Habilitation Thesis

Visibility-Based Optimizations for Image Synthesis

Jiří Bittner

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction
Karlovo nám. 13, 12135 Praha 2

Preface

This thesis is a collection of nine selected research papers which I and my co-authors created during last five years. These papers either directly address optimization of visibility computations yielding faster image synthesis algorithms, or they exploit visibility in order to improve the quality of the synthesized images by gaining understanding of the given virtual model. Five of the presented papers were published in impacted international journals, two papers have been recently accepted for a journal publication and will be in print in the next months. Two other papers were published at established conferences in the field.

The thesis contains an introductory part providing the reader a background information. The introduction is followed by a summary of contributions of the presented papers and prospects for future work in the area. The thesis contains paper reprints in the form they have been published or submitted to a publisher.

Prague, June 1st, 2013 Jiří Bittner

Acknowledgements

First of all I would like to express my gratitude to all my collaborators who contributed to the work presented in this thesis. In particular I want to thank Vlastimil Havran, Michael Wimmer, Oliver Mattausch, Peter Wonka, Ladislav Čmolík, Michal Hapala, Marek Vinkler, Tomáš Barák, Ari Silvennoinen, Kaichi Zhou, and Eugene Zhang. I am very grateful to Jiří Žára the head of Department of Computer Graphics and Interaction for creating a stimulating environment for research and education and for his encouragement to finalize this thesis. I also want to express my thanks to all my colleagues from the Department of Computer Graphics and Interaction for many fruitful discussions and support. Last but certainly not least I want to thank my family for their constant support and for their patience particularly in the moments of paper deadlines, including those which are not part of this thesis.

Contents

Chapter 1

Introduction

Image synthesis is the process of creating synthetic images of virtual environments. The final stage of the image synthesis process, often referred to as *rendering*, constitutes an important part of computer graphics as a research discipline. Within the rendering stage one of the key problems to be solved is visibility. At some point practically all rendering methods need to determine visibility either to compute which parts of the virtual environment should appear in the synthesized image or to determine which parts should be illuminated by light sources or by light reflected from scene surfaces. Thus computing visibility is important and in many cases visibility computations actually create a bottleneck of the whole image synthesis process. Optimizing visibility computations in various ways leads to faster image synthesis methods and in turn increases interactivity and realism of the image synthesis applications.

This thesis addresses visibility problems from different application domains: it deals with the long standing issue of real-time rendering of large scale scenes using rasterization, the problem of efficiently solving visibility in ray tracing based algorithms, and it also touches other areas of image synthesis where visibility information can be exploited such as geometrical model analysis and visualization. This chapter provides a background for the work presented in the thesis, Chapter 2 presents an overview of contributions of the work presented in thesis, and Chapter 3 provides a summary and prospects for future work. The reprints of research papers forming the main part of the thesis are provided in appendices.

1.1 Visibility in Rasterization Pipeline

Rasterization is a classical paradigm of image synthesis. The image is synthesized by projecting scene primitives to a projection plane, in which they are sampled into fragments which lie in a discrete raster. In order to synthesize a correct image of the scene we need to resolve visibility of these fragments. In particular we need to determine which of these fragments are the closest to the observer (camera). The most commonly used technique for solving this problem is the *z-buffer* algorithm, which maintains the depth of each of the rasterized fragments. This algorithm exhibits linear time complexity in terms of the number of fragments that are created. It turns out that for highly complex scenes in which many scene primitives do not contribute to the image this linear complexity is not sufficient to achieve desirable rendering performance. One of the possible solutions for improving the performance of rasterization are online occlusion culling methods, which aim to quickly cull whole parts of the scene which are invisible to the camera in order to avoid the rasterization of invisible scene primitives.

The online occlusion culling methods have gained a lot of interest since they can be applied for fully dynamic scenes. The current Graphics Processing Units (GPUs) provide inherent support for occlusion culling by means of *hardware occlusion queries*. In the most common setting the hardware occlusion query determines whether any fragment being rasterized inside the query passes the depth test and thus is visible. However the occlusion queries also come at some cost and the naive usage of these queries does not improve performance for rendering highly complex scenes. Although several methods have been proposed, which address this problem, still for some complex views of the scene the occlusion culling could actually reduce the performance of rendering. The thesis addresses this problem first for rendering images of large scale dynamic scenes (Appendix A), and then it applies the proposed method for efficient rendering of shadows in such scenes (Appendix B).

