MegaViews: Scalable Many-View Rendering with Concurrent Scene-View Hierarchy Traversal

Timothy R. Kol[†1], Pablo Bauszat[‡1], Sungkil Lee[§2] and Elmar Eisemann[¶1]

[1]Delft University of Technology [2]Sungkyunkwan University

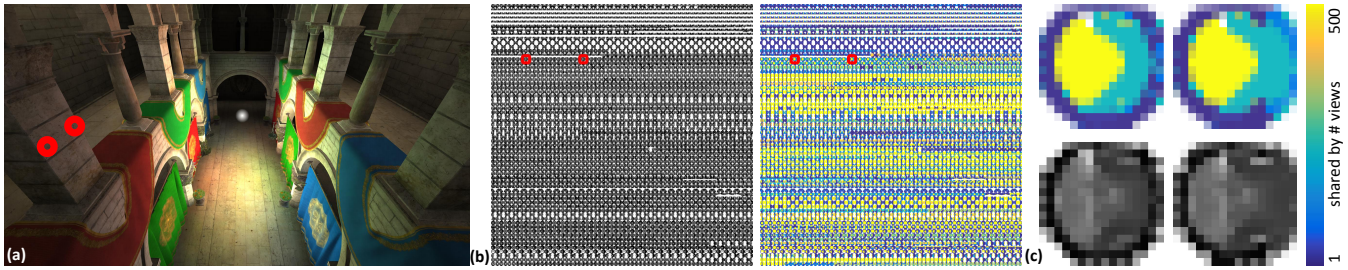

Figure 1: *Indirect illumination computed from* 1*M animated virtual point lights (VPLs) with shadow maps of* 16^2 *resolution generated at interactive rates (*100 *ms, out of* 194 *ms for the image in total) by our many-view rendering algorithm (a). We show shadow maps of a subset of* 2048 *VPLs, for which many pixels are shared and rendered only once for multiple views (b). We highlight two close VPLs in (a) and (b), which can share a large part of their rendering (c). We note that faraway pixels are logically shared by more views.*

Abstract

We present a scalable solution to render complex scenes from a large amount of viewpoints. While previous approaches rely either on a scene or a view hierarchy to process multiple elements together, we make full use of both, enabling sublinear performance in terms of views and scene complexity. By concurrently traversing the hierarchies, we efficiently find shared information among views to amortize rendering costs. One example application is many-light global illumination. Our solution accelerates shadow map generation for virtual point lights, whose number can now be raised to over a million while maintaining interactive rates.

Keywords: Global illumination, Many-view rendering

CCS Concepts

•**Computing methodologies** → *Visibility; Massively parallel algorithms;*

1. Introduction

Recent work has shown that producing many views simultaneously can be very beneficial for realistic rendering [DKH*14]. For example, when many light sources are present in a scene, each requires its own shadow map. Similarly, indirect illumination can be well approximated when first distributing virtual point lights (VPLs) that each illuminate the scene [Kel97, WFA*05, HPB07]. Also, reflective objects can be simulated by creating cube maps from various locations on the surface [BN76, SKALP05, HREB11]. Unlike typical multi-view rendering, such as stereoscopy, soft-shadow mapping, and motion or defocus blur [ABC*91, CPC84, HA90], indirect lighting scenarios show less coherence among the views. Furthermore, the number of views has to be high to ensure a convincing quality, while maintaining a high framerate for interactive applications. This many-view rendering problem is addressed by our work.

The use of a hierarchy is the most common way to obtain sublinear rendering scalability. Coarse representations [RGK*08] or scene hierarchies are widely used [LWC*03]. For each view, an adequate level of detail (LOD) can be chosen, typically represented by a cut through the hierarchy that determines the nodes whose content will

† t.r.kol@tudelft.nl
‡ p.bauszat@tudelft.nl
§ sungkil@skku.edu (corresponding author)
¶ e.eisemann@tudelft.nl

be rendered. However, the use of only a scene hierarchy does not scale well with the number of views. The cost per view is reduced, but the total cost stays linear in the amount of viewpoints.

MegaViews is a novel scalable many-view rendering algorithm. It provides sublinear performance on both the scene complexity and number of views. The idea is to rely on two hierarchies; one on the scene and one on the views. We concurrently traverse both hierarchies, with pairs of scene and view nodes fed into the double traversal. This way, we can exploit coherence among different views, which enables us to employ early culling techniques, as well as shared rendering. A double-hierarchy traversal has been used for efficient intersections in ray tracing or visibility processing [JWSP05, RAH07, MBWW07, MBJ*15]. However, we focus on the rendering of complete images for many views. Our solution is well adapted to GPUs and achieves interactive rates for a large amount of views (we demonstrate a million 16^2 views) in complex scenes on standard hardware. We show the benefit of our solution in several applications, including many-light global illumination [Kel97]. The major contributions of this paper can be summarized as:

- a scene-view hierarchical representation;
- an efficient traversal method;
- a shared rendering solution; and
- many-light applications using our approach.

2. Related work

Level-of-detail representations can reduce the rendering workload per view. A well-explored area, there are many surveys [DFKP05] and books [LWC*03] on this topic, and we refer the interested reader to this literature. Here, we will discuss only approaches closer to our work that amortize costs over several views.

Rendering many views is naturally required for devices such as stereoscopic displays [ABC*91]. Realistic rendering also benefits from many views over time, lenses, and area or volume lights [CPC84, HA90]. For some of these problems either a small number of views is sufficient, or they follow a certain regular pattern, leading to many approaches that exploit this predictable consistency [Hal98, HAM06, LES10]. Nevertheless, other scenarios show less coherence. For example, indirect illumination requires rendering thousands of relatively random views, making it much harder to propose an efficient solution [HPB07].

A relatively direct way to handle visibility for many views is the use of *imperfect shadow maps* [RGK*08]. Here, the scene is sampled and the points are distributed randomly over all views. In this way, the rendering time is independent of the number of views, but the quality becomes increasingly worse if the sampling rate is not increased. The approach relies on hole filling to complete sparse images [MKC07]. A key insight is that low-resolution shadow maps tend to work well for low-frequency indirect lighting, and even imperfections do not necessarily create visible artifacts. We build upon these insights to share rendering between views in our work.

Other approaches [REG*09, Chr08] produce mostly accurate renderings for many views, by relying on a scene hierarchy that is traversed for each view individually. While the solution is well suited for mapping it onto the GPU [REG*09], the workload distribution is not optimal, as each view can take a very different path through the hierarchy. ManyLoDs [HREB11] build upon this insight and enforce a traversal that takes one step at a time. All node-view pairs have the same cost per iteration, making it much more efficient on modern GPUs. Still, the cost remains linear in the number of views.

To reduce the workload further, there are attempts to reduce the number of VPLs or cluster their contributions. *Lightcuts* cluster VPLs, defining a cut through a light hierarchy [WFA*05, WABG06]. The number of VPLs can also be reduced by choosing an effective subset [GS10, REH*11]. The effect of VPLs is also the basis of matrix row-column sampling (MRCS), which sparsely samples combinations of senders (light sources) and receivers (scene) via shadow maps, organized in a matrix [HPB07]. This solution can be combined with lightcuts [OP11] and extended to animated scenes [HVAPB08], as the sparse view evaluation by itself leads to flickering. However, the involved matrix analysis is often too costly for real-time performance. Furthermore, their goal is to choose a low number of good views, while we actually consider many views.

Light culling and selection is also used by screen-space clustering methods, linked to tiled shading [OA11, OBA12, HMY12]. Views are then produced for each tile instead of each light in the form of a cube map that can be organized into a tiled virtual shadow map, which facilitates resolution optimizations [OSK*14, OBS*15]. Nevertheless, the method is mostly limited to light gathering, as no actual renderings are produced for the VPLs. Furthermore, the performance gain depends on the effectiveness of the employed resolution heuristics, which can overestimate. Tiled methods usually build upon a cutoff of the VPL influence in screen space. However, this leads to lower quality compared to randomized sampling [TH16], which benefits from higher resolution shadow maps and a shadow map per VPL. Our solution can produce many shadow maps and is more general in terms of view placement and the choice of resolution.

