OctaVis:
Optimization Techniques for Multi-GPU Multi-View Rendering

Eugen Dyck[1], Holger Schmidt[1], Martina Piefke[2], Mario Botsch[1]

[1]Computer Graphics Group, [2]Physiological Psychology,

Bielefeld University

Abstract

We present a high performance—yet low cost—system for multi-view rendering in virtual reality (VR) applications. In contrast to complex CAVE installations, which are typically driven by one render client per view, we arrange eight displays in an octagon around the viewer to provide a full 360° projection, and we drive these eight displays by a single PC equipped with multiple graphics units (GPUs). In this paper we describe the hardware and software setup, as well as the necessary low-level and high-level optimizations to optimally exploit the parallelism of this multi-GPU multi-view VR system.

Keywords: Multi-View Rendering, Multi-GPU Rendering, Virtual Reality

1 Introduction

Thanks to the steady increase in computational resources and rendering performance over the last decade, virtual reality (VR) techniques have developed into valuable tools for a large variety of applications, such as automotive design, architectural previews, game development, or medical applications, to name just a few. In all these applications a high level of immersion is both desired and required.

First presented at the GI VR/AR Workshop 2010, extended and revised for JVRB

Our approach is motivated by the medical research project CITmed, which aims at the development of a novel VR platform for the diagnosis and rehabilitation in neurology, neuropsychology, and psychiatry. The fields of application are primarily disturbances of brain functions resulting from stroke, cerebral trauma caused by accidents, and neurological or psychiatric diseases. In order to enable the transfer of the patient's training success to real-life situations, a sufficiently realistic and immersive VR-version of an everyday task has to be used for training. In the CITmed project, this task is shopping in a virtual supermarket.

CAVE installations are known to provide a very high level of immersion, but disqualify for our project because of their high cost and maintenance effort, which is mainly due to the fact that CAVEs are usually driven by one rendering node per view. In contrast, we propose a cost-efficient visualization system that consists of eight standard (non-stereo) touch screen displays arranged in an octagon around the patient, thereby providing a full 360° horizontal view and a simple interaction with the scene. To minimize the stress on the patient we abandon stereo rendering. In order to reduce hardware costs and maintenance effort, our so-called OctaVis solution is driven by a single PC, which consequently is equipped with several graphics processing units (GPUs) (see Figure 1).

This paper is an extended version of [DSB10] and describes our hardware and software architecture, parallelization efforts, as well as low-level and high-level performance optimizations that eventually enable the real-time rendering of complex VR environments on eight views using a single PC.

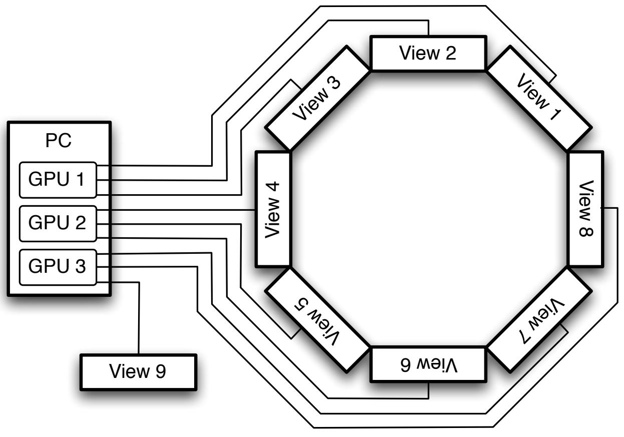

Figure 1: In our OCTAVIS setup a single PC is equipped with three GPUs that drive an operator display and eight displays that are arranged in an octagon to provide a full 360° visualization.

2 Related Work

The standard approach to drive an eight-display system is to use a render cluster consisting of one application server and eight render clients. This solution, however, is complex in terms of both hardware and software. First, it requires nine individual PCs that are properly synchronized and connected through a sufficiently fast network (Gigabit Ethernet, Infiniband). Second, the rendering has to be distributed to the render clients, by either employing a distributed scene graph (e.g., OpenSG [Ope10b]) or by distributing OpenGL commands (e.g., Chromium [HHN+02]).

