# Light Field Propagation and Rendering on the GPU

Jesper Mortensen[1] Pankaj Khanna[1] Mel Slater[1,2]

[1]Department of Computer Science, University College London, London, UK
[2]ICREA-Universitat Politècnica de Catalunya, Department de LSI, Barcelona, Spain
{j.mortensen | p.khanna | m.slater}@cs.ucl.ac.uk

## Categories and Subject Descriptors
I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Shading*

## Keywords
Global Illumination, Light-fields, Rendering, Graphics Hardware

## ABSTRACT
This paper describes an algorithm that provides fast propagation and real-time walkthrough for globally illuminated synthetic scenes. A type of light field data structure is used for propagating radiance outward from emitters through the scene, accounting for any kind of $L(S|D)^*$ light path. The light field employed is constructed by choosing a regular point subdivision over a hemisphere, to give a set of directions, and then corresponding to each direction there is a rectangular grid of parallel rays. Each rectangular grid of rays is further subdivided into rectangular tiles, such that each tile references a sequence of 2D images containing outgoing radiances of surfaces intersected by the rays in that tile. We present a novel propagation algorithm running entirely on the Graphics Processing Unit (GPU). It is incremental in that it can resolve visibility along a set of parallel rays in $O(n)$ time and can produce a light field for a moderately complex scene - with complex illumination stored in millions of elements - in minutes and for simpler scenes in seconds. It is approximate but gracefully converges to a correct solution as verified by comparing images with path traced counterparts. We show how to render globally lit images directly from the GPU data structure without CPU involvement at real-time frame rates and high resolutions.

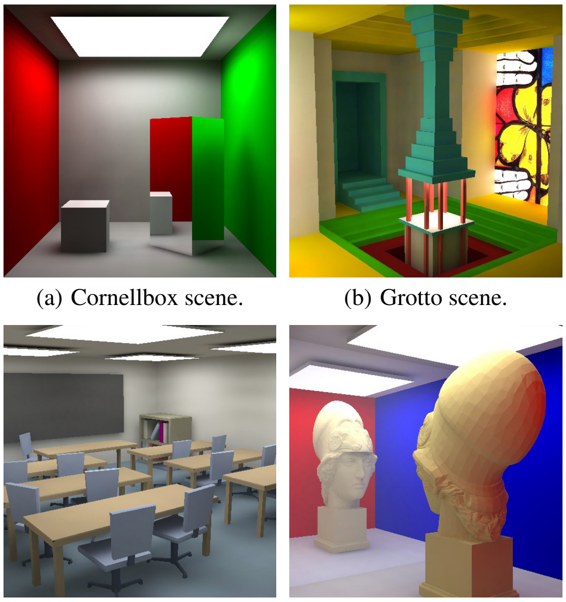

(a) Cornellbox scene. (b) Grotto scene.
(c) Classroom scene. (d) Atenea scene.

**Figure 1: Our system can propagate light through scenes with arbitrary BRDFs and millions of ir/radiance elements in seconds 1(a) to minutes 1(d) and render them at frame-rates exceeding 120fps at 1024×768 screen resolution.**

## 1. INTRODUCTION
A virtual light field (VLF) approach to global illumination was introduced in [37]. The VLF provides a solution to the problem of real-time walkthrough in scenes that are globally illuminated with ideal diffuse and specular surfaces (and mixtures of these). The method exploits the idea of light fields [22] (or lumigraphs [15]), though the particular type of light field representation used is similar to that in [2] and also similar to a data structure used for visibility culling in [6]. It exploits Layered Depth Images [35] where each ray in the light field maintains radiance information for each surface it intersects rather than just the first one. In this way a projected image can be reconstructed from any viewpoint and direction.

The method presented in [37] offered a proof of concept, but was nevertheless impractical, since even very small scenes with hundreds of polygons could result in propagation of energy through the scene taking many days. Also, rendering could only reach real-time rates at modest resolutions. In this paper we recast the algorithm to one using point sampling, with partly pre-computed visibility and show how to exploit the GPU in order to accomplish propagation in a few minutes for moderately complex scenes (∼10K polygons) with complex illumination with millions of diffuse and non-diffuse elements. Additionally, we show how to render directly from this GPU light field data structure without CPU involvement at real-time frame rates greatly exceeding those achieved in [37].

In the next Section we discuss further background information, including a brief recap of the main concepts of the VLF approach to global illumination. In Sections 3 and 4 the GPU propagation approach and the GPU rendering method are explained. Implementation details are given in Section 5, and results are presented in Section 6, followed by conclusions and directions for future work in Section 7.

## 2. BACKGROUND

The performance and features of consumer graphics hardware – especially those of higher-end GPUs has opened many avenues to achieve fast walkthrough and propagation for global illumination. This has been aided by the availability of high-level GPU programming languages like Cg[24] and GLSL[18]. A general review of approaches for global illumination and real-time walkthrough was presented in [37]. Here we briefly discuss approaches that specifically employ GPUs for global illumination computation.

Several methods have been presented to use the GPU as a basic ray-intersection engine [3, 31, 30]. These and other GPU based ray-tracing methods have until recently been limited by the GPU's inability to efficiently represent traditional spatial hierarchies for intersection acceleration. The work of Carr et. al. [5] presents an effective GPU based method using a threaded bounding volume hierarchy for dynamic scenes. More efficient representations have been proposed in [13, 36].

Radiosity has been realised on the GPU by [4] in an approach that attempts to solve the radiosity matrix. The method described in [8] provides an alternative approach using stereographic projections on the GPU albeit with several visual artifacts.

The GPU based ray-tracing method of [31] has been extended to photon map rendering in [32]. Another approach for photon mapping has been presented in [23]. Both of these methods are again limited by the data structures that can be effectively realised and therefore rely on regular grid-structures and hash-tables. [21] uses a CPU-GPU method using the GPU during image generation to perform final-gather and filtering from a CPU computed photon map. Another GPU based final-gather is presented in [16], this time gathering with parallel ray-bundles [39].

A non-diffuse global illumination solution is provided by [27]. The approach consists of projecting the pre-computed radiance function at a set of sample points into spherical harmonic bases. Radiance at any point is subsequently obtained by interpolating the nearest sample points. The method however has some limitations including that of positioning sample points.

