A GPGPU-based Pipeline for Accelerated Rendering of Point Clouds

Christian Günther[1] guechr

Thomas Kanzok[1] tkan

Lars Linsen[2] l.linsen

Paul Rosenthal[1] ropau

[1] Chemnitz University of Technology
Department of Computer Science
Visual Computing Laboratory
Straße der Nationen 62
09111 Chemnitz, Germany
[acronym]@hrz.tu-chemnitz.de

[2] Jacobs University
School of Engineering & Science
Visualization and Computer Graphics Laboratory
Campus Ring 1
28759 Bremen, Germany
[acronym]@jacobs-university.de

ABSTRACT

Direct rendering of large point clouds has become common practice in architecture and archaeology in recent years. Due to the high point density no mesh is reconstructed from the scanned data, but the points can be rendered directly as primitives of a graphics API like OpenGL. However, these APIs and the hardware, which they are based on, have been optimized to process triangle meshes. Although current API versions provide lots of control over the hardware, e.g. by using shaders, some hardware components concerned with rasterization of primitives are still hidden from the programmer. In this paper we show that it might be beneficial for point primitives to abandon the standard graphics APIs and directly switch to a GPGPU API like OpenCL.

Keywords

OpenCL, GPGPU, OpenGL, Point Cloud Rendering

1 INTRODUCTION

In architecture as well as archaeology laser scanning has become a valuable tool to capture spacious environments for processing and inspection. The common use cases include airborne Lidar-scanning [RD10] or terrestrial laser scanning systems for urban reconstruction [NSZ+10], virtual inspection of caves and catacombs [SZW09], as well as documentation of excavation sites [LNCV10]. Regardless of the particular application, the scanning process usually produces a dense point cloud, often consisting of several hundred million samples. Those are comprised of a geometric locus and are sometimes enhanced with color or normal information. In order to work with the data in an interactive manner it has to be visualized efficiently, which often involves preprocessing the point clouds to cope with the data size [Lev99, SMK07].

While some approaches aim at the reconstruction of a mesh surface from the data [PV09], several others content with the direct visualization of the point data, either as simple point primitives [WS06] or by using splatting [WBB+08]. Although the latter usually produces renderings of higher quality, there are several preprocessing steps involved, which limit the applicability for direct on-site previews of the captured data.

Raw point primitives on the other hand suffer from the fact that closed surfaces can only be achieved if the sampling density, projected to the screen, exceeds the viewport resolution. However, also in the presence of slightly undersampled data satisfactory depictions can be produced by filtering and post-processing the generated rendering in screen space [KLR12, RL08].

For the actual rendering of its preferred representation virtually every approach uses one of the standard graphics APIs, like DirectX or - in the scientific community predominantly - OpenGL. These APIs are designed to make use of the massive parallel processing power of todays GPUs by following a quite strict pipeline, which all graphics primitives have to traverse before they are displayed to the screen. Although the APIs provide different primitive types to render, what they are optimized for is the primitive type that is commonly used in the consumer market - triangle meshes, like they occur in practically all modern 3D games.

The widespread use of this geometric structure has led to spezialized hardware with dedicated processing units for the different pipeline stages. Although many units are programmable using shaders, some parts of the hard-

ware, in particular the rasterization units, are still not exposed to the programmer. These units, although immensely useful for dealing with large numbers of triangles, do not offer any benefit for direct point rendering, where each point gets projected to exactly one fragment during vertex processing. Unfortunately, they can not be deactivated or circumvented in the current API implementations and therefore form an unnecessary bottleneck for point cloud rendering by restricting the primitive throughput to just a few rasterization processors instead of thousands of shader cores.

Direct access to parallel computing hardware is provided by OpenCL or CUDA, which leads to the question if reimplementing a graphics pipeline specifically tailored towards point primitives could provide significant performance improvements when dealing with this kind of data. In this paper we present such an implementation using OpenCL and show results, which show major speedups compared to the standard OpenGL pipeline in real world datasets.

The rest of the paper is organized as follows: After giving a short overview over the field of point cloud rendering and related work on self-implemented rendering pipelines we describe the challenges and design considerations that went into our implementation. Afterwards we present the implementation of our pipeline and provide a detailed performance analysis on both real and synthetic data.

2 RELATED WORK

Point based rendering is a well studied field in computer graphics and there exists a large amount of literature on that topic. Since this paper's contribution does not lie in any new rendering technique, we give only a short overview of the two main classes of local surface reconstruction methods used in this field and refer the reader to the survey literature [GP07, KB04] for a more in-depth explanation.