In contrast, we decided for a single-PC multi-GPU solution for driving our OCTAVIS system. Our workstation is a standard, off-the-shelf PC, equipped with three GPUs that can drive three displays each. This allows us to use nine displays in a configuration as depicted in Figure 1. The main question is how to design the rendering architecture such that the parallel performance of this multi-GPU system is exploited in an optimal manner.

The major graphics vendors already provide solutions for combining several GPUs in order to increase rendering performance (NVIDIA's SLI, ATI's CrossFire). Note, however, that these techniques only support a single graphics output, and hence are not applicable in our multi-view setup. NVIDIA's QuadroPlex is a multi-GPU solution that combines up to four Quadro GPUs in an external case. Two QuadroPlex boxes can therefore be used to drive eight displays, but such a system comes at a price of more than $20,000. ATI's EyeFinity is a technology for driving several displays by one graphics board, and hence is rather a multi-view approach, but not a multi-GPU solution.

In our system we do not make use of any of these techniques, but rather handle each view and each GPU individually. This requires an efficient mechanism for distributing the rendering to the different GPUs.

To this end, we first experimented with higher-level APIs, such as the distributed scene graph OpenSG [Ope10b] and the multi-GPU-aware scene graph OpenSceneGraph [Ope10a]. However, these frameworks turned out not to be flexible enough to give (easy) control over the crucial details affecting multi-GPU performance (as discussed in Section 7).

A distribution of low-level OpenGL commands based on Chromium [HHN+02] was done by Rabe et al. [RFL07], who built a system similar to ours. However, their performance results are rather disappointing, mainly due to the overhead induced by the Chromium layer. An elegant abstraction-layer for parallel rendering is provided by the Equalizer framework [EMP09], which allows for distributed OpenGL rendering using render clusters, multi-GPU setups, or any combination thereof.

Our design goal was to keep the OCTAVIS system as simple as possible—both in terms of hardware and software—since this allows for easy maintenance and future extension. To this end, we do not employ any higher-level API for distributed or parallel rendering, but instead custom-tailor a (simple) low-level OpenGL solution for our target system.

3 Rendering Architecture

Since we do not build our rendering architecture on top of a high-level framework, we have to take care of the distribution of render commands to the different GPUs (Section 3.1) and the data management (Section 3.2). This low-level control will later enable us to employ crucial performance optimization techniques (Sections 4 and 5).

3.1 Distributing OpenGL Commands

Our multi-GPU multi-view architecture consist of three GPUs with three outputs each, which drive the eight VR displays of our OCTAVIS system. We therefore have to be able to address OpenGL commands to a particular display attached to a particular GPU.

GPUs	Views/GPU	Views	FPS NVIDIA	FPS ATI
1	1	1	58.3	273.6
2	1	2	27.5	253.8
3	1	3	19.3	258.8
1	2	2	30	129.7
2	2	4	14.6	121.5
3	2	6	9.8	121.5
1	3	3	–	82.4
2	3	6	–	79.4
3	3	9	–	78.1

Table 1: Comparing frame rates for a 870k triangle model using NVIDIA GeForce 9800 GX2 and ATI Radeon HD 5770 cards, respectively, in several multi-GPU and multi-view setups.

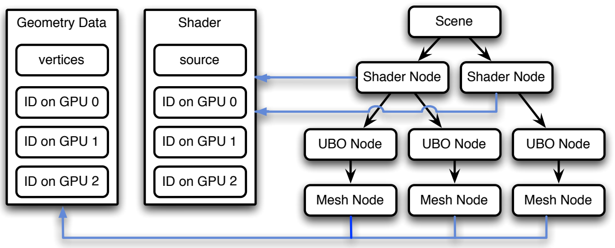

Figure 2: Scene graph nodes share data though shared OpenGL contexts to reduce memory consumption.

At application start-up, we generate a single full-screen window for each of the eight views, with each window having its own OpenGL context. For Unix operating systems distributing render commands is easy, since an individual X-server can be explicitly assigned to each view. In our medical research project, however, external constraints require Windows as an operating system, which does not provide this explicit control.