Image-space clustering is also employed in point-based global illumination (PBGI) [WHB*13], where tiles are repartitioned using a k-means clustering. Assuming coherence of grouped pixels, a baseline cut through the scene hierarchy is established per tile. This cut is rendered into a texture, which is shared per tile. It is then refined per cluster and new views are stored. The performance gain lies in the incremental cut refinement [HREB11], which requires additional memory, and the shared map. Still, sharing information in this way can lead to artifacts if a cluster covers a large extent of the scene and depth fusion can be incorrect. Furthermore, at least one full traversal is performed per tile; the cost per generated view, hence, remains linear in the number of views. Our solution handles arbitrary views and lowers the rendering cost.

Finally, ray-space hierarchies and their traversal have been extensively used in conjunction with object-space hierarchies, including impostor placement [JWSP05], ray tracing [RAH07], potential visibility sets [MBWW07], ray-packet reordering [BWB08], and coherent hierarchical culling [MBJ*15]. They commonly achieve high efficiency by addressing the double-hierarchy traversal with different subdivision criteria (render cost, memory cost, distance, and visibility) on the ray-object pair. Similar to our approach, CHC+RT subdivides the ray-object pair node with the largest normalized area in screen or object space, respectively [MBJ*15].

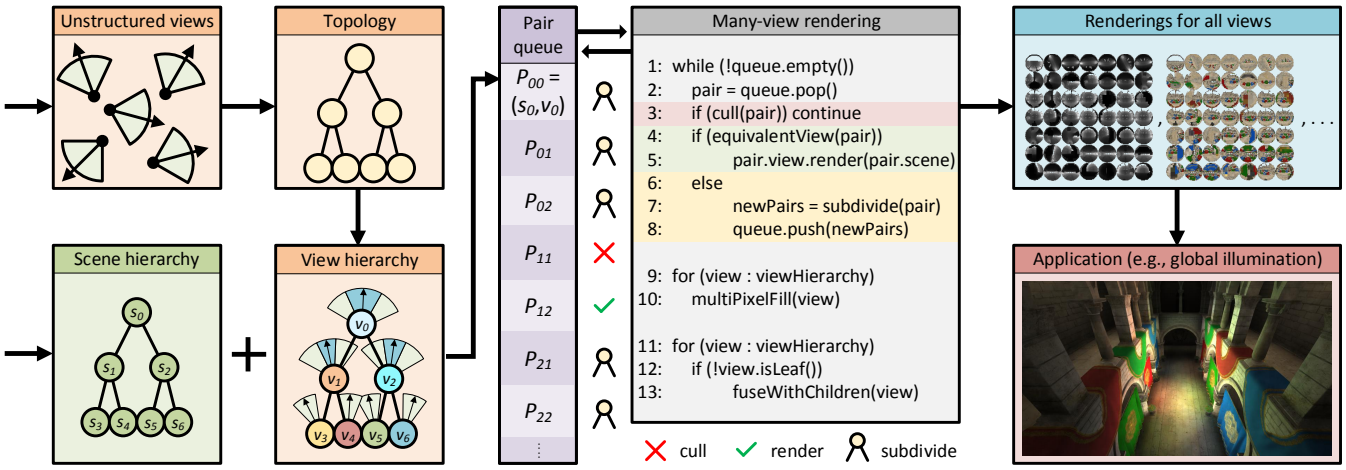

Figure 2: *Overview of our framework. Many unstructured views are organized in a hierarchy. Together with the scene hierarchy, this serves as the input to our many-view rendering solution. It keeps a work queue, initialized with a pair of both roots, to efficiently process scene-view node pairs in parallel. Pairs are either culled, rendered or subdivided. The resulting renderings can be applied, e.g., for global illumination.*

In contrast, we consider both node volumes in object space, and rather than being a heuristic, our subdivision is constrained to facilitate shared rendering, as explained in Section 3.2. Furthermore, previous methods typically render for a single view and frustum [RAH07, MBWW07, BWB08, MBJ*15], whereas we produce full images for thousands of viewpoints. We generate the renderings on the fly, rather than storing the association to the object hierarchy in a preprocess [JWSP05]. Here, our shared rendering leads to high efficiency in terms of both rendering and memory costs.

3. Scalable Many-View Rendering

In this section, we present *MegaViews*, our solution to process multiple scene elements and views together to render images with sublinear performance. Figure 2 shows an overview of the algorithm.

3.1. Scene and View Hierarchies

Scene Hierarchy We assume the scene to be provided in the form of a multi-resolution spatial tree structure, such as an octree. Each node stores scene attributes: color (or material), position, and a surface normal (or normal cone, hierarchically grouping a set of normals). The material property or color is typically chosen to be the average of its children. For each node, a bounding volume is assumed available, which is typically a box or bounding sphere enclosing all children. Such scene hierarchies can be generated offline, but dynamic solutions exist [CG12]. In this sense, our approach is not limited to a static scene hierarchy, although we consider this problem orthogonal to our approach.

View Hierarchy Besides a scene hierarchy, we also rely on a view hierarchy, which groups views spatially in a tree structure. Each view node stores attributes similar to a scene node, but the normal (view direction) is now defined by a *cone* [WFA*05, JWSP05, RAH07] encompassing the view directions of all contained cameras (Figure 3). Then, for each node, we have $v_j := (\mathbf{p}_j, \mathbf{n}_j, \theta_j, \phi_j)$, where $\mathbf{p}_j$ is the center of projection, $\mathbf{n}_j$ the viewing direction, θ_j half the angular

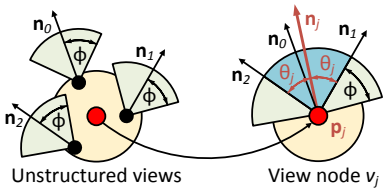

Figure 3: *Cone-based representation of a multi-view node.*

extent of the bounding cone, and ϕ_j half the field of view of the frustum. If a view is omnidirectional, we assume $\phi_j = \pi$, and we refer to a global variable ϕ if all cameras share the same opening angle. Again, we assume bounding volumes are available for each node (the yellow circle in Figure 3).

Rendered-Image Representation It might sound counterintuitive at first, but instead of rendering the actual image that corresponds to each camera, we always produce an omnidirectional map from the camera's position. We rely on the actual view direction to then query the relevant information from this omnidirectional map. The globally consistent parametrization is crucial to facilitate the shared rendering among many different views, as each node in the view hierarchy will contain an omnidirectional map that is a partial rendering of the scene, shared by all its children. Several options exist for view parameterization, and we only need to impose that all views are parameterized in the same way, including the orientation. In practice, we opted for dual paraboloid maps [BAS02], which are the spherical expansion of a paraboloid map [HS98]. Nevertheless, our solution can be implemented with different representations and we will simply refer to omnidirectional maps in the following.

3.2. Many-View Rendering

Given the scene and view hierarchies, we concurrently traverse them in a top-down fashion during rendering. To keep track of the cut through the double hierarchy, we rely on scene and view node pairs

```
Culling
1:  function cull(pair)
2:      vs = pair.scene.position - pair.view.position
3:      α = acos(dot(pair.view.normal, normalize(vs)))
4:      ψ = min(π, pair.view.θ + ϕ)
5:      fs = sin(α - ψ) * length(vs)
6:      return α > ψ && fs > pair.scene.radius + pair.view.radius
```

Figure 4: *Pseudocode for culling with spherical bounding volumes.*

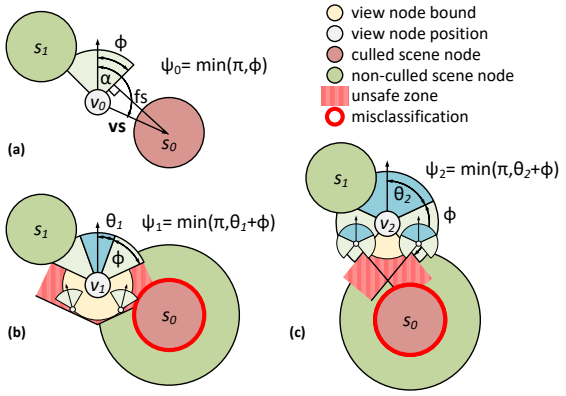

Figure 5: *Culling. For a single view (a), we can cull scene nodes outside the view frustum. For multiple views (b), we test by virtually enlarging the scene node bound (large green area behind misclassification). Hereby, we avoid incorrect culling, as for s_0. We use the same process for nodes with an angular extent $2(\theta_j + \phi) > \pi$ (c).*

$P_{ij} = (s_i, v_j)$, where s_i and v_j are scene and view nodes in their own hierarchies, respectively. A breadth-first traversal is employed, maintaining a work queue of these pairs, initialized with $P_{00} = (s_0, v_0)$, corresponding to the roots of both hierarchies (Figure 2).