A common problem when using point primitives is that, unless the sampling density of the model is really high, there can always be more or less prevalent holes in the rendering. We divide the reconstruction techniques used to produce a hole-free rendering into object space approaches, which require some kind of preprocessing to work, and pure image space approaches, which operate only on the rendering of the raw point cloud.

The most dominant object space approach has arguably become splatting, which was initially introduced for volumetric data by Westover [Wes90]. The approach was later adapted to only use surface samples [PZvBG00, RL00] and it has undergone several extensions and improvements since [GGP04, PSL05, ZPvBG01], even making it applicable for ray tracing [LMR07, SJ00].

However, all these approaches have to compute local surface parameters like splat size, normal direction, curvature etc. in advance. This can take a considerable amount of time when processing really large point clouds as produced by modern laser scanning systems. On the other hand, GPU computing capabilities have increased vastly in recent years, making it possible to compute the splat parameters purely in image space for only those points that are actually visible [PJW12]. When the data is dense enough to only exhibit small holes in the rendering these holes can be filled using interpolation [GD98, PGA11] or special morphological filters [DRL10, RL08].

With triangle rasterization being implemented in graphics cards there has not been much practical need to reimplement the process in software. There are some cases that benefit from a custom rasterizer, although they are mostly limited to applications on gaming consoles [Val11] or to the usage of non-triangular parametric surface patches [Eis09].

Actual GPGPU implementations of triangle rasterizers using CUDA have mainly been developed out of academic interest and as a benchmark application for the current state of GPGPU computing [LK11, LHLW10]. These primarily have to overcome the problem of assigning the triangles to different threads in a way that maximizes the amount of parallel coverage computations for them [MCEF08].

However, their results suffer heavily from the necessary sorting and therefore still do not reach the performance of the hard-wired hardware implementation. Since point primitives only cover one fragment, these prior results are not applicable to our problem. In the following chapters we show that GPGPU-based software rasterization of point clouds is not only not slower than the hardware pipeline, but can yield a significant performance increase with datasets common in architecture and archaeology.

3 DESIGN CONSIDERATIONS

Our goal in this work is to develop a point cloud rendering pipeline that produces results true to the ones obtained when using OpenGL rendering. For best performance in OpenGL, we setup a minimalistic OpenGL 4 pipeline using the early depth test, which saves some fragment shader instantiations under certain conditions. In order to achieve the same results we have to rebuild the basic OpenGL rendering pipeline in GPGPU software. Since we are dealing with zero-dimensional primitives we can omit a tessellation or geometry shader stage (although the latter could be added on demand – provided its output is point primitives again). Also we did not yet include normal information for shading and restricted our first prototype to only use geometry and color information. However, these additions can be easily implemented when needed without major changes in relative render times. The pipeline implemented in our software is shown in Figure 1. We decided to use

OpenCL in favor of CUDA to ensure platform independence. Theoretically, our renderer would not even have to run on a graphics card but could also be used on any other multicore processing unit, e.g. CPUs, hybrids of CPU and GPU like AMDs Accelerated Processing Units [Bro10] or Intels recently introduced Xeon Phi coprocessors [Xeo12].

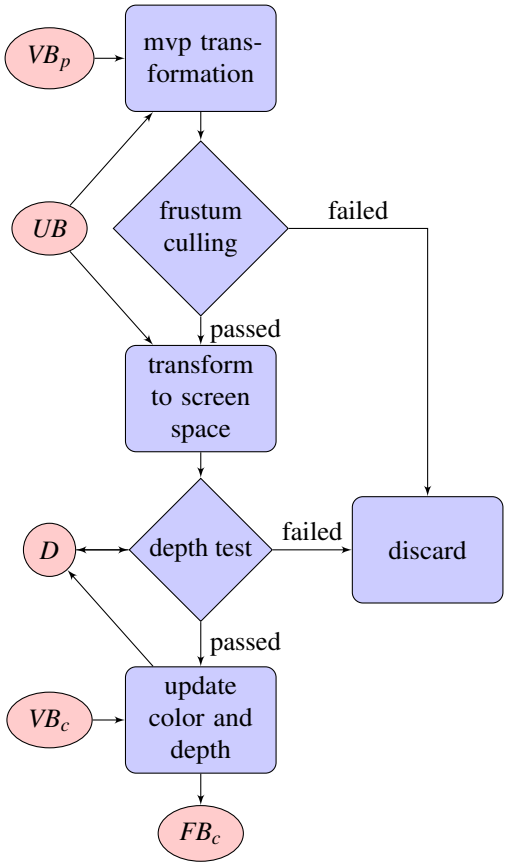

Figure 1: The rendering pipeline implemented by our software. After fetching the vertex coordinates from a vertex buffer object VB_p we project them from object space to clip space via a model-view-projection matrix from a uniform buffer object *UB*. In this space we can already discard many points outside the view frustum. The remaining points are transformed to screen space using the screen resolution from the same uniform buffer object, where their respective positions in the depth buffer *D* is known and can be used to discard occluded points. For the remaining points, we fetch their color information from the vertex buffer object VB_c and write color and depth to the depth buffer and color buffer FB_c, respectively.

