Eurographics Symposium on Parallel Graphics and Visualization (2019)
H. Childs, S. Frey (Editors)

Screen Partitioning Load Balancing for Parallel Rendering on a Multi-GPU Multi-Display Workstation

Yangzi Dong [1†] and Chao Peng[1‡]

[1] Computer Science Department, The University of Alabama in Huntsville, USA

Abstract

Commodity workstations with multiple GPUs have been built by engineers and scientists for real-time rendering applications. As a result, a high display resolution can be achieved by connecting each GPU to a display monitor (resulting in a tiled large display). Using a multi-GPU workstation may not always produce a highly interactive rendering rate due to imbalanced rendering workloads among GPUs. In this work, we propose a parallel load balancing algorithm based on a screen partitioning strategy to dynamically balance the amount of vertices and triangles rendered by each GPU. Each GPU renders a screen region whose size may be different from the screen regions of other GPUs, but the amounts of vertices and triangles in those screen regions are balanced. It is possible that a screen region rendered by a GPU has to be displayed by another GPU. We propose a frame exchanging algorithm that allows GPUs to exchange screen regions efficiently. The inter-GPU communication overhead is very small since the data transferred between GPUs are a small amount of image pixels.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems—Distributed/Network Graphics; I.3.m [Computer Graphics]: Miscellaneous—Parallel Rendering

1. Introduction

Large 3D models are typical output in many research areas such as scientific simulation and computer-aided design. To increase storage efficiency, a model is usually divided into many objects where each object is a self-contained mesh composed of vertices and triangles and with intertwined details and complicated topologies. In a visualization application, demands are not only high rendering performance but also a high rendering resolution on a large screen [CSR*03], so that intensive graphical contents can be presented with a sufficient number of pixels. Such demands motivate us to build a multi-GPU multi-display commodity workstation for parallel rendering, in which each GPU connects to one or multiple display monitors, and together they form a large tiled display at low costs.

As we know, hardware approaches for multi-GPU rendering are commercially available (e.g., Nvidia SLI [NVI11] and AMD Crossfire [AMD17]). A common load balancing strategy provided by hardware approaches is to make successive frames (in their entirety or fractions) rendered by different GPUs. However, the hardware approaches cannot configure a workstation with a per-GPU per-display setting. When the SLI or Crossfire is enabled, one GPU has to be treated as the master device; and other GPUs become workers. Only the master GPU is capable of driving display monitors. Software approaches for multi-GPU rendering are also available. For example, the Equalizer [EMP09, ESP18] is a state-of-the-art parallel rendering framework. It supports a per-GPU per-display setting; however, it requires input meshes to be represented in a hierarchical structure such as a k-d tree, and has to perform recursive traversal of the tree at the runtime in order to balance workloads among GPUs. Major drawbacks of Equalizer are that the hierarchical structure for a complex model usually consumes a large amount of memory, and the traversal is usually hard to be parallelized on the GPU.

Contributions. Advancing the existing hardware and software approaches, our parallel rendering approach supports a per-GPU per-display setting and does not require a hierarchal structure for data representation. In our approach, a novel load balancing algorithm is developed. It performs a fine-grained parallelization on the GPU at an object level and balances the vertices and triangles assigned to GPUs upon the dynamic change of the camera's viewpoint. Also, a novel frame exchanging algorithm is developed to identify and transfer portions of a rendered frame between GPUs, rather than transferring the full screen or geometry data, which reduces the inter-GPU communication overhead.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 provides a system overview. Section 4 explains the algorithm to configure GPUs settings. Section 5 describes our parallel load balancing algorithm. Section 6 explains

† yangzi.dong@uah.edu
‡ chao.peng@uah.edu

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

DOI: 10.2312/pgv.20191111

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

our frame exchanging algorithm. Section 7 shows the evaluation results. Section 8 concludes our work and proposes future work.

2. Related Work

Recently, many computing systems have been built with multi-GPU workstations or GPU clusters for high-performance applications. Related to our work, we review parallel rendering approaches with a single workstation and a distributed system. One pioneer work in parallel rendering was proposed by Whitman [Whi94]. In that work, a parallel algorithm took the advantages of locally cached memory and increased execution efficiency. Eilemann [Eil07] wrote a white paper summarizing middleware solutions for parallel rendering, including OpenGL Multipipe SDK [BRE05], Chromium [HHN*02], CAVELib [Pap97], and Equalizer [EMP09]. Those middleware solutions require data replications and lack functionality to incorporate with mesh simplification techniques.

Nvidia SLI [NVI11] and AMD Crossfire [AMD17] are hardware solutions for parallel rendering with multiple GPUs in a single workstation. They configure the GPUs as one hardware entity. The hardware solutions support two rendering modes: (1) assigning split regions of the screen to different GPUs (SFR/Scissors) and (2) switching GPUs after each frame (AFR). Both modes require data replications and use a master-slave model which requires that the display monitors connect to the master GPU.

Allard and Raffin [AR05] utilized a graphics cluster to perform distributed rendering tasks using hardware shaders on networked PCs. However, their approach does not consider load balancing issues and requires the entire dataset to be retained on the GPUs. Liu et al. [LWW*11] separated the rendering stage and compositing stage. Their approach adopts a data-partitioning strategy that assigns an arbitrary data portion to each GPU, and each GPU has to render the full frame. Wang et al. [WLL*11] presented a "compositeless" algorithm which removes the compositing stage from the pipeline. In their approach, the screen is divided into tiles, and each GPU is assigned with the data in the corresponding tile. Their approach does not balance the workload distribution among GPUs. Eilemann et al. [EBA*12] analyzed the asynchronous parallel rendering system on hybrid Multi-GPU clusters and evaluated the optimizations for improving the scalability of the system.

