Eurographics Symposium on Rendering 2011
Ravi Ramamoorthi and Erik Reinhard
(Guest Editors)

Volume 30 (2011), Number 4

ManyLoDs: Parallel Many-View Level-of-Detail Selection for Real-Time Global Illumination

Matthias Holländer[1] Tobias Ritschel[1,2] Elmar Eisemann[1] Tamy Boubekeur[1]

[1]Telecom ParisTech - CNRS/LTCI [2]Intel Visual Computing Institute

Abstract

Level-of-Detail structures are a key component for scalable rendering. Built from raw 3D data, these structures are often defined as Bounding Volume Hierarchies, providing coarse-to-fine adaptive approximations that are well-adapted for many-view *rasterization. Here, the total number of pixels in each view is usually low, while the cost of choosing the appropriate LoD for each view is high. This task represents a challenge for existing GPU algorithms. We propose ManyLoDs, a new GPU algorithm to efficiently compute many LoDs from a Bounding Volume Hierarchy in parallel by balancing the workload within and among LoDs. Our approach is not specific to a particular rendering technique, can be used on lazy representations such as polygon soups, and can handle dynamic scenes. We apply our method to various many-view rasterization applications, including Instant Radiosity, Point-Based Global Illumination, and reflection / refraction mapping. For each of these, we achieve real-time performance in complex scenes at high resolutions.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Shading I.3.1 [Computer Graphics]: Hardware Architecture—Parallel Processing I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Object hierarchies

Keywords: multi-view, many-view, real-time, level-of-detail, GPU

1. Introduction

Level-of-Detail (LoD) algorithms [LRC*02] are a necessity for efficient rendering techniques that seek to depict today's complex and ever-growing virtual worlds. Besides direct rendering of geometry into the view of a virtual observer, there exists a range of techniques that require rendering the scene from many additional views, e. g. into classic shadow or reflection maps. While LoDs are well-understood for direct rendering and often computed incrementally [XV96, Hop96], the novelty of our approach lies in its specific target of many-view rasterization, which means that not one but *many* LoDs have to be extracted concurrently. Our ManyLoDs approach renders a high number of views using a fast LoD extraction algorithm that is designed to fit modern parallel graphics hardware (GPUs). Typical examples are Instant Radiosity or Point-Based Global Illumination (GI), where the number of views can easily reach many thousands each of them requiring a specific LoD. For such applications, most principles of current GPU LoD techniques are contradicted; the rasterization cost is comparatively low (only few pixels are actually drawn), but the cost for selecting the various LoDs for each view is high. Our basic idea to address such *many-view* problems is to exploit fine-grained parallelism to progressively define many cuts in a tree, where each cut corresponds to the LoD of a particular view.

The parallelized many-view LoD is achieved by dynamically creating many threads based on a small-scale iterative data-amplification mechanism (e. g. via the geometry shader's stream output). To preserve a balanced workload, we can limit the action of each thread to a 1-edge walk (either up or down) in the tree structure. We demonstrate our approach on a classical point tree based on a Bounding Sphere Hierarchy (BSH) and apply it to several rendering techniques, including adaptive point-based rendering, Instant Radiosity, Point-Based GI and reflection / refraction mapping. With our many-view cuts, we make the following contributions:

- Fine-grained parallel LoD selection for a large number of views;
- Adaptation of incremental and lazy update schemes to many-view problems;

2. Previous Work

A large number of techniques cover LoDs [LRC*02] for 3D shapes and aim at representing and rendering geometric data, with a target amount of available time and / or memory. In this section, we focus on recent Hierarchical Level of Detail (HLoD) techniques based on trees and their applications to GI methods.

Hierarchical Data Structures. Hierarchical space subdivision structures [Ben75, JT80, FKN80] – and particularly Bounding Volume Hierarchies (BVHs) (e. g. using spheres) [Hub93] – are often used to represent different LoDs of a 3D shape. The idea is to define leaves of a spatial tree as geometric samples (points, vertices) and inner nodes as an approximation of their subtree (coarser polygons, sparser point sets). The construction can be done in a top-down manner by recursively splitting the shape's bounding volume, or bottom-up, by successively merging neighboring elements into larger, i. e. upper, internal nodes.

Mesh-based LoDs [Hop96] including frame coherence [XV96] have recently been implemented effectively on GPUs [HSH09]. However, a number of rendering techniques work even better using a much simpler, *point-based* structure.

Point-based HLoDs can be generated from unorganized point clouds without explicit connectivity information by storing point sets at various resolutions directly as internal tree nodes and leaves. In particular, QSplat [RL00] uses a BSH organized as a binary tree where leaf nodes store surfels [PZvBG00] covering the surface (point, normal and optionally color samples), while internal nodes store representative bounding spheres and normal cones of their respective subtree. Progressive visualization is possible by a coarse-to-fine level extraction. While QSplat was originally developed for out-of-core CPU execution, alternative HLoDs have been made in-core parallel for GPU execution via early GPU programmability [DVS03]. These so-called sequential point trees can eventually be used in an out-of-core context, by decomposing a large point set into a forest of such trees defined as leaves of a coarse-grain out-of-core octree [WS06]. Hybrid techniques have also been introduced to combine point-based LoDs with mesh representations at finer scale, either in-core [BRS05] or out-of-core [GM05]. Ultimately, a given LoD is defined as a view-dependent *cut* in the underlying HLoD structure (see Fig. 1).

