High Performance Graphics (2016)
Ulf Assarsson and Warren Hunt (Editors)

Lightcut Interpolation

Hauke Rehfeld and Carsten Dachsbacher

Karlsruhe Institute of Technology

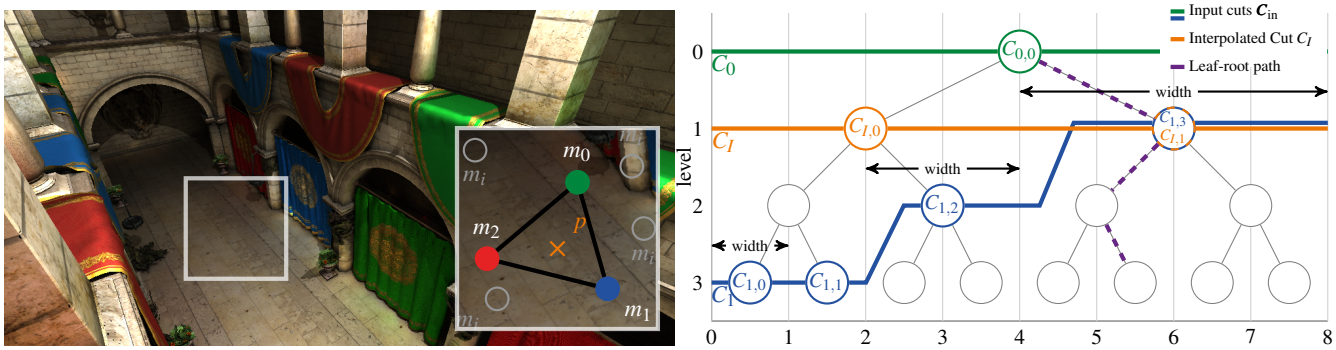

Figure 1: *When using Lightcuts, computing the cuts through the hierarchy of lights makes up a large portion of the render time. Our technique computes exact cuts only at a fraction of shading points and keeps them as cache records. For shading, we directly interpolate the nodes of these cuts.* ***Left:*** *The lightcut for shading point p is interpolated from the lightcuts at cache records m_0, m_1, m_2, which are selected from the set of cache records* $\boldsymbol{R}$ *distributed in image space.* ***Right:*** *Lightcuts through the light hierarchy. Here, C_0 and C_1 are the input lightcuts* $\boldsymbol{C}_{in}$ *to our algorithm. We present an efficient interpolation scheme to obtain a plausible lightcut C_I for the shading point p. It iterates over the input cuts, and ensures that the output is a valid cut which has only one intersection for every leaf-root path (purple line).*

Abstract

Many-light rendering methods replace multi-bounce light transport with direct lighting from many virtual point light sources to allow for simple and efficient computation of global illumination. Lightcuts build a hierarchy over virtual lights, so that surface points can be shaded with a sublinear number of lights while minimizing error. However, the original algorithm needs to run on every shading point of the rendered image. It is well known that the performance of Lightcuts can be improved by exploiting the coherence between individual cuts. We propose a novel approach where we invest into the initial lightcut creation at representative cache records, and then directly interpolate the input lightcuts themselves as well as per-cluster visibility for neighboring shading points. This allows us to improve upon the performance of the original Lightcuts algorithm by a factor of $4-8$ compared to an optimized GPU-implementation of Lightcuts, while introducing only a small additional approximation error. The GPU-implementation of our technique enables us to create previews of Lightcuts-based global illumination renderings.

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

1. Introduction

Physically-based rendering is key to computing photorealistic images and has been rapidly adopted in the feature-film and gaming industries in the past years. In this paper, we focus on rendering global illumination with many-light methods [DKH*13]. These methods build on the idea of approximating the global light transport by computing the direct illumination from many virtual point lights (VPLs). While they do not handle all light transport components without hassle, these methods enable scalable and robust rendering algorithms, which typically have more predictable render times compared to other techniques. The challenge with many-light methods is to efficiently render up to millions of VPLs [WHY*13a], which are required for complex scenes and materials.

Lightcuts [WFA*05], and several works in the same direction, aims to alleviate this problem by clustering lights into a hierarchy, so that multiple VPLs can be approximated with a single representative VPL per cluster. During rendering, for each shading point the light hierarchy is traversed to find a cut (a set of clusters) whose joint contribution approximates the lighting from all VPLs within a certain error bound. The size of these cuts typically only weakly depends on the number of VPLs and leads to sublinear shading cost.

In the Lightcuts algorithm the main computation costs are:

a) computing the cuts for each shading point, and b) test the visibility between the shading point and each cluster of the respective cut. In this paper, we compute the lightcuts only for a sparse subsampling of the image space and propose to *interpolate the cuts* directly for all other shading points. This is a novel approach making use of the observation that cuts of nearby shading points are typically similar [WFA*05, DBD08, WXW11, HPB07, OP11]. Interpolation typically is much cheaper than recomputing (or refining) cuts. It also avoids artifacts from clustering shading points, as each point is shaded using a fully valid lightcut. And lastly, akin to (Ir-)Radiance Caching [WRC88, KGPB05] we effectively interpolate incident radiance, which carries more information than just interpolating shading and enables us to *interpolate cluster visibility* along with the cuts, saving costly ray casting. In summary, our contributions are:

- An interpolation algorithm working directly on (light-) cuts through (light) hierarchies.
- A rendering algorithm with subsampled lightcuts creation and subsequent interpolation of the cuts and cluster visibilities.
- A fully GPU-based implementation that enables fast previews of lightcuts-based many-light rendering.

2. Previous Work

Simulating global illumination is a challenging problem that has been addressed by many researchers. In the following we review the most related previous work.

(Ir-)Radiance Caching Ward et al. [WRC88] propose to cache irradiance in diffuse scenes and interpolate it at every shading point that finds nearby cache records. If no records are found, irradiance is computed and a new record is created. The method was later extended to also store irradiance gradients which are then used to extrapolate irradiance [WH92]. Radiance Caching stores incoming radiance in the records to allow for non-diffuse materials [KGPB05]. Later extensions acknowledge that on-the-fly record placement during rendering can lead to suboptimal record distributions and try to refine the cache completely before computing illumination for shading points [KBPZ06]. The error metric was further improved by using second order gradients including occlusion information [SJJ12]. Radiance gradients computed using Lightcuts deform the area of influence of cache records anisotropically [HMS09].

Pre-convolved Radiance Caching [SNRS12] proposes to store incident radiance in a texture with one texel per direction and pre-convolve it—in essence creating a mip-map-pyramid of the radiance texture with accumulated radiance in the higher levels. Additionally, a low-resolution mip-map level is cosine folded to store irradiance. Using these two textures, evaluating a radiance cache requires only two texture lookups instead of re-evaluating the reflection functions. Rehfeld et al. [RZD14] introduce an algorithm for interactive radiance caching using pre-convolved radiance caching. They propose an interactive, GPU-based radiance cache distribution scheme that works in image space. To distribute the caches they use a clustering approach akin to Clustered Deferred Shading [OBA12], and then place one cache in each cluster. While the resulting distribution does not miss high-frequency detail, it is clearly tailored for runtime budgets of below 100ms. Recently, interactive methods that create distributions that are more similar to traditional radiance caching distributions have been proposed [Ulb15], see Sec. 3.1. Wang et al. [WWZ*09] place caches by first building a mip-map hierarchy over all shading points and then using a simplified split-sphere model metric [WH92] to approximate the amount of coherence in each quad of a mip-map level. Next, they distribute a fixed number of caches proportionally to this approximated coherency top-down onto the shading points. After this distribution, however, they require to run a few iterations of k-means clustering.