A naive traversal would subdivide pairs (by popping them from the queue and pushing its children) when either of the nodes have children, and renders once both nodes are leaves. This process however does not take advantage of redundancy and does not scale well; a million scene nodes with as many views can produce a trillion pairs. We therefore want to process and render for multiple elements from *both* hierarchies at once, which means sharing renderings among many views. Using only scene [HREB11] or view hierarchies [WFA*05] misses a large amount of this shared information, and can not lead to sublinear rendering performance over both the scene complexity and the number of views.

We improve the traversal as follows. As shown in Figure 2, for each pair $P_{ij} = (s_i, v_j)$, we conservatively test if s_i would contribute to any of the views in v_j, and if not, cull it. Otherwise, if s_i projects to less than a pixel for all children of v_j, we verify if the rendered result would activate the *same* pixel in all views of s_i. If so, we render s_i into the omnidirectional map of v_j, which is shared by all of its children. Otherwise, we subdivide in a way favoring the aforementioned conditions, and process the new pairs in the next iteration. In what follows, we describe the details of our algorithm.

Culling Each view will typically only see a part of the scene, which enables us to cull scene nodes, similar to frustum culling. The test

```
Equivalent view
1:  function equivalentView(pair)
2:      vs = pair.scene.position - pair.view.position
3:      vsConservativeLength = max(0, length(vs) - pair.view.radius)
4:      α = acos(abs(normalize(vs).x))
5:      ps = projectedSize(pair, vsConservativeLength, α)
6:      return ps < pixelSize || (pair.scene.isLeaf() && pair.view.isLeaf())
```

Figure 6: *Pseudocode for view equivalence computation for dual paraboloid mapping and spherical bounding volumes.*

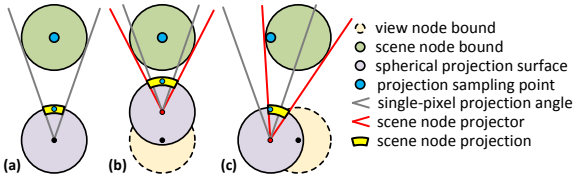

Figure 7: *Discrepancies of scene-node projections for vertical (b) and horizontal (c) displacement of a child view against the projection for the center of the view node (a).*

should be conservative and light-weight to minimize any overhead. Figure 4 shows pseudocode for the case of spherical bounding volumes; see Figure 5 for symbols.

For a single view, culling means ignoring a scene node if its bounding volume lies outside the view frustum (e.g., s_0 in Figure 5a). To ensure we only cull nodes that are entirely outside the frustum, we require $\alpha > \psi$ and fs (line 5) larger than the scene node's bounding radius (line 6); the bounding radius of a single view is zero since it only needs to encompass a single point. The extension to other bounding volumes is straightforward; we can either take a sphere encompassing the bounding volume, or use a tighter bound, resulting in a more complex computation.

For multiple views in a hierarchy node, we want to avoid an individual test per view. While the stored normal cone conservatively contains all children's view directions, the assumption that all centers of projection coincide with the center of the view node's bounding volume can lead to misclassifications (s_0 in Figure 5b). In the worst case, child views are located on the bounding surface with a view frustum parallel to that of the parent (the unsafe zones in Figure 5 indicate where incorrect culling can occur). The extent of this unsafe zone is at most equal to the view node bound radius.

To avoid misclassifications, the tests with bounding spheres can efficiently be made conservative. We can cull as before, with the additional requirement that fs is larger than the scene node radius plus that of the view node (the large green area behind s_0 in Figure 5b). This addition of the view node radius is not smaller than the extent of the unsafe zone, resulting in a conservative test (line 6). Figure 5c shows that we can apply the same strategy with angular frustum extent $2(\theta_2 + \phi) > \pi$.

Shared Rendering In addition to culling, we employ a second acceleration technique. The idea is to avoid rendering a scene node s_i into each individual view of a view node v_j if the rendered result would be the same for all children of v_j. In other words, for a view node v_j, we will test if the projection of the scene node s_i would fill

```
Pair subdivision

1:  function subdivide(pair)
2:      if (pair.scene.level < pair.view.level)
3:          for (child : pair.scene.children)
4:              newPairs.add(createPair(pair.view, child))
5:      else
6:          for (child : pair.view.children)
7:              newPairs.add(createPair(pair.scene, child))
8:      return newPairs
```

Figure 8: *Pseudocode for pair subdivision for octree structures.*

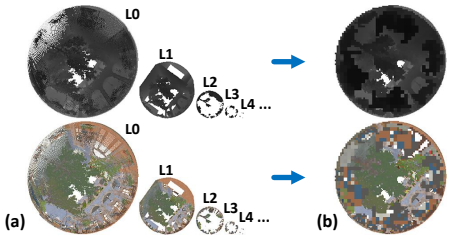

Figure 9: *Multi-level point-only renderings at a* 256^2 *resolution for a single view (a) using mipmap-based hole filling (b).*

the *exact same* single pixel in the omnidirectional map of each child view. If so, we render s_i directly into the omnidirectional map of v_i (and not into that of its children) and remove the pair from the queue. This technique quickly becomes effective, as distant geometry will only have minimal parallax if views differ slightly.

Figure 6 shows pseudocode to test the view equivalence, in the case of bounding spheres and dual paraboloid mapping. We test for the projected size of s_i, which needs to be less than a pixel. We can directly compute the projection (line 5) using the length of **vs**, which is the vector from the camera to the scene node (line 2), and the angle α between **vs** and the camera direction. For a dual paraboloid parameterization with the front view looking down the positive x-axis, α is the angle between **vs** and the positive or negative x-axis, depending on whether s_i projects into the front or back view, respectively (line 4). To compute the projected size (line 5), we need to take the camera parameterization into account.

To handle a view node v_j that contains multiple cameras, we need to give a conservative upper bound on the projected size of s_i for all child views in v_j. Again, the individual views are not guaranteed to be at the center of v_j's bounding volume. As shown in Figure 7b, the result is that the projected size of s_i can vary depending on the child view's displacement, with the worst case being a vertical offset in the direction of **vs**. A conservative test for bounding spheres is then to shorten the length of **vs** by the bound radius of v_j, which results in a larger projected size and a conservative upper bound (line 3).

Given that the projection is smaller than a pixel (line 6), we want to predict if it projects to the same pixel for all views in v_j. A conservative assumption is to consider any position inside of v_j's bounding volume as a potential view location. A horizontal displacement towards the bound surface as in Figure 7c is a worst-case scenario. When the bounding volume of s_i is not smaller than that of v_j, the horizontal offsetting results in filling the same pixel, since s_i is sampled for all views in v_j and its projection remains identical (Figure 7c). Our view equivalence algorithm is therefore valid, if we keep the view node bound at most equal to the scene node's. As shown in Figure 2, if the equivalence test fails, the pair is subdivided. However, this is only possible when one of the scene and view nodes is not a leaf, which we confirm on line 6.

Pair Subdivision Whenever a scene-view pair $P_{ij} = (s_i, v_j)$ is taken from the queue and a subdivision is required, it is not obvious whether to descend into the scene hierarchy from s_i or into the view hierarchy from v_j. Always subdividing the scene node first would negate the benefits of the scene hierarchy, while first subdividing the view node reduces the approach to a scene-only hierarchy. To benefit from our double hierarchy, we instead opt for a strategy that allows us to optimize for shared rendering.

To validate our aforementioned determination of view equivalence, our subdivision strategy compares the node bounds. If the smallest bounding volume of the children of s_i is smaller than that of v_i, we subdivide v_i. Otherwise, we subdivide s_i. In other words, the view node's bounding volume is always ensured to be smaller than or equal to that of the scene node. For two identical octree structures encoding the view and scene hierarchies, this strategy results in an alternating subdivision; see Figure 8 for pseudocode.