Parallel rendering approaches can be classified into three categories [MCEF94]: sort-first, sort-middle, and sort-last. A sort-first approach distributes primitives before the stage of rendering. A sort-middle approach distributes the primitives during the stage of rendering. A sort-last approach distributes the rendered-frame after the stage of rendering. Samanta et al. [SZF*99] introduced a sort-first parallel rendering system running on a PC cluster. Each processor of the system rendered a balanced workload corresponding to a virtual tile on a projector. Moreland et al. [MWP01] presented a sort-last method for the parallel rendering of large data sets on a tiled display. Their method evenly distributed polygons among all processors in a PC cluster and composed the rendered images for each tile. They presented parallel composition strategies to optimize sort-last rendering performance. Abraham et al. [ACCC04] proposed a sort-first method along with a time-guided load balancing strategy. The screen partitioning position of the current

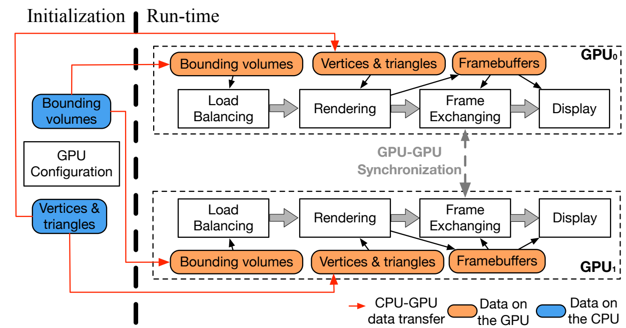

Figure 1: *The execution sequence of components in our approach illustrated with two GPUs.*

frame is adjusted according to the rendering time spent on the previous frame. Moloney et al. [MWMS07] described a scalable sort-first algorithm for dynamic load balancing. The data set was evenly divided into uniform bricks and distributed between nodes based on the pre-calculated rendering cost on each pixel. Erol et al. [EEP11] presented the cross segment method for load balancing. Their method evaluated the computational time of each GPU spent on the rendering of previous frames and assigned more rendering tasks to the GPUs that had less computational time so that all GPUs could be balanced in terms of computational time. Steiner et al. [SPEP16] distributed rendering tasks to client nodes. Their method is adapted to either sort-first and sort-last rendering. In this work, our approach is a hybrid of sort-first and sort-last. The load balancing algorithm is executed in the sort-first phase, and the screen partitioning algorithm is executed in the sort-last phase.

3. System Overview

Figure 1 illustrates the general execution sequence of components in our approach. At the initialization, the *GPU Configuration* component organizes GPUs of the system into the form of a binary search tree (see Section 4). Each GPU is controlled by a CPU core and connects to a display monitor. Each CPU core is controlled by a process. The GPUs conduct to both general-purpose computation and standard triangle rasterization tasks. We also compute a bounding volume for each object of the model. Bounding volumes are used by the *Load Balancing* component at the runtime. In order to render the model, bounding volumes, vertices, and triangles are sent to the GPUs at the initialization, and then they are kept in the GPU memory.

During the runtime, the *Load Balancing* component identifies appropriate screen partitioning positions and updates the number of triangles that would be rendered on the GPU it serves for (see Section 5). The *Rendering* component renders the triangles assigned to the GPU into pixels and stores them in the GPU's *Framebuffers*. In the *Frame Exchanging* component, for the GPU that renders the screen region larger than the region it displays, it sends the extra screen regions to other GPUs (see Section 6). The *Display* component presents the composed rendering results to the display monitor that the GPU connects to.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

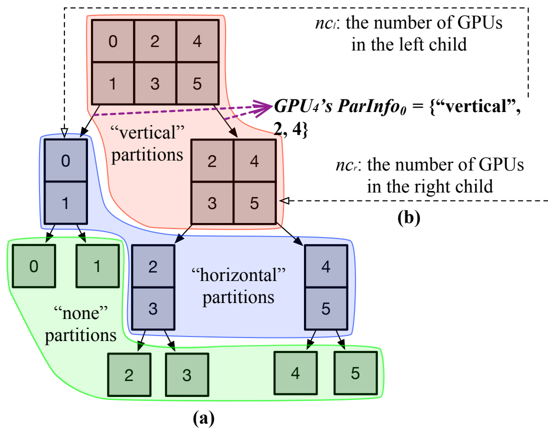

Figure 2: *A GPU tree example of six GPUs in the format of* 2×3. *(a) shows the building of the tree. The nodes in the red zone are in accordance to the results of vertical screen partitioning. The nodes in the blue zone are in accordance to the results of horizontal screen partitioning. Each leaf node in the green zone contains a single GPU. (b) shows the first* $ParInfo$ *of* GPU_4*, where the values are recorded from the split of the root node.*

The GPUs may finish the *Frame Exchanging* component asynchronously due to differences on rendered frame sizes and inter-GPU communication time. In our approach, the *GPU-GPU Synchronization* is done through shared memory. It forces faster GPUs to wait until all GPUs finish their rendering tasks, and ensures the rendered frame is composed properly on the GPU before it can be displayed.

4. Multi-GPU Configuration

We configure the physical installation of GPUs and their display monitors in a matrix format, denoted as $m \times n$, where m is the row dimension, and n is the column dimension. For example, given a total of six GPUs, the column-major order gives possible configurations of 2×3, 3×2, 1×6, or 6×1. Furthermore, we label those GPUs in column-major order.

We use a binary search tree to group GPUs. Each internal node of the tree contains a subset of consecutive labels of GPUs. The internal nodes are further split to the next level until reaching the leaf level, where each leaf node contains a single GPU. At each non-leaf level, the screen region associating to a tree node (a subset of GPUs) can be partitioned either horizontally or vertically. Let's define the total number of GPUs as K. We initialize the tree's root node to contain all GPUs, denoted as the set $\{GPU\}_0^{K-1}$, where the subscript is the label of the first GPU in the set, and the superscript is the label of the last GPU in the set. The left and right child nodes of the root are denoted as $\{GPU\}_0^{\frac{K}{n}-1}$ and $\{GPU\}_{\frac{K}{n}}^{K-1}$, respectively. We recursively build the tree in favor of splitting a tree node by partitioning the screen vertically. Figure 2-(a) gives an example of the GPU tree which is built from six GPUs in the form of 2×3. If the column dimension of the GPUs in a tree node is equal to 1, the node will be split horizontally, as shown in the blue region of Figure 2-(a).

