Eurographics Symposium on Parallel Graphics and Visualization (2019)
H. Childs, S. Frey (Editors)

Statistical Analysis of Parallel Data Uploading using OpenGL

M. Wiedemann[1,2] and D.Kranzlmüller[2,1]

[1]Leibniz Supercomputing Centre, Bavarian Academy of Sciences, Germany
[2]MNM-Team, Ludwig-Maximilians-Universität, Germany

Abstract

Modern real-time visualizations of large-scale datasets require constant high frame rates while their datasets might exceed the available graphics memory. This requires sophisticated upload strategies from host memory to the memory of the graphics cards. A possible solution uses outsourcing of all data uploads onto concurrent threads and disconnecting prohibitive data dependencies. OpenGL provides a variety of functions and parameters but not all allow minimal interference on rendering. In this work, we present a thorough and statistically sound analysis of various effects introduced by choosing different input parameters, such as size, partitioning and number of threads for uploading, as well as combinations of buffer usage hints and uploading functions. This approach provides insight into the problem and offers a basis for future optimizations.

CCS Concepts

• ***Computing methodologies*** → ***Model development and analysis; Computer graphics; Parallel algorithms;***

1. Introduction

Graphics and visualisation require powerful hardware to deliver sufficiently good output for the users' needs. Thanks to ongoing development of graphics hardware driven by the entertainment industry, prices dropped and capabilities increased. This results in even more powerful graphics hardware with interesting features for rendering. Areas such as scientific visualization benefit from these possibilities by being able to visualize bigger datasets with more detail using commodity hardware. At the same time, target data sets from simulations and Big Data analytics grow even bigger and bigger. Yet, the increased amounts of data handled by graphics card still need to adhere to the imperative of hard deadlines for rendering, e.g. rendering individual frames in less than 16.7ms to achieve 60Hz frame refresh rates, to enable smooth interactions and to avoid cybersickness caused by lag. This disparity of growing datasets and hard real-time requirements calls for sophisticated approaches in visualization software.
One idea is to limit the amount of data needed for rendering. In most scenarios for visualizing large-scale datasets, not all data points need to be rendered in every frame. Consequently, the datasets can be partitioned by several criteria, eg. time or location dependent. This is especially important when considering that graphics card memory is only upgradeable by replacing the whole card. In the context of having datasets that do not fit completely into graphics memory and using partitioning, one must be able to exchange parts of the datasets on the fly. Our goal is to minimize the impact of this exchange on the rendering process by utilizing copy engines and parallel uploading strategies. In order to accomplish this, we developed a framework that disconnects data dependencies between rendering and uploading and to utilize parallel threads to concurrently upload data onto the graphics card. Building upon this, we are able to measure the impact of various input parameters on rendering and uploading, individually and in combination. To be able to minimize impact on rendering, we perform a thorough analysis and identify key parameters that can be used to fine tune uploading and can have a great impact on performance. This work provides a scientific foundation to the question of parallel data transfers in order to avoid relying on guidelines and estimates that are usually given (and "proven" by small experiments) in the computer graphics community.

The rest of this paper is organised as follows. Section 2 gives an overview on related work. Section 3 details our general framework and the evaluated input space. The methodology and statistical basis of our evaluation as well as the experimental setup is described in Section 4, followed by the presentation of its results in Section 5. Conclusions summarize our paper in Section 6.

2. Related Work

Visualisation of large-scale datasets need methods for partitioning the data in order to render it at high frame-rates. One possibility is to have certain Level-Of-Details (LODs), where each level represents different resolutions, i.e. how many triangles, of a mesh model. Having discrete LODs might introduce situations where one wants to exchange different LODs on the fly based on some evaluation function, e.g. [GKM93] or [FS93].
For large-scale datasets, a lot of work has been done for rendering large terrain datasets, e.g. [Paj98], [Hop98] , [LP02], [SW06],

