Ray Tracing on a Cell Cluster for Virtual Environments

Florian Bingel, Florian Mannuß, André Hinkenjann†
Bonn-Rhein-Sieg University of Applied Sciences, Sankt Augustin, Germany

Abstract
We present an interactive system that uses ray tracing as a rendering technique. The system consists of a modular Virtual Reality framework and a cluster-based ray tracing rendering extension running on a number of Cell Broadband Engine-based servers. The VR framework allows for loading rendering plugins at runtime. By using this combination it is possible to simulate interactively effects from geometric optics, like correct reflections and refractions.

Categories and Subject Descriptors (according to ACM CCS):
I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality
I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Ray Tracing

1. Introduction

Traditional VR frameworks often started as wrappers around scene graph APIs. With this configuration the rendering quality was limited to the capabilities of the wrapped API, usually textured, Gouraud-shaded triangles. Due to the support of shaders in modern scene graph APIs the visual quality can be improved a lot. Hardly any VR system supports advanced rendering algorithms, though. The use of ray tracing in Virtual Environments enables the suitability for applications like interactive optics simulation in education or the simulation of more realistic worlds in general. On the other hand, because of the compelling speed of hardware-based graphics, some simulations still have to be realized using graphics hardware. To be able to employ advanced renderers and hardware-based renderers in a single VR system, that system has to be very flexible. Ideally the system can be configured to use one of the available renderers without the need to rewrite the application. This in turn results in a separation of data and rendering, contrasting to the scene graph API approach.

Several approaches for fast, realistic rendering have been presented [PMpJS ’99] [WBW01] [RSH05] [BSP06] [GS08], also for the Cell Broadband Engine [BWSF06]. While these approaches represent the state of interactive ray tracing, they are stand-alone renderers. Bikker [Bik07] included a ray tracing kernel in a game engine. To achieve the highest performance and because of the game focus, the renderer is tightly coupled with the engine. In [WDB ’06] the authors present a fast ray tracing approach for visualizing CAD models in Virtual Environments. While the authors present their realistic, interactive rendering approach in detail, they leave the problem of integrating a renderer into a VR framework open.

In this paper we present a combination of a highly modular VR system together with a cluster-based ray tracing renderer. The renderer implementation is done using the Cell processor by IBM, Sony and Toshiba [KDH ’05]. The VR system allows for the exchange of renderers by simply editing the configuration file.

In this paper we present a combination of a highly modular VR system together with a cluster-based ray tracing renderer. The renderer implementation is done using the Cell processor by IBM, Sony and Toshiba [KDH ’05]. The VR system allows for the exchange of renderers by simply editing the configuration file.

Section 2 gives a short overview of our VR framework and the special interface it has to rendering plugins. After that we give some details of the ray tracer running on the Cell chip. Load balancing of the render servers is the topic of section 4. We also present some results of the current status of the setup in section 5.

2. VR System

The software used for the system is a modular VR framework, called basho [HM04]. Figure 2 shows the top level structure of basho. It consists of a compact kernel that is handling interfaces for interaction devices (user input), actions
(things that happen, when user input is detected), scripting, scene data (separated from rendering), engines (parts of the system that alternate the scene) and sense renderers (not limited to visual output: e.g. audio rendering).

This way, the most useful renderer can be selected, depending on the application. If ultimate speed is needed, graphics hardware is still faster for many scenes. If physically correct refraction and reflection is needed, the cluster-based ray tracing renderer is a better choice.

3. Cell-based Ray Tracing

The render server implements a ray tracing algorithm on the Cell chip. The Cell chip consists of eight fast and power efficient vector units (called SPEs) and one traditional CPU, based on the Power 4 processor core (called PPU).

The platforms supported by the renderer are: PS3, QS21/22 and GigaAccel. While the PS3 is a very cost effective system allowing to build an inexpensive cluster with huge floating point performance, it lacks memory capacity. Much more memory is available on the QS22 blades which in turn are more expensive than the PS3. The GigaAccel 180 is a PCIe card for PCs that has an additional Ethernet network interface.

The ray tracer uses the bounding interval hierarchy (BIH) [WK06] to accelerate ray intersection queries. The BIH has the advantage that it can be rebuilt quite fast. This is because the strategy to place the separating planes of the spatial data structure uses a greedy heuristic in contrast to the surface area heuristic (SAH) [Hav00]. Although the SAH produces more efficient trees, it is slower than the original BIH approach when dynamic scenes require a constant rebuild of the acceleration structure.

Currently the data structure is built on the PPU, allowing for build-rates of 70 ms for 18000 triangles. We already have an SPE version of the BIH-build, which is not yet faster than the PPU-build. We plan to parallelize the build across the SPEs.

Rays are traced in packets of configurable size. We found a packet size of 4x4 rays to be the most efficient for primary rays for many scenes, using the available SPE registers best. When using larger packets, the limited local store (storing data and code) on the SPEs allows only for small caches, decreasing memory performance. When using smaller packets, coherency is not used well enough. Figure 3 shows the performance of a small scene, depending on the packet size, normalized to the rendering time needed for single rays.

The render server uses the software cache of the IBM SDK for Multicore Acceleration in conjunction with double buffering for memory access. It supports alpha transparency and bilinearly filtered textures.

Every change of the scene (which is supervised by basho) is propagated to the render servers by the PS3 ray tracer component. We implemented a lean and efficient protocol for the messages sent across the network.
4. Load Balancing

In the system, load balancing is done in the PS3 ray tracer component of basho. It is a separate module that is easily exchangeable (cf. figure 2). We have implemented two versions of load balancing for the render servers:

1. **Fixed size work packages**: This strategy assigns fixed sized square regions of the view port to the render servers on demand. Whenever a server has finished its job, it is assigned a new region to render. When the number of tiles is (much) larger than the number of servers, this results in reasonable load balancing. The following left image in figure 4 shows good load balancing with four render servers and four equally sized tiles, even though the number of tiles is not larger than the number of servers. However, when the most complex region (the eye) is moved into one region, this type of load balancing is not working well with this configuration, as shown in figure 4, right.

