Eurographics Symposium on Parallel Graphics and Visualization (2012)
H. Childs and T. Kuhlen (Editors)

Multi-GPU Image-based Visual Hull Rendering

Stefan Hauswiesner, Rostislav Khlebnikov, Markus Steinberger, Matthias Straka and Gerhard Reitmayr

Graz University of Technology, Institute for Computer Graphics and Vision

Abstract

Many virtual mirror and telepresence applications require novel viewpoint synthesis with little latency to user motion. Image-based visual hull (IBVH) rendering is capable of rendering arbitrary views from segmented images without an explicit intermediate data representation, such as a mesh or a voxel grid. By computing depth images directly from the silhouette images, it usually outperforms indirect methods. GPU-hardware accelerated implementations exist, but due to the lack of an intermediate representation no multi-GPU parallel strategies and implementations are currently available. This paper suggests three ways to parallelize the IBVH-pipeline and maps them to the sorting classification that is often applied to conventional parallel rendering systems. In addition to sort-first parallelization, we suggest a novel sort-last formulation that regards cameras as scene objects. We enhance this method's performance by a block-based encoding of the rendering results. For interactive systems with hard real-time constraints, we combine the algorithm with a multi-frame rate (MFR) system. We suggest a combination of forward and backward image warping to improve the visual quality of the MFR rendering. We observed the runtime behavior of the suggested methods and assessed how their performance scales with respect to input and output resolutions and the number of GPUs. By using additional GPUs, we reduced rendering times by up to 60%. *Multi-frame rate viewing can even be ten times faster.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors

1. Introduction

Our main application scenario is an augmented reality system which allows users to view themselves on a screen from arbitrary directions. Moreover, the system features a mirror mode which sets the virtual viewpoint to reflect the user like a conventional mirror does. This system is suitable for various mixed reality applications, like a virtual mirror or teleconferencing (see Figure 1). As the user immediately sees the rendering of himself, high rendering performance and low latency are crucial for a satisfactory user experience. This is is also important to avoid simulator sickness during extended sessions. Moreover, user-interaction through gestures benefits from low latency.

In our system, a set of ten cameras is mounted around the area where the user is allowed to move. The cameras provide a color video stream. As the user can be segmented using background subtraction, the image-based visual hull (IBVH) algorithm can be applied for the purpose of rendering the user from an arbitrary viewpoint. The IBVH algorithm performs a depth map reconstruction directly from the silhouette images, and is therefore a very lightweight approach to rendering [HSR11].

The high performance to cost ratio of a single PC equipped with multiple GPUs makes it an an attractive platform for such complex interactive visualization tasks. Moreover, data transfers from main memory to GPUs are fast compared to network transfers. Several visual hull algorithms have been modified in order to run on several PCs or GPUs in parallel. These algorithms usually involve explicit, view-independent intermediate results in the form of meshes or voxel grids that can be computed independently before merging them in a final step. In contrast, IBVH rendering does not have such a representation, and is therefore not parallelizable with standard methods.

The contributions of this paper are three ways of distributing the workload of an IBVH pipeline over several GPUs. After analyzing the computation times and data flow of an existing single-GPU IBVH pipeline, we derive which stages are suit-

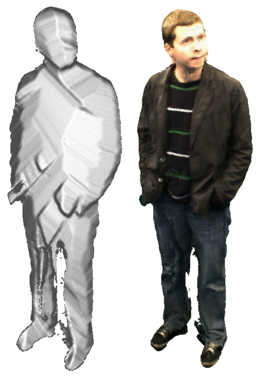


Figure 1: *IBVH rendering of a person in our system. Left: phong-shaded depth map, right: same depth map rendered with view-dependent texture mapping.*

able for parallelization. We start with a sort-first approach, which is simple but has drawbacks. Then, we introduce a sort-last approach by regarding cameras as scene objects. In addition, we suggest a compact buffer representation that reduces the bus traffic. The last approach is a multi-frame rate setup that decouples the viewing process from the image generation to achieve high frame rates even for large resolutions. To improve visual quality, we suggest a combined forward- and backward warping method. We evaluated all approaches by measuring and comparing runtimes for different configurations.

2. Related work

Extracting geometry from a set of calibrated camera images when a foreground segmentation is available is a common technique in computer vision called shape-from-silhouette (for example, [YLKC07]). When using this information for display, the process is often called visual hull rendering. Visual hulls can be reconstructed very efficiently, which makes them ideal for interactive systems.

2.1. Visual hull rendering

Previous work in the field can be categorized according to the data representation that is reconstructed from the input silhouettes. Frequently, an explicit data representation, like meshes or voxel grids is extracted. A voxel-based reconstruction is shown in [NNT07] using shader programs. Meshes are usually more efficient [Boy03, dAST*08] for both reconstruction and rendering.

But also other methods have been developed that do not have an explicit data representation. CSG-based approaches use depth-peeling of silhouette cones for correct results [Li04]. Texture-mapping approaches render the cones and rely on alpha-tests to carve the final silhouette [LCO06]. [GG07] describe a plane-sweep algorithm to find point-correspondences on the GPU and use it for triangulation. While these approaches may be mapped to the conventional rendering pipeline, they are bound to scale badly with resolution and depth complexity, as excessive fragment read/write operations in device memory are required.

When only the rendering of novel viewpoints is required, and not an explicit reconstruction, the detour of intermediate geometry can be avoided to reduce latency. To directly create novel views from silhouette images, the image-based visual hull (IBVH) [MBR*00] method was introduced. It involves on-the-fly ray-silhouette intersection and CSG operations to recover a depth map of the current view.

