CEIG – Spanish Computer Graphics Conference (2021)
A. Chica and L. Ortega (Editors)

Comparison of GPU-based methods for handling point cloud occlusion

Alfonso López[1], Juan M. Jurado [1], Emilio J. Padrón[2], Carlos J. Ogayar[1] and Francisco R. Feito[1]

[1]Department of Computer Science, University of Jaén
[2]Department of Computer Science, University of A Coruña

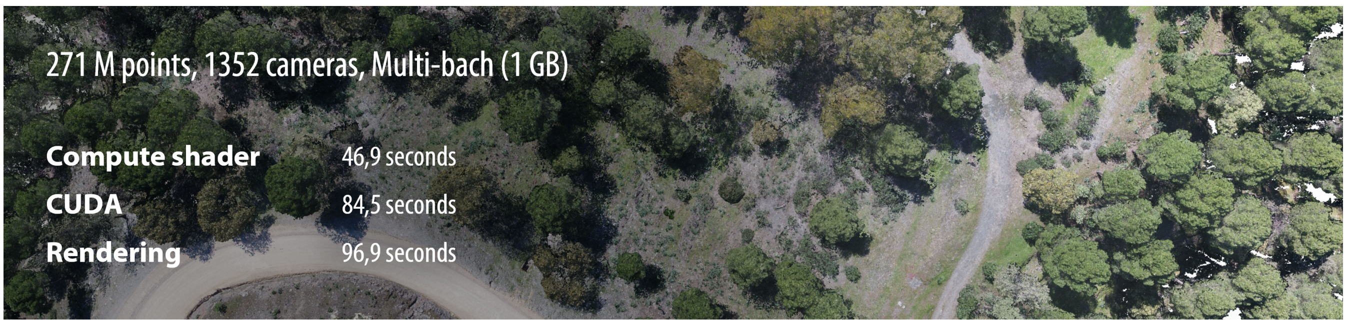

Figure 1: *Performance comparison: OpenGL and CUDA implementations for a dataset of 271 million points and 1352 different viewpoints.*

Abstract

Three-dimensional point clouds have conventionally been used along with several sources of information. This fusion can be performed by projecting the point cloud into the image plane and retrieving additional data for each point. Nevertheless, the raw projection omits the occlusion caused by foreground surfaces, thus assigning wrong information to 3D points. For large point clouds, testing the occlusion of each point from every viewpoint is a time-consuming task. Hence, we propose several algorithms implemented in GPU and based on the use of z-buffers. Given the size of nowadays point clouds, we also adapt our methodologies to commodity hardware by splitting the point cloud into several chunks. Finally, we compare their performance through the response time.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Massively parallel algorithms;*** *Visibility; Point-based models;*

1. Introduction

Remote sensing of world environments typically outputs three-dimensional (3D) point clouds that integrate points with no connectivity. There is a significant variety of tools for surveying a scene as a point cloud, from laser scanners measuring the distance from the sensor to the surface to image-based sensors that rely on post-processing techniques. Photogrammetry methods estimate 3D models from a sequence of overlapping images through the detection of features visible on multiple images, thus allowing to reconstruct 3D positions and camera's motion. Structure from Motion (SfM) or SfM-MVS (Multi-View Stereo) algorithms are typically used as part of software solutions for generating 3D point clouds.

Projection from 2D to 3D world allows estimating a 3D structure. In addition, points can be associated with further data, mainly from other sensors active while inspecting the environment. Naive reconstruction methods assign colour information from the sequence of images used for generating the 3D point cloud, e.g. RGB values. Recently, mapping complementary sensors to a primary point cloud is gaining interest, as 3D features are also relevant for analysis purposes along with visible spectrum data. For instance, [JOCF20] proposed a method focused on mapping multispectral images on previously estimated RGB point clouds, whereas [JGPS19] combines RGB and thermal imagery in the same point cloud. Fusion of several sources of information can also be performed through the alignment of multiple point clouds [WWRJ18], although this method is proved to be prone to inaccurate results [JGPS19].

