Efficient Rendering of Digital Twins Consisting of Both Static And Dynamic Data

Aleksandar Atanasov, Benedikt Kottler, Dimitri Bulatov

Fraunhofer Institute of Optronics, System Technologies and Image Exploitation (IOSB),
Gutleuthausstrasse 1, 76275 Ettlingen, Germany
(aleksandar.atanasov, benedikt.kottler, dimitri.bulatov)@iosb.fraunhofer.de

Keywords: Digital Twin, Rendering, GPU, Urban Morphology, Memory Management, Vegetation

Abstract

The high number of vertices and faces present in a digital twin of a large scene poses a challenge when executing large-scale simulations that require the data to be rendered multiple times with slight changes such as the light conditions in a day-night cycle during a long period of time (weeks and even months). Adding urban morphology such as tree planting introduces a new layer of complexity due to the requirement for evaluating different tree configurations and their effect on the local environment in the context of urban heat islands. Using advanced rendering techniques such as splitting a single buffer into sub-regions with inplace updates as well as buffer orphaning multiple methodologies are evaluated.

1. Introduction

Urban Heat Island (UHI) effects, which can raise temperatures by up to 12°C due to factors like dark surfaces and scant greenery, necessitate mitigation strategies in urban planning such as the use of sustainable materials, the creation of green spaces, and smart design (Osmond and Sharifi, 2017, Williams et al., 2012, Kjellstrom et al., 2009). Furthermore, urban planners utilize year-long simulations on digital twins to assess and optimize urban designs, a process complicated by the high complexity of large scenes and the need for frequent renderings to capture light changes over extended periods. Tree planting is particularly effective against Urban Heat Islands (UHIs) because trees provide shade, store heat efficiently, enhance the city’s visual appeal, and make it more livable (Wong and Yu, 2005).

While there are many definitions for a digital twin, in the context of remote sensing and photogrammetry the term can be summarized as the virtual depiction of a physical model. From the raw sensor data, such as airborne LiDAR or photgrammetric point cloud, digital surface (DSM) or terrain model (DTM) are calculated; Land cover classification is performed; Building and tree instances at a relevant level of detail are retrieved. Additional data is added and removed according to the scenario that the twin is covering. Trees to be virtually planted represent a good example of such a configuration.

The underlying 3D data is usually a large set of meshes (vegetation, buildings, vehicles etc.) that has to be simulated in the thermal spectrum to find the optimal with respect to the average / peak temperature tree configuration. However, processing and visualizing such data can pose a challenge due to the high number of vertices and faces that a scene consists of. The fact that the scene is not fully static, leading to elements being (off)loaded frequently, reduces the overall performance.

This paper outlines repeated renderings with geometrical minor changes over day-night cycles spanning weeks or months required in tree planning scenarios. Inspired by (Bulatov et al., 2020) and (Bulatov et al., 2023) we use advanced rendering techniques such as splitting a single buffer into sub-regions as well as buffer orphaning, multiple solutions are provided based on whether the tree configurations differ in size. In particular, we explore multiple options for memory management in order to speed up the rendering of a large scale scenes consisting of a static digital surface model (DSM) as well as dynamically added tree configurations. We explore two advanced techniques from the field of computer graphics – partial updates of buffer regions as well as buffer orphaning – with the goal of optimizing the (off)loading of data that changes very often and, thus, improving the performance of any digital twin that has a visual aspect to it. The main focus is not the computation of light per face but rather the optimal memory layout and update mechanism of the tree configurations that affect the local climate in the context of urban heat islands.

2. Related Work

This paper presents an ongoing effort to improve our occlusion analysis capabilities that was presented in (Bulatov et al., 2020) and (Bulatov et al., 2023). The analysis utilizes a simple algorithm where triangles are sequentially projected onto the scene, with their current depth determined by their barycentric coordinates. If this depth is less than the initial depth for any pixel, that pixel’s depth and foreground index (or feature) map are updated with the triangle’s values. Initially, the depth and index maps are set to infinity. Various enhancements to this basic process have been proposed, such as reordering triangles based on the proximity of their center of gravity to the image plane. Typically, transparent triangles are rendered from the background forward, while opaque triangles are rendered from the foreground backward. To avoid exhaustive searches across large images for pixels not meeting the initial depth map’s criteria, the search range is often limited. In the work of (Guo et al., 2018) streamlined sun radiation calculations for thermal simulations through pre-rendering scenes at specific sun positions, simplifying simulations at the cost of higher memory demands for numerous scenes. (Jones et al., 2012) improved solar radiation computations using periodic pre-computations and B-spline interpolations. Recent approaches incorporate machine learning, especially generative adversarial networks, to synthesize realistic views, enhancing simulation accuracy and visual quality (Srinivasan et al., 2019). Ray tracing improves intersection checks with spatial indexes. Handling large models efficiently in GPU memory has been explored (Feldmann, 2015, Hapala et al., 2011, Landaverde et al., 2014). Another approach found in literature is Unified Virtual Memory (UVM). It simplifies memory management between GPU and CPU by automating page management, reducing complexity for programmers. However, large memory footprints can cause thrashing and degrade performance (Kim and Han, 2024) (Li and Chapman, 2019) (Long et al., 2023). Solutions include thread throttling to optimize memory use, improving performance by 3.44× (Kim and Han, 2024), and a new UVM framework reducing page thrashing by 64.4

3. Preliminaries

3.1 Synchronization between CPU and GPU

Rendering data using a GPU requires it to be transferred to a designated memory so that it is made available to the GPU's multiprocessors. Outside of the context of technologies such as CUDA or geometric and especially compute shaders that allow generating data directly on the GPU, a standard approach is to first create the data on the CPU side (RAM) and then copy it to the GPU (VRAM)[1]. This creates a situation where resources are **shared** between the two subsystems, which poses a problem whenever changing the data between draw calls.