Many extensions of the original approach have been developed. [SSS*02, FWZ03] extend IBVH with photo-consistency, resulting in improved quality. [YZC03] use IBVH and find body parts to improve rendering of convex parts. [WFEK09, GHKM11] show recent GPU implementations of the IBVH algorithm and the use of silhouette segment caching to speed up intersection tests.

2.2. Parallel visual hull rendering

Parallel visual hull rendering is usually performed by using an explicit data representation. This way it is easier to compute intermediate results in parallel and merge them afterwards. Often, voxels are used for this [LBN08, TLMpS03, GM03, WTM06]. However, voxels can not efficiently provide the desired output resolution for our system and induce heavy bus traffic between the rendering nodes.

The Grimage system [FMBR04, AFM*06] utilizes a PC cluster for visual hull reconstruction. Their method is mesh-based, which makes it more efficient than a voxel representation while also being view-independent. However, the additional latency that comes from computing an explicit data representation remains. In this case this amounts to triangulating the viewing edges.

The IBVH algorithm does not have an explicit data representation and is therefore harder to parallelize on multiple GPUs or CPUs. The work of [FLZ10] utilizes two GPUs to compute two images separately: one for each eye in their stereo setup. In contrast, the goal of our work is to distribute the rendering of one image over multiple GPUs.

2.3. Multi-frame rate rendering

Multi-frame rate rendering [SBW*07] decouples display updates from image generation in a pipeline with asynchronous communication. The display update stage can guarantee fast viewing frame rates and nearly latency-free response to interaction, while one or multiple GPUs in the backend stage can produce new high quality images at their own, slower pace.

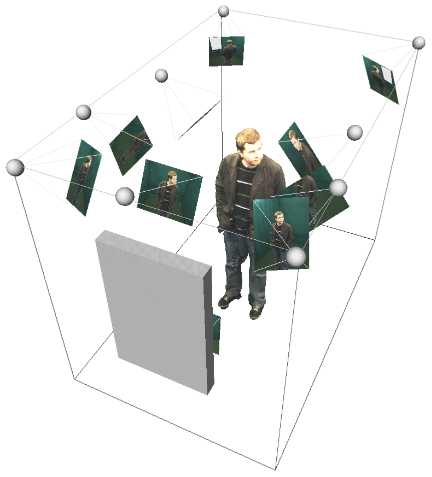


Figure 2: *A rendered illustration of the capturing room and its cameras.*

Multi-frame rate systems make extensive use of temporal/frame coherence, because they use the last rendering result to synthesize novel viewpoints until a new image becomes available. [SvLBF09] allow for motion in the scene as long as it can be described by a transformation matrix. [MMB97] introduced image warping to compensate for jitter induced by latency.

Prerequisites

The capturing room that is used for this project consists of a 2x3 meter footprint cabin with green walls [SHRB11]. Ten cameras are mounted on the walls: two at the back, two at the sides and six at the front. The cameras are synchronized and focused at the center of the cabin, where the user is allowed to move freely inside a certain volume (see Figure 2). All cameras are calibrated for their projection matrices and connected to a single PC via three Firewire cards. The PC is equipped with four Nvidia GTX 480 GPUs, each plugged into a PCIe x16 slot. We decided to use only one PC in order to avoid network latency. The output device is a 42" TV that is mounted to the front wall in a portrait orientation.

In such a scenario, silhouettes can be extracted from the camera images quickly and robustly by background subtraction. Silhouettes make novel view synthesis very efficient. The image-based visual hull (IBVH) algorithm creates a depth map from a novel viewpoint [HSR11]. See Figure 1 for an example output. Such a depth map can be textured with the camera images for realistic image-based rendering. To achieve repeatability during our evaluations, we use a set of recorded AVI videos instead of live video streams.

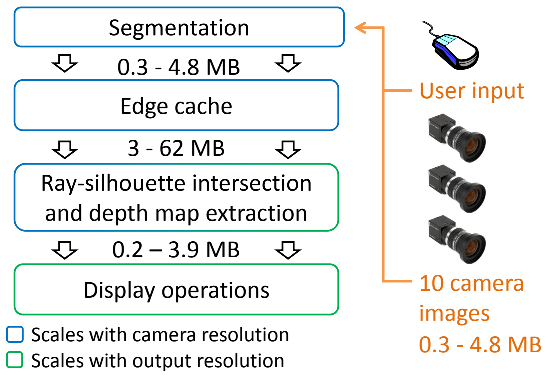


Figure 3: *A typical image-based visual hull rendering pipeline. The first part consists of segmentation and undistortion. The second part builds a cache data structure to speed up the third part, the ray-silhouette interval extraction and interval intersection. The last part is responsible for texturing and display. The data traffic between stages is indicated for a single rendering pass. The value ranges cover all camera resolutions and a maximum output resolution of 2 Megapixels.*

3. Image-based visual hull rendering

The general architecture of our system consists of several main modules (see Figure 3). Ten cameras deliver images with a resolution of 320x240, 640x480 or 1280x960 pixels.

The following section describes the general layout of our visual hull rendering pipeline on a single GPU. The pipeline starts with the segmentation module.

Segmentation The camera images are uploaded to the GPU and segmented there by applying background subtraction. We assume a static background that is captured before the user enters the scene, and a contrasting foreground. We use background subtraction in normalized RGB space in order to handle shadows correctly. The subtracted images are thresholded and holes are closed by applying morphological dilation and erosion to the segmented image.