To maximize parallel throughput we have to relax the OpenGL paradigm that "Commands are always processed in the order in which they are received. [...] This means, for example, that one primitive must be drawn completely before any subsequent one can affect the frame buffer" [SA12]. One could argue that we consider complete point clouds as one primitive for which the restriction holds. Anyhow, we found that this does not pose a serious limitation since point cloud data from laser scanners does usually not contain transparent points for which the ordering of draw calls would make a difference. What *can* happen is a temporal flickering of points when z-fighting (the term is used here to refer to points that receive the same depth value after projection) occurs under a random draw order. However, this was not noticeable in our experiments.

The central problem of parallel software rendering is to ensure a thread-safe depth test. This test is necessary to guarantee that for each fragment only the point closest to the viewer gets drawn to the frame buffer. To achieve this in a thread safe way, we have to use a global depth buffer that is shared over all compute units of the GPU in connection with atomic operations provided by OpenCL. Unfortunately, as of now it is not possible to use the actual OpenGL depth buffer as shared OpenCL buffer. That is why we allocated our own pure OpenCL buffer for this proof of concept. We could implement a thread-safe depth test on an integer buffer using OpenCL atomics (`atomic_min`). However, this would only work if we wanted to render *only* to the depth buffer. When we also want to write color or other attributes (see Algorithm 1) this approach could lead to a race condition, as depicted in Figure 2.

1: **if** zDepth $<$ atomic_min(depthMap,zDepth) **then**
2: write color information for the current fragment to the color buffer
3: **end if**

Algorithm 1: *Not* thread-safe depth test

To overcome this, we have to expand our critical section (i.e. the section of code, which may not be executed in parallel) to include *all* buffer reads and writes. Such a behavior can be implemented using an atomic flag that indicates whether any thread is currently accessing the section and in this way assures mutual exclusion (mutex). The following section will provide details on the implementation of this solution.

4 IMPLEMENTATION

4.1 Basic Approach

OpenCL offers the possibility to share buffer objects with OpenGL (except depth buffers), which makes it possible to use nearly the same buffer layout for OpenGL and OpenCL rendering. In particular we are using two vertex buffer objects VB_p and VB_c for the point cloud's position (as an array of `float[3]`) and color (as an array of `unsigned char[3]`), respectively. The rendering is done into the color attachment FB_c of a frame buffer object. To transfer the necessary information about the transformation matrices and viewport resolution, we are using a shared uniform buffer object *UB* with the following layout:

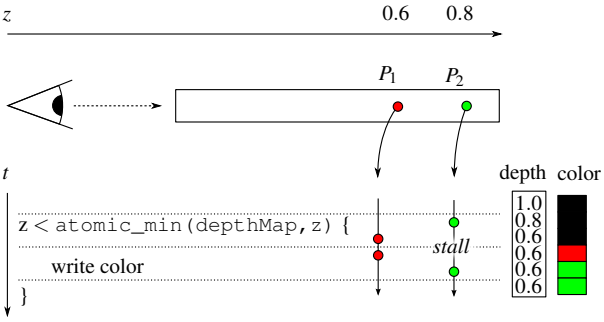

Figure 2: Race condition with a naive depth buffering approach. Assume two points P_1 and P_2 get processed in parallel by the GPU and they are projected to the same fragment. After projection P_1 is closest to the viewer and therefore its color should be written to the color buffer. However, if the first line of Algorithm 1 is processed first for P_2 (letting P_2 pass the depth test for now) and the processing thread is stalled for some reason before line two is reached, then the color of P_1, which passed the depth test in the meantime and was written to the color buffer, would be overwritten without re-checking the depth buffer.

- UB_{MVP} the pre-multiplied model-view-projection matrix
- UB_r the viewport resolution

Additionally, we allocate our own depth buffer D that contains the current closest depth value D_z interleaved with a binary lock D_l for each fragment (see also Figure 1 for the usage pattern of these buffers). The points are processed in parallel, so each work item is responsible for exactly one point.

The thread-safe variant of our rasterization kernel is depicted in Algorithm 2. After extending the vertex coordinates to homogeneous coordinates, applying the standard transformations, and discarding all points that are not in the view frustum (lines 1 to 5), we try to lock the fragment to which our point was projected by using `atomic_cmpxchg` on the lock-component of our depth buffer. If we are able to obtain the lock, we can perform the depth test and write new color and depth information if the test was successful. If we can *not* acquire the lock we have to wait for it to be freed again. This "busy waiting" is a waste of processing time, since many points would probably not pass the depth test in the first place.

