Walkthroughs with Corrective Texturing

Marc Stamminger Jörg Haber Hartmut Schirmacher Hans-Peter Seidel
Computer Graphics Group
Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
Email: {stamminger, haberj, htschirm, hpseidel}@mpi-sb.mpg.de

Abstract. We present a new hybrid rendering method for interactive walkthroughs in photometrically complex environments. The display process starts from some approximation of the scene rendered at high frame rates using graphics hardware. Additional computation power is used to correct this rendering towards a high quality ray tracing solution during the walkthrough. This is achieved by applying corrective textures to scene objects or entire object groups. These corrective textures contain a sampled representation of the differences between the hardware generated and the high quality solution. By reusing the textures, frame-to-frame coherence is exploited and explicit reprojections of point samples are avoided. Finally, we describe our implementation, which can display interactive walkthroughs of fairly complex scenes including high quality global illumination features.

1 Introduction

Photorealistic rendering has a major drawback: its performance is far from interactive. In contrast, hardware-assisted rendering allows walkthroughs in complex environments at interactive frame rates, but lacks the visual complexity and quality of sophisticated global illumination solutions.

Approaches exist for pushing photorealistic rendering towards interactivity, as well as for increasing the realism of interactive hardware-assisted visualization systems. Both directions of research have advanced drastically over the past few years. Yet it can be foreseen that even with the fast increase of computation power and graphics hardware capabilities, the gap between interactive rendering and global illumination will not vanish in the near future.

In this paper, we propose a way to bridge this gap. A scene is first rendered by means of graphics hardware. This rendering can include global illumination effects, e.g. shadows, but most importantly, it guarantees a certain frame rate. Although this approximate rendering contains all geometric features of the scene (which is important for navigation), it can in general not cover the whole range of lighting effects, as for example multiple reflections and refractions, or complex reflection characteristics. In the following we will refer to this rendering as the *interactive solution*.

More desirable, however, is a *high quality solution*, which is typically obtained by ray tracing-based algorithms. Depending on the employed technique, these high quality solutions exhibit both important and subtle global illumination effects, usually at high computational costs. Clearly, we cannot trace every pixel in an interactive display loop, but we can use additional computation power — in between the display of frames or, if available, on parallel processors — to *correct* the interactive solution towards a photorealistic, view-dependent image.

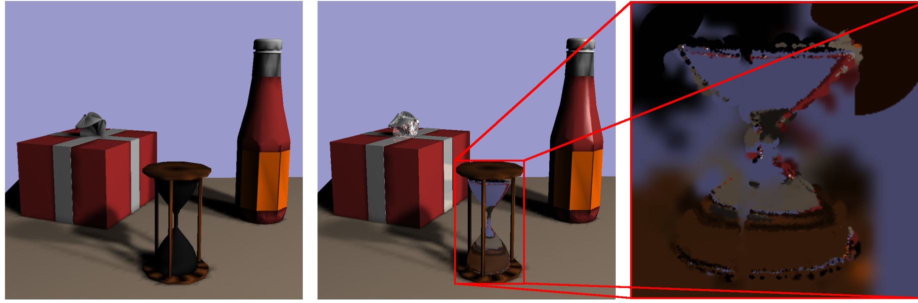

Fig. 1. Corrective texturing example. Left: *interactive solution* that can be displayed at high frame rates. Center: *high quality solution* obtained by applying corrective textures that represent the differences between the interactive solution and the high quality samples. Right: *corrective texture* of the hourglass.

To this end, high quality samples are acquired asynchronously. The resulting error values, i.e. the differences between these samples and the interactive solution, are stored in *corrective textures* which are mapped onto the corresponding object during the interactive display process (cf. Fig. 1).

Using textures has three major advantages. First, corrections are restricted to the object (or cluster) the texture is assigned to. Second, intra-frame coherence is exploited, since the texture resolution can be adapted to match the current sample density from the high quality solution, and a single corrective texel can affect several pixels, using the graphics hardware for texture reprojection and filtering. Finally, inter-frame coherence is also exploited, since the corrections will not change drastically for small camera movements. This is why we can reuse most of the textures, and only need to adapt some of the corrections for each new view point.

2 Previous Work

Allowing interactive exploration of high quality global illumination solutions (so-called walkthroughs) is an important goal in many application areas (e.g. architecture, lighting design, simulation). Two major problems have to be addressed: *geometric* complexity, which is the number of geometric primitives to be processed for a view, and *photometric* complexity, which is the time needed for computing the actual view-dependent appearance of a surface. The latter can be split further into the *local illumination* computation (called *shading*) and the determination of *global illumination* effects such as indirect illumination.

A photometrically simple kind of a walkthrough scene consists of purely diffuse polygonal objects, e.g. the solution of a hierarchical radiosity computation [3]. Since the appearance of the objects does not depend on the viewpoint, this kind of scene can be rendered at high frame rates using off-the-shelf graphics hardware.