One contribution of this paper is to put these existing ideas of LoD selection algorithms into the context of GI.

Points Points are a well-studied rendering primitive often resulting from a 3D capture process by a laser scanner, stereovision or from a (re-)sampling process applied to a digital scene. The resulting high density point cloud usually consists of tens or hundreds of millions of points [KRG*00]. Connectivity information is not always available and not even required for a convincing visual representation [ZPVBG01].

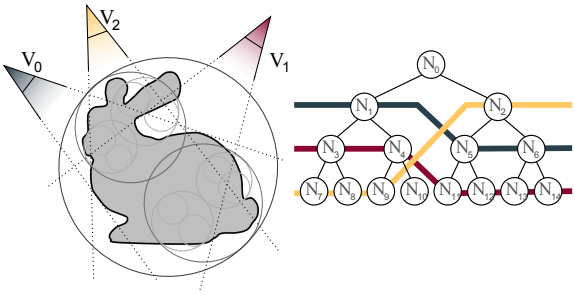

Figure 1: *Three cuts (blue, red and yellow line) for three views V1, V2 and V3 on the scene (Bunny) contained in its Bounding Volume Hierarchy.*

An efficient data structure for rendering such a high number of points is crucial and HLoDs are a natural match to this challenge.

Point Sampled LoD for Global Illumination. The use of point sets for interactive GI has recently gained attention: Virtual Point Lights (VPLs) [Kel97] have inspired many economic approximations for GI techniques. Laine et al. [LSK*07], assume neglectable motion, which allows the reuse of VPLs over time, Hašan et al. do so by temporal clustering [HVAPB08].

Many GI methods rely on the ability to render many views of the scene from various locations. Consequently, for such many-view rasterizations, the process boils down to the definition of numerous cuts in the scene's hierarchy, i. e. one for each point of view. In practice, considering a given HLoD structure and a given frame, hundreds or thousands of different cuts have to be generated.

Most HLoD methods have only focused on single-view extraction. Defining a cut sequentially – even if all cutting processes [REG*09] or collections of nodes are treated in parallel [HSH09] – does not exploit modern graphics architectures to the fullest. In particular, when all cuts have to be adaptive, a non-uniform distribution of computational workload is common. Balancing this workload is an important issue.

It turns out that many of the recent light-gathering methods rely on such a large number of views to be effective [Bun05, Chr08, REG*09], but only little work has been devoted to this task. In particular, one key problem is to define a parallel many-cut algorithm which can be balanced on a fine-grained parallel architecture *within* and *among* the views' LoD extraction. This issue motivated us to address all views at once and to generate many LoDs in parallel instead of processing views independently.

GPU LoD selection, such as in the context of intersection test acceleration [ZHWG08, LGS*09, GKCC10] and parallel link

creation for radiosity [MESD09], can be done efficiently on modern graphics architectures [EML09].

We exploit the fact that graphics pipelines are best suited for high numbers of threads (Sec. 3). Sequential-over-n and parallel-in-each-view approaches do not produce enough threads to be efficient (Sec. 6). We demonstrate how our approach can be successfully applied to smooth indirect shadows [RGK*08] and natural illumination from environment maps, among others.

3. ManyLoDs

Our algorithm computes a multi-cut in a BVH, corresponding to the many LoDs from a large number of viewpoints, using a fine-grained parallelism that fits modern GPUs. The terminology used in this paper is as follows: A BVH is a hierarchical set of nodes with a tree structure: BVH $:= \{N_i\}$. A *node-view* is a pair (N_i, V_j) corresponding to a node in the BVH and a given view. Further, we define a *cut* C through a BVH according to a criterion c, which is a scalar function of node-views, with the property
$\forall N_i, N_j, V_k : c(N_i, V_k) \leq c(N_j, V_k) \Leftrightarrow \text{level}(N_i) < \text{level}(N_j)$
that is, c decreases monotonically with increasing BVH level. In our experiments c is defined as the pixel size of node N_i in view V_j or 0 if the node is culled. The cut C is the set
$C := \{(N_i, V_j) \mid c(N_i, V_j) < \varepsilon \;\&\; c(\text{parent}(N_i), V_j) > \varepsilon\}$
where $\text{parent}(N_i)$ is the parent node of N_i and ε a user-defined threshold. We define a node-view (N_i, V_j) as *valid* iff $c(N_i, V_j) < \varepsilon$.

In the following, we will describe our algorithm for a general parallel machine which can append elements to lists, i. e. using prefix scan [Ble89] or geometry shader vertex stream output. Further, we assume that a BVH is given. The basic method (Sec. 3.1) is extended to an incremental version (Sec. 3.2) and a lazy-update approach (Sec. 3.3). Details for a geometry-shader implementation are given in Sec. 4.