For that reason we added an early-out mechanism by enclosing the whole critical section into the else branch of an upstream z-test (see Algorithm 3).

The possibility to discard points early that would not influence the final rendering causes major speedups in cases where there are lots of points projected to one fragment.

1: $\mathbf{p} \leftarrow VB_p(i)$ {load one point position $\mathbf{p}$ per thread i, append homogeneous 1}
2: $\mathbf{p} \leftarrow UB_{MVP} \cdot \mathbf{p}$ {transform to clip space}
3: perform frustum culling
4: $\mathbf{p} \leftarrow \mathbf{p} \cdot \frac{1}{p_w}$ {transform to normalized device coordinates}
5: $\mathbf{p} \leftarrow (\mathbf{p} + [1,1,1,1]^T) \cdot 0.5 \cdot [UB_{r.x}, UB_{r.y}, 1, 1]^T$ {transform to screen space}
6: **while** $\mathbf{p}$ not processed **do**
7: try to lock $D_l(p_x, p_y)$ with `atomic_cmpxchg`
8: **if** got the lock **then**
9: **if** $p_z < D_z(p_x, p_y)$ **then**
10: $D_z(p_x, p_y) \leftarrow p_z$
11: $FB_c \leftarrow VB_c(i)$
12: **end if**
13: mark as processed
14: free the lock $D_l(p_x, p_y)$
15: **end if**
16: Barrier
17: **end while**

Algorithm 2: The basic point rasterization algorithm with thread-safe depth test.

1: ...
2: **while** $\mathbf{p}$ not processed **do**
3: **if** $p_z \geq D_z(p_x, p_y)$ **then**
4: mark as processed
5: **else**
6: [lines 7 to 16 of Algorithm 2]
7: **end if**
8: **end while**

Algorithm 3: The early-out optimization for the thread-safe depth test of Algorithm 2.

4.2 Challenges and Solutions

During implementation we noticed several pitfalls and shortcomings of current drivers and the OpenCL API, which we had to find workarounds for. In this section we will present the problems we found and explain how we were able to solve them.

4.2.1 Implicit Compiler Optimizations

The OpenCL compiler, which is included in the vendors graphics card driver, performs lots of implicit optimizations on the code. It reorders the OpenCL code to achieve the highest possible instruction level parallelism. This is achieved by analyzing the code, especially finding reads and writes to the same memory locations and evaluating dependencies in computations. Unfortunately for a mutex structure, the actual critical section works on completely different memory than the lock, which can be missed by the compiler. To ensure correct locking behavior, we have to insert a memory barrier in our code. This barrier ensures that the reordering does not exceed this point during compilation *and* run time.

4.2.2 Accuracy

Strict frustum culling is essential for the software rendering. One has to make sure to only read or write in the valid ranges of the depth and color buffers (otherwise the graphics driver may freeze). Unfortunately, it seems that unsafe internal math optimizations can lead to out-of-bound buffer accesses when only doing the culling in clip space. Therefore we have to add an additional range check after transforming the coordinates to the viewport.

In addition, the performance of our approach can receive a huge performance boost when using the full floating point range for depth testing. Normally, OpenGL takes normalized real depth values in the range of $[-1,1]$ from the normalized device coordinates (after the division by w) of a point, maps them to $[0,1]$, and stretches this interval to a 24 bit integer which is used for the depth test. Our implementation uses a floating point buffer with 32 bit precision. Mapping these depth values to the $[0,1]$ interval may introduce discretization errors in the binary representation that can be alleviated when omitting the division by w for the depth value.

This does not lead to inconsistent depth values as it would when using triangles, since we do not have to interpolate this value in image space.

The described procedure does not only lead to a significant boost in performance, due to the more efficient early-out mechanism (see Table 1), but also eliminates the occasional depth flickering which was noticeable with the previous approach.

4.2.3 Depth Sharing

Sharing an OpenGL depth buffer with OpenCL is still not supported in current drivers, although there are efforts in this direction via a proposed extension [CLE12]. As of now, one has to render the depth buffer to the OpenGL depth buffer in a consequent pass. Since the extension specification was published in November 2012, we are confident that future drivers will support shared depth buffers, which makes it easier to integrate OpenCL rendering into existing engines.