The process of updating and sharing the data between CPU and GPU is called synchronization. In its most basic form it can be described as a continuous and repetitive cycle of reads and writes, where data is first written by the CPU, transferred to the GPU and then read by the GPU as visualized in Figure 1.

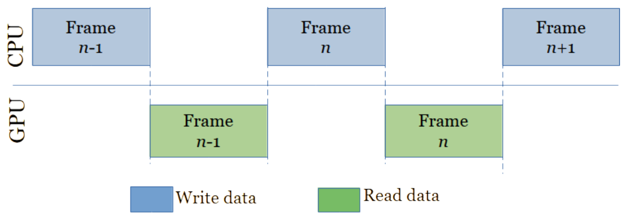


Figure 1. Simplified visualization of the CPU and GPU synchronization. Each dotted line represent a point in time where the synchronization takes place and possible stalling may occur.

There are two types of synchronizations – *explicit* and *implicit*. The former occurs whenever a developer explicitly requests a synchronization. This is the case when the queue has been flushed, thus forcing the driver to execute all commands as soon as possible (including cached ones) or when using fences, synchronization objects and memory barriers. Such mechanisms give greater control over the flow of command loading and execution. While more difficult to use, these provide a clearer and more predictable behavior pattern between draw calls. Contrarily, the *implicit* synchronization occurs without the developer's knowledge. It is part of many OpenGL operations and may introduce unwanted stalling in a rendering pipeline.

[1] VRAM – video random-access memory

3.2 Buffer orphaning

One specific scenario, where such stalling can occur very often, is whenever a buffer object is being streamed. This process consists of continuously delivery of new data to the GPU between draw calls at a high frequency. A GPU's OpenGL driver is allowed to stall the execution of commands in the command queue in order to give the GPU time to execute as many as possible. Changing the buffer's data introduces a new command that may be in conflict with previous draw commands in need for the same resource that was changed. The driver will automatically halt the thread where the resource is being modified until all conflicts are resolved. This triggers an implicit synchronization.

Also called buffer re-specification, buffer orphaning is the process of invalidating the old buffer by setting its corresponding reference to a `NULL`, while retaining other parameters such as usage hint (the hint that is given to the GPU driver such as `GL_STATIC_DRAW` or `GL_DYNAMIC_DRAW`). By doing so, the same reference can be re-used for allocating a new buffer with the same or new size on the CPU side, while the old one is still being used by the GPU. The general assumption in this case is that new storage allocation is faster than the implicit synchronization between the CPU and GPU. The old storage will continue to be in used by previously queued OpenGL commands, thus preventing an implicit synchronization. The new storage will be available for every new command placed onto the queue after the orphaning has taken place. Note that an explicit synchronization may still take place if the driver deems necessary.

However, this methodology has one major drawback, namely that it is not defined in the OpenGL specification (Segal and Akeley, 2022), thus not being viewed as an approach capable of delivering consistent results across platforms. Therefore, it is up to the GPU manufacturer and GPU driver developer (may differ) how this behaviour is implemented.

3.3 Buffer subdata and multi-buffering

A common practice whenever working with large data is to allocate as much of the required memory as possible and re-use it instead of re-allocating constantly. By splitting a buffer into multiple regions OpenGL allows partial updates without having to fully re-create it. While this method has an implicit synchronization guarantee (the data will be copied before moving to the next command), in a mutli-threaded context this may lead to such an implicit synchronization since meanwhile other threads may have modified the data.

The most basic data in a rendering context can be split into two main types based on its usage – vertices and faces. For information loaded from common formats such as Wavefront OBJ, also used in our setup, vertex data can be further split into buffers on a per-vertex component basis such as position, color, normals, texture coordinates etc.. This is done mostly for convenience in order to mirror the structure imposed by the employed data format but may not be optimal for rendering. Splitting can be avoided by employing interleaving – the process of combining multiple types of data into a single buffer. At the cost of additional complexity in the implementation, this technique provides improved data caching that is beneficial to the whole process.

3.4 Structure of a scene

It is usually not possible to load all of the data into the VRAM due to its sheer volume. Therefore, it needs to be split so that it can fit into the available physical memory.

Our scene can be divided into two subsets based on the frequency which it is changed at. We make the assumption that the mesh of the original scene does not change over time and from now on will refer to it as **static data**. The assumption comes from the nature of the experiment as well as the data. The latter represents a DSM where the vertex and face data will remain constant. A change in the vertices and/or faces would mean that either a natural (e.g. hill, river, bush) or artificial (e.g. house, wall, bridge) component has undergone a change. Since the full setup (see related works on occlusion analysis) has the purpose of evaluating natural non-destructive phenomena (here light exposure of each face), such a change is not possible.

All subsequently loaded data is referred to as **dynamic data**. In the following section, we present two different methodologies for handling the dynamic data, represented by sets of tree meshes (tree configurations). Depending on the scenario, these configurations can be adjusted based on the digital twin’s requirements (e.g. buildings, vehicles etc.), which are placed in an iterative manner at different locations. We concentrate on the data management and not the occlusion analysis. A simplified visualization of this division is shown in Figure 2.

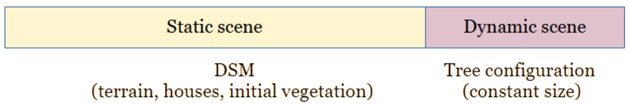


Figure 2. A single buffer split into multiple regions, separating it into rarely and more frequently updated data.