3.1. Basic approach

To find all cuts through an n-ary tree of maximum height h for m views in parallel, we manage two lists I and A of node-views: List I is write-only and contains *inactive* node-views, that is, node-views which are found to be in the cut and do not require further processing. Node-views in A are considered *active* and need further processing (see Fig. 2 left side only).

Initially, we set $I = \emptyset$ to be empty and $A_0 = \{(N_0, V_0), \ldots, (N_0, V_{m-1})\}$ to contain the root node in all views. To find the appropriate LoD for all views, we perform h steps: In step $k = 1 \ldots h$ a parallel kernel is executed on all node-views $a \in A_{k-1}$. The kernel appends $a := (N_a, V_a)$ to I if it is valid. Otherwise, new node-views resulting from a splitting operation are appended to A_k, one for each child node of N_a and with the view V_a. After h steps, I contains the multi-view cut and can be used for further adaptive processing and rendering. This cut can be used as input to our incremental algorithm (next section).

3.2. Incremental Approach

Instead of starting from $I = \emptyset$ and $A_0 = \{(N_0, V_0), \ldots, (N_0, V_{m-1})\}$ we start from the resulting cut of the last frame [XV96], i. e. last frame's I is used as A_0, building on the assumption that cuts are likely to remain similar (see Fig.2, right side). This requires some modifications to our algorithm. Firstly, node-views must be merged when $(\text{parent}(N_i), V_j)$ becomes valid, e. g. the camera moves further away. Secondly, node-views that are culled must not be discarded to preserve them as a starting point for later frames, e. g. a node-view gets out of the frustum and back in again. However, we can merge culled nodes when the parent node is culled. Consequently, the new kernel performs one out of three different actions on a node-view $a \in A_{k-1}$ (mind the definition of *validity* of Sec. 3):

1. If a is valid and its parent a' is not: a is appended to I.
2. If a is valid, its parent a' is valid and a is the first child: a' is appended to A_k. Note, that none of the other children of a' are moved to I. This prevents having n copies of the same parent node-view in I.
3. If a and its parent a' are not valid, all children of a are appended to A_k.

If the scene has temporal coherence (which is the case for our applications), i. e. the geometry and the views change smoothly, the amount of work (i. e. operations to produce I) has shown to be at least one order of magnitude less when compared to a full restart of our algorithm at the root node-views.

3.3. Lazy Update Approach

When approximate cuts are sufficient, we can limit the number of iterations on A to q. For example, if $q = 1$ during each frame only a single parallel pass is made over all active node-views which is instantaneously turned into a list of inactive ones. Consequently, a node-view is either left *as is*, culled, merged with sibling node-views, or split, but the process is not repeated within a frame. Therefore, the precision of the cuts might lag behind because I contains invalid node-views. However, the algorithm is simplified drastically by limiting the number of traversed edges to q per node-view.

4. Implementation

This section provides implementation details for our algorithm regarding pre-processing and runtime. Our approach consists of two main parts. First, we build the BVH in a sequential *pre-process*. Without loss of generality, we use a binary BVH [RL00] with a bounding sphere and bounding