A GPU-based method for computing caustics interactively was presented in [43]. In this approach, a set of sample points selected on glossy surfaces are treated as pin-hole cameras that project incident radiance onto diffuse surfaces with some speed-quality trade offs.

In [14] a radiance caching scheme with significant performance improvements over CPU-only schemes is proposed. Radiance records are computed at required scene points and combined on the image plane using weighted splats and radiance gradients instead of traditional nearest-neighbour queries. The method uses efficient data structures on the GPU which necessitate several CPU-GPU data transfers/updates creating speed and synchronisation bottlenecks.

Vivanloc et. al. in [41] present a CPU-GPU approximation to global illumination by storing emitted photon radiance in an octree data-structure which are clustered and represented by virtual directional lights (VDL). Though rendering is interactive, the VDLs do not cast shadows and there are several artifacts. Similar to their use of texture atlases, our approach also reduces texture fetch costs by such combined representation.

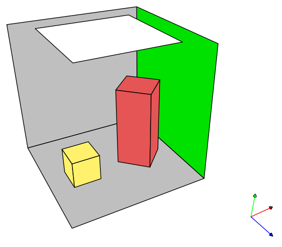
In the hardware accelerated multipath method of [40], radiance is transferred between patches by bundles of parallel rays along a set of global directions in a method similar to ours in several ways. However, our approach differs significantly since it is implemented fully on the GPU and computes a light field for a scene thus storing non-diffuse outgoing radiance for real-time walkthrough. In [33] global lines are used for fast visibility queries to compute global illumination partly using the GPU. Frame-rates are, at best, interactive and the approach does not support high-frequency lighting such as reflections.

We use depth sorted polygon sequences along the global lines thus partly pre-computing visibility. Our approach employs a modified Newell sorting algorithm [26]. Visibility orderings have also been proposed in [10, 44, 38] and using a GPU in [25]. Everitt [12] proposed a GPU depth sorting algorithm for order independent transparency that with slight modifications could be used in our system. The approach, however, requires that the scene is rendered multiple times.

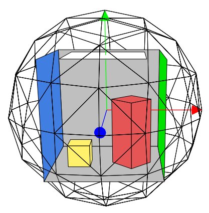
Light-field rendering on the GPU was suggested by [42], with fast rendering via visibility culling and data binning and reuse. In our approach, we store and propagate directional radiance with a DPP light-field[1] also used for final rendering. This is similar to the rendering method described in [11] although we do not require the additional blending and texture-warping passes. We instead render entirely on the GPU from the DPP light-field, efficiently fetching and interpolating radiance values on a per-pixel basis as determined by the view-camera.

### 2.1 VLF Notation

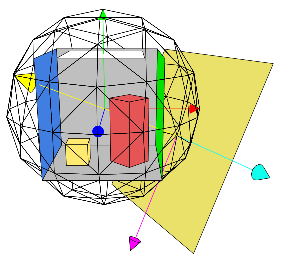
Before presenting our GPU based approach we briefly recap the method and some essential notation introduced in the original VLF method [37]. Given a scene in world coordinates (WC) we first apply a transformation that translates and uniformly scales the scene such that it is enclosed by the unit sphere centred at the origin; we refer to this as the VLF coordinate system (see Figure 2(b)). If $l$ points with spherical coordinates $\omega_i = (\theta_i, \phi_i)$ are chosen on the unit sphere, each serve as a normal for a plane orthogonal to $\omega_i$ large enough to enclose the projection of the unit sphere. The points $(\omega_0, \omega_1, \ldots, \omega_{l-1})$ are shown as the vertices of the spherical mesh enclosing the scene in Figure 2. Each such unique plane $i$ is discretised into a regular grid of $N \times N$ cells, each of which is the origin of a ray parallel to $\omega_i$, we refer to such set of $N \times N$ parallel rays as a *parallel subfield* (see Figure 2(c)). Each $PSF_i$ has an associated rotation that aligns the PSF coordinate frame $(\overrightarrow{u}, \overrightarrow{v}, \overrightarrow{n})$ with $(X, Y, Z)$, this is the canonical PSF representation (see Figure 2(d)). In the canonical PSF representation the ray origin is given by the tuple $(x, z)$ describing a unique ray parallel to the Y-axis and $y$ gives a point along this ray. In practice we work with a *hemisphere* of *bi-directional* directions such that $PSF_\omega$ accounts for directions $\omega$ and $-\omega$. With this discretisation the need to transform points between coordinate frames occurs frequently and must be dealt with accurately and quickly. To this end we store a number of transformation matrices that can achieve this. One such matrix



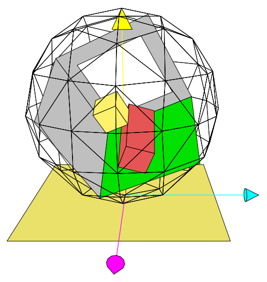
(a) World coordinate system.



(b) VLF coordinate system.



(c) A parallel subfield.



(d) canonical PSF coordinate system.

**Figure 2: Coordinate systems.**

is $M_{WC \rightarrow PSF_i}$ and its inverse $M_{WC \leftarrow PSF_i}$ that can transform points between $WC$ and $PSF_i$.

Now consider any ray in the canonical PSF. This will intersect a number of surfaces in the scene. If we parameterise the ray in the form $r(t) = r_0 + kt$, $t \geq 0$, and $r_0$ is the ray origin, then the intersection points can be characterised as an array of non-decreasing parametric values $[t_1, t_2, ..., t_n]$. With each one of these intersection points additional information could be stored: the identifier of the surface at that intersection, and eventually the outgoing radiance from the surface at that intersection point. Although this approach is possible, no use would have been made of the great coherence between neighbouring rays, and the memory costs would be substantial. Instead, the grid of cells in a PSF is subdivided into tiles, each at a resolution $m \times m$ cells, where $1 \leq m \leq N$ and $\frac{N}{m}$ is integral. Each tile maintains a sequence of surface identifiers with associated radiance maps for faces that are intersected by any ray which has its origin within the tile. This is illustrated in Figure 3. Thus a radiance value can be retrieved from this parametrisation using the following notation $L_s(\omega, s, t, u, v, p)$, where $\omega$ indicates $PSF_\omega$, $(s, t)$ is a tile for face $p$, and $(u, v)$ is a cell within this tile.