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

DOI: 10.2312/ceig.20211364

Beyond the projection problem, mapping alternative data sources to a primary point cloud involves a commonly omitted problem, given by the occlusion of visible object surfaces. Consequently, it leads to assigning wrong colour data from foreground surfaces to occluded objects. Provided the utility of working with 3D points and reliable data, this work is focused on solving the occlusion problem for 3D point clouds. Although world surfaces are here discretized as points, dense reconstructions allow using depth-testing techniques. However, it requires large point clouds ranging from a few millions points to several hundred million of them. On the other hand, alternative methods based on mesh reconstruction are more likely to fail on complex surfaces, as occurs with the vegetation represented in our case study.

Therefore, parallel computing is an adequate alternative to solve the occlusion problem with minimal response time. Modern graphics processing units (GPUs) can work with several millions of points in real-time, whereas hundreds of millions of points require further optimizations, e.g. using Level of Detail (LOD) methods for rendering. Therefore, this work aims to provide an efficient GPU method to determine which points are visible from a sequence of viewpoints (cameras). Furthermore, there exist a wide variety of frameworks to develop GPU-accelerated solutions, either focused on rendering, such as OpenGL, or High-Performance Computing, such as CUDA, OpenCL or OpenACC.

Hence, our work proposes several methods for occlusion testing suitable for commodity hardware. Whereas further preprocessing is possible, we aim to provide a simple and efficient workflow that minimizes the use of GPU memory. Therefore, point clouds are processed through several batches to identify visible points on each image pixel. For that purpose, we use the OpenGL cross-platform to develop both rendering and compute based methods to solve the same problem. In addition, their performance is compared to a CUDA implementation.

2. Related work

Regarding fusion techniques in remote sensing, few works study the augmentation of point clouds by projecting images to the 3D space, as alignment-based methodologies are simpler to implement. Among them, occlusion tests are frequently omitted. [JCO*20] detects occluded points by estimating a minimal triangle mesh for each point and its surrounding points. However, complex objects require sophisticated approaches in order to reconstruct them, e.g. synthetic tree modelling. [SHLW10] addresses the occlusion problem with a complex solution based on ground segmentation and 3D clustering for LiDAR point clouds and RGB imagery. [HL12] detects shadows and occlusion in high-resolution imagery using Z-buffers in GPU, although they apply this test as a post-processing technique from a previously generated orthophoto. Therefore, it is observed that the occlusion problem has barely been addressed to project additional sources of information to 3D point clouds.

On the other hand, the speedup of methods focused on point cloud processing and occlusion testing has been previously assessed using several compute mechanisms and buffer ordering methods [SKW21], thus proving the benefits of different OpenGL pipelines.

3. OpenGL Rendering pipeline

We propose an approach for occlusion testing based on the use of a depth buffer (also known as z-buffer) since it is an efficient alternative to prevent wrong colouring in large point clouds. Furthermore, this problem can be straightforwardly solved through the OpenGL Rendering Pipeline as it already addresses the occlusion for a frame, either it is rendered as a point cloud (`GL_POINTS`) or not. More specifically, it is the merging phase at the end of the classic rasterization pipeline that is in charge of the visibility test, thus determining which primitive is visible on each pixel [AMHH*18]. In our case study, the size of a depth buffer depends on the resolution of the imagery. Furthermore, we do not aim to solve the visibility test by itself but to determine which point (primitive) is visible on a pixel. Then, colours from additional sources of information could be aggregated from the 3D point indexed by a given pixel. For that purpose, points are transferred to the GPU as a set of `vec4` values (i.e. four floating-point numbers) to fit the data alignment rules of subsequent buffers.

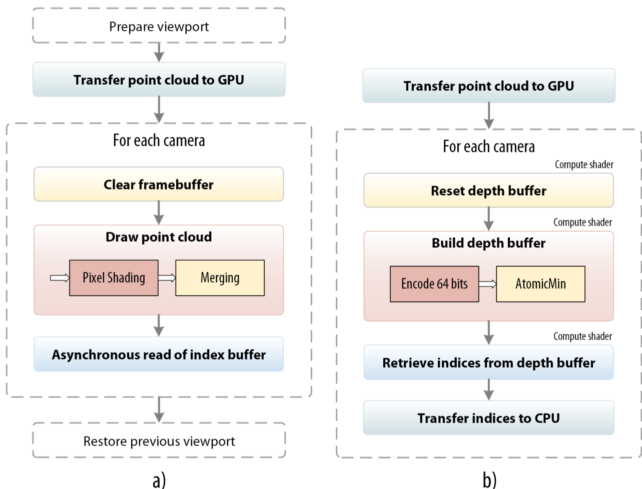

Figure 2: *Overview of a) rendering and b) compute shader methodologies for a single batch of 3D points.*