The Windows Display Driver Model 1.1 used in Windows 7 provides improved support for multi-GPU applications, but it turned out that the actual performance strongly depends on the GPU driver. Our experiments revealed that the NVIDIA driver dispatches *all* OpenGL render commands to *all* available GPUs. This obviously prevents efficient parallelization. In order to address a specific NVIDIA GPU the OpenGL extension `WGL_NV_gpu_affinity` has to be used, but this extension is only available for (expensive) Quadro-GPUs.

In contrast, the ATI driver dispatches render commands to just the one GPU responsible for the current window, which is the GPU attached to the display the window was created on. It therefore turned out to be crucial to create the eight windows at the correct initial position on the respective view. Creating all windows on the first view and moving them to the proper position afterward does not work. When taking this subtle information into account, the ATI driver allows for efficient parallelization between different GPUs.

Table 1 compares the performance scalability of NVIDIA GeForce 9800 GX2 GPUs (two outputs each) and ATI Radeon HD 5770 GPUs (three outputs each) with varying numbers of GPUs and varying numbers of views per GPU. All experiments were performed on a standard PC with an Intel Core i7 930 CPU and running Windows 7.

It is clearly visible that the NVIDIA system scales inversely proportional to the number of views: No matter whether two views are driven by one GPU or by two GPUs, the performance drops by a factor of about two. This is an immediate consequence of the NVIDIA driver sending OpenGL commands to all available GPUs, as also described in [EMP09].

In contrast, the ATI system scales almost perfectly. Rendering one view per GPU gives the same performance on one, two, or three cards. When keeping the number of GPUs fixed, the performance decreases almost linearly with the number of attached views per GPU. Hence, the ATI system can fully exploit the parallelization between multiple GPUs. Because of these reasons we decided for ATI Radeon HD 5770 GPUs for our OCTAVIS rendering system, which have recently been replaced by ATI Radeon HD 5850 cards.

3.2 Data Management

Thanks to our simple single-PC architecture we do not have to distribute scene data or render states over the network to individual render clients. However, in order to render the scene in our multi-view application each OpenGL context (i.e., each view) needs access to the scene data, such as, e.g., geometry, textures, and shaders. In order to minimize memory consumption, we do not duplicate the scene data for each view, but instead store one copy per GPU only, which is then shared by all views attached to this GPU.

This behavior can easily be implemented by shared OpenGL contexts. We incorporate this functionality into a very simplistic scene graph with standard nodes for shaders, shader uniforms, and triangle meshes. Shader nodes, for instance, then store a reference to a shader object only. The shader object in turn stores the shader-data on each available GPU (but *not* for each

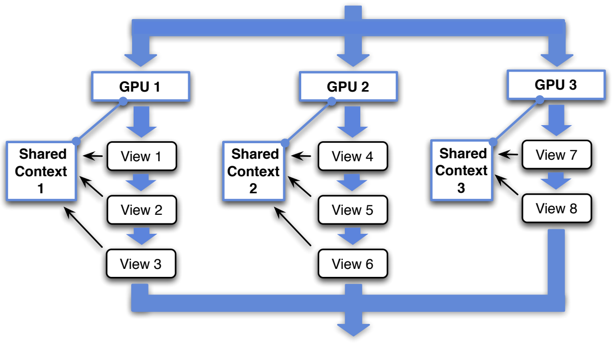

Figure 3: All views attached to the same GPU share a render context and are drawn consecutively, while the rendering is parallelized over the available GPUs.

GPUs	Views/GPU	Views	FPS VA	FPS VBO	FPS RO
1	1	1	38.8	214.3	344.2
2	1	2	22.3	214.6	344.8
3	1	3	14.2	214.7	344.7
1	2	2	19.2	107.7	171.3
2	2	4	10.2	107.5	171.2
3	2	6	7.0	107.5	171.1
1	3	3	12.7	71.6	113.6
2	3	6	7.2	71.4	113.5
3	3	9	4.7	71.3	113.3

Table 2: Frame rates for different optimizations (vertex arrays, vertex buffer objects, cache-friendly re-ordering), using ATI Radeon HD 5770 GPUs.

individual view). See Figure 2 for an illustration. In our case, where each GPU drives up to three views, the memory consumption is reduced by a factor of three.