Every leaf node contains a single GPU, and its *depth* in the tree indicates the number of screen partitioning operations the GPU will perform. This is essential information in order to obtain the screen region for each GPU to render (see details in Section 5). For each GPU, we traverse the tree in a depth-first manner and find the list of screen partitioning operations involving this GPU; and then, we store them in an array structure, denoted as $ParInfo$. The size of $ParInfo$ is the $depth$ of the GPU in the tree. At the ith iteration of tree building, $ParInfo_i = \{paritionMethod, nc_l, nc_r\}$, where nc_l is the number of GPUs in the left child of the current node containing this GPU, and nc_r is the number of GPUs in the right child of the current node containing this GPU, as shown in see Figure 2-(b).

5. Load Balancing

We describe a novel parallel algorithm which computes balanced rendering workloads among GPUs. The algorithm takes the view frustum and the triangle counts in objects as input. Each GPU executes an instance of the algorithm. The view frustum corresponds to the full screen projected on the entire tiled display, so all GPUs have the same input view frustum. The triangle counts in objects are presented in an array structure, denoted as T, where T_i is the number of triangles of the ith object. The load balancing algorithm crops the view frustum into a sub-frustum for the GPU, and correspondingly modifies the values of the T array for the GPU. As a result, only the objects inside the sub-frustum of the GPU will remain their triangle counts in the T array. In other words, if the ith object is outside the sub-frustum of the GPU, the algorithm will change T_i to zero.

The sub-frustums of all GPUs may be in different sizes, but their triangle counts are balanced. The balancing process is performed in screen space on the near plane of the view frustum. We first introduce the idea of screen and frustum strips in Section 5.1. We then describe the load balancing algorithm and its parallelization in Section 5.2.

5.1. Screen and Frustum Strips

Let's start with the method to partition a screen. The essential operation of screen partitioning is to specify a partitioning line, which cuts through the screen either horizontally or vertically. The choices to specify a partitioning line are limited to the screen resolution. For example, given a dual-GPU workstation with the 1×2 configuration for their monitors, the full-screen will be partitioned vertically. If the screen resolution of each GPU is 1024×768 pixels, the full screen resolution is 2048×768 pixels. Thus, there are a total of 2048 possible partitioning lines, with each at one pixel on the width dimension. On the near plane of the view frustum, those partitioning lines produce *screen strips*, which are small screen regions of the same size. Then, we use the screen strips to subdivide the view frustum into *frustum strips*, which are small truncated pyramid volumes created from the screen strips and the viewpoint position of the camera. Like the example illustrated in Figure 3, the screen is evenly partitioned into four screen strips, and correspondingly four frustum strips are created.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

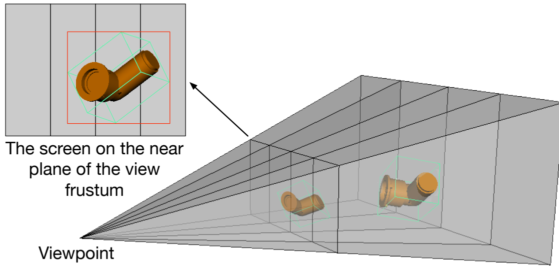

Figure 3: *Four frustum strips generated by evenly partitioning the screen into four regions. The red rectangle on the near plane is the bounding rectangle of the projected object's bounding volume.*

We define a parameter called $stripNum$, ranging in $(2, Q)$, to control the number of frustum strips on each screen axis, where Q is the total number of pixels along this axis. A higher value of $stripNum$ gives more and narrower strips, and the load balancing algorithm can, therefore, use them to produce a finer balancing result.

5.2. Load Balancing Algorithm and Parallelization

The goal of load balancing algorithm is to find an appropriate partitioning line so that for the objects projected on the partitioned regions of the screen, their triangle counts are balanced. As mentioned in Section 4, GPUs are hierarchically grouped into a binary search tree. This requires a repetitive execution of a partitioning operation in order to progressively update the modified triangle counts T (denoted as T') and sub-frustums in descendent tree nodes, as shown in Algorithm 1. The number of partitioning operations that a GPU should perform is equal to the depth of this GPU (the corresponding leaf node) in the tree. In order to ensure all GPUs receive the balanced workload, the intermediate partitioning operation for a non-leaf node is weighted based on the number of GPUs that its two child nodes contain. Here, we introduce a term for such weighting called *balRatio*, so we have *balRatio* = $\frac{ParInfo_i.nc_l}{ParInfo_i.nc_r}$, where i is the ith partitioning operation that associates to the non-leaf node. After each iteration of screen partitioning, the value of $balRatio$ needs to be recalculated (see line 4 in Algorithm 1). The partitioning operation takes the computed *balRatio*, the sub-frustum, bounding volumes of objects, and triangle counts as input, and then finds the screen partitioning position (*p-pos* of this non-leaf node (see line 5 in Algorithm 1). As a result after all iterations of screen partitioning, we obtain an array of *p-pos* values, where each *p-pos* value corresponds to the result of a screen partitioning operation.

Here, we want to explain the single step algorithm executed at each iteration of screen partitioning. Figure 4 illustrates the execution of the algorithm using an example composed of 16 frustum strips and 10 objects. Note that the triangle count of each object is already known and retrieved from T_i. The GPU allocates two arrays in the memory whose sizes are equal to the number of frustum strips. The first one is called *Start* array, denoted as S; and the second one is *End* array, denoted as E. S_i is the sum of triangle counts of the objects intersecting with the ith strip, and this ith strip must be the first strip the objects intersect with. E_i is also the sum of triangle counts of the objects intersecting with the ith strip, but this ith strip must be the last strip the objects intersect with. In Figure 4, we have $S_3 = T_1 + T_9$ because only Obj_1 and Obj_9 use the third strip as the first intersected strip. If an object crosses the left boundary of the screen, we assume the first strip is the left-most strip (e.g., Obj_3 in Figure 4). Similarly, if an object crosses the right boundary of the screen, we assume the last strip is the right-most strip (e.g., Obj_8 in Figure 4).