Unfortunately, the number of polygons the graphics hardware can render at a certain frame rate is limited. Geometry-based or image-based *level-of-detail (LOD)* techniques can be used to reduce the polygon count. *Impostors* [14, 19, 5, 21], for example, replace a number of distant objects by a single textured polygon. The impostor assignment has to take into account the depth range and apparent size of the objects. LOD techniques generally only take into accound geometric properties and not *photometric* features.

Another way of increasing the frame rate for rendering is given by techniques like *post-rendering 3D warp* [16, 15]. Here the display process generates additional in-between frames by reprojecting from neighbouring images. However, only a relatively small speedup is achieved and a number of warping artifacts are introduced.

A variety of methods exists for computing high quality non-diffuse global illumination effects, see for instance [8] for a good overview. Since most of them perform a large number of view-dependent computations and trace light paths through the scene, these algorithms require computation times which are far from interactive frame rates. Several techniques have been proposed that decouple view-independent (e.g. diffuse) and view-dependent (e.g. specular, glossy) global illumination effects and employ some kind of *multi-pass global illumination* technique [25, 12, 22, 23, 13]. Despite a considerable speedup, most of these methods are still not fast enough for interactive use.

With increasing bandwidth and additional features in todays graphics hardware, more and more high quality shading effects can be computed at interactive frame rates using *multi-pass OpenGL rendering*. This includes different types of shadows, specular and glossy reflections on planar and curved objects, sampled approximations of arbitrary reflection functions, and bump maps and normal maps [20, 17, 6, 10, 2, 26, 1]. Each of theses techniques solves a very special subset from the broad range of lighting effects, and it is not always obvious how to combine them all. Usually a number of constraints are imposed on the scene, e.g. number and shape of light sources and reflectors, distance between objects and the reflected environment, and so on.

Finally, some viewing applications perform *massively parallel computations*, sometimes loosely coupled with a *progressive display process* [18, 29]. Udeshi and Hansen use a combination of hardware-assisted techniques for shadow and indirect lighting, plus ray tracing for specular objects [24]. They exploit both parallel CPU power and parallel graphics pipelines, and reuse results from the hardware-assisted rendering for speeding up the software ray tracing step.

Walter *et al.* [27] describe a fully decoupled display process, the so-called *render cache*, which maintains and reprojects the current set of samples and replaces outdated samples by newer ones from the ray tracing clients using a sophisticated importance-weighted, diffusion-like sampling strategy. The render cache can be used for nearly any kind of per-pixel global illumination computation. The down side of this is that during camera motion, a large fraction of the pixels appear as black holes, as long as no information is available for them from the ray tracing processes. The work presented in this paper was motivated by the results of the render cache. Our goal was to build a similar system, that avoids these warping holes by rendering the scene in hardware with full geometric detail, and adding the lighting obatined by ray tracing.

3 Corrective Texturing

Corrective texturing provides a means for refining an approximate, *interactive* solution with corrective samples obtained from a *high-quality* global illumination computation. In our approach, we maintain a set of textures that represent the difference between those two computations. The corrective textures are updated through an adaptive and lazy sample tracing. The actual display process draws the approximate interactive solution, augmented by the corrective textures.
Our method consist of several collaborating tasks that are sketched in Figure 2:

- the *sample clients* deliver high quality samples using a ray tracing-based approach;

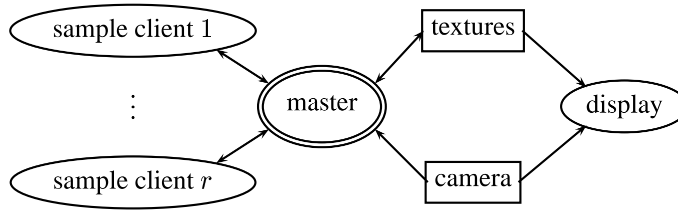

Fig. 2. The main components of our rendering system. The master task keeps track of the camera movements and requests high quality samples from the sample clients. Furthermore, it inserts the resulting samples into the textures which are used by the display task for correcting the approximate solution.

- the *master task* adaptively requests and collects the samples, and assigns, creates, and updates the corrective textures;
- the *display task* renders the hardware-accelerated fast approximation of the scene and applies the corrective textures.

The master task uses an adaptive scheme to request high quality ray samples in image regions where the error of the current scene rendering is probably large. It processes the data returned from the tracing clients and finds the texture assigned to the sample's corresponding scene point. Next, it computes the error of the currently displayed value with respect to the high quality sample and splats the corresponding correction into the texture. The texture resolution is adapted to the chosen sample density, and textures are assigned to scene objects or clusters based on their appearance in the current view. The master also "ages" the texels with respect to the amount of camera movement, so that fresh samples replace older ones.