4 Low-Level Optimizations

In order to optimize rendering performance one has to identify and eliminate the typical performance bottlenecks: CPU load, data transfer from CPU to GPU, and GPU load. In a multi-view multi-GPU environment, even more attention has to be paid to these issues.

Rendering geometry in *immediate mode* quickly makes the application CPU-bound due to the massive amount of `glVertex()` function calls. We therefore store vertex positions and triangle indices in *vertex arrays* (VA), which allows to render meshes with a single function call and thereby eliminates the CPU bottleneck. However, in our multi-view setup the data has to be transferred from main memory to the GPU eight times (for each view) in each frame, such that data transfer immediately becomes the bottleneck.

Data transfer can be eliminated by storing the vertex arrays in *vertex buffer objects* (VBO) on the GPU. This turned out to be absolutely crucial in our multi-view setting. The respective data storage follows the same shared context paradigm as described in the previous section. With VBOs the bottleneck is no longer data transfer, but the per-vertex computations of the GPU.

This GPU load can be reduced by caching computations performed for individual vertices. If a triangle is rendered and one of its vertices has been processed before and is still in the cache, these computations can be re-used. To maximize cache-hits we re-order the individual vertices and triangles of the mesh using the method described in [YLPM05], which (depending on the model) yields a significant performance gain.

Finally, in order to optimally exploit all available GPUs, the render traversal is parallelized: Each GPU is served by a dedicated render thread that processes all views attached to that GPU in a serial manner. Parallelizing over the (two or three) views on the same GPU did not increase performance. Figure 3 gives an overview of the rendering process.

Table 2 compares the different optimization techniques using three ATI Radeon HD 5770 GPUs. For this experiment we used an early version of our VR supermarket, consisting of about 1.7M triangles. In this scene all instances of a particular object (typically several copies of one shopping item in a shelf) are merged into a single triangle mesh. This results in 75 meshes in total, which are stored either in vertex arrays or vertex buffer objects, respectively.

When the geometry is stored in vertex arrays, but not in VBOs, the geometry data is transferred to the GPU for each active view. Consequently, the performance drops with each additional view, even when they are attached to different GPUs. Storing the geometry in VBOs eliminates the transfer costs, which then yields the convincing scaling behavior discussed above. After re-ordering vertices and triangles for each individual VBO, our rendering architecture is able to visualize the 1.7M triangle scene on nine displays at a rate of more than 100 frames per second.

5 High-Level Optimizations

The performance optimizations described in the previous section were sufficient for the virtual supermarket consisting of 1.7M triangles. However, in order to further increase the realism and the immer-

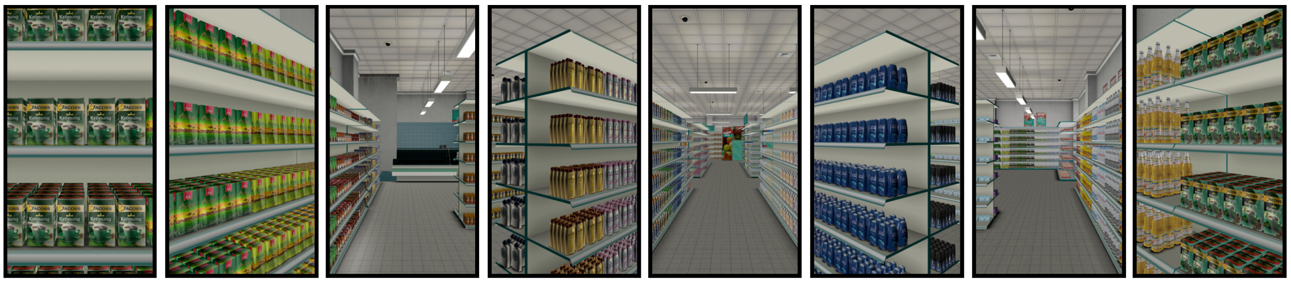

Figure 4: Eight views of the VR supermarket (4M triangles), corresponding to an "unfolded octagon". This model was used for the optimizations described in Sections 5.2 and 5.3.

sion in the CITmed project, we switched to a more detailed VR supermarket. The new model was constructed from 680 individual objects (furniture, shopping items), from which 94k instances were created, eventually resulting in 4M triangles (see Figure 4).