Once this data structure is built, propagation is in principle straightforward. Radiance is emitted from light sources following the paths provided by fixed bundles of parallel rays in the PSFs, which are used as approximations for true ray directions. Coherence is exploited by following parallel bundles of rays rather than dealing with individual rays.

Once energy has been propagated, rendering can be achieved with a variety of methods (i) direct rendering from the information stored in the VLF data structure, and using nearest-neighbour interpolation to approximate radiance along specific rays from rays available in the database (ii) using a version of backwards ray tracing until the first diffuse surface is hit (iii) a progressive refinement method that combines the above two methods.

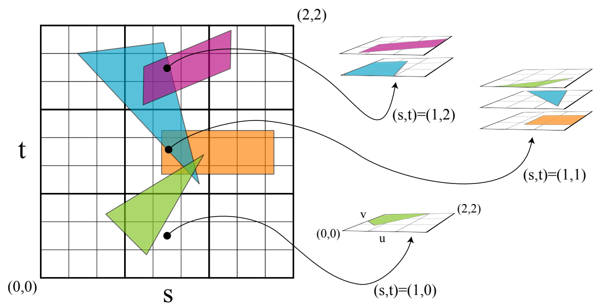


**Figure 3: Examples of tile lists for four polygons projected to a PSF where $n = \frac{N}{m} = 3$ and $m = 3$. Bold lines mark the tile boundaries.**

## 3. PROPAGATION ON THE GPU

As in the earlier CPU based approach [37], propagation of radiance is performed for each global bundle of parallel rays specified by the PSFs stored in the data structure. Radiance is shot outwards from emitters and secondary emitters in subsequent iterations. The main bottlenecks of the earlier approach were solving the visibility and performing the radiance transport between pairs of mutually visible faces. Visibility was solved discretely for each receiver face using an OpenGL z-buffer to render a false colour image yielding the face indices of visible faces along the PSF direction. This was rendered at a supersampled resolution and read back to main memory where it defined a set of potential sender-receiver pairs of elements for which transport was computed using a continuous clipping algorithm to determine the mutually visible area. Computing visibility was $O(n^2)$ in the number of faces and reading this back to main memory is an expensive operation. Furthermore, the clipping algorithm was expensive and is typically performed intensively; where the number of texel exchange pairs was orders of magnitudes larger than the number of faces.

This paper recasts the propagation algorithm on two levels. First, the clipping to compute mutual visibility is replaced by dense point sampling. Secondly, the visibility is solved incrementally using information computed once in a pre-compute step. Finally the algorithm is implemented on programmable graphics hardware using rasterisation to compute visibility incrementally and texture mapping using render-to-texture functionality to perform the transfer.

### 3.1 GPU Data Structure

In order to perform the propagation on the GPU the radiance data must be available locally on the GPU. Reading and writing to the GPU is still prohibitively expensive; memory access between main memory and GPU memory is still on an order of magnitude slower than local memory access on the GPU (even with PCI-Express). There are two types of data that must be available to the GPU: Textures containing radiance caused by diffuse illumination over a polygonal patch, and tiles that contain radiance generated by non-diffuse illumination. A diffuse surface has a single texture map assigned to it since its emitted radiance is independent of outgoing direction. On the other hand non-diffuse surfaces have radiance maps representing unique outgoing radiance for each direction in the set of global directions; with one texture map for each PSF direction. In order to save memory these maps are tiled, as described earlier, and only tiles overlapping the face (projected to the PSF plane) are stored (see Figure 4). The number of non-diffuse tiles

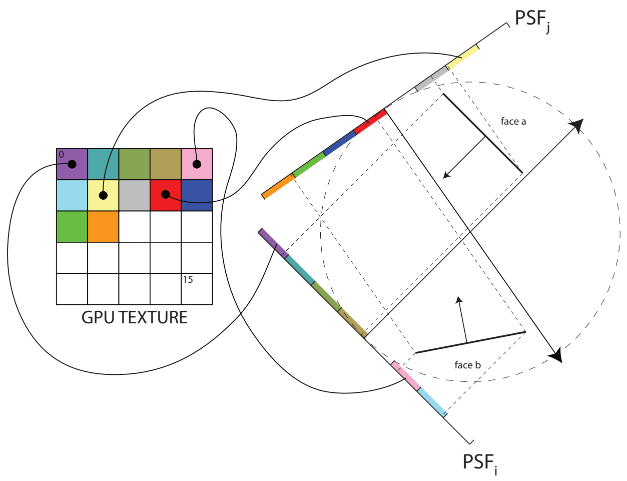


**Figure 4: GPU datastructure for non-diffuse faces is sparsely tiled and linearly stored in a large texture. Here two non-diffuse faces are projected to $PSF_i$ and $PSF_j$ and the tiles needed are shown.**

for a scene can be high, for example the scene shown in figure 1(a), which includes specular surfaces has ∼15k tiles and this is a relatively simple scene. It would be inefficient to store these tiles as separate textures since binding a texture to a framebuffer for render-to-texture operations has an associated overhead. The solution is to store many tiles packed in large textures on the GPU. For similar efficiency reasons diffuse textures are also stored in texture atlases.

In addition to the tiled texture map an auxiliary data structure is needed that can locate a tile in a texture for a given PSF and face. Figure 4 illustrates this; two faces have tiles stored in two PSFs which are linearly stored in a single texture. For example face b has four tiles associated with $PSF_j$, these are stored at the linear positions 8-11 (assuming position 0 is the upper left hand corner and indexing order is left to right, top to bottom).

During the propagation stage temporary storage is needed for unshot ir/radiance. Tiles and diffuse textures store three copies of each texture; the *current* map storing the outgoing ir/radiance in the current iteration, the *next* map storing the received ir/radiance in the current iteration (to be shot out in the next iteration) and, finally, the *total* map containing the accumulated ir/radiance over all iterations. During rendering only the latter is needed and the *current* and *next* maps can be discarded.