Edge extraction and segment caching From each silhouette image, all edges are transformed to 2D line segments. Every pixel is checked for whether it lies at the border of the silhouette, and a line segment is emitted if it does. Pixels are processed in parallel, and interior silhouette edges are included, because they help to identify regions where the user's body forms a loop, e.g., with his hands.

Since the IBVH algorithm performs many line-line intersection tests, a caching data structure can speed up the process of identifying potentially intersecting line segments [WFEK09]. Line segments are assigned to bins that share a common angular range around their epipole.

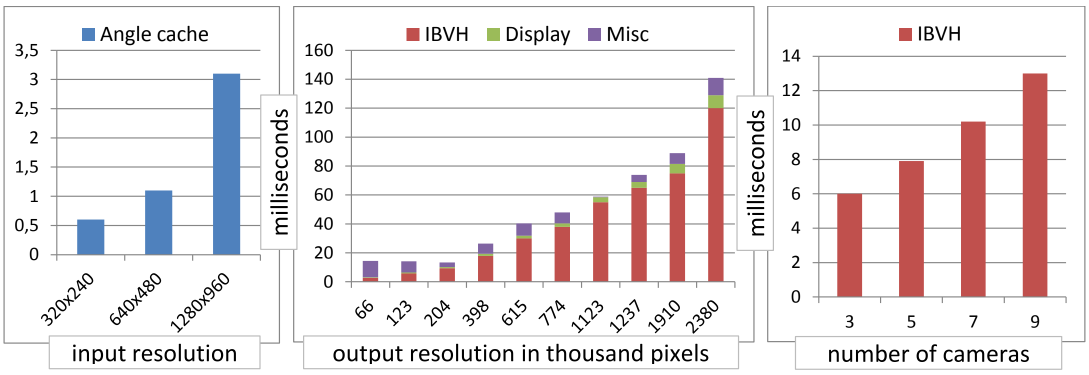


Figure 4: *Kernel execution times on a single GPU with different numbers of cameras and different input (=camera) and output (=screen) resolutions. Angle caching scales with the camera resolution, while the other kernels scale with the output resolution. The IBVH algorithm also scales with the number of cameras. Unless specified otherwise, the timings are given for* 10 *cameras, an input resolution of* 640x480 *pixels and an output resolution of* 480x870 *pixels.*

The image-based visual hull algorithm At this stage, a viewing ray is created for every pixel. Rays start from the desired viewpoint and intersect the image plane at the pixel's location and end at the back plane. The subsequent steps are performed for each ray and camera image. The viewing rays are projected on each camera's image plane. The angle of the projected viewing ray can be used to index the correct bin in the line segment cache. All segments in the bin are checked for 2D line intersection, using the line segment itself and the projection of the viewing ray. Intersections are sorted and lifted back onto the 3D ray by back projection which results in a list of intervals along each viewing ray. These intervals have to be intersected in order to find which intervals along the ray are located inside the object. From these intervals, the front-most interval starts with the front-most ray-object intersection which in turn yields the depth buffer value at this pixel.

Display The depth map from the last stage can now be used for display. This stage is not part of the core algorithm and may vary in complexity depending on the task at hand. A color value can be computed for each pixel, or the depth value can be used directly to compute occlusions or act as a proxy geometry for display. For this work, we use a simple phong-shading kernel to visualize the depth values. The result of this stage is the output image that is written to the frame buffer on the primary GPU.

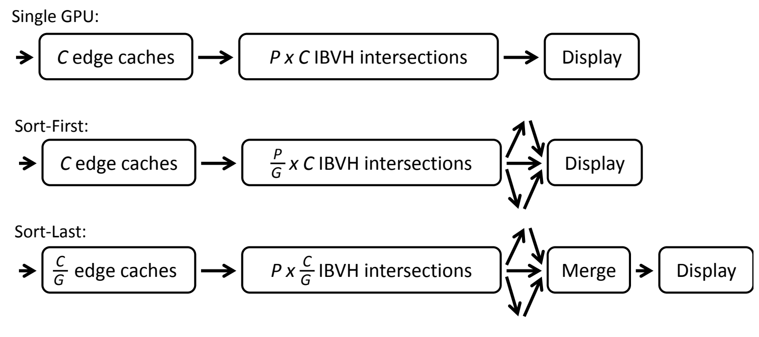
3.1. Data flow

Before the first stage begins, the camera images are read from the driver and uploaded to the GPU. There, the images are segmented and passed to the angle binning kernels. The amount of data that is passed is proportional to the camera resolution and camera count. After angle binning, the bins are passed to the intersection kernel. The size and number of the angle bins must be chosen such that the overall capacity (size multiplied by number of bins) is sufficient to hold all line segments of a camera. Choosing many small bins is beneficial for performance, while few large bins require less memory. Usually, this part of the pipeline creates the largest data flow, which makes it a bad spot for sharing data between GPUs. After the IBVH kernel, the desired depth map is computed and passed to a shading kernel. This final stage and the associated data flow has only minor impact on the overall performance and therefore marks the end of the pipeline that is analyzed here. Figure 3 shows the amount of data that flows through our pipeline.

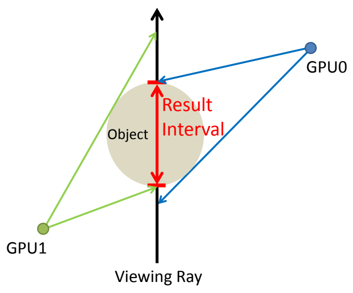
3.2. Scaling with inputs

Before parallelizing the IBVH pipeline, we need to understand how the execution times of the stages scale with respect to their inputs. First, the segmentation and edge cache generation extract information from the camera images. These tasks are defined per camera image, which means the runtime is proportional to the number of cameras and their resolution. Next, the IBVH core computes a depth map from the cached edges. This stage scales with the output resolution, and, to a lesser extent, with the number of edges. The number of edges is driven by the number of cameras and their resolutions. The final display step scales only with the output resolution. Figure 4 shows the runtime of the stages for different resolutions and number of cameras.