Algorithm 1 Load Balancing executed according to the GPU's depth in the GPU Tree

Balancing(
Input: $ViewFrustum$, $BoundingVolumes$, T, $stripNum$, $depth$;
Output: T', an array of *p-pos*)

1: $T' \leftarrow T$;
2: *sub-frustum* $\leftarrow ViewFrustum$;
3: **for** the ith level of the $depth$ **do**
4: $\quad balRatio \leftarrow \frac{ParInfo_i.nc_l}{ParInfo_i.nc_r}$; ▷ also see Section 4
5: $\quad$ {T', *p-pos*$_i$, *sub-frustum*} $\leftarrow$ **SingleStepPartitioning**(*sub-frustum*, *BoundingVolumes*, T', *balRatio*, *stripNum*);
6: **end for**

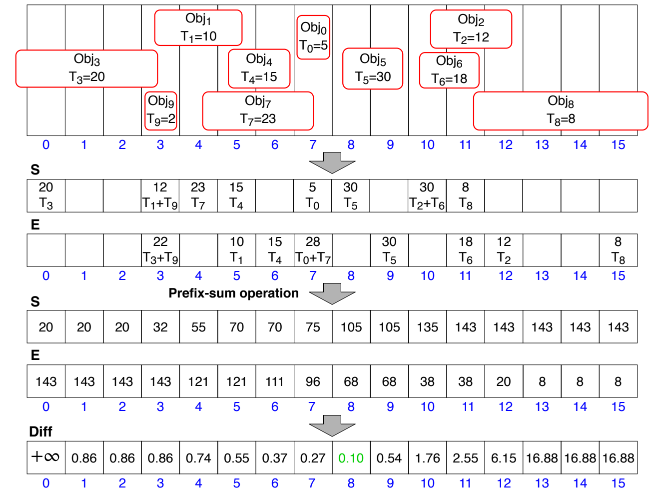

Figure 4: *An example of the load balancing algorithm for a non-leaf node in the GPU tree. A total of 10 objects are projected on the screen. The* $stripNum$ *is set to 16. Assume that the number of GPUs in two child nodes of this non-leaf node is same , so that the value of balRatio is equal to 1.0.*

Algorithm 2 shows the single step of screen partitioning with object-level parallelization. The algorithm returns the modified triangle counts T', the screen partitioning position (*p-pos*), and the sub-frustum that will be used by the next level of partitioning. The value of *p-pos* is a normalized value ranging in $[0, 1]$; so if the value of *p-pos* is equal to 0.5, it indicates the view frustum is divided at the middle. The algorithm first needs to find out the strips that an object interests with. As shown in Figure 3, the intersection test is performed in screen space using the bounding rectangle of the ob-

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

Algorithm 2 A Single Step of Screen Partitioning with Object-Level Parallelization

SingleStepPartitioning(
Input: $ViewFrustum$, $BoundingVolumes$, T, $balRatio$, $stripNum$;
Output: T', *p-pos*, *sub-frustum*)

1: Initialize S, E, Diff, *p-pos*;
2: $BoundingRectangles \leftarrow$ compute the bounding rectangles **in parallel**;
3: **for** ith element in $BoundingRectangles$ **in parallel do**
4: Find the range of intersected screen strips $\rightarrow [min, max]$;
5: S_{min} += T_i;
6: E_{max} += T_i;
7: **end for**
8: $S \leftarrow$ Prefix sum of S;
9: $E \leftarrow$ Postfix sum of E;
10: **for** ith element in $Diff$ **in parallel do**
11: **if** S_i or $E_i == 0$ **then**
12: $Diff_i \leftarrow +\infty$;
13: **else**
14: **if** $i == 0$ **then**
15: $Diff_i \leftarrow +\infty$;
16: **else**
17: $Diff_i \leftarrow ||\frac{S_{i-1}}{E_i} - balRatio||$;
18: **end if**
19: **end if**
20: **end for**
21: $index \leftarrow$ find the $Diff$ element having the minimum value **in parallel**;
22: $p\text{-}pos \leftarrow index/stripNum$;
23: *sub-frustum* $\leftarrow$ compute the sub-frustum using $ViewFrustum$ and *p-pos*;
24: **for** ith element in $BoundingVolumes$ **in parallel do**
25: **if** ith $BoundingVolumes$ is outside *sub-frustum* **then**
26: $T'_i \leftarrow 0$;
27: **else**
28: $T'_i \leftarrow T_i$;
29: **end if**
30: **end for**

ject's projected bounding volume. According to the theory of perspective projection in computer graphics, if the bounding rectangle of an object intersects with a screen strip, the bounding volume of this object will intersect with the corresponding frustum strip. We parallelize the execution of Algorithm 2 by assigning one object to one GPU thread. Each thread finds the x-value range or y-value range of the bounding rectangle, uses them to identify a continuous sequence of overlapped screen strips, and then updates corresponding elements in the array S and E (lines 3-7 in Algorithm 2). It is possible that multiple threads access the same element in S or E. For example, multiple objects may start intersecting with the same frustum strip. To avoid such a race condition, when the thread is updating an array element, we use atomic functions from CUDA to prevent the interference from other threads.

After array S and E are generated, Algorithm 2 applies the prefix sum to S and the postfix sum to E (lines 8-9). The implementation is done with CUDA Thrust library [BH11]. As a result, each element in S contains the total number of triangles to its left (including the element itself); and each element in E contains the total number of triangles to its right, as shown in the third step in Figure 4. We compute an array of ratio difference, denoted as $Diff$, which is equal to $||\frac{S_{i-1}}{E_i} - balRatio||$. We then find the minimum value in the array $Diff$. The strip index whose corresponding element in $Diff$ has the minimum value is used to compute the value of *p-pos*. (see lines 10-22 of Algorithm 2). The sub-frustum is generated based on the value of *p-pos*. At the end, Algorithm 2 modifies T'. If the object is outside the sub-frustum, the object's corresponding element in T' is set to zero (see lines 24-30 of Algorithm 2).

