The Visual Computer manuscript No.
(will be inserted by the editor)

A Parallel Multi-View Rendering Architecture

Wallace Lages · Carlúcio Cordeiro · Dorgival Guedes

Received: date / Accepted: date

Abstract Multiple view rendering is a common problem for applications where multiple users visualize a common dataset, as in multi-player games and collaborative engineering tools. For a system to be able to render a large number of views at interactive rates efficiently, parallel processing is an attractive technique. In this work we present the implementation of a pipelined multiview light field renderer using a cluster with GPUs and MPI. We discuss the parallelization model and the problem of partitioning the tasks of the pipeline among the cluster machines based on the pipeline model and the costs of the stages. Our solution achieves 83% efficiency with ten machines, against only 11% efficiency of a naive parallelization.

Keywords Parallel rendering · Multiple viewpoint rendering

1 Introduction

Multiple views are typically used to provide a sense of depth and perspective for stereo and auto-stereoscopic displays [1–5]. However, multiple viewpoints are also necessary if we want to support several users visualizing a shared dataset at the same time. Among these applications we can include engineering projects, virtual reality environments, geographic models and massive multi-player computer games. In many cases, each user is connected to the rendering cluster by a thin client and is only interested on a particular view of the stored dataset.

To be able to render hundreds or thousands of views of a common dataset at interactive rates we need a solution which is both scalable and efficient. Clusters with GPUs have become an attractive platform for rendering tasks, since they have an excellent price-to-performance ratio and are easily available.

In this paper we present a parallel architecture designed to render multiple views interactively on a cluster with GPUs. The main challenges are to obtain an efficient parallelization, minimize the overhead due to communication and be able to scale as both more users and more machines are added to the system.

Our idea is to combine sampler nodes with image-based renderers so that computation can be parallelized and reused among views. We analyze the theoretical speedup of our proposed solution with respect to the number of views and provide experimental evidence that the proposed architecture is feasible for the task.

This paper is organized as follows: In Section 2 we discuss relevant related work. In Section 3 we describe the architecture and parallelization strategy, while in Section 4 we explain details of the light field renderer. In Section 5 we provide experimental results that support our findings and in Section 6 we discuss future work and conclude.

This paper is partially supported by IBM, CNPq, CAPES, FINEP and Fapemig

W. Lages · C. Cordeiro · D. Guedes
Deep Computing Visualization Center, Universidade Federal de Minas Gerais, LVAD-DCC-ICEx Sala 4010, Av. Antonio Carlos, 6627, Belo Horizonte, MG, 31270-010, Brazil
Tel.: +55-31-3409-5960, Fax: +55-31-3409-5858
E-mail: {wlages,carlucio,dorgival}@dcc.ufmg.br

2 Background and related work

In order to better discuss our contributions, we first provide a brief overview on multiple viewpoint rendering and parallel rendering.

2.1 Multiple viewpoint rendering

Rendering multiple views using the standard graphics pipeline is a difficult problem. The rasterization architecture currently adopted requires an early decision of the viewpoint, which makes it difficult to explore coherence between views in a common environment. For this reason, Halle [2] suggested rendering the views into a spatio-perspective volume, created by rendering epipolar images. Hübner et al. [4] developed a multi-view splatting technique which renders the necessary views for each splat just one time. They use the GPU to project each splat to all views at once.

The concept of using a sampled 4D buffer to generate new views is known as lumigraph or light field rendering. A light field [6] is a dense sampling of the 5-dimensional plenoptic function [7], which describes the radiance of every point in space (P_x, P_y, P_z) in all directions (θ, ϕ). If we consider only the subset of rays leaving a bounded object, the function can be further reduced to 4 dimensions [8].

In our work, we are interested in using the light field of sample images as a way to share rays among views and avoid duplicated computational effort. Even using ray sharing, the cost of rendering multiple views interactively is still larger than the power available in a typical graphics card. For this reason, we propose a scalable parallel architecture. Annen et al. [1] describes a distributed system for rendering multiple views for parallax displays. However, they did not explore any sort of coherence in the views, and focused only at scalability and load balancing issues. Yang et al. [9] developed a system that distributes some of the steps necessary to reconstruct a single view, achieving scalable bandwidth costs. Similar to ours, their solution does not use a ray buffer, so it can be applied to dynamic scenes.

2.2 Sampling issues in light field rendering

The minimum number of samples required to reconstruct a signal is a classic problem in image processing and computer graphics. Sampling analysis in image based rendering is a difficult problem because it involves a complex relationship among three elements: the depth and texture information of the scene, the number of sample images, and the rendering resolution [10–12].

It is well known that depth information, besides improving rendering quality, also affects sampling rate. Since per-pixel depth information is free on synthetic images, it has been used on some recent work [13, 14], although without considering parallelization.

It is also important to use a good sampling structure. For objects it is common to use a spherical parametrization [15, 14]. For environments, however, this is not so clear. The simplest approach is to sample the scene using a regular grid [16]. Another possibility is to generate samples adaptively as in [17]. In our case, we will use a number of fixed views as samples.

2.3 Parallel rendering

Parallelism has been used many times for accelerating the rendering process. To achieve parallelism, we can either divide the rendering steps among the processors (task parallelism, or pipelining) or split the rendering data (data parallelism). Another option is to explore the time dimension, computing views for different moments in parallel.