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

DOI: 10.2312/pgv.20191114

[PG07]. [Joy09] gives a great overview on various techniques for visualising large-scale datasets.
There are several preliminary works that address host-device uploading strategies. Grottel et al. [GRE09] investigated four different methods using OpenGL to upload data from main memory to graphics card memory. Their evaluation gives the time needed for both rendering and uploading together and error bars that symbolize the variation of their measurements. Hrabcak and Masserann in the book OpenGL insights [HM12], show different strategies for host-device uploading and evaluate them with multi-threaded OpenGL. Their evaluation as well includes the mean time needed for both rendering and uploading and furthermore the mean impact on using multiple OpenGL contexts. Falk et al. [FGKR16] further investigated data uploading strategies of [GRE09]. They include in their evaluation the use of the newer OpenGL concept of buffer storages and asynchronous data transfers and give in their evaluation times needed for both rendering and upload. While all of those works give a great overview on the average impact on rendering, we want to deepen this analysis to give a more detailed view on what the key factors are and how they apply on rendering and uploading individually. Gómez-Luna et al. [GLiGLBG12] analyse CUDA streams and derive through benchmarking an analytical model to predict how many streams should be used depending on kernel execution and data transfer times. They verify their model using (adapted) example programs from the CUDA SDK. [FAN*13] researched different strategies for plain uploading of data using cuda and the open source driver for nvidia graphics cards. In their evaluation of different sized packages to be uploaded they could identify that either using DMA engines or directly writing to / reading from PCIe-address spaces proves to give the best performance, depending on the scenario. Van Werkhoven et al. [WMSB14] extend the approach of Gómez-Luna et al. to a more sophisticated analytical model for newer graphic cards.
Hoefler and Belli [HB15] give an extensive guide on how benchmarks of execution times on parallel systems should be designed and analysed. They focus on the high-performance computing field, but this work can easily be applied to heterogeneous applications, as it is the case for this work. Therefore, we follow their lead and designed our analysis of different OpenGL functions for parallel uploading data while rendering to adhere to statistical practises.
This also constitutes our contribution to the topic. As most computer tasks are subject to random noise, introduced by various effects ranging from system interrupts to passive cooling techniques, every measurement/benchmark is also subject to this random noise and needs to be analysed in a statistical sound way. Therefore, we present in this work a robust statistical analysis of parallel uploading methods using OpenGL. Our goal is to give a detailed view whether or not the different input parameters influence rendering and uploading times, each individually and in combination. From this, further research can be conducted that gives a detailed view on how those parameters change performance for different scenarios.

3. Concept

There exists a wide range of possibilities how rendering and uploading times can be prioritised for 3d graphic applications. Example requirements for parallel uploading data while rendering might be:

(1) Lowest rendering times, uploading times irrelevant
(2) Lowest uploading times, rendering times irrelevant
(3) Lowest combination of uploading and rendering times
(4) Lowest deviation in rendering times (predictable delay)
(5) Lowest deviation in uploading times (predictable delay)
(6) Lowest deviation in combination of rendering and uploading times (predictable delay)

This list shows boundary conditions on the vast possibilities that can constitue the inputs space. Gradual weights that fine tune priorities between the listed requirements are as well imaginable which inflate the input space enormously. Based on this list we implemented our own OpenGL based rendering framework that allows us to work with multiple threads for uploading data while rendering. This framework will be described in the following subsection. After that, we discuss the different input parameters whose impact will be analysed in this work.

3.1. Framework

In this work we want to analyse the influence of different input parameters on concurrent data uploading while rendering. For that we implemented a software framework that allows us to measure the times needed for rendering and uploading without one waiting for the other. To accomplish this, the framework is build around to the following requirements:

1. Rendering of arbitrary datasets (using rasterization approaches)
2. Upload in concurrent threads
3. Disconnect dependencies between rendering and uploading
4. Capability to vary the number of uploading threads
5. Exchange of functions used for uploading

To implement those requirements, the framework itself is structured in four different tasks that are each implemented using their own threads: **Rendering**, **interface**, **preparation** and **uploading**.

3.1.0.1. Rendering The rendering task renders the objects that are already present in graphics memory. When an upload of a newer object is finished, this worker uses them for the next frame and deletes the replaced objects.

3.1.0.2. Interface The interface task is for checking if any object needs to be updated, in other words, if a new time step (or LOD) is needed. For that it iterates over the rendered objects and extracts all necessary information for uploading and puts it into an uploading queue. This queue is designed to be ordered via priorities, but for the current use case the priority for each node is the same so that the queue is processed in a FIFO fashion.

3.1.0.3. Preparation The preparation worker has three tasks: Preparing the meta information needed for rendering (e.g. how the data is structured in memory), generating VBOs for the elements in the uploading queue, and to finish the uploading process. The latter means that the uploaded object is given to the **Rendering** worker to be used in the next frame. Generating VBOs is done via the OpenGL function `glGenerateBuffers`. After preparation, the elements are moved into a second queue, the process queue.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

3.1.0.4. Uploading The uploading worker can consist of 1 to n thread(s). They process the elements of the process queue in a FIFO fashion. Here, graphics memory is allocated and the host to device data transfer is executed. The used functions are modularly implemented and therefore can be exchanged easily.