Algorithm 3 Determine and send screen portions that need to be displayed by other GPUs

ExchangingScreenRegion(
Input: $depth$, an array of RS, an array of *p-pos*;
Output: T', x_{min}, y_{min}, x_{max}, y_{max})

1: $x_{min} \leftarrow 0$;
2: $y_{min} \leftarrow 0$;
3: $x_{max} \leftarrow width$;
4: $y_{max} \leftarrow height$;
5: **for** the ith level of the $depth$ **do**
6: **if** $RS_i ==$ "$left$" **then**
7: $x_{max} \leftarrow (x_{min} + (x_{max} - x_{min}) \times p\text{-}pos_i)$;
8: **else if** $RS_i ==$ "$right$" **then**
9: $x_{min} \leftarrow (x_{min} + (x_{max} - x_{min}) \times p\text{-}pos_i)$;
10: **else if** $RS_i ==$ "$bottom$" **then**
11: $y_{max} \leftarrow (y_{min} + (y_{max} - y_{min}) \times p\text{-}pos_i)$;
12: **else if** $RS_i ==$ "top" **then**
13: $y_{min} \leftarrow (y_{min} + (y_{max} - y_{min}) \times p\text{-}pos_i)$;
14: **end if**
15: **end for**
16: Check intersection between the screen region of $[(x_{min}, y_{min}), (x_{max}, y_{max})]$ and other GPUs' display regions;
17: **if** there are intersections **then**
18: Send intersected portions of the screen region to other GPUs, on which they should be displayed;
19: **end if**
20: Receive the portions of the screen regions which are rendered by other GPUs but should be displayed by this GPU;

6. Frame Exchanging between GPUs

We denote the screen region rendered by a GPU as the pixel index range of $[(x_{min}, y_{min}), (x_{max}, y_{max})]$. The recorded $partitionMethod$ ("vertical" or "horizontal") in each element of $ParInfo$ indicates the configuration relationship between the GPUs in the current node and the GPUs in the sibling node. Such relationship is one of enumeration types of {"left", "right", "top", "bottom"}. We denote this relationship as RS. Algorithm 3 shows the process to find the screen region that needs to be rendered by a GPU. Initially, the size of the screen region of the GPU is set to the full-screen size. At each screen partitioning operation, a boundary of the screen region is modified by using the corresponding element in the array of *p-pos*. (see lines 5-15 of Algorithm 3).

At the end of the algorithm, the GPU sends the potions of the

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

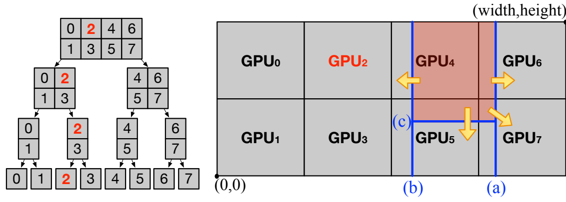

Figure 5: *An example of identifying the GPU_2's screen region that it should render. There are a total of eight GPUs. GPUs and their display monitors are configured with the column-major order. They are grouped into a binary search tree that has three tree node levels ($\log_2 8 = 3$), as the GPU tree shown in the left image. In the right image, the red screen region will be rendered by GPU_2. (a) is the vertical partitioning line with the ratio to partition the full screen. The first-level node that GPU_2 belongs to obtains the screen region of $[(0,0),(x_{(a)},height)]$. (b) is the vertical partitioning line with the ratio to partition the screen region generated from (a). The second-level node that GPU_2 belongs to obtains the screen region of $[(x_{(b)},0),\ (x_{(a)},height)]$. (c) is the horizontal partitioning line with the ratio to partition the screen region bounded between (a) and (b). GPU_2, which is now at the leaf level, obtains the screen region of $[(x_{(b)},y_{(c)}),(x_{(a)},height)]$.*

screen region that should be displayed on other GPUs. In the meantime, the GPU receives the screen regions which are rendered by other GPUs but should be displayed on its display monitor (see lines 16-20). Figure 5 illustrates an example showing the frame exchanging among 8 GPUs. The arrows indicate GPU_2 transfers portions of the rendered screen region to GPU_4, GPU_5, GPU_6 and GPU_7.

We use framebuffers to exchange and composite screen regions among GPUs. The data transferred among GPUs are RGB pixels. Each GPU is assigned with N framebuffers, including one full screen-size framebuffer to hold the rendered frame and $(N-1)$ monitor-size framebuffers to receive the screen portions from other GPUs. In the extreme case that one GPU renders the full screen, the GPU has to send a monitor-size framebuffer to all other GPUs.

7. Evaluations

We built a workstation with an Intel i7-5930K 3.50GHz CPU (12 cores), 64 GBytes of RAM, PCI Express ×16 Gen3, and four Nvidia GeForce GTX980 Ti 6 GB GPUs. The experimental software was implemented on the 64-bit Linux Mint MATE 18.1 system using C++, CUDA 9.2, and OpenGL with the Nvidia driver version 396.26. We used Open MPI version 2.1.0 to create processes. Figure 6 shows the workstation we built for the experiment. We ran the experimental system that has one, two (1×2), three (1×3), and four (2×2) GPUs with the column-major display settings. Different from the hardware support techniques such as Nvidia SLI and AMD Crossfire, our system is not configured as a master-worker model. Thus, the display resolution can scale up as more GPUs are added to the system.

We used a large scene composed of up to 20 copies of the UNC Power Plant model. As a result, the scene is composed of 120.12 million vertices, 254.09 million triangles, and 3.02 million objects, and consumes 5.38 GB memory for storage (including vertices, triangles and bounding volumes).

7.1. Performance

We created a walkthrough camera path that produces 1200 frames for the scene. We first set the resolution of each display monitor to be 1024×1024. Table 1 shows the performance breakdowns. The columns of "# of vertices" and "# of triangles" are the numbers of vertices and triangles of the entire scene, respectively. The column of "Frame Time" is the averaged computational time at a frame. The number of executions of the load balancing algorithm is equal to the logarithm of the number of GPUs. Thus, more GPUs cause more execution time of *Load Balancing* component. Although the load balancing algorithm has been executed multiple times, the *Load Balancing* component never becomes a performance bottleneck. The *Rendering* component uses OpenGL to rasterize triangles into pixels. We generated OpenGL buffer objects to store vertices, triangles and colors on GPUs. The *Rendering* component is the most time-consuming component. Its execution time increases as the number of triangles increases, and decreases as the number of GPUs increases. The execution time of the *Synchronization* component is small and does not vary much with different number of GPUs. The execution time of the *Frame Exchange* component is sensitive to the number of GPUs. The more GPUs are used in the system, the more data sending and receiving operations are involved to exchange screen portions among GPUs. Transferring those screen regions among GPUs has little cost. The *Display* component ports the composited frame to the monitor, which is very fast.