Many-light methods Many-light methods simulate global illumination by shooting photons from light sources and placing many virtual point lights (VPLs) on the surfaces that are hit [Kel97]. This is similar to a bi-directional path tracing with eye paths of length one, where the high correlation between eye-light path connections can be exploited for efficiency. We refer the reader to [DKH*13] for a recent review of many-light methods.

Lightcuts Walter et al. [WFA*05] recognize that similar lights can be clustered together into a light hierarchy, which can then be used to find clusters that approximate sets of lights. The set of inner nodes (clusters) and leaf nodes (VPLs) that do not need to be refined further w.r.t. an error criterion form a cut through the light hierarchy, a *lightcut* (cut). To create the lightcut, the light hierarchy is traversed for each shading point starting at the root node. At each node the geometric and material terms of the current shading point and the current cluster are bounded to find the cluster's bounded error. This is the maximum error made if refinement would stop at this point and the cluster would be evaluated.

Walter et al. [WFA*05] also introduce a perception-based error criterion, where clusters are refined if their error bound is larger than a fraction (2%) of the total illumination estimate for the shading point. This estimate is obtained by saving a heap of nodes sorted by their error bounds during traversal; at each step the node with the highest error bound is inspected. This criterion leads to large sizes of the lightcuts in dark areas, which is why the authors limit the size of a single lightcut to 1000 nodes.

Once a cut fulfills the error criterion, clusters are evaluated like a VPL using their total intensity and representative light. Visibility from the shading point to the representative light is checked, the BRDF is evaluated and the contribution of the light is added to the shading point's radiance.

Originally, the authors suggest to use a greedy bottom-up approach to construct the light hierarchy, which is expected to be very costly to compute, but also hint at using a simple top-down approach with good results. Later they suggest to use agglomerative clustering to build the hierarchy in reasonable time [WBKP08].

Effort has been made to leverage the parallel computing power of the GPU to compute lightcuts. Davidovic et al. [DGS12] propose a consistent progressive algorithm to relax the VPL clamping when computing lightcuts on the GPU, while maintaining a constant memory footprint. Wang et al. [WHY*13b] use a Lightcuts-based approach for out-of-core rendering on the GPU.

Exploiting similarity of neighboring lightcuts Reconstruction Cuts [WFA*05] computes full lightcuts at the corners of image space rectangles. Various strategies are required to ensure these

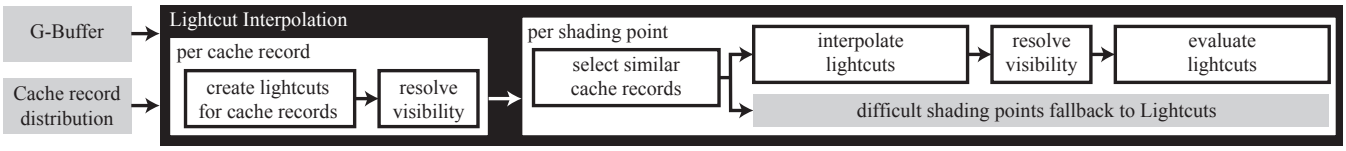

Figure 2: *Lightcut Interpolation algorithm overview.*

rectangles do not miss high-frequency geometry. A hierarchy of directional impostor lights that mimic the radiance estimate from the original clusters is then created for each reference lightcut. The original Lightcuts refinement criterion is replaced by a comparison of these impostors; nodes are evaluated using interpolated impostors, but may fallback to a full cluster evaluation including shadow rays. Coherent Lightcuts [DBD08] also uses bounding boxes on blocks of the image to refine shading points in unison until a heuristic decides that they are too incoherent. Wang et al. [WXW11] cluster shading points and use the cluster's common cut to start lightcut refinement.

In the context of Pre-computed Radiance Transfer [SKS02], Cheslack et al. [CPWAP08] propose a merging algorithm for pre-computed visibility cuts [AUW07]. They save visibility cuts for the vertices of an input mesh and merge them during runtime using barycentric coordinates. See the discussion in Sec. 5.

Matrix Interpretation The many-light problem can be interpreted as a matrix that relates shading points to lights, which can be analyzed and subdivided into submatrices either globally for all shading points [HPB07], or for refined clusters of shading points [OP11]. Recent extensions use matrix separation techniques to numerically recover the visibility in the submatrices so that fewer shadow rays are required [HWJ*15]. As the latter approaches do not rely on a global clustering of lights as Lightcuts does, they can be very successful to capture high-frequency illumination. However, they are limited to offline rendering due to their long rendering times.

3. Lightcut Interpolation

The Lightcuts algorithm refines the lightcut for a given shading point when certain clusters of VPLs of the light hierarchy are more important (i.e. their contributions to this shading point are larger). We exploit the similarity between lightcuts of geometrically close shading points. For this, our algorithm examines the lightcuts of close cache records distributed over the image space, and interpolates the *levels* of their individual nodes to create a cut with plausible refinement. This allows us to shade using valid lightcuts at all shading points while skipping light hierarchy traversal completely for all interpolated shading points—one of the bottlenecks of the original Lightcuts algorithm even in optimized GPU implementations.

First we create lightcuts using the original Lightcuts algorithm at a set of cache records placed in image space and save them. Then for each shading point we select the best cache records to use as input lightcuts $\boldsymbol{C}_{\text{in}}$ to our interpolation algorithm. In any cut, each path from a leaf node to the root of the light hierarchy intersects exactly one node of the cut (see Fig. 1 (right)). We exploit this property and merge nodes from the input lightcuts by identifying which nodes intersect which leaf-root paths. The interpolated lightcut will then consist of nodes that cover the same leaf-root paths, but at the levels of the light hierarchy that correspond to the weighted average of input levels. This ensures that the output is always a valid lightcut.

Optionally, we sample the binary visibility for each lightcut node at the cache records before interpolation and interpolate it with the nodes. This gives us a visibility in $[0,1]$ for each node in the interpolated cut without further cost for tracing shadow rays. We evaluate all steps of our algorithm in Sec. 6.

Our algorithm consists of these steps, which we detail in the following sections (Fig. 2, see Fig. 3 for notation):

1. For each cache record r from a set of input cache records $\boldsymbol{R}$:
   a. Traverse light hierarchy to create lightcuts $\boldsymbol{C}_{\boldsymbol{R}}$,
   b. Shoot shadow rays for all nodes in the lightcuts $\boldsymbol{C}_{\boldsymbol{R}}$.
2. Then, for each shading point p:
   a. Select cache records $\{r_j \in \boldsymbol{R}\}$ for interpolation,
   b. Interpolate their lightcuts $\boldsymbol{C}_{\text{in}}$ yielding the interpolated cut C_I,
   c. Shade using interpolated lightcut C_I with interpolated or accurate visibility (tracing rays).