4.2.4 Caching Effects

Normally the driver would cache read and write operations to buffer objects, which is generally not a problem, since most buffers are either read-only or write-only. For our depth buffer, however, we need reading as well as writing in combination with atomic operations. This pattern can lead to problems when caches of several compute units are involved in the computation. In fact, it can happen that one thread uses a value from its cache while the actual depth value in the buffer has already changed. To overcome this problem, the depth buffer has to be declared `volatile`, causing the driver to broadcast changes to every compute unit that uses the buffer.

5 RESULTS AND DISCUSSION

To evaluate the performance of our approach, we performed benchmarks on a synthetic and a real-world dataset. The synthetic one enables us to evaluate the two determining factors – the number of points in the dataset and the number of z-tests in the depth buffer – in detail. The real-world scan provides a direct comparison between a minimalistic OpenGL 4 pipeline with early depth test and our own OpenCL rendering pipeline.

For our experiments we used a viewport with a size of 1024×1024 pixels and designed the synthetic dataset to align with the viewport pixels when using parallel projection. For now we focused our experiments on AMD hardware, i.e. the Radeon HD 7970 GPU, because it supports all modern OpenCL features and offers the best balance between large memory – useful for experiments with large amounts of data – and an affordable price. Nevertheless we also validated our results with an Nvidia GeForce GTX 680 graphics card.

5.1 Synthetic Data

The synthetic dataset was created as a cuboid of n planes with 1024^2 points each. This facilitates direct control over the number of z-tests during rendering. The number of planes was chosen to be the largest possible such that the entire dataset fitted into the 3GB of GPU memory. To eliminate possible effects caused by the structure of the data, all vertices were shuffled in memory to ensure a random point distribution. Finally, smaller subsets with $k < n$ planes were created to be used for incremental growth of the dataset during testing.

The left part of Figure 3 shows the rendering performance in milliseconds per frame while we were gradually increasing the number of points in the dataset. In each step one plane of 1024^2 points was added to the cuboid, effectively increasing the number of z-tests per fragment by one. The benchmarks indicate that our pipeline is – with a rendering speed of 4.47ms vs. 2.24ms per frame – slightly slower than the OpenGL implementation when there are few z-tests in the image. As the number of z-tests increases the speed of OpenCL rendering compared to OpenGL rendering grows linearly until the OpenCL rendering is as fast as the one with OpenGL at 46 z-tests per fragment.

When the full GPU memory capacity was reached with 170 planes (around 178 million points) the speed compared to OpenGL reached 121%. Afterwards we had to rearrange the dataset to increase the number of z-tests. We decreased the base resolution until the whole dataset lay in one single pixel. The performance results of this step can be seen in the right part of Figure 3. The OpenGL rendering times are still increasing while our pipeline gets even faster with more z-tests, because large amounts of points can be discarded early.