Then, we tested the system performance at different screen resolutions by rendering a total of 15 Power Plant models, as shown in Table 2. We chose the maximum possible number of strips at each resolution, as listed in the column of "# of strips" of the table, so that the system can achieve the most balanced workload. As shown in the table, the averaged computational time at a frame increases as the screen resolution increases. The *Frame Exchange* component has a major impact on the overall performance of the system. A higher resolution is chosen for the system, more pixels are transferred among GPUs to exchange frames; consequently, more execution time is spent on this component. Also, as the screen resolution increases, more time is spent on the *Display* component due to the need of displaying a larger size of the rendered frame, but the execution of *Display* component is efficient and the execution time is always less than 1 millisecond.

7.2. Efficiency of Load Balancing

We compared our approach to the approach without load balancing. Without load balancing, each GPU renders the vertices and triangles inside its corresponding display monitor. Thus, the frame size for each GPU to render is perfectly balanced, but the number of vertices and triangles assigned to each GPU may not be balanced. The approach without load balancing does not need a *Frame Exchange* component. We used the quad-GPU system for this comparison. We set $stripNum$ to be 2048 in our approach.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

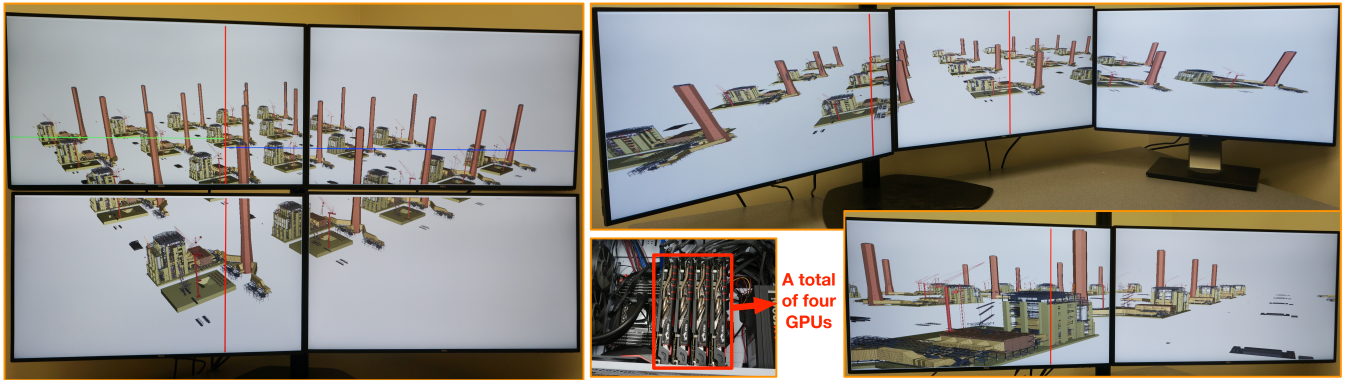

Figure 6: *The workstation we built with four GPUs and four display monitors. It supports the customized rendering of complex polygonal models using one, two, three, or four GPUs, where each GPU connects to one display monitor.*

Table 1: *Performance breakdowns for the system with* $stripNum = 2048$. *The values are averaged over 1200 frames.*

Configurations				Frame Time (millisecond)	FPS	Component Execution Times (millisecond)				
# of Power Plant	# of vertices (million)	# of triangles (million)	# of GPUs			Load Balancing	Rendering	Synchronization	Frame Exchange	Display
6	17.25	39.30	One	52.36	19.10	—	50.39	—	—	0.03
			Two	39.17	25.53	2.48	26.47	3.67	1.84	0.27
			Three	34.63	28.88	3.18	19.08	5.22	2.09	0.23
			Four	31.89	31.36	3.87	15.98	3.67	2.50	0.20
10	23.25	53.50	One	66.38	15.06	—	63.98	—	—	0.03
			Two	45.55	21.95	2.98	33.77	2.96	1.84	0.23
			Three	39.19	25.52	3.70	24.03	3.66	2.43	0.33
			Four	34.93	28.63	4.44	19.94	4.44	2.44	0.25
15	32.81	75.86	One	92.39	10.82	—	89.47	—	—	0.04
			Two	60.16	16.62	3.75	47.52	2.97	1.75	0.24
			Three	48.05	20.81	4.68	32.25	4.12	2.61	0.29
			Four	44.98	22.23	5.32	26.63	5.67	2.36	0.26
20	38.90	90.55	One	96.85	10.33	—	93.62	—	—	0.04
			Two	67.41	14.83	4.36	53.12	3.32	2.08	0.25
			Three	53.11	18.83	5.32	36.21	4.12	2.61	0.29
			Four	48.34	20.69	6.26	30.35	5.37	2.51	0.30

Table 2: *Performance breakdowns for the system at different screen resolutions. A total of 15 Power Plant models (32.81 million vertices and 75.86 million triangles) are rendered, and the values in the table are averaged over 1200 frames.*

Resolution (width×height)	# of strips	Frame Time (millisecond)	FPS	Component Execution Times (millisecond)				
				Load Balancing	Rendering	Synchronization	Frame Exchange	Display
512×512	512	41.13	24.14	5.39	26.20	5.69	0.49	0.07
1024×1024	1024	42.07	23.77	5.13	26.24	6.25	0.88	0.14
2048×2048	2048	43.60	22.49	5.32	26.63	5.67	2.36	0.26
4096×4096	4096	49.15	20.35	5.80	27.06	5.65	5.40	0.40