We want our interpolation to be local to not miss higher-frequency illumination details, and thus limit our interpolation to a maximum of three input lightcuts. As we focus our algorithm on preview rendering, we also impose the limit for performance reasons. Note that our lightcut interpolation algorithm works for an arbitrary number of input cuts.

3.1. Input Cache Records

The cache records are the locations where we compute and store exact lightcuts. While the distribution of these cache records is mostly orthogonal to our interpolation algorithm, the creation of the input set of cache record locations is crucial for fast preview render-

Symbol	Description
$N(i,l)$	node at level l intersecting the leaf-root path from the i-th leaf
$N(0,0)$	root node of the light hierarchy
$W(n)$	number of leaf nodes below node n
$\boldsymbol{C}_{\text{in}}$	input cuts to the interpolation
w_j	interpolation weight of input cut C_j
$C_j(i)$	node from cut C_j above the i-th leaf
$C_{j,k}$	k-th node from cut C_j
$L(C_{j,k})$	level of the node $C_{j,k}$
$p(C_j)$	pointer to the current node of cut C_j
$W(0..p(C_j))$	sum of widths of nodes up to and including node $p(C_j)$
sync pos.	from the next nodes in the input cuts, more than one extend over the fewest leaves

Figure 3: *Notations and symbols used in our paper.*

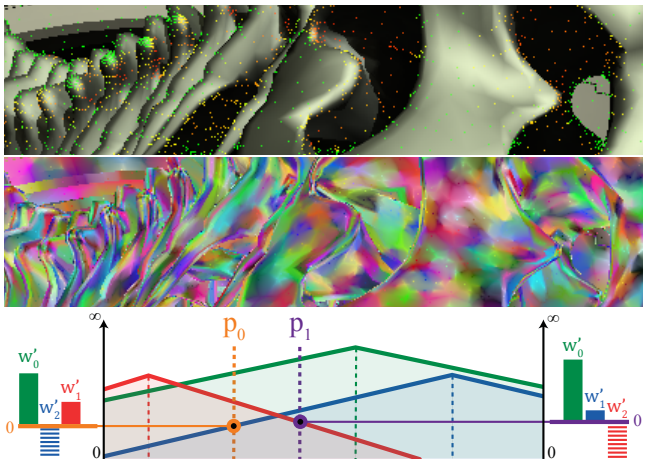

Figure 4: ***Top:*** *Distribution of cache records over the image space. Colors represent cache radius: red (small) to green (large).* ***Middle:*** *Interpolation weights. Colors correspond to cache record IDs and are then interpolated with each shading point's weights.* ***Bottom:*** *Best records are chosen by subtracting the $n+1$-th weight.*

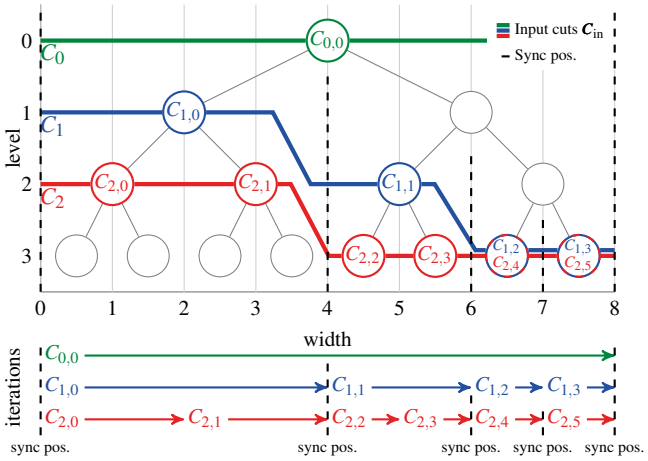

Figure 5: *Finding sets of nodes that intersect the same leaf-root paths and need to be interpolated. Our algorithm iterates over the nodes of the input lightcuts $\boldsymbol{C}_{in}$, fetching nodes from the cut with the smallest width $W\left(0..p\left(C_j\right)\right)$. Colored lines C_0, C_1, C_2 are three input cuts with their nodes, e.g. $C_{0,0}$ is the first node of the input cut C_0. Nodes with two colors are members of two input cuts. Vertical dashed lines denote sync positions (Sec. 3.3), where interpolation (Fig. 6) is triggered to create nodes for the preceding input nodes.*

ings. Refining a set of cache records or letting it fully converge (as in [KBPZ06, SJJ12]) is prohibitively expensive, and a direct, parallel approach fits our needs better. Also, we need to decouple from the tessellation of the geometry to not miss high-frequency details. We use the tile-based radiance cache placement method suggested by Ulbrich [Ulb15], which pre-computes the weights between shading points within a image space tile, and iteratively chooses those with the highest influence on others as cache records until all shading points have found at least some usable records. Traditional radiance cache weight functions (see Sec. 3.2) are supported, but no radiance is taken into account while creating the distribution. Our algorithm receives the list of coordinates of cache records and for each record an approximate harmonic mean distance to the surrounding geometry (shown in Fig. 4 (top)), which is required for the weight function used in cache record selection. For each cache record, we compute its lightcut, shoot a shadow ray for each node in the cut, and store both together for later interpolation.

3.2. Selecting Cache Records for Interpolation

Before interpolating lightcuts for a shading point, we determine the set of input lightcuts from nearby cache records and the corresponding weights, see Fig. 4 (middle). As cache records are distributed in image space, we cannot simply use barycentric interpolation (e.g. like [CPWAP08]). Instead, we select the best cache records heuristically: we iterate over the cache records that are close in image space, and evaluate a weighting function to favor cache records with position and orientation similar to the current shading point (see Fig. 1 (left, inset)).

We use Tabellion and Lamorlette's [TL04] extension of the sphere-model [WRC88] (ir-)radiance cache weighting function:

$$w^{TL}(p_p, p_r, \vec{n}_p, \vec{n}_r) = 1 - \max(d_p(p_p, p_r), d_n(\vec{n}_p, \vec{n}_r))$$

$$d_p(p_p, p_r) = \frac{\|p_p - p_r\|}{\max\left(\min\left(\frac{R}{2}, R_+\right), R_-\right)}$$

$$d_n(\vec{n}_p, \vec{n}_r) = \sqrt{\frac{1 - \vec{n}_p \cdot \vec{n}_r}{1 - \cos(\alpha_+)}}, \text{ where}$$

- p_p, p_r, $\vec{n}_p$ and $\vec{n}_r$ are the world-space positions and normals of the shading point and the cache record,
- R is the harmonic mean distance of the cache record to the geometry, which we receive as input from the cache record placement (see Fig. 4 (top)),
- R_- and R_+ clamp R with multiples of the projected pixel area, which we choose as 4 and 15 for slightly less dense cache records,
- α_+ is a maximum angle which we use to reject cache records completely if normals differ too much. Contrary to [TL04] it is always chosen as 15°.

We also reject cache records that are in front of the current shading point [WRC88], i.e. if $(p_p - p_r) \cdot (n_p + n_r) < 0$.