3.1 Why Textures?

The corrective textures are used to spread corrections across several pixels within the same frame (intra-frame coherence), as well as for reusing the same corrections in further frames until fresher samples are acquired (inter-frame coherence). Our approach of using textures brings along advantages over approaches like the render cache [27], which stores point samples that have to be reprojected and splatted for every frame.

First, the performance of our rendering process is almost independent of the screen resolution due to the use of hardware texture mapping. A single corrective texel can affect several pixels through the use of hardware-assisted texture reprojection and filtering.

Second, we remove the problem of holes in the displayed image caused by undersampling and missing information. The interactive solution visualizes *all* geometric features of the scene, and the corrective textures always map a *dense* set of corrections onto the objects. By using separate textures for different objects, these corrections are restricted to the object boundaries. This guarantees that the corrections will not be mapped onto the wrong object, as it can occur with warping approaches if the visibility situation changes. However, this also requires a well chosen hierarchy for the texture assignment (see below).

Third, we can afford to splat new samples adaptively into their corrective texture and to blend the new samples with their neighbours, resulting in visually pleasant, smooth textures. These relatively expensive operations cannot be employed if the point samples

have to be reprojected and splatted for every new frame as in warping-based display algorithms. Furthermore, the texture resolution can be adapted to match the density of the high quality samples.

3.2 Texture Projection

We use two different modes for applying textures to their associated scene objects, both based on texture projection. Projective texture mapping [20] projects the texture onto the specified scene objects from some virtual point or direction (for perspective or parallel projections, respectively). The projection can either be attached to the object's coordinate system (object-local) or to the current camera system (camera-local).

For flat objects, i.e. objects with only slightly varying normal, we use object-local parallel projection along the surface normal. By this, we have a simple automatic way to parameterize entire object groups (e.g. many triangles in a mesh). Because of the object-local projection, the texture is "glued" onto the object. This means that view independent corrections (e.g. caustics on diffuse surfaces) are reprojected correctly into novel views.

If the textured object is not flat or not even convex, an object-local projection leads to serious problems, since any texel may map to multiple scene points, and surfaces parallel to the projection direction are undersampled.

Instead, for such non-flat objects as well as for object clusters we employ a camera-local texture projection. By placing the center of projection into the current eye point, ambiguity problems are avoided completely. Only visible parts of the scene are assigned corrective samples (which makes the mapping bijective), and one single texture is sufficient per object. We define our texture mapping by projecting a bounding volume of the object into camera space and then using its bounding rectangle on the image plane as the valid texture coordinate domain. This is a sheared perspective transformation which can be applied by the graphics hardware. Furthermore, for camera-local textures we can easily adjust the texture sampling rate to match that of the image.

Our texturing approach shares some ideas with image impostor methods [14, 19], but instead of replacing an object cluster by a single textured polygon, we augment the original geometry and appearance by a *photometric* correction texture. Note that the two approaches fit together well, allowing photometric corrections on simplified geometry.

The drawback of camera-local projection is that it changes with the view point, which means that the texture floats in front of the object rather than sticking onto it. If the object is concave, the correspondence between texel and surface point can even be discontinuous. This is weakened by the fact that typical camera movements in a walkthrough application only lead to a very small texture flow. Furthermore, this kind of artifact mainly affects clusters close to the viewer. Such clusters, however, tend to be split and textured on a lower level (see next Section 3.3).

3.3 Texture Assignment

Textures are not only assigned to single geometric primitives, but also to composite scene objects and object clusters, e.g. provided by spatial subdivision schemes and the scene's modeling hierarchy.

The assignment of textures to scene nodes is adapted on the fly for each novel view point by traversing the scene hierarchy top-down and verifying the validity of the current allocation. The observed motion of a point depends on that point's distance to

the camera, so different parts of the object move in a different way (parallax effect). Using a single image-plane texture would cause the texture to float on the geometry.

To avoid this artifact, we estimate the parallax effect by computing the depth range Δr of the textured object in relation to its distance d to the viewer. An object is assigned a single texture if $\Delta r/d < \epsilon$, with ϵ being a parallax threshold. In our tests, values of $\epsilon = 0.2 \ldots 0.5$ led to parallax distortions that were hardly visible. The threshold is used to control the number of allocated textures, which is reduced further by simple visibility frustum culling. Figure 5 depicts the textures assigned to various objects on a table, with a constant threshold of $\epsilon = 0.3$.

Also the projected size of the object can be included into this test. In this way, also for a large object with no parallax artifacts several textures will be assigned. If the sample density is strongly varying over the object, the texture resolutions can thus be selected separately for different parts of the object (see below).

Building a proper hierarchy for this process is a difficult problem on its own, which has a large impact on the quality of the results. We use the modeling hierarchy for this purpose or a modified hierarchical bounding volume method that does not only consider spatial criteria, but also material properties.

