DiReC – Distributing the Render Cache to PC-Clusters for Interactive Environments

Nils Beck, André Hinkenjann
Computer Graphics Lab
Bonn-Rhein-Sieg University of Applied Sciences (BRSU)
Grantham-Allee 20, 53754 Sankt Augustin, GERMANY
DiReC@nils.beck.name, Andre.Hinkenjann@fh-brs.de

ABSTRACT
The Render Cache [1, 2] allows the interactive display of very large scenes, rendered with complex global illumination models, by decoupling camera movement from the costly scene sampling process. In this paper, the distributed execution of the individual components of the Render Cache on a PC cluster is shown to be a viable alternative to the shared memory implementation.

As the processing power of an entire node can be dedicated to a single component, more advanced algorithms may be examined. Modular functional units also lead to increased flexibility, useful in research as well as industrial applications.

We introduce a new strategy for view-driven scene sampling, as well as support for multiple camera viewpoints generated from the same cache. Stereo display and a CAVE multi-camera setup have been implemented. The use of the highly portable and inter-operable CORBA networking API simplifies the integration of most existing pixel-based renderers. So far, three renderers (C++ and Java) have been adapted to function within our framework.

Categories and Subject Descriptors: I.3.2 [Computer Graphics]: Distributed/network graphics
Keywords: Clusters, Render Cache.

1. INTRODUCTION
Interactive Environments commonly employ hardware-based renderers to achieve the frame rates necessary for an interactive experience. However, current GPUs still fall short of creating accurate global illumination simulations.

At the same time, software-based high-fidelity renderers usually perform at speeds that remain orders of magnitude below the requirements of interactivity. Ray tracers are theoretically at an advantage in comparison to raster-based GPUs, due to their logarithmic scalability in scene size. But in reality they are only now slowly beginning to be competitive with standard solutions, at medium resolutions and only for very large scenes [3]. In addition, more computationally expensive techniques are often required to obtain the necessary realism in lighting.

The goal of image-based acceleration techniques such as the one presented in this paper is to reach interactive frame rates with existing slow renderers by allowing temporary image degradation during movement and scene changes.

Commodity hardware clusters are often more cost-effective than shared memory systems, but exhibit comparatively low bandwidth and high latency. Our objective was to examine the feasibility of a fully distributed Render Cache and to investigate new setup possibilities in this testbed.

2. PREVIOUS RENDER CACHE WORK
Our work is based on the Render Cache by Walter et al. [1, 2]. Their approach is to decouple the renderer from the user feedback loop, which includes frame generation, display and the resulting user input. This is done by caching renderer results as a 3D point cloud and reprojecting this to the current viewpoint. The underlying assumption is that reprojection, post-processing and display of the point cloud requires less time than re-rendering every pixel of the frame using the renderer.

Of all existing image-based acceleration techniques, only the Render Cache remains unaffected by large scene sizes, as its frame rate is based purely on the size of the point cloud. Therefore, the scalability benefits of ray tracing for very large scenes are not lost. Other systems, such as the Shading Cache by Tole et al. [4] require knowledge of the scene geometry and render it using linearly scaling GPUs. This also makes them less flexible regarding the scene description format used by the renderer.

Reinhard et al. base their Parallel Point Reprojection system [5] on the Render Cache and identify the single reprojection process as a bottleneck. The point cloud size must increase linearly with the image resolution to avoid gaps. Thus, reprojection becomes a limiting factor at higher resolutions. Their solution is to separate the frame into several tiles and to maintain a point cloud for each tile. Each point cloud may then be reprojected using a different processor on their NUMA system.

Finally, Bala et al. [6] improve the Render Cache post-processing step by introducing a new data structure to efficiently detect the edges of scene geometry and shadows. This allows areas without edges to be sampled very coarsely, as applying large-scale interpolation no longer results in loss of detail at edges.
The renderers are distributed to a cluster, but unlike our system, no other components are separated.

3. THE DISTRIBUTED RENDER CACHE

The main decision regarding the DiReC framework design was to split the Render Cache system into all of its functional units. Parts of the system requiring expensive computation can thereby be assigned to their own node, exploiting MIMD parallelism. Components with lower computational requirements may also benefit from this approach, as their functionality and thus their need for processing power can be extended without influencing the performance of components which already cause heavy load. To our knowledge, the DiReC framework is the first Render Cache-based system to distribute other components than the renderers on a PC cluster.