Consequently, a framebuffer with two textures of the same size is required during this procedure. The first texture saves unsigned integer values, whereas the second texture is the aforementioned z-buffer. Resetting the framebuffer implies clearing the depth buffer and assigning a large value (∞) to the indexing texture. The usage of this framebuffer is given by the workflow depicted in Figure 2. Once cleared, we draw the whole point cloud over the framebuffer by applying the camera projection matrix, thus retrieving the nearest primitive visible on each pixel. The first texture is then transferred to the CPU through an asynchronous reading (`glReadPixels`). As opposed to limited GPU memory, CPU capacity is considered large enough to maintain a buffer of size $n_v \cdot g_x \cdot g_y$, provided that n_v is the number of viewpoints and g_x, g_y is the imagery resolution.

However, large point clouds cannot be completely contained in a single buffer in OpenGL. Hence, we propose an alternative version of the workflow depicted in Figure 2, where the maximum amount

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

of active points is bounded by the number of usable gigabytes. Nevertheless, the ordering of received points is not known, therefore the occlusion is not solved by disjoint stages that affect distinct areas of the index buffer. Consequently, the asynchronous reading step implies reading the content of both the depth and index buffers, which are then transferred to GPU for the next batch of points (instead of being reset). To transfer the downloaded z-buffer content to the GPU, another rendering stage is needed for each iteration, where its core is given by the built-in attribute `gl_FragDepth` in the vertex shader. Also note that the framebuffer is not cleared on each iteration anymore, but only for the first batch of points. An alternative solution would be to solve the occlusion problem for each viewpoint on a single iteration, although data transferring from CPU to GPU increases and so does the response time, since every point cloud batch is transferred once per viewpoint ($n_v \cdot n_b$ data transfers, instead of n_b, provided that n_b is the number of point cloud subdivisions).

4. Compute shaders

Compute shaders are general-purpose shaders for GPGPU programming. Thus, it can be used for tasks not related to rendering, although it is also convenient for rendering purposes as it gets rid of a static pipeline [SKW21]. However, z-buffers are not self-contained in this shader stage. Consequently, we need to simulate a z-buffer through a Shader Storage Buffer Object (SSBO).

The main drawback of depth buffers in multi-threaded environments is to control the concurrent access to buffer objects. Whereas compute shaders allow performing atomic operations, our problem requires swapping two values through the same atomic block (distance, stored in the z-buffer, and point index, for our index buffer). However, distance and its associated index can be packed into a single integer of 64 bits (`uint64_t`), where the depth is stored in the most significant bits. Consequently, encoded values are sorted according to their distance to the viewpoint, whereas the corresponding index is simply carried out during the `atomicMin` operation. By splitting 64 bits equally, we can represent up to 4 billion different values for distance and index. A naive solution to fit float values into an unsigned integer of 32 bits is to multiply the depth by a factor k, e.g. 10000. The larger k is, the more accurate is the depth sorting. Nevertheless, we can obtain the encoding of a floating-point value as an unsigned integer through `floatBitsToUint` while it preserves bit-level representation. The encoded depth is then shifted 32 bits to occupy the most significant bits.

Before depth ordering, the z-buffer must be initialized with ∞ so that pixels not visible from a viewpoint are represented by the maximum value of 64 bits. Once the z-buffer is computed, it is simplified as a single buffer of 32 bits per pixel, thus discarding the depth term. Consequently, the allocated memory in CPU is half the memory needed for allocating the z-buffer in GPU. Not to mention that response time for transferring from GPU to CPU is also reduced. When working with several partitions of the point cloud, the workflow is similar to the described for the OpenGL methodology. However, previous optimizations related to resource allocation are here omitted, since we need to transfer the whole buffer of encoded values to CPU, as each point cloud batch only solves the occlusion problem partially ($64\ bits \cdot n_v \cdot g_x \cdot g_y$).

