance: Random forest feature - We report the 10 most important features and the least five respectively above and under the red line. We use a logarithmic scale for the x-axe and the scores are obtained from the FTRF predictor permutation importance