From this data we can derive that for increasing display resolutions, the computation of the depth map (the core IBVH step) becomes the dominant performance factor. This part also scales with the number of cameras. The resolution of the camera images on the other hand does not have a strong impact. This means that in any case we want to distribute the depth map workload over all GPUs. For the other stages



(a)



(b)

Figure 5: *(a) different configurations of a synchronized multi-GPU IBVH pipeline and a single-GPU pipeline as a reference. C denotes the number of cameras, P the number of output pixels, G the number of GPUs. Sort-last configuration (b): a viewing ray intersects silhouette cones from two GPUs. The intervals are intersected again to compute the result interval.*

of the pipeline it might be acceptable to compute them for all inputs on every GPU to avoid data traffic. Data transfers require time and synchronization, and therefore might decrease the performance unnecessarily. This is especially true for smaller resolutions.

4. Parallelizing the image-based visual hull algorithm

We suggest several multi-GPU configurations of our pipeline. The main difference between them is the number and placement of synchronization points. At a synchronization point, the GPUs wait until the current stage of the pipeline has been completed on all GPUs. Afterwards, data is shared between the GPUs to allow later stages to have access to all intermediate results. Figure 5 (a) shows these synchronization points and the amount of work that needs to be performed by each stage.

The core part of the pipeline is the IBVH stage that produces a depth map from a set of angle bins. It requires most of the computation time. All parallelization configurations therefore focus on how to split and distribute this stage. The segmentation, angle binning and shading step are the same for all configurations.

4.1. Sort-first configuration

This configuration uses a sort-first workload arrangement. The IBVH algorithm is defined as a per-pixel operation, which makes it very similar to raycasting in terms of how independent the computations are from each other. The workload can be easily split between the GPUs by dividing the output image into equally sized boxes. The IBVH stage can run in parallel. After IBVH computation, the synchronization point is reached and data is shared. The final display stage joins the subimages to form the output image. See Figure 6 (a) for an illustration of the workload distribution.

Sort-first parallelization is easy to implement, but suffers from the memory transfer that is required to pass the input data to the computing nodes. In our case, this means that all camera images have to be uploaded on all GPUs. Moreover, to achieve maximum scalability, the exact location where the screen is split needs to be determined by a robust load balancing mechanism [MCEF08].

4.2. Sort-last configuration

Sort-last approaches usually distribute the scene objects (models, or triangles, or bricks of volumetric data) across the computing nodes. Nodes render their chunk of data in parallel, and send the result image plus depth information to one or more compositing nodes. Compositing is more complex than the merging step of sort-first approaches: it involves testing or sorting all fragments according to their depth.

Visual hull rendering is usually focused on a single scene object: a person in our case. Even systems that can capture multiple objects in front of their cameras usually can not distinguish between the objects before actually rendering them. To distribute workload across multiple nodes, we therefore suggest to assign a subset of the cameras to each node (=GPU).

For example, when two GPUs are used, each can process five camera images to achieve a total of ten. Each GPU computes the visual hull of only a subset of all available cameras (see Figure 5 (b)). However, unlike conventional rendering, such a subset is not a stand-alone rendered image with depth. Instead, it is only an intermediate state in the sequence of ray-silhouette intersections that make up the IBVH algorithm. This state consists of a list of intervals along each viewing ray that must not be collapsed to a single depth value until all subsets are intersected.

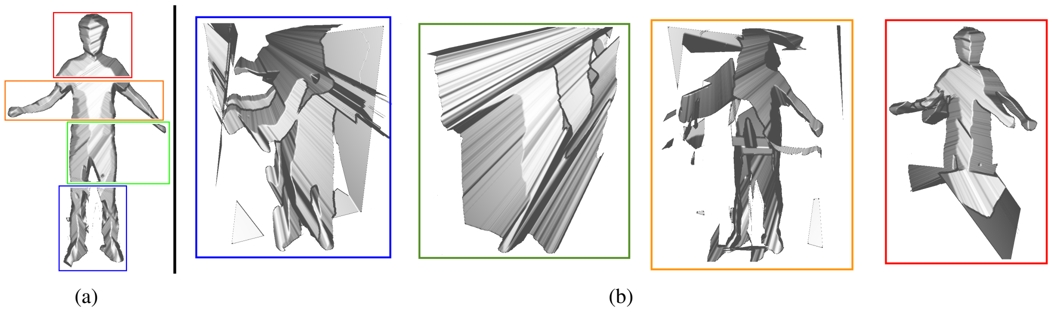


Figure 6: *Sort-first (a) and sort-last (b) configuration intermediate results of four GPUs rendered with phong shading.*

In our system we have ten cameras, which means that each GPU has to handle a maximum of five when multi-GPU computing is desired. For the application of rendering people, we found that two intervals along each viewing ray are sufficient. Each interval can be described by two float values that denote the boundaries of the interval along the viewing ray. This means that each GPU produces a buffer that has the resolution of the output image and four float values per pixel. See Figure 6 (b) for an illustration of the intermediate results.