## 3.2 Visibility Computations

By far the most expensive operation in a global illumination method is determining visibility. For each point on a sender surface the surface identifier of the nearest visible point along the direction of a PSF must be found. In the previous approach the z-buffer was utilised to determine this. For each receiver face all other faces were rendered in false colour and the image was read back from the framebuffer. This algorithm was $O(n^2)$, where *n* is the number of faces in the scene. If we make the assumption that all faces are planar we can do better. The basic idea is to pre-sort the faces in the direction of the PSF using the painter's algorithm [26]. The sorting step is $O(n^2)$ in the worst case but for typical scenes it is $O(n \log n)$ on average. In [39] a similar approach was taken for a random walk algorithm. If $n$ is large the $n \log n$ term becomes expensive. This can be overcome by splitting the scene into smaller lists, sorting these individually and then merging the sorted results together. If $merge(sort(K), sort(L)) < sort(K + L)$ then it is gainful to do so. We found that for scenes with 4K faces or more it was beneficial to subdivide the scene using a BSP tree, placing the splitting plane at the centre of mass of the scene, and orthogonal to the PSF direction. In practice the relation above held as long as the sublists were longer than 2K polygons. Furthermore, objects enclosed by a bounding volume not intersected by any other object in the scene can be taken out of the scene and sorted separately. Later, this bounding volume can be used in place of the object to produce the final sorted sequence. Then in the final sequence the separately sorted list for the object is inserted in the position of the bounding volume in the global list. This benefits scenes organised hierarchically.

Given this initial sorting a propagation step can be performed in $O(n)$ time by visiting the faces from front to back and maintaining a *radiance-interface* (*RI*) which stores the radiance sent so far in the direction of transport. Faces oriented along the direction of transport (*senders*) update the *RI* with emitted radiance, and faces oriented opposite to the direction of transport (*receivers*) receive radiance from the *RI*. Finally this is repeated in back to front order so that receivers become senders and vica versa. Figure 5 illustrates

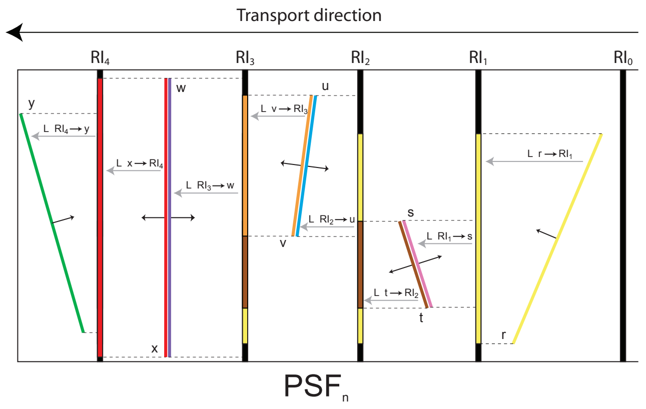


**Figure 5: Incremental approach for radiance transfer with pre-computed sorting. Transfer takes place from right to left.**

this concept. Initially the *RI* (*RI*$_0$) is empty. Then face *r* updates the *RI* (*RI*$_1$) with emitted radiance. Face *s* receives radiance from (*RI*$_1$) and face *t* updates the *RI* (*RI*$_2$) partly overwriting the radiance written by *r*. Face *u* then receives radiance from (*RI*$_2$) which is radiance partly coming from face *r* and face *t*. This process is repeated until all faces have been visited. Then the process is repeated in the opposite direction. So far, the incremental approach requires that the scene consists of *closed* polyhedra, support for two-sided faces (thus *open* polyhedra) is trivial and would require that receiver faces store ir/radiance values for both sides of the face, and are treated both as a receiver *and* sender when visited.

As noted this has $O(2n) \equiv O(n)$ complexity where *n* is the number of faces in the scene.

## 3.3 Radiance transport

The previous section dealt with the high level interactions between faces. In this section the low level radiance transport to/from the *RI*

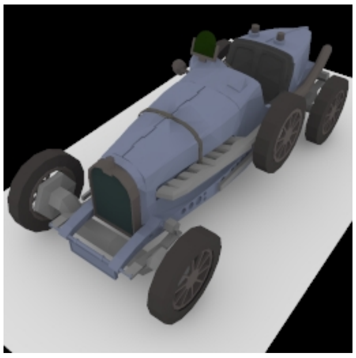


**Figure 6: Skylight rendering.**

is described. There are a number of ways this can be done. The main distinction is between *continuous* and *discrete* techniques. Both could be used with the high level transport described above or even with a combination of techniques. Refer to [39] for a good overview of various approaches to this in a similar context to ours. In our approach we deal with a GPU implementation and although it would be possible to implement a (partly) continuous approach given the programmable nature of the current generation of GPUs it would be inefficient compared to a discrete one based on rasterisation which is what the GPU does best. An important part of most continuous techniques is a continuous clipping operation that determines overlaps between faces; these normally require looping and branching which is possible to do on current GPUs but it is inefficient. GPUs are highly parallel streaming processors that are best at performing the same operations on an (independent) stream of input data. Besides, by observing the sampling theorem and sampling the signal adequately, a discrete technique would produce similar results [7]. So in order to leverage the power of the GPU the transport technique will be a discrete one.

In order to sample adequately the sampling scheme needs to be carefully chosen. Since an *RI* is merely a temporary container for radiance, care must be taken not to introduce artifacts when transferring radiance to/from it. On a CPU based simulator we tried various schemes. One such scheme is to fix the number of point samples used per face element (*texel*) and make sure that the resolution of the *RI* is at least as high (per unit area) as that of the faces (per unit projected area along the PSF). However, this can be wasteful when the angle between the face normal and PSF direction is large, this would use far more samples than necessary, and to counter this we tried an adaptive scheme that scaled the sampling density with the projected area of a texel upon the *RI*. This produced nearly identical results but was three times faster.

Although such a scheme would be possible to implement on the GPU [28] a much more efficient approach is to use the built in hardware texture mapping. Current hardware supports floating point textures, and with mip-mapping and anisotropic filtering this is viable as long as enough samples are taken when performing the texture mapping operation. Two such texture mapping operations will be necessary in this context, one that maps a face onto the *RI* and one that maps the *RI* onto a face. By ensuring that the *RI* is supersampled aliasing can be avoided.

Skylight rendering is trivially supported with this method, it simply requires resetting the $RI$ with a suitable radiance value other than black before propagating a PSF. This is illustrated in Figure 6.