Rasterization pipeline is not limited to computing only direct illumination. One possible way of increasing the realism of rendered scenes is to include indirect illumination by using the idea of *instant radiosity*, which represents indirect illumination by a large number of virtual point lights (VPLs). Then for each point visible in the image we need to determine visibility of the VPLs in order to compute its illumination, i.e. we need to compute whether the point lies in shadow with respect to each VPL. The traditional shadow computation methods for solving this problem like shadow maps are too slow for interactive applications if the number of VPLs is higher. One possible solution is to use the *imperfect shadow maps* (ISMs), which consist of many approximate low resolution shadow maps

rendered within a single rendering pass using a resampled scene geometry. Appendix C presents a method for improving the ISM based algorithm by adaptive and temporally coherent selection of VPLs. It also describes a GPU friendly method for resampling scene geometry in order to accelerate the ISM creation.

1.2 Ray Tracing Based Visibility

Ray tracing based methods allow to synthesize highly realistic images of virtual environments. The image is computed by tracing a large number of rays which can possibly bounce off the scene surfaces to determine indirect illumination effects like color bleeding, caustics, or reflections. In important class of these methods is based on Monte Carlo integration, which is able to provide an unbiased estimate of the synthesized images. This involves solving a recursive integral, the rendering equation, for all pixels in the image. Within the integral an important factor is visibility, which is either used in the form of determining the closest visible point or determining mutual visibility between two points in the scene. A typical image requires a huge number of rays to be traced and therefore solving the ray tracing (or ray casting) queries efficiently is a key issue.

In order to accelerate ray based visibility computation we need to organize scene primitives in a data structure, which serves as a spatial index. i.e. it allows to focus the ray intersection computations only to objects lying in the proximity of the ray. Many such data structures and associated construction algorithms have been proposed, such as regular grids, hierarchical grids, kd-trees, octrees, or bounding volume hierarchies (BVH). The most efficient solutions nowadays use the *surface area heuristics* (SAH), which is based on a cost model expressing the expected number of operations for solving the visibility queries. This cost model is then used to drive the construction of adaptive data structures such as kd-trees or BVHs. The thesis presents two new approaches improving on the traditional SAH method. The first one is presented in Appendix D and it aims to better model the ray distribution used in the cost model during the construction of kd-trees. The second one presented in Appendix E is an optimization technique for bounding volume hierarchies, which can be used to improve the performance of a BVH initially constructed using an arbitrary method.

Not only it is important to construct efficient hierarchical data structures, but specifically for the case of dynamic scenes it is also important to optimize the speed of the construction process itself. One possibility is to exploit the raw computation power of massively parallel architectures such as GPUs. However parallel construction of hierarchical data structures is complicated as the computation

involves various dependencies which limit the amount of exploitable parallelism. In Appendix F the thesis addresses this topic by providing a general framework for massively parallel hierarchical scene processing which can be used for constructing spatial data structures as well as directly solving ray object intersections.

1.3 Visibility for Model Analysis and Visualization

Visibility-based optimization of image synthesis need not involve the actual optimization of visibility queries, but it can also be used in order to exploit visibility information in different ways. One possibility is to obtain a global knowledge about the visibility in the scene by precomputing visibility for all possible camera positions. The scene is subdivided into *view cells* for which potentially visible sets are computed that represent a superset of objects visible from any point inside the view cell. Precomputing visibility is a complex problem as it involves determining visibility from all possible camera positions towards all possible view directions. Although several analytic solutions to this problem have been proposed it turns out that these solutions are not practical for complex and detailed scenes due to their computational complexity and numeral stability issues. An alternative to the analytic methods are conservative precomputed visibility techniques, which on the other hand can lead to significant overestimation of the potentially visible sets. Appendix G presents a sampling based approach which efficiently explores the domain of all possible sight-lines by adaptive sampling, yielding a progressively improving solution to precomputed visibility. This interactive global visibility solution then allows to use the visibility information in new applications like interactive analysis of visibility hot-spots in level-design for computer games or interactive camera path optimizations for story-telling.

Visibility information can also be used to improve robustness of methods aiming at understanding of the structure of a polygonal model and a potential repair of model errors. Appendix H shows a technique using mutual visibility information in the framework of graph based algorithm revealing the structure of the given model. The method can robustly extract layers of the model and to detect some model errors which would be hard to discover by methods relying only on spatial proximity and connectivity instead of visibility. Appendix I deals with annotation of technical illustration using labels. By using the knowledge of the visible parts of the model and also the visibility of associated labels the presented method shows how to optimize the position of the labels in order to better annotate the underlying model using a particular annotation style.