4. Methodology

There are two options for loading the complete scene depending on whether we **prioritize the iteration over all tree configurations first** or **the time frame we want to render** (e.g. several hours throughout the day). While the output is the same, both present a frequency of (off)loading the data. The following notations apply:

S	the **static data** of the scene (here a meshed DSM)
t	instance of **dynamic data** to be added, here a tree
C	dynamic data, the full set of tree configurations
c	a single tree configuration within C
L	the time frame, (sun positions representing a specific time of day each)
l	a single sun position that illuminates S and C at the specific angle
R	output: the full set of renderings for all tree configurations and light conditions
$R(l, c)$	a single rendering for a given l and c

Table 1. Notation used inthe course of this paper.

Prioritizing the tree configurations means that for ever time step l (e.g. hour) within the given time frame L, all c from C are iterated over before moving to $l+1$. This approach is described in Algorithm 1 and requires (off)loading data more often. It is useful whenever rendering the full scene has the biggest impact on overall performance.

```
Data: $C$, $L$, $S$
Result: $R$
load $S$;
foreach $l$ in $L$ do
    foreach $c$ in $C$ do
        load $c$ into buffer region;
        render $R(l, c)$;
    end
end
```

Algorithm 1: Memory update cycle when prioritizing sun position

On the other hand, prioritizing the sun positions (as seen in Algorithm 2) leads to loading each c and then obtaining the final rendering for ever l from L before moving to $c+1$. It is useful if the biggest impact on overall performance is the (off)loading of the data and not the rendering. The frequency which we switch between c at determines where to optimize first.

```
Data: $C$, $L$, $S$
Result: $R$
load $S$;
foreach $c$ in $C$ do
    load $c$ into buffer region;
    foreach $l$ in $L$ do
        render $R(l, c)$;
    end
end
```

Algorithm 2: Memory update cycle when prioritizing tree configurations

Both options follow the division into static and dynamic data. Since we will be using data extracted from OBJ files, we can expand the layout shown in Figure 2 as seen in Figure 3) for the purpose of reducing the complexity of interleaving the data ourselves.

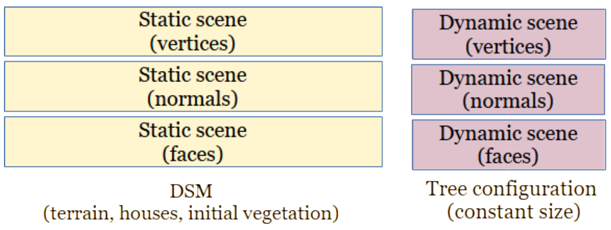


Figure 3. Regardless of the nature of the data (dynamic or static) we can define three distinct buffers each holding a specific component (here vertex positions and normals as well as face indices).

Throughout the rest of this article, in order to achieve higher efficiency we will always use multiple buffers for each each mesh component (vertices and faces). In order to simplify future figures, we will use the notation in Figure 2 as a substitute for Figure 3.

4.1 Buffer orphaning

For this methodology we double the number of buffers as seen in Figure 4.

The first set of buffers holding the static data remains untouched for the whole duration of the simulation. The second one holds the dynamic data and is resized based on the size of the current tree configuration.

A possible benefit here is that it allows to set the usage hint for each type of data – `GL_STATIC_DRAW` (static) and `GL_STREAM_-`

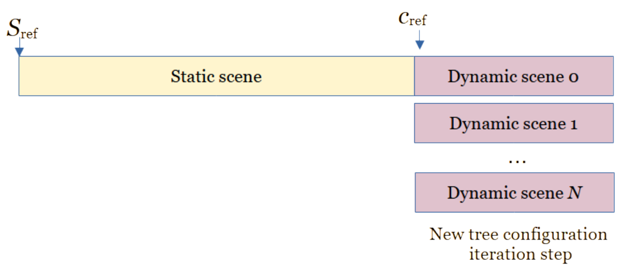


Figure 4. Using orphaning we now have one set of buffers for the **static data** (with the specific optimization flag), while the dynamic data is offloaded to a completely separate set of buffers, which can be shrunk based on the size of the current tree configuration that is being processed.

DRAW (dynamic). Further, unlike the previous method, we can stream new tree configurations into the scene since we do not have to calculate c_{max}. The only restriction comes from the available physical memory.

However, orphaning may experience performance penalties if the tree configurations are not of equal size. In this case the driver is forced to re-allocate new memory for every configuration in addition to loading the data from the CPU's memory.

4.2 Buffer subdata

Unlike the previous method, using a single buffer per component to hold both S as well as the respective c_t means that we are dealing with a constant size per buffer (otherwise we need to employ orphaning, which will be discussed next). However, depending on the scenario different tree meshes (e.g. consisting of different types of trees) and even tree configurations (ranging between a single and possibly thousands of trees) may be employed leading to a conflict between the restriction of the buffers' sizes and the dynamic data they need to accommodate. In order to create a more flexible solution, we can specify the total size of each buffer as the sum of bytes of both the static and dynamic data (refer to Figure 5).

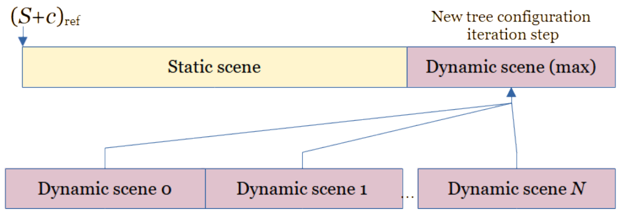


Figure 5. A single buffer per component for both the static and dynamic data allows easy access by simply using an offset (for the dynamic data it always starts from the end of the memory block that stores the static one). Using `BufferSubData()` we iterate the tree configurations before every draw call.