## 3.4 Non-diffuse Scattering

When non-diffuse surfaces are involved in the transport the directionally dependent radiance stored for the surface needs to be taken into account. To recap, every non-diffuse surface stores a set of tiles for each PSF which contain the outgoing radiance in that PSF's direction. Two copies of these maps are stored; one contains the unshot radiance to be distributed in the current iteration (*cURM*) and the other the in-scattered radiance to be distributed in the next iteration (*nURM*). These are aligned with the PSF plane and cover the projection of the face onto the PSF plane. Figure 7 illustrates this concept. Updating the *RI* with non-diffuse radiance is trivial

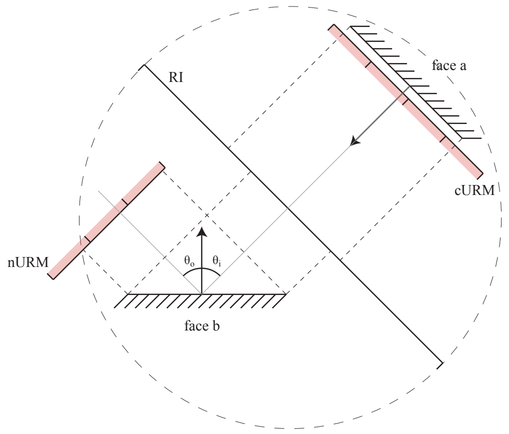


**Figure 7: Non-diffuse transfer and scattering. Tiles for face *a* are shown only in the direction of the current PSF, tiles for face *b* are shown for the current PSF and for the PSF in the reflected direction.**

since the tiles are aligned with the PSF. The radiance values from the tiles that fall within the projection of the face onto the *RI* are simply written directly to the *RI*. When a non-diffuse face is a receiver the situation is slightly more complex. In this case the radiance stored on the *RI* within the projection of the face needs to be scattered off the face into a set of tiles for the PSF in the reflected direction. In practice a PSF may not exist in the *exact* reflected direction in which case the radiance is scattered to the three nearest PSFs with barycentric weights based on the angular distance to the *exact* reflected direction.

In order to perform the scattering a backwards mapping operation is performed. The corners of each tile in the reflected direction are transformed into the PSF of the current transport direction via the plane of the reflecting face. These serve as texture coordinates into the *RI* and the scattering is performed by a texture mapping operation weighted by the barycentric weight and the *BRDF* of the reflector.

## 3.5 High Quality Propagation

As discussed above $l$ directions are uniformly selected on the hemisphere for radiance storage and propagation purposes. Each direction accounts for an area on the sphere in which the stored radiance is assumed to be constant. During rendering these constant radiances are interpolated to produce radiance values for pixels in the image plane.

In many cases propagation along these fixed directions produce satisfactory results. However, if exceptionally high quality is required sampling the solid angle of each direction with a *single* centrally placed sample can be insufficient. In such scenarios additional *virtual* directions are added during transport. They do not store radiance maps but improve the quality of the stored radiance samples by sampling the solid angle subtended by each *fixed* direction with a number of samples. These samples are randomly chosen and distributed over the area of the solid angle subtended by each fixed direction providing a set of perturbed directions used exclusively during transport and then discarded.

This approach greatly improves quality without increasing memory consumption. Propagation time, however, increases linearly with the number of added virtual directions. Also, since these samples use distinct perturbed directions they, like fixed directions, need to be depth sorted. This is currently carried out on the CPU in parallel with propagation performed on the GPU. This is possible on even low end machines without multiple cores due to the wide availability of hyper threading technology.

## 4. RENDERING ON THE GPU

When the VLF propagation step has converged, the GPU can render novel views from the data structure by interpolating between samples stored in the diffuse textures and non-diffuse view-dependent radiance tiles. Diffuse surfaces can be rendered directly using texturing with the diffuse textures available in the texture atlases. The texture hardware performs interpolation efficiently in this case.

Flat specular faces can be rendered with ray-tracing by recursively following a view ray reflected in the specular face until it strikes a diffuse face where the visible radiance can be collected. A similar idea is to use the stencil buffer to render a reflected view of the scene as seen through the specular face and then paste this onto the face [20]. These methods are only efficient if few specular surfaces are present in the scene and do not apply to, for example, glossy BRDFs.

A more general method is to resample images from the directionally dependent radiance stored in the non-diffuse radiance tiles. Recall that the data structure can be formalised as $L_s(\omega, s, t, u, v, p)$, where $\omega$ indicates the PSF for direction $\omega$, $(s, t)$ a tile belonging to face $p$, $(u, v)$ a cell in this tile. This effectively retrieves a radiance value travelling in direction $\omega$, from a point on $p$ described by the intersection of the canonical ray $(s, t, u, v)$ with $p$. Due to the discrete representation a PSF matching *exactly* the direction $\omega$ is rarely available. We thus use an interpolation scheme similar to that described in Section 3.4. In detail, the three PSFs $(\omega_i, \omega_j, \omega_k)$ at the vertices of the spherical triangle in which $\omega$ falls are used weighted by barycentric weights $(\alpha_i, \alpha_j, \alpha_k)$ for an interpolated value:

$$\begin{aligned} L_s(\omega, s, t, u, v, p) = {} & \alpha_i * L_s(\omega_i, s, t, u, v, p) \\ & + \alpha_j * L_s(\omega_j, s, t, u, v, p) \\ & + \alpha_k * L_s(\omega_k, s, t, u, v, p) \end{aligned}$$

In a pre-pass the camera is placed at the centre of the unit sphere and the spherical triangles are rendered in false colour to a texture. This produces the indices of the three nearest PSFs for each pixel. This is repeated in a second pass this time setting vertex colours for each spherical triangle to $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$. The rendering hardware interpolates this over each triangle, resulting in a texture with three barycentric weights for each pixel. In order to determine $p$, a third false colour pass is rendered, this time rendering the scene geometry, yielding a texture with a face identifier for each pixel. A fourth pass renders the scene geometry again where each vertex is coloured with its world coordinate position, interpolation across the geometry produces a texture with the world coordinate position of the intersection of the viewing ray for that pixel with the face $p$. Note, that ray casting could replace these last two passes. A fifth and final pass renders the final radiances to the image. For each pixel this is achieved by mapping the hit position to each of the three PSFs by applying the respective $M_{WC\rightarrow PSF}$ matrix (see Section 2.1) to the hit position, producing an $(x, y, z)$ value in canonical PSF coordinates where $(x, z)$ trivially maps to a tile/cell pair $(s, t, u, v)$. The tiled data structure is then looked up and a radiance value for each PSF is weighted by its corresponding barycentric weight and written to the image.