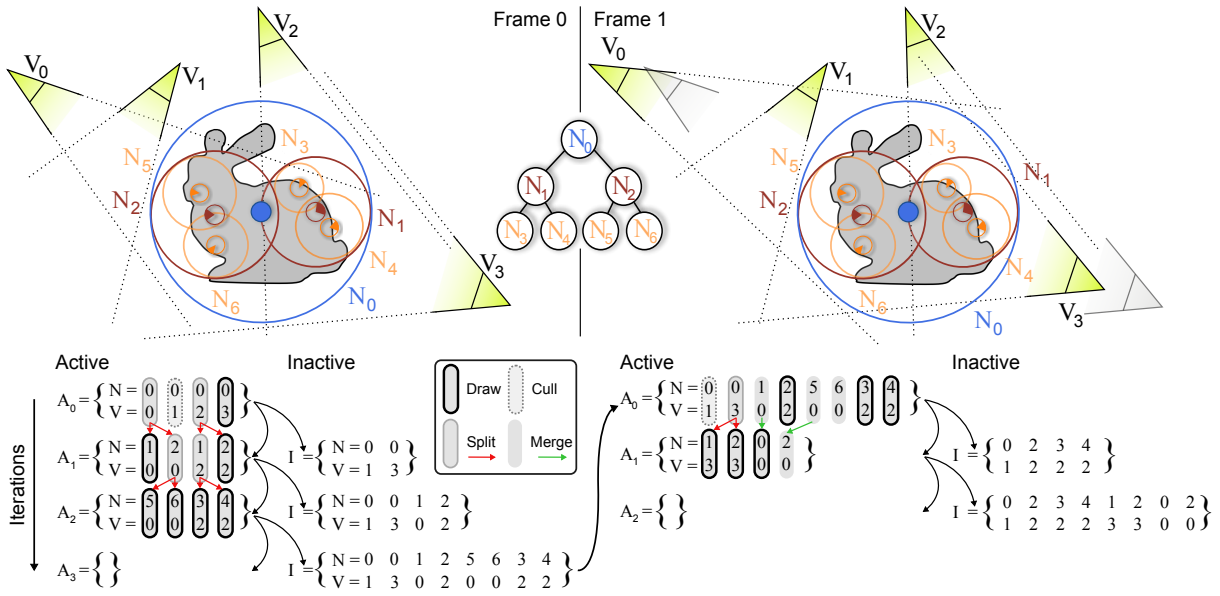

Figure 2: *Basic flow of our algorithm. In two consecutive frames (left and right) the scene geometry (Bunny) represented as a BVH (spheres) is rendered into multiple views (V1 to V4). Left: the tree is traversed using multiple iterations (vertical arrow) in parallel over all node-views. The traversal starts with the active list A_0 containing the root node N_0 for each view. In every iteration, the node-views are either culled, merged, split, or drawn directly (red and green arrows). The remaining active list is used as input for the next iteration step. Culled nodes are not removed from the list to preserve them as a starting point for future frames. After all iterations the active list A_3 is empty and the inactive list I contains the cut for all views. Right: The result of the last frame is used as input for the next frame with altered camera positions. This leads to fewer iteration steps (incremental version, see Sec. 3.2).*

cone for each node bounding the positions and the normal field of its subtree. We use a perfect and complete tree to simplify our implementation. Second, for each frame at *runtime* the BVH is updated once, if necessary, and the multi-cut is computed.

4.1. Pre-process

First, we sample the scene's surface uniformly into a set P of $n = 2^d$ points. To transfer scene deformations to the BVH [RGK*08], each point $p \in P$ is expressed in local coordinates $p = (s,t,i_{\text{tri}})$, referencing the barycentric coordinates $(s,t,u := 1-s-t)$ of p on triangle i_{tri}.

Second, we build the topology of a complete binary tree of height $d+1$ from P in k steps, during which we virtually split the set of nodes by re-ordering elements in P. To avoid sorting in every step of iteration, we use the approach of Wald and Havran [WH07]. Here, all nodes are sorted once for each dimension and then a divide-and-conquer approach is applied on these three sorted lists to construct the tree in $O(N \log N)$. Step $k \in [0 \ldots d+1]$ considers 2^k subsequences Q_j of length 2^{d-k}. For each Q_j, we sum up the volume of the left and the right sequence based on the x-, y- and z-median element. Then, we split Q_j into two sequences by selecting the splitting plane which gives the minimal summed volume. Both resulting sets become nodes in the next step.
Each resulting node $N_0 \ldots N_{2^{d+1}-1}$ stores a sphere, bounding the positions, and a cone, bounding the normals, of all nodes below. While this tree is admittedly simple (for a discussion see Sec. 7), it is very easy to update in every frame and has a small memory footprint due to its implicit structure.

4.2. Runtime

During each frame, we update the BVH (in case of dynamic scenes) before computing the multi-cut.

Update The BVH update is performed in a bottom-up manner. For every sample point (s,t,i_{tri}), i. e. every leaf node, we start a thread and update the point positions using the barycentric coordinates (s,t) and triangle i_{tri}. This essentially transfers the deformation from the scene polygons to the BVH [RGK*08].
To update the bounding information of the remaining nodes, we proceed in a bottom-up order and start threads for all nodes of a level. Each thread merges the bounds of its children. This is similar to a 1D mipmap construction, but with special merging rules for bounding spheres and bounding

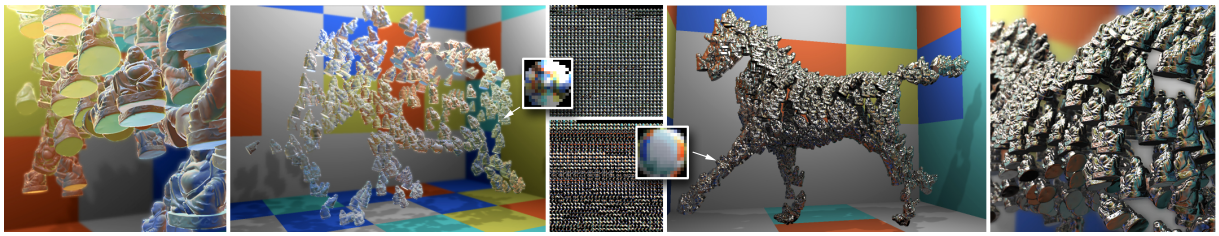

Figure 3: *Rendering of 400 refracting (left) and reflecting (right) objects (2 M polygons, 2 M points), animated in the shape of an animated animal inside a Cornell box. Our approach rasterizes all objects into every object's reflection map (middle) in 40 ms.*

cones. Note, that this step is only performed once, independently of the number and positions of the views.