3.2. Input Space

The general idea of this paper is to analyse the behaviour of different parameters for uploading data using OpenGL from host to device memory while rendering. Those parameters consist of buffer hints, the OpenGL functions used to copy the data and the properties of the data, i.e. size and partitioning. This input space is structured as follows:

- Dataset
  - Size
  - Partitioning
- Used functions
  - OpenGL functions
  - Buffer hints
- Number of threads for uploading

Varying sizes and numbers of partitions lies in the nature of 3D rendering. 3D models have their individual number of vertices and vertex properties, depending on how detailed they are and what kind of properties are attached to them. For that we vary the size of the tested datasets as well as the number of partitions. This also sheds light onto the question if having one big block of data containing all vertex properties, e.g. position, colour, texture coordinates, gives better performance for rendering or uploading as well as on the influence of the dataset size.
The used functions are highly driver and hardware dependent. Here the idea is to analyse if there are differences in the implementation of a used function and if the hints specified while creating a buffer result in different performance numbers.
The number of threads that are used concurrently to upload data blocks allows to investigate whether there are dedicated copy engines on the graphics card and if and how they are used.

4. Methodology

In the following subsection, the structure of the evaluation of the conducted experiments will be described. It is followed by the experimental setup.

4.1. Structure of Evaluation

In contrast to preceding works, we will rely on statistical methods for the analysis of the measured data. Following the guidelines of [HB15] we verified using the Shapiro-Wilk test that our measurements are not normal distributed. This can also be deducted from a theoretical point of view: There exists a threshold how fast the data can be uploaded. One upper bound to this is the peak transfer rate of the PCIe bus. Assuming there are effects causing random noise that affecting the measurements, those effects can only add to the measured time and never subtract, so there can't be a normal distribution of our measurements. Possible effects for random noise might consist of race conditions, the asynchronous nature of a graphics card, scheduling of CPU and GPU processes as well as system interrupts. Obviously, this is by far not a complete list of all possible effects causing delays for either rendering or uploading data.
Following the central limit theorem, which implies that increasing the number of sample sets (each consisting of several samples) from a random distribution leads to the means of the sample sets being normally distributed, we chose to sample 100 sample sets, each consisting of at least 30 measurements. Again we used the Shapiro-Wilk test and can conclude that not all of our measurements fulfil the normality requirement. Furthermore, the variance of our measurements is not constant among all the measurements. Therefore, for further analysis we use quantile regression which does not require a normal distribution or a constant variance.

Our hypotheses for the experiments are:

- The number of threads has an impact
- The combination buffer hint with chosen method impacts uploading/rendering times
- Partitioning of datasets influences the uploading/rendering times
- Size influences the uploading/rendering times

In order to be able to evaluate our hypotheses we designed in total 3240 different experiments, each consisting of at least 30 uploads and 30 renderings in parallel and individually. This means, for the parallel experiments, the exact number of either uploads or renderings might be higher, depending on how long the respective other action takes. For the individual cases, where we either uploaded data or rendered data, each experiment consisted of exact 30 iterations. We conducted an exhaustive search within certain limits of the described input space. This means, that in each of the experiments, we only changed one of the following variables while keeping the others unchanged:

- Buffer hint
- Function used
- Usage of PBOs
- Size of dataset
- Number of partitions
- Number of threads (for uploading)

Each of those experiments was conducted 100 times.

For the following analysis, we define a 0.05 significance level for rejecting the null hypotheses, meaning if the p-value of the quantile regression is less or equal to 0.05, we can reject the null hypotheses and can reject that this input variable has no impact on the looked at process. In other words, if the p-value is less or equal to 0.05, we assume with 95% confidence that this input parameter has an impact on rendering or uploading, respectively, with the shown change-rates. As we calculate the quantile regression for 10 different quantiles, we can only reject the null hypotheses if the p-value for every quantile is below our significance level of 0.05.

4.2. Experimental Setup

In order to test a wide range of input parameters the following experimental setup was used:

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

For the dataset, we chose to artificially create one in order to be able to fine tune GPU and PCIe bus usage. For that, we divide a plane forming an OpenGL fullscreen quad, i.e. spanning from $(-1,-1,-0.5)$ to $(1,1,-0.5)$, into 50×50 quads, with each quad consisting of two triangles. This means, one plane consists of $50 \times 50 \times 2 \times 3 = 15000$ vertices. The triangles of this plane are copied as needed to fill a dataset with a predetermined size. In order to analyse the impact of partitioning a dataset, we copy the dataset by the number of partitions and reuse it. By deactivating depth tests, we force the rendering pipeline to render each triangle and each generated fragment and consequently are able to maintain a high GPU usage without optimizations.
The reasoning for the partitioning is that we are bound to having a graphics cards with 2 copy engines and a CPU that has 4 cores (and 4 threads). We want to investigate what effects are caused by parallelising uploads and making use of both copy engines, but also want to see if there are possible overlaps of CPU time by having 3 parallel copies when using enough threads and partitions.
Using different functions for uploading is bound to the