3.1 System Components

Each component of the DiReC system (fig. 1) is implemented as an individual executable CORBA server and may be accessed across the network via its name using a CORBA name service. Since the highly optimized omniORB [7] be present in the system, allowing multi-screen applications (fig. 2). Multiple stereoscopic outputs could provide images for an entire CAVE using the contents of a single cache.

![Diagram of DiReC system](image1)

**Figure 1: DiReC system overview**

CORBA networking code is multithreaded, calculations and communication are parallelized where possible, and performance approaches that of raw sockets. A client application may set up the entire cluster simply by running RenderCacheMain and passing the desired configuration via CORBA. Other activities (e.g. shutdown, camera changes, etc.) are also done by passing messages to RenderCacheMain, thus making it the single point of control.

In the following paragraphs we will briefly describe the components of the DiReC system.

1. Rectangles represent separate distributable executables.

Input

The input provides the framework with all the information necessary to generate the desired output. A cluster has exactly one input. The input may either be part of a stand-alone application or it may be integrated into a renderer plug-in that acts like a single renderer to the client application, replacing the then distributed original renderer.

Output

The output receives the raw reprojected color and depth buffers and processes them. Processing may mean anything from simply writing to a buffer in the client application to performing complex post-processing steps and displaying the result in full-screen mode on a dedicated node.

Because the output is executed independently of the framework, it is possible to create a wrapper that simultaneously acts as input and output to the framework, thus enabling it to implement the full interface of the renderer that is to be replaced in an existing system. A single output may be shared by several projectors, allowing the creation of outputs suitable for stereo display. Multiple outputs may also be present in the system, allowing multi-screen applications (fig. 2). Multiple stereoscopic outputs could provide images for an entire CAVE using the contents of a single cache.

![Diagram of Multiple output setup](image2)

**Figure 2: Multiple output setup (e.g. CAVE)**

RenderCacheMain

In each cluster, a single RenderCacheMain component coordinates configuration, as well as scene and camera data distribution. This is the only part of the system that is exposed to the client application code (i.e. the input). RenderCacheMain is also responsible for execution of the other components of the cluster, as defined in the configuration data passed to it after startup.

The advantage of a single point of control currently far outweighs the risk of this component becoming a bottleneck.

Renderer

Each cluster will contain one or more instances of the renderer component. Each renderer actively polls the sample multiplexer for new jobs and sends results to the cache. The encapsulated original renderer code needs to provide little information to its wrapper: some type of color representation for the rendered sample and either the hit point of the primary ray or depth information (which will generally be sufficient to re-calculate the hit point in the wrapper).

Cache

The cache maintains the most recent samples sent to it by the renderers and provides this point cloud to projectors.
upon request. Aside from occasionally purging outdated samples, it is entirely passive. By separating the cache from the projector, multiple projectors may be present using a single cache. Thus, the available rendering capacity does not have to be split among different viewpoints.

**Projector**

For every viewpoint in the setup, an individual projector instance is responsible for reprojecting the cache contents according to the current camera parameters. The color and depth frames resulting from the reprojection are sent to a pre-specified output instance and also temporarily remain in memory, in case the sampler associated with the projector has completed an iteration and requests the current frame.

**Sampler**

A sampler is associated with each projector. Its purpose is to analyze projected frames and to generate new sample requests that direct the available rendering capacity to the areas most in need of sampling. The generated priority map is adaptively normalized such that the job production rate remains approximately equal to the total job consumption rate of all renderers. New jobs contain a list of selected pixels and the camera used for the analyzed frame\(^2\), and are added to a queue. This task pool approach guarantees automatic load balancing. It is appropriate as jobs are generated dynamically and the cost of each is unknown in advance.

**Sample Multiplexer**

In order to allow multiple simultaneous viewpoints and thus multiple projector-sampler pairs to exist in a cluster, job requests by the renderers must be arbitrated among the samplers. For this purpose, the renderers query the sample (request) multiplexer instead of the samplers. Each viewpoint is assigned a separate sampling priority, making preferred treatment of a main screen possible. Job requests are randomly passed on to one of the samplers taking their priority into account.

### 3.2 Differences to Walter et al.

The decision with the strongest impact on image quality was to refrain from cross-referencing the points in the 3D point cloud and the 2D reprojection in our design. Cross-references would have allowed more precise purging of the cache, which would have resulted in faster reprojection due to the reduced amount of useless samples. Also, this would have allowed premature aging of specific samples.

However, in our distributed setup such information would have to pass over a low-bandwidth network. This was considered to be unnecessary, as old samples are regularly purged once the (variable) cache size has exceeded a user-defined threshold.

