Hindawi
Scientific Programming
Volume 2019, Article ID 9195845, 15 pages
https://doi.org/10.1155/2019/9195845

Research Article

Avionics Graphics Hardware Performance Prediction with Machine Learning

Simon R. Girard,[1] Vincent Legault,[1] Guy Bois ,[1] and Jean-François Boland[2]

[1]*Computer Engineering Department, Polytechnique Montréal, C.P. 6079, Succ. Centre-Ville, Montréal, Québec, Canada H3C 3A7*
[2]*Electrical Engineering Department, École de Technologie Supérieur (ETS), 1100 Rue Notre-Dame Ouest, Montréal, QC, Canada H3C 1K3*

Correspondence should be addressed to Guy Bois; guy.bois@polymtl.ca

Received 21 December 2018; Revised 15 April 2019; Accepted 6 May 2019; Published 3 June 2019

Guest Editor: Vicente García-Díaz

Within the strongly regulated avionic engineering field, conventional graphical desktop hardware and software application programming interface (API) cannot be used because they do not conform to the avionic certification standards. We observe the need for better avionic graphical hardware, but system engineers lack system design tools related to graphical hardware. The endorsement of an optimal hardware architecture by estimating the performance of a graphical software, when a stable rendering engine does not yet exist, represents a major challenge. As proven by previous hardware emulation tools, there is also a potential for development cost reduction, by enabling developers to have a first estimation of the performance of its graphical engine early in the development cycle. In this paper, we propose to replace expensive development platforms by predictive software running on a desktop computer. More precisely, we present a system design tool that helps predict the rendering performance of graphical hardware based on the OpenGL Safety Critical API. First, we create nonparametric models of the underlying hardware, with machine learning, by analyzing the instantaneous frames per second (FPS) of the rendering of a synthetic 3D scene and by drawing multiple times with various characteristics that are typically found in synthetic vision applications. The number of characteristic combinations used during this supervised training phase is a subset of all possible combinations, but performance predictions can be arbitrarily extrapolated. To validate our models, we render an industrial scene with characteristic combinations not used during the training phase and we compare the predictions to those real values. We find a median prediction error of less than 4 FPS.

1. Introduction

In recent years, there has been an increased interest in the avionics industry to implement high-performance graphical applications like synthetic vision systems (SVSs) that display pertinent and critical features of the environment external to the aircraft [1]. This has promoted the advent of faster graphical processing hardware. Because it is a highly regulated field, conventional desktop and embedded graphics hardware could not be used because they do not conform to the DO-178C and DO-254 avionic certification standards (the international standards titled "RTCA DO-178C—Software Considerations in Airborne Systems and Equipment Certification" and "DO-254—Design Assurance Guidance for Airborne Electronic Hardware" are the primary standards for commercial avionics software and hardware development) [2, 3]. Considering the need of avionic hardware with higher performance, we observe that graphical application development tools and hardware benchmarks and simulators available for conventional embedded or desktop graphical applications seem still to be missing for avionics applications. This fact is made especially clear when a quick search through the specifications of the most renowned tools such as Nvidia Nsight [4], AMD PerfStudio [5], or SPECViewPerf [6] lead to the same conclusions. Even though, there are some WYSIWYG ("what you see is what you get") GUI toolboxes available for the ARINC-661 standard [7, 8], it seems that there is no performance benchmark, performance prediction tool, or performance-correct simulator available for graphical avionic hardware. The interest in having such tools is especially significant because most development processes include a design phase before the actual implementation. Taking the example on the classical V-Model [9], designers must make choices in regard to the purchase or the in-house development of graphical hardware. But as they want to evaluate the performance of such hardware relating to the choices made, they need some kind of performance metrics and benchmarks. This benchmarking tool should be provided by the software development team but as the project is still in the design phase, they have not yet necessarily implemented a graphical engine to enable performance testing. Performance prediction can be useful to (1) further extrapolate the benchmark performance results for any volume of graphical data sent to the hardware and (2) reduce the number of benchmarks required to evaluate various use cases. Going further, the performance models generated can then be used to develop a performance-correct hardware simulator that developers can use on their workstation, in order to have a general preview of the efficacy of their software, before executing it on the real system. As this is the case with the most hardware emulation tools, this reduces the development costs by facilitating functional verification of the system [10].

In this work, we demonstrate two prototype tools that can be used as a pipeline to evaluate and then predict the performance of graphical hardware. The first tool is a benchmark that can generate and then render custom procedural scenes according to a set of scene characteristics such as the number of vertices, size of textures, and more. It evaluates and outputs the number of frames generated per seconds (FPS). The second tool takes the output of a certain number of executions of the benchmark and generates a nonparametric performance model, by using machine learning algorithms on the performance data. Those performance models can then extrapolate predictions of performance for any dense 3D scene rendered on this piece of hardware. We evaluated the distribution of prediction errors experimentally to find that most prediction errors will not exceed 4 FPS.

In the rest of the paper, Section 2 presents the main problems which make inadequate the aforementioned existing tools for the avionics industry. It also presents the work related to the various algorithms and methods used by those standard tools. Section 3 presents the first contribution which is the graphical avionic application benchmarking tool. Then, Section 4 presents the second contribution which is the performance model generation tool. Section 5 presents the experimental method used to evaluate the prediction power of these models. Section 6 presents analyses and discusses the achieved prediction error distributions. Finally, Section 7 provides information for those who would like to repeat the experiment.

2. Background

Among the differences between conventional (consumer market) and avionics graphic hardware development, three are denoted as especially standing out. The first is the use of OpenGL SC instead of OpenGL ES or the full OpenGL API to communicate with the hardware [11]. The second is the use of a fixed graphics pipeline instead of letting the possibilities of using shaders or custom programs that can be sent to the graphics hardware to modify the functionalities of certain areas of the rendering pipeline [12]. Finally, there is the research interest in the development of DO-254-compliant graphical hardware, in the form of software GPUs, FPGAs, and CPU/GPU on-a-chip to name a few [2]. The nature of the hardware is then not necessarily a processor, and certain metrics specific to that nature cannot be applied, such as instruction count. Also, avionics graphical hardware is usually a very secured black box that cannot be intruded to actually perform the instruction count metrics on the internal programs. Thus, performance benchmark and prediction tools in an avionics context should account for these specificities. By only using the functions available in OpenGL SC to evaluate the performance of the hardware, we make sure of the following two points: (1) to use the standard fixed pipeline that accompanies this version of the API and (2) to be independent on the nature of the underlying hardware beyond that interface.

To the best of our knowledge, graphics hardware performance prediction in the avionics context does not exist in the literature. Looking for methods to closely related fields would thus be the best approach. It is then interesting to look over the literature to find the methods that have been used in a conventional desktop and embedded context. It is also interesting to widen this review to general computer hardware and microarchitecture, as well as graphical hardware performance prediction. Numerous benchmarks for graphic hardware exist in the conventional context, such as SPECViewPerf or Basemark [6, 13], to name a few. Even if they do not satisfy the special problematic of the avionics needs, their workflow can be a source of inspiration. For example, SPECViewPerf allows users to create a list of tests, each varying different characteristics of the scenes or the render state, such as local illumination models, culling, texture filters, simple, or double buffering. It then returns the average FPS attained during the rendering of the scene for each test. The use of the average FPS might be more significant for the user, but because the FPS distribution is not normal it loses a lot of statistical significance. As for the performance prediction tools, they tend to be made available by the graphic hardware manufacturers such as the NVIDIA® Nsight™ [4] or the AMD GPU PerfStudio [5]. The problem with these tools is that they are only available for the desktop and embedded domain and are not adapted to the needs of avionics, as explained previously. It is still interesting to review the literature to better understand how those profiling tools might work internally. There are three main approaches to generate models: analytical modeling, parametric modeling, and machine learning. Analytical models attempt to create mathematical models that represent performance as a set of functions describing the hardware. They often use metrics such as instruction count and properties such as frequency clock of the processor [14–17]. The main issue with these methods is that they require a good understanding of the hardware's inner workings, which is difficult in an avionics context because they are secured black box entities. Also, because of the fixed pipeline of the graphics card, system engineers cannot obtain the inner programs operating the pipeline and thus cannot use analytical metrics such as instruction count. Also, the literature seems to indicate that it is very difficult to truly identify all factors influencing performance and thus to mathematically model them. However, there is one analytical model that has the potential to be used in an avionic context. It is a function that estimates the transfer time of data from the main to the graphic memory [18].