Best Three Caches and Interpolation Weights The metric enables us to rank the cache records for similarity, but the weights cannot be used directly for interpolation as they do not yield C^0-continuous interpolation. To this end, we first rank the *four* best cache records by their weights $w_0^{TL}, \ldots, w_3^{TL}$, with $w_i^{TL} \geq w_{i+1}^{TL}$. We subtract the fourth weight w_3^{TL} from the other three, i.e. $w'_i = w_i^{TL} - w_3^{TL}$ to receive the three interpolation weights. As shown in Fig. 4 (bottom), this ensures that the weights change smoothly, as weights of the least important cache records will approach zero if

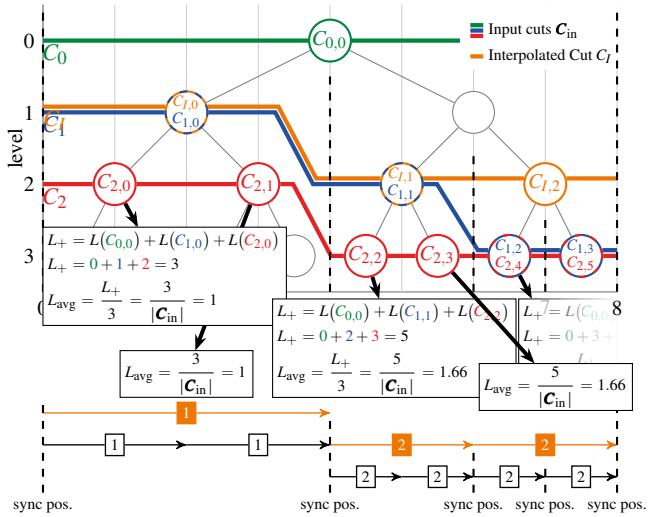

Figure 6: *Lightcut Interpolation walks the hierarchy from left to right. In this example and for brevity cut weights are $\frac{1}{3}$ each and are omitted. We iterate over the nodes at the lower edge of the input lightcuts C_{in}, at each step interpolating the levels of the current nodes (bottom). When a sync position (see Fig. 5) is reached, a node is appended to the interpolated cut C_I if it does not reach farther to the left or right than all current nodes (bottom).*

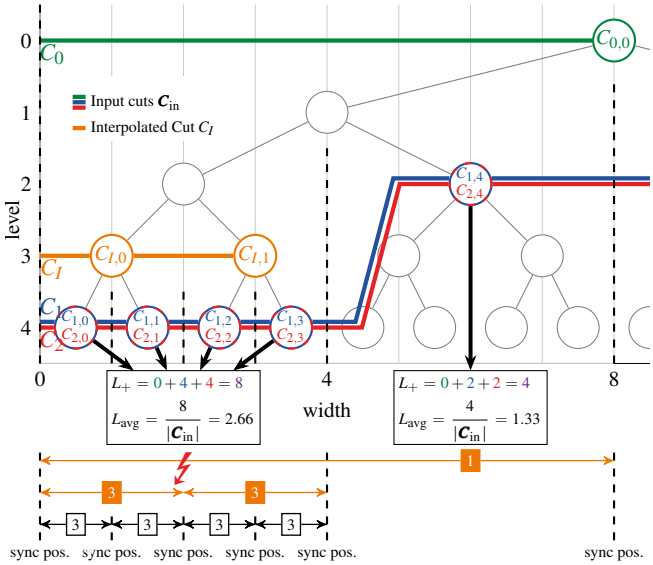

Figure 7: *Special care needs to be taken to ensure that interpolated lightcuts are always valid. The interpolation algorithm walks the hierarchy from left to right, and would first append nodes at level three to the interpolated cut C_I. However, at node $C_{1,4}$ a node of level 1 would be appended, which would reach into the previously handled subtree resulting in an invalid cut that violates Eq. 1.*

they are about to swap their rank, or the least important one is exchanged with a different cache record. Before the w'_i are used for interpolation, the weights are normalized so that they sum to one. Final weights are shown in Fig. 4 (middle).

Note that the interpolation also works if the shading point lies outside the triangle of selected cache records or if it does not lie in the same plane. If a shading point has no valid cache records, we fall back to calculating a lightcut using the original algorithm, but this happens only for a *very* negligible fraction of shading points.

3.3. Interpolation Algorithm

We have determined the set of input lightcuts $\boldsymbol{C}_{\text{in}}$ and their associated weights for interpolation. Now, we iterate over them in unison and find corresponding nodes that intersect the same *leaf-root paths* (see Fig. 1 (right)). The heights of these nodes are then interpolated, and a valid lightcut is constructed for the current shading point.

Terminology Note that for *simplicity in the explanation* we will assume a full binary tree, i.e. all leaves are at the same level and all nodes have two children—our algorithm works for arbitrary trees. Depending on its level, a node spans over a number of leaf nodes in the light hierarchy; we will refer to this number as the node's *width* $W(n)$ (see Fig. 1 (right)). We require that the nodes of all input cuts are iterable from left to right (as shown in our illustrations). We further define the *cut width* $W(0..p(C))$ as the sum of the widths of all nodes in a cut C up to the node at index $p(C)$. Note that the width of a complete cut at its last node equals the number of leaves in the hierarchy, as a cut always spans the whole hierarchy.

Finding Corresponding Nodes To interpolate the levels of the nodes of the input cuts $\boldsymbol{C}_{\text{in}}$, we need to find corresponding sets of nodes from the input cuts. For this, we walk through the width of the hierarchy once, from the leftmost to the rightmost leaf, with a step size equal to the smallest node width from any of the input cuts covering the current leaf. This is equivalent to walking along the lowest edge of the input cuts (nearest to the leaf nodes), see Fig. 1 (right). As cut nodes are sorted from left to right, this can be achieved with a single combined iteration over the input cuts, see List. 1.

During traversal we maintain for each input cut C_j the pointer $p(C_j)$ to the current node as well as the current width $W(0..p(C_j))$. To walk along the lowest edge we choose the cut C_S with the smallest (next) node, i.e. $W(0..p(C_S)) \leq W(0..p(C_j)) \;\; \forall C_j \in \boldsymbol{C}_{\text{in}}$. After each step, we test whether the current node of any other input cut ends at the same width, i.e. $\exists C_j \in \boldsymbol{C}_{\text{in}} : \;\; W(0..p(C_S)) = W(0..p(C_j))$. This is a *sync position*, where the lowest input cut C_S can switch between input cuts. More importantly, at this point we *trigger interpolation* (see below) for the nodes that were just traversed. We iterate until all nodes in the input cuts have been processed, i.e. the width of the interpolated cut equals that of the hierarchy: $W(0..p(C_I)) = W(N(0,0))$.

Interpolation Now we have to a) determine the interpolated levels of the output nodes, and b) ensure that the output cut is valid. We compute the average node level at every node of the lowest cut (while finding corresponding nodes), and at sync positions we try to emit new nodes of the interpolated cut.