For sizes we used the factors $s \in 1,2,4$. The sizes are determined by the following formula:

$$Size = factor \cdot 4 \cdot 3 \cdot 85 \cdot 1024 \quad (1)$$

The 4 in this case stands for bytes per float, the 3 for floats per vertex and the 85 to get near to 1024 in order to have roughly $1024 \cdot 1024$ bytes, i.e. 1MiB. For partitioning we split the sizes by a factor $p \in 1,2,4,8,16$. This means, if we have a dataset of roughly 4MiB, it can consist of one 4MiB block or up to 16 block which have the size of 255kiB. The reasoning for the partitioning is that we are bound to having a graphics cards with 2 copy engines and a CPU that has 4 cores (and 4 threads). As we wanted to investigate what effects are caused by parallelising uploads and making use of both copy engines, but also want to see if there are possible overlaps of CPU time by having 3 parallel copies when using enough threads and partitions, we chose to use one to 16 partitions to cover the whole range.
The used functions depend on buffer hints that describe the planned usage of a VBO and consist of mainly two parts: the frequency of usage and the nature of access. Frequency can be one of three: Stream - modified once and used at most a few times- , static - modified once, used many times- and dynamic - modified repeatedly and used many times. Nature of access describes how the buffer is accessed: Draw - modified by host and used by the device-, read - modified by the device and read from the host (i.e. download)- , copy - modified by the device and used by the device. Any combination of one of frequency and one from nature of access is possible, so in total there are the following nine possibilities as buffer hints written as OpenGL enums:

- GL_STREAM_DRAW
- GL_STATIC_DRAW
- GL_DYNAMIC_DRAW
- GL_STREAM_COPY
- GL_STATIC_COPY
- GL_DYNAMIC_COPY
- GL_STREAM_READ
- GL_STATIC_READ
- GL_DYNAMIC_READ

For the actual uploading, we have implemented four different methods using the OpenGL functions `glBufferData`, `glBufferSubData`, `glMapBuffer` and `glMapBufferRange`. The pseudo code for those is given in Algorithm 1.

Algorithm 1 Used Uploading Functions

```
procedure UPLOADFUNCTION1(target, size, data, hint)
    glBufferData(target, size, data, hint);
procedure UPLOADFUNCTION2(target, size, data, hint)
    glBufferData(target, size, 0, hint)
    ptr = glMapBufferRange(target, 0, size, mapRangeBit)
    memcpy(ptr, data, size)
    glUnmapBuffer(target)
procedure UPLOADFUNCTION3(target, size, data, hint)
    glBufferData(target, size, 0, hint)
    ptr = glMapBuffer(target, 0, size, mapBit)
    memcpy(ptr, data, size)
    glUnmapBuffer(target)
procedure UPLOADFUNCTION4(target, size, data, hint)
    glBufferData(target, size, 0, hint);
    glBufferSubData(target, size, data, hint);
```

In Algorithm 1 the following input parameters are used:

- **target** may be either GL_ARRAY_BUFFER or GL_PIXEL_UNPACK_BUFFER
- **size** corresponds to the size in bytes which will be uploaded
- **data** refers to the actual data being uploaded
- **hint** is one of the before described buffer hints

Additionally, **mapRangeBit** stands for GL_MAP_WRITE_BIT and **mapBit** for GL_WRITE_ONLY.
All four methods have in common that they use `glBufferData` to allocate graphics memory, which is only initialised by uploadfunction4 where the data is directly uploaded. In all other cases, different functions are used to transfer the data to device memory. As both `glBufferData` and `glBufferSubData` are asynchronous functions, we use OpenGL SyncObjects (glSync) to determine when an upload is finished for uploadfunction3 and uploadfunction4. Please note that this means if there is a draw call submitted from a different thread inbetween the uploading call and the creation of the glSync object, the time measured for uploading will longer by the time needed for that rendering call. For uploadfuntion1 and uploadfunction2 we do not need to use a glSync object, as the memcpy functions blocks until it is finished.
Furthermore, we vary the number of uploading threads described in section 3.1 from 1 to 3.

For the experiments we used Ubuntu 16.04.5 as operating system and an Intel Xeon E5-1607 CPU with 3.10GHz and 64GB DDR3 main memory working with 2133 MHz. As graphics card we used a MSI Geforce GTX 1080 Gaming X 8G with the proprietary nvidia driver in version 410.79. We deactivated Threaded Optimizations as this lead to deadlocks within our application. The rendering was done into an off screen framebuffer with the size of 1920x1080 Pixels, four 32bit float colour channels and

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

a 24bit renderbuffer in order to avoid any visibility influences. Furthermore, we used OpenGL Core Profile in version 4.5, GLX 1.4 and X.Org X Server 1.18.4 with X Protocol Version 11, Revision 0. For context creation we use GLFW 3.1.2 from the Ubuntu repository.