5. CUDA

The CUDA kernel for computing the nearest 3D point for every pixel and viewpoint is analogous to the compute shader workflow. The main challenge in the CUDA implementation is to hide the CPU $\leftrightarrow$ GPU transfers by overlapping computation and communication as much as possible. Multiple CUDA streams exploit asynchronous transfer operations and kernel executions to achieve an efficient result. Hence, while points from a batch are being projected and their depth tested/updated by an atomic operation in the CUDA kernel, the next batch is being transferred to GPU. Furthermore, while a viewpoint is being processed by the current kernel, the results from the previous one are being transferred to the CPU.

6. Projection behaviour

To simulate a real case study, each viewpoint has its projection matrix. Furthermore, images commonly present induced aberrations, e.g. fisheye distortion. For that purpose, 3D points are projected into the image plane and subsequently transformed to obtain the distorted pixel from where the colour should be retrieved. Accordingly, radial and tangential distortions are straightforwardly solved as described in [BB01]. Consequently, the resulting point can either be within the image area or not.

7. Evaluation

We have measured the performance of the GPU-based methods considering several configurations of the same scene. From the initial environment, with 271 million points and 1352 cameras, we have extracted a simplified dataset with 66 million points and 180 cameras. Hence, up to four configurations are reported throughout this section. The evaluation was performed on a PC with Intel Core i9-9900 3.1 GHz, 48 GB RAM, RTX 2080 Ti GPU with 11 GB RAM (Turing architecture) and Windows 10 OS. The proposed methodology is implemented in C++ along with OpenGL (Open Graphics Library) both for rendering and massively parallel computing tasks, whereas the CUDA method is also developed in C++.

Table 1 shows the response time measured from a block of code that includes data transfers to GPU or CPU as well as the described behaviour. Single batch configuration loads the point cloud as a single buffer when it is possible. However, the maximum size of an SSBO is far behind the GPU memory. Therefore the main difference between **Single batch** and **Multiple batches** is that the first approach allocates buffers of the maximum allowed size, although it works as a **Multiple batches** method for large point clouds. On the other hand, CUDA allows loading the largest point cloud (271M points) in a single buffer. For each configuration, the reported results are the lowest response time from five executions.

From the results reported in Table 1 and Figure 3, we can observe that **Compute Shader** significantly improves the response time obtained using **Rendering Pipeline** and **CUDA**. Also, **Rendering Pipeline** offers slightly better results than **CUDA** for the single batch workflow, although their response time is indeed very similar. Regarding the use of multiple batches, **Rendering Pipeline** and **Compute Shader** approaches worsen its performance, whereas the **CUDA** method presents results similar to the previously reported response time.

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

Table 1: *Response time of OpenGL and CUDA methodology, including data transfers between CPU and GPU and the core behaviour.*

Proposed methods		Environment			
		180 cameras		1352 cameras	
		66M points	271M points	66M points	271M points
Single batch	CUDA	2,97s	10,30s	23,58s	84,45s
	Rendering Pipeline	2,52s	10,38s	20,70s	77,28s
	Compute Shader	**1,43s**	**6,21s**	**8,90s**	**37,75s**
Multiple batches (1 GB)	CUDA	3,22s	11,19s	25,42s	84,50s
	Rendering Pipeline	3,71s	13,35s	29,15s	96,99s
	Compute shader	**2,57s**	**7,61s**	**18,16s**	**46,90s**

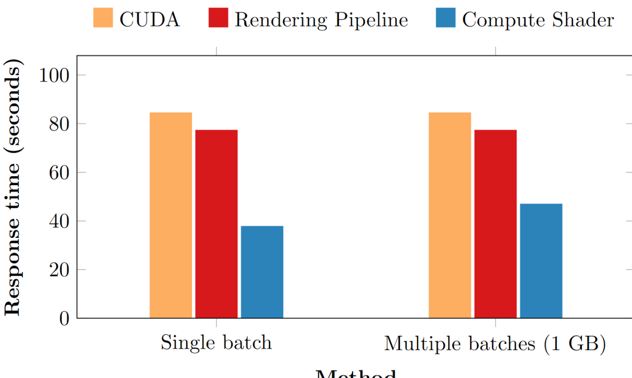

Figure 3: *Response time in seconds of both OpenGL methods and CUDA implementation for the most complex configuration (271M points, 1352 viewpoints).*