Sloan and Hansen [20] present three techniques for reconstructing lumigraphs in parallel. However, they use as target a distributed shared memory ccNUMA computer, not a cluster, which is a more common architecture at the present. A more recent work has been developed by Strasser et al. [21], the MLIC system, which uses an image cache to decouple image rendering from visualization and distributes the rendering calculations among multiple computers. They decompose the space around the view-point into six pyramids. Images are placed at fixed positions with respect to the viewpoint and may be further refined with a kd-tree. Displaying the image database involves visiting all the polyhedra and traversing the accompanying kd-trees in a top-down, back-to-front manner.

Many hybrid geometry/image systems have also been proposed for parallel rendering of massive models. The basic idea is to render objects far from the viewpoint using faster image-based techniques, since they will result in smaller elements in the display. Among them we may cite the work of Wilson and Manocha [22], Aliaga et al. [23], and Debevec et al. [24].

3 Parallel multi-view rendering

In order to analyse our solution, we first develop an analytical model for it. We are interested in understanding how the multiple elements of the problem affect the resulting rendering time and how scalable it will be when we increase the number of machines. We compare our solution with what we call the naive solution, which would be just to render each view independently.

First we define T_{naive}, the time required to render a number of views V using the standard rendering pipeline. That turns out to be just gV, where g is the time to render a single view:

$$T_{naive} = (gV)$$

To render multiple views in a scalable fashion, we need a parallel architecture. The most naive parallelization is to

divide the views among the available processors P, so that the time between two consecutive views is:

$$T_{||naive} = (gV)/P \tag{1}$$

In the most optimistic case, we can assume that an image can be sliced between two or more processors, so the load is evenly distributed. This architecture is depicted in Figure 1.

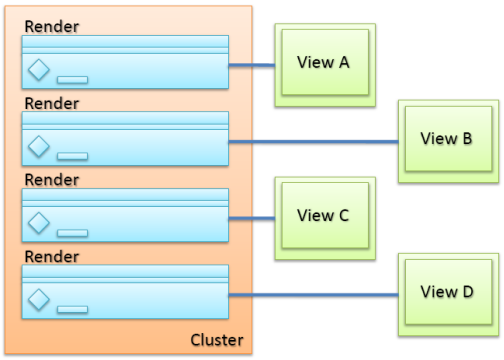

Fig. 1 Intuitive architecture: the views are distributed among the available GPUs. Each GPU executes the entire pipeline for each view.

However, to improve the performance while rendering multiple views, we would like to reuse computation that can be shared among the views. We propose the use of a light field renderer to render the views from scene samples instead of the original geometry. If S is the number of images necessary to sample the environment and c the time to compose a new image from the samples, we can split the processing effort into sampling and compositing steps, so that:

$$T_{lfr} = (Sg + Vc)$$

This formulation is faster than T_{naive} above a certain number of views as long as $c < g$. The efficient parallelization of this new algorithm requires a change in the architecture, since all compositing tasks must access the samples being generated, so we must guarantee that samples are computed first. To achieve that we separate the processors into a 2-level superscalar pipeline: the first level generates the samples, while the second level for combines the views (Figure 2).

From the P processors available, K are used to produce the S samples of the environment, while the remaining $(P-K)$ are used to compose the final V views. The time needed to produce all samples will be divided by the K processors, as the time to compute the views will de divided by the remaining processors. In the new system, as in any pipeline, the time between the output of two consecutive images from

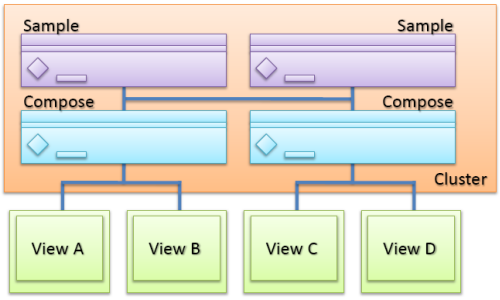

Fig. 2 Proposed architecture: the pipeline is divided into sampling and compositing stages. The GPUs are assigned to each stage as a function of the number of views.

a view is equal to the time taken by the slowest stage of the pipeline:

$$T_{||lfr} = Max\left(\frac{Sg}{K}, \frac{Vc}{P-K}\right) \tag{2}$$

We know that $T_{||lfr}$ is minimum when both stages takes the same time[1]. The value of K at this point can be found by equaling the expressions for each stage and solving for K:

$$K_{opt}(S,g,V,c,P) = \frac{SgP}{Vc+Sg} \tag{3}$$

Equation 3 defines the value of K which minimizes the time in a ideal system, where each processor can be partitioned freely among the two stages. In a real scenario, however, only one type of process will run on each GPU. This division makes the integration with existing pipelines easier, considering their current limitations. For these reasons, we will consider that only an integer number of processors can be allocated for each stage.

One may think that simply rounding K_{opt} to the nearest integer would give an adequate solution. However, as in many integer programming problems, that is not the case.

The restriction that $K \in \mathbb{Z}_*^+$, limits the choices of K so that a perfect balancing cannot be achieved for all V. This transforms the problem of finding the minimum of a continuous function into a problem of minimizing a function that only admits integer solutions. Specifically we want to find a function that computes the value of K which makes the $T_{||lfr}$ the smaller possible for each scenario.