   We also found that this strategy is performing worse than the other strategy, because during a frame the SPEs are assigned different regions of the view port. Usually this means that data in the caches cannot be reused.

2. **Dynamic work packages**: To overcome the above mentioned problems, we have implemented another strategy that assigns variable sized regions to the servers. For n servers we have n regions. The regions are possibly re-sized after each frame. This is done by analyzing the rendering times of the servers. By this, the system always reacts to load imbalances with a delay of one frame, which is increasingly tolerable with higher frame rates. Predictive strategies are difficult to realize, because one has to predict the effects of ray tracing, which may be too time consuming. This is a topic of future research. Figure 5 (left) presents a screen shot of this strategy. Note that when the high load area (the eye) is moved, the tiles are adapted accordingly, as shown in figure 5, right.

5. Results

As described earlier, the system is freely configurable, including the choice of renderers and the number of renderers. Here, we report on a cluster setup of 1 - 8 PS3 consoles as ray tracing render servers. The setup can be seen in figure 6.

   It consists of a variable number of PS3s (render servers) and a Mac running basho including the render plugin with
client. The system is interconnected by a gigabit Ethernet network. In addition, figure 6 shows large TFT display which is used for demonstration purposes.

5.1. Test Applications

We have implemented some demonstrations showing different aspects of the state of our combination of ray tracing and interactive environments:

Eye surgery simulation: To include correct refraction in an eye surgery simulator, we used an eye model (courtesy VRmagic GmbH) with a refractive lens (cf. figure 7). The user can move the camera using a 2DOF device (mouse) or a 6DOF device.

Figure 7: Ray tracing for eye surgery simulation

Lens optics: An experiment with biconvex lenses showing the magnifying effect (cf. figure 8). The scene is fully interactive. The lenses can be moved around freely. Using the buttons on the desk, the refractive index of the lenses as well as their colors can be changed.

Figure 8: Lens experiments

Assembly simulation: This shows the interactivity of the system (cf. figure 9). Every part of the car can be disassembled and moved around. Like in the other interactive applications, the scene data structure is rebuilt from scratch every frame.

Figure 9: Assembly simulation

5.2. Benchmarks

We measured the average performance of the whole system and the average server rendering time to determine the overhead. The overhead is largely composed of the time needed to send commands and results over the network, to combine the results into a final image an writing it to the frame buffer of the graphics card. The benchmarks showed the eye scene (about 50k polygons) mentioned in subsection 5.1. The load balancing strategy was "dynamic work packages".

The following table shows the benchmark results for a resolution of 800 by 800 pixels:

<table>
<thead>
<tr>
<th>no of servers</th>
<th>overall time (s)</th>
<th>render time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.46</td>
<td>0.42</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
<td>0.21</td>
</tr>
<tr>
<td>4</td>
<td>0.14</td>
<td>0.10</td>
</tr>
<tr>
<td>6</td>
<td>0.10</td>
<td>0.065</td>
</tr>
<tr>
<td>7</td>
<td>0.09</td>
<td>0.058</td>
</tr>
<tr>
<td>8</td>
<td>0.085</td>
<td>0.055</td>
</tr>
</tbody>
</table>

The following table shows the benchmark results for a resolution of 1280 by 800 pixels:

<table>
<thead>
<tr>
<th>no of servers</th>
<th>overall time (s)</th>
<th>render time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.50</td>
<td>0.44</td>
</tr>
<tr>
<td>2</td>
<td>0.285</td>
<td>0.23</td>
</tr>
<tr>
<td>4</td>
<td>0.157</td>
<td>0.11</td>
</tr>
<tr>
<td>6</td>
<td>0.120</td>
<td>0.07</td>
</tr>
<tr>
<td>7</td>
<td>0.110</td>
<td>0.06</td>
</tr>
<tr>
<td>8</td>
<td>0.1</td>
<td>0.058</td>
</tr>
</tbody>
</table>

One can clearly note a saturation effect with higher number of servers. We think that this is due to the limited network performance. The Linux implementation of the PS3 runs on a hypervisor, preventing direct access to the hardware. Because of this, the throughput of the network is limited to about 580 Mbit/s (measured with the netperf TCP stream benchmark). More important, the single network interface of the computer running basho limits the overall throughput to less than 1 Gbit/s.

The overhead of network communication, combining the image and writing it to the frame buffer is nearly constant at about 40 ms.
6. Conclusions and Future Work

We presented a Cell cluster based ray tracing renderer that is used together with a modular VR framework. The VR framework allows for the exchange of renderers by just editing a configuration file. No line of code needs to be rewritten for that and it is not needed to recompile the application.

A general problem with exchanging renderers is that different renderers need different render parameters that are not easily obtainable. Indices of refraction are not included in nearly all CAD models or models that can be found in the Internet.

In earlier experiments [MH05] we have evaluated results from rendering scenes with different renderers intra-frame. Some parts were rendered using point-based rendering (especially those parts far away from the camera, using implicit level of detail mechanisms of point-based rendering), some were rendered on graphics hardware. In addition, some parts were rendered using a ray tracing cluster. While it is technically possible to do so by composing the final image doing depth comparisons, there are some problems with this approach, especially the missing energy exchange between the involved renderers. This a topic of future work.

Acknowledgments

This work was sponsored by the German Federal Ministry of Education and Research (BMBF) under grant no 1762X07. We would like to thank VRmagic GmbH for providing the eye model and IBM Deutschland GmbH for the close cooperation.

References