5. Results

In total we gained 23,245,904 observations for rendering and 17,278,237 for uploading in parallel. For the individual scenarios we have exactly 9,720,000 measurements. As there are some extreme outliers, we chose to use the R function `boxplots.stats` with a coefficient of 50 to identify and remove those extreme outliers. This outlier removal is done as follows:
By default, the width of the box of a boxplot in R is defined as the size between the 75th and 25th percentile of the data. In our case, we chose to reject every measurement that is 50 times this width farther away from the 75th or 25th percentile. In total we removed 0.46% of the measurements for the only uploading case, 1.57% for the uploading and 1.07% for the rendering measurements in the parallel scenario and 0.22% of the plain rendering scenario.
To further reduce the number of measurements and to make the following analysis more robust, we calculated for each of the 100 samples the median of the 30 (or more) measurements and used that for the further evaluation. This resulted in having 3,240,000 observations for each rendering and uploading, a sample size of 100 for all single experiments and to have a more robust data basis with respect to outliers caused by different effects not part of the rendering/uploading pipeline. As input formula to model the data using quantile regression we use the following:

$$\Delta_{\text{time}} = mode * hint + threads + n_{\text{partitions}} + s \quad (2)$$

Here, Δ_{time} stands for time needed to upload/render in microseconds. $n_{\text{partitions}}$ stands for number of partitions. *mode*, *hint* and *threads* are categorical variables and treated as factors for the quantile regression. They stand for functions used, buffer hint specified and number of threads, respectively. The mapping from mode to function and hint to the used buffer hint are shown in table 1. *threads* is defined as $threads_1$ for one thread, $threads_2$ for two threads and $threads_3$ for using three threads. s stands for the size in MiB of the dataset.
This results in the following model formula:

$$\begin{aligned}\Delta_{\text{time}} = b_0 + &\sum_{i=1,j=1}^{i=7,j=8} \left(b_{11+8\cdot i+j} \cdot mode_i \cdot hint_j\right) + \\ &+ \sum_{i=1}^{i=7} (b_{4+i} \cdot mode_i) + \sum_{j=1}^{j=8} \left(b_{4+7+j} \cdot hint_j\right) \\ &+ b_1 \cdot threads_2 + b_2 \cdot threads_3 + \\ &+ b_3 \cdot n_{\text{partitions}} + b_4 \cdot s\end{aligned} \quad (3)$$

Here, b_0 stands for the intercept, meaning every quantitative input variable is set to zero and for the categorical variables, $mode_0$, $hint_0$ and $threads_1$ is chosen. The remaining b_l parameters stand for the change-rate which in our case is variable and depending on the quantiles. For all categorical variables, i.e. $mode_X$, $hint_X$ and $threads_X$, they are either 0 or 1 in case they are not or are present, respectively. This means, if $mode_0$, $hint_0$ and $threads_1$ is chosen, the formula can be simplified to:

$$\Delta_{\text{time}} = b_0 + b_3 \cdot n_{\text{partitions}} + b_4 \cdot s. \quad (4)$$

For having $mode_1$, $hint_1$ and $threads_2$, it changes to:

$$\Delta_{\text{time}} = b_0 + b_{20} + b_5 + b_{12} + b_2 + b_3 \cdot n_{\text{partitions}} + b_4 \cdot s. \quad (5)$$

Please note that in the second example, all three sums result in a b_l parameter, as the mode is different from $mode_0$, the hint different from $hint_0$, and both in combination are different from $mode_0$ and $hint_0$.

$mode_0$	⇒	UploadFunction1
$mode_1$	⇒	UploadFunction2
$mode_2$	⇒	UploadFunction3
$mode_3$	⇒	UploadFunction4
$mode_4$	⇒	UploadFunction1 + PBO
$mode_5$	⇒	UploadFunction2 + PBO
$mode_6$	⇒	UploadFunction3 + PBO
$mode_7$	⇒	UploadFunction4 + PBO
$hint_0$	⇒	GL_STREAM_DRAW
$hint_1$	⇒	GL_STATIC_DRAW
$hint_2$	⇒	GL_DYNAMIC_DRAW
$hint_3$	⇒	GL_STREAM_COPY
$hint_4$	⇒	GL_STATIC_COPY
$hint_5$	⇒	GL_DYNAMIC_COPY
$hint_6$	⇒	GL_STREAM_READ
$hint_7$	⇒	GL_STATIC_READ
$hint_8$	⇒	GL_DYNAMIC_READ

Table 1: *Mapping of mode and hint to used method and to used driver hint.*

We divide the analysis in the following into three distinct subsections. First we investigate the impact of the input parameters on plain rendering, meaning there is no uploading happening in parallel. As second we look at plain uploading without any rendering. In the third part, we evaluate the parallel rendering while also uploading data.