Observing the expression of $T_{||lfr}$ (Eq. 2), we note that when only the number of viewpoints change, the time is given by the greatest value between a constant function of

[1] It can be shown that if that is not the case, the time would be that of the longer stage, and its time could be reduced by transferring some of the resources from the other stage to it, until the point when that procedure would turn the other stage's time higher, when the process would have to be reverted.

value Sg/k and a line of slope $c/(P-k)$. So, for $K \in [1,P)$, we have:

$$\begin{cases} T_{||lfr} = \frac{Sg}{K}, \forall\, V \leq \frac{Sg}{c}. \\ T_{||lfr} = \frac{Vc}{(P-K)}, \forall\, V \geq \frac{Sg}{c}. \end{cases}$$

Figure 3 shows the discussed behavior for the possible values of K for a cluster of five machines. When the value of K decreases, we obtain lines of greater slope and smaller constants. The curve of smaller time begins at $K = 4$ (1 composer and 4 samplers), and decreases until $K = 1$ (4 composers and 1 sampler). Generally speaking, if $(Vc)/(P-K)$ is greater than the curve for the next K ($(Sg)/(K-1)$) the next value of K should be used.

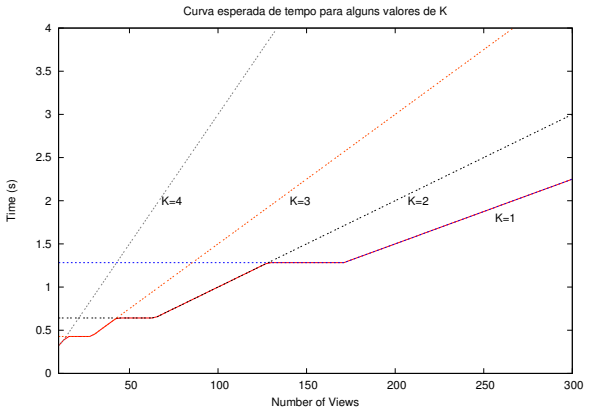

Fig. 3 Total time for different values of K for a five GPU cluster. The continuous curve is the optimal time obtained using Equation 4. The sampling time was $g = 0.3$ sec, the composition time was $c = 0.03$ sec, and the number of samples was $S = 4$.

The point where $T_{||lfr}(K) = T_{||lfr}(K-1)$ happens when:

$$\frac{Vc}{P-K} = \frac{Sg}{K-1}$$

Isolating K we get the right integer version of Eq. 3:

$$K_{opt}(S,g,V,c,P) = \left\lfloor \frac{SgP+Vc}{Vc+Sg} \right\rfloor, \; 0 < K_{opt} < P \qquad (4)$$

4 Light field rendering

Our light field renderer builds upon the work on per-pixel depth information for the correction of rays during reconstruction [13,14,25,26].

There are many ways of parametrizing the 4D light field. The best one depends on the application domain, but they usually have good sampling characteristics and can ease the reconstruction task. Among the parametrizations proposed in the literature we may cite: two planes [6], cylindrical [27], spherical [14,15], two spheres, plane-sphere [28], and non structured [13,29,30].

For this paper we chose to use a heightfield terrain. For this reason, we decided that a 2 plane parametrization (2PP) would be a good fit. In 2PP, each ray entering the scene is indexed by a pair of points (u,v) and (s,t) on the two planes. The plane for (s,t) is known as the view point plane and the one for (u,v), as the image plane.

4.1 Acquiring depth images

Each block of the scene is enclosed in the volume defined by the planes $z = 1$, $z = 0$, $x = \pm 1$, and $y = \pm 1$. Four cameras are at positions $(-1,-1,z)$, $(-1,1,z)$, $(1,-1,z)$, and $(1,1,z)$ as shown in Figure 4. At each frame, the 4 cameras render the scene and store the depth in the alpha channel as:

$$d = \frac{|P_g - P_1|}{|P_1 - P_0|}$$

where P_1 and P_0 denote the intersections of the viewing ray with the planes $z = 1$ and $z = 0$, respectively. P_g is the point where the viewing ray intersects the geometry (This is demonstrated in Figure 5, for the ray exiting camera Ca).

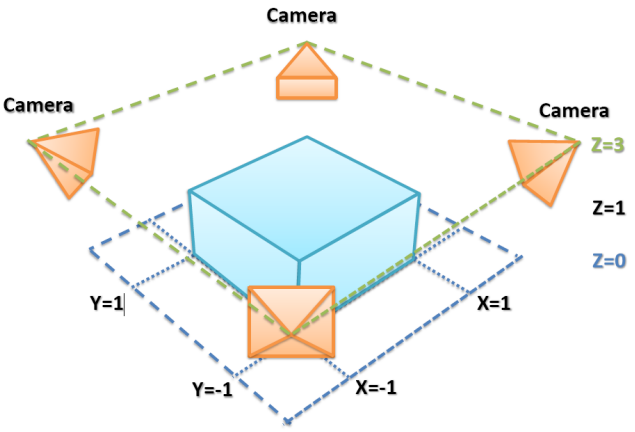

Fig. 4 Four cameras are used to sample each tile. The sample volume is delimited by the planes $z = 0$, $z = 1$, $x = \pm 1$, $y = \pm 1$

4.2 Rendering new views

The rendering algorithm was implemented in a GLSL fragment program. To render a new view, we need the position of the viewing camera C_{vp}, the position for each of the k sample cameras C_{kp} and the image plus depth captured by each one C_{ki}. We also need a projection matrix $Proj$ to specify the camera model. In this work we consider that all the cameras

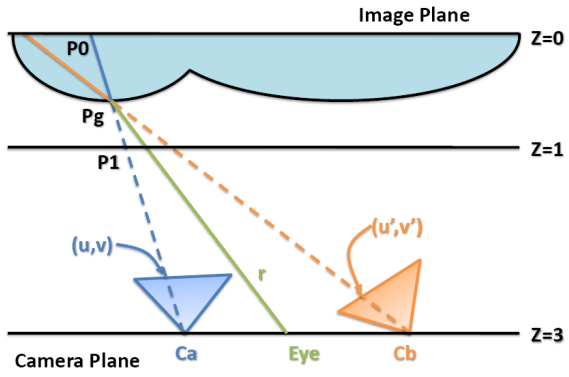

Fig. 5 Given the ray r, we must find the coordinates (u,v) and (u',v') which correspond to the intersection point P_g

are always looking to the center of the tile, so orientation information is not needed.