Computing the Cut We implemented our algorithm using the transform feedback / geometry stream-out capabilities of our Shader Model 5.0-compliant graphics card. Such hardware allows us to separately append elements to individual output streams while still maintaining a high number of threads. We store I and A_k in vertex streams and process them in a geometry shader as depicted in Listing 1. Furthermore, we can draw nodes directly to the output when the corresponding node-view is inserted into I. This avoids one final iteration over all inactive node-views after the cuts have been computed. Consequently, we have the following options for a node-view a: split (create n children based on the arity of a and append them to the active list), merge (append the parent of a to the active list if a is the first child), draw (draw the node to the corresponding view and appends it to the inactive list), cull (append a to the inactive list, to not lose it as a starting point in future frames). This geometry shader appends only a small number of values to a list (if any), checks the validity of two node-views and traverses not more than one edge up or down the tree. Such small-scale data amplification is considered an optimal scenario for a geometry shader.

As our implementation uses a BSH, we simply draw a point (GL_POINT) with radius according to the respective bounding sphere. Each view is represented by a tile in a destination texture, which effectively avoids rendering m cuts into m separate textures (Fig. 5,3 right side).

```
while(!isEmpty(A_k)) {
 for each a in A_k parallel {
   if(!isValid(parent(a)) {
     if(isValid(a)) {
       draw(a);
       append(I, a);
     } else {
       append(A_{k+1}, allChildren(a));
     }
   } else if(isValid(a) & isFirstChild(a)) {
     append(A_{k+1}, parent(a));
}}}
```

Listing 1: *Pseudo code for our incremental algorithm in a geometry shader using geometry stream-out capabilities. The definition of validity is given in Sec. 3.*

5. Applications

We will demonstrate our algorithm for various well-known techniques such as GI based on Instant Radiosity, Point-Based GI and reflection / refraction mapping.

Instant Radiosity Instant Radiosity [Kel97] has proven to be an excellent means for efficient GI. So-called Virtual Point Lights are emitted from the primary light sources to simulate one bounce of indirect illumination. However, computing visibility from VPLs is a bottleneck and often evaluated either lazily [LSK*07], leading to temporal lag, or imperfectly [RGK*08], limiting the size of the scene. Here, a shadow map is computed for each VPL, and, for a realistic representation, a large number of VPLs is usually required. We compute the cut of the scene for all VPLs in parallel and improve upon previous work [RGK*08] in terms of quality at comparable speed, as demonstrated in Fig. 7 and for a glossy setting in Fig. 4. The figures exhibit the typical Imperfect Shadow Maps (ISM) imperfections. Given the same computation time, our method adapts the point density to avoid holes. To achieve temporal coherence, we enforce that each VPL maps to itself in subsequent frames by fixing the sampling pattern over time.

Point-Based Global Illumination Point-Based Global Illumination has become a valuable alternative to ray tracing and is increasingly used in production [Bun05, Chr08, REG*09]. These techniques proceed as follows: The scene is rendered once for the actual view and the visible pixels are backprojected into the scene. For many of those pixels the scene is rendered again from their world-position into a set of indirect lighting views. This fits well to many-view rasterization. Finally, every indirect lighting view image is convolved with the BRDF simulating one-bounce GI. Previous work was sequential [Chr08] or parallel-over-views [Bun05, REG*09]

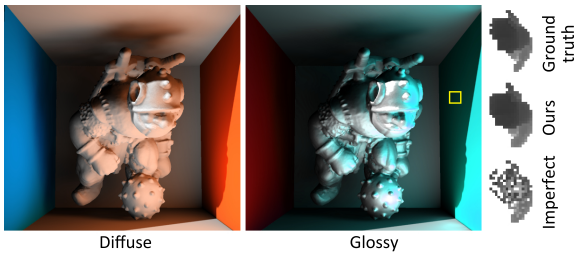

Figure 4: *Instant Radiosity using 1024 Virtual Point Lights for the Grog mesh (500 k triangles, 500 k samples). Rendering time of 2 ms for diffuse (left) and glossy (middle) material at a resolution of* 1024×1024*. For a fixed time budget of 2 ms our algorithm is closer to the shadow mapping ground truth than ISM. The latter reveils holes for close occluders as indicated for the marked region.*

with a sequential geometry loop in every thread. In contrast, we parallelize everything and find all cuts in a lit BVH for all visible pixels.

While our quality matches previous work [Bun05, Chr08, REG*09], it shows increased performance (Fig. 5).