Chapter 2

Overview of Contributions

This chapter summarizes the main contributions of the work collected in this thesis. The overview is structured into three sections addressing the contributions in the area of rasterization based methods, ray tracing optimization, and methods for model preprocessing, analysis, and visualization.

2.1 Rasterization-Based Methods

The thesis presents three methods targeting rendering optimization within the rasterization pipeline. The first two presented methods address efficient rasterization of large scale scenes both from the camera view as well as the light view. The third method presents an efficient and temporally coherent way to establish a set of VPLs in the context of the imperfect shadow maps based instant radiosity methods.

CHC++: Coherent Hierarchical Culling Revisited. Appendix A presents an algorithm for efficient occlusion culling using hardware occlusion queries. The algorithm significantly improves on previous techniques by making better use of temporal and spatial coherence of visibility. This is achieved by using adaptive visibility prediction and query batching. As a result of the proposed optimizations the number of issued occlusion queries and the number of rendering state changes are significantly reduced. We also propose a simple method for determining tighter bounding volumes for occlusion queries and a method which further reduces GPU pipeline stalls. The proposed method provides up to an order of magnitude speedup over the previous state of the art. The new technique is simple to implement, does not rely on hardware calibration and integrates well with

modern game engines.

Shadow Caster Culling For Efficient Shadow Mapping. Appendix B presents a novel method for efficient construction of shadow maps by culling shadow casters which do not contribute to visible shadows. The method uses a mask of potential shadow receivers to cull shadow casters using a hierarchical occlusion culling algorithm. We propose several variants of the receiver mask implementations with different culling efficiency and computational costs. For scenes with statically focused shadow maps we designed an efficient strategy to incrementally update the shadow map, which comes close to the rendering performance for unshadowed scenes. The proposed method achieves 3x-10x speedup for rendering large city like scenes and 1.5x-2x speedup for rendering an actual game scene.

Temporally Coherent Adaptive Sampling for Imperfect Shadow Maps. Appendix C describes a new adaptive algorithm for determining virtual point lights (VPL) in the scope of real-time instant radiosity methods, which use a limited number of VPLs. The proposed method is based on Metropolis-Hastings sampling and exhibits better temporal coherence of VPLs, which is particularly important for real-time applications dealing with dynamic scenes. We evaluate the properties of the method in the context of the algorithm based on imperfect shadow maps and compare it with the commonly used inverse transform method. The results indicate that the proposed technique can significantly reduce the temporal flickering artifacts even for scenes with complex materials and textures. Further, we propose a novel splatting scheme for imperfect shadow maps using hardware tessellation. This scheme significantly improves the rendering performance particularly for complex and deformable scenes. We thoroughly analyze the performance of the proposed techniques on test scenes with detailed materials, moving camera, and deforming geometry.

2.2 Ray Tracing Optimization

The thesis presents three methods, which aim at optimizing ray tracing based visibility queries. The first two techniques improve the quality of data structures used for ray tracing acceleration, while the third method provides a framework for parallelization of the construction of the hierarchical data structures for ray tracing as well as the parallelization of the ray tracing queries.

RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. Surface area heuristics is currently the most popular method for view independent construction of spatial hierarchies for ray tracing. Appendix D presents a

method which modifies the surface area heuristics by taking into account the actual distribution of rays in the scene. This is achieved by subsampling the rays to be cast and using these rays in order to estimate the probabilities of rays traversing through nodes of the constructed hierarchy. The main aim of our work is to analyze the potential of taking the ray distribution into account. The results indicate that we can achieve a minor speedup of ray traversal compared to standard SAH. For large densely occluded scene we can also save the construction time and memory consumption of the hierarchy by not subdividing parts of the scene where no rays are traced.

Fast Insertion-Based Optimization of Bounding Volume Hierarchies. Appendix E presents an algorithm for fast optimization of bounding volume hierarchies (BVH) for efficient ray tracing. The proposed method performs selective updates of the hierarchy driven by the cost model derived from the surface area heuristic. In each step the algorithm updates a fraction of the hierarchy nodes in order to minimize the overall hierarchy cost. The updates are realized by simple operations on the tree nodes: removal, search, and insertion. Our method can quickly reduce the cost of the hierarchy constructed by the traditional techniques such as the surface area heuristic. We evaluate the properties of the proposed method on fourteen test scenes of different complexity including individual objects and architectural scenes. The results show that our method can improve a BVH initially constructed with the surface area heuristic by up to 27% and a BVH constructed with the spatial median split by up to 88%.