3.4 Sample Insertion

Every texture contains color correction values ΔL_i which approximate the difference between the high quality solution L_i and the basic interactive solution L_i': $\Delta L_i \approx L_i - L_i'$. When a new sample is to be inserted, it should affect a certain texture area rather than a single texel, because we assume some amount of spatial coherence of the approximation error. Ideally, the sampling density determines how far the correction is spread. Therefore we restrict the splat region of a sample to its cell in a Voronoi diagram of the sample points on the texture.

This can be achieved rather easily by an approach similar to the discrete Dirichlet domain creation described in [11]. Every texel i is assigned a validity value v_i, which is basically the *negative* distance of the texel to the sample that is responsible for the texel's value. If we want to insert a new sample, the corresponding texel is inserted with maximum validity 0. Then neighbored texels j with distance d are examined. If the validity of j is smaller than $-d$, the texel gets replaced with the new, more appropriate sample and new validity $-d$. This is continued for growing distances d until for one neighborhood ring no texel could be updated. We call the radius of the last updated ring *splat radius* r. Blending between new and old values according to their respective validity smoothes out harsh boundaries between the splats. Figure 3 shows an example obtained by splatting 20 and 100 samples into a texture.

3.5 Texel Aging

In addition to this purely distance-based splatting criterion, the age of the existing texels has to be taken into account. To avoid updating every texel for each new view point, every texture has one current global maximum validity value $v_{\max}$. The validity/distance values for inserting new texels start with $v_{\max}$ rather than 0 at the sample's center texel, so that newer texels gain priority over the rest of the texture.

If the camera changes, $v_{\max}$ is modified accordingly. The offset reflects both the texture flow appearing for projected textures and an estimated possible change of the correction due to the new viewing directions. The first is zero for object-local projections. For floating projections, it is estimated by the ratio of the visual depth over the distance to the viewer. The latter is estimated by the maximum change of the object's

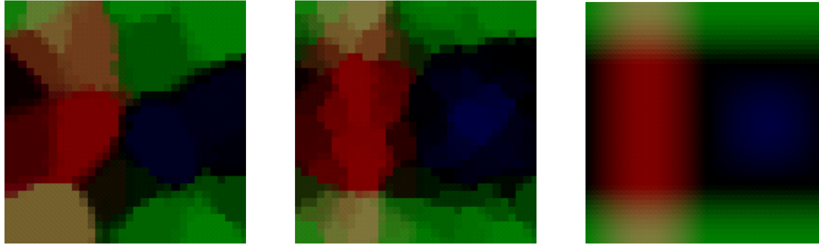

Fig. 3. Sample insertion: Splatting 20 (left) and 100 (center) samples into a texture using our validity measure. Right: reference solution containing exactly one sample per texel.

BRDFs. We use heuristics for combining these two. This way glossy surfaces age quickly, whereas textures on diffuse objects with fixed textures do not age at all, and the samples remain valid for all possible new view points.

3.6 Texture Resolution

The average splat radius of the last samples inserted into a texture is a valuable hint about the relation between sample density and texture resolution. If the radius is large, the texture resolution is probably too high compared to the number of samples. As a result, inserting a single sample is expensive. On the other hand, a very small average splat radius indicates that the texture resolution is too low, so information gets lost if two samples share one texel.

After each sample insertion, we test the average of previous splat radii against a lower and an upper threshold value. If one threshold is exceeded, the resolution of the texture is increased or decreased by a factor of two. This way, the initially coarse textures become finer the higher the density of the received samples is. If the splat size increases again, for example after aging (cf. Sec. 3.4), this indicates that there is no high frequency detail in the texture, and so the resolution is decreased again to speed up the splatting process. For those textures mapped into the image plane directly (cf. Sec. 3.2), the maximal texture resolution can be derived from twice the sampling rate of the the screen. By using bilinear interpolation for texture refinement and box filtering for texture coarsening a smooth transition between the resolution levels is obtained.

3.7 Adaptive Sample Acquisition

For acquiring the necessary high quality samples at a reasonable rate, it is imperative to use an adaptive scheme that focuses on image regions with large error. It is important to account for the present approximate correction value ΔL_i. Therefore we measure the error $E_i := L_i - (L'_i + \Delta L_i)$, which is the difference of the high quality sample L_i and the value of the currently displayed and corrected pixel $L'_i + \Delta L_i$. Note that in this step we have to check whether the object visible through that pixel is the same in the interactive and in the high-quality solution. In order to converge against the desired solution, the sampling scheme must make sure to cover all pixels of the image if the user stands still long enough.

The heart of the master process is a priority queue that contains sampling requests, sorted according to their anticipated importance for an improved solution. Each request L_i in this queue contains a sampling direction, a *domain radius* s_i, which denotes the size of the image domain this request represents, and its priority value.