Samples in the cache also do not contain any ID of the object they represent, so rigid body transformations cannot be applied to the cache contents to prolong the reuse of samples when the corresponding object has been moved. This would have caused the framework to require information on object IDs from all renderers, making the integration of existing renderers much less flexible.

Our output implementation includes several options for post-processing. The best results were achieved using a 7x7 moving average pre-filter with a 3x3 weighted average filter. Several changes were made to the sampling step. Our system analyzes the raw depth-culled reprojection, rather than the post-processed image. Retrieving the interpolated version from the output component is not practical, as latency between projection and the production of corresponding jobs would be increased and sampling would be based on a less accurate representation of the cache.

Many scenes are not closed to all sides, leading the classic sampling strategy to direct a large part of the rendering resources to areas that will never return any results. Thus, instead of treating areas with no neighbors as highest priority, our sampler uses only a small amount of random sampling to detect possible new or moved objects in "empty" areas. If an object is found, our new 7x7 neighborhood selection (fig. 3) results in a more rapid coverage of the entire shape. Finally, the binary selection decision based on the priority map is not done via diffusion dithering. Instead, for each pixel a random value is compared to the priority map at that position and the pixel selection is based on the outcome.

![Figure 3: Priorities applied to empty pixels around an existing sample.](image)

### 4. RESULTS

Our cluster consists of 8 nodes with dual Intel XEON 2.6 GHz and 2 GB RAM each. All nodes are connected through a fully switched 1000baseT gigabit ethernet network. For our experiments we employed a readily available ray tracer. Higher quality renderers such as path tracing renderers could be used though the increase in the ratio of frame rate to renderer speed would need to be compensated by adding more rendering nodes.

As the system is indifferent to scene complexity, the humble scene (single light source, 14046 triangles) used for our tests should not be used as an indicator of performance.

#### 4.1 Integration into the Basho VE Framework

Basho [8] is a highly modular framework for virtual environments currently under development. It can be extended to handle diverse aspects of virtual environments, such as input device management, interaction events, scene data management and rendering of visual, physical or audible effects. Our system has been integrated into basho by using a wrapper that simultaneously acts as a basho renderer plug-in and as an input component to the DiReC framework. Basho supports the SpaceMouse as input device, which we used for our subjective tests.

The system responds instantly to camera movements by the user and therefore our goal of interactive navigation has been reached. The latency between movement and initial resampling depends on the renderer but in our tests remained below a second. At a resolution of 640x480, the system exhibits frame rates between 7.5 and 11 fps even though...
the components are not yet fully optimized (little communication double buffering, no vector processing or assembly code). At 1024x768, around 3 fps are reached, mostly due to unoptimized reprojection and interpolation code.

To test the effects of the low-bandwidth network, comparisons were made between running individual pairs of components on the same machine and on separate nodes. Network effects are eliminated when processes on the same host communicate, as in this case omniORB uses UNIX sockets. No significant impact on performance was discovered.

4.2 Using Multiple Renderers

Multiple instances of the same renderer may be used to improve the sample rate of slow renderers by exploiting SPMD parallelism. Our system exhibits linear scalability (fig.4) and, given a sufficient number of nodes, allows the system to accelerate even very slow renderers. An increase in the number of renderers results in faster cache updates. This in turn leads to a decrease of the temporal range present in the cache and thus, more accurate lighting during movement (fig.5) and more rapid convergence after movement.

![Figure 4: Linear scalability (max. 5 nodes)](image)

Using multiple caches would evade the problem of mixing different viewpoints, but the points in both views would be unrelated, reducing the 3D effect, thereby causing irritation and fatigue. Also, this would divide the available rendering capacity among the caches.

5. CONCLUSION AND FUTURE WORK

The Render Cache remains a practicable approach in its distributed form, as both the frame rate and the convergence of image quality are acceptable, even in the current unoptimized state.

Algorithm complexity especially in post-processing and sampling poses no problem, as each component can be assigned to a dedicated node. Further optimization of the reprojection and of the time spent waiting for mutexes will doubtlessly lead to a significant performance gain.

It is important to note that with a well-chosen node setup the frame rate of this system is limited only by the speed of the reprojection, which in turn depends on the size of the point cloud. The rest of the system – including network transfers, post-processing and display – allows higher frame rates.

Because of the loose interconnection of components, various workspace setups are possible, and implementations may be exchanged simply by switching executables. The integration with the basho framework and the adaption of three renderers demonstrate the flexibility of our framework, which was designed to support a broad range of interactive applications.

6. REFERENCES