The creation of parametric models implies the use of parametric regressions such as linear or polynomial regressions. It has been used for microarchitecture and CPU design-space exploration [19–22], but because we only have access to the interface of the hardware, it is hard to identify all the factors influencing the performance. Thus, these methods would have difficulty to explain most of the variance of the performance data and can then perform poorly. This is usually solved by using nonparametric regression models created with machine learning.

Even though there are a large number of machine learning that can be used to create performance models, we identified four algorithms that are mainly used throughout the literature to generate performance models in the specific case of processors, microarchitecture explorations, or parallel applications: regression trees, random forest, multiple additive regression trees (MART), and artificial neural networks. Performance and power consumption prediction in the case of design-space exploration of general purpose CPUs has been achieved with random forests [23] and MART [24, 25]. Regression trees [26] have been used for performance and power prediction of a GPU. Artificial neural networks have successfully been used for performance prediction of a parallelized application [21], but also for workload characterization of general purpose processors [27, 28], superscalar processors [29], and microarchitectures [30]. A variant of the MART method has also been used for predicting performances of distributed systems [31]. Regression trees are usually less accurate, and its more robust version, the random forest, is usually preferred. Madougou et al. [25] used nvprof, a visual profiler, to collect performance metrics (cache miss, throughput, etc.) and events of CUDA kernels running on NVIDIA GPUs. The data are stored in a database and further used for model building. This approach seems very promising but cannot be easily adapted to the needs of avionics, as explained previously.

Another problem with tree-based methods is their low performance to predict values from predictors out of the range of the values of the observations used to train them (extrapolation). Recent work on hybrid models generated from a mix of machine learning and first-principle models has also yielded good results for similar applications [32–34].

3. Avionics Graphic Hardware Performance Benchmarking

There are two main steps in the creation of our performance models. First, a benchmark must be executed to gather performance data for various scene characteristics. The GPU benchmarking consists in itself in the generation of a customizable synthetic 3D scene and in the analysis of the render time of each frame. Second, performance models are generated with machine learning from the performance data obtained in the first step. For our experimental purposes, we add a model validation phase to evaluate the predictive power of those performance models by comparing the predictions with the render time of a customizable and distinct validation 3D scene. Figure 1 presents this dataflow. The remainder of this section will present the requirements and the implementation of our proposed avionics GPU benchmarking tool.

Performance data acquisition for a piece of hardware is achieved with our benchmarking tool as follows. First, a synthetic scene is generated according to various parameters. Then, the scene is rendered and explored by following a specific camera movement pattern. Finally, a last analysis step is performed to evaluate for each frame the percentage of the number of vertices of the scene that has been rendered. We use a study case from an industrial partner to enable us to enumerate the various characteristics of graphical data that might have an impact on the rendering performance of an avionic graphical application. The study case was a SVS using tile-based terrain rendering.

System performances can be measured in several ways by test benches. For a 3D graphics system, one of the most effective methods is to try to reproduce the behaviour of a real system [35]. Thus, we characterized our industrial partner's study case to extract an exhaustive list of all the graphical features to take into account while implementing our benchmarking tool. Our tool tests these features one by one, by inputting a set of values to test per feature. It then outputs results that give a precise idea of how each graphical feature impacts rendering performance. The graphical features involved in our industrial partner's study case and which we tested in our tool are as follows:

(1) Number of vertices per 3D object (terrain tile)
(2) Number of 3D objects per scene
(3) Size of the texture applied on 3D objects
(4) Local illumination model, either per-vertices (flat) or per-fragment (smooth)
(5) Presence or absence of fog effect
(6) Dimension of the camera frustum
(7) Degree of object occlusion in the scene

Those parameters follow the hypothesis that the principal factors that would influence the performances are directly related to the amount of data sent through the graphic pipeline. The amount of work required by the graphic hardware would be in relation to the amount of data needed to be rendered because of the amount of operations required to send all these data across the pipeline.

Compared to the list of features tested in a contemporary graphical benchmark, this set list brings us back to the beginnings of 3D graphics era. The reason why this list is made of basic graphic features is because critical graphical avionic software uses a version of OpenGL which is stripped

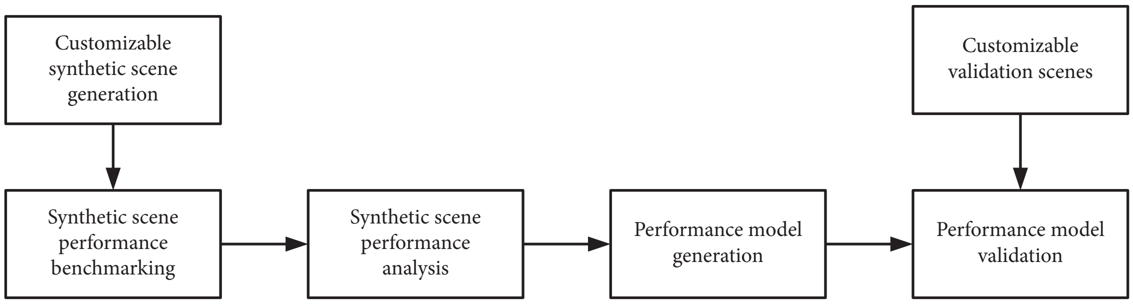
Customizable synthetic scene generation
Customizable validation scenes
Synthetic scene performance benchmarking
Synthetic scene performance analysis
Performance model generation
Performance model validation

Figure 1: Dataflow of the proposed tool in an experimental context.

down to its simplest form: OpenGL Safety Critical (SC) [36]. This simple API was specifically designed for safety-critical environments because it is easier and faster to certify. OpenGL SC was also designed for embedded system limited hardware resources and processing power.

Nowadays, graphical benchmarks are designed for new generation graphic cards, game consoles, or smartphones. These platforms were designed to deliver a gigantic graphical processing power in a noncritical environment. It is thus normal that this kind of product has its own set of benchmarking tools. Basemark® ES 2.0 [37] (and now Basemark ES 3.0 [13]), one of the most popular graphical benchmarking suite for GPUs used in the mobile device industry, offers benchmarks that are not suitable for avionics platforms for several reasons:

(1) It uses OpenGL ES 2.0 which is an API too rich and complex to be a reasonable candidate for the full development of a certifiable driver on safety-critical platform [12].

(2) Sophisticated lighting methods are tested like per-pixel lighting (used to create bump mapping effects), lightmap (used to create static shadow effects), or shadow mapping (used to create dynamic shading effects). Although these lighting methods offer rich visual effects, they remain pointless within an avionic context, where rendering accuracy is the main concern for 3D graphics.

(3) Various image processing effects are tested like tilt and shift effect, bloom effect, or anamorphic lens flares effect. More complex lightning models such as Phong interpolation and particle effects are also tested (usually by implementing shaders). Once again, these visual effects will not enhance the accuracy of rendering, and also shaders are not supported in OpenGL SC.

(4) Results and metrics resulting from these kinds of benchmarks are usually undocumented nebulous scores. Those scores are useful when we want to compare GPUs between them, but they cannot be helpful when system architects want to figure out if a particular piece of hardware satisfied a graphical software processing requirement. When designing a safety-critical system, metrics like GPU's RAM loading time (bandwidth, latency), level of image detail (maximum number of polygons and 3D objects per scene), and maximum size of textures or the processing time per frame are much more significant data.

Since the current graphical benchmarking tools were not designed for a safety-critical environment, we thus decided to implement specialized benchmarking tool for the avionic industry.

Based on this analysis, from those 7 factors, we divide a tile-based synthetic scene that would evaluate rendering performance based on these factors. The benchmark tool takes a list of "tests" as input. Each test influences the generation of the procedural 3D scene by manipulating a combination of those factors. The output of the benchmark tool is a file with time performance according to the inputted characteristics. Each test is designed to evaluate the performance of the scene rendered by varying one of the characteristics and keeping fixed every other. Consider, for instance, the tile resolution test, for each value, the benchmark will be executed, and a vector of performance will be outputted. During this test every other characteristics (e.g., the number of tiles or the size of textures) shall be fixed. It is important to mention that tile-based scenes are stored as height maps or even dense 3D scenes, removing the need for analyzing the number of triangles or faces because it can always be derived or approximated from the number of vertices.

3.1. Synthetic Scene Generation. Each tile of our synthetic scene contains a single pyramidal-shaped mesh. We used this shape because it can model various ground topography by varying the height of the pyramid. Furthermore, it enables the possibility to have an object occlusion when the camera is at a low altitude and it is oriented perpendicularly to the ground. Also, this shape is easy to generate from a mathematical model. The remaining of this subsection presents how the visual components of the procedural 3D scene are generated and how they help to produce more representative performance data.