The queue is initially filled with several requests of high priority which are uniformly distributed over the image. The domain radius of these requests is given by their average distance. Then the master iteratively removes the sampling request with the highest priority from the queue and sends it to the sample tracer(s). The new correction value ΔL_i gets splatted into the appropriate texture.

After request L_i has been processed completely, we generate a number of N child requests that cover the affected domain more densely. The domain radius s_c of such a next-generation request c is set to $s_i/\sqrt{N}$, since the domain area is inversely proportional to the number of requests. The product of the parent's correction error E_i and the new radius s_c yields the new priority for request c. If this priority is too small, the request is discarded, otherwise it is inserted into the priority queue.

For each frame, a number n_{sig} of the most *significant* samples are memorized separately, where the sample's significance is determined by its correction error E_i alone. Each time the camera changes, these significant samples are reprojected into the novel view and added to the priority queue as requests (cf. Fig. 4) with high priority. Thus the sampling is directed into regions exhibiting large errors in the previous frame.

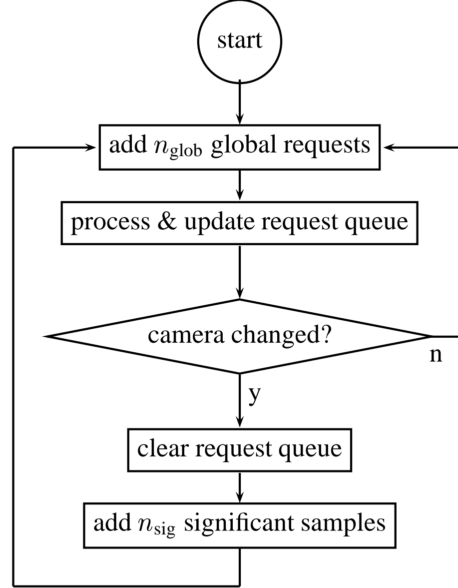

Fig. 4.: The sampling process adds global requests and processes the request queue as long as the user stands still. If the camera moves, some significant samples from the previous frame are reused to yield a faster update in visually important regions.

If the queue has been processed completely, new randomly positioned global requests with high priorities are inserted with the goal to detect new features that require correction.

If the camera stands still long enough, every pixel of the image finally corresponds to the value of the high-quality solution. We use a mask of sampled pixel positions to immediately discard new requests belonging to already sampled pixels. So after a region is sampled at image resolution, the sampling also spreads to less erroneous parts of the image. If most pixels of the image are covered, the sampling process creates a special queue containing the remaining pixels, thereby avoiding to spend too much time on finding free pixel positions for the last few requests.

4 Implementation

4.1 Interactive and High Quality Solutions

In our implementation we display the result of a hierarchical radiosity preprocess as the interactive solution, which can be rendered at high frame rates by the graphics hardware. It contains soft shadows, but completely lacks effects like glossy highlights, mirroring, and refraction.

For the high quality samples, we use standard distributed ray tracing [4], i.e. at non-diffuse surfaces an eye ray spawns several reflection rays which are traced recursively

up to a user defined depth. Illumination due to area light sources is computed by casting stochastic shadow rays.

For each high quality sample, the difference to the interactive solution is required. Unfortunately, reading back the frame buffer content is still a fairly expensive operation on almost all current graphics platforms. To avoid this step, we compute the difference value for each high quality sample by performing the Gouraud interpolation of the radiosity value in software.

Using the above setting reveals an interesting effect. Because we use Monte-Carlo sampling for the area light sources in the high quality solution, the penumbra regions are noisy and mostly less accurate than in the radiosity solution. Especially the noise in the "reference" solution imposes severe problems, because it is considered as fine detail in the illumination. As a result, many high quality samples are wasted to capture this supposed detail. Furthermore, the splatting of the noisy corrections produces visually unsatisfactory results, unless the entire penumbra is sampled densely.

The first solution is to increase the number of shadow rays, which works well, but has to be bought by more expensive high quality samples. For performance critical scenes, we solve the problem differently: The high quality ray tracing step itself reuses the diffuse components computed in the radiosity precomputation and only computes specular reflection additionally. This implies that the radiosity solution is precise enough. Furthermore, with this combination it can be guaranteed that all corrections are positive, which speeds up hardware rendering significantly on some platforms (see Sec. 4.3).