For interleaved data, the size ($\#$) of a single tree t can be obtained by

$$\#t = \#v + \#n + \#f, \tag{1}$$

where $\#v, \#n$ and $\#f$ represent the size of its vertices, normals, and face indices, respectively. Additional components can be added accordingly. Due to the separation of the components, we can assume that the size of a tree per component equals the total size of the respective component. This relationship is expressed by the following equation

$$\#t = \#v \textbf{ or } \#n \textbf{ or } \#f. \tag{2}$$

In a similar fashion, the total size for a whole tree configuration c can be calculated as the the sum of the sizes of each tree t, where t ranges from 0 to a j, and j can vary between configurations. This is expressed in the following equation:

$$\#c = \sum_{i=0}^{j} \#t_i. \tag{3}$$

Similarly, the memory requirements on a per-component basis can be calculated as

$$\#c = \sum_{i=0}^{j} \#t_{v,i} \textbf{ or } \sum_{i=0}^{j} \#t_{n,i} \textbf{ or } \sum_{i=0}^{j} \#t_{f,i}, \tag{4}$$

depending on what has been chosen in Eq. 2. The memory that is reserved for the dynamic data for each component is calculated by taking the largest c. This requires either to pre-load all files one after the other and extract the information about the meshes in them or by providing metadata (e.g. a CSV file) along with the original files that will hold the total number of vertices, face indices etc. for each configuration. In order to update the data in the buffers, a simple offset is required that indicates the end of the static data and the beginning of the dynamic one. In addition all indices need to be recalculated based on the indices of the static data. Otherwise, the mesh will be rendered incorrectly due to the sequential nature of indexing vertices (starting from 0).

If a configuration c has a smaller size than that of c_{max}, the remaining "available" (*de facto* the memory chunks are reserved but not used) space will not be taken into consideration during the rendering due to the previously mentioned offset. Since each buffer represents mixed data we cannot fine tune its usage hint, which may lead to reduced performance if the hint is taken into consideration by the GPU at all (depends on the OpenGL implementation in the driver).

This option is inefficient if we decide to stream new tree configurations with size exceeding the one used during the allocation due to the space restriction that each buffer imposes. That is why knowing c_{max} is important. Otherwise, all memory (each buffer) will have to be re-allocated multiple times.

Regardless of the methodology at hand, both provide better performance compared to simply reloading the complete scene $S + c$ every time it needs to be rendered.

5. Experiment and results

In this section, we provide the necessary details on how we prepared the data for as well as how we conducted the experiment. In the end, we show the results, accompanied by a discussion.

5.1 Setup

For the **static data** we use a subset of the LiDAR composite DTM (Environment Agency, 2023) provided by the UK Environment Agency with a spacial resolution of 10m (see Figure 6), which is high enough for demonstrating the two methodologies we have described in Section 4.

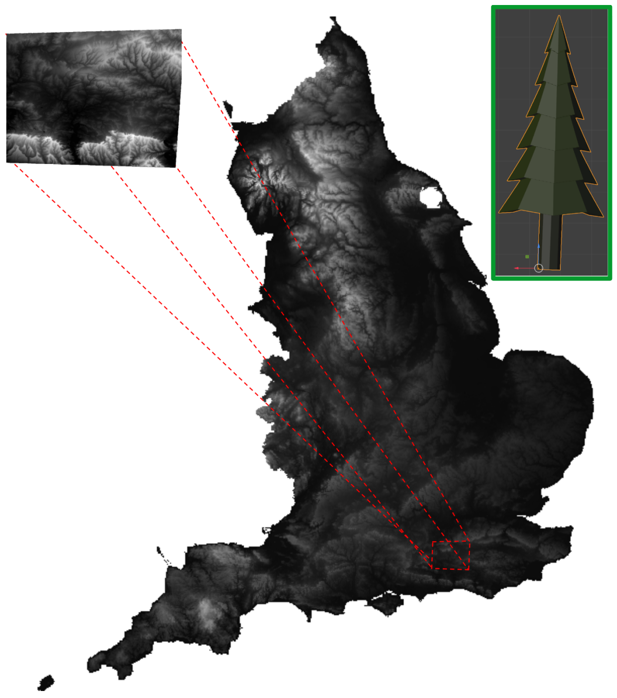


Figure 6. The region of interest, taken from the DTM provided by the UK Environment Agency with a spacial resolution of 10m. The close up of the region of interest is shown near upper left corner and represents our static scene S. The low-poly tree model t is shown in the green-framed rectanle in the upper right corner. It will be used for tree configuration.

After converting the GeoTIFF region of interest to a PNG, we use our custom mesh generator to create the mesh, followed by several post-processing steps in Blender in order to remove various artifacts such as NODATA values on the border of the image that resulted in 3D spike-like shapes.

For the tree configurations, we use a low-poly 3D model (see top right part of Figure 6)[2], which is also employed by the particle system for hair in Blender to generate multiple tree configurations consisting of 10000 or variable number of instances (refer to Figure 7). The emitter is using a hair length of 20m, a local coordinate system at the bottom center of the trunk, rotation around the tree’s Z axis and a scaling factor of 0.050.

Further, using a modifier in Blender we are able to create actual meshes from the instances and export those as a single OBJ file per configuration. This allows the creation of unlimited amount of test data.