5.1. Plain Rendering

For the plain rendering scenario, the whole framework introduced in section 3.1 is used but no updates of meshes are planned and consequently, the uploading worker threads are idle.
We start by analysing the p-values. For all 10 quantiles, the p-value of the intercept is below the significance value and therefore is with 95% confidence the time needed for rendering no data using $mode_0$, $hint_0$ and one thread is different to 0. This states the time needed to use the framework.
Table 2 shows if the combination of a specific *mode* with a specific *hint* makes a difference for the time needed for rendering (symbolised by *). We see that we have not evidence that $mode_4$ has any

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

*	$mode_0$	$mode_1$	$mode_2$	$mode_3$	$mode_4$	$mode_5$	$mode_6$	$mode_7$
$hint_0$	*	*	*	.	.	*	*	.
$hint_1$	.	.	.	.	.	.	.	.
$hint_2$	.	.	.	.	.	.	.	.
$hint_3$	.	.	.	.	.	*	.	.
$hint_4$	.	.	.	.	.	.	.	.
$hint_5$	.	.	.	.	.	*	.	.
$hint_6$	.	*	*	*	.	*	*	*
$hint_7$	.	.	.	.	.	.	.	.
$hint_6$	.	*	*	*	.	*	*	*

Table 2: *Significance for all ten quantiles for different modes in combination with a specific buffer hint for plain rendering. "*" stands for having an impact and "·" for there is no evidence, that choosing this parameter has an impact*

Rendering	2 threads	3 threads	partitions	size
Significance	·	*	*	*

Table 3: *Significance for all ten quantiles threads, partitioning and size for plain rendering. "*" stands for having an impact and "·" for there is no evidence, that choosing this parameter has an impact*

impact on rendering times. This means that there is no evidence that using or not using a Pixel Buffer Object in combination with the function `glBufferData` for copying data from host to device memory is changing the needed time for rendering. This is also true for choosing either $hint_1$, $hint_2$, $hint_4$ or $hint_7$. Please note that in table 2 $mode_0$ together with $hint_0$ constitutes the intercept and the first row and first column describe the significance of the impact of modes and hints individually.
Further, as in table 3 illustrated, we found evidence that using 3 threads, changing the number of partitions(submeshes) or the size has an impact on rendering times.

5.2. Plain Uploading

For the uploading scenario, only data is exchanged and nothing rendered. Table 4 shows if the p-values for all 10 quantiles are below the significance level for the different combination of *modes* and *hints*. Again, $mode_0$ together with $hint_0$ constitutes the intercept as well as the first row and column show the significance of modes and hints individually. The picture here is quite different. Except for $mode_4$, which is the same function as $mode_0$ but with usage of PBOs, choosing a different *mode* has an impact on uploading times. The same yields for choosing a *hint*. Here only $hint_2$ has no evidence of changing the result.

For the remaining input parameters, choosing two or three threads, as well as changing the size and number of submeshes, evidence suggests that there is a significant impact on uploading times, see also table 5.
Exemplarily, we found for the examined experimental setup that choosing a different *mode-hint* combination can have a quite large impact on the time needed for uploading data. For example, the combination of choosing $mode_1$ together with $hint_1$ results (with significant evidence) in lower uploading times, in best case scenarios this can lower them by over 4000 microseconds. Please note, that for a full prediction of change on uploading times, one would need to include the values from the individual parameters for modes and hints, i.e. the ones represented in the first row and first column.

*	$mode_0$	$mode_1$	$mode_2$	$mode_3$	$mode_4$	$mode_5$	$mode_6$	$mode_7$
$hint_0$	*	*	*	.	.	*	*	.
$hint_1$	.	*	*	.	.	*	*	.
$hint_2$	.	.	.	.	.	.	.	.
$hint_3$	.	*	*	*	.	*	*	*
$hint_4$	.	*	*	.	.	*	*	.
$hint_5$	.	*	*	*	.	*	*	*
$hint_6$	.	.	.	*	.	.	..	*
$hint_7$	.	*	*	.	.	*	*	.
$hint_8$	.	.	.	*	.	o	o	*

Table 4: *Significance for all ten quantiles for different modes in combination with a specific buffer hint for plain uploading. "*" stands for having an impact and "·" for there is no evidence, that choosing this parameter has an impact*

Upload	2 threads	3 threads	partitions	size
Significance	*	*	*	*

Table 5: *Significance for all ten quantiles threads, partitioning and size for plain uploading. "*" stands for having an impact and "·" for there is no evidence, that choosing this parameter has an impact*

5.3. Rendering while Uploading

For the parallel scenario, meaning the GPU is busy rendering a dataset, while another dataset is uploaded (which will replace the rendered dataset once the upload is finished), we again take a look on the p-values of our quantile regression model. We start with the *mode-hint* combinations for the rendering times, see table 6. Here, only choosing a combination using $hint_2$ or $mode_4$ has no evidence supporting that it changes the time needed for rendering. For the remaining parameters, using two or three threads, changing the number of partitions and the size are also well supported to change the resulting rendering times, see also table 7.