The performance is dependent on the time taken to resolve visibility, the remaining passes and radiance retrieval is constant time per pixel. Either ray tracing or rasterisation can be used to resolve the visibility, here we use the latter. One of the main points of the VLF approach is that global illumination values can be retrieved directly from a data structure, no further shadow rays or sampling is necessary, resulting in stable, predictable frame-rates.

## 5. IMPLEMENTATION DETAILS

The implementation was written in Cg [24] and OpenGL on an Nvidia GeForce 8 series GPU. Recent OpenGL 2.0 extensions were heavily used such as floating point textures and framebuffer objects for simple render-to-texture functionality. In order to avoid severe penalties for texture state changes a texture atlas was employed for textures; packing many textures in a few high resolution texture maps. Further, costly state changes such as setting projection and viewing matrices were avoided by storing matrices for the global set of directions on the GPU and setting them there in a vertex shader; thus exploiting the high internal bandwidth of the GPU rather than sending the matrices from the CPU to the GPU across the (much slower) PCI-Express bus.

Currently, only diffuse, specular and mixed diffuse/specular surfaces are supported. It is, however, trivial to add more complex BRDFs.

## 6. RESULTS

Performance of the algorithm was tested on four scenes, two consist of only diffuse surfaces (Figure 1(b) and Figure 1(c)) and the remaining two of diffuse and specular surfaces (Figure 1(a) and Figure 1(d)).

All figures use 512 bi-directional global directions, and Figure 1(a) use PSFs constructed at a resolution of 64x64 cells (8x8 tiles, 8x8 cells per tile), whereas Figure 1(d) use 256x256 cells (16x16 tiles, 16x16 cells per tile). The mirror wall in Figure 1(d) is partly diffuse and specular, containing a slightly bluish diffuse element. The ratio of directions to spatial resolution is a function of the distance between mutually visible polygons and can be computed automatically. Here, however, we set these parameters manually.

Propagation timing results from the scenes are presented in Table 1. They are all based on three propagation iterations. Timings were obtained on an Intel Core 2 Quad (QX6700) 2.66GHz processor with a GeForce 8800 GTX with 768MB graphics memory and 4GB of host memory. Initial sorting was performed on the CPU in four threads using a 2-level BSP subdivision along the depth of the PSF. For the Cornellbox scene the sorting was faster on a single core

| Scene | Faces | Diffuse elements | Non-diffuse elements | Sort (sec) | GPU init (sec) | Propagation (sec) | Total (sec) | Memory (propagate) | Memory (render) | Rendering (1024×768) |
|---|---|---|---|---|---|---|---|---|---|---|
| Cornellbox | 19 | 34,816 | 931,136 | 0.031 | 0.041 | 3.995 | 4.067 | 23.2MB | 2.9MB | 122fps |
| Grotto | 318 | 816,640 | 0 | 0.549 | 0.003 | 16.650 | 17.202 | 19.6MB | 2.5MB | 124fps |
| Classroom | 3,024 | 905,984 | 0 | 11.149 | 0.028 | 52.518 | 63.695 | 21.7MB | 2.7MB | 121fps |
| Atenea | 9,410 | 2,434,560 | 7,629,824 | 18.656 | 1.416 | 133.136 | 153.208 | 241.5MB | 30.2MB | 121fps |

**Table 1: Performance of the VLF-GPU method.**

without BSP subdivision due to the scene's low number of polygons.

A 64-bit floating-point RGBA format was used for storing radiance and irradiance. Memory consumption is given for the propagation stage as well as the rendering stage. After propagation all *current* and *next* ir/radiance maps were discarded accounting for two-thirds of the memory and the *total* ir/radiance maps were tonemapped and stored as 24bit RGB for display. For offline storage the maps can be further compressed using standard image compression methods. Storage during propagation can be roughly halved using 4 byte shared exponent format (similar to Greg Ward's RGBE format), which was introduced as a native pixel-format with the GeForce 8 series.

This approach clearly trades off memory for propagation and rendering speed. Memory consumption is proportional to the area of non-diffuse surfaces and the frequency of their associated BRDFs. Low-frequency BRDFs can be adequately represented with few directions, whereas, high-frequency BRDFs such as mirrors require very many directions to represent faithfully. Due to the use of perturbed directions (see Section 3.5), the effects of high-frequency BRDFs, such as caustics, are still well preserved even with a low directional discretisation. Extensions to allow for larger scenes with many non-diffuse surfaces include adding compression during propagation or using methods such as final gathering [17] in order to reconstruct directly visible non-diffuse surfaces.

If we, in a *hypothetical* scenario, were to consider each sample transported in the propagation step as requiring *one* ray intersection computation for visibility, we would in effect achieve $\sim$33M ray intersections per second on average for the four scenes in Table 1 (between $\sim$14Mrays/sec for the Cornell scene and $\sim$74Mrays/sec for the Grotto scene). This is partly due to the fact that we exploit coherence by "tracing" bundles of parallel rays and partly because the incremental approach eliminates the need for traversing a data structure to resolve visibility. To our knowledge, the fastest ray tracing method to date is *Multi-level Ray Tracing* (MLRT) [34] reporting between 16Mrays/sec and 51Mrays/sec, for shaded non-textured primary rays. Comparably, GPU ray tracing implementations have not proven effective [13, 3, 31], reporting less than $\sim$1Mray/sec (extrapolated to current hardware). Our memory bandwidth is much higher than that of MLRT since each ray intersection requires HDR texture lookups of radiance values; this yields a memory bandwidth of $>$3.56GB/sec for the Grotto scene. Even so, our visibility performance is comparable to that of MLRT. We stress, however, that the method here does not easily apply to general ray tracing, since general ray queries are much less coherent than bundles of parallel rays and using global rays with sorted face sequences is not readily applicable to a general ray tracer. The numbers are given in order to put the visibility performance of the incremental approach in context.

Overall, propagation time shows >2.5 orders of magnitude improvement over earlier VLF propagation results [37] for simple scenes with <1000 polygons. More complex scenes are intractable with the previous approach due to its $O(n^2)$ complexity. This significant improvement is due equally to the improved propagation scheme, incremental transfer and GPU implementation.