Massively Parallel Hierarchical Scene Processing with Applications in Rendering. Appendix F presents a novel method for massively parallel hierarchical scene processing on the GPU, which is based on sequential decomposition of the given hierarchical algorithm into small functional blocks. The computation is fully managed by the GPU using a specialized task pool which facilitates synchronization and communication of processing units. We present two applications of the proposed approach: construction of the bounding volume hierarchies and collision detection based on divide-and-conquer ray tracing. The results indicate that using our approach we achieve high utilization of the GPU even for complex hierarchical problems which pose a challenge for massive parallelization.

2.3 Model Analysis and Visualization

The thesis presents three different techniques which establish visibility in order to understand the structure of the given virtual model and use this information for further processing, analysis, or visualization. The first method addresses ef-

ficient global visibility computation, the second method analyses the structure of the given model based on visibility relationships, and the last method uses the visibility information in the process of optimizing the label layout for interactive model annotation.

Adaptive Global Visibility Sampling. Appendix G presents a global visibility algorithm which computes from-region visibility for all view cells simultaneously in a progressive manner. The method casts rays to sample visibility interactions and use the information carried by a ray for all view cells it intersects. The main contribution of this work is a set of adaptive sampling strategies based on ray mutations that exploit the spatial coherence of visibility. Our method achieves more than an order of magnitude speedup compared to per-view cell sampling. This provides a practical solution to visibility preprocessing and also enables a new type of interactive visibility analysis applications, where it is possible to quickly inspect and modify a coarse global visibility solution that is constantly refined.

Visibility-driven Mesh Analysis and Visualization through Graph Cuts. Appendix H presents an algorithm that operates on a triangular mesh and classifies each face of a triangle as either inside or outside. We present three example applications of this core algorithm: normal orientation, inside removal, and layer-based visualization. The distinguishing feature of the proposed algorithm is its robustness even if a difficult input model that includes holes, coplanar triangles, intersecting triangles, and lost connectivity is given. Our algorithm works with the original triangles of the input model and uses sampling to construct a visibility graph that is then segmented using graph cut.

Layout-aware optimization for interactive labeling of 3D models. Appendix I describes a novel method for computing labeling of 3D illustrations in real-time. We solve a multiple criteria optimization problem in which we consider the desired layout already in the stage of searching for salient points of the labeled areas. In the solution we employ fuzzy logic combined with greedy optimization. The method runs on the GPU and achieves interactive rates on medium sized models. The results indicate that the method compares favorably to the state-of-the-art interactive labeling techniques.

Chapter 3

Summary and Future Work

The thesis collects results which either directly address optimization of visibility computations yielding faster rendering algorithms or which exploit visibility to gain understanding of the model structure that is further exploited in the process of image synthesis. All methods collected in the thesis provide a contribution in their specific domain. Some of the collected results are very close to the optimal solutions measured by the golden standard methods (particularly the online occlusion culling methods presented in Appendices A and B). However some other results exploit novel directions of research and it is certainly possible to improve on these methods or find a broader range of their potential applications (particularly the methods presented in Appendices E, F, G, and H).

One very promising area of future work are methods providing scalable accuracy of the resulting visibility information. In some cases we do not have to obtain a precisely correct result of visibility queries, which can be exploited for further optimization. A related issue is the analysis of errors caused by approximate visibility computations inside the image synthesis process. The approximate visibility methods might exploit continuous, non-binary, description of visibility. There is a potential in studying how such visibility descriptions could be used in a scalable framework including establishing the error bounds for combining several continuous visibility descriptors together.

Currently there is a strong trend towards massive parallelization of image synthesis computations. This trend involves parallelization of the rendering algorithms as well as algorithms for constructing and maintaining associated data structures. A major challenge in this area is to find a good balance between the adaptivity of the methods and the reduction of computational dependencies required for the massively parallel methods to work efficiently. It is particularly worth to investi-

gate scalable approaches, which are able to resample the scene representation into a form, which adaptively adjust details required for rendering the given image including global illumination effects. Such scalable techniques could also cope with the increasing amount of details present in the scenes both in terms of their geometrical structure as well as material appearance descriptions.