For the uploading, again, we have a quite different picture, see table 8 and table 9. Only the following combinations have evidence suggesting they impact uploading times: Either $mode_1$, $mode_2$, $mode_5$ or $mode_6$ together with $hint_0$, $hint_3$, or $hint_5$. The remaining function-hint combinations are not supported to have an influence on the uploading times. Changing the number of threads, partitions and the size however, are within 95% confidence resulting in higher

*	$mode_0$	$mode_1$	$mode_2$	$mode_3$	$mode_4$	$mode_5$	$mode_6$	$mode_7$
$hint_0$	*	*	*	.	.	*	*	.
$hint_1$	.	*	.	.	.	*	*	.
$hint_2$	.	.	.	.	.	.	.	.
$hint_3$	.	*	*	.	.	*	*	.
$hint_4$	.	*	*	.	.	*	*	.
$hint_5$	.	*	*	.	.	*	*	.
$hint_6$	.	*	*	*	.	*	*	*
$hint_7$	.	*	.	.	.	.	*	.
$hint_8$	.	*	*	*	.	*	*	*

Table 6: *Significance for all ten quantiles for different modes in combination with a specific buffer hint for parallel scenario o rendering times. "*" stands for having an impact and "·" for there is no evidence, that choosing this parameter has an impact*

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

Rendering	2 threads	3 threads	partitions	size
Significance	*	*	*	*

Table 7: *Significance for all ten quantiles threads, partitioning an size for parallel scenario - rendering times. "*" stands for having an impact and "·" for there is no evidence, that choosing this parameter has an impact*

*	$mode_0$	$mode_1$	$mode_2$	$mode_3$	$mode_4$	$mode_5$	$mode_6$	$mode_7$
$hint_0$	*	*	*	·	·	*	*	·
$hint_1$	·	·	·	·	·	·	·	·
$hint_2$	·	·	·	·	·	·	·	·
$hint_3$	·	*	*	·	·	*	*	·
$hint_4$	·	·	·	·	·	·	·	·
$hint_5$	·	*	*	·	·	*	*	·
$hint_6$	·	·	·	·	·	·	·	·
$hint_7$	·	·	·	·	·	·	·	·
$hint_8$	·	·	·	·	·	·	·	·

Table 8: *Significance for all ten quantiles for different modes in combination with a specific buffer hint for parallel scenario - upload times. "*" stands for having an impact and "·" for there is no evidence, that choosing this parameter has an impact*

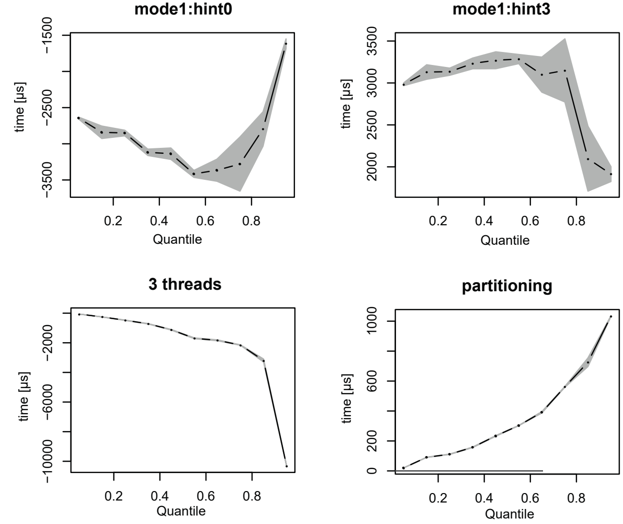

Figure 1: *Change-rate in microseconds vs. quantiles for different parameters in the parallel scenario - uploading times. 95% confidence intervals are illustrated as grey bands around the estimated change-rate (black points) and the interpolation between them (dashed lines).*

or lower times needed. Exemplarily for this particular test setup, see fig. 1 for getting an idea how great this impact can be. For example, choosing to use 3 threads instead of one for uploading can improve the uploading times between 0 and 10ms. Considering a frame refresh rate of 60Hz, this means that by choosing only one thread instead of three can, in the worst case, slow down uploading by more than half the time available to render one frame. Another example is changing the used driver hint that can worsen or improve uploading times by over 3ms. Please note, that again, for a complete prediction, one would also need to include the parameters for mode and hint individually. Additionally, increasing the number of partitions can result in an increase of up to 1ms for uploading.