To be able to use the fragment shaders, we need a render surface. So, for each tile described in the previous Section, we draw a quad. The quad is parallel to the plane $z = 0$ and positioned at the origin. This will guarantee a render surface as long as the desired camera has a Z coordinate greater than the near clipping distance.

To render a new view we must decide, for each pixel, which samples to use. If the depth is known, we can use image warping to project pixels from one view to another[31]. This can be done performing a search along the view ray, re-projecting the point into each camera and comparing the distance with the value stored in the image [14].

First, we need to compute the viewing vector for each pixel. This can be computed passing the vertex position as an interpolated variable in the vertex shader:

$$v = normalize(VertexPos - D_v)$$

Next we need to compute the full projection matrix for each camera C_{kp}. This can be achieved composing the projection matrix $Proj$ with each camera position and orientation:

$$Proj_k = Proj * rot_k * offset_k$$

If more than one camera is looking at the same point, we choose one of the samples.

We chose to apply inverse warping so that we could use GPU interpolation and avoid holes in the image. In order to find the correspondence between the cameras we need the depth of each pixel in the new image. There are many ways to obtain that, but they all involve some kind of search along the view ray v, for the intersection with the surface implicitly defined by the other cameras. For the sake of simplicity, we decided to use a linear search.

The search starts at the point where v intersects the plane $z = 1$. At each iteration, the current point can be expressed by:

$$P = plane.s1 + v * \delta$$

where

$$\delta = \frac{|plane.s - plane.s1|}{steps}$$

and $plane.s1$ and $plane.s$ correspond to the intersections between v and $z = 0$ and $z = 1$ respectively.

To find the intersection with the surface, we employ an approach similar to the one used by Todt [14]: at each step we project the current point onto each camera k. This will yield a (x,y) pair that can be used to retrieve the surface position C_{ks} stored for each camera along the ray $P - C_{kp}$:

$$C_{ks} = plane.s * C_{ki}.depth + (1 - C_{ki}.depth) * plane.s1$$

However, instead of comparing the position P with the estimate C_{ks}, we compare the distance from C_{kp} to the current point P and the distance from C_{kp} to the surface C_{ks}. if $(|C_{ks} - C_{kp}| - |P - C_{kp}|) < 0$, the point P has entered the surface and we can refine the position with a binary search.

We don't interrupt the search until P is considered inside the surface for all the camera estimatives C_{ks}. At that point we tested two rules to select the best sample. In the first one, we chose the sample from the camera with the best view of the surface. For each camera we computed the projection of the sample viewing ray $P - C_{kp}$ into the viewing vector v. The sample with the greater projection, thus more closer to the output viewing ray, was selected. In the second alternative we chose the sample with the smallest difference between the current point and the camera surface estimative C_{ks}. This corresponds to the sample with a better estimative of the surface. As we show in Section 5.3, the resulting image for the rules is similar, altough the selection of samples may differ significantly.

5 Experimental results

In this Section we discuss our results in terms of speedup, efficiency, communication costs and load balancing for a real implementation. We have implemented the system described in this paper on a Linux cluster composed of 11 dual-Pentiums 3.40GHz 64bits machines connected by a Gigabit Ethernet Switch. Each PC had 3GB of memory and a Nvidia GeForce 7900 GTX/PCI/SSE2 GPU. We used OpenGl and GLSL for graphics and LAM/MPI 7.0.6 for interprocess communication.

As input database for the performance evaluation we used a scene composed of a highly tessellated tree over a heightfield terrain. The scene has approximately 500 thousands triangles and was lit by a simple phong shader. An

output of the system is shown in Figure 6. For the quality analysis we used the well known Stanford Bunny model [2] as input, as shown in Section 5.3.

During the simulation we measured:

1. the time to render a view from the original data (the sample time g);
2. the time to render a view from the obtained samples (the composition time c), which was measured from the instant triangles are sent to the GPU to the time the frame buffer is ready;
3. the time to transfer data from CPU to GPU and back (calls to *glTexImage2D()* and *glReadPixels()*, respectively);
4. the time spent on communication, waiting for data from the network (the time the second stage has to wait for the samples);
5. the total running time, the time elapsed from the beginning to the end of the rendering.

It is important to notice that, due to the intrinsic overlap of communication and computation in our pipelined architecture, the time spent on communication is almost always smaller than the total time of the network transmission.

Fig. 6 Render of the scene used in the performance and scalability evaluation

For each camera we rendered 5 frames, moving the viewpoint in them. The samples were taken with a 45 degree projection and a resolution of 400x300 pixels. Output views were rendered at 200x150 resolution. To improve the throughput, network transmission was intercalated with rendering. The sample images were transmitted uncompressed using point-to-point communication.

5.1 General performance

For the performance measurementes we used five GPUs, distributed among the stages according to Equation 4 with values fro the compostion and render times measured during a previous execution ($c = 0,03s$ and $g = 0,3s$). The results were $K = 2$ for $V \leq 128$, and $K = 1$ for $V \geq 129$. To validate the value of K (number of sampling GPUs) obtained, we ran the systems with the two options for the entire range of V. As can be seen in Figure 7, above $V = 130$ the optimal choice is to allocate just one GPU for sampling, as derived from the model.