Furthermore, it should be possible to interact with each individual object instance, for example, in order to buy individual products by simply picking them. Note that this is not possible when storing all instances of a particular product within a single VBO, since then individual instances cannot easily be turned off after having been bought. Representing each object instance by a separate scene graph node, i.e., splitting the scene from 680 objects into 94k objects, allows for simple interaction with object instances, but reduces the performance to about 1 fps due to CPU load.

At this point we apply three high-level optimizations to achieve our goals: Geometry instancing allows naturally to handle scenes with many duplicated objects (Section 5.1), view-frustum culling (Section 5.2) and multi-GPU load balancing (Section 5.3) further increase performance. All experiments shown in this section have been performed using three ATI Radeon HD 5850 GPUs.

5.1 Geometry Instancing

Recent versions of OpenGL provide hardware-accelerated geometry instancing, which allows to render multiple instances of a single object with a single function call. This technique needs two data streams, one supplying the object data (geometry, textures, shaders) and the other providing the positions/orientations of the individual instances. For each instance a shader transforms the geometry according to the given instance transformation. Like this geometry and texture data have to be stored once per object only, instead of once per instance.

For maximum performance, both the object data and the instance data have to be stored on the GPU, since otherwise the data transfer immediately becomes the bottleneck. We therefore store all instance transformations within a single *Uniform Buffer Object* (UBO) on the GPU. Another UBO holds binary on/off information for each object instance, which we use for turning off objects once they have been bought.

Geometry instancing allows for conceptually clean and efficient handling of object instances. In terms of performance, however, geometry instancing does not make a difference for the high resolution supermarket of 4M triangles. Storing all object instances in 680 VBOs or employing geometry instancing both yields 25 fps. The main advantages of geometry instancing are therefore (i) the ability to interact with individual object instances (pick or buy each single product) and (ii) the largely reduced consumption of GPU memory. Geometry instancing reduces GPU storage from 243 MB per GPU for single VBOs down to 8 MB per GPU (2 MB mesh data, 6 MB instance data). This significant reduction is mainly due to our highly repetitive supermarket scene with a ratio of 680 objects to 94k object instances.

5.2 View-Frustum Culling

To achieve real-time performance even for the more complex supermarket model of 4M triangles, we exploit the fact that all object instances are arranged spatially close to each other, namely in the same product shelf. This allows us to compute a bounding box around each group of instances and to perform view-frustum culling during scene graph traversal for each of the eight views. We compared CPU-based view-frustum culling to GPU-based occlusion culling, and for our ATI GPUs the former turned out to be slightly more efficient than the latter.

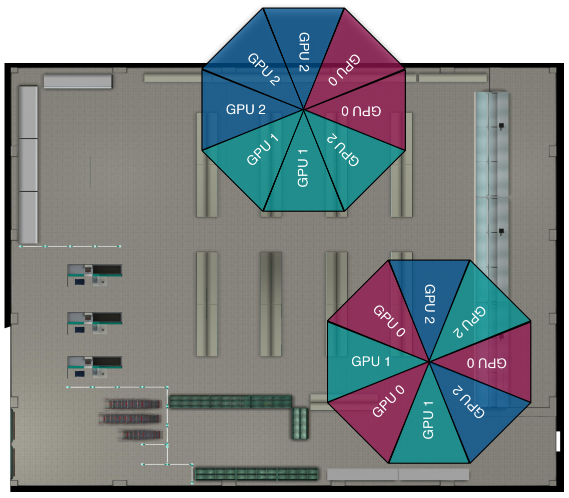

Figure 5: Top-view of the supermarket scene, showing *neighboring* view configuration (top) and *interleaved* view configuration (bottom).