Rendering can easily be done in real-time at high resolutions. The frame-rates given in Table 1 are for 1024×768 frames. Frame-rates are stable regardless of the viewpoint due to the fact that the multiple rendering passes are performed for *all* pixels in the frame. Clearly, this could be optimised by only performing the view-dependent rendering on non-diffuse pixels. This would however make the frame-rate variable with a worst case performance of that given in Table 1 when the viewpoint is such that only non-diffuse pixels are viewed.

Additionally, the results were verified by comparing images computed with PBRT [29] a path-tracing system used widely in teaching and by academics in the field of global illumination. The path tracer used importance sampling and 1024 paths per pixel see Figure 8(a). Two images were produced with our method one using 2k fixed directions only (Figure 8(b)) and one using 2k fixed directions and 36 perturbed samples per fixed direction (Figure 8(c)). The PBRT image and the "low quality" VLF-GPU image were subtracted and the MSE was around $1e^{-3}$. The peak errors were around $1e^{-2}$ and concentrated in areas where the low object-space resolution of the ir/radiance was inadequate to efficiently represent high frequency illumination in image space. One example is the shadow at the base of the tall specular box which appears blurred with the VLF method, also the caustic on the right-hand side of the ceiling is less defined than with PBRT. However, the noise apparent in the path traced image even with 1024 rays per pixel is absent from the image computed with VLF-GPU. The high-quality VLF-GPU image has no visible error in the caustic, much less blurring of the shadows and virtually no noise.

## 7. CONCLUSIONS AND FUTURE WORK

The aim of this work has been to significantly improve propagation performance for moderately complex scenes in the VLF context. The several orders of magnitude improvement in performance over previous work [37] presents very favourable results. The algorithm now scales linearly $O(n)$ in the number of input polygons. Our performance is comparable to that of [8] (extrapolating to current hardware), but the VLF-GPU method has fewer artifacts and supports non-diffuse BRDFs. The current implementation supports specular, diffuse and mixed specular/diffuse, thus accounting for

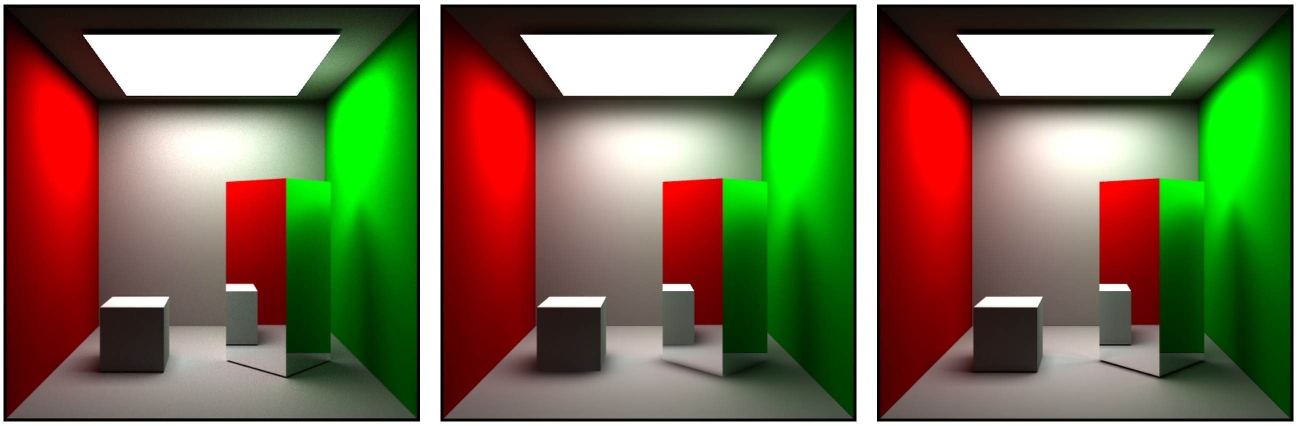

(a) Rendered with PBRT. (b) Rendered with VLF-GPU (low quality). (c) Rendered with VLF-GPU (high quality).

**Figure 8: Comparison of VLF-GPU and path tracing for a specular Cornellbox scene.**

$L(S|D)^*E$ paths, whereas [8] is diffuse only. The scene in Figure 1(b) is a slightly modified version of the Grotto scene used in [8].

We have presented an algorithm that runs entirely on the GPU requiring the CPU only to do minor bookkeeping tasks. The method supports textures that are fully integrated in the propagation stage replacing the constant $K_d$ term with one that varies across a surface as defined in a HDR texture. Textured emitters are similarly supported, see Figure 1(b). Also, emitters are not distinguished from other surfaces. Any surface can have emission; propagation and rendering time is invariant to the number and cumulative area of emissive surfaces, which is a desirable property achieved by few global illumination methods. Skylight rendering is also supported without any additional performance penalty.

Rendering from a converged VLF can be done in real-time with very stable frame rates, making this a practical solution for virtual reality applications, where the frame-rate *must* be real-time and constant, even temporary drops in frame-rate can cause the subject to lose orientation, and maybe cause motion sickness. Lack of stable frame-rates is a weakness of many caching algorithms where a sudden change in viewpoint can produce a view that is not fully represented in the cache, causing a temporary drop in fidelity or frame-rate. Similarly dynamic techniques such as ray tracing for global illumination can also exhibit variable frame-rates when the viewpoint changes from a complex region to a less complex region in terms of illumination.

We are currently in the process of porting this system to the CAVE® environment [9] in preparation for studies of the effect of global illumination on the sense of presence [19].

In future work we plan to extend the algorithm in several ways including efficient support for dynamic scenes, curved surfaces and relighting with the VLF. BRDFs are simple with the VLF-GPU method and we are currently adding glossy reflection to the repertoire. We are also looking into methods to have memory efficient adaptive representations for texture maps and varying directional representations for specular and BRDF surfaces so as to significantly improve performance for larger, more complex scenes. We are also investigating methods of distributed representation on multi-GPU systems.