3.1.1. Tiles' Dimensions. Tiles have a fixed dimension in OpenGL units, but the number of vertices it can contain can vary depending on the corresponding benchmark input value. To simplify the vertices count of our models, we use a per-dimension count c for the square base of the pyramids, meaning that each tile has a resolution of c^2 vertices. When the perspective distortion is not applied, each vertex of the pyramid is at equal distance of its neighbours in the XZ plane (Figure 2).

3.1.2. Noise. To further reproduce a realistic ground topography, we add random noise to the pyramid faces to unsmooth them. The quantity of noise applied is more or less 10% of the height of the pyramids and is only applied to the Y coordinates (attributed to the height) as shown in Figure 3. This proportionality helps to keep the general shape of the pyramid, regardless of its height.

3.1.3. Grid Generation. The grid of tiles is generated according to the corresponding benchmark input value. As for the tile resolutions, the grid size is measured as a per-dimension value v, meaning that the total grid size is v^2, and thus the grid has a square shape. In a real context, a LOD functionality is usually implemented, making farthest tiles load at a lower resolution and nearest tiles at a full resolution. However, because we evaluate the worst-case execution performance of the hardware, every tile has full resolution.

3.1.4. Pyramid Height. The height of the pyramids varies from tile to tile, depending on their position in the tile grid, but the maximum height will never exceed the quarter of the length of the scene. This constraint enables the possibility to have various degrees of object occlusion for the same scene, depending on the position and orientation of the camera. Because of the positioning of the camera and the movement pattern (explained previously), the bigger the tile grid is, the higher the pyramids are. To obtain consistent scene topologies for each benchmark test, the pyramid height is calculated from the index of the tile in the grid (Figure 4) and is always a factor of two from the maximum pyramid height.

3.1.5. Texture Generation. The OpenGL SC API requires the use of texture dimensions that are powers of two. For simplicity, we create RGB-24 procedural textures which consist of an alternation of white and black texels. For each vertex of the tile, the texture itself and the texture coordinates are computed before the frame rendering timer starts. In real cases, this information is normally already available in some kind of database and not generated in real time, so it should not be taken into account by the timer measuring the period taken to draw the frame.

3.2. Camera Movement Pattern. According to the case study, there are three typical use cases for the camera movement and position patterns:

(1) Low altitude: a small percentage of the scene is rendered with the possibility of much object occlusions

(2) Midrange altitude: about half of the 3D objects are rendered with possibly less object occlusions

(3) High altitude: the whole scene is potentially rendered with low chances of object occlusions

For each test of a benchmark, the camera position goes through each of these use cases. To achieve this, the camera always starts at its maximum height over the tile at the middle of the grid. The camera then performs a 360 degrees rotation in the XZ plane, while also varying its inclination over the Y-axis depending on its height. After each 360 degrees rotation, the camera height is reduced and there are eight possible values for each test. The inclination angle over the Y-axis is not constant throughout the various heights taken by the camera, in order to cover the highest possible number of viewpoints of the scene. At the maximum height, the inclination leans towards the edges of the grid, and at the lowest height the camera points perpendicularly towards the ground. The camera inclination for every camera height is calculated with a linear interpolation between the inclinations, at maximum and minimum heights. Overall, each 360 degrees rotation of the camera will yield 32 frames for a total of 320 frames for each benchmark run.

The camera frustum is created to mimic the one used by the study case SVS. It implements a 45 degrees horizontal field of view and a vertical field of view corresponding to the 4 : 3 ratio of the screen, according to the OpenGL standard perspective matrix. Also, to maximize the precision of the depth buffer, it is desirable to show a maximum of vertices with the smallest frustum possible. The last important parameter is to define the maximum height of the camera. We set this limit to the value of the length of the scene in OpenGL units because the scene will be smaller than the size of the screen passed that length. Thus, the far plane of the frustum must carefully be chosen in regards of the scene length as the maximum depth of the scene will most likely vary accordingly.

3.3. Loading Data to the Graphic Memory. To help reduce the randomness of the performance of the graphics hardware and due to the internal properties of most rendering pipelines, we apply the tipsify algorithm [38] to the vertex index buffer before sending it to the graphics pipeline. This should reduce the average cache miss ratio of the standard internal vertex program of the fixed pipeline. All the 3D data is loaded to the graphics memory before beginning the rendering and the performance timer. Because the scene is static, no further data need to be sent to the hardware, so the loading time does not influence the overall performance. This would not be the case in a real context, but as presented in Section 2, the literature presents at least one method to estimate the influence of data loading during the rendering process. If needed, the benchmark tool can return the time required to load this static data from the main memory to the graphics memory.

3.4. Analysis of the Percentage of Scene Drawn. The data sent to the graphical hardware for rendering usually contain 3D objects that could be ignored during the rendering process because they are either unseen or hidden by other 3D

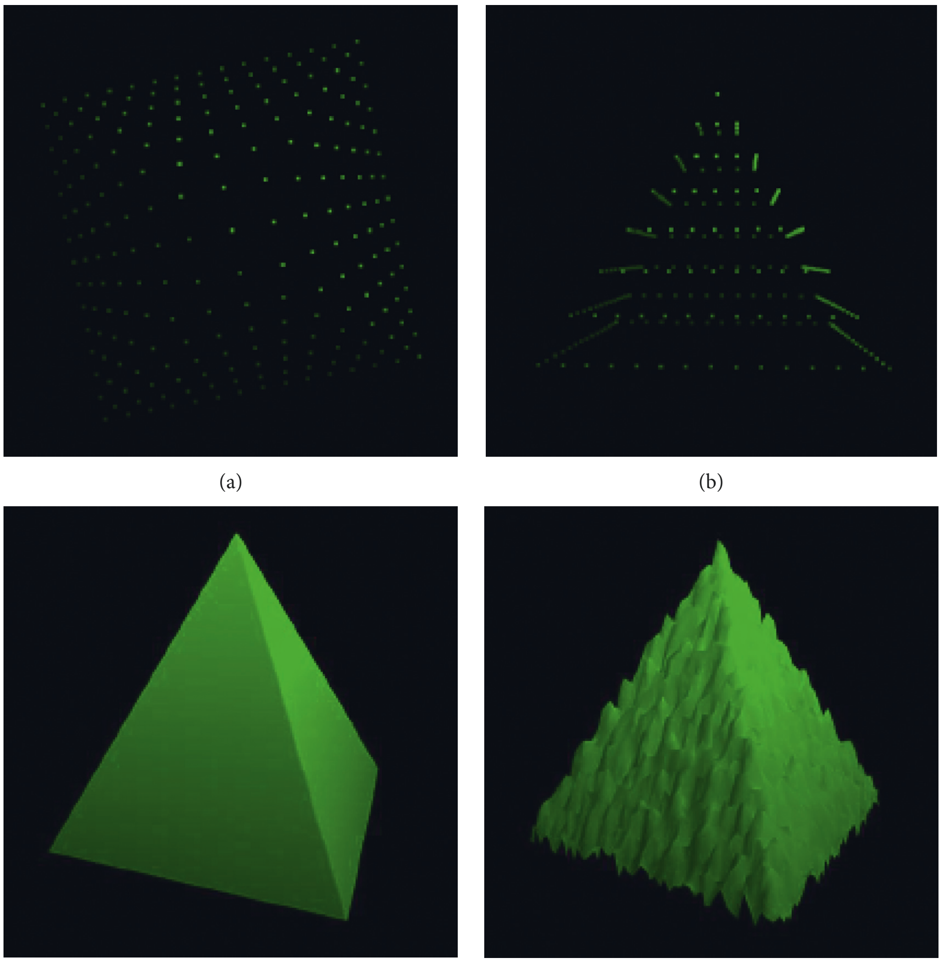

Figure 2: Pyramid vertices generated with a c-by-c dimension top facing (a) and front facing (b). Pyramid rendered mesh without added noise (c) and with added noise (d).