8. Conclusions and future work

We have explored multiple massively parallel methodologies to solve the occlusion problem when adding a new source of information to a 3D point cloud. For that purpose, we used the rendering pipeline as well as general compute shaders from the OpenGL framework, and we compared them with a CUDA implementation. The main drawback of the rendering procedure is the use of a set of static stages, instead of early discarding some points not visible from a viewpoint. On the other hand, compute shaders were proved to be more flexible, although it implied implementing a z-buffer through a buffer of encoded values. Nevertheless, the compute shader solution was proved to be more efficient than CUDA and rendering-based approaches for every configuration, although we observed that the performance of CUDA implementation was more stable throughout the different proposed tests.

In future work, we would like to conduct a deeper study on the performance impact from reordering the point buffer. Also, spatial data structures can be relevant for reducing the response time. Finally, we could provide even more competitive results with multiple GPUs in heterogeneous architectures such as clusters.

Acknowledgements

This work has been partially supported by the Spanish Ministry of Science, Innovation and Universities via a doctoral grant to the first author FPU19/00100 and through the research projects TIN2017-84968-R and PID2019-104184RB-I00/AEI/10.13039/501100011033.

References

[AMHH*18] AKENINE-MÖLLER T., HAINES E., HOFFMAN N., PESCE A., IWANICKI M., HILLAIRE S.: *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, Boca Raton, FL, USA, 2018. 2

[BB01] BEAUCHEMIN S. S., BAJCSY R.: Modelling and removing radial and tangential distortions in spherical lenses. In *Multi-Image Analysis*. Springer Berlin Heidelberg, 2001, pp. 1–21. doi:10.1007/3-540-45134-x_1. 3

[HL12] HU X., LI X.: Fast occlusion and shadow detection for high resolution remote sensing image combined with LiDAR point cloud. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences XXXIX-B7* (Aug. 2012), 399–402. doi:10.5194/isprsarchives-xxxix-b7-399-2012. 2

[JCO*20] JURADO J. M., CÁRDENAS J. L., OGAYAR C. J., ORTEGA L., FEITO F. R.: Semantic segmentation of natural materials on a point cloud using spatial and multispectral features. *Sensors 20*, 8 (Apr. 2020), 2244. doi:10.3390/s20082244. 2

[JGPS19] JAVADNEJAD F., GILLINS D. T., PARRISH C. E., SLOCUM R. K.: A photogrammetric approach to fusing natural colour and thermal infrared UAS imagery in 3d point cloud generation. *International Journal of Remote Sensing 41*, 1 (July 2019), 211–237. doi:10.1080/01431161.2019.1641241. 1

[JOCF20] JURADO J. M., ORTEGA L., CUBILLAS J. J., FEITO F. R.: Multispectral mapping on 3d models and multi-temporal monitoring for individual characterization of olive trees. *Remote Sensing 12*, 7 (Mar. 2020), 1106. doi:10.3390/rs12071106. 1

[SHLW10] SCHNEIDER S., HIMMELSBACH M., LUETTEL T., WUENSCHE H.-J.: Fusing vision and LIDAR - synchronization, correction and occlusion reasoning. In *2010 IEEE Intelligent Vehicles Symposium* (June 2010), IEEE. doi:10.1109/ivs.2010.5548079. 2

[SKW21] SCHÜTZ M., KERBL B., WIMMER M.: Rendering point clouds with compute shaders and vertex order optimization, 2021. arXiv:2104.07526.

[WWRJ18] WEBSTER C., WESTOBY M., RUTTER N., JONAS T.: Three-dimensional thermal characterization of forest canopies using UAV photogrammetry. *Remote Sensing of Environment 209* (May 2018), 835–847. doi:10.1016/j.rse.2017.09.033. 1

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.