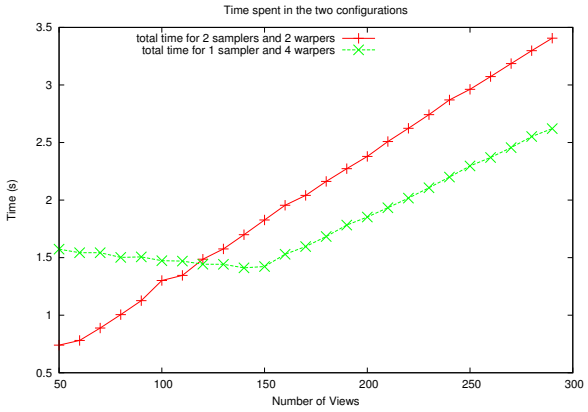

Fig. 7 Time spent in two different configurations of K for the same number of GPUs.

It is interesting to notice that the shape of the measured curves fits well to those of the theoretical model (Fig. 3), although rendering and sampling times were not the same.

Our system rendered 290 views in 2.61 seconds, spending approximately 9 ms per frame. We measured the total rendering time for and increasing number of views and it is plotted in Figure 8 against the theoretical values for $T_{||naive}$ (Eq. 1) and $T_{||lfr}$ (Eq. 2). We can see that the measured time follows the time predicted by the model with a small increase, probably due to communication overhead.

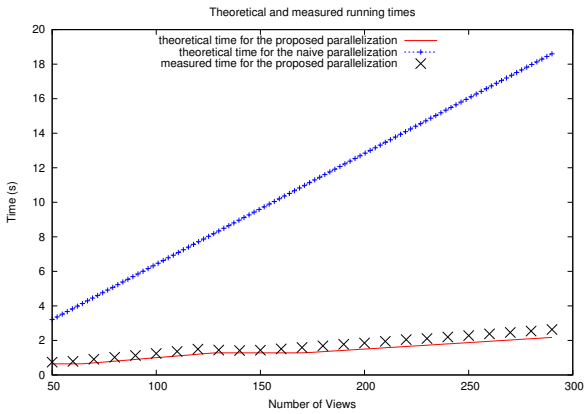

Fig. 8 Time spent by the proposed parallelization using 5 GPUs

[2] http://graphics.stanford.edu/data/3Dscanrep/

To verify that, we show the time spent in each sub-task in Figure 9. The results show that the GPU/CPU communication costs (texture download/upload) were negligible. The time spent in MPI receive routines increased when more nodes were allocated to sampling, as expected, since we used a point-to-point protocol, and they account for almost all the difference from the theoretical model. A collective communication function might help keep this from growing too fast, but this is not a realm problem in this case, since the sampling costs are independent from the number of views.

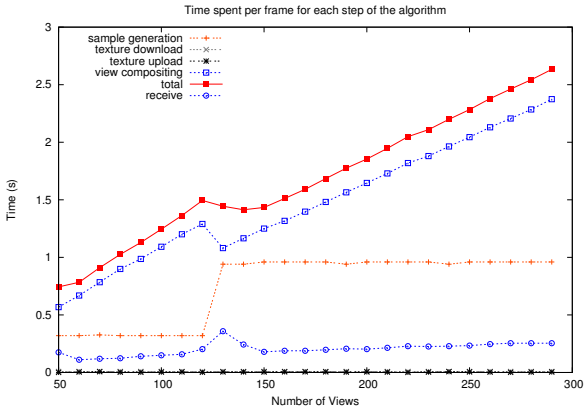

Fig. 9 Time spent in the main steps of the algorithm

5.2 Scalability

To evaluate the scalability of our parallel light field renderer, we ran it under varying configurations and loads, measuring its speedup and efficiency in each case. we also compared it to a parallel implementation of the naive renderer (just by computing all views directly from the dataset in parallel using all machines available). We computed the speedup of both implementations by calculating the ration of the execution time of the most efficient single-machine implementation between the two versions considered, the light field renderer, to the total running time of each parallel implementation. We used the sequential light field renderer because it would be faster than the naive renderer in that case (and it would not be fair to compare the parallel versions to the slower implementation). The efficiency was computed as the ratio of the speedup to the number of machines used. An ideal parallel implementation would achieve linear speedup (it would perform N times as fast as the sequential version with N machines), which would lead to 100% efficiency. Due to limitations in the parallel execution, like synchronization and communication overheads, and load unbalance, most parallel applications do not perform that well.

First, we varied the number of machines from 5 to 10, keeping only one machine assigned to the sampling stage in the light field renderer ($K = 1$). Each configuration was used to compute 1000 views and the execution times were measured. Table 1 shows the efficiency results based on the measured times. Clearly, the parallel light field renderer performs better, reaching 83% efficiency with 10 machines, agains just 11% of the naive renderer.