5.2 Experiment

The test data is loaded in a sequence between every frame following Algorithm 1. Our experiment presupposes application of a simple shader for coloring each vertex instead of using proper lighting. We are also putting an emphasis on (off)loading data as much as possible since this is the focus of this article. For reading the OBJ files we use the Python library `PyWavefront`. For every configuration we merge all the meshes of all the trees into a single one in order to reduce the draw calls. Otherwise, instead of a single draw call we need to make 10000 per tree configuration. All data is stored in `NumPy` arrays.

[2] Model can be found at https://free3d.com/3d-model/blender-lowpoly-nature-assets-pack-36502.html

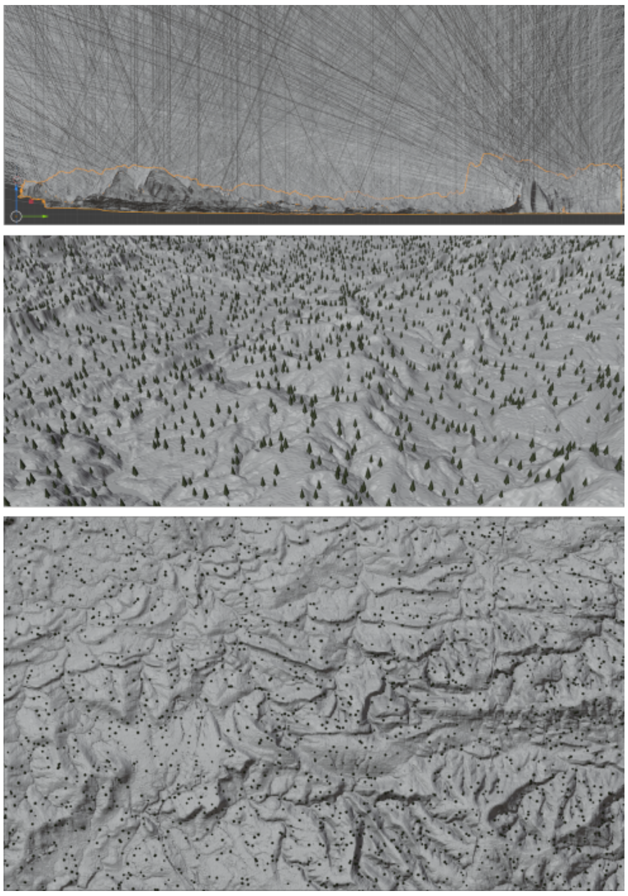


Figure 7. Using Blender provides an easy and fast way to generate random tree configurations on top of the DTM by employing a hair particle system. The result can then be instanced (briefly discussed in Section 6) and exported as OBJ files for benchmarking. *Top*:side view of the particles rendered as paths and not object instances, *middle* and bottom – portion of the scene (perspective and orthographic view) with the instanced object in a higher and lower resolution, respectively.

The relevant parameters of the machine we use can be seen in Table 2.[3]

Device	Specifications Integrated GPU	Specifications Dedicated GPU
CPU	Intel Core 11th Gen i9-11980HK	Intel Core 11th Gen i9-11980HK
GPU	Intel UHD Graphics 750 (integrated)	Nvidia GeForce RTX 3080 Mobile Max-Q
VRAM	variable (shared with CPU)	16GB (dedicated)
RAM	32GB	32GB
OS	Xubuntu 22.04	Xubuntu 22.04

Table 2. The system specifications for the integrated and dedicated GPU.

Since we need to cover two scenarios – constant and variable tree configuration size – along with the DTM, we create two sets with 10 OBJ files each. All files from the first set are constant in size (approx. 127MB representing 10 000 randomly positioned trees), while the latter contains files with variable sizes

[3] All Intel and Nvidia related names are owned by the respective companies.

(ranging between 104KB for 100 and 127MB for 10000 randomly positioned trees), sorted in an ascending order based on the number of trees inside. An example of the experiment can be seen in Figure 8. We do not employ land cover classification to determine whether a tree's position is spatially plausible.

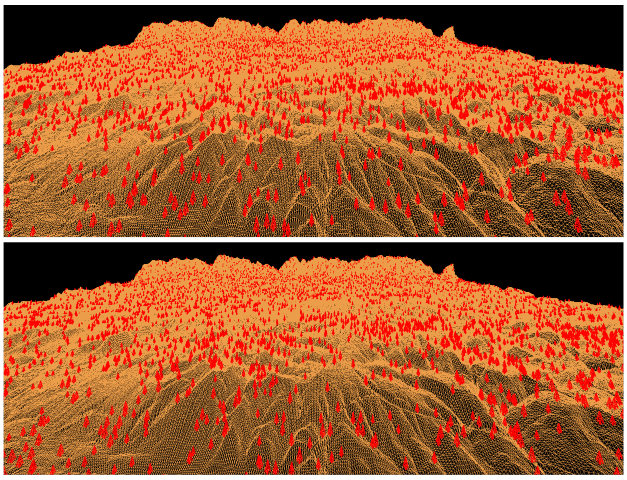


Figure 8. Perspective view of our ROI. Example with two tree configurations (top to bottom), all containing 10000 tree models in our renderer. Wireframe is enabled to better visualize the underlying terrain.

5.3 Results

In order to evaluate the experiment, we record the measured values of FPS and CPU processing time (in seconds). The measurement duration per data set per scenario and methodology was set to 10s because, after conducting several longer tests, we found out that due to the cyclic nature of our experiment, the result remains roughly the same than what we gathered during the first couple of seconds. We selected FPS and CPU processing time since these are common parameters evaluated during benchmarks of rendering engines. Further, these two parameters are related to one another as we will see later on when we present our results. They are also easy to obtain, therefore reducing the possibility for gathering faulty data and skewing our results due to incorrect implementation.