For the 4M-triangle supermarket our CPU-based view-frustum culling increases the performance from 68 fps to 205 fps for single-view single-GPU rendering, and from 25 fps to 56 fps for eight views rendered using three GPUs.

5.3 Load-Balancing

View-frustum culling can lead to a considerable performance gain—depending on how much of the scene is outside the frustum and therefore is culled. Since this proportion varies with camera position and viewing direction, frustum culling inevitably leads to an unbalanced load distribution for the eight individual views of our multi-view setup. For instance, at a location close to a wall in our supermarket some views mainly have to render the wall, while others have to process almost the whole scene. For due to view synchronization the slowest GPU determines the frame rate of the overall rendering, some kind of load balancing should be employed.

Since each GPU has to render two or three views, we can try to counter-balance the varying load per view on the level of GPUs. The most straightforward assignment of views to GPUs is the one shown in Figure 1, where each GPU is responsible for a set of *neighboring* views. Assigning the views in an *interleaved* manner (Figure 5), however, results (in general) in a much better load balancing.

View Configuration	GPU 0	GPU 1	GPU 2
Neighboring	205	56	109
Interleaved	91	74	154

Table 3: Frame rates of each individual GPUs for different GPU/view configurations.

Table 3 compares the rendering performance for the two view configurations show in Figure 5, with camera positions being randomly positioned in the scene. The timings are taken separately for each of the three GPUs, where the slowest GPU (in this case GPU 1) determines the overall frame rate. This experiment clearly shows the interleaved view configuration to yield better load balancing, resulting in a performance gain from 56 fps to 74 fps. Note that this result is almost independent of the location in the supermarket, leading to a general performance improvement.

6 User Study

To evaluate our OCTAVIS setup the Department of Psychology at Bielefeld University conducted an empirical within-subject study [PK10]. Comparing our platform to a single-screen desktop with mouse-keyboard interaction they measured the subject's sense of presence during several consecutive search tasks. To this end, the participants filled out the Witmer and Singer questionnaire [WS98] after the experiment.

The exact procedure was as follows. After a short time of customization to the navigation concept the subject was asked to perform the following six actions:

1. "Go to the meat counter and count the sausages"
2. "Go to the cheese counter and count the cheese"
3. "Go to the cereal shelf"
4. "Position yourself in front of the egg shelf"
5. "Go to the whiskey shelf"
6. "Leave the supermarket"

After each subtask the participant stopped and reported his success to the advisor before he was allowed to proceed. These assignments had to be accomplished in both setups. The order in which the setups were tested by the same person was randomized.