Total GPUs	Efficiency (Ligth field)	Efficiency (Naive)
5	0.7147	0.1029
6	0.7567	0.1049
7	0.7922	0.1063
8	0.8117	0.1080
9	0.8184	0.1084
10	0.8257	0.1096

Table 1 Efficiency of proposed and naive parallelizations for 1000 views. One GPU was allocated to sampling in all cases

Finally, we fixed the cluster size and varied the number of views, evaluating the efficiency of both parallel versions as we did so. We used five machines in the experiment, and configured them based on the value of K_{opt} given by Eq. 4, as in Section 5.1 ($K \in \{1,2\}$). The results for the computed efficiency are shown in Figure 10. The naive implementation has a better efficiency with fewer views, since there is less communication overhead in that case, but it never goes above 20%. As the number of views increase, however, the efficiency drops, getting closer to 10%. The parallel light field renderer, on the other hand, performs better in all cases. For fewer views, $K_{opt} = 2$ and the performance is above 80%. As the number of views approaches 130, the performance clearly degrades, since the system is reaching the point where a reconfiguration is needed. Once the reconfiguration takes place ($K_{opt} = 1$), efficiency rises again and stays above 80% for the rest of the experiment.

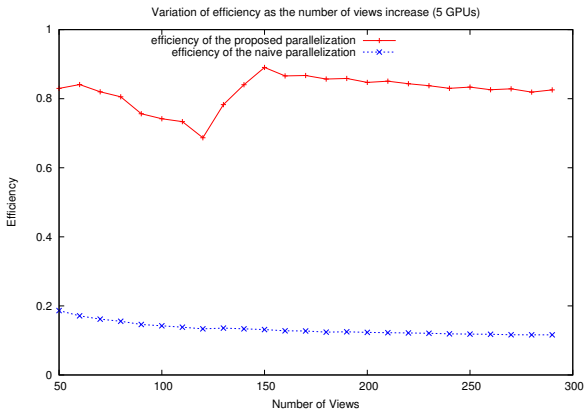

Fig. 10 Scalability of the naive and proposed parallelizations

5.3 Rendering Quality

Ideally, an image generated by the light field renderer would be identical to an image generated directly from the geometry. However, as discussed before, the quality of output image depends on many different factors such as sampling rate and the strategy used to find the surface. In our tests we used four sample cameras, which is a reasonable number for our scenes. We also used a very simple search strategy to find the surface intersection (linear search). To make the comparison easier to evaluate, we used the well known Stanford Bunny model[3] as input.

In Figure 11 we can see four different views of the bunny, rendered from the acquired samples. Pixels from each sample were colored in a different color (red, green, blue, yellow). We can see that the model is well covered by the four cameras. In fact, we observed that most of the surfaces are covered by at least three cameras (not shown).

To evaluate the quality of the renderer, we used the Perceptual Image Diff Utility[4]. This utility makes use of a computational model of the human visual system to compare two images. A comparison between an image produced by our method and one rendered from the original geometry is shown in Figure 12. A fixed number of 100 steps were used on this test. Visible differences are highlighted in red. Most of the differences occur along the object outline.

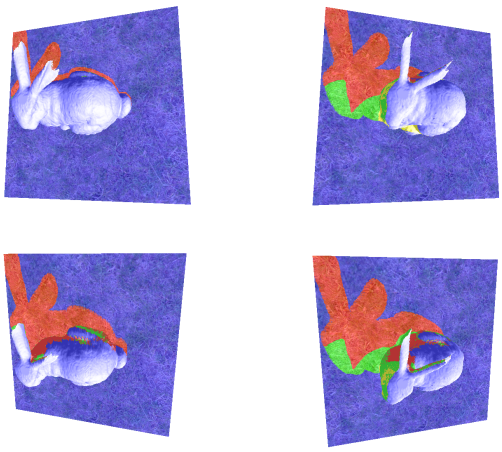

Fig. 11 Contribution of the input samples to the final image. Pixels from each sample were colored in a different color (red, green, blue, yellow).

As discussed previously, the image quality also depends on the quality of the search along the viewing ray v. In our case, the search is heavily influenced by the size of steps

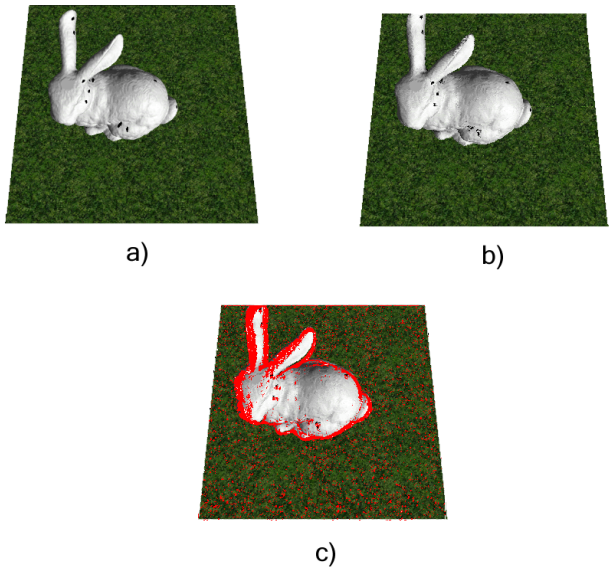

Fig. 12 Output quality: a) render from the geometry; b) output of the light field, c) perceptual difference

on the linear search. If the step is too large, it will miss smaller features of the surface. If it is too small, the render will take too much time. Another important parameter is the number of cameras that confirm that a surface exists. Our algorithm considers that a surface was found when *all* four cameras hits a surface. If some feature is not sampled by all the cameras, the algoritm will search to the end of the volume. This is ilustrated on Figure 13. Images 13.a, 13.b and 13.c were rendered with 100, 200 and 300 steps respectively, with 100% of confidence (4 cameras must confirm the existence of a surface). Image 13.d was rendered with 100 steps and 75% of confidence (3 cameras must confirm the existence of a surface). A smaller confidence will improve the rendering of some features but if it is too low, false surfaces and artifacts will appear (look between the ears of the rabbit).