Formally, the average level above a leaf node is computed from the input cuts $\boldsymbol{C}_{\text{in}}$, where w_j is the interpolation weight of C_j, and

$L(C_j(i))$ is the level of cut C_j above the i-th leaf node:

$$L_{\text{avg}}(i) = \sum_{C_j \in \boldsymbol{C}_{\text{in}}} w_j \cdot L(C_j(i)).$$

Naively, the interpolated, *not yet valid* cut C_I then would be:

$$C_I = \{N(i, \|L_{\text{avg}}(i)\|) : i \in \{0..W(N(0,0))\}\},$$

where $N(i,l)$ is the node at width i and level l of the hierarchy. However, this cut can contain nodes whose subtrees overlap, whereas in a valid cut each leaf-root path intersects exactly one node. Formally, let $C_I(i)$ be the node from C_I above the i-th leaf, then, in a valid cut:

$$\forall i \in \{0..W(N(0,0))\} \quad |C_I(i)| = 1. \tag{1}$$

Ensuring Validity of Cuts To avoid creating invalid cuts containing overlapping nodes using the naive interpolation scheme described above (see also List. 1), at each sync position we perform the following: we compute the average of the interpolated levels of the nodes examined since the preceding sync position (weighted by the number of corresponding leaves), see Fig. 6. This is the level of the nodes we insert into the output cut. We defer the output of a node if its width would exceed the current sync position; the output level for this subtree could still change as it is not fully explored yet. This can happen if one of the input cuts has a high level so that the interpolated node is above the current node (see Fig. 6, far right).

Furthermore, we also must not insert nodes whose subtrees overlap with those of nodes written previously. Fig. 7 shows an example, where the interpolation would result in such a node. In these cases, we lower the level of the output nodes until no overlap happens. This results in more refined cuts and thus is a conservative strategy for rendering. The pseudo code is given in List. 2.

Formally, the aforementioned algorithm leads to a different definition of the interpolated level: every node $C_{I,k}$ in the interpolated cut needs to be at the average level of *all* nodes from the input cuts that it covers:

$$L(C_{I,k}) = \sum_{i=W(0..C_{I,k-1})+1}^{W(0..C_{I,k})} \frac{L_{\text{avg}}(i)}{W(0..C_{I,k}) - W(0..C_{I,k-1})}.$$

4. Implementation

Our implementation of both the reference Lightcuts and our lightcut interpolation is fully GPU-based. All significant parts of the per-frame algorithm are implemented in CUDA, and we use OptiX Prime for BVH building and raycasting. Because of these constraints, some parts of the Lightcuts algorithm on the GPU had to be changed subtly. In this section we will highlight these and other important implementation details.

Lightcuts Each lightcut can contain up to 1024 nodes, and one ray per node per shading point needs to be traced. As OptiX Prime is dispatched on lists of input rays only, and GPU memory is limited (4 GB), we had to change the Lightcuts implementation scheduling to work in tiles over the image space. We set the tile size as large as possible within the memory constraints of our GPU (typically 128^2). Tiles are dispatched separately and we traverse the light hierarchy for each shading point, write its lightcut and shadow rays to pre-allocated memory. The rays are then traced using OptiX Prime and the visibility is encoded into the first bit of the lightcuts nodes' indices to reduce the number of memory reads when the lightcut is evaluated for the shading point.

Building the Light Hierarchy Originally, a greedy bottom-up approach was suggested in Lightcuts [WFA*05], but, as expected, we

```
p(C_I) = −1; p(C_j) = −1   ∀C_j ∈ C_in
l_inter = 0; w_interp = 0
C_S = C_0
while W(0..p(C_I)) ≤ W(N(0,0)):
    # advance all synced cuts
    p(C_j)+=1   ∀ C_j ∈ {C_k : W(0..p(C_k)) = W(0..p(C_S)), C_k ∈ C_in}
    C_S = argmin_{C_j ∈ C_in} W(0..p(C_j)) # smallest cut
    C_N = argmin_{C_j ∈ C_in \ C_S} W(0..p(C_j)) # next smallest cut

    # to next sync pos.
    while W(0..p(C_S)) < W(0..p(C_N)):
        w_interp+=W(p(C_S)) # covered width
        l_inter+=Σ_{C_j ∈ C_in} w_j · l(p(C_j)) # sum of levels
        p(C_S)+=1
    # interpolate once in any case
    w_interp+=W(p(C_S))
    l_inter+=Σ_{C_j ∈ C_in} w_j · l(p(C_j))

    # trigger interpolation, see List. 2
    interpolate(w_interp, l_inter)
```

Listing 1: *Finding corresponding nodes algorithm.*

```
def interpolate(w_interp, l_inter):
  global l_0 = 0
  # write nodes up to current sync pos.
  while W(0..p(C_I)) ≤ W(0..p(C_S)):
    l = l_inter / w_interp
    # l_0 is either pointing at right child or start of next subtree
    # if level of next node is lower, then it must be a left node
    lastleft = l > l_0
    if !lastleft:
      l = l_0
    n = N(W(0..p(C_I)), l)
    # do not interp. if C_I would advance beyond current sync pos.
    if W(0..p(C_I)) + W(C_{I,i+1}) > W(0..p(C_S)):
      break
    C_{I,i+1} = n; p(C_I)+=1
    evaluate(C_{I,i})
    if lastleft: l_0 = l_inter
    else: l_0 =level_of_next_subtree(C_{I,i})
```

Listing 2: *Interpolation algorithm.*

found this to be very slow. Instead we use a simple top-down object-median longest-axis approach [WFA*05]. As with BVH-building, many extensions to accelerate the building process exist (see 2), however, these are orthogonal to our work.

Lightcut Interpolation After lightcuts creation for the cache records and tracing shadow rays, we determine the most similar cache records for each shading point. We examine nearby cache records using a simple image space grid acceleration structure, apply the weight function (see Sec. 3.2), and write the indices and weights of the best cache records to GPU memory. These tuples are then sorted by the indices of the cache records, so that later kernels run as coherent as possible. Next, another kernel starts to interpolate and either immediately shades if visibility is interpolated, or else outputs the cut for visibility resolving and subsequent shading. If shading points did not find suitable cache records (weights below a user-defined threshold, i.e. ε), we calculate full lightcuts for them using the original Lightcuts algorithm.

5. Comparison to Other Methods

Our algorithm is related to *Nonlinear Cut Approximation* (NCA) [CPWAP08], but we make several essential additions and improvements. First, NCA is tailored for visibility cuts; it produces an interpolated value for each node on the lowest edge of the input cuts, i.e. the result contains overlapping nodes and further means would be required to generate valid output cuts. As an input to their algorithm they require indices in post order (hierarchy traversal order), which might be problematic if other algorithms working on the cuts (e.g. Lightcuts) are better suited for other traversal orders.

Our method reads only the *level* of each input node and thus does not share this limitation. Also for each input node NCA requires to load two indices in addition to the value that is to be interpolated, which is at least twice the amount of data that needs to be read from memory compared to our method (and which can also be compressed much worse than the level alone). Generally, traversal order and memory I/O is very important and we found that the performance of our GPU-implementation scales approximately linearly with the number of memory reads per interpolation step. Lastly, NCA saves the input cuts at the vertices of the input geometry. Lightcut Interpolation works in image space only and does not require any explicit mesh representation.

(Ir-)Radiance Caching (see Sec. 2) convolves contributions of emitters (here VPLs) in the upper hemisphere of a cache record with their visibility. Thus only directly visible emitters are represented in the record. In contrast a lightcut does not pre-convolve with visibility and represents VPLs also from the lower hemisphere (albeit not refined very much). We directly interpolate the nodes in the cut, and then evaluate each node at a shading point (optionally we interpolate visibility along with the nodes' levels).