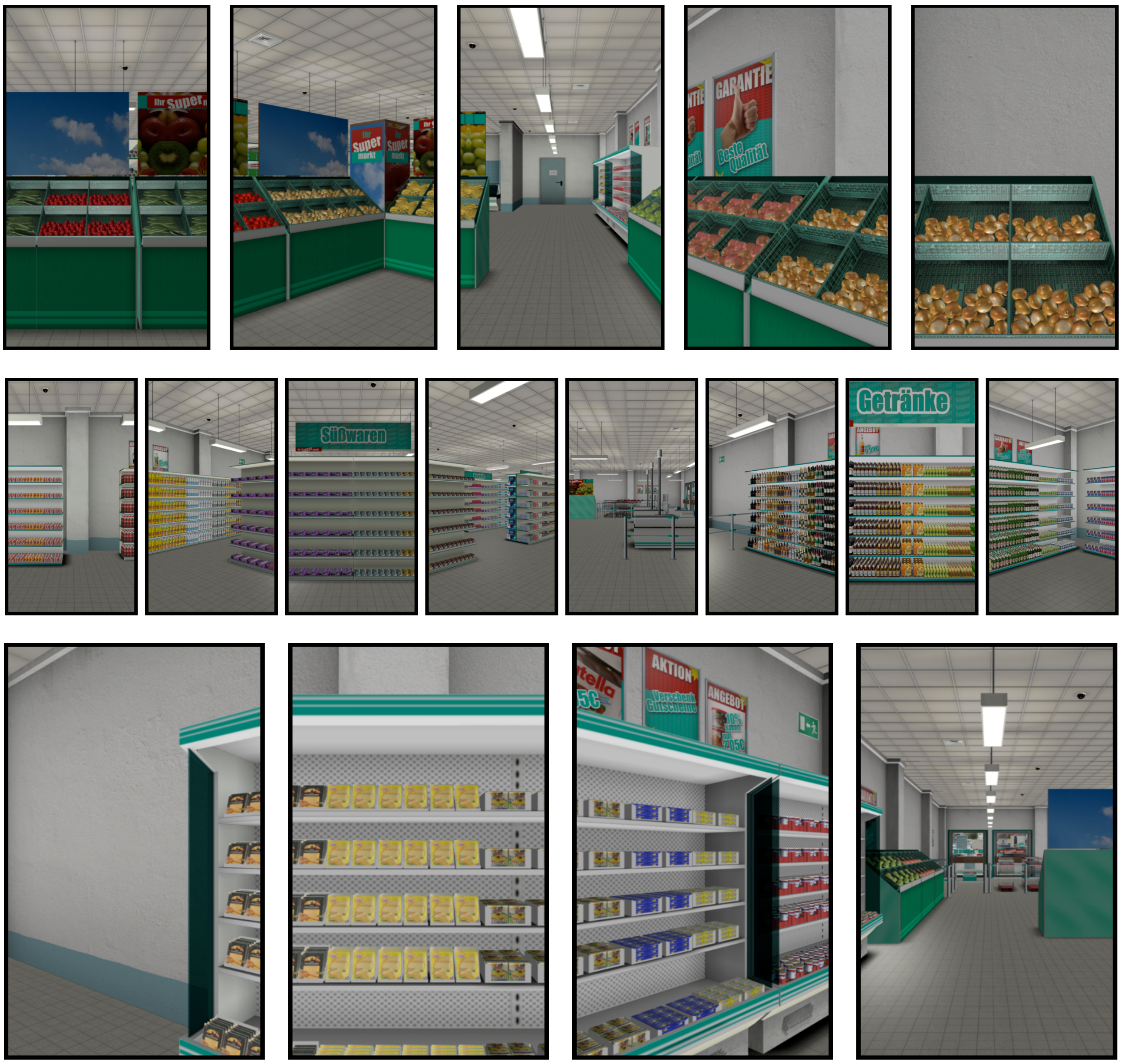

Figure 6: Some example views of the high-detail supermarket, consisting of about 94k object instances and 4M triangles. Each row shows several views for different viewing positions in the scene.

27 subjects in the age of 18–35 were tested in total. The results show a significantly higher sense of presence in the OCTAVIS-condition ($Z = -4.37$, $p < .0001$). The mean values for the presence score are $M = 130.52$ ($sd = 17.36$) in the OCTAVIS-condition and $M = 104.26$ ($sd = 15.2$) in the single-screen condition.

Due to our use of touch screen displays we could not employ seamless displays, resulting in clearly visible seams between the eight screens. Particularly asked about the effect of these seams in an additional custom questionnaire, 21 subjects stated them to be not disturbing at all, 3 were disturbed slightly, and 3 were confused by them.

7 Discussion

Our low-level and high-level optimizations result in a multi-view rendering system that is capable of visualizing a complex 4M triangle scene on eight displays at a rate of 74 fps, which corresponds to 2.4G triangles/second. At the same time, our system allows for fast interactions on an object instance level. Figure 6 shows some impressions of our current model.

As mentioned in Section 2, we experimented with the two popular scene graph libraries OpenSceneGraph [Ope10a] and OpenSG [Ope10b], but did not achieve a comparable performance, mainly due to the following reasons:

OpenSG is specialized to distributed rendering, therefore our implementation used eight local render server processes to drive the eight displays. Since each render server requires a full scene copy, this consumes significantly more memory than our shared contexts and therefore does not allow for highly complex scenes. Moreover, due to sub-optimal window creation and setup (see Section 3.1) OpenGL commands are dispatched to all GPUs, which slows down the rendering to about 1 fps.