4.2 Parallelization

We have tested our implementation of the components sketched in Figure 2 on several different machines running under either IRIX or Solaris. In our computing environment, we use a single-processor `sgi` Octane (MIPS R12000) with hardware-supported texture mapping (EMXI graphics board) as the display process host. The master process and the ray tracing processes run on either a cluster of `sgi` workstations connected to a common 100 MBit-Ethernet, or on a Sun Enterprise 10000 server with 8 available processors and shared-memory communication. The communication between the master process and the ray tracers is implemented using the Message-Passing Interface [9] for both the cluster and the shared-memory approach. Due to the lack of interoperability between different vendor implementations of MPI, we use a standard UNIX protocol and sockets for the communication between the master and the display process. This communication is mostly unidirectional: only upon camera movement, a message containing the new camera parameters has to be sent from the display process to the master. The master process, on the other hand, sends all the ray traced samples along with their texture coordinates to the display process.

4.3 Texturing

The corrective texturing has been implemented through the basic blending functions provided by OpenGL. More specifically, we use an additional blending equation provided by the `blend_subtract` OpenGL extension. This approach requires the scene to be drawn three times: first with the original color and texture, a second time for adding *positive* correction values, and a third time for *subtracting* corrections where the approximation appears too bright. While this method runs on all contemporary `sgi` machines, it has of course the drawback of requiring three rendering passes of the scene. If the interactive solution and the high quality solution are selected appropriately (see

above), all corrections will be positive, so the third pass can be omitted. Some newly available graphics boards such as the NVIDIA GeForce system support multi-texturing with general combiner functions. In that case the entire rendering, including additive and subtractive correction, only requires a single rendering pass, and should allow considerably higher frame rates.

5 Results

To test our implementation, we have created several scenes of different geometric complexity. The number of geometric primitives (i.e. the objects our ray tracer can handle as entities) in our test scenes varies between 3,500 and almost 60,000. The corresponding number of triangles generated by our hierarchical radiosity algorithm lies between 15,000 and 415,000. All our test scenes contain several photometrically complex materials with properties such as anisotropic reflection [28], perfect mirroring, and refractive transmission.

We displayed the scenes on a single processor sgi Octane R12000 with EMXI graphics board at a resolution of 1024×1024 pixels. The master process and the sample clients ran on a remote Sun Enterprise 10000 with 8 processors available (cf. Sec. 4.2). Our ray tracer uses BSP trees [7] for spatial subdivision and distributed ray tracing [4] for light rays and reflection rays.

Using all of the 8 processors on the Sun we were able to move through our scenes at a frame rate of 1.5–5 fps. The lower frame rate is achieved for our most complex scene with 4,000 samples being traced and splatted within each frame, while the higher frame rate corresponds to 1,000 samples per frame and our simplest scene. As soon as the camera stopped moving, the displayed images started converging at a frame rate of 2–6.5 fps.

The percentage of time for rendering, sampling and texture update (splatting the samples, refining and coarsening) within each frame depends heavily on the complexity of the scene and the number of ray traced samples. As long as the geometric complexity of the scene is not too high, the balance of rendering : sampling : texture update is about 30:60:10 for 1,000 samples and 10:70:20 for 4,000 samples. For our most complex scene we obtained a balance of 60:30:10 for 1,000 samples and 35:50:15 for 4,000 samples.

The convergence of our method is visualized in Figure 6. Starting from a radiosity solution, more and more samples are traced and added into the corrective textures. The last image in this series is visually almost identical to a fully converged image. Note that our ray tracer can handle shadows of transparent objects and therefore brightens the dark shadow of the glass from the radiosity solution.

In Figure 7, the same scene is first rendered from a a particular point of view. Then the view changes without updating the corrective textures. The next image has been generated after tracing 10,000 new samples. The distribution of these new samples is visualized in the last image. One can clearly see that most of the samples concentrate on the image regions where the error is large.

During an interactive session, the displayed solution can suffer from texture update artifacts. Especially corrective textures that change rapidly with a new view point (for instance on mirroring or refractive objects) need to be updated with quite a few samples. If not enough time is available for acquiring these samples, flickering in the textures becomes visible. Similar effects appear if small lighting details are missed by the initial samples. Such effects then suddenly pop up later on, when they are hit by accident.

6 Conclusions and Future Work

We presented a technique for augmenting hardware-accelerated rendering by results obtained from expensive pixel-based ray tracing methods. The corrections are applied to the objects through textures that are continuously adapted during a walkthrough. By this, we always have full geometric detail, whereas the richness of lighting depends on the available additional computation power. We thus blend between the interactive and the high quality solution. In the worst case, the user sees the interactive solution, but the more computation power is available, the closer we get to the desired high quality solution.

With our resulting system we achieve good frame rates, but if not enough computation power is available and the sampling is too coarse, the lighting still changes visibly. After rapid view changes, it takes about one second until the lighting converged visually, so that later corrections are still noticable, but subtle.

Our method is in the spirit of the render cache or the Holodeck. However, we additionally exploit graphics hardware, and only spend additional computation power where the hardware-based solution is erroneous. Working with this approach revealed several benefits, but also some limitations.