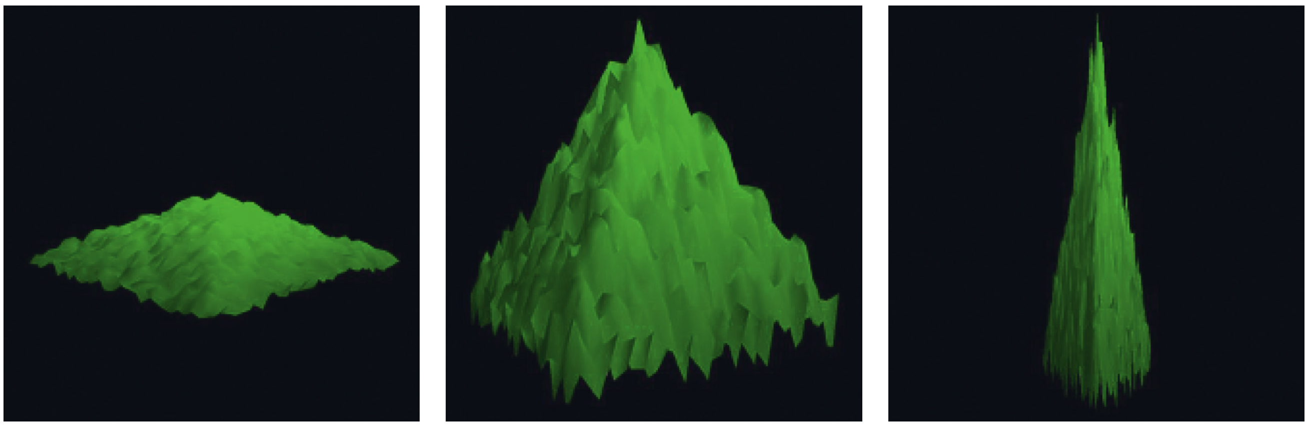

Figure 3: Various intensity of noise depending on the height of the pyramid. Low-noise amplitude (a) to high-noise amplitude (c).

objects, due to their spatial positioning. Thus, culling methods are commonly used to avoid the rendering of such objects. These methods are (1) the frustum culling which ignores the rendering of triangles outside of the camera frustum, (2) the back-face culling which ignores the rendering of triangles that are facing away from the camera orientation, and (3) the Z-test which ignores the per-fragment operations such as the smooth lighting.

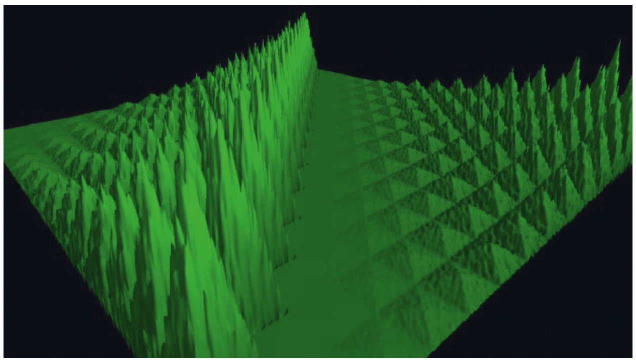

Figure 4: Overall generated synthetic scene with pyramids height varying according to their position in the grid.

As mentioned earlier, the final step of the benchmarking process is the analysis of the percentage of the vertices of the scene that were used during the rendering process. To do this, we count the number of vertices that are in the camera frustum and also that are part of front facing surfaces. We used a geometric approach by comparing the position of each vertex with the six planes of the frustum in order to determine if the vertex is inside it or not. If it is, the next step is to determine if it is front or back facing. To do this, we first transform the normal of the surface, of which the vertex is a part, from world-space to camera-space. Then, we compare the angle between the normal and the camera-space eye vector, which is a unit vector pointing in the Z-axis. If the angle is between 90 and 270 degrees, then the vertex is considered to be front facing. Finally, we can evaluate the percentage of vertices drawn as the size of the set of vertices that pass both tests, divided by the total number of vertices in the scene.

3.5. Rendering Performance Metrics. The performance metric returned by the benchmark is the instantaneous frame per second (IFPS), which is measured for each frame by inverting the time it took to render the frame. It is more desirable than a moving-average FPS over multiple frames because we can then apply more specialized smoothing operations to eliminate further any outliers. On the other hand, the benchmark uses a vertical sync of twice the standard North American screen refresh rate: 120 Hz. We found that not using vertical sync yields a very high rate of outliers for IFPS greater than 120. Also, using a vsync of 60 FPS may create less accurate models as most of the scene characteristics will yield the maximum frame rate. Finally, to ensure the proper calculation of the IFPS for each frame, we use the *glFinish* command to synchronize the high resolution timer with the end of the rendering.

4. Avionics Graphic Hardware Performance Modeling

Performance modeling of 3D scenes by using the characteristics of the latter is challenging because it is hard to take into account the noise made by random events in the abstracted hardware (processor pipeline stall, branching, etc.). The first step in the creation of a performance model is to evaluate which of the benchmark scene characteristics best explains the variance of FPS. In preliminary tests, we ran the benchmark by varying the values of one characteristic, while keeping fixed every other. This is repeated for each characteristic until all of them have been evaluated. We concluded that the size of the grid of tiles and the resolution of vertices in each tile are the most significantly well predicted characteristics by the machine learning algorithms. The size of the screen and the size of the texture also contribute to the variation of the FPS, but they were harder to model using our method. To predict IFPS in terms of texture sizes and screen sizes, a distinct performance model must be generated for each of their combinations, by training it with a subset of every possible combination of grid size, tile resolution, and percentage of vertices rendered. This limitation is being worked on as a future contribution. To generalize tile-based scenes to dense voxelized scenes without fixed resolutions in each voxel, the tile resolutions can be replaced by the mean number of vertices per voxel.

Besides, another key factor in the creation of good models is the fact that they can be generated efficiently without the need to feed the FPS obtained for every possible combination of scene characteristics to the learning algorithms. This number has been calculated to be about 58 million combinations of grid size and number of vertices. Compared to this number, the number of combinations of texture sizes (10) and of screen sizes (5) is relatively small. Generating a distinct performance model for each combination of texture and screen sizes would be a more trivial task, if the number of combinations of grid size, tile resolution, and percentage of vertices rendered could be reduced. To organize those performance models, we create a three-level tree, where the first two levels represent the combinations of texture and grid size. The third level contains a leaf pointing to a nonparametric model trained with machine learning that predicts IFPS in terms of grid sizes and tile resolutions. Those nonparametric models are created by feeding the machine learning algorithms with only a small percentage of the performance of the whole combinations of grid sizes, tile resolutions, and percentage of vertices, while keeping fixed the texture and screen size characteristics according to the leaf parents. The choice of the subset of total grid sizes and tile resolution combinations to use is chosen by selecting those inducing the worst-case rendering performance. We run the benchmark only twice by running the tile resolution and grid size variation tests and by concatenating the output performance vectors of each. The characteristics evaluated by the benchmark for the training dataset are shown in Table 1.

4.1. Machine Learning Algorithms Configuration. As stated in Section 2, three machine learning algorithms are of special interest for the task of performance prediction in the case of processors and parallel applications: random forest, MART, and artificial neural networks. We offer the comparison between the predictive powers of nonparametric models trained with each of these algorithms in order to determine

Table 1: Values used for the tile resolution grid size tests.

Variation test	Values of tile resolution	Values of grid size
Tile resolution	7; 9; 13; 17; 21; 25; 31; 37; 45	25
Grid size	25	7; 9; 13; 17; 21; 25; 31; 37; 45

Table 2: Parameters for the machine learning algorithms used.

Algorithm	Parameter nature	Optimal range
Artificial neural network	Number of hidden layers	1
	Number of nodes in the hidden layer	[5; 15]
MART	Number of weak learners	[500; 1000]
Random forest	Number of bootstrapped trees	[50; 150]

which one is the most suited for this application. We chose those algorithms to reflect the previous work of the scientific programming community as we felt they would be the best fit for GPU prediction. Other machine learning methods are present in the literature such as Bayesian networks and gradient boosting machines, to name a few, but have not been considered in the current experiment. Each of these algorithms has to be configured before its use: the number of hidden layers and the number of nodes per layers in the artificial neural networks, the number of bootstrapped trees in the case of random forest, or the number of weak learners in the case of MART. Most of the time, there is no single optimal parameter. It usually takes the form of a range of values. These ranges were found during preliminary experimentation and are given in Table 2. In the case of artificial neural networks, we used a multilayer feedforward perceptron trained with backpropagation based on the Levenberg–Marquardt algorithm. We also try to improve the problem generalization and reduce overfitting by using early stopping methods and scaling the input performance values in the range $[-1, 1]$. Early stopping methods consist in stopping prematurely the training of the neural network when some conditions are met. It can be after a certain number of epoch (1000 in our case), when the performance metric has reached a certain threshold (mean squared error is 0.0). But the most commonly used early stopping method is to divide the training dataset in two subsets: a training subset (70% of the initial dataset) and a validation subset (remaining 30%). After each epoch, the neural network makes predictions using the validation dataset and training is stopped when those predictions are accurate enough (e.g., error metric between the prediction and the predicted value is lower than a certain threshold).