The suggested method corresponds to a sort-last approach, where cameras are regarded as scene objects. In contrast to conventional sort-last approaches, the compositing step is more involved than testing or sorting depth values. The intervals at each pixel need to be intersected to produce the final visual hull. After intersecting, the first interval that describes the object surface can be used to produce the output depth value.

Compact interval representation Sort-last approaches can be optimized by only transferring image parts that contain an object [MCEF08]. This is called *sparse* sort-last and usually achieved by tight fitting bounding boxes. Unfortunately, for sort-last IBVH all pixels within the viewing volume produce data. As a result, the data traffic after rendering is considerable and prevents the algorithm from scaling well with the output resolution and the number of GPUs.

The interval data that are transfered between the GPUs correspond to depth values generated from a subset of the cameras. While depth buffer compression techniques are often tuned for triangle data [HAM06], the interval data for IBVH rendering shows a different structure, as depicted in Figure 6. Nevertheless, more general depth buffer compression algorithms can also be used for the ray-like depth structure found in the interval data buffers.

To efficiently compress and uncompress the data without hardware supported compression, we use a method similar to Orenstein et al. as described in [HAM06]. We divide the interval data into blocks of 8*x*8 pixels. For each block we pick a representative pixel and distribute its interval values across the entire block. Every other pixel within the block computes the difference of its values to the representative values. The differences are stored with reduced bit length into a global buffer.

To tune the compression ratio, we support different bit lengths per value. To decide which bit length to use, all pixels in the block publish their required bit length and the maximum is taken. With lossless compression, data traffic rates can be reduced by approximately 33% to 45% depending on the resolution. But there is no need for full precision as long as there is no perceivable difference in the results. Therefore, we drop the four least significant bits, reducing the bus load by 60% to 70%.

To handle the entire memory transfer with a single transaction, we pack the encoded data compactly in memory. We use a single atomically modified counter which is increased by the required amount of memory for each block. In this way, every block retrieves its individual offset in memory placing blocks with different compression rates next to each other. The block offset is stored together with information about the used bit length in a header structure. Blocks are decompressed independently of each other. Each block reads its offset and bit length from the header. Then, the representative values and differences are read and the decompressed values can be generated.

4.3. Multi-frame rate configuration

Multi-frame rate (MFR) rendering decouples viewing from image generation and typically uses separate computing nodes or GPUs for each of the tasks. Viewing means linear camera or object transformation, as it is common in many applications. The image generation method can be arbitrary, as long as a depth value can be computed for each pixel. Generated images are transferred to the viewing node, but the viewing node does not wait for images to become available. Instead, it uses the last known image for viewing. In general there is a slight difference in the desired viewing transformation and the one that was used to generate the last

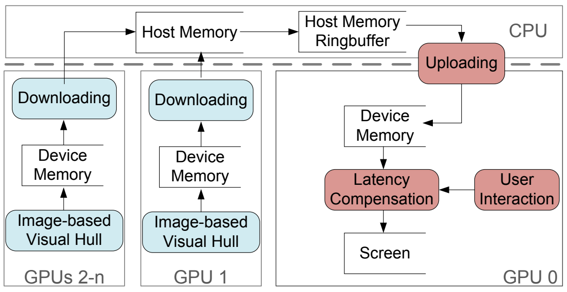


Figure 7: *Multi-frame rate configuration that uses GPU0 for viewing and the other GPUs for parallel image-based visual hull rendering.*

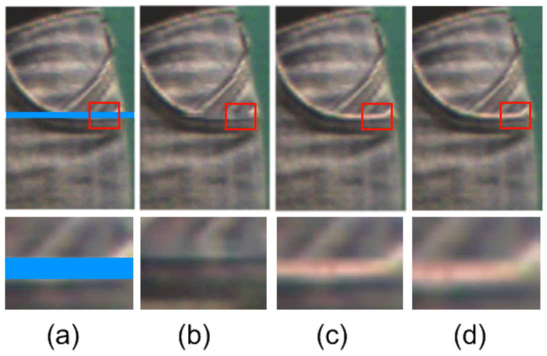


Figure 8: *The effect of two-pass image warping: holes after forward image-warping (a, blue) can be filled by interpolating neighboring warp vectors as in (c) instead of interpolating color values as in (b). (d) shows the ground truth. The red boxes are magnified below.*

image. This difference can be covered by image-warping. Hardware-accelerated image-warping and the asynchronous communication behavior guarantees very high frame rates at the viewing node. The advantage over synchronized parallel rendering grows with increasing scene complexity and output resolution.

IBVH rendering is a complex algorithm and therefore benefits from MFR rendering. However, the visual hull transforms non-rigidly every time a new set of camera images becomes available. Therefore, the high viewing performance of MFR rendering can only be exploited between camera image updates. The frame rate of the image generation node(s) must be higher than the update rate of the cameras. In practice we observed that this is not a limitation: the camera update rates are usually not as high as the desired viewing frame rates due to limited bus and network bandwidths and latencies that slow down camera image transfer. See Section 6 for a performance analysis.

Two-pass image-warping For image-warping we use a combination of forward and backward image-warping. Forward image-warping projects pixels from a source image to the current view. This is a fast operation, but suffers from holes in the result. Backward image-warping on the other hand does not produce holes, but involves a search in the original image to find the correct pixels and is therefore slow [SCK06].

Our two pass warping approach combines the advantages of both approaches. First, a forward warping step is used to project the last frame's pixel coordinates according to the current view. The holes in this buffer which arise from forward warping are closed using linear interpolation on the pixel coordinates (in contrast to color values in traditional approaches). Now, this buffer forms a lookup table into the last frame with sub-pixel accuracy. In a final step, this lookup is performed to compute the new color values, resulting in a backward warping approach. Figure 8 illustrates the differences between color value interpolation and our approach. Our approach does not blur the image. Implementation details are described in Section 5.