We have also compared the two methods proposed to choose the sample to use for each pixel, as discussed at the end of Section 4.2, for both datasets considered in this work. In Figure 14, the first column shows the cases where the sample with the best view of the surface was chosen, while the second column shows what happened when the sample with the better estimative of the surface was chosen. As we can see, although the output images are quite similar, the contribution of each sample camera may be very different, depending on the model used.

[3] http://graphics.stanford.edu/data/3Dscanrep/
[4] http://pdiff.sourceforge.net/

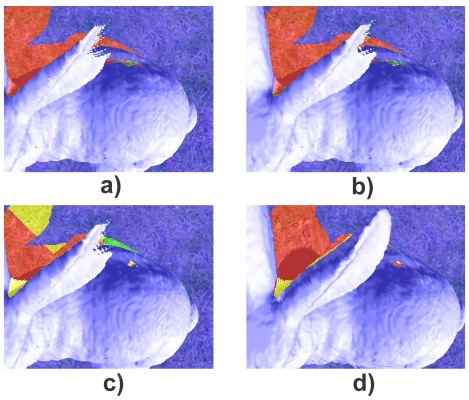

Fig. 13 Influence of the number of steps and confidence on the output image: a) 100 steps, 4 cameras b) 200 steps, 4 cameras, c) 300 steps, 4 cameras d) 100 steps, 3 cameras

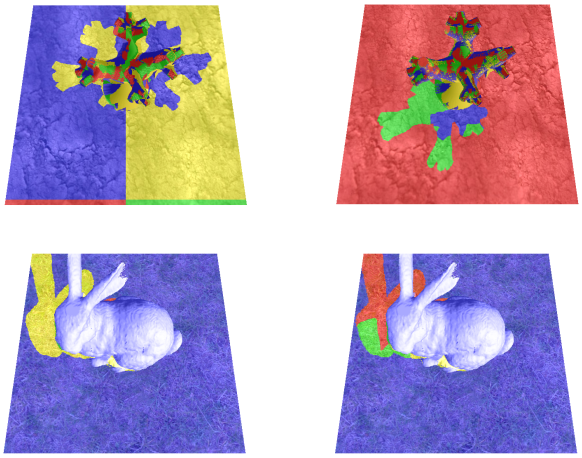

Fig. 14 Comparison of the two methods of choosing pixels. First column: sample with the best view of the surface. Second column: sample with better estimative of the surface.

6 Conclusions and future work

In this paper we presented a parallel architecture for rendering multiple views in a cluster of GPUs. We have shown a theoretical analysis of speedup and scalability of the proposed system, showing how it can be superior to a naive parallelization. In the results, we have supported this analysis by running experiments on a rendering cluster, when we achieved more than 80% efficiency for the proposed method, against only 11% efficiency for the naive solution.

The main insight of this work was the use of an image-based renderer to avoid duplicate work among the views. At the same time, an efficient parallelization was found, what enabled the system to scale very well with the number of views.

The main trade-off of the system is in the sampling process, which may be complex for some scenes. In a future work, we would like to experiment with adaptive camera positioning, so that we can improve the sampling of more complex scenes.

Another area of work is the overall optimization of the system, using faster texture transfer primitives (pixel/frame buffer objects), dynamical load balancing and faster image composition, using better search strategies.

Acknowledgements The authors would like to thank João Luiz Campos and the Tecgraf Group at PUC University, Rio de Janeiro, for their support with the experiments. The Deep Computing Visualization Center at Universidade Federal de Minas Gerais was started by an IBM Shared University Research (SUR) Grant.

References

1. Annen, T., Matusik, W., Pfister, H., Seidel, H.P.Z.M.: Distributed rendering for multiview parallax displays. Technical report, Mitsubishi Electric Research Laboratories (2006) "Distributed Rendering for Multiview Parallax Displays", SPIE Conference Stereoscopic Displays and Virtual Reality Systems XIII, Vol. 6055, pp. 231-240, January 2006, SPIE Proceedings.
2. Halle, M.: Multiple viewpoint rendering. In: SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques, New York, NY, USA, ACM (1998) 243–254
3. Hasselgren, J., Akenine-Möller, T.: An efficient multi-view rasterization architecture. In Akenine-Möller, T., Heidrich, W., eds.: Eurographics Workshop/ Symposium on Rendering, Nicosia, Cyprus, Eurographics Association (2006) 61–72
4. Hübner, T., Zhang, Y., Pajarola, R.: Multi-view point splatting. In Lee, Y.T., Shamsuddin, S.M.H., Gutierrez, D., Suaib, N.M., eds.: GRAPHITE, ACM (2006) 285–294
5. Stewart, J., Bennett, E., McMillan, L.: Pixelview: A view-independent graphics rendering architecture. In Akenine-Möller, T., McCool, M., eds.: in: proc of Graphics Hardware. (2004) 75–84
6. Levoy, M., Hanrahan, P.: Light field rendering. In: SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, New York, NY, USA, ACM (1996) 31–42
7. Adelson, E.H., Bergen, J.R.: The plenoptic function and the elements of early vision. Computational Models of Visual Processing (1991) 3–20
8. Gortler, S.J., Grzeszczuk, R., Szeliski, R., Cohen, M.F.: The lumigraph. In: SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, New York, NY, USA, ACM (1996) 43–54
9. Yang, J.C., Everett, M., Buehler, C., McMillan: A real-time distributed light field camera. In: In: proc. of the 13th Eurographics workshop on Rendering, Italy. (2002) 77–86
10. Chai, J.X., Chan, S.C., Shum, H.Y., Tong, X.: Plenoptic sampling. In: SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, New York, NY, USA, ACM Press/Addison-Wesley Publishing Co. (2000) 307–318
11. Lin, Z., Shum, H.: On the number of samples needed in light field rendering with constant-depth assumption. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR-00), Los Alamitos, IEEE (2000) 588–597