Multi-Pixel Filling for Nearby Geometry Our view equivalence test ensures that most rendered scene nodes project to less than a pixel. However, a pair of leaf nodes cannot be subdivided further, forcing us to potentially falsely report equivalence (line 6 in Figure 6). If a leaf node is very close, it might project to an area larger than a single pixel, especially when using high-resolution renderings. If the view node represents multiple views, the projection of the scene node can then potentially differ. Consequently, we would need to render the scene nodes into each view individually. While this degrades performance, it is relatively uncommon; in practice, it occurs for $< 5\%$ of the rendered pairs and only for scene nodes in direct proximity. We observe no real perceivable difference when rendering into the leaf node rather than the individual views, as long as we fill all pixels the scene node projects to.

We could fill the pixels one by one. However, mipmap splatting [LH13] is more efficient. Here, render targets are defined in multiple levels of coarser resolutions. Whenever a scene node projection is larger than a pixel, we splat it into a higher mipmap level. If desired, we can postprocess each map after rendering by pushing the higher level pixels down to the lower levels, which is a push-only application of a pull-push synthesis [SKE06, RGK*08] (Figure 9).

Image Queries After the entire rendering is completed, we can query any pixel of any view in the scene. To this extent, we first map the pixel of the view to its corresponding pixel in the omnidirectional map. Then, we descend the view hierarchy from the root and look up the values in this location in each view node's map. The last encountered non-empty value corresponds to the wanted pixel value.

If many queries are performed, it can be beneficial to perform a *fusion* of the omnidirectional maps to produce a complete image per single view. To this extent, it is sufficient to perform a top-down processing, where the pixel values of the parent node are fused with the map of the child nodes, which means that we fill up holes in

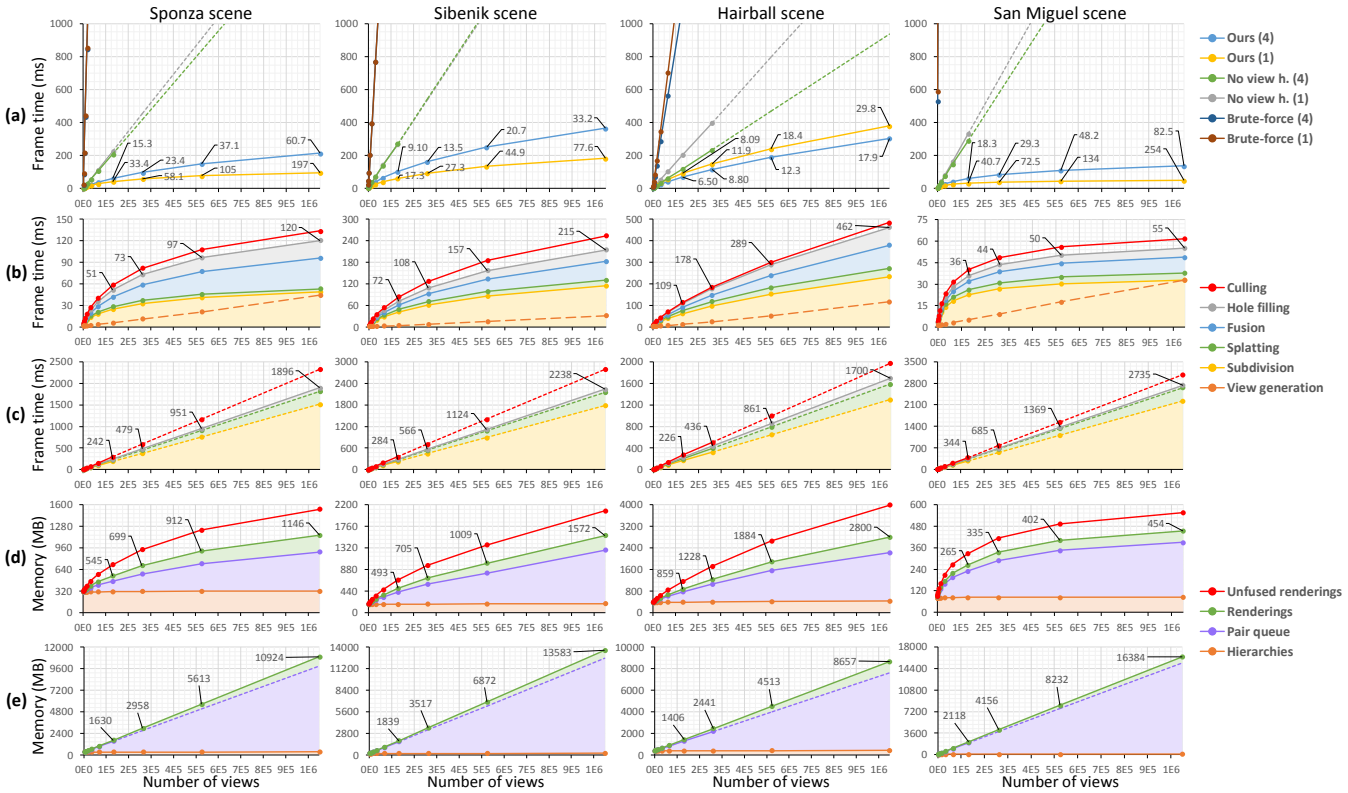

Figure 10: *View render timings for four scenes against the number of views (a). We compare our method to not using a view hierarchy (ManyLoDs [HREB11]). To test two distributions of views, we initialized the view set as* 1- *and* 4-*bounce VPLs. Additionally, a brute-force sequential rasterization without any hierarchy is presented. Dotted lines represent an extrapolation for missing data due to memory limitations on the pair queue. Data labels denote the average number of single views that share a scene node rendering in (a). We break down the results for individual components of our method (b) and ManyLoDs (c), and show our GPU memory consumption (d) and that for ManyLoDs (e).*

the child map with the content of the parent map. Ultimately, this process results in a completely filled image for each leaf view.

Finally, some applications, like shadow mapping, require depth information. Initially, we use the distance to the center of v_j as the depth value for its omnidirectional map. If we query an individual view v, there would then be a small discrepancy with regard to the actual depth value. This difference is easily rectified during the fusion step by taking the actual positions of v and v_j into account.

4. Results

We implemented our solution entirely on the GPU using the OpenGL API, with no CPU-GPU communication at runtime. We tested it on a GeForce GTX 1080 Ti at a 1920×1080 resolution. We made use of sparse voxel octrees with 11 levels for both hierarchies. We use bounding sphere volumes encompassing the cubical voxels for our culling and view equivalence computation. Further, we use a 16^2 resolution for single views. At this resolution, multi-pixel filling is not necessary in practice and therefore excluded in timings except when specifically mentioned. Fusion is always enabled, however.

The scene SVO is generated in a few seconds with an unoptimized depth peeling preprocess [KSA13], which builds the hierarchy down to the specified maximum depth of 11 levels. We consider more efficient voxelization an orthogonal problem. Using advanced solutions [ED06, SS10, CG12] would significantly reduce construction time and could even enable animated scenes, since our many-view rendering does not rely on any precomputation on the hierarchies.

We construct the view hierarchy each frame and support fully dynamic lights. After initialization with a root node, the view hierarchy is generated from a set of single views. For each view, we refine the tree down to the deepest level, such that it ends up in a leaf node. We count the number of views per leaf node, which is used to construct an offset into a global array, containing all information about single views. We then compute view node attributes in a bottom-up fashion, after which the global array can be discarded.

4.1. Many-View Rendering Performance and Memory

We tested four scenes: *Sponza* (Figure 1, 6M leaf nodes), *Sibenik* (Figure 13, 3.1M leaf nodes), *Hairball* (Figure 14, 6.9M leaf nodes), and *San Miguel* (Figure 17, 1.6M leaf nodes). As indicated in Section 3, the views are rendered using an omnidirectional map with the same coordinate system regardless of the view direction. We tested two different view distributions of up to 1M views. The first are 1M VPLs generated directly from the light source, the second are 4-bounce VPLs. Here, 256K VPLs were released from the light

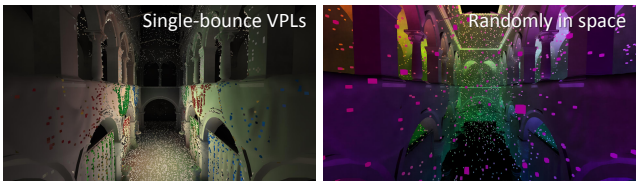

Figure 11: *Different distributions of* 64*K views in the* Sponza *scene.*

and bounced three times, leaving one VPL behind at each bounce and at the final impact point, storing their propagated radiance as attributes. Each VPL has a hemispherical frustum ($\phi = \pi/2$), which is taken into account for our culling.