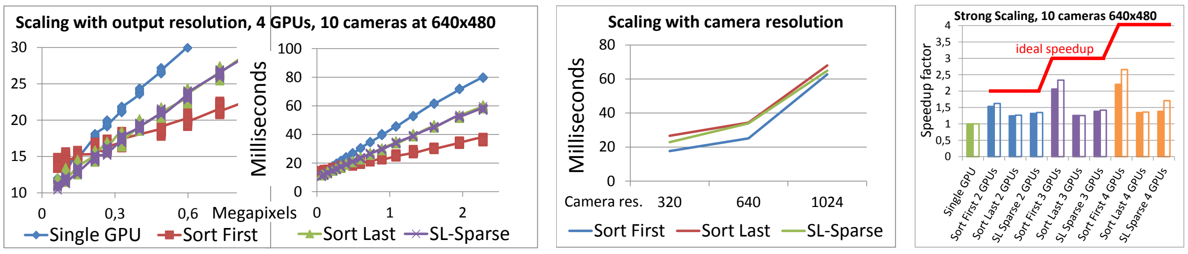
Combined synchronous and asynchronous rendering Note that large holes that come from disocclusions can not be filled in such an efficient way. Data that is not present can not be interpolated. Here we rely on rendering performance: disocclusions are less likely with a quick image source.

To achieve the required performance, we want to use multi-GPU rendering as the image source for multi-frame rate rendering. Our system allows to combine multi-GPU (synchronous) and multi-frame rate (asynchronous) IBVH rendering. For example, the image source can be a sort-first or sort-last IBVH renderer that uses all but one GPUs. The one remaining GPU is used for image-warping and display. See Figure 7 for an illustration of such a configuration and section 6 for a performance evaluation.

5. Implementation

The whole pipeline is implemented in Nvidia CUDA [HSR11] and uses streams to distribute command sequences to the GPUs. All memory copies are issued asynchronously, which makes the system more robust to differing workloads: when a GPU has finished processing it can already send the result and thus help to avoid later bus collisions. Memory is transferred between GPUs through host memory.

All nodes are provided with the camera images (4 bytes per pixel) that they need, state information in the form of a projection matrix (4x4 float) and bounding box information to avoid unnecessary data traffic. Pixels of depth image segments are either compressed to two bytes of precision (half float), or encoded per block for sort-last with compact interval representation.



(a) (b) (c)

Figure 9: *Performance evaluation of our pipeline. (a) shows how the suggested approaches scale with respect to output resolution. The left figure is a magnification of the first Megapixel range. (b) shows the scaling with camera resolution for a fixed output resolution of* 1 *Megapixel. (c) illustrates how additional GPUs influence processing times at an output resolution of* 2.3 *Megapixels (filled bars) and* 7.36 *Megapixels (outlined bars).*

In the case of multi-frame rate rendering, the memory transfers are double-buffered in host memory to facilitate asynchronous communication without stalls [HKS10]. For image-warping the rendered images have the projection matrix attached that was used to create them.

The buffer that is used for two-pass image warping is a screen-sized array of unsigned integers. Screen coordinates are stored in the buffer by packing them with their depth value into a single unsigned integer. The depth values occupy the ten most significant bits, the screen coordinates share the rest. We can achieve efficient depth-buffering when such data packets are written to the buffer by using atomic minimum operations. For hole filling, the neighboring data packets are decoded, screen coordinates are averaged and stored at the hole pixel.

6. Evaluation

In this section, the configurations suggested above are evaluated for their performance. The IBVH kernel is responsible for most of the computation time and is therefore the main focus of our parallelization methods and the corresponding evaluations. As stated in the beginning, the performance of this stage scales with the number of cameras and their resolution, the display resolution and the number of GPUs. For our evaluations we assume a fixed number of cameras of ten. While our system works with any number of cameras, this specific number proved to work best for the application of rendering people with high quality. We do, however, evaluate the system for different camera resolutions, display resolutions and number of GPUs to find out how the suggested methods scale. The used camera images are read from the hard-disk for repeatability. Note that we excluded the hard-disk access times from the evaluation, because in the live system the camera images are also not loaded from the disk. We therefore place the images in the host memory and upload them to the GPUs every frame, just like the live system would.

Scaling with display resolution In the first test series we rendered a representative IBVH scene (see Figure 6) from a viewpoint that is close enough to have the object fill the screen. We averaged the rendering times of several frames at varying display resolutions. Figure 9 (a) shows performance measurements for four GPUs and a single-GPU as a reference. The multi-GPU approaches outperform the single-GPU approach especially for larger output resolutions.

Scaling with camera resolution For this test we rendered three scenes with different camera resolutions. The scenes consist of the same person standing in similar poses. All measurements are again averaged over multiple frames. See Figure 9 (b) for results. While the resolutions quadruple every step, the execution times maximally double. This is promising for higher camera resolutions in the future.

Scaling with number of GPUs: strong scaling In this test series we use again a representative scene (see Figure 6) and render with one, two, three and four GPUs at a resolution of 2.3 and 7.36 Megapixels. Performance measurements are given as speedup factors. Speedup factors compare the performance of *n* GPUs to the performance of one GPU. A factor of *n* is considered ideal and is illustrated as a red line. All measurements are again averaged over multiple frames. Figure 9 (c) illustrates how the performance benefit of adding an additional GPU declines with the total number of GPUs.