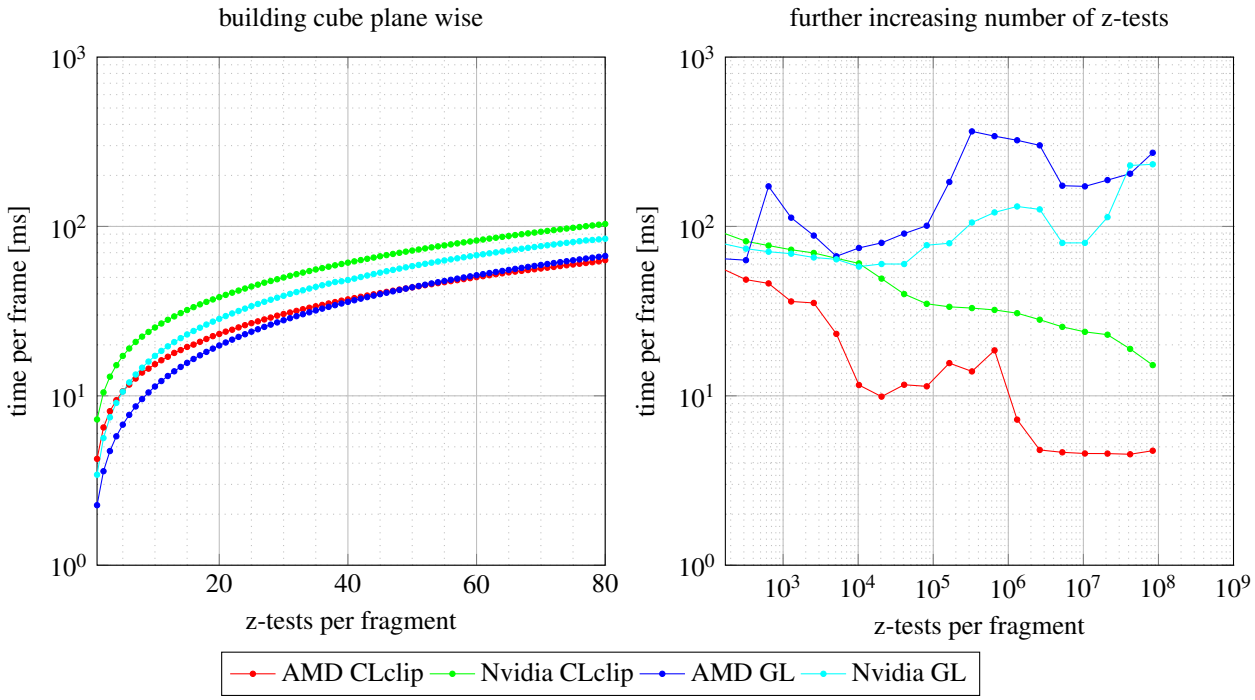

Figure 3: Rendering time in milliseconds (on a logarithmic scale) for different dataset sizes. The left plot shows the results of gradually increasing the number of planes in a 1024^2 cuboid, leading to increased amount of necessary z-tests. In the right plot the overall size of the dataset stays the same, while we gradually rearranged it to cover fewer pixels. In each step the effective resolution was halved in alternating directions, leading to a doubling of the amount of z-tests. In the end the whole dataset is projected to one single pixel. The timings represent the median of 100 distinct measurements. The flattening in the end of the OpenCL series for the Radeon HD 7970 is probably due to efficient caching in local memory (It begins at 32 covered fragments which corresponds to one fragment per each of the 32 compute units).

5.2 Real World Data

To compare the real-world performance of our rendering approach with the fixed-function one, we use a real laser scanning dataset with 138 million points. The data was obtained from a bridge and is composed of five single scans registered into one dataset.

In Figure 4 we show a comparison of the rendering results using OpenGL and OpenCL for the same view of the dataset. There are no noticable differences in rendering quality, but the performance of the OpenCL renderer was more than ten times as high as the OpenGL performance (18ms vs. 129ms or 56fps vs. 5fps, respectively). This can be attributed to the immensely large number of z-tests, especially in the vicinity of the scanners (see Figure 5).

To further analyze these results, we also measure the rendering performance of our approach from a variety of different representative viewpoints in the scene. The results, which can be seen in Table 1, show that the OpenCL renderer outperforms the OpenGL one in all given situations. Although Nvidia's driver support for OpenCL lacks the efficiency of their CUDA one, we also tested the same situations on a GeForce GTX680. Because Nvidia's OpenCL driver does not allow for sharing larger buffer objects, we had to downsample the dataset to half its original size. In this test we were able to achieve better performance in most cases, while the complete overview with plenty of z-tests was faster in OpenGL when using normalized depth values. We assume that our early depth test does not perform as good on Nvidia's hardware as on the one by AMD. However, when doing the depth test in clip space this is accelerated by a factor of 5. In any case the whole pipeline is faster in common use cases.

It can, however, only reach its full potential, when there is a large amount of z-tests. This makes it difficult to make any assumptions as to how OpenCL rendering could influence existing point cloud rendering systems [RD10, WBB+08, WS06], since they all incorporate some level-of-detail mechanism, which inherently minimizes the amount of z-tests. What *could* be done in such a case to further improve performance in our renderer, is to superimpose some space partitioning structure to enable early frustum culling. We left this for future work, since we wanted to investigate the theoretical possibilities of the brute-force method first.

(a) (b)

Figure 4: Comparison of (a) OpenGL rendering and (b) OpenCL rendering. As one can see, rendering results are almost identical. There are only small differences in far away parts of the scene, because we use eight bits more depth buffer precision than OpenGL. While the OpenGL version runs at about 5 frames per second, our implementation reaches 56 fps.

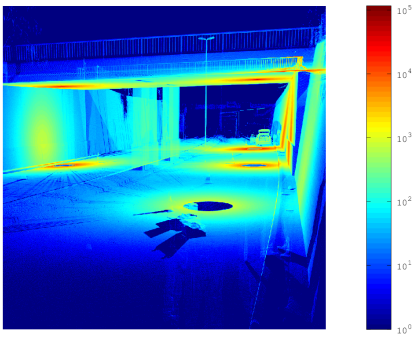

Figure 5: An overview of the amount of z-tests in the view of Figure 4. The scale is logarithmic and ranges from 1 (blue) to 116243 (red) points per fragment.

6 CONCLUSIONS

We have presented an implementation of a point rendering pipeline using OpenCL. Although the approach is certainly not completely optimized yet, point cloud rendering in OpenCL already promises huge gains in rendering time compared to OpenGL. We were able to achieve speedups of one order of magnitude for typical datasets and view parameters. Hierarchical frustum culling and clever reordering strategies on the GPU could push the boundaries even further.