While *Reconstruction Cuts* (see Sec. 2) can discard nodes in the input lightcuts quickly if a node's contribution is zero, Lightcut Interpolation needs to create full, valid lightcuts that cover each leaf of the hierarchy. Thus, we have to iterate once over all nodes in the input cuts, even if they have no contribution. However, Reconstruction Cuts does traverse the hierarchy of directional lights for every

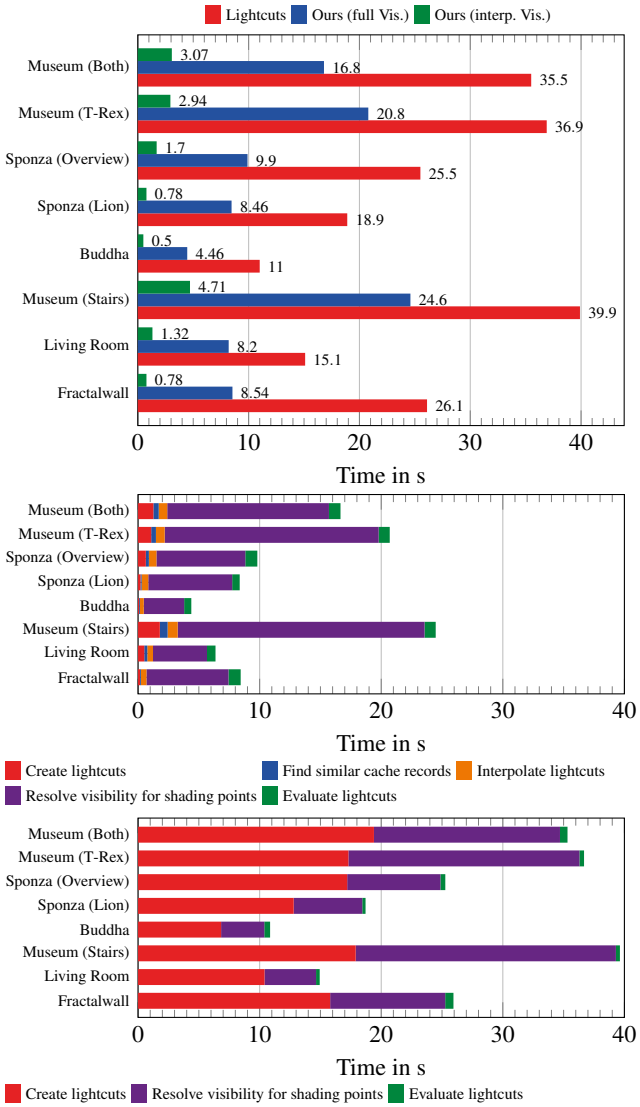

Figure 8: ***Top:*** *Performance of our method compared to Lightcuts in diffuse scenes. Lightcut Interpolation needs only 4.50-8.32% (interp. Vis.) resp. 40.4-56.4% (full Vis.) of the time to compute an image.* ***Middle:*** *Performance Breakdown of the individual stages of Lightcut Interpolation with full visibility. Shooting rays for shading points dominates the costs.* ***Bottom:*** *Performance Breakdown of the individual stages of Lightcuts. Hierarchy traversal makes for approx. half the cost even in complex scenes.*

shading point, and in general requires to read *a lot* of data from memory. As GPUs are often memory-I/O-bound, this complicates a fast GPU implementation.

When *merging cuts conservatively*, emitting the lowest nodes of all input cuts, perceivable discontinuities arise where the influence of cache records ends, due to an abrupt change between neighboring cuts. Interpolation ensures that cuts always change smoothly.

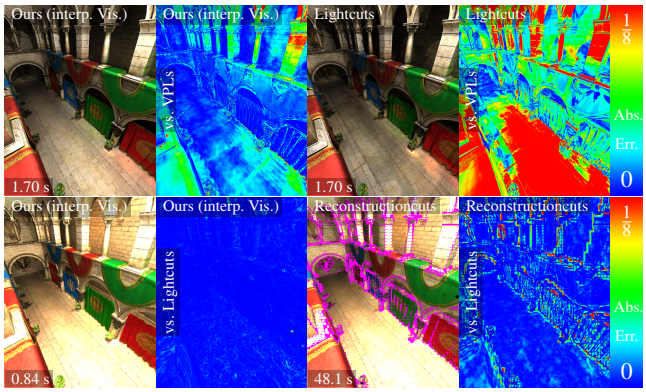

Figure 9: ***Top:*** *Original Lightcuts and Lightcut Interpolation (ours) for the* SPONZA (OVERVIEW) *scene. The refinement threshold is tuned to yield an equal-time comparison. Our method captures indirect illumination with only minor artifacts. Lightcuts cannot refine cuts sufficiently and indirect illumination is almost completely off.* ***Bottom:*** *Lightcut Interpolation compared to Reconstruction Cuts (no visibility or shadow rays in our prototype). Pixels where a method cannot create a cut are marked in magenta, and are treated as zero error in the difference image, see Sec. 6.*

6. Results and Discussions

All images were rendered with 10^6 VPLs on an Nvidia GTX 980 (4 GB RAM), with a resolution of 1024^2 unless stated otherwise.

Fig. 10 shows the scenes we used for comparisons. Scenes either use completely diffuse materials, or force a Phong exponent of 150. We compare to both a naive VPL implementation [Kel97] (far left) and Lightcuts [WFA*05]; the latter is also compared to VPLs (second and third columns). For Lightcuts, we use 1% of the estimated contribution as the relative threshold. We show results of Lightcut Interpolation (rightmost three columns) with *full visibility* (shadow rays are shot for every node in a shading point's lightcut, top left of the images), and *interpolated visibility* (results of shadow tests are interpolated from nodes of the input cuts, bottom right). The comparison images show the absolute error in the output radiance. The false-color mapping scale is the same for all images. We also include timings, showing that our method with interpolation of lightcuts and visibility is typically more than an order of magnitude faster, while introducing little approximation error.

Full Visibility In **diffuse scenes**, Lightcut Interpolation (full Vis.) introduces a minor bias compared to Lightcuts in areas with high-frequency details and geometric discontinuities. This can be observed in SPONZA (LION) on the small rounded discontinuities of the lion head, or in MUSEUM (BOTH) where the back wall near the windows has many small discontinuities.

In most scenes, Lightcuts itself introduces large errors compared to VPLs, especially in bright areas where its cut refinement allows for a large error per node (see Sec. 2). Other authors have also observed that Lightcuts can introduce non-negligible error [OP11, HWJ*15]. The additional bias of our method is comparably insignificant.

With **glossy materials**, Lightcut Interpolation introduces slightly larger bias, especially on strongly curved surfaces. FRACTALWALL (highly tessellated and displaced wall of a Cornell box) is a pathological case: here our method shows consistent bias on most curved surfaces, but the resulting deviations are small and only apparent in the difference images. The original Lightcuts shows less bias, mostly in the glossy highlights.

In the MUSEUM (T-REX) scene the interpolation introduces larger errors in the bright glossy highlights on the T-Rex head. Lightcuts also consistently shows very large errors at these highlights. Compared to VPLs, it is hard to distinguish our method from Lightcuts. This trend can be observed in all glossy results: Lightcut Interpolation introduces artifacts where Lightcuts also seems to introduce bias, especially at glossy highlights.