Total Timings We compare to ManyLoDs [HREB11] and a brute-force rasterization (not relying on any hierarchies) in Figure 10a. We also tested a solution with a view hierarchy but no scene hierarchy. Here, computation times became unfeasible as soon as more than a hundred views were used. Other previous work operates in screen space, or focuses on ray tracing or visibility processing instead of generating complete views, preventing a direct comparison. Since ManyLoDs suffer from high memory consumption, we extrapolate data where memory grew out of bounds, denoted by the dotted lines.

Our approach achieves sublinear performance in all scenes, whereas the competing method shows a clearly worse scalability with respect to the number of views. For a million views, our solution outperforms ManyLoDs by roughly an order of magnitude on average. Single-bounce VPLs in close proximity of a smaller part of the scene, like in the *Sponza* and *San Miguel* scenes, have a high correlation and result in the largest speedup. We can see a clear link between performance and the degree of shared rendering; we show the average number of views that share a single rendered scene node as data labels for 128*K*, 256*K*, 512*K* and 1*M* VPLs. The only exception is the *Hairball* scene, which has slightly better performance for 4-bounce VPLs due to light rays leaving the scene.

As we can already partly see from the *Hairball* scene, in the worst case, views are distributed more uniformly in space, reducing the coherence. We show an extreme case for the *Sponza* scene in Figure 11, where the random distribution here results in rendering times a factor of four slower than those for single-bounce VPLs. Still, we observe sublinear performance as we scale up to many views. Naturally, if the number of views is sufficiently reduced, we lose opportunities for shared rendering, which causes our performance to roughly match that of ManyLoDs for VPL numbers below 2048.

Individual Analysis We break up our timings into individual components for single-bounce VPLs in Figures 10b (ours) and 10c (ManyLoDs). Subdivision relatively takes up a lot of time due to the writing of new pairs to memory. However, its performance is greatly improved in comparison to ManyLoDs thanks to our shared rendering. We splat the scene nodes of valid pairs into the corresponding renderings, which are subsequently fused down to the leaf views. While fusion adds a significant overhead, querying without it is an order of magnitude slower, since each query visits a number of potentially sparse renderings up to the hierarchy depth. For VPLs, querying is often a bottleneck, which makes fusion a valuable option. We include the hole filling to show the behavior of all components, with data labels denoting the total rendering time. We show the merit of our culling by displaying the overhead if it were to be disabled. In our tests, culling reduces the frame time by 20% at most. In cases such as the *Hairball* scene, where each VPL on the wall potentially sees the entire scene, culling provides only little gain, but our tests never indicated a negative impact. In these scenarios, the speedup is mostly due to our double-hierarchy traversal and shared rendering. Finally, we show the performance for our view hierarchy generation in Figure 10b, which includes VPL placement.

Memory Consumption We identify four major components that consume GPU memory during runtime; Figures 10d (ours) and 10e (ManyLoDs) show their memory uses for single-bounce VPLs, with the combined usage denoted by data labels. The pair queue that is kept for the double-hierarchy traversal contains 64 bits per pair, and we report the peak memory usage. For the rendered images themselves, we look at fused 16^2 shadow maps with 32-bit depth values. For our method, the unfused renderings associated with non-leaf view nodes can be discarded after fusion. The scene and view hierarchies contain 512 and 256 bits of information for non-leaf and leaf nodes, respectively. We report their combined memory use, since the view hierarchy's consumption is typically negligible compared to that of the scene. For ManyLoDs, we report the sum of the scene hierarchy and individual view information, with 256 bits per view. Thanks to our early subdivision termination due to shared rendering, we can keep the pair queue memory usage to a minimum, while our hierarchy reduces the number of views for which we need to produce a shadow map. Without any further optimization, we again note on average an order of magnitude of memory reduction, with sublinear scalability with respect to the number of views.

4.2. Multi-Pass Many-View Rendering

Our previous experiments showed that our *MegaViews* far outperform ManyLoDs for low-resolution renderings. To assess the scalability of our solution, we investigate performance and memory usage for higher resolutions. However, memory usage easily grows beyond hardware limitations. To be able to evaluate performance without overflow, we devise a multi-pass solution similar to [RAH07].

For our method, we effectively split the work into eight sequential sub-jobs on the octree structures, taking each of the view root's child nodes separately as starting points for our double-hierarchy traversal. If this still proves to be insufficient, we split the children again, until we reach a level (which we call the multi-pass level) at which the pair queue fits into memory. For ManyLoDs to roughly match our algorithm, we define a multi-pass level as subdividing the unordered set of individual views into eight equal-sized parts to be treated sequentially. While the work subdivisions can be different for both methods, we observe no major change in performance for different subdivision schemes, so that a comparison is still valid.

Total Timings In Figure 12, we compare our method to ManyLoDs, much in the spirit of Figure 10, using the multi-pass solution for higher rendering resolutions. We use 1*M* 1- and 4-bounce VPLs. Here, we always enable hole filling, which is necessary for the higher resolutions. As we can see in Figure 12a, after a small constant