One major feature is that we can incorporate arbitrary interactive lighting algorithms and exploit the high pixel fill rates from graphics hardware. In contrast to previous approaches, we removed the need of reprojecting samples and filling holes manually. From this point of view, our method is well suited for high screen resolution as long as the graphics hardware supplies sufficiently high frame rates. On the down side, for some interactive rendering methods our approach forces a frame buffer read-back, which is expensive for technical reasons.

In our implementation, we avoid reading the framebuffer by recomputing the framebuffer values for desired samples on the fly. For more sophisticated interactive rendering methods this is probably not feasible. For complex test scenes with about half a million polygons, the interactive rendering time becomes the limiting factor in our implementation. However, this limit could be pushed a lot using more recent graphics hardware (fewer rendering passes), and an improved rendering method (e.g. visibility/occlusion culling). We also did not yet integrate more elaborated interactive rendering techniques, e.g. geometric level-of-detail methods, or multi-pass OpenGL rendering for special lighting effects.

References

1. R. Bastos, K. Hoff, W. Wynn, and A. Lastra. Increased photorealism for Interactive Architectural Walkthroughs. In *1999 Symp. Interactive 3D Graphics*, pages 182–190, April 1999.
2. B. Cabral, M. Olano, and P. Nemec. Reflection Space Image Based Rendering. In *Computer Graphics (Proc. SIGGRAPH '99)*, pages 165–170, August 1999.
3. M. F. Cohen and J. R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press, London, 1993.
4. R. L. Cook, T. Porter, and L. Carpenter. Distributed Ray Tracing. In *Computer Graphics (Proc. SIGGRAPH '84)*, pages 137–145, July 1984.
5. X. Decoret, G. Schaufler, F. X. Sillion, and J. Dorsey. Multi-Layered Impostors for Accelerated Rendering. In *Computer Graphics Forum (Proc. Eurographics '99)*, volume 18, pages C61–C72, September 1999.
6. P. J. Diefenbach and N. I. Badler. Multi-pass Pipeline Rendering: Realism For Dynamic Environments. In *1997 Symp. Interactive 3D Graphics*, pages 59–70, April 1997.

7. H. Fuchs, Z. M. Kedem, and B. F. Naylor. On Visible Surface Generation by a priori Tree Structures. In *Computer Graphics (Proc. SIGGRAPH '80)*, pages 124–133, July 1980.
8. A. S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers, San Francisco, CA, 1995.
9. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI — Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, MA, 1994.
10. W. Heidrich and H.-P. Seidel. Realistic, Hardware-accelerated Shading and Lighting. In *Computer Graphics (Proc. SIGGRAPH '99)*, pages 171–178, August 1999.
11. K. E. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. In *Computer Graphics (Proc. SIGGRAPH '99)*, pages 277–286, August 1999.
12. D. S. Immel, M. F. Cohen, and D. P. Greenberg. A Radiosity Method for Non-Diffuse Environments. In *Computer Graphics (Proc. SIGGRAPH '86)*, pages 133–142, August 1986.
13. D. Lischinski and A. Rappoport. Image-based rendering for non-diffuse synthetic scenes. In *Eurographics Rendering Workshop 1998*, pages 301–314. Springer Wien, June.
14. P. W. C. Maciel and P. Shirley. Visual Navigation of Large Environments Using Textured Clusters. In *1995 Symp. Interactive 3D Graphics*, pages 95–102, April 1995.
15. Y. Mann and D. Cohen-Or. Selective pixel transmission for navigating in remote virtual environments. volume 16, pages 201–206, 1997.
16. W. R. Mark, L. McMillan, and G. Bishop. Post-Rendering 3D Warping. In *1997 Symp. Interactive 3D Graphics*, pages 7–16, April 1997.
17. E. Ofek and A. Rappoport. Interactive Reflections on Curved Objects. In *Computer Graphics (Proc. SIGGRAPH '98)*, pages 333–342, July 1998.
18. S. Parker, W. Martin, P.-P. J. Sloan, P. Shirley, B. Smits, and C. D. Hansen. Interactive Ray Tracing. In *1999 Symp. Interactive 3D Graphics*, pages 119–126, April 1999.
19. G. Schaufler. Dynamically Generated Impostors. In *Proc. GI Workshop MVD'95*, pages 129–136, November 1995.
20. M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast Shadows and Lighting Effects Using Texture Mapping. In *Computer Graphics (Proc. SIGGRAPH '92)*, pages 249–252, July 1992.
21. J. W. Shade, D. Lischinski, D. H. Salesin, T. DeRose, and J. Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. In *Computer Graphics (Proc. SIGGRAPH '96)*, pages 75–82, August 1996.
22. F. X. Sillion, J. R. Arvo, S. H. Westin, and D. P. Greenberg. A Global Illumination Solution for General Reflectance Ditributions. In *Computer Graphics (Proc. SIGGRAPH '91)*, pages 187–196, July 1991.
23. W. Stürzlinger and R. Bastos. Interactive Rendering of Globally Illuminated Glossy Scenes. In *Proc. 8th EG Rendering Workshop*, pages 93–102, 1997.
24. T. Udeshi and C. D. Hansen. Towards Interactive, Photorealistic Rendering of Indoor Scenes: A Hybrid Approach. In *Proc. 10th EG Rendering Workshop*, pages 63–76, June 1999.
25. J. R. Wallace, M. F. Cohen, and D. P. Greenberg. A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods. In *Computer Graphics (Proc. SIGGRAPH '87)*, pages 311–320, July 1987.
26. B. Walter, G. Alppay, E. P. F. Lafortune, S. Fernandez, and D. P. Greenberg. Fitting Virtual Lights for Non-Diffuse Walkthroughs. In *Computer Graphics (Proc. SIGGRAPH '97)*, pages 45–48, August 1997.
27. B. Walter, G. Drettakis, and S. Parker. Interactive Rendering using the Render Cache. In *Proc. 10th EG Rendering Workshop*, pages 27–38, June 1999.
28. G. J. Ward. Measuring and Modeling Anisotropic Reflection. In *Computer Graphics (Proc. SIGGRAPH '92)*, pages 265–272, July 1992.
29. G. Ward Larson. The Holodeck: A Parallel Ray-Caching Rendering System. In *Proc. 2nd EG Workshop on Parallel Graphics and Visualization*, pages 17–30, September 1998.