Appendices – Paper Reprints

Appendix A

CHC++: Coherent Hierarchical Culling Revisited

Mattausch, O. - Bittner, J. - Wimmer, M.: CHC++: Coherent Hierarchical Culling Revisited. Computer Graphics Forum. 2008, vol. 27, no. 2, p. 221-230. ISSN 0167-7055. **IF=1.636**

EUROGRAPHICS 2008 / G. Drettakis and R. Scopigno
(Guest Editors)

Volume 27 (2008), Number 3

CHC++: Coherent Hierarchical Culling Revisited

Oliver Mattausch[1], Jiří Bittner[2], Michael Wimmer[1]

[1]Vienna University of Technology, Austria
[2]Czech Technical University in Prague, Czech Republic

Abstract

We present a new algorithm for efficient occlusion culling using hardware occlusion queries. The algorithm significantly improves on previous techniques by making better use of temporal and spatial coherence of visibility. This is achieved by using adaptive visibility prediction and query batching. As a result of the new optimizations the number of issued occlusion queries and the number of rendering state changes are significantly reduced. We also propose a simple method for determining tighter bounding volumes for occlusion queries and a method which further reduces the pipeline stalls. The proposed method provides up to an order of magnitude speedup over the previous state of the art. The new technique is simple to implement, does not rely on hardware calibration and integrates well with modern game engines.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

1. Introduction

Occlusion culling is an important technique to reduce the time for rendering complex scenes. The availability of so-called hardware occlusion queries has made runtime determination of visibility attractive. Hardware occlusion queries are a mechanism by which graphics hardware can quickly report the visibility status of simple proxy geometry. However it was only by exploiting temporal coherence, e.g. in the Coherent Hierarchical Culling (CHC) algorithm [BWPP04], that using hardware occlusion queries became feasible, as this avoids CPU stalls and GPU starvation.

The CHC algorithm works well in densely occluded scenes, but the overhead of hardware occlusion queries makes it fall behind even simple view-frustum culling (VFC) in some situations. This was recognized by Guthe et al. [GBK06], who provide an algorithm, called Near Optimal Hierarchical Culling (NOHC), which reduces the number of queries based on a clever statistical model of occlusion and a hardware calibration step. However, it turns out that even the optimum defined by Guthe et al. can still be improved by exploiting further sources of simplification.

In this paper, we propose CHC++, a method that significantly improves on previous online occlusion culling methods (see Figure 1). The core of the algorithm remains simple, requires no calibration, and allows easy integration into a game engine. The major contributions of the method are:

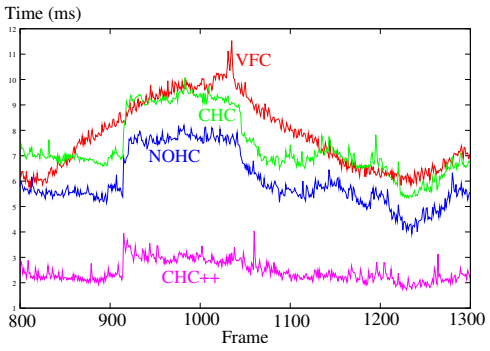


Figure 1: *Frame time comparison for a walkthrough of the Powerplant model for View Frustum Culling (VFC), Coherent Hierarchical Culling (CHC), Near Optimal Hierarchical Culling (NOHC), and our new algorithm (CHC++).*

- **Reduction of state changes.** Despite its importance, the reduction of state changes was not explicitly addressed by previous occlusion culling methods. Our method provides a powerful mechanism to minimize the number of state

changes by using batching of queries. As a result the total number of state changes is reduced by more than an order of magnitude (see Figure 2).

- **Reduction of number of queries.** Reducing the number of queries was a major goal of previous research on hardware based occlusion culling. For example, the NOHC algorithm proposed by Guthe et al. [GBK06] is very successful at reducing the number of queries for views with low occlusion. We propose two new methods for further reduction of the number of queries. The first method resolves visibility of many nodes in the hierarchy by a single query, the second method exploits tighter bounding volumes for the queries without the need for any auxiliary data structures like oriented bounding boxes or k-dops. As a result we achieve a significantly lower number of queries than the "optimal" algorithm defined by Guthe et al. [GBK06] (see Figure 2).
- **Reduction of CPU stalls.** The CHC algorithm does a good job at reducing CPU stalls, however in certain scenarios stalls still occur and cause a performance penalty. We propose a simple modification which provides further reduction of the wait time, which at the same time integrates well with our method for reducing state changes.
- **Reduction of rendered geometry.** Tighter bounding volumes will reduce the overestimation of visibility caused by bounding volumes and therefore reduce the amount of geometry classified as visible.
- **Integration with game engines.** Most game engines incorporate a highly optimized rendering loop in which sorting by materials and shaders is performed in order to minimize rendering state changes. Our method allows the rendering engine to perform such a sort on a batch of primitives stored in a render queue. Additionally the proposed technique significantly reduces the number of engine calls.