The actual measurement takes place from the updating of the respective buffers to the end of the draw calls. Due to the nature of OpenGL calls being asynchronous and also the possibility of a driver to cache multiple commands before sending those to the GPU, the measurement cannot determine the precise time needed for each frame to be shown onto the screen.

Further, we conducted the experiment on both the integrated as well as the dedicated GPU in order to outline the difference between shared and dedicated VRAM.

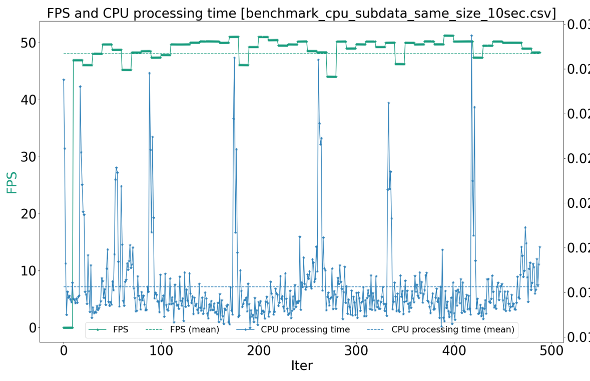
The average values can be seen in Table 3 while a full report on the measurements can be seen in Figure 9, 10, 11 and 12.[4]

Based on the results we notice several important aspects, namely the relationships between the FPS and CPU processing time, the difference in performance between the integrated and dedicated GPU as well as the difference in performance between the two methodologies.

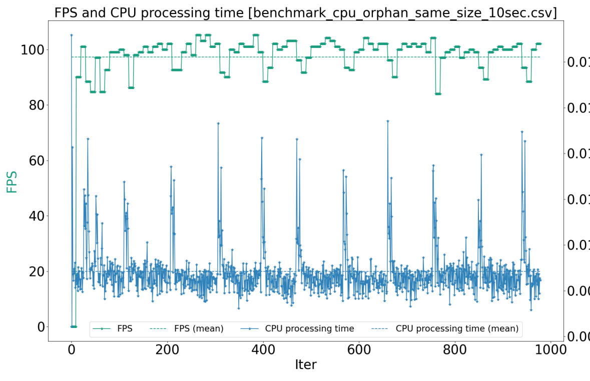
[4] The dip/spike in every plot is due to the requirement for at least 1s to pass in order to estimate FPS as well as the internal changes in the CPU (e.g. CPU frequency boost) due to the change in workload.

Multi-col-row		Scenario			
		Same size (av.)		Different size (av.)	
		FPS	CPU	FPS	CPU
Sub-data	**int.**	48.0840	0.0183	115.7615	0.0071
	ded.	60.4534	0.0162	206.0668	0.0047
Orph	**int.**	97.3502	0.0088	196.1540	0.0025
	ded.	91.2254	0.0108	210.801	0.0044

Table 3. Average values of FPS and CPU processing time for integrated (**int.**) and dedicated (**ded.**) GPU for both scenarios and methodologies.



(a)



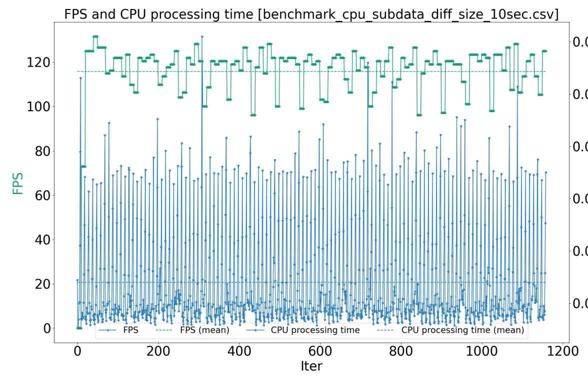
(b)

Figure 9. Integrated GPU benchmark for ”same size” scenario. FPS and CPU processing time (in seconds) for the (a) sub-region and (b) orphaning methodology.

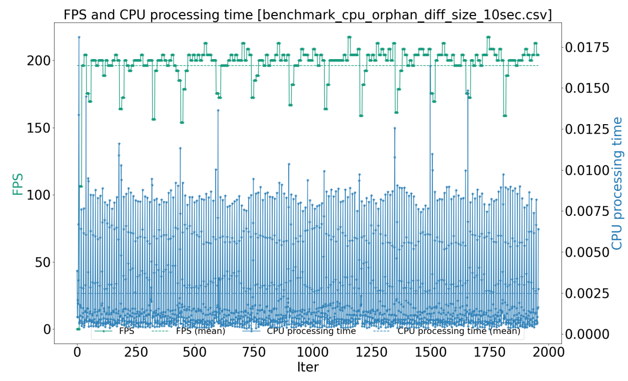
Every time data is being loaded, a copying procedure is triggered. Regardless of whether we use an integrated or dedicated GPU, the data needs to be moved to a specific location in the video memory that is used for the rendering. This is not the case if pinned memory is used, which is something we briefly discuss in Section 6. Copying the data requires the involvement of the CPU that leads to a temporary increase in CPU processing time and reduced FPS.

Due to the underlying hardware and memory layout, the benefit of using a dedicated GPU is reduced due to the involvement of the CPU and transfer of data between the RAM and the dedicated VRAM, which is not the case with the integrated GPU.