Because of the random nature of the optimality of a parameter value, we create three performance models for each machine learning algorithm. The first model uses the lowest parameter value, the second model uses the middle range one, and the last model uses the upper one. During the validation phase, we retain the model which yields the lowest prediction error.

4.2. Performance Data Smoothing. The performance data output by the benchmark itself is randomly biased because it cannot explain some of the variance of IFPS, which can vary even for scenes with similar characteristics. The fact that we only analyze the input and output of a graphical hardware black box partially explains the variance because many internal factors can influence the output for the same input: cache misses ratio, processor instruction pipeline, and instruction branching unpredictability to name a few. Because the program running on the hardware is fixed, we can assume that these factors are not enough random to make the analysis of input/output unusable for performance prediction. To reduce this noise in the benchmark output data, we apply an outlier-robust locally weighted scatterplot smoothing. This method, known as LOESS, requires larger datasets than the moving average method but yields a more accurate smoothing. Similar to the moving average method, this smoothing procedure will average the value of a data point by analyzing its k-nearest points. In the case of the LOESS method, for each point of the dataset, a local linear polynomial regression of one or two degrees is computed with their k-nearest points, by using a weighted least square giving more importance to data points near the analyzed initial point. The analyzed data point value is then corrected to the value predicted by the regression model. More information is available in [39]. In our case, the k-nearest points correspond to the IFPS of the 6 frames preceding and the 6 frames following the analyzed frame. The use of the k-nearest frames is possible because the characteristics of the scene between adjacent frames are spatially related. This can be generalized to most graphical applications because the movement of the camera is usually continuous.

4.3. Quantifying Scene Characteristics Equivalency. To further help the machine in the performance modeling, we transform the output format of the benchmark (IFPS in *terms of* the number of points, scene or grid size, and percentage of scene drawn), into a format that is more similar to the scene characteristics that will be given by the system designer (IFPS *in terms of* the number of points and scenes or grid size). Also, the tool can be more easily used if the percentages of scene drawn parameter could be omitted: to have to choose a percentage of scene drawn when querying the tool for predictions might lead to confusion. Thus, it is necessary to internally find a proportionality factor that can help evaluate the equivalency of performance between various points in the training data. The basic assumption is that the IFPS of a scene drawn with a certain set of characteristics (IFPS_1) will be somewhat equivalent to the IFPS of the same scene drawn with another set of characteristics (IFPS_2) if the characteristics of both scenes follow a certain proportionality. The first characteristic in this case is the size of the scene without accounting for depth: (v_1 for the first set of characteristics and v_2 for the other) either in OpenGL units or in the size of the grid of voxels or tiles in the case of tile-based applications. The other characteristic is the tile resolutions in each tile or voxel c_1 and c_2. As mentioned earlier, those concentrations and those sizes in the case of our benchmark are expressed as a per-dimension value. Thus, they are always equal in 2D (length and width). For simplicity, we use the following notation:

$$c_i = c^2_{\text{width}i} = c^2_{\text{depth}i}, \tag{1}$$

and

$$v_i = v^2_{\text{width}i} = v^2_{\text{depth}i}. \tag{2}$$

It implies that

$$\text{IFPS}_1 \approx \text{IFPS}_2 \Leftrightarrow \frac{c_1}{c_2} \propto \frac{v_1}{v_2}. \tag{3}$$

Considering that a scene drawn at a certain percentage p_1 with a set of characteristics, then v_1 represents the fraction of the total v_2 area that is drawn as

$$v_1 = p_1 * v_2. \tag{4}$$

In this case, since v_1 and v_2 are taken from the same scene, we have $c_1 = c_2$. Therefore,

$$\frac{v_1}{v_2} = p_1 * \frac{c_1}{c_2}, \tag{5}$$

where p_1 is the proportionality factor.

This example uses a single scene with a single set of characteristics to help find the proportionality factor, but the formula can also be used to compare scenes with different initial characteristics, which is a powerful metric for extrapolation. During the design of the typical use case of our tool, we assumed that the designer would want to request a performance estimation of the rendering of the scene when it is entirely drawn, and not just drawn at a certain percentage (worst-case scenario). A way had to be found to use the p_1 factor during the training phase, but to remove the need to use it in the performance queries, once the model is generated.

From equation (5), we obtain

$$p_1 * \frac{c_1}{v_1} = \frac{c_2}{v_2}. \tag{6}$$

Then, we found that the machine learning can create slightly more precise models if p_1 is expressed in terms of the concentration of triangles instead of in terms of the concentration of vertices. Considering that in our tile-based application the number of *facesPerTile* is obtained as follows:

$$\#facesPerTile = (\sqrt{c} - 1)^2 * 2. \tag{7}$$

From the proportionality function (6) and from (7), we deduce

$$p_1 * \frac{(\sqrt{c_1} - 1)^2 * 2}{v_1} = \frac{(\sqrt{c_2} - 1)^2 * 2}{v_2} = K. \tag{8}$$

The left part of equation (8) can be obtained by the scene characteristics, and the benchmark output performance metrics provide a value K which in turn allows to find values for c_2 and v_2. We are thus capable of obtaining approximately equal IFPS values between that scene rendered with c_2 and v_2 at 100% and the same scene drawn with c_1 and v_1 at p_1 percent. The machine learning algorithms are then trained with a vector containing a certain number of tuples (IFPS *in terms of K* and the number of points drawn) where

$$\#pointsDrawn = p_1 * \#totalNumberOfPoints, \tag{9}$$

and

$$\#totalNumberOfPoints = c_1 * v_1. \tag{10}$$

The designer can then create a prediction query by inputting K and the number of points he desires to render without having to mind about a percentage of scene drawn p_1. The tool would then output an IFPS prediction for the input parameters.

4.4. Identifying the Percentage of Space Parameters Evaluated. The best way to predict the performance would be to evaluate the rendering speed of the scene with every combination of characteristics. In this case, we would not even need to create nonparametric models, but this could require the evaluation of millions of possibilities, ending in way too long computation times (about a year). This performance prediction tool thus evaluates a very small subset of all those combinations in reasonable time (about half an hour). In the following, we determine the percentage of the number of combinations that our tool needs to evaluate.

Given:

(i) $n_{\text{screen}} = \{640 \times 480,\ 800 \times 600,\ 1024 \times 768,\ 1152 \times 864,\ 1920 \times 960\}$, the discrete number of studied screen sizes

(ii) $n_{\text{texture}} = \{x \mid x = 2^i \wedge 1 \le i \le 10 \wedge x \in \mathbb{N}\}$, the set of studied texture sizes

(iii) $n_{\text{vertices}} = \{x \mid 1 \le x \le 1{,}300{,}000 \wedge x \in \mathbb{N}\}$, the set of all possible quantity of points

(iv) $n_{\text{grid}} = \{x^2 \mid 1 \le x \le 45 \wedge x \in \mathbb{N}\}$, the set of studied tile grid sizes such that each tile has the same size in OpenGL coordinates

We generalized the concept of tile-based scenes for any dense scene by removing the tile resolution concept and replacing it by the concentration of any number of vertices lower than 1,300,000 divided by any grid size in n_{grid}. We chose this maximum number of vertices and also this maximum grid size arbitrarily as it should cover most data volumes in most of the hardware use cases. We can then evaluate the total number of combinations of characteristics influencing the density of points for every studied screen and texture sizes N_{Total} as

$$N_{\text{Total}} = |n_{\text{vertices}}| * |n_{\text{grid}}| * |n_{\text{screen}}| * |n_{\text{texture}}| = 2{,}925{,}000{,}000. \tag{11}$$

The tool then needs only to test a small fraction of all those combinations to produce adequate performance models. As mentioned earlier, the tool only needs to run two tests of the benchmark to construct adequate models for a fixed screen and texture size. Each test is configured initially to execute the benchmark with nine varying test parameters as shown in Table 1.

Given:

(i) N_{Tool}, the total number of combinations of characteristics analyzed by the tool

(ii) $S_{\text{TrainingSet}} = 5760$, the size of any training dataset output by the benchmark for a grid size and tile resolution test with fixed texture and screen size which corresponds to the number of frames rendered for both tests of the benchmark

We then suppose that each frame rendered during the benchmarking represents one unique combination of those billions and find the number of combinations tested by the tool N_{Tool} as

$$N_{\text{Tool}} = S_{\text{TrainingSet}} * n_{\text{screen}} * n_{\text{texture}} = P_{\text{TrainingSet}} * 288{,}000. \quad (12)$$

The tool is then guaranteed to train successful models by using only about 0.0098% of the total combinations of characteristics.