A performance and a timing breakdown is shown in Fig. 8. Lightcut Interpolation takes only 40.4-56.4% (diffuse) resp. 40.5-57.6% (glossy, not shown) of the time of original Lightcuts. It is obvious that per shading point shadow ray casting dominates the rendering costs. Even in the worst case, MUSEUM (STAIRS), all other steps take less than 18% of total time—the *actual interpolation itself takes less than 2.5%* of total render time. This shows the success of our method in reducing lightcut creation cost.

Interpolated Visibility As an alternative to shadow ray casting, we also evaluate interpolating visibility from the input cuts' nodes. Unsurprisingly, this introduces some bias where radiance changes rapidly, e.g. near geometry discontinuities; sometimes to a lesser degree also on almost planar spaces. The BUDDHA scene with full visibility has almost no perceivable error. Compared to this, the additional artifacts from interpolated visibility can be observed particularly at the breasts and the folds of the kimono. In SPONZA (OVERVIEW) the bias is visible in the indirect shadow below the banderoles and the curtains on the right. Insufficiently sampled indirect shadows are most obvious in the LIVING ROOM scene, where the shadows of the vase and pictures on the cupboard are cast by indirect light. These artifacts can sometimes be perceived even without comparison images. Also, the small trims on the cupboards and small detail of the accessories in the LIVING ROOM scene seem to be missed completely, which might be caused by an undersampling of the hemisphere of cache records for the cache radius [TL04, Ulb15].

The impact of the visibility interpolation artifacts is amplified by the cache record placement which ignores the actual radiance distribution, and thus does not increase the density of cache records where illumination changes rapidly [Ulb15]. Any cache record distribution scheme that takes radiance into account (e.g. a fully converged radiance cache distribution) could reduce the error significantly.

Lightcut Interpolation with interpolated visibility is almost interactive in moderately complex scenes, see Fig. 8 (top). Rendering an image takes only 4.50-8.32% (diffuse), and 4.84-11.7% (glossy, not shown) of the time of original Lightcuts. With reduced resolution, it can be used interactively, e.g. to position the camera or to preview material and shading. Interpolating visibility during lightcut node level interpolation is very cheap, but of course shadow rays have to be cast for the nodes in the lightcuts of the cache records, which can take up to 2 seconds in complex scenes.

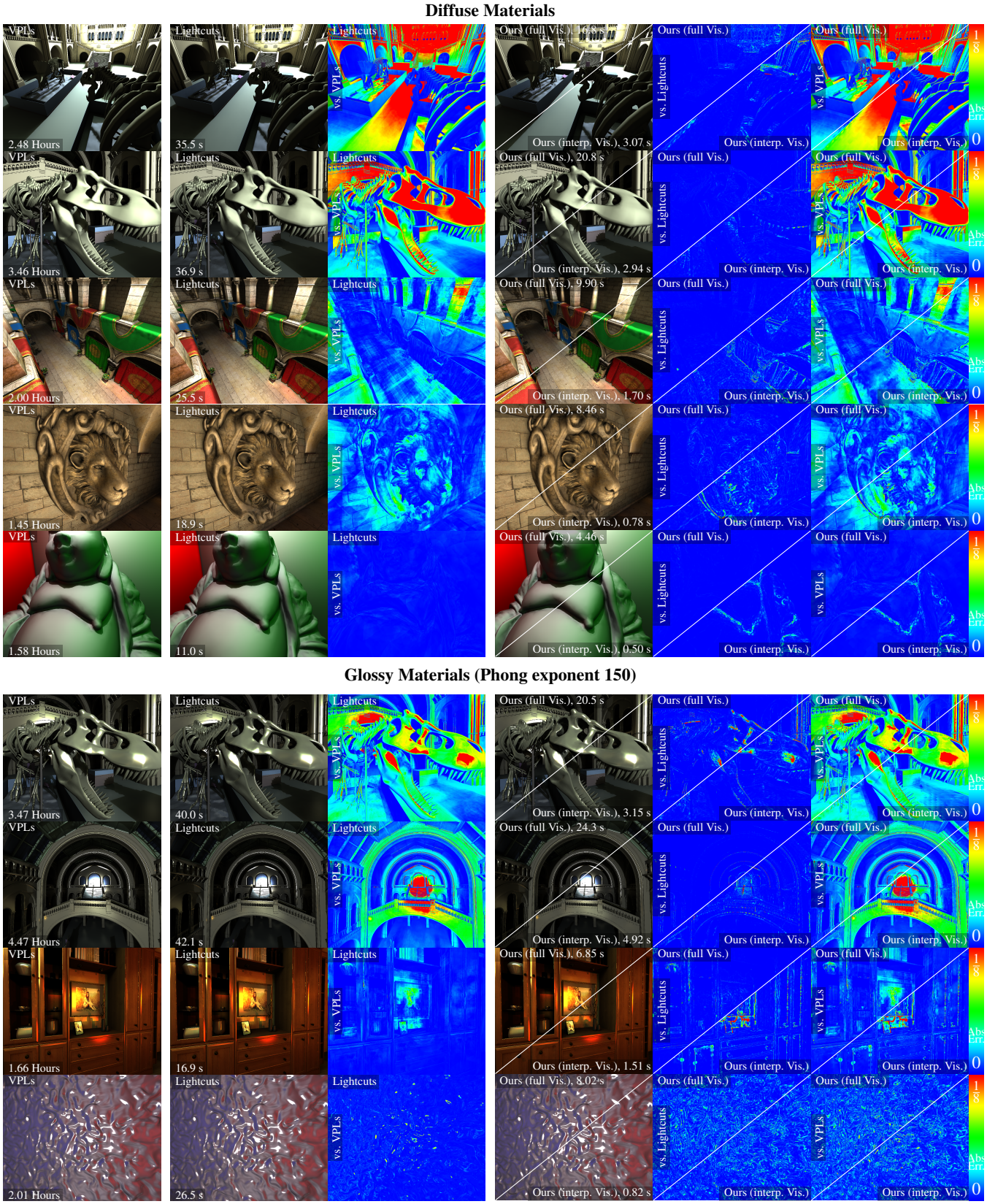

Figure 10: *Lightcut Interpolation (ours) compared to VPLs and Lightcuts in both diffuse and glossy (Phong exponent 150) scenes. False color images show absolute error compared to VPLs and/or Lightcuts. From top to bottom:* MUSEUM (BOTH), MUSEUM (T-REX), SPONZA (OVERVIEW), SPONZA (LION), BUDDHA, MUSEUM (T-REX), MUSEUM (STAIRS), LIVING ROOM, FRACTALWALL.

Equal Time Fig. 9 shows an equal time comparison between Lightcut Interpolation (interp. Vis.) and Lightcuts, where its refinement threshold is tuned to yield an equal-time comparison. It is obvious that even for previews Lightcuts cannot capture illumination convincingly with these extreme settings, while our method produces an image that represents the lighting in the SPONZA (OVERVIEW) scene well, despite some artifacts below the curtains.