Multi-frame rate scaling The evaluation data of the multi-frame rate approach is shown separately in Figure 10. We used the scene from above and measured performance for two and four GPUs. The dual-GPU setup uses a viewing GPU (GPU0) and a single GPU for IBVH rendering (GPU1). The quad-GPU setup also uses GPU0 for viewing, but three GPUs for sort-first parallel IBVH rendering.

The GPUs that are responsible for image generation show a similar performance to a stand alone setup. The performance goal for them is to stay below the camera update rate in order

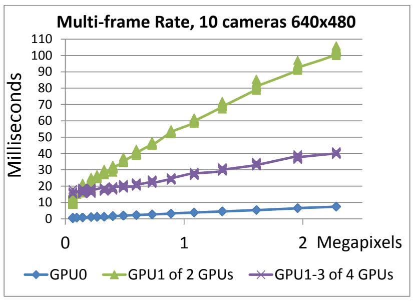


Figure 10: *Multi-frame rate performance for a setup with two and four GPUs. GPU0 is always used for viewing. Its performance is comparable for both configurations. The quad-GPU setup uses three GPUs for sort-first parallel IBVH rendering.*

to avoid missing any image. The camera update rate in our system usually is 15 Hz. The quad-GPU setup stays well below this mark.

The performance of the viewing GPU outperforms all other approaches easily, because viewing is a less complex task than IBVH rendering by far. In combination with an image-generation backend that does not miss any camera image, this is a very powerful method for interactive IBVH rendering.

Interpretation and discussion In terms of performance we observed a behavior that is common to sort-first and sort-last parallel rendering. Sort-first nodes need to receive all the input data, whereas sort-last nodes need to send all the output data. Therefore, when data transfer to the nodes - in our case the camera images - is dominant, then sort-last with compression is a good option. Figure 9 (a) left shows such a setup. When traffic is output-heavy due to a relatively high output resolution then sort-first performs best. Figure 9 (a) right illustrates this.

The performance measurements in Figure 9 (c) reveal that the scaling of the pipeline with additional GPUs is far from optimal. To find the reason for this, we analyzed all stages of the pipeline. The IBVH kernel itself scales almost ideally with an increasing number of GPUs. However, the memory transfer times between the GPUs increase stronger. At 2 Megapixels output resolution and four GPUs, the sort-last approach uses 28% of the time for transferring output fragments, and we even subtracted the transfers that overlap with execution from this number. Only about 58% of the time was spent with actual processing. The sort-first approach has better scaling characteristics. For the same configuration, only 4.5% were fragment transfers and around 68% was processing. To increase the processing time relative to the overall execution time, we also evaluated the pipeline at 7.36 Megapixels output resolution. At this very high resolution the sort-first approach spent around 63% of the time with processing and 5.5% with fragment transfer. From this data we derive that it is mostly the memory traffic and the overhead computations (kernel launches, memory initializations) that prevent the presented pipeline from scaling well beyond two or three GPUs.

The multi-frame rate setup can utilize a sort-first or sort-last IBVH renderer as its image source. We found this configuration to be particularly useful, because it provides over 100 frames per second on the viewing GPU even for output resolutions of 2 Megapixels. At the same time, a dual or triple-GPU image source can provide enough computation power to process every camera image. At these frame rates, the visual quality of image warping is high because disocclusion artifacts can hardly be observed.

7. Conclusions and future work

In this paper we have introduced methods to parallelize the image-based visual hull algorithm on multiple GPUs. First, we analyzed the pipeline for possible parallel execution. We identified two methods, following the common sorting-classification: sort-first and a sort-last. For sort-last we suggested to regard the cameras as scene objects, and introduced how the compositing step needs to be adapted. In addition, we suggested a block-based packing scheme that reduces memory traffic drastically. Finally, we enhanced the system by multi-frame rate rendering to achieve even higher frame rates for viewing applications. We introduced two-pass warping to achieve better hole-filling. We evaluated the performance of all approaches and were ably to verify that a triple or quad-GPU multi-frame rate setup can achieve very high interactivity without sacrificing the visual quality.

In the future we want to investigate how frame-to-frame coherence methods can further enhance the performance of IBVH rendering.

Acknowledgments This work was supported by the Austrian Research Promotion Agency (FFG) under the BRIDGE program, project #822702 (NARKISSOS).

References

[AFM*06] Allard J., Franco J., Menier C., Boyer E., Raffin B.: The grimage platform: A mixed reality environment for interactions. In *Computer Vision Systems, 2006 ICVS '06. IEEE International Conference on* (jan. 2006), p. 46. 2

[Boy03] Boyer E.: A hybrid approach for computing visual hulls of complex objects. In *In Computer Vision and Pattern Recognition* (2003), pp. 695–701. 2

[dAST*08] de Aguiar E., Stoll C., Theobalt C., Ahmed N., Seidel H.-P., Thrun S.: Performance capture from sparse multi-view video. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers* (New York, NY, USA, 2008), ACM, pp. 1–10. 2

[FLZ10] Feng J., Liu Y., Zhou B.: Real-time stereo visual hull rendering using a multi-gpu-accelerated pipeline. In *ACM*

SIGGRAPH ASIA 2010 Sketches (New York, NY, USA, 2010), SA '10, ACM, pp. 52:1–52:2. 2

[FMBR04] Franco J.-S., Menier C., Boyer E., Raffin B.: A distributed approach for real time 3d modeling. In *Proceedings of the 2004 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'04) Volume 3 - Volume 03* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 31–. 2