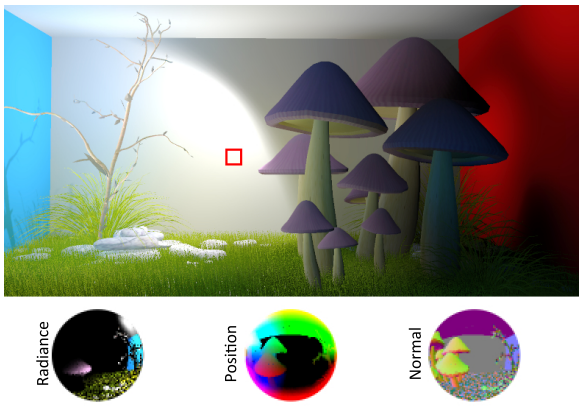

Figure 5: *Top: Point-Based GI in a scene (700 k triangles, 1 M points) rendered at a resolution of* 1024×1024 *with 4096 views in 29 ms. Bottom: Radiance, position and normal for one indirect lighting view (red box).*

Reflections Besides ray-tracing, environment mapping is a well-known method to render reflections [BN76, SKALP05] and refractions [Wym05]. However, computing many environment maps is time-consuming, as all geometry needs to be processed. With our approach, we can produce an environment map per object by finding cuts in the BVH in parallel. In Fig. 3, we computed reflection maps on-the-fly for hundreds of objects.

Further applications Our algorithm is also applicable to a variety of point-based rendering techniques, e. g. single-view adaptive point-based rendering. Single-view rendering is just a special case of many-view rendering (with $m = 1$). Nevertheless, our approach can accelerate such techniques, e. g. rasterization of large meshes (Fig. 6).

Technique	Dragon 4 M	Grog 1.3 M
Polys	3.2 s	1.1 s
8 k points (ISMs)	114 ms	120 ms
Ours, non-incremental	120 ms	55 ms
Ours, incremental	8 ms	5.3 ms
Ours, lazy	4.9 ms	2.9 ms

Table 1: *Performance of our algorithms versus other approaches. We use the scene from Fig. 7's with 1024 VPLs and move the mid-sized area light by* $\approx$ *10 deg. per frame.*

	Dosch 2.1 M ply 1.3 M pts	Grog 1.3 M ply 1.3 M pts	Sponza 72 k ply 1.3 M pts
Build	10 s	9 s	7 s
Update	26 ms	26 ms	15 ms
Poly	46 ms	11 ms	3.8 ms
$c = 1$ px	3.0 ms	2.7 ms	3.4 ms
$c = 2$ px	1.4 ms	1.7 ms	1.6 ms
$c = 4$ px	0.7 ms	0.8 ms	3.8 ms

Table 2: *Performance of our algorithm for building and updating the BVH and cut finding for different levels of quality in* 1024×1024*. Our terminal condition c is the size of the bounding sphere of a node in screenspace given in pixels.*

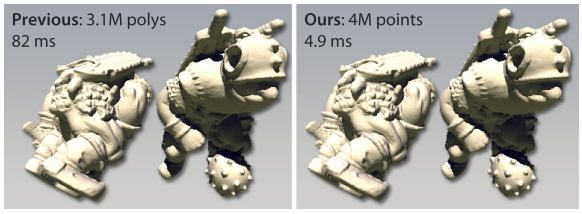

Figure 6: *Timing (shading cost not included) for the Grog model (1.4 M vertices) using plain OpenGL (left) and our method (right).*

6. Results

We implemented our algorithm using OpenGL 4.1 and measured its performance using an NVIDIA GeForce GTX 480 equipped with 1.5 GB of memory. Timings are given for meshes of different sizes in Table 1 and each individual component of our algorithm in Table 2. A complete rebuild of the underlying BVH takes substantially more time than a simple update due to the re-sorting of the sampled-points and the

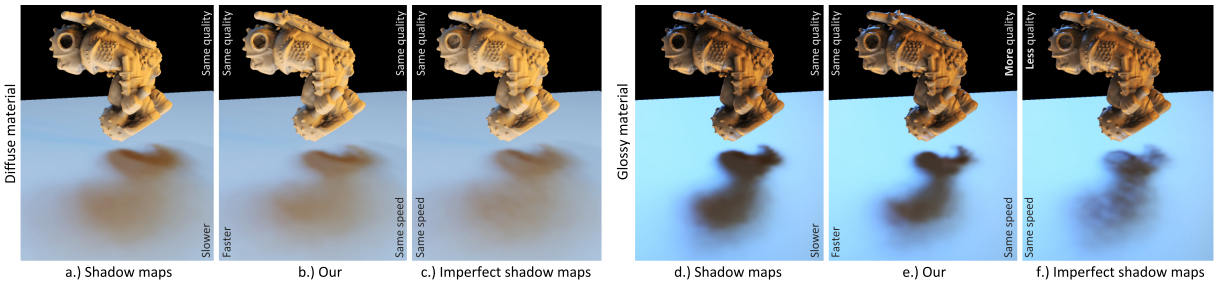

Figure 7: *A dynamic character (500 k polygons, 500 k points) under natural illumination, using diffuse (left) and specular (right) materials. Natural illumination using 1024 point lights and comparison to common reference shadow maps (a,d), our many-view approach to rasterize shadow maps (b,e) and ISM (c,f). Our approach and ISM were adjusted to the same computation time of 4 ms. For diffuse surfaces, our approach and ISM result in similar quality at similar speed. For glossy surfaces, the imperfections in ISM are visible, whereas the quality of our approach remains uncompromised.*

update of the GPU buffers.