5. Prediction Error Evaluation

This section presents the experimental setup and also the experimental considerations used to validate the predictive power of the performance model.

5.1. Experimental Setup. We used a Nvidia QuadroFX570, a graphic card model which should have consistent performance with the current avionic hardware. Since OpenGL SC consists of a subset of the full OpenGL API's functions and that this subset of functions is very similar to the OpenGL ES 1.x specification, we worked around the absence of an OpenGL SC driver by using an OpenGL ES 1.x simulator which transforms the application's OpenGL ES 1.x function calls to the current installed drivers which is OpenGL. A meticulous care has been taken to make sure to use only functions available in the current OpenGL SC specification with the exception of one very important method, named *vertex buffer objects*.

The experiment begins in the training phase with the generation of the performance models as presented in Section 4. The prediction power of those models is then validated during the validation phase for which an industrial scene is benchmarked many times with varying characteristics. Those performances are then compared to the predictions generated by the models. The following sections explain this validation phase in detail.

5.2. Performance Model Validation. To validate the performance model created, we used a 3D scene from the World CDB representing a hilly location in Quebec, Canada. The scene was subsampled to various degrees to enable the validation of the model at various resolutions. The models were first validated for their interpolation predictive power by comparing the predictions with the validation scene rendered with characteristics similar to the ones used to train the models. The models were then validated for their extrapolation predictive power with the same method but by rendering the validation scene with characteristics untreated during the model training. It is also important to select well those characteristics in order to produce scenes that are not too easy to render. Because it is easier for the models to predict the maximum V-synced FPS, the validation scene characteristics should be selected in a way that makes the rendering operations generate various percentages of frames drawn with maximum IFPS. We analyzed the influence of having about 0%, 50%, or 100% of frames in the dataset drawn with maximum IFPS. As with the synthetic scenes, the whole validation scene is loaded in graphic memory prior to rendering the scene. Therefore, the loading time is not taken into account during the FPS calculation.

To validate a model, the benchmark is executed, but instead of displaying the usual synthetic scene, it renders the one from the World CDB subsampled according to the various parameters shown in Table 3. Figure 5 shows an example of the rendering of a CDB tile in our application at various resolutions. The output of the validation dataset is then smoothed in the same way as the training dataset. The size of each dataset is 5760 observations and is closely related to the number of frames produced by each run of the benchmarking tool from Section 3 (320 frames per run).

We then compare the smoothed instantaneous FPS of each frame to the predicted ones with the following metrics.

5.3. Metric Choice and Interpretation. The choice of a metric to evaluate the prediction errors is still subject to debate in the literature [40]. Especially in the case of models generated with machine learning, the error distribution are rarely normal-like and thus more than one metrics are commonly used to help understand and quantify the central tendency of prediction errors. We use the mean absolute prediction error MAE presented in equation (9) and also the rooted-mean-squared prediction error RMSE presented in equation (10). To conform to the literature, we also give the MAE in terms of relative prediction errors PE presented in equation (13). Because the error distributions will be most likely not normal, we also give the median error value which could yield the most significant central tendency. This last metric contribution to the understanding of the distribution is to indicate that 50% of the prediction errors are lesser than its value.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}\left|\widehat{\text{FPS}}_i - \text{FPS}_i\right|, \quad (13)$$

$$\text{RMSE} = \sqrt{\frac{1}{(n-1)}\sum_{i=1}^{n}\left(\widehat{\text{FPS}}_i - \text{FPS}_i\right)^2}, \quad (14)$$

$$\text{PE} = \frac{1}{n}\sum_{i=1}^{n}\frac{\left|\widehat{\text{FPS}}_i - \text{FPS}_i\right|}{\text{FPS}_i} * 100\%, \quad (15)$$

where $\widehat{\text{FPS}}_i$ is the i-th prediction, FPS_i is the i-th measured value, and n is the number of prediction/measurement pairs.

Table 3: Values of the World CDB scene characteristics.

Dataset	Name	Varied parameter	Tile resolution	Grid size	% of frames with maximum IFPS
Validation (CDB scene)	#1	Grid size	25	7; 9; 13; 17; 21; 25; 31; 37; 45	44.38
	#2	Grid size	17	7; 9; 13; 17; 21; 25; 31; 37; 45	44.51
	#3	Tile resolution	7; 9; 13; 17; 21; 25; 31; 37; 45	25	0%
	#4	Tile resolution	7; 9; 13; 17; 21; 25; 31; 37; 45	17	100%

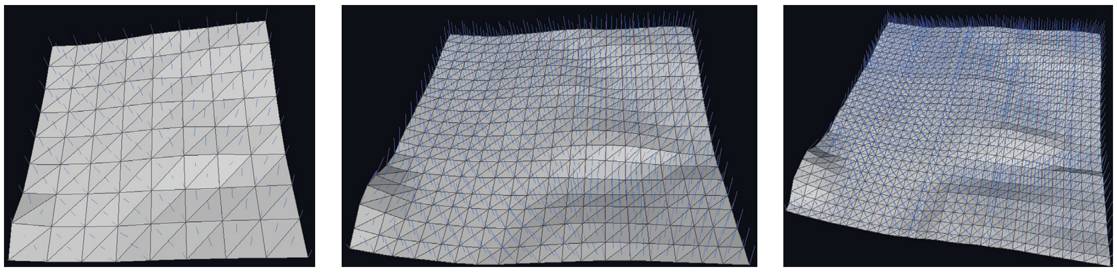
(a) (b) (c)

Figure 5: Mesh and normals of one tile of the validation scene sampled at a resolution of 9×9 (a), 19×19 (b), and 31×31 (c) shown before applying the randomized texture.

6. Results

Results (e.g., Figures 6–9) show that the prediction errors do not follow a normal distribution, but even though there is some skewness in them, they still retain a half-bell like appearance. The most adequate central tendency metric is thus the median. Furthermore, the artificial neural network has a better prediction than the two other algorithms most of the time, followed closely by random forest. Table 4 shows that the gap between the median prediction errors of both of these algorithms never exceeds 1 FPS. On the other hand, the MART method performed poorly on all datasets with a gap of up to about 12 FPS. Also, the performance models trained with artificial neural networks made quite good predictions in an interpolation and extrapolation context, as shown by the central tendencies of errors of all validation datasets confounded. Another explanation for the low performance of the tree-based methods comes from the fact that they do not perform well for extrapolation as mentioned in Section 2. The central tendency gap between those two sets never exceeds 4 FPS in this experiment. By analyzing the maximum absolute prediction error of most datasets, we see that there is a small chance that a very high prediction error is produced. These high errors can be as high as 43 FPS in the third dataset. Even though the general accuracy of the model is pretty good, the precision can be improved. Hopefully, the mode of each error distribution is always the first bin (range of values) of the histogram, which means that the highest probability of a prediction error is always the lowest error.

7. Discussion

Figure 10 somehow illustrates why parametric modeling would perform poorly as it would be hard to assume a geometric relationship between the IFPS and the predictors. Preliminary work also demonstrated the inefficiency of those methods, and thus they were not included in this work.

Because there is a very small chance (less than 1%) that a prediction might have a high error, the final prediction offered by the tool for a combination of characteristics should be a weighted average or a robust local regression of a small set of performance with similar scene characteristics, in order to help reduce the influence of these prediction outliers. Also, the scene used to train and validate our models are all dense, and thus our experiment cannot imply any significance for sparse scenes. But, as most graphical applications in an avionics context uses dense scenes, this should not be a major issue.

We also do not use fog effects in our tools which is a feature that could be used in a real industrial context as this feature will be part of next releases of OpenGL SC.

On the other hand, because of the high costs of avionics hardware, we had to abstract desktop graphic hardware behind an OpenGL ES 1.x environment to simulate an OpenGL SC environment which might weaken the correlation of our results to the ones that would be obtained with real avionics hardware. Related to this issue, we used standard desktop graphics hardware for the same reason. The presented method has been developed to abstract the nature of the hardware underlying the API, and the use of full-fledged avionics graphics hardware would improve the credibility of the results. However, this does not mean that our method would work for any use case of standard desktop graphics hardware. It has been designed for the specific problematic of the avionics industry: fixed graphics pipeline, use of OpenGL SC (or equivalent), and abstraction of the underlying hardware. This is fundamentally the opposite of the average desktop graphics hardware use case.

We also used only one validation scene subsampled into four distinct scenes. It could be of interest to reproduce

Table 4: Central tendencies of the prediction error distributions for each machine learning algorithm and for each validation dataset.