There are several points we had to postpone for now, because the current specifications of OpenCL do not account for the desired behavior. If those possibilities were to be added in the future we would like to include them in our implementation. First of all, using a shared depth buffer will be essential for combined OpenCL-OpenGL rendering. Although the necessary extension has been specified, we have to wait for hardware vendors to implement it in their drivers. Also it would be great (but is not planned at the moment) if a possibility existed to stall waiting threads directly from the OpenCL code. That way waiting threads could be rescheduled for a while, letting the respective compute unit process other points while some fragment is locked. Third, a further investigation of register usage by the OpenCL compiler could provide helpful insights. There seems to be quasi-random utilization of GPU registers by the OpenCL compiler, even when using very strict scopes for variables. This leads to significant performance differences when compiling nearly identical code on the same or on different platforms. Overcoming this problem seems to be impossible with the current drivers

7 ACKNOWLEDGMENTS

The authors would like to thank the enertec engineering AG (Winterthur, Switzerland) for providing us with the data and for their close collaboration. This work was partially funded by EUREKA Eurostars (Project E!7001 "enercloud - Instantaneous Visual Inspection of High-resolution Engineering Construction Scans").

8 REFERENCES

[Bro10] Nathan Brookwood. Amd Fusion^{TM} Family of Apus: Enabling a Superior, Immersive PC Experience. Technical report, Advanced Micro Devices, Inc., March 2010.

(a) View 2

(b) View 3

(c) View 4

	Point distribution CL_c			Radeon HD 7970					GeForce GTX 680				
View	Culled	Early Out	Z-Test	GL	CL_n	GL/CL_n	CL_c	GL/CL_c	GL	CL_n	GL/CL_n	CL_c	GL/CL_c
1	6.50	128.78	3.26	5.21	58.68	11.27	61.46	11.80	6.98	19.24	2.76	23.26	3.33
2	8.70	126.24	3.62	5.73	55.04	9.61	59.91	10.45	6.97	19.73	2.83	23.96	3.43
3	130.57	5.31	2.67	13.14	97.56	7.43	102.50	7.80	7.04	77.29	10.97	78.95	11.20
4	0	138.07	0.48	2.91	15.36	5.28	90.66	31.14	6.22	2.13	0.34	36.38	5.84

Table 1: Rendering performance for some representative points of view in frames per second. Compared is the rendering performance of OpenGL (GL) and OpenCL with depth test in clip space (CL_c) or in the space of normalized device coordinates (CL_n). The views include large amounts of z-tests in combination with some frustum culling (View 1, see Figure 4, and View 2), intense frustum culling with few z-tests (View 3) and an overview of the complete dataset with an enormous amount of z-tests and no frustum culling (View 4). The same experiments were carried out on an Nvidia GeForce GTX680 (right). Unfortunately, we had to reduce the amount of points to 50% here, because the driver did not allow the sharing of larger buffer objects. Additionally we measured the amount of points that were processed in the different stages of our pipeline (all values are given in millions). The exact relation between early out and z-test depends on scheduling and latencies, which are slightly influenced by the measurement itself. Summing up the points that were culled, rejected by the early-out mechanism and finally processed by the depth test gives the dataset size of 138 million Points.

[CLE12] The OpenCL Extension Specification Version 1.2. Technical report, Khronos OpenCL Working Group, November 2012.

[DRL10] Petar Dobrev, Paul Rosenthal, and Lars Linsen. Interactive Image-space Point Cloud Rendering with Transparency and Shadows. In Vaclav Skala editor, *Communication Papers Proceedings of WSCG, The 18th International Conference on Computer Graphics, Visualization and Computer Vision*, pages 101–108, Plzen, Czech Republic, 2 2010. UNION Agency–Science Press.

[Eis09] Charles Loop Christian Eisenacher. Real-time patch-based sort-middle rendering on massively parallel hardware. Technical report, Microsoft Research, May 2009.

[GD98] J. P. Grossman and William J. Dally. Point sample rendering. In *Rendering Techniques 98*, pages 181–192. Springer, 1998.

[GGP04] Loïc Barthe Gaël Guennebaud and Mathias Paulin. Deferred splatting. *Computer Graphics Forum*, 23(3):653–660, 2004.

[GP07] Markus Gross and Hanspeter Pfister, editors. *Point-Based Graphics*. Morgan Kaufmann, 2007.

[KB04] Leif Kobbelt and Mario Botsch. A survey of point-based techniques in computer graphics. *Computers &; Graphics*, 28(6):801 – 814, 2004.