[FWZ03] Fitzgibbon A., Wexler Y., Zisserman A.: Image-based rendering using image-based priors. In *ICCV '03: Proceedings of the Ninth IEEE International Conference on Computer Vision* (Washington, DC, USA, 2003), IEEE Computer Society, p. 1176. 2

[GG07] Geys I., Gool L. V.: View synthesis by the parallel use of gpu and cpu. *Image Vision Comput. 25*, 7 (2007), 1154–1164. 2

[GHKM11] Graf H., Hazke L., Kahn S., Malerczyk C.: Accelerated real-time reconstruction of 3d deformable objects from multi-view video channels. In *Digital Human Modeling*, Duffy V., (Ed.), vol. 6777 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2011, pp. 282–291. 2

[GM03] Goldluecke B., Magnor M.: Real-time, free-viewpoint video rendering from volumetric geometry. In *Visual Communications and Image Processing 2003* (Lugano, Switzerland, June 2003), Ebrahimi T., Sikora T., (Eds.), vol. 5150 of *SPIE proceedings*, The International Society for Optical Engineering (SPIE), SPIE, pp. 1152–1158. 2

[HAM06] Hasselgren J., Akenine-Möller T.: Efficient depth buffer compression. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (New York, NY, USA, 2006), GH '06, ACM, pp. 103–110. 6

[HKS10] Hauswiesner S., Kalkofen D., Schmalstieg D.: Multi-frame rate volume rendering. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)* (2010). 8

[HSR11] Hauswiesner S., Straka M., Reitmayr G.: Coherent image-based rendering of real-world objects. In *Symposium on Interactive 3D Graphics and Games* (New York, USA, 2011), ACM, pp. 183–190. 1, 3, 7

[LBN08] Ladikos A., Benhimane S., Navab N.: Efficient visual hull computation for real-time 3d reconstruction using cuda. *Computer Vision and Pattern Recognition Workshop 0* (2008), 1–8. 2

[LCO06] Lee C., Cho J., Oh K.: Hardware-accelerated jaggy-free visual hulls with silhouette maps. In *VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 2006), ACM, pp. 87–90. 2

[Li04] Li M.: *Towards Real-Time Novel View Synthesis Using Visual Hulls*. PhD thesis, Universität des Saarlandes, 2004. 2

[MBR*00] Matusik W., Buehler C., Raskar R., Gortler S. J., McMillan L.: Image-based visual hulls. In *SIGGRAPH '00 proceedings* (New York, NY, USA, 2000), ACM Press/Addison-Wesley Publishing Co., pp. 369–374. 2

[MCEF08] Molnar S., Cox M., Ellsworth D., Fuchs H.: A sorting classification of parallel rendering. In *ACM SIGGRAPH ASIA 2008 courses* (New York, NY, USA, 2008), SIGGRAPH Asia '08, ACM, pp. 35:1–35:11. 5, 6

[MMB97] Mark W. R., Mcmillan L., Bishop G.: Post-rendering 3d warping. In *In 1997 Symposium on Interactive 3D Graphics* (1997). 3

[NNT07] Nitschke C., Nakazawa A., Takemura H.: Real-time space carving using graphics hardware. *IEICE - Trans. Inf. Syst. E90-D*, 8 (2007), 1175–1184. 2

[SBW*07] Springer J. P., Beck S., Weiszig F., Reiners D., Froehlich B.: Multi-frame rate rendering and display. In *VR* (2007), Sherman W. R., Lin M., Steed A., (Eds.), IEEE Computer Society, pp. 195–202. 2

[SCK06] Shum H.-Y., Chan S.-C., Kang S. B.: *Image-Based Rendering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 7

[SHRB11] Straka M., Hauswiesner S., Ruether M., Bischof H.: A free-viewpoint virtual mirror with marker-less user interaction. In *Proc. of the 17th Scandinavian Conference on Image Analysis (SCIA)* (2011). 3

[SSS*02] Slabaugh G. G., Slabaugh G. G., Schafer R. W., Schafer R. W., Hans M. C., Hans M. C.: Image-based photo hulls. In *In The Proceedings of the 1 st International Symposium on 3D Processing, Visualization, and Transmission* (2002), pp. 704–708. 2

[SvLBF09] Smit F. A., van Liere R., Beck S., Fröhlich B.: An image-warping architecture for vr: Low latency versus image quality. In *VR* (2009), IEEE, pp. 27–34. 3

[TLMpS03] Theobalt C., Li M., Magnor M. A., peter Seidel H.: A flexible and versatile studio for synchronized multi-view video recording. In *Vision, Video and Graphics, p.9-16, Bath, UK* (2003). 2

[WFEK09] Waizenegger W., Feldmann I., Eisert P., Kauff P.: Parallel high resolution real-time visual hull on gpu. In *ICIP'09: Proceedings of the 16th IEEE international conference on Image processing* (Piscataway, NJ, USA, 2009), IEEE Press, pp. 4245–4248. 2, 3

[WTM06] Wu X., Takizawa O., Matsuyama T.: Parallel pipeline volume intersection for real-time 3d shape reconstruction on a pc cluster. In *Proceedings of the Fourth IEEE International Conference on Computer Vision Systems* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 4–. 2

[YLKC07] Yous S., Laga H., Kidode M., Chihara K.: Gpu-based shape from silhouettes. In *proceedings of GRAPHITE '07* (New York, NY, USA, 2007), ACM, pp. 71–77. 2

[YZC03] Yue Z., Zhao L., Chellappa R.: View synthesis of articulating humans using visual hull. In *ICME '03: Proceedings of the 2003 International Conference on Multimedia and Expo* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 489–492. 2