Validation dataset name	Supervised learning algorithm	RMSE (FPS)	Relative prediction error (%)	Mean absolute prediction error (FPS)	Median absolute prediction error (FPS)
	Random forest	2.71	2.12	1.36	0.45
#1	MART	9.37	10.15	6.85	4.86
	Neural net	2.29	1.99	1.26	0.50
	Random forest	10.87	8.74	5.85	2.06
#2	MART	16.53	17.65	12.16	10.39
	Neural net	7.90	8.50	5.16	3.51
	Random forest	5.94	5.98	4.10	2.79
#3	MART	9.99	10.60	7.51	6.15
	Neural net	2.97	3.03	2.39	2.13
	Random forest	15.80	9.12	10.94	3.98
#4	MART	30.20	18.78	22.53	15.02
	Neural net	4.02	2.58	3.10	3.01

Figures 6–9 present the error distribution of each nonparametric performance model for the four validation scenes.

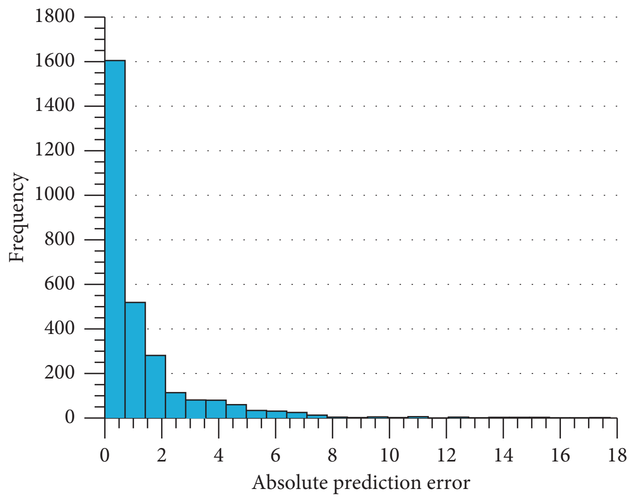

Figure 6: Prediction error distribution of the validation dataset #1 for the artificial neural network.

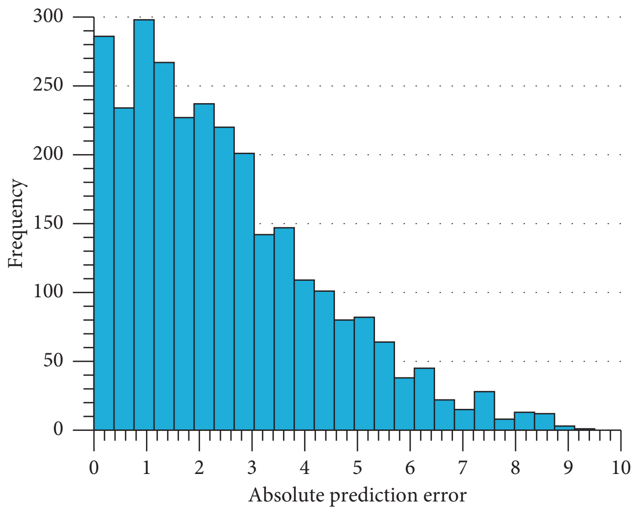

Figure 8: Prediction error distribution of the validation dataset #3 for the artificial neural network.

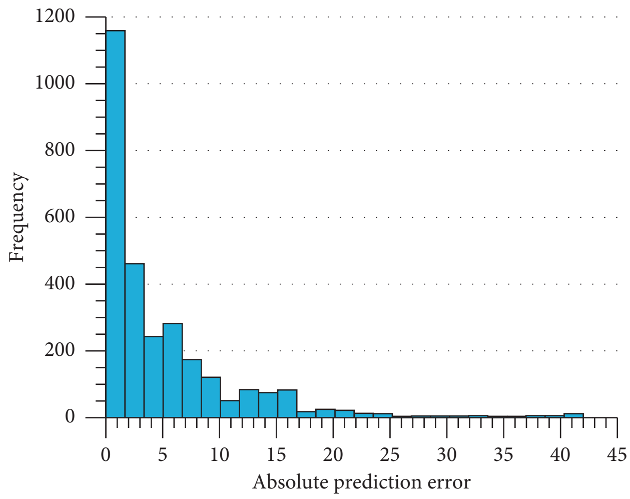

Figure 7: Prediction error distribution of the validation dataset #2 for the artificial neural network.

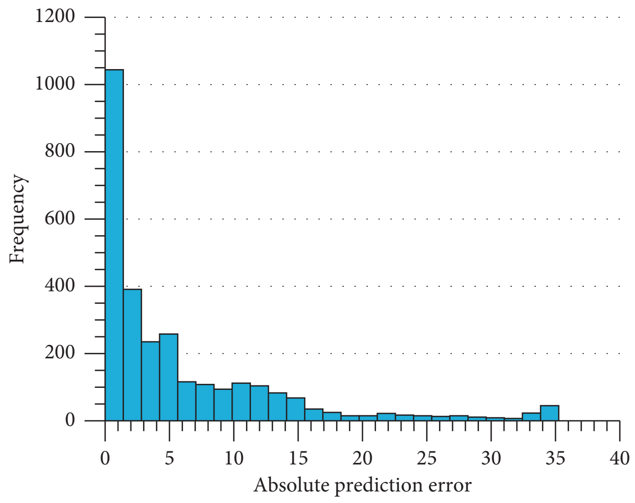

Figure 9: Prediction error distribution of the validation dataset #4 for the artificial neural network.

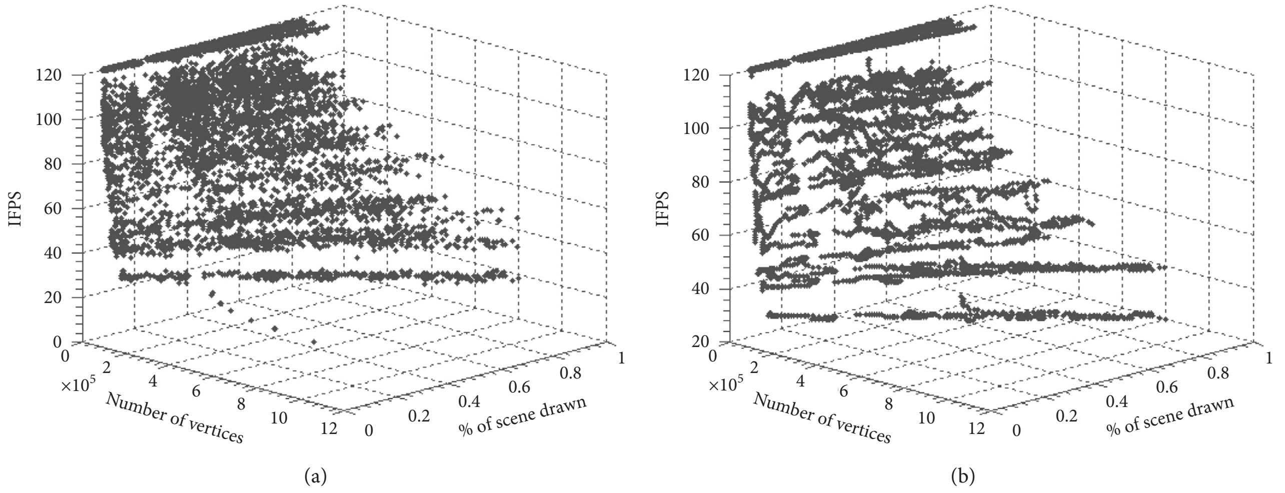

Figure 10: Comparison between unsmoothed (a) and smoothed (b) performance data.

the experiment with a bigger dataset of dense scenes. Considering our results as reproducible, the prediction errors made by our tool would be low enough for industrial use. Otherwise, we confirm that the central tendency of our prediction error distributions is similar to the ones presented in the literature, when these methods are applied to other kinds of hardware or software performance predictions.

8. Reproducing the Results

As our method includes two important experimental steps: dataset generation and machine learning, we foresee that the reader may want to reproduce experimentally either one or both of these experimental steps by using our dataset or by regenerating their own datasets to train the neural network.

To generate our training dataset, we used our procedurally generated 3D scenes. The reader may either want to create its own rendering tool following our specifications or we could make the C++ code available [41]. To generate the validation dataset, we rendered a private 3D scene, not available to the public, but any 3D scene consisting of a rendered height field could be used instead. We provide our raw datasets [41], more precisely the performance data generated by our tool for both 3D scenes, raw and unsmoothed.

Using our datasets or with a dataset generated by a third party, the actual training of performance models can be reproduced. The reader will have to smooth the data as described using the LOESS algorithm and use the Matlab Curve Fitting Toolbox to achieve this. Then, the reader will have to generate the performance models with the Neural Network ToolBox. Our Matlab code could also be made available [41].