To summarize our experiments of all three scenarios, we see that using three threads instead of one for uploading, changing the size of the dataset or the number of the partitions have an impact on the time needed for executing the particular task. In this case, that impact can mean that uploading times are 5ms longer or shorter, just depending on the used function and driver hint combination. Other changes can lie within a couple hundred microseconds and might have only marginal impact on the resulting times.

6. Conclusion

In this work we have implemented a general framework that enabled us to perform a thorough and statistical sound evaluation of using different parameters for uploading data from main memory to graphics memory using OpenGL while simultaneously rendering. The presented results show evidence supporting that changing the input parameters alter the corresponding time needed for uploading, rendering or both which confirm our hypotheses. By disconnecting rendering from the uploading process, we are able to individually evaluate the influence of the examined input parameters on either of those processes. Treating uploading and rendering in isolation or running both in parallel result each in their own distinct combination of parameters that have a performance impact. Furthermore, in the parallel scenario, choosing one set of parameters can evidently have an impact on, for example, uploading (or rendering) while there is no support for altering the timings of the respective other. This means that depending on the scenario and priorities for either of those processes, different performance tuning strategies might be necessary.
This also implicates future work: While we have evidence that there are changes in timings when altering parameters, we do not have explicit numbers on what the change looks like for different systems. For that we need to broaden the experimental setup and perform experiments with a range of different graphics cards, drivers, operating systems and computing systems to extract a general model describing parallel uploading mechanisms.

Upload	2 threads	3 threads	partitions	size
Significance	*	*	*	*

Table 9: *Significance for all ten quantiles threads, partitioning an size for parallel scenario - upload times. "*" stands for having an impact and "·" for there is no evidence, that choosing this parameter has an impact*

Acknowledgement

The authors thank Matthias Maiterth for his valuable input through various discussions and feedback.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

References

[FAN*13] FUJII Y., AZUMI T., NISHIO N., KATO S., EDAHIRO M.: Data transfer matters for gpu computing. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on* (2013), IEEE, pp. 275–282. 2

[FGKR16] FALK M., GROTTEL S., KRONE M., REINA G.: Interactive gpu-based visualization of large dynamic particle data. *Synthesis Lectures on Visualization 4*, 3 (2016), 1–121. 2

[FS93] FUNKHOUSER T. A., SÉQUIN C. H.: Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (1993), ACM, pp. 247–254. 1

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (1993), ACM, pp. 231–238. 1

[GLiGLBG12] GÓMEZ-LUNA J., I. GONZÁLEZ-LINARES J. M., BENAVIDES J. I., GUIL N.: Performance models for cuda streams on nvidia geforce series. *J. Parallel Distrib. Comput. 72*, 9 (2012), 1117 – 1126. 2

[GRE09] GROTTEL S., REINA G., ERTL T.: Optimized data transfer for time-dependent, gpu-based glyphs. In *Visualization Symposium, 2009. PacificVis' 09. IEEE Pacific* (2009), IEEE, pp. 65–72. 2

[HB15] HOEFLER T., BELLI R.: Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis* (2015), ACM, p. 73. 2, 3

[HM12] HRABCAK L., MASSERANN A.: Asynchronous buffer transfers. In *OpenGL Insights*, Cozzi P., Riccio C., (Eds.). CRC press, 2012. 2

[Hop98] HOPPE H.: Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Visualization'98. Proceedings* (1998), IEEE, pp. 35–42. 1

[Joy09] JOY K. I.: Massive data visualization: a survey. *Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration* (2009), 285–302. 2

[LP02] LINDSTROM P., PASCUCCI V.: Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer graphics 8*, 3 (2002), 239–254. 1

[Paj98] PAJAROLA R.: Large scale terrain visualization using the restricted quadtree triangulation. In *Visualization'98. Proceedings* (1998), IEEE, pp. 19–26. 1

[PG07] PAJAROLA R., GOBBETTI E.: Survey of semi-regular multiresolution models for interactive terrain rendering. *The Visual Computer 23*, 8 (2007), 583–605. URL: http://dx.doi.org/10.1007/s00371-007-0163-2, doi:10.1007/s00371-007-0163-2. 2

[SW06] SCHNEIDER J., WESTERMANN R.: Gpu-friendly high-quality terrain rendering. *WSCG 2006 International Programme Committee* (2006). 1

[WMSB14] WERKHOVEN B. V., MAASSEN J., SEINSTRA F. J., BAL H. E.: Performance models for CPU-GPU data transfers. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on* (2014), IEEE, pp. 11 – 20. 2

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.