OpenSceneGraph correctly creates and initializes windows, supports for shared OpenGL contexts, and allows for precise control of VBOs. However, its rendering performance is still only about 70% of ours, which we assume is due to the higher overall complexity of this scene graph system.

8 Conclusions

Our experiments clearly demonstrate the potential of a multi-view rendering system based on a single PC equipped with multiple GPUs. However, the results also indicate that the multi-GPU performance can crucially depend on a few seemingly minor implementation details, on optimization techniques, and on GPU drivers and operating systems.

This paper tries to provide a recipe for circumventing common pitfalls and instead achieving a high performance multi-GPU system. In terms of hardware, our proposed OCTAVIS system is cheap and easy to maintain. In terms of software, the rendering architecture scales almost perfectly thanks to few but carefully done performance optimizations.

Future work includes more detailed geometric models and more realistic rendering, for instance by using screen-space ambient occlusion or other real-time global illumination techniques. Further performance optimizations might result from deferred shading or more sophisticated spatial data structures and sorting.

Acknowledgments

The authors are grateful to Christian Fröhlich and Bernhard Brüning for their support with the initial OpenSG-based version of the rendering system. Mario Botsch is supported by the Deutsche Forschungsgemeinschaft (Center of Excellence in “Cognitive Interaction Technology”, CITEC). Eugen Dyck, Holger Schmidt, and Martina Piefke are supported by the EFRE project “CITmed: Cognitive Interaction Technologies for Medical Applications”.

EUROPÄISCHE UNION
Investition in unsere Zukunft
Europäischer Fonds
für regionale Entwicklung

References

[DSB10] Eugen Dyck, Holger Schmidt, and Mario Botsch, OctaVis*: A Simple and Efficient Multi-View Rendering System*, Proceedings of GI VR/AR Workshop (2010), 1–8.

[EMP09] Stefan Eilemann, Maxim Makhinya, and Renato Pajarola, *Equalizer: A Scalable Parallel Rendering Framework*, IEEE Transactions on Visualization and Computer Graphics **15** (2009), no. 3, 436–452, ISSN 1077-2626.

[HHN$^+$02] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter Kirchner, and James T. Klosowski, *Chromium: a stream-processing framework for interactive rendering on clusters*, ACM Transactions on Graphics (SIGGRAPH) **21** (2002), no. 3, 693–702, ISSN 0730-0301.

[Ope10a] OpenSceneGraph, *OpenSceneGraph*, `http://www.openscenegraph.com`, 2010.

[Ope10b] OpenSG, *OpenSG*, `http://www.opensg.org`, 2010.

[PK10] Martina Piefke and Sina Kühnel, *Empirie- und Beobachtungspraktikum: Physiologische Psychologie, Departement of Psychology, Bielefeld University*, Winter Term 2009/2010.

[RFL07] Felix Rabe, Christian Fröhlich, and Marc Erich Latoschik, *Low-Cost Image Generation for Immersive Multi-Screen Environments*, Workshop of the GI VR & AR special interest group, 2007, pp. 65–76.

[WS98] Bob G. Witmer and Michael J. Singer, *Measuring Presence in Virtual Environments: A Presence Questionnaire*, Presence: Teleoperators and Virtual Environments **7** (1998), no. 3, 225–240, ISSN 1054-7460.

[YLPM05] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha, *Cache-Oblivious Mesh Layouts*, ACM Transactions on Graphics (SIGGRAPH) **24** (2005), no. 3, 886–893, ISSN 0730-0301.

Citation

Eugen Dyck, Holger Schmidt, Martina Piefke, Mario Botsch: OctaVis*: Optimization Techniques for Multi-GPU Multi-View Rendering*, Journal of Virtual Reality and Broadcasting, 9(2012), no. 6, November 2012, urn:nbn:de:0009-6-35119, ISSN 1860-2037.