The surprising result comes from the overall comparison between both methodologies. Our initial assumption was that using a single large buffer and updating part of it without the need to allocate a new one would be faster. For the sub-region approach we tried multiple usage hints including `GL_STREAM_DRAW`, also employed by the orphaning) without any noticeable

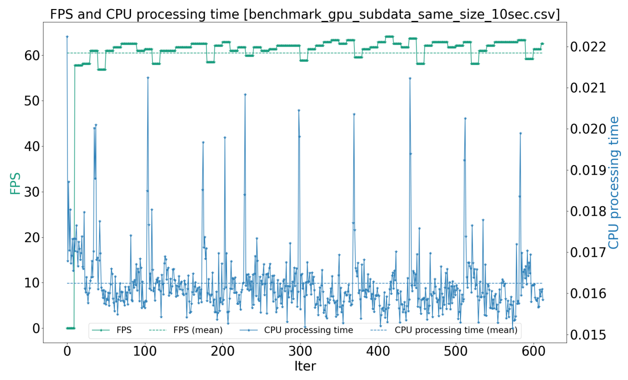


(a)

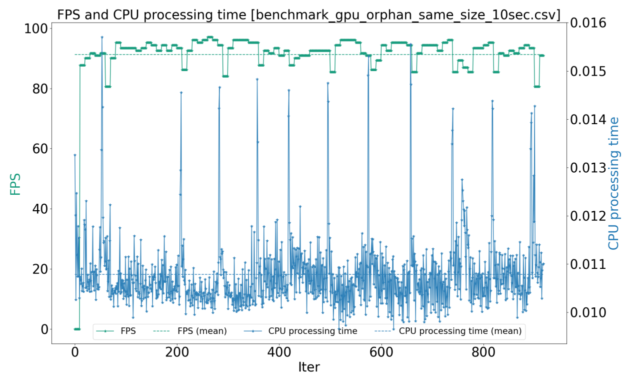


(b)

Figure 10. Integrated GPU benchmark for ”different size” scenario. FPS and CPU processing time (in seconds) for the (a) sub-region and (b) orphaning methodology.



(a)



(b)

Figure 11. Dedicated GPU benchmark for ”same size” scenario. FPS and CPU processing time (in seconds) for the (a) sub-region and (b) orphaning methodology.

difference. However, in our prediction we did not take into consideration the impact that our transformations of the underlying NumPy arrays may have. Beside the flattening of the arrays (a requirements imposed by OpenGL), adding of the offset adds to the CPU computation time. While such calculations are minuscule, whenever working in sub-milliseconds time ranges they do make a difference. Further optimizations made by the driver and the GPU remain hidden and cannot be directly evaluated.

6. Coclusion and outlook

We explored two very popular techniques for rendering large scenes in OpenGL in the context of remote sensing – sub-region buffer updates and buffer orphaning. Based on the experiment we conclude that buffer orphaning is the preferable approach for our setup.

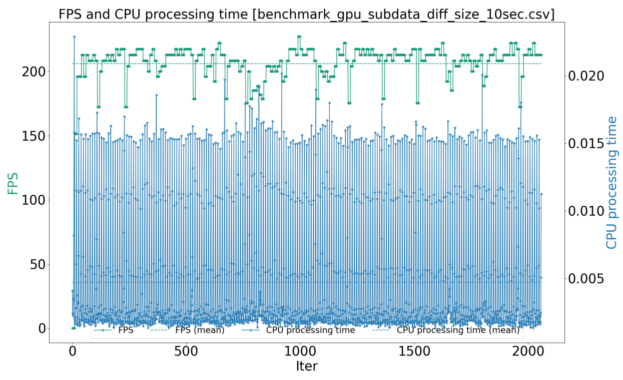
While memory layout plays an important role in efficiency, reducing the memory footprint is another parameter that needs to be investigated and improved upon. For example, our current full setup for occlusion analysis does not include mesh instancing. This is a common technique that allows drawing a single model multiple times at the cost of only the respective world transformation (rotation, translation and scaling), given that the same vertex data is used as seen in Figure 13.

While for the experiment described in Section 5.1 the tree we used is defined by a rather simple mesh, in many digital twins that is not the case depending on the level of detail they offer. Therefore, we need to be able to work with high number of vertices and faces per model that appears many times in our scene. Creating even a tree configuration of 100 trees for a small urban area would require roughly 1GB of memory if we use the model from Figure 13. Contrarily, for the same setup instancing the object would roughly require 10MB plus 100 transformation matrices of negligible size. These transformations can be precalculated on the CPU side and chained together using simple matrix multiplication.

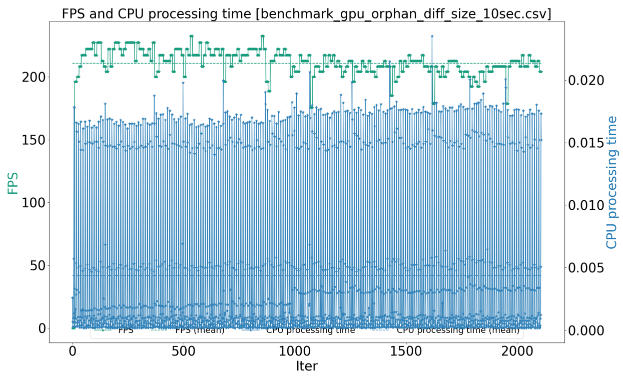
Another possible technique worth investigating is the use of persistent mapped buffers. It allows a more efficient way of transferring data between the CPU and GPU at the cost of having to deal with explicit synchronization thus increasing the complexity of the implementation, using triple buffering for the actual rendering as well as facing possible performance issues for large data (which is our case) and its allocation. It is useful for very frequently updated buffers, which is the case with the dynamic data in our scene.