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

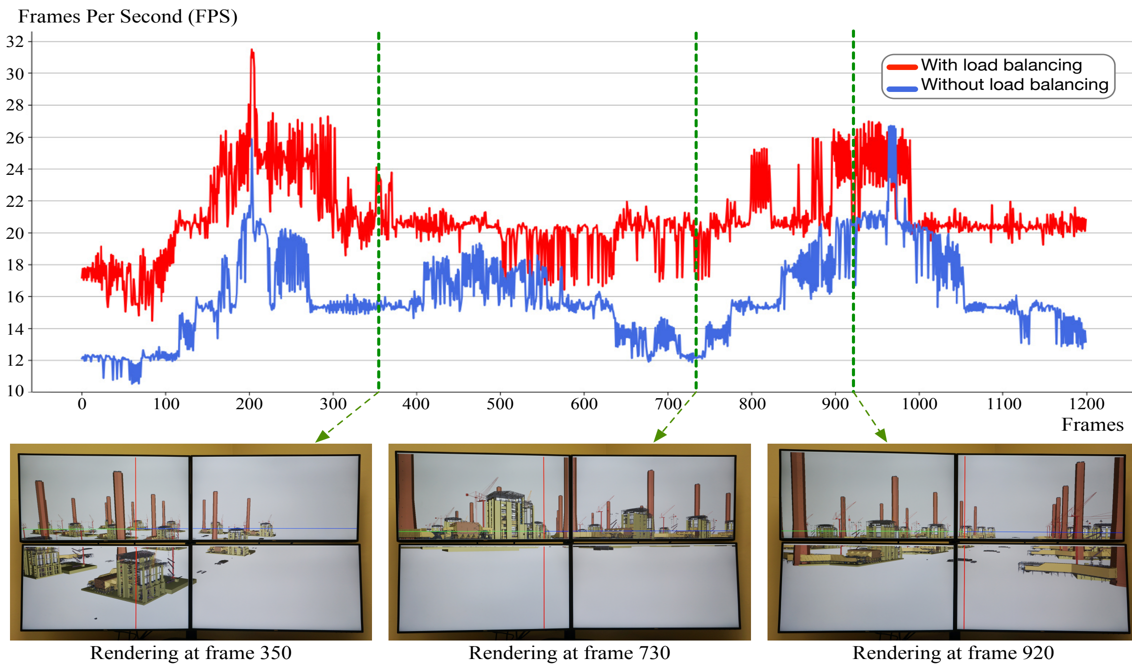

Figure 7: *Performance with and without load balancing over 1200 frames of the walkthrough camera path. The scene is composed of 20 Power Plant models. The three images at the bottom are the rendering results at specific frames.*

Figure 7 shows the performance comparison over the total of 1200 frames on the walkthrough camera path. We used the quad-GPU system. The scene used for this comparison is composed of 20 Power Plant models. Our approach achieved 20.95 FPS on average, while the approach without load balancing achieved 15.95 FPS on average. At the bottom of Figure 7, there are three examples of rendered frames. The overall performance of the system is determined by the GPU that renders the largest amount of triangles. At the frames 350 and 730, the FPS of our approach is much higher than the approach without load balancing. Our approach partitions the screen unevenly in order to balance the amount of triangles among GPUs. The same scenario has also occurred in the frame ranges of frame 0-420, frame 650-900, and frame 1060-1200. At frame 920, our approach partitions the screen near to the middle. In this case, the approach without load balancing also achieves a nearly balanced workload. Thus, its FPS in such a case raises to a value close to ours.

We compared our approach with the state-of-the-art dynamic load balancing techniques implemented in Equalizer, including *cross segment*, *2D* (sort-first), and *dynamic DB* (sort-last). We ran the *eqPly* mesh renderer application on our quad-GPU system. Both our approach and *eqPly* application rendered a scene composed of 15 Power Plant models and used a circular path that produces 300 frames. The path ensures that all triangles are inside the view frustum so that our approach and *eqPly* application always balance the same amount of data. Equalizer took more than 20 minutes to build the k-d tree for the scene of 15 Power Plant models. Different from that, our approach does not require a spatial hierarchy to represent input data. In our experimental results, our approach achieved an average of 22.16 FPS. The Equalizer's *eqPly* application achieved an average of 11.71 FPS when using the *cross segment*, 17.65 FPS when using the *2D*, and 10.72 FPS when using the *dynamic DB*. Thus, our approach resulted in better rendering performance than those dynamic load balancing techniques in Equalizer.

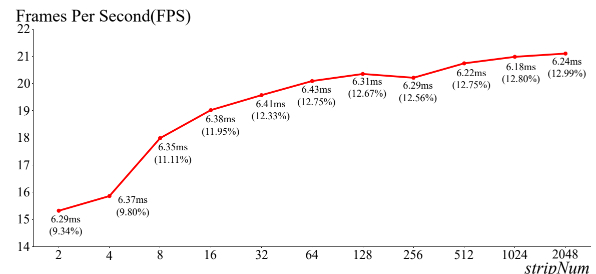

Figure 8: *The influence of different $stripNum$ values on the frame rate on the quad-GPU system for the scene composed of 20 Power Plant models. Each data point associated with the execution time of the load balancing algorithm. The value in the parenthesis is the percentage that the algorithm's execution time takes out of the total execution time.*

Figure 8 illustrates how different vaules of $stripNum$ influence the overall frame rate. As the value of $stripNum$ increases, the time to execute the load balancing algorithm does not vary significantly

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

(diff. $< 0.5ms$), and is always below 13% of the total execution time. Thus, a more balanced workload distribution (a larger value of *stripNum*) results in a higher frame rate. The total screen resolution of the quad-GPU system is 2048×2048, so the largest *stripNum* is equal to 2048, which means one strip corresponds to one pixel offset. When the *stripNum* is 2, the system can only partition the screen once at the middle, and would lead to an unbalanced workload. As the value of *stripNum* increases, the workload among GPUs becomes more balanced, and subsequently the overall frame rate increases.

8. Conclusion and Future Work

In this paper, we proposed an approach to balance the workload among multiple GPUs for rendering complex models composed of hundreds of millions of vertices and triangles. Our load balancing and screen partitioning algorithms are designed with a fine-grained parallelization on the GPU, and support the rendering on a large tiled display. The algorithms provide good scalability on the different number of GPUs. GPUs in our system perform independent computing and rendering tasks, and they are synchronized for screen composition and display. The GPU-to-GPU communication is only for transferring portions of the rendered frame rather than transferring 3D geometries so that the communication overhead does not become a performance bottleneck.

In this work, we demonstrated the efficiency of our approach using CAD models, where objects are individual design components that are usually crafted by engineering designers. Our approach can also be applied to a surface model after partitioning it into grids of primitives in 3D space. In the future, we would like to employ a grid-based spatial partitioning technique to preprocess surface models and incorporate them into our rendering system.