9. Conclusion

We have presented a set of tools that enable performance benchmarking and prediction in an avionics context. These were missing or not offered in the literature. We believe that avionics system designers and architects could benefit from these tools as none other are available in the literature. Also, the performance prediction errors were shown to be reasonably low, thus demonstrating the efficacy of our method.

Future work will include the development of a performance-correct simulator for avionics graphics hardware and also the addition of other scene characteristics like fog effects or antialiasing in the performance models. Also, it is of interest to evaluate the possibility to enhance our modeling by using Bayesian networks, gradient boosting machines, and hybrid models made from machine learning. Finally, we plan to automate more our processes and then experiment different use cases and parameters. This will help us to determine with more precision the upper bound of the cost reduction.

Data Availability

The data that support the findings of this study not available in [41] can be obtained from the corresponding author upon reasonable request.

Conflicts of Interest

The authors declare that they have no conflicts of interests.

Acknowledgments

Special thanks are due to CAE Inc. for providing experimental material, industrial 3D scenes from the World CDB. We would also like to thank our other partners, contributors, and sponsors (CMC Electronics, CRIAQ, Mitacs, and NSERC) who particularly helped for the completion of two master theses [42, 43], certain results of which inspired this publication.

References

[1] L. J. Prinzel and L. J. Kramer, “Synthetic vision system,” US Patent LF99-1309, 2009, https://ntrs.nasa.gov/search.jsp?R=20090007635.

[2] M. Dutton and D. Keezer, "The challenges of graphics processing in the avionic industry," in *Proceedings of the 29th Digital Avionics Systems Conference*, Salt Lake City, UT, USA, October 2010.

[3] V. Hilderman, T. Baghai, L. Buckwalter et al., *Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware)*, Avionics Communication Inc., Leesburg, VA, USA, 2007.

[4] Nsight, Nvidia, https://developer.nvidia.com/nsight-graphics.

[5] PerfStudio, GPU, https://gpuopen.com/archive/gpu-perfstudio/.

[6] Standard Performance Evaluation Corporation, "What is this thing called "SPECviewperf®"? https://www.spec.org/gwpg/gpc.static/whatis_vp8.html.

[7] "ARINC-661," https://www.presagis.com/en/product/arinc-661/.

[8] Z. Wang, Y. Hui, and X. Zhou, "A simulation method of reconfigurable airborne display and control system," in *Proceedings of the First Symposium on Aviation Maintenance and Management-Volume 2*, pp. 255–263, Springer, Berlin, Germany, 2014.

[9] C. Haskins, *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities (ver. 3)*, International Council on Systems Engineering, San Diego, CA, USA, 2006.

[10] L. Lavagno, G. Martin, and L. Scheffer, *Electronic Design Automation for Integrated Circuits Handbook-2 Volume Set*, CRC Press, Boca Raton, FL, USA, 2006.

[11] *OpenGL SC Overview*, https://www.khronos.org/openglsc/.

[12] P. Cole, "OpenGL ES SC-open standard embedded graphics API for safety critical applications," in *Proceedings of the 24th Digital Avionics Systems Conference*, Washington, DC, USA, October 2005.

[13] "Basemark GPU1.1," https://www.basemark.com.

[14] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-M. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming-PPoPP '10*, Bangalore, India, January 2010.

[15] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, Austin, TX, USA, June 2009.

[16] K. Kanter, "Predicting AMD and Nvidia GPU performance," April 2011, https://www.realworldtech.com/amd-nvidia-gpu-performance/.

[17] Y. Zhang and J. Owens, "A quantitative performance analysis model for GPU architectures," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, San Antonio, TX, USA, February 2011.

[18] M. Boyer, J. Meng, and K. Kumaran, "Improving GPU performance prediction with data transfer modeling," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, Cambridge, MA, USA, May 2013.

[19] P. Joseph, K. Vaswani, and M. Thazhuthaveetil, "A predictive performance model for superscalar processors," in *Proceedings of the 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Orlando, FL, USA, December 2006.

[20] B. C. Lee and D. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems-ASPLOS-XII*, San Jose, CA, USA, October 2006.

[21] B. Lee and D. Brooks, "Illustrative design space studies with microarchitectural regression models," in *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, Scottsdale, AZ, USA, February 2007.

[22] G. Marin and J. Mellor-Crummy, "Cross-architecture performance predictions for scientific applications using parameterized models," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, June 2004.

[23] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ATI GPU: a statistical approach," in *Proceedings of the IEEE Sixth International Conference on Networking, Architecture, and Storage*, Dalian, Liaoning, China, July 2011.

[24] B. Li, L. Peng, and B. Ramadass, "Accurate and efficient processor performance prediction via regression tree based modeling," *Journal of Systems Architecture*, vol. 55, no. 10–12, pp. 457–467, 2009.

[25] S. Madougou, A. L. Vabanescu, C. D. Laat, and R. V. Nieuwpoort, "A tool for bottleneck analysis and performance prediction for GPU-accelerated applications," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Chicago, IL, USA, May 2016.

[26] E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. A. Doshi, and S. Abraham, "Using model trees for computer architecture performance analysis of software applications," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, San Jose, CA, USA, April 2007.

[27] R. M. Yoo, H. Lee, K. Chow, and H.-H. Lee, "Constructing a non-linear model with neural networks for workload characterization," in *Proceedings of the IEEE International Symposium on Workload Characterization*, San Jose, CA, USA, October 2006.

[28] D. Nemirovsky, T. Arkose, and N. Markovic, "A machine learning approach for performance prediction and scheduling on heterogeneous CPUs," in *Proceedings of the 29th International Symposium on Computer Architecture and High Performance Computing*, Campinas Sao Paulo, Brazil, October 2017.

[29] P. J. Joseph, K. Vaswani, and M. Thazhuthaveetil, "Construction and use of linear regression models for processor performance analysis," in *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, Austin, TX, USA, February 2006.

[30] E. İpek, S. A. Mckee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, San Jose, CA, USA, October 2006.

[31] E. Thereska and G. R. Ganger, "Ironmodel: robust performance models in the wild," in *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Annapolis, MD, USA, June 2008.

[32] S. Ren, Y. He, and K. S. Elnikety, "Exploiting processor heterogeneity in interactive services," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC)*, San Jose, CA, USA, June 2013.

[33] C. Stewart, T. Kelly, A. Zhang, and K. Shen, "A dollar from 15 cents: cross-platform management for internet services," in

Proceedings of the ATC'08 USENIX 2008 Annual Technical Conference, Boston, MA, USA, June 2008.

[34] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: automatic resource inference and allocation for Map Reduce environments," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, Karlsruhe, Germany, June 2011.

[35] G. Fatini and C. A. P. D. S. Martins, "A configurable and portable benchmark for 3D graphics," in *Proceedings of the XIV Brazilian Symposium on Computer Graphics and Image Processing*, Florianopolis, Brazil, October 2001.

[36] "Safety-Critical Profile Specification," March 2009, https://www.khronos.org/registry/OpenGL/specs/sc/sc_spec_1_0_1.pdf.

[37] B. Klug, *Right Ware Lauches Basemark ES 2.0 Taiji Free for Android, iOS*, December 2011, https://www.anandtech.com/show/5263/rightware-launches-basemark-es-20-taiji-free-for-android-ios.

[38] P. Sander, D. Nehab, and J. Barczak, "Fast triangle reordering for vertex locality and reduced overdraw," *ACM Transactions on Graphics*, vol. 26, no. 3, 2007.

[39] W. S. Cleveland and S. J. Devlin, "Locally weighted regression: an approach to regression analysis by local fitting," *Journal of the American Statistical Association*, vol. 83, no. 403, pp. 596–610, 1988.

[40] T. Chai and R. R. Draxler, "Root mean square error (RMSE) or mean absolute error (MAE)?-arguments against avoiding RMSE in the literature," *Geoscientific Model Development*, vol. 7, no. 3, pp. 1247–1250, 2014.

[41] https://github.com/guybois/HWPerfPredwithML.

[42] R.-G. Simon, "Prédiction de performance de matériel graphique dans un contexte avionique par apprentissage automatique," Masters thesis, École Polytechnique de Montréal, Montreal, Canada, 2015.

[43] L. Vincent, "Méthodologie expérimentale pour évaluer les caractéristiques des plateformes graphiques avioniques," Masters thesis, É Sters Thesisacte de MontratPe, Montreal, Canada, 2014.