Yet another improvement would be the use of pinned memory. This technique makes it possible to lock a specific memory range so that certain computer subsystems (the GPU in our case) can use direct memory access (DMA). By doing so the subsystem is able to retrieve the data from the RAM without the help of the CPU. It also removes the chances of a page fault (the application tries to access data that is currently not in RAM). Both classic computer graphics as well as modern high performance computing (e.g. using CUDA) workflows greatly benefit from this approach.

Last but not least, especially for large scenes and long rendering times, a distributed approach will be beneficial. A centralized



(a)



(b)

Figure 12. Dedicated GPU benchmark for ”different size” scenario. FPS and CPU processing time (in seconds) for the (a) sub-region and (b) orphaning methodology.

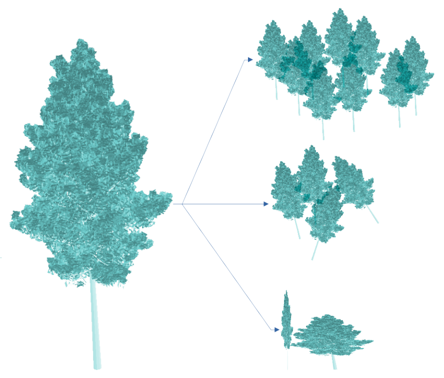


Figure 13. Instancing allows using the same vertex data (here represented as the tree mesh on the left, consisting of more than 157000 vertices and approx. 303500 faces) multiple times only at the cost of simple world transformation as seen on the right (top to bottom – translation, translation and rotation, translation and scaling).

location (e.g. a database) may provide the meshes as well as sun positions to multiple computing nodes (e.g. a cloud cluster). One or multiple sun positions or tree configurations (depending on the prioritization as seen in the beginning of Section 4) will then be downloaded by the respective node, triggering the rendering process. The results can then be sent separately or in bulk for final processing and storage. A more complex approach would be to also separate the static data into smaller, independent from one another chunks, leading to a more evenly distributed and scalable solution.

References

Bulatov, D., Burkard, E., Ilehag, R., Kottler, B., Helmholz, P., 2020. From multi-sensor aerial data to thermal and infrared simulation of semantic 3D models: Towards identification of urban heat islands. *Infrared Physics & Technology*, 105, 103233.

Bulatov, D., Hecht, M., Kottler, K., Mispelhorn, J., Strauss, E., 2023. On Acceleration of Thermal Simulation of Urban Scenes with the Application of an Evolutionary Algorithm to Tree Planting Strategies. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, X, 2–7.

Environment Agency, 2023. LIDAR Composite Digital Terrain Model (DTM) 10m. Online dataset. `https://www.data.gov.uk/dataset/7f31af0f-bc98-4761-b4b4-147bfb986648/lidar-composite-digital-terrain-model-dtm-10m` [Accessed: 2024-05-01].

Feldmann, D., 2015. Accelerated ray tracing using R-trees. *GRAPP*, 247–257.

Guo, S., Xiong, X., Liu, Z., Bai, X., Zhou, F., 2018. Infrared simulation of large-scale urban scene through LOD. *Optics express*, 26(18), 23980–24002.

Hapala, M., Davidovič, T., Wald, I., Havran, V., Slusallek, P., 2011. Efficient stack-less bvh traversal for ray tracing. *Proceedings of Spring Conference on Computer Graphics*, 7–12.

Jones, N. L., Greenberg, D. P., Pratt, K. B., 2012. Fast computer graphics techniques for calculating direct solar radiation on complex building surfaces. *Journal of Building Performance Simulation*, 5(5), 300–312.

Kim, H., Han, H., 2024. GPU thread throttling for page-level thrashing reduction via static analysis. *Journal of Supercomputing*, 80(7), 9829 – 9847. Cited by: 0.

Kjellstrom, T., Holmer, I., Lemke, B., 2009. Workplace heat stress, health and productivity–an increasing challenge for low and middle-income countries during climate change. *Global Health Action*, 2(1), 2047.

Landaverde, R., Zhang, T., Coskun, A. K., Herbordt, M., 2014. An investigation of unified memory access performance in cuda. *IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 1–6.

Li, L., Chapman, B., 2019. Compiler assisted hybrid implicit and explicit gpu memory management under unified address space. Cited by: 19; All Open Access, Bronze Open Access.

Long, X., Gong, X., Zhang, B., Zhou, H., 2023. An Intelligent Framework for Oversubscription Management in CPU-GPU Unified Memory. *Journal of Grid Computing*, 21(1). Cited by: 1; All Open Access, Green Open Access.

Osmond, P., Sharifi, E., 2017. *Guide to urban cooling strategies*. Low Carbon Living CRC.

Segal, M., Akeley, K., 2022. The OpenGL® Graphics System: A Specification, v4.6 (Core Profile). `https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf` [Accessed: 2024-05-15].

Srinivasan, P. P., Tucker, R., Barron, J. T., Ramamoorthi, R., Ng, R., Snavely, N., 2019. Pushing the boundaries of view extrapolation with multiplane images. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 175–184.

Williams, S., Nitschke, M., Weinstein, P., Pisaniello, D. L., Parton, K. A., Bi, P., 2012. The impact of summer temperatures and heatwaves on mortality and morbidity in Perth, Australia 1994–2008. *Environment International*, 40, 33–38.

Wong, N. H., Yu, C., 2005. Study of green areas and urban heat island in a tropical city. *Habitat International*, 29(3), 547–558.