Our approach has great potential to be integrated with level-of-detail (LOD) and out-of-core techniques. The integration will boost the system's capability to handle a larger dataset that cannot even be stored in the GPU memory. While this paper presents the contribution related to load balancing, we have started looking for possible LOD and out-of-core techniques suitable for GPU architectures. When a dataset requires a storage size beyond the GPU's memory capability, it can be stored in the CPU main memory. With a given GPU memory budget, a LOD selection component could be applied to objects and determine a simplified version of the mesh (fewer vertices and triangles), and then fetch them from CPU to GPU using an out-of-core algorithm. We will implement this idea of integration, and evaluate the rendering quality and performance.

In the future, we will test our approach using more GPUs on a cluster system. In that case, the workload would not only be balanced among GPUs within a single workstation but also be balanced among cluster nodes. Furthermore, besides testing with CAD models, we would like to experience with other model types such as volume data and point cloud data, in which the parallelization would distribute data at a finer primitive level.

Acknowledgements

This work was supported by the National Science Foundation Grant CNS-1464323. We thank Nvidia for donating the GPU device that has been used in this work to run our approach and produce experimental results. The Power Plant model is brought through the courtesy of the University of North Carolina at Chapel Hill.

References

[ACCC04] ABRAHAM F., CELES W., CERQUEIRA R., CAMPOS J. L.: A load-balancing strategy for sort-first distributed rendering. In *Proceedings. 17th Brazilian Symposium on Computer Graphics and Image Processing* (Oct 2004), pp. 292–299. doi:10.1109/SIBGRA.2004.1352973. 2

[AMD17] AMD: AMD crossfire technology, https://www.amd.com/en/technologies/crossfire, 2017. URL: https://www.amd.com/en/technologies/crossfire. 1, 2

[AR05] ALLARD J., RAFFIN B.: A shader-based parallel rendering framework. In *Visualization, 2005. VIS 05. IEEE* (2005), IEEE, pp. 127–134. 2

[BH11] BELL N., HOBEROCK J.: Thrust: A productivity-oriented library for cuda. *GPU computing gems Jade edition 2* (2011), 359–371. 5

[BRE05] BHANIRAMKA P., ROBERT P. C., EILEMANN S.: Opengl multipipe sdk: A toolkit for scalable parallel rendering. In *Visualization, 2005. VIS 05. IEEE* (2005), IEEE, pp. 119–126. 2

[CSR*03] CZERWINSKI M., SMITH G., REGAN T., MEYERS B., ROBERTSON G. G., STARKWEATHER G. K.: Toward characterizing the productivity benefits of very large displays. In *Interact* (2003), vol. 3, pp. 9–16. 1

[EBA*12] EILEMANN S., BILGILI A., ABDELLAH M., HERNANDO J., MAKHINYA M., PAJAROLA R., SCHÜRMANN F.: Parallel Rendering on Hybrid Multi-GPU Clusters. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012), Childs H., Kuhlen T., Marton F., (Eds.), The Eurographics Association. doi:10.2312/EGPGV/EGPGV12/109-117. 2

[EEP11] EROL F., EILEMANN S., PAJAROLA R.: Cross-Segment Load Balancing in Parallel Rendering. In *Eurographics Symposium on Parallel Graphics and Visualization* (2011), Kuhlen T., Pajarola R., Zhou K., (Eds.), The Eurographics Association. doi:10.2312/EGPGV/EGPGV11/041-050. 2

[Eil07] EILEMANN S.: An analysis of parallel rendering systems. *White Paper: http://www.equalizergraphics.com/documents/ParallelRenderingSystems.pdf* (2007), 1–8. 2

[EMP09] EILEMANN S., MAKHINYA M., PAJAROLA R.: Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics 15*, 3 (May 2009), 436–452. doi:10.1109/TVCG.2008.104. 1, 2

[ESP18] EILEMANN S., STEINER D., PAJAROLA R.: Equalizer 2.0 — convergence of a parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics* (2018), 1–1. doi:10.1109/TVCG.2018.2870822. 1

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph. 21*, 3 (July 2002), 693–702. doi:10.1145/566654.566639. 2

[LWW*11] LIU H., WANG P., WANG K., CAI X., ZENG L., LI S.: Scalable multi-gpu decoupled parallel rendering approach in shared memory architecture. In *2011 International Conference on Virtual Reality and Visualization* (Nov 2011), pp. 172–178. doi:10.1109/ICVRV.2011.46. 2

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 23–32. doi:10.1109/38.291528. 2

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

[MWMS07] MOLONEY B., WEISKOPF D., MÖLLER T., STRENGERT M.: Scalable sort-first parallel direct volume rendering with dynamic load balancing. 2

[MWP01] MORELAND K., WYLIE B., PAVLAKOS C.: Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-data Visualization and Graphics* (Piscataway, NJ, USA, 2001), PVG '01, IEEE Press, pp. 85–92. URL: http://dl.acm.org/citation.cfm?id=502125.502141. 2

[NVI11] NVIDIA: *SLI best practices*. Tech. rep., Technical report, NVIDIA Corporation, 2011. URL: http://developer.download.nvidia.com/whitepapers/2011/SLI_\Best_Practices_2011_Feb.pdf. 1, 2

[Pap97] PAPE D.: pfcave cave/performer library (cavelib version 2.6). *Online documentation from the Electronic Visualization Laboratory, University of Illinois at Chicago, USA* (1997). 2

[SPEP16] STEINER D., PAREDES E. G., EILEMANN S., PAJAROLA R.: Dynamic work packages in parallel rendering. In *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization* (Goslar Germany, Germany, 2016), EGPGV '16, Eurographics Association, pp. 89–98. doi:10.2312/pgv.20161185. 2

[SZF*99] SAMANTA R., ZHENG J., FUNKHOUSER T., LI K., SINGH J. P.: Load balancing for multi-projector rendering systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (1999), ACM, pp. 107–116. 2

[Whi94] WHITMAN S.: Dynamic load balancing for parallel polygon rendering. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 41–48. doi:10.1109/38.291530. 2

[WLL*11] WANG P., LIU H., LI S., ZENG L., CAI X.: Multi-gpu compositeless parallel rendering algorithm. In *2011 12th International Conference on Computer-Aided Design and Computer Graphics* (Sept 2011), pp. 103–107. doi:10.1109/CAD/Graphics.2011.66. 2

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.