[KLR12] Thomas Kanzok, Lars Linsen, and Paul Rosenthal. On-the-fly luminance correction for rendering of inconsistently lit point clouds. *Journal of WSCG*, 20(2):161 – 169, 2012.

[Lev99] M. Levoy. The digital michelangelo project. In *3-D Digital Imaging and Modeling, 1999. Proceedings. Second International Conference on*, page 2–11, 1999.

[LHLW10] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. Freepipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, pages 75–82, New York, NY, USA, 2010. ACM.

[LK11] Samuli Laine and Tero Karras. High-performance software rasterization on gpus. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 79–88, New York, NY, USA, 2011. ACM.

[LMR07] Lars Linsen, Karsten Müller, and Paul Rosenthal. Splat-based ray tracing of point clouds. *Journal of WSCG*, 15(1-3):51–58, 2007.

[LNCV10] José Luis Lerma, Santiago Navarro, Miriam Cabrelles, and Valentín Villaverde. Terrestrial laser scanning and close range photogrammetry for 3D archaeological documentation: the upper palaeolithic cave of parpalló as a case study. *Journal of Archaeological Science*, 37(3):499–507, March 2010.

[MCEF08] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pages 35:1–35:11, New York, NY, USA, 2008. ACM.

[NSZ+10] Liangliang Nan, Andrei Sharf, Hao Zhang, Daniel Cohen-Or, and Baoquan Chen. SmartBoxes for interactive urban reconstruction. In *ACM SIGGRAPH 2010 papers*, page 93:1–93:10, New York, NY, USA, 2010. ACM.

[PGA11] Ruggero Pintus, Enrico Gobbetti, and Marco Agus. Real-time rendering of massive unstructured raw point clouds using screen-space operators. In Franco Niccolucci, Matteo Dellepiane, Sebastián Peña Serna, Holly E. Rushmeier, and Luc J. Van Gool, editors, *VAST*, pages 105–112. Eurographics Association, 2011.

[PJW12] Reinhold Preiner, Stefan Jeschke, and Michael Wimmer. Auto splats: Dynamic point cloud visualization on the gpu. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, 2012.

[PSL05] Renato Pajarola, Miguel Sainz, and Roberto Lario. Xsplat: External memory multiresolution point visualization. In *IASTED International Conference on Visualization, Imaging and Image Processing*, JUL 2005.

[PV09] Shi Pu and George Vosselman. Knowledge based reconstruction of building models from terrestrial laser scanning data. *ISPRS Journal of Photogrammetry and Remote Sensing*, 64(6):575–584, November 2009.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[RD10] Rico Richter and Jürgen Döllner. Out-of-core real-time visualization of massive 3D point clouds. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, page 121–128, 2010.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: a multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[RL08] Paul Rosenthal and Lars Linsen. Image-space point cloud rendering. In *Proceedings of Computer Graphics International*, pages 136–143, 2008.

[SA12] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 4.3 (Core Profile)). Technical report, The Khronos Group Inc., August 2012.

[SJ00] Gernot Schaufler and Henrik Wann Jensen. Ray tracing point sampled geometry. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 319–328, 2000.

[SMK07] Ruwen Schnabel, Sebastian Möser, and Reinhard Klein. A parallelly decodeable compression scheme for efficient point-cloud rendering. In *Proceedings of Symposium on Point-Based Graphics*, page 214–226, 2007.

[SZW09] Claus Scheiblauer, N. Zimmermann, and Michael Wimmer. Interactive domitilla catacomb exploration. In Kurt Debattista, Cinzia Perlingieri, Denis Pitzalis, and Sandro Spina, editors, *VAST*, pages 65–72. Eurographics Association, 2009.

[Val11] Michal Valient. Practical occlusion culling in Killzone 3. Siggraph2011, 2011.

[WBB+08] Michael Wand, Alexander Berner, Martin Bokeloh, Philipp Jenke, Aarno Fleck, Mark Hoffmann, Benjamin Maier, Dirk Staneker, Andreas Schilling, and Hans-Peter Seidel. Processing and interactive editing of huge point clouds from 3D scanners. *Computers & Graphics*, 32(2):204–220, 2008.

[Wes90] Lee Westover. Footprint evaluation for volume rendering. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, pages 367–376, New York, NY, USA, 1990. ACM.

[WS06] Michael Wimmer and Claus Scheiblauer. Instant points: Fast rendering of unprocessed point clouds. In *Proceedings Symposium on Point-Based Graphics 2006*, page 129–136, 2006.

[Xeo12] Intel® Xeon Phi™ Coprocessor. Technical Report 328209-001EN, Intel Corporation, November 2012.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 371–378, New York, NY, USA, 2001. ACM.