The quality of the BVH depends on the mesh deformations because the tree structure does not change unless a full rebuild is performed. For most common deformations, such as character animations, we can re-use the tree structure, avoiding a complete tree rebuild. If there is little coherence, as in the case of fast animation or camera movement, our lazy updates can be used instead (Fig. 8). The difference is perceptually insignificant due to the low-frequency nature of most global illumination effects.

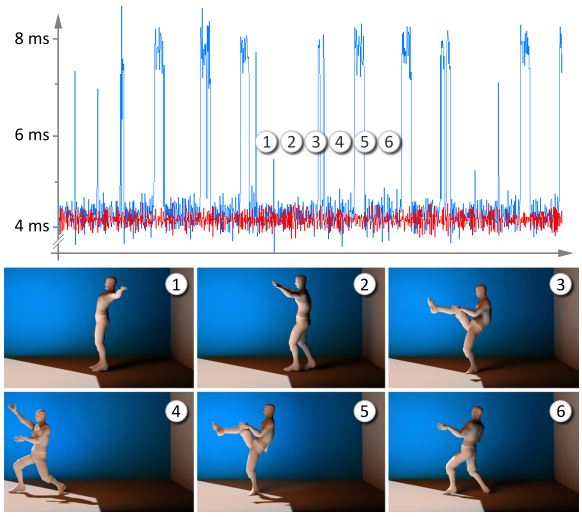

Figure 8: *A plot of the computation time for every frame (Top) for the kicking animation (Bottom). The blue plot shows the computation time for the full, the red one the computation time for the lazy approach. Computation time increases for fast, i. e. incoherent parts of the animation (kicking in 3 and 5). Using the lazy approach, this can be prevented, at virtually the same quality (See the video for a comparison).*

7. Discussion and Conclusion

We presented a method to traverse a BVH for multiple views in parallel. Our fine-grained refinement kernel exploits GPU parallelism effectively. As long as camera motion and mesh deformations are coherent our incremental approach can be applied, otherwise a full traversal of the BVH is required. Unfortunately, very incoherent deformations, e. g. explosions, result in completely different BVHs and are kept as future work.

For the demonstrated applications we only experimented with binary BVHs. However, higher arity trees can be addressed as well and require only to create n children in the case of a splitting event. Also, the completeness of the tree is not strictly necessary. An unbalanced tree can be represented by a double-linked-list representation to support dynamic updates but requires additional memory. In general, our algorithm is compatible to any tree structure that can be mapped to a GPU, but, implicitly, relies on the performance of the respective tree traversal operations. Using a complete tree though, gives a good tradeoff between tree-update performance and a sufficient sampling of the scene. Future hardware, supporting to draw points and triangles simultaneously from within the same shader, would allow a tree with triangles stored at leaf-nodes and arbitrary bounding volumes for inner nodes.

Future work Besides the presented ones, other rendering algorithms can profit from our method. Multi- or binocular-view-stereo, depth-of-field and motion blur can be achieved by rasterizing many views distributed across a stereo domain, a lens or in time. Irradiance volumes [GSHG98] require computing incoming radiance for a grid of n^3 view samples. Image-based photon mapping [YWC*10] shoots photons using environment maps placed at n optimized view positions which could be generated using our approach. Further, acceleration techniques such as LightCuts [WFA*05] or row-column sampling [HPB07] may help to reduce the number of views, but still require to rasterize many of them. We would also like to consider transparency, e. g. participating media and out-of-core data streaming. Even non-rendering

applications can benefit from our approach; for path planning, a per-agent visibility can be used, e. g. for obstacle or enemy avoidance [Rey87].

Acknowledgements This work has been partially funded by the French National Research Agency *MediaPGU* project, the *3D Life* European Network of Excellence and Intel Visual Computing Insititute at Saarland University. We would like to thank PlayAll for the character mesh and J. Tierny for the video voice-over.

References

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (1975), 509–517.

[Ble89] BLELLOCH G. E.: Scans as primitive parallel operations. *IEEE Transactions on Computers 38* (1989), 1526–1538.

[BN76] BLINN J. F., NEWELL M. E.: Texture and reflection in computer generated images. *Commun. ACM 19*, 10 (1976), 542–547.

[BRS05] BOUBEKEUR T., REUTER P., SCHLICK C.: *Surfel stripping*. Tech. rep., Proc. of ACM Graphite, 2005.

[Bun05] BUNNELL M.: *GPU Gems 2 - Dynamic Ambient Occlusion and Indirect Lighting*. 2005.

[Chr08] CHRISTENSEN P.: Point-based approximate color bleeding. *Pixar Technical Notes* (2008).

[DVS03] DACHSBACHER C., VOGELGSANG C., STAMMINGER M.: Sequential point trees. *ACM Trans. Graph. 22*, 3 (2003), 657–662.

[EML09] EISENACHER C., MEYER Q., LOOP C.: Real-time view-dependent rendering of parametric surfaces. In *Proc. I3D* (2009), pp. 137–143.