## Acknowledgments
This research was funded by EPSRC grant EP/C511824/1.

## 8. REFERENCES

[1] E. Camahort and D. Fussell. A geometric study of light field representations. Technical Report TR9935, Department of Computer Sciences, The University of Texas at Austin, 1999.
[2] E. Camahort, A. Lerios, and D. Fussell. Uniformly sampled light fields. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop '98)*, pages 117–130, 1998.
[3] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, 2002.
[4] N. A. Carr, J. D. Hall, and J. C. Hart. GPU algorithms for radiosity and subsurface scattering. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 51–59, 2003.
[5] N. A. Carr, J. Hoberock, K. Crane, and J. C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of the 2006 conference on Graphics interface*, 2006.
[6] Y. Chrysanthou, D. Cohen-Or, and D. Lischinki. Fast approximate quantitative visibility for complex scenes. In *Proceedings of Computer Graphics International '98 (CGI '98)*, pages 220–227, 1998.
[7] R. L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, Jan. 1986.
[8] G. Coombe, M. J. Harris, and A. Lastra. Radiosity on graphics hardware. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 161–168, 2004.
[9] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. The cave: audio visual experience automatic virtual environment. *Commun. ACM*, 35(6):64–72, 1992.
[10] M. de Berg, M. Overmars, and O. Schwarzkopf. Computing and verifying depth orders. In *SCG '92: Proceedings of the eighth annual symposium on Computational geometry*, pages 138–145, New York, NY, USA, 1992. ACM Press.
[11] M. Escriva, A. Domingo, F. Abad, R. Vivo, and E. Camahort. Modeling and rendering of DPP-based light fields. In

*Geometric Modeling and Imaging–New Trends, 2006*, pages 51–56, July 2006.

[12] C. Everitt. Interactive order-independent transparency. Nvidia White Paper, 2001. http://www.nvidia.com.

[13] T. Foley and J. Sugerman. KD-tree acceleration structures for a GPU raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, 2005.

[14] P. Gautron, J. Krivanek, K. Bouatouch, and S. N. Pattanaik. Radiance cache splatting: A GPU-friendly global illumination algorithm. In *Rendering Techniques 2005 (Proceedings of the Eurographics Symposium on Rendering Techniques)*, pages 55–64, June 2005.

[15] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 43–54, 1996.

[16] T. Hachisuka. *GPU Gems 2*, chapter 38 - High-Quality Global Illumination Rendering Using Rasterization, pages 615–633. Addison Wesley, 2005.

[17] H. W. Jensen. Global illumination using photon maps. In *Rendering Techniques '96 (Proceedings of the 7th Eurographics Workshop on Rendering)*, pages 21–30, London, UK, 1996. Springer-Verlag.

[18] J. Kessenich, D. Baldwin, and R. Rost. *The OpenGL Shading Language 1.1*. 3Dlabs, Inc. Ltd., April 2004. Document Revision 59.

[19] P. Khanna, I. Yu, J. Mortensen, and M. Slater. Presence in response to dynamic visual realism: A preliminary report of an experiment study. In *VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 264–267, Limassol, Cyprus, November 2006.

[20] M. J. Kilgard. Improving shadows and reflections via the stencil buffer. White paper, Nvidia Corporation, 2002.

[21] B. D. Larsen and N. J. Christensen. Simulating photon mapping for real-time applications. In *Rendering Techniques 2004 (Proceedings of the 15th Eurographics Workshop on Rendering)*, pages 123–132, 2004.

[22] M. Levoy and P. Hanrahan. Light field rendering. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42, 1996.

[23] V. C. H. Ma and M. D. McCool. Low latency photon mapping using block hashing. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 89–99, 2002.

[24] W. R. Mark, R. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics*, 22(3):896–907, 2003.

[25] D. M. Naga K. Govindaraju, Ming Lin. Vis-sort: Fast visibility ordering of 3-d geometric primitives. Technical report, University of North Carolina at Chapel Hill, 2004.

[26] M. Newell, R. Newell, and T. Sancha. A solution to the hidden surface problem. In *ACM'72: Proceedings of the ACM annual conference*, pages 443–450, 1972.

[27] M. Nijasure, S. Pattanaik, and V. Goel. Real-time global illumination on the GPU. *Journal of Graphics Tools*, 10(2):55–71, 2005.

[28] J. Novosad. *GPU Gems 2*, chapter 27 - Advanced High-Quality Filtering, pages 417–435. Addison Wesley, 2005.

[29] M. Pharr and G. Humphreys. *Physically Based Rendering : From Theory to Implementation*. Morgan Kaufmann, 2004.

[30] T. J. Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, Mar. 2004.

[31] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[32] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 41–50, 2003.

[33] Z. Ren, W. Hua, L. Chen, and H. Bao. Intersection fields for interactive global illumination. *The Visual Computer*, 21(8–10):569–578, September 2005.

[34] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Transactions on Graphics*, 24(3):1176–1185, 2005.

[35] J. Shade, S. Gortler, L. wei He, and R. Szeliski. Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242, 1998.

[36] L. O. Simonsen and N. Thrane. A comparison of acceleration structures for GPU assisted ray tracing. Master's thesis, University of Aarhus, 2005.

[37] M. Slater, J. Mortensen, P. Khanna, and I. Yu. A virtual light field approach to global illumination. In *Proceedings of Computer Graphics International (CGI 2004)*, pages 102–109. IEEE Computer Society Press, June 16-19 2004.

[38] C. M. Stein, B. G. Becker, and N. L. Max. Sorting and hardware assisted rendering for volume visualization. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 83–89, Oct. 1994.

[39] L. Szirmay-Kalos. Global ray-bundle tracing. Technical Report TR-186-2-98-18, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 1998.

[40] L. Szirmay-Kalos and W. Purgathofer. Global ray-bundle tracing with hardware acceleration. In *Rendering Techniques '98 (Proceedings of the 9th Eurographics Rendering Workshop)*, pages 247–258, June 1998.

[41] V. Vivanloc, J.-C. Hoelt, C. B. Hong, and M. Paulin. Light Octree: Global Illumination Fast Reconstruction and Realtime Navigation. In *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, University of West Bohemia, Campus-Bory Plzen*, volume 15 of *Journal of WSCG*, pages 41–50, January 2007.

[42] C. Vogelgsang and G. Greiner. Hardware accelerated light field rendering. Technical Report 11, Universitat Erlangen-Nurnberg, 1999.

[43] M. Wand and W. Straßer. Real-time caustics. In *Computer Graphics Forum*, volume 22(3), pages 611–620, 2003.

[44] P. L. Williams. Visibility-ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.