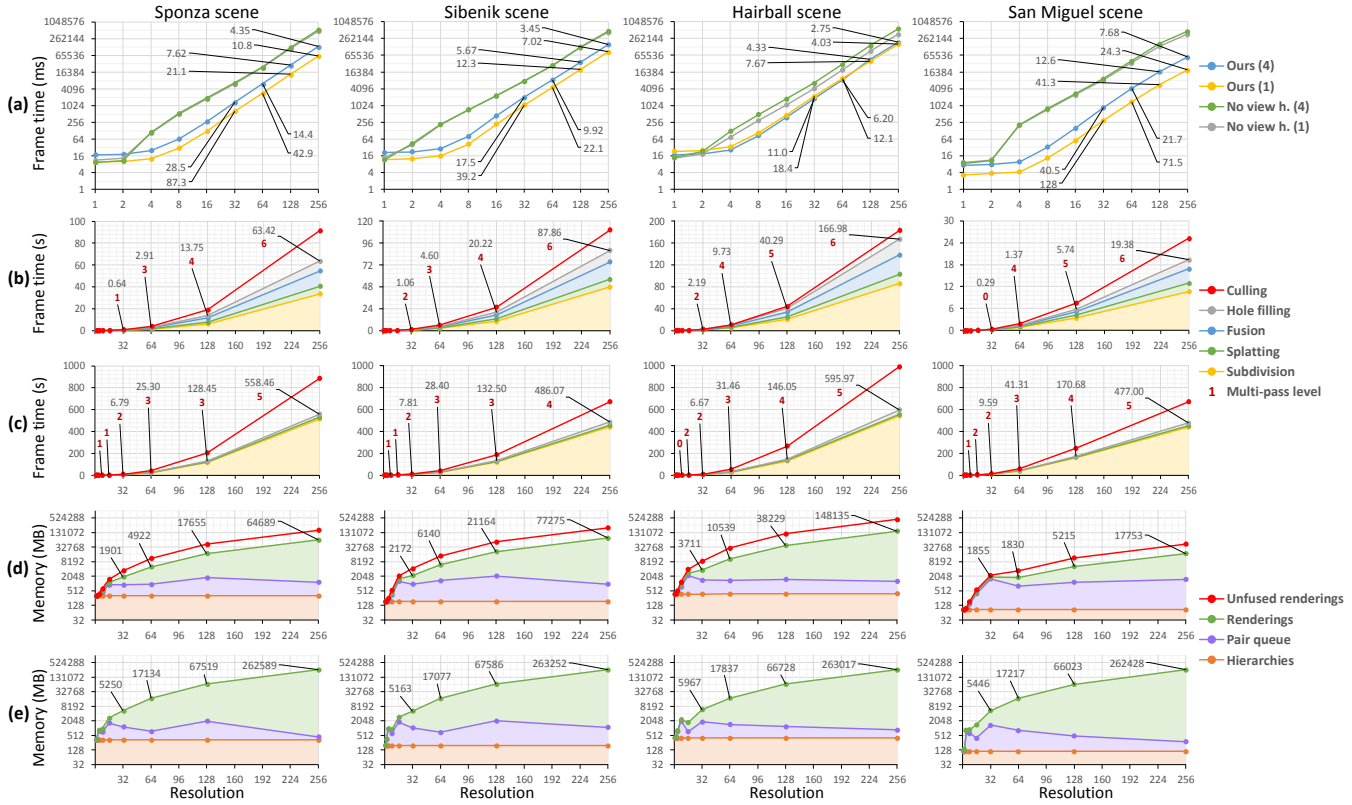

Figure 12: *View render timings for four scenes against the rendering resolution (a). We compare our method to not using a view hierarchy (ManyLoDs [HREB11]). Again, we initialized the view set as* 1- *and* 4*-bounce VPLs. Data labels denote the average number of single views that share a scene node rendering in (a). We break down the results for individual components of our method (b) and ManyLoDs (c), additionally showing the multi-pass levels as data labels, and show our GPU memory consumption (d) and that for ManyLoDs (e).*

overhead, frame times roughly quadruple for a doubled resolution for both methods, corresponding to the increase in pixels. We do note a slight increase of the multiplication factor for our approach in most cases, since there is no shared rendering possible between different passes. Noting the logarithmic scale, however, we still clearly outperform ManyLoDs, and at a degree only slightly less than what we observed for a 16^2 resolution. The reduced effectiveness stems from the decreased shared rendering, which we denote again by data labels for resolutions of 32^2, 64^2, 128^2 and 256^2.

Individual Analysis Figures 12b (ours) and 12c (ManyLoDs) show the individual timings for single-bounce VPLs. We show total frame times, and multi-pass levels for subdivided view sets as data labels.

Memory Consumption Figures 12d (ours) and 12e (ManyLoDs) show memory consumption in a logarithmic scale. The fluctuations we see at low resolutions are caused by the pair queue's peak memory decreasing as the multi-pass level increases. While we use significantly less memory than ManyLoDs, due to the renderings, in-core storage becomes infeasible. Therefore, results need to be written to the disk, or queried on the fly. For instance, VPL gathering can be done for those in the current subdivision, after which the results of each pass are composited. Note that all techniques producing high-resolution renderings for $1M$ views face this problem.

5. Applications

Our algorithm is general but particularly well suited for low-resolution views or an extreme amount of views, as the amount of shared information increases. For this reason, real-time global illumination techniques are a very good test case for our solution.

5.1. Instant Radiosity

We rely on our *MegaViews* approach to generate shadow maps for many VPLs, but producing a final image still requires gathering the VPL contributions for each screen pixel. Recovering all contributions would be too costly for an interactive application. Fortunately, our algorithm enables an acceleration. We can apply our culling during the gathering step as well. For this traversal, we stop at a coarse level in the hierarchies, and cull pairs as before. Additionally, we enable an optional distance-based cutoff to prevent gathering from distant, often negligible VPLs, which is a common approximation [OBS*15]. This test can be conveniently accelerated using the view hierarchy by culling faraway view nodes.

For interactive performance, we employ a per-pixel random subsampling of the VPLs, after which we apply a cross-bilateral filter [ED04, PSA*04, MML12], which generates smooth results due to the very large number of VPLs that we sample from.

Figure 13: *Effect of hierarchy resolution in the* Sibenik *scene. While the shadow map rendering cost is significantly reduced, using a too low hierarchy resolution causes inaccurate shadow maps, resulting in missed or exaggerated occlusions and artifacts.*

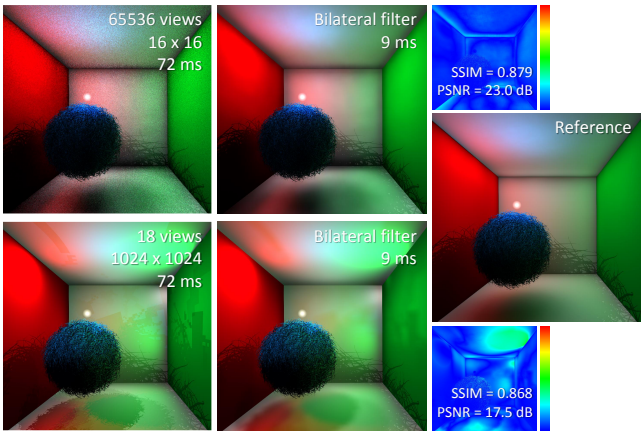

Figure 14: *Effect of shadow map resolution in the* Hairball *scene.*

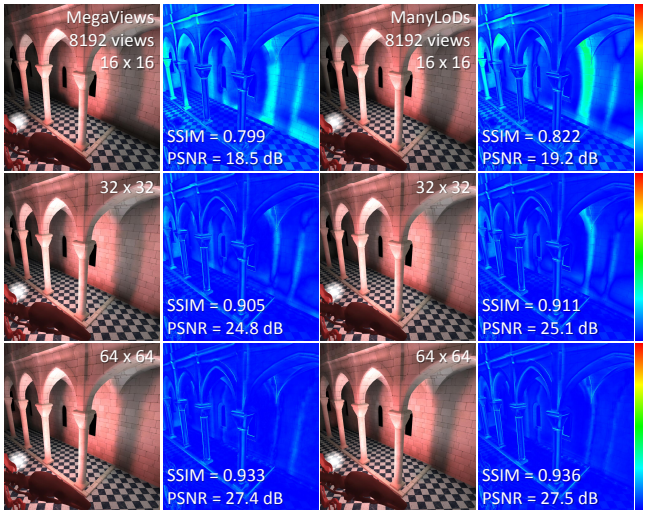

Figure 15: *Effect of shadow map resolution in the* Sibenik *scene.*

We illustrate the effect of reducing the resolution of both hierarchies for the *Sibenik* scene in Figure 13, while maintaining $64K$ 4-bounce VPLs. Reducing from 11 to only 8 levels speeds up shadow map (SM) rendering from 31 ms to 7 ms due to the faster hierarchy traversal. Nevertheless, the resulting shadow maps lose precision, which translates to missed or exaggerated occlusions. Consequently, artifacts start to appear, stemming from the increased number (15 times more) of scene leaf nodes that project to more than a pixel for view leaf nodes containing multiple individual views. These potentially introduce errors, as discussed in Section 3.2.

The effect of changing the shadow map resolution is shown for the *Hairball* scene in Figure 14, where we compare a 16^2 resolution for $64K$ single-bounce VPLs to a 1024^2 resolution. For the latter, an equal-time comparison results in 18 VPLs. Such a small amount of views cannot deliver a convincing quality. Low-resolution shadow maps are very fast to compute, and can still deliver good quality, as the light energy is distributed across many VPLs. However, we do see some over-estimation of occlusion due to the lower precision of the shadow maps. We show an SSIM [WBSS04] and PSNR comparison to a reference solution, and absolute difference images. These shortcomings are not due to our method, but are shared by all VPL-based solutions when relying on low-resolution shadow maps.