[FKN80] FUCHS H., KEDES Z. M., NAYLOR B. F.: On visible surface generation by a priori tree structures. In *SIGGRAPH* (1980).

[GKCC10] GORADIA R., KASHYAP S., CHAUDHURI P., CHANDRAN S.: GPU-Based Ray Tracing of Splats. In *Pacific Graphics* (2010), pp. 101–108.

[GM05] GOBBETTI E., MARTON F.: Far voxels: A multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM Trans. Graph. 24*, 3 (2005), 878–885.

[GSHG98] GREGER G., SHIRLEY P., HUBBARD P., GREENBERG D.: The irradiance volume. *IEEE Computer Graphics and Applications* (1998), 32–43.

[Hop96] HOPPE H.: Progressive meshes. *Computer Graphics 30*, Annual Conference Series (1996), 99–108.

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

[HSH09] HU L., SANDER P., HOPPE H.: Parallel view-dependent refinement of progressive meshes. In *I3D* (2009), pp. 169–176.

[Hub93] HUBBART P.: Interactive collision detection. In *Proc. IEEE Symp. on Research Frontier in Virtual Reality* (1993).

[HVAPB08] HAŠAN M., VELÁZQUEZ-ARMENDÁRIZ E., PELLACINI F., BALA K.: Tensor clustering for rendering many-light animations. In *EGSR* (2008), vol. 27, pp. 1105–1114.

[JT80] JACKINS C., TANIMOTO S.: Oct-trees and their use in representing three-dimensional objects. *CGIP 14* (1980), 249–70.

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

[KRG*00] KARI M., RUSINKIEWICZ S., GINZTON M., GINSBERG J., PULLI K., KOLLER D., ANDERSON S., SHADE J., PEREIRA L., DAVIS J., ET AL.: The digital Michelangelo project: 3D scanning of large statues.

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. In *Computer Graphics Forum* (2009), vol. 28, pp. 375–384.

[LRC*02] LUEBKE D., REDDY M., COHEN J., VARSHNEY A., WATSON B., HUEBNER R.: *Level of detail for 3D graphics*. Morgan Kaufmann, 2002.

[LSK*07] LAINE S., SARANSAARI H., KONTKANEN J., LEHTINEN J., AILA T.: Incremental instant radiosity for real-time indirect illumination. In *Proc. EGSR* (2007), pp. 277–286.

[MESD09] MEYER Q., EISENACHER C., STAMMINGER M., DACHSBACHER C.: Data-parallel hierarchical link creation for radiosity. *Proc. EGPGV* (2009).

[PZvBG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M.: Surfels: Surface elements as rendering primitives. In *SIGGRAPH* (2000), ACM, pp. 335–342.

[REG*09] RITSCHEL T., ENGELHARDT T., GROSCH T., SEIDEL H., KAUTZ J., DACHSBACHER C.: Micro-rendering for scalable, parallel final gathering. *ACM Trans. Graph (Proc. SIGGRAPH Asia)* (2009), 1–8.

[Rey87] REYNOLDS C. W.: Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph. 21* (1987), 25–34.

[RGK*08] RITSCHEL T., GROSCH T., KIM M. H., SEIDEL H.-P., DACHSBACHER C., KAUTZ J.: Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph (Proc. SIGGRAPH Asia)* (2008), 1–8.

[RL00] RUSINKIEWICZ S., LEVOY M.: QSplat: A multiresolution point rendering system for large meshes. In *SIGGRAPH* (2000), pp. 343–352.

[SKALP05] SZIRMAY-KALOS L., ASZÓDI B., LAZÁNYI I., PREMECZ M.: Approximate ray-tracing on the GPU with distance impostors. *Proc. Eurographics 24*, 3 (2005), 695–704.

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

[WH07] WALD I., HAVRAN V.: On building fast kd-trees for ray tracing, and on doing that in $O(N \log N)$. In *Interactive Ray Tracing 2006, IEEE Symp. on* (2007), pp. 61–69.

[WS06] WIMMER M., SCHEIBLAUER C.: Instant points. In *Proc. Symp. on Point-Based Graphics 2006* (2006), pp. 129–136.

[Wym05] WYMAN C.: An approximate image-space approach for interactive refraction. *ACM Trans. Graph. 24* (2005), 1050–53.

[XV96] XIA J., VARSHNEY A.: Dynamic view-dependent simplification for polygonal models. In *Proc. Visualization* (1996), pp. 327–334.

[YWC*10] YAO C., WANG B., CHAN B., YONG J., PAUL J.-C.: Multi-image based photon tracing for interactive global illumination of dynamic scenes. *Computer Graphics Forum (Proc. EGSR) 29*, 4 (2010), 1315–1324.

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. In *Siggraph Asia, ACM Trans. Graph.* (2008), pp. 1–11.

[ZPVBG01] ZWICKER M., PFISTER H., VAN BAAR J., GROSS M.: Surface splatting. In *Proc. Siggraph* (2001), pp. 371–378.