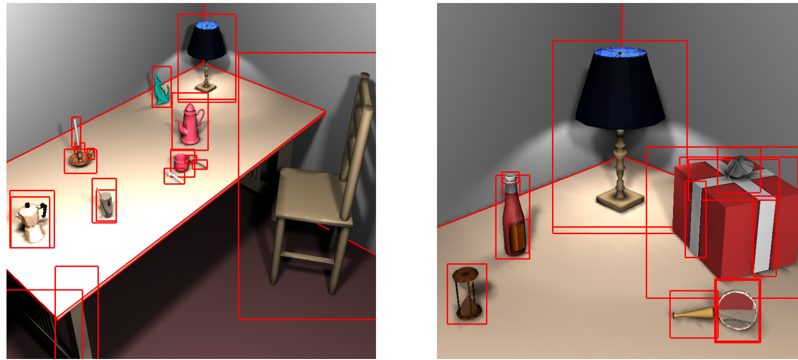

Fig. 5. Adaptive texture assignment according to the relative depth range of scene objects according to the depth range of the objects.

Fig. 6. A radiosity solution (left) is corrected towards a ray tracing solution using 5,000 (0.48 %), and 20,000 (1.9 %) ray tracing samples (center and right).

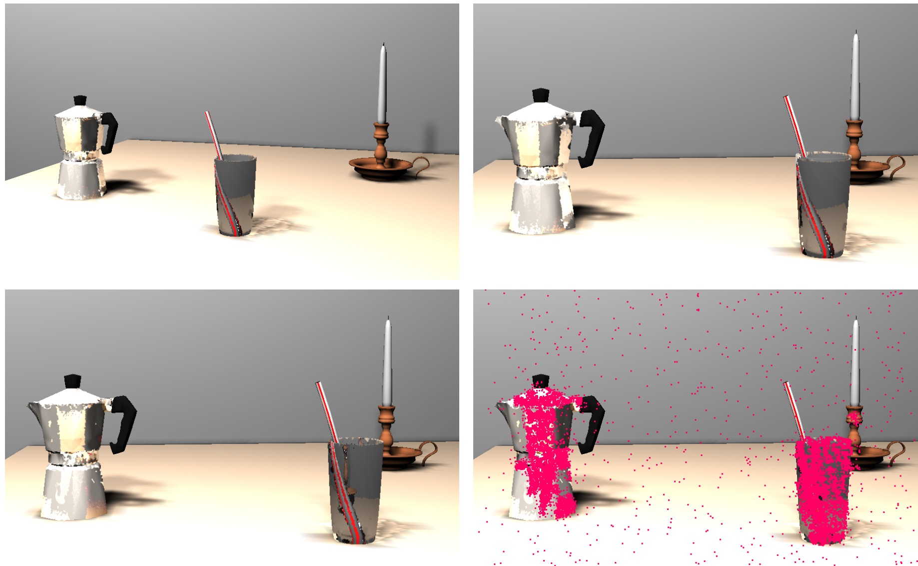

Fig. 7. Exploitation of frame-to-frame coherence: The top left view is obtained with corrective textures. By moving the camera without updating the textures, the top right image is obtained. The bottom left image shows the same view after shooting 10,000 samples (0.95 % of all pixels), the distribution of which is visualized in the bottom right image.