Low resolutions work relatively well for the *Hairball* scene, with its large indirect shadow. However, in the presence of indirect shadows cast by thinner geometry, such as the pillars in the *Sibenik* scene, very low resolutions may fail to sufficiently capture the details. As shown for $8K$ single-bounce VPLs in Figure 15, this can result in over-estimating the indirect shadow for an unfortunate placement of the light source. Here, a resolution of 64^2 produces much better results, as illustrated by the comparison. As we have demonstrated in Section 4.2, our algorithm still easily outperforms ManyLoDs under these resolutions. We also show a direct visual comparison to ManyLoDs. Our method is only slightly more prone to errors due to the aforementioned scene leaf nodes projecting to more than a pixel. As expected, it has more impact for nearby geometry, as we can see from the indirect shadow of the closest pillar in the difference

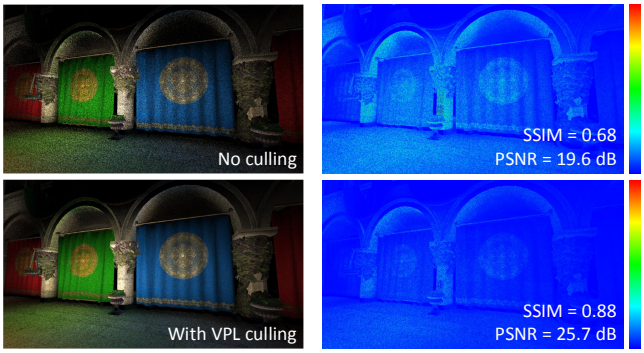

Figure 16: *Hierarchical culling in VPL gathering. By only sampling from non-culled VPLs, noise is significantly reduced.*

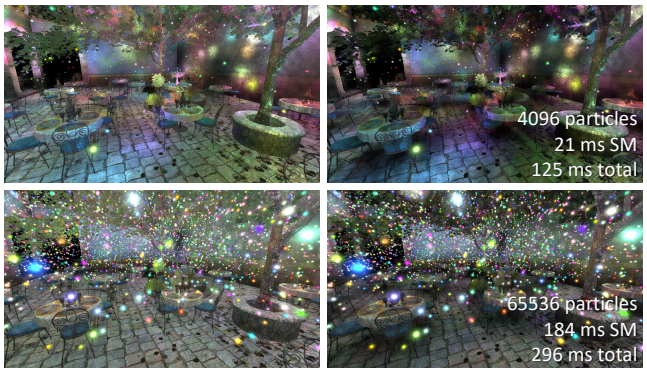

Figure 17: *Examples of glowing-particle rendering without (left) and with (right) shadows. By simply setting the view volume to a sphere, we can render shadow maps for glowing particles.*

images. Furthermore, for low resolutions in particular, the grouping of VPLs in a leaf node can produce discrepancies when compared to ManyLoDs. However, grouping geometry in a scene leaf node, which is employed by both methods, is an approximation of the same magnitude, since both hierarchies use the same resolution. Indeed, it does not make our results look less plausible.

We also evaluate our culling and the distance-based cut-off during gathering. Here, we use 64 random samples per pixel from $1M$ single-bounce VPLs for the *Sponza* scene, which were all rendered using our solution. We can eliminate on average 94% of the leaf view nodes during our concurrent traversal up to a hierarchy level of 6. Consequently, mostly samples are used that in fact contribute to a pixel's indirect illumination. This significantly reduces the noise, as becomes apparent from the comparison in Figure 16.

5.2. Glowing Particles

Similar to instant radiosity, we can perform many-light rendering. Again, we build the view hierarchy on the lights each frame to enable animation. Our many-view rendering enables us to efficiently approximate visibility for many light sources, which results in higher realism compared to not evaluating the resulting shadow maps.

We show results for glowing particles in Figure 17. Since they represent omnidirectional lights, each node's view frustum is now a complete sphere, making it impossible to use culling. Additionally, since the particles are randomly distributed in space, performance is reduced compared to VPLs, since there is less coherence. In fact, this is a worst-case scenario for our approach. In the *San Miguel* scene, our solution requires 21 and 184 ms for rendering shadow maps for $4K$ and $64K$ particles, respectively. Still, our approach is nearly twice as fast as ManyLoDs for $64K$ views, with better scaling as we increase the number of glowing particles.

6. Discussion and Limitations

Our method scales well with the amount of views and scene nodes. We presented sublinear performance for both dimensions, which makes our solution very effective and future-oriented. Several applications could benefit from our solution. We presented indirect illumination using our method, but other examples, such as visibility for crowd simulation, fast collision detection or reflections via cube maps are also possible applications. Our method is relatively easy to implement and can be entirely executed on modern graphics hardware in an efficient manner, since our hierarchy traversal ensures just one operation per thread: either culling, rendering, or subdivision.

A limiting factor of our approach is memory consumption. While we already reduce the pair queue size using our multi-pass solution at a small decrease in performance, the other components can also take up a lot of memory. The renderings themselves can be compressed using texture compression, sparse-texture extensions (typically 30% of the fused maps are non-filled even for a spherical frustum), or, in the case of shadow maps, precision reduction; our 32-bit depth values can be reduced to 16 bits. When using SVOs, the scene hierarchy can be compressed using directed acyclic graphs [DKB*16, DSKA17], while the view hierarchy overhead is typically negligible, since it is a sparse subset of the scene hierarchy.

As for micro-rendering solutions, choosing a low resolution can lead to aliasing and occlusions can be overestimated (e.g., sub-pixel objects still fill entire pixels). One remedy is to increase resolution, but it results in additional compute time. While our approach scales linearly in resolution, adequate anti-aliasing solutions are an interesting avenue for future work. Similarly, the resolution of the hierarchies needs to be carefully chosen to find an acceptable tradeoff between visual quality, and requirements on performance and memory. In our experiments, we could no longer perceive any visual difference for hierarchy resolutions above 11 levels.

Furthermore, as in all VPL approaches, temporal coherence is an interesting factor. It is possible to reuse information over time if scene and view changes are insignificant. Our shared rendering solution seems like a good starting point by keeping high-level omnidirectional maps in the hierarchy stable over several frames.

Our approach is compatible with a different parametrization of the omnidirectional maps. Our choice was inspired by its usefulness in an instant radiosity context. An interesting direction would be adaptively controlling the resolution based on the image content.

7. Conclusion

We have presented *MegaViews*, a scalable algorithm to efficiently render complex scenes from a very large number of viewpoints.

Our concurrent traversal on both scene and view hierarchies enables shared rendering and early culling. Consequently, we reach sublinear performance over the scene complexity and the amount of views. Our algorithm is general enough to be applied to many multi-view problems, and fits well with real-time many-light rendering. For future work, we want to exploit coherence in animation. A first solution could reuse cuts from previous frames [HREB11].

Acknowledgements

This work was partially supported by the NWO VIDI Grant NextView, the FP7 European Project Harvest4D, and the ITRC program (IITP-2018-2016-0-00312).

References

[ABC*91] ADELSON S. J., BENTLEY J. B., CHONG I. S., HODGES L. F., WINOGRAD J.: Simultaneous Generation of Stereoscopic Views. *Computer Graphics Forum 10*, 1 (1991), 3–10. 1, 2

[BAS02] BRABEC S., ANNEN T., SEIDEL H.-P.: Shadow Mapping for Hemispherical and Omnidirectional Light Sources. *Advances in Modelling, Animation and Rendering* (2002), 397–408. 3

[BN76] BLINN J. F., NEWELL M. E.: Texture and Reflection in Computer Generated Images. *Comm. of the ACM 19*, 10 (1976), 542–547. 1

[BWB08] BOULOS S., WALD I., BENTHIN C.: Adaptive Ray Packet Reordering. In *Proc. Interactive Ray Tracing* (2008), pp. 131–138. 2, 3

[CG12] CRASSIN C., GREEN S.: Chapter 22: Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer. In *OpenGL Insights*. AK Peters, 2012, pp. 303–320. 3, 6

[Chr08] CHRISTENSEN P.: *Point-Based Approximate Color Bleeding*. Tech. rep., Pixar, 2008. 2

[CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed Ray Tracing. *ACM Trans. Graphics 18*, 3 (1984), 137–145. 1, 2

[DFKP05] DE FLORIANI L., KOBBELT L., PUPPO E.: A Survey on Data Structures for Level-of-Detail Models. In *Advances in Multiresolution for Geometric Modelling*. Springer, 2005, pp. 49–74. 2

[DKB*16] DADO B., KOL T. R., BAUSZAT P., THIERY J.-M., EISEMANN E.: Geometry and Attribute Compression for Voxel Scenes. *Computer Graphics Forum 35*, 2 (2016), 397–407. 10

[DKH*14] DACHSBACHER C., KŘIVÁNEK J., HAŠAN M., ARBREE A., WALTER B., NOVÁK J.: Scalable Realistic Rendering with Many-Light Methods. *Computer Graphics Forum 33*, 1 (2014), 88–104. 1

[DSKA17] DOLONIUS D., SINTORN E., KÄMPE V., ASSARSSON U.: Compressing Color Data for Voxelized Surface Geometry. *IEEE Trans. Visualization and Computer Graphics* (2017). 10