12. Lin, Z., Shum, H.Y.: A geometric analysis of light field rendering. Int. J. Comput. Vision **58**(2) (2004) 121–138
13. Schirmacher, H., Vogelgsang, C., Seidel, H.P., Greiner, G.: Efficient free form light field rendering (2001)
14. Todt, S., Rezk-Salama, C., Kolb, A.: Fast (spherical) light field rendering with per-pixel depth. Technical report, University of Siegen (2007)
15. Ihm, I., Park, S., Lee, R.K.: Rendering of spherical light fields. In: PG '97: Proceedings of the 5th Pacific Conference on Computer Graphics and Applications, Washington, DC, USA, IEEE Computer Society (1997) 59
16. Shade, J., Lischinski, D., Salesin, D.H., DeRose, T., Snyder, J.: Hierarchical image caching for accelerated walkthroughs of complex environments. In: SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, New York, NY, USA, ACM (1996) 75–82
17. Jeschke, S., Wimmer, M., Schumann, H., Purgathofer, W.: Automatic impostor placement for guaranteed frame rates and low memory requirements. In: I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games, New York, NY, USA, ACM (2005) 103–110
18. Molnar, S., Cox, M., Ellsworth, D., Fuchs, H.: A sorting classification of parallel rendering. Computer Graphics and Applications, IEEE **14**(4) (1994) 23–32
19. Crockett, T.W.: Parallel rendering. Icase report 95-31; nasa cr-195080, Institute for Computer Applications in Science and Engineering (ICASE) (1995)
20. Sloan, P.P., Hansen, C.: Parallel lumigraph reconstruction. In: PVGS '99: Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics, Washington, DC, USA, IEEE Computer Society (1999) 7–14
21. Strasser, J., Pascucci, V., Ma, K.L.: Multi-layered image caching for distributed rendering of large multiresolution datasets. In Raffin, B., Heirich, A., Santos, L.P., eds.: Eurographics Symposium on Parallel Graphics and Visualization, Braga, Portugal, Eurographics Association (2006) 171–177
22. Wilson, A., Manocha, D.: Simplifying complex environments using incremental textured depth meshes. ACM Trans. Graph. **22**(3) (2003) 678–688
23. Aliaga, D., Cohen, J., Wilson, A., Baker, E., Zhang, H., Erikson, C., Hoff, K., Hudson, T., Stuerzlinger, W., Bastos, R., Whitton, M., Brooks, F., Manocha, D.: Mmr: an interactive massive model rendering system using geometric and image-based acceleration. In: I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics, New York, NY, USA, ACM (1999) 199–206
24. Debevec, P.E., Taylor, C.J., Malik, J.: Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In: SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, New York, NY, USA, ACM (1996) 11–20
25. Heidrich, W., Schirmacher, H., Kück, H., Seidel, H.P.: A warping-based refinement of lumigraphs. In Thalmann, N., Skala, V., eds.: Proc. WSCG '99. (1999)
26. Vogelgsang, C., Greiner, G.: Adaptive lumigraph rendering with depth maps. Technical Report 3, IMMD 9, Universitaet Erlangen-Nuernberg (2000)
27. McMillan, L., Bishop, G.: Plenoptic modeling: An image-based rendering system. Computer Graphics **29**(Annual Conference Series) (1995) 39–46
28. Camahort, E., Lerios, A., Fussell, D.: Uniformly sampled light fields. Technical report, University of Texas at Austin, Austin, TX, USA (1998)
29. Buehler, C., Bosse, M., McMillan, L., Gortler, S., Cohen, M.: Unstructured lumigraph rendering. In: SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, New York, NY, USA, ACM (2001) 425–432
30. Takahashi, K., Naemura, T.: Unstructured light field rendering using on-the-fly focus measurement. In: ICME, IEEE (2005) 205–208
31. McMillan, L.: An Image-Based Approach to Three-Dimensional Computer Graphics. PhD thesis, University of North Carolina at Chapel Hill (1997)

Wallace Lages is CG researcher at the Deep Computing Visualization Center of Federal University of Minas Gerais (UFMG). He received his Bachelor's and Master degree in Computer Science from UFMG in 2006 and 2008, respectively. His main interests are real-time rendering, scientific visualization, computer games and human computer interaction. He has recently joined the Virtual Architecture Studio Laboratory at School of Architecture of UFMG.

Carlúcio Cordeiro is a graduate student of the Computer Science Department fo the Federal University of Minas Gerais (UFMG), Brazil. He has a B.S. in Computer Science from UFMG. His research interests include Games, Multi-view rendering, etc.

Dorgival Guedes is an Associate Professor of the Computer Science Department at the Federal University of Minas Gerais (UFMG), Brazil. He has a Ph.D. in Computer Science (1999) from the University of Arizona, Tucson, a M.S. in Computer Science and a B.S. in Electrical Engineering from the Federal University of Minas Gerais, Brazil (1992 and 1989, respectively). At UFMG, he is one of the directors of the Deep Computing Visualization Center, focusing on parallel solutions to visualization problems. His research interests include Computer Networks, Distributed Systems and Parallel Programming in general.