Lightcut Interpolation vs. Reconstruction Cuts We implemented a prototype of Reconstruction Cuts (see Sec. 2) for experiments and comparisons. Our prototype runs on the GPU, but does not support visibility interpolation and omits some of the strategies to adapt to high-frequency geometry, but instead simply masks difficult pixels. These are then ignored in difference images. In Fig. 9 we can see that Lightcut Interpolation has less bias than Reconstruction Cuts in all regions, and is more than 50× faster than our prototypical Reconstruction Cuts implementation. This is not surprising, as Reconstruction Cuts does not eliminate hierarchy traversal and requires many memory reads per shading point, see Sec. 5.

7. Conclusion and Future Work

In this paper we presented a novel method for exploiting the coherence in lightcuts [WFA*05]. In contrast to previous work, our method is not based on clustering shading points, but instead computes lightcuts at image space cache records and interpolates the levels of their nodes for all shading points. This enables an efficient, GPU-friendly preview rendering of global illumination computed from a large number of (virtual) light sources.

There is strong potential to accelerate Lightcut Interpolation further. During cut interpolation, it is easy to check where the input lightcuts differ too much. Then, it would be possible to simply refine one of the input lightcuts for these nodes using a partial hierarchy traversal, yielding reference lightcuts. If *radiance* of cache records differs too much, additional shadow rays could be shot where the input nodes' visibility differs. Lastly, with a temporally stable distribution of cache records, lightcuts for the cache records could be saved and reused across frames to save the substantial cost of creating the cache records' lightcuts.

References

[AUW07] AKERLUND O., UNGER M., WANG R.: Precomputed visibility cuts for interactive relighting with dynamic BRDFs. In *Computer Graphics Forum (Proc. Pacific Graphics)* (2007), pp. 161–170.

[CPWAP08] CHESLACK-POSTAVA E., WANG R., AKERLUND O., PELLACINI F.: Fast, realistic lighting and material design using nonlinear cut approximation. *ACM Trans. on Graphics (Proc. SIGGRAPH Asia) 27*, 5 (2008), 128:1–128:10.

[DBD08] DE BODT T., DUTRÉ P.: *Coherent Lightcuts*. Tech. rep., Katholieke Universiteit Leuven, 2008.

[DGS12] DAVIDOVIČ T., GEORGIEV I., SLUSALLEK P.: Progressive lightcuts for GPU. In *ACM SIGGRAPH 2012 Talks* (2012), p. 1.

[DKH*13] DACHSBACHER C., KRIVÁNEK J., HASAN M., ARBREE A., WALTER B., NOVÁK J.: Scalable realistic rendering with many-light methods. *Computer Graphics Forum* (2013).

[HMS09] HERZOG R., MYSZKOWSKI K., SEIDEL H.-P.: Anisotropic radiance-cache splatting for efficiently computing high-quality global illumination with lightcuts. *Computer Graphics Forum (Proc. of Eurographics) 28*, 2 (2009), 259–268.

[HPB07] HAŠAN M., PELLACINI F., BALA K.: Matrix row-column sampling for the many-light problem. *ACM Trans. on Graphics 26*, 3 (2007), 26.

[HWJ*15] HUO Y., WANG R., JIN S., LIU X., BAO H.: A matrix sampling-and-recovery approach for many-lights rendering. *ACM Trans. on Graphics* (2015).

[KBPZ06] KRIVÁNEK J., BOUATOUCH K., PATTANAIK S. N., ZARA J.: Making radiance and irradiance caching practical: Adaptive caching and neighbor clamping. *Proc. Eurographics Symposium on Rendering* (2006), 127–138.

[Kel97] KELLER A.: Instant radiosity. In *SIGGRAPH '97* (1997), pp. 49–56.

[KGPB05] KRIVÁNEK J., GAUTRON P., PATTANAIK S., BOUATOUCH K.: Radiance caching for efficient global illumination computation. *IEEE Trans. on Visualization and Computer Graphics 11*, 5 (2005), 550–61.

[OBA12] OLSSON O., BILLETER M., ASSARSSON U.: Clustered deferred and forward shading. In *Proc. of ACM SIGGRAPH / Eurographics conference on High Performance Graphics* (2012), pp. 87–96.

[OP11] OU J., PELLACINI F.: Lightslice: matrix slice sampling for the many-lights problem. *ACM Trans. on Graphics (Proc. SIGGRAPH Asia) 30*, 6 (2011), 179.

[RZD14] REHFELD H., ZIRR T., DACHSBACHER C.: Clustered preconvolved radiance caching. In *Proc. of Eurographics Symposium on Parallel Graphics and Visualization* (2014).

[SJJ12] SCHWARZHAUPT J., JENSEN H. W., JAROSZ W.: Practical hessian-based error control for irradiance caching. *ACM Trans. on Graphics (Proc. SIGGRAPH Asia) 31*, 6 (2012), 193.

[SKS02] SLOAN P.-P., KAUTZ J., SNYDER J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. on Graphics 21*, 3 (2002), 527–536.

[SNRS12] SCHERZER D., NGUYEN C. H., RITSCHEL T., SEIDEL H.-P.: Pre-convolved radiance caching. *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering) 4*, 31 (2012).

[TL04] TABELLION E., LAMORLETTE A.: An approximate global illumination system for computer generated films. *ACM Trans. on Graphics 23*, 3 (2004), 469–476.

[Ulb15] ULBRICH J.: *Efficient Screenspace Distribution of Irradiance Cache Records by Analyzing and Evaluating Existing Error Metrics*. Master's thesis, Karlsruhe Institute of Technology, Germany, 2015.

[WBKP08] WALTER B., BALA K., KULKARNI M., PINGALI K.: Fast agglomerative clustering for rendering. In *IEEE Symposium on Interactive Ray Tracing* (2008), pp. 81–86.

[WFA*05] WALTER B., FERNANDEZ S., ARBREE A., BALA K., DONIKIAN M., GREENBERG D. P.: Lightcuts: a scalable approach to illumination. *ACM Trans. on Graphics (Proc. SIGGRAPH) 24*, 3 (2005), 1098–1107.

[WH92] WARD G. J., HECKBERT P.: Irradiance gradients. In *Proc. Eurographics Workshop on Rendering* (1992), pp. 85–98.

[WHY*13a] WANG R., HUO Y., YUAN Y., ZHOU K., HUA W., BAO H.: GPU-based out-of-core many-lights rendering. *ACM Trans. on Graphics (Proc. SIGGRAPH Asia) 32*, 6 (2013), 210:1–210:10.

[WHY*13b] WANG R., HUO Y., YUAN Y., ZHOU K., HUA W., BAO H.: GPU-based out-of-core many-lights rendering. *ACM Trans. on Graphics (Proc. SIGGRAPH Asia) 32*, 6 (2013), 210:1–210:10.

[WRC88] WARD G. J., RUBINSTEIN F. M., CLEAR R. D.: A ray tracing solution for diffuse interreflection. *Computer Graphics (Proc. SIGGRAPH) 22*, 4 (1988), 85–92.

[WWZ*09] WANG R., WANG R., ZHOU K., PAN M., BAO H.: An efficient GPU-based approach for interactive global illumination. *ACM Trans. on Graphics 28*, 3 (2009), 91:1–91:8.

[WXW11] WANG G., XIE G., WANG W.: Efficient search of lightcuts by spatial clustering. In *SIGGRAPH Asia Sketches* (2011), p. 26.