[ED04] EISEMANN E., DURAND F.: Flash Photography Enhancement via Intrinsic Relighting. *ACM Trans. Graphics 23*, 3 (2004), 673–678. 8

[ED06] EISEMANN E., DÉCORET X.: Fast Scene Voxelization and Applications. In *Proc. I3D* (2006), pp. 71–78. 6

[GS10] GEORGIEV I., SLUSALLEK P.: Simple and Robust Iterative Importance Sampling of Virtual Point Lights. In *Proc. Eurographics Short Papers* (2010), pp. 57–60. 2

[HA90] HAEBERLI P., AKELEY K.: The Accumulation Buffer: Hardware Support for High-Quality Rendering. *ACM Trans. Graphics 24*, 4 (1990), 309–318. 1, 2

[Hal98] HALLE M.: Multiple Viewpoint Rendering. In *Proc. SIGGRAPH* (1998), pp. 243–254. 2

[HAM06] HASSELGREN J., AKENINE-MÖLLER T.: An Efficient Multi-View Rasterization Architecture. In *Proc. EGSR* (2006), pp. 61–72. 2

[HMY12] HARADA T., MCKEE J., YANG J. C.: Forward+: Bringing Deferred Lighting to the Next Level. In *Proc. Eurographics Short Papers* (2012). 2

[HPB07] HAŠAN M., PELLACINI F., BALA K.: Matrix Row-Column Sampling for the Many-Light Problem. *ACM Trans. Graphics 26*, 3 (2007), 26. 1, 2

[HREB11] HOLLÄNDER M., RITSCHEL T., EISEMANN E., BOUBEKEUR T.: ManyLoDs: Parallel Many-View Level-of-Detail Selection for Real-Time Global Illumination. *Computer Graphics Forum 30*, 4 (2011), 1233–1240. 1, 2, 4, 6, 7, 8, 11

[HS98] HEIDRICH W., SEIDEL H.-P.: View-Independent Environment Maps. In *Proc. Graphics Hardware* (1998), p. 39ff. 3

[HVAPB08] HAŠAN M., VELAZQUEZ-ARMENDARIZ E., PELLACINI F., BALA K.: Tensor Clustering for Rendering Many-Light Animations. *Computer Graphics Forum 27*, 4 (2008), 1105–1114. 2

[JWSP05] JESCHKE S., WIMMER M., SCHUMANN H., PURGATHOFER W.: Automatic Impostor Placement for Guaranteed Frame Rates and Low Memory Requirements. In *Proc. I3D* (2005), pp. 103–110. 2, 3

[Kel97] KELLER A.: Instant Radiosity. In *Proc. SIGGRAPH* (1997), pp. 49–56. 1, 2

[KSA13] KÄMPE V., SINTORN E., ASSARSSON U.: High Resolution Sparse Voxel DAGs. *ACM Trans. Graphics 32*, 4 (2013), 101. 6

[LES10] LEE S., EISEMANN E., SEIDEL H.-P.: Real-Time Lens Blur Effects and Focus Control. *ACM Trans. Graphics 29*, 4 (2010), 65:1–65:7. 2

[LH13] LEI K., HUGHES J. F.: Approximate Depth of Field Effects using Few Samples per Pixel. In *Proc. I3D* (2013), pp. 119–128. 5

[LWC*03] LUEBKE D., WATSON B., COHEN J. D., REDDY M., VARSHNEY A.: *Level of Detail for 3D Graphics*. Elsevier Science Inc., 2003. 1, 2

[MBJ*15] MATTAUSCH O., BITTNER J., JASPE A., GOBBETTI E., WIMMER M., PAJAROLA R.: CHC+RT: Coherent Hierarchical Culling for Ray Tracing. *Computer Graphics Forum 34*, 2 (2015), 537–548. 2, 3

[MBWW07] MATTAUSCH O., BITTNER J., WONKA P., WIMMER M.: Optimized Subdivisions for Preprocessed Visibility. In *Proc. GI* (2007), pp. 335–342. 2, 3

[MKC07] MARROQUIM R., KRAUS M., CAVALCANTI P. R.: Efficient Point-Based Rendering Using Image Reconstruction. In *Proc. SPBG* (2007), pp. 101–108. 2

[MML12] MCGUIRE M., MARA M., LUEBKE D.: Scalable Ambient Obscurance. In *Proc. HPG* (2012), pp. 97–103. 8

[OA11] OLSSON O., ASSARSSON U.: Tiled Shading. *J. Graphics, GPU, and Game Tools 15*, 4 (2011), 235–251. 2

[OBA12] OLSSON O., BILLETER M., ASSARSSON U.: Clustered Deferred and Forward Shading. In *Proc. HPG* (2012), pp. 87–96. 2

[OBS*15] OLSSON O., BILLETER M., SINTORN E., KÄMPE V., ASSARSSON U.: More Efficient Virtual Shadow Maps for Many Lights. *IEEE Trans. Visualization and Computer Graphics 21*, 6 (2015), 701–713. 2, 8

[OP11] OU J., PELLACINI F.: LightSlice: Matrix Slice Sampling for the Many-Lights Problem. *ACM Trans. Graphics 30*, 6 (2011), 179. 2

[OSK*14] OLSSON O., SINTORN E., KÄMPE V., BILLETER M., ASSARSSON U.: Efficient Virtual Shadow Maps for Many Lights. In *Proc. I3D* (2014), pp. 87–96. 2

[PSA*04] PETSCHNIGG G., SZELISKI R., AGRAWALA M., COHEN M., HOPPE H., TOYAMA K.: Digital Photography with Flash and No-Flash Image Pairs. *ACM Trans. Graphics 23*, 3 (2004), 664–672. 8

[RAH07] ROGER D., ASSARSSON U., HOLZSCHUCH N.: Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU. In *Proc. EGSR* (2007), pp. 99–110. 2, 3, 7

[REG*09] RITSCHEL T., ENGELHARDT T., GROSCH T., SEIDEL H.-P., KAUTZ J., DACHSBACHER C.: Micro-Rendering for Scalable, Parallel Final Gathering. *ACM Trans. Graphics 28*, 5 (2009), 132. 2

[REH*11] RITSCHEL T., EISEMANN E., HA I., KIM J. D. K., SEIDEL H.-P.: Making Imperfect Shadow Maps View-Adaptive: High-Quality Global Illumination in Large Dynamic Scenes. *Computer Graphics Forum 30*, 8 (2011), 2258–2269. 2

[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. Graphics 27*, 5 (2008), 129. 1, 2, 5

[SKALP05] SZIRMAY-KALOS L., ASZÓDI B., LAZÁNYI I., PREMECZ M.: Approximate Ray-Tracing on the GPU with Distance Impostors. *Computer Graphics Forum 24*, 3 (2005), 695–704. 1

[SKE06] STRENGERT M., KRAUS M., ERTL T.: Pyramid Methods in GPU-based Image Processing. In *Proc. VMV* (2006), pp. 169–176. 5

[SS10] SCHWARZ M., SEIDEL H.-P.: Fast Parallel Surface and Solid Voxelization on GPUs. *ACM Trans. Graphics 29*, 6 (2010), 179. 6

[TH16] TOKUYOSHI Y., HARADA T.: Stochastic Light Culling. *J. Computer Graphics Techniques 5*, 1 (2016). 2

[WABG06] WALTER B., ARBREE A., BALA K., GREENBERG D. P.: Multidimensional Lightcuts. *ACM Trans. Graphics 25*, 3 (2006), 1081–1088. 2

[WBSS04] WANG Z., BOVIK A. C., SHEIKH H. R., SIMONCELLI E. P.: Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Trans. Image Processing 13*, 4 (2004), 600–612. 9

[WFA*05] WALTER B., FERNANDEZ S., ARBREE A., BALA K., DONIKIAN M., GREENBERG D. P.: Lightcuts: A Scalable Approach to Illumination. *ACM Trans. Graphics 24*, 3 (2005), 1098–1107. 1, 2, 3, 4

[WHB*13] WANG B., HUANG J., BUCHHOLZ B., MENG X., BOUBEKEUR T.: Factorized Point Based Global Illumination. *Computer Graphics Forum 32*, 4 (2013), 117–123. 2