2. Related Work

Even in CPU-limited applications, which often occur with today's rapidly evolving graphics hardware, visibility culling can significantly reduce the time spent in the graphics driver and the rendering API, allowing better usage of the graphics hardware. For a general overview of visibility culling please refer to the thorough surveys of Cohen-Or et al. [COCSD02] and Bittner and Wonka [BW03].

Visibility algorithms can be roughly categorized into those that work as a preprocessing step and those that work at runtime. While preprocessing algorithms have no runtime overhead, they are often difficult to implement and work for static scenes only. Online occlusion culling on the other hand does not rely on a lengthy preprocessing step, is potentially more accurate as it computes visibility from a point, and allows for fully dynamic scenes. As most online culling algorithms work in image space, they allow automatic occluder fusion using rasterization. Before dedicated hardware

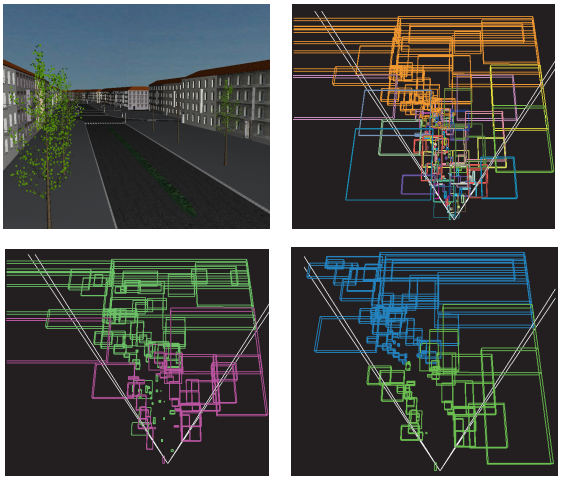


Figure 2: *Top left: A sample view point in a city scene. Top right: State changes required by the CHC algorithm (number of state changes = number of different colors of hierarchy nodes). Bottom left: State changes required by the CHC++ algorithm. Bottom right: Multiqueries: all invisible nodes are covered by only two occlusion queries (shown in different colors).*

support existed, online occlusion culling was mostly considered too costly for practical use, with some notable exceptions such as the Hierarchical Occlusion Maps from Zhang et al. [ZMHH97] or the dPVS system from Aila et al. [AM04].

With the introduction of hardware accelerated occlusion queries, online occlusion culling gained a lot of popularity. Hardware occlusion queries are relatively lightweight instructions that return the number of visible pixels of proxy geometry without the need of reading back the frame buffer. They opened the field for a variety of algorithms [KS01, HSLM02, GSYM03, SBS04, KS05]. However, the queries still come with a cost, so a naive implementation can be very slow due to idle time of the GPU and CPU that is caused by waiting for the query to return.

Coherent Hierarchical Culling [BWPP04], and later Near Optimal Hierarchical Culling [GBK06] avoid idle time by making use of temporal coherence. They will be discussed in more detail in the following section.

3. Overview

In this section we briefly review the CHC and the NOHC algorithms and discuss some of their issues. Then we describe the major components of the new CHC++ algorithm.

3.1. CHC and its problems

The Coherent Hierarchical Culling algorithm [BWPP04] makes use of temporal and spatial coherence to reduce the

overhead and latency of hardware occlusion queries. It traverses the hierarchy in a front-to-back order and issues queries only for previously visible leaves and nodes of the previously invisible boundary. Previously visible leaves are assumed to stay visible in the current frame, and hence they are rendered immediately. The result of the query for these nodes only updates their classification for the next frame. The invisible nodes are assumed to stay invisible, but the algorithm retrieves the query result in the current frame in order to discover visibility changes. Refer to Figure 6 for the pseudocode of the original original CHC algorithm (unmarked parts).

The reduction of the number of queries (queries are not issued on previously visible interior nodes) and clever interleaving reduced the overhead of occlusion queries to an acceptable quantity. The algorithm works very well for scenarios that have a lot of occlusion. However, on newer hardware where rendering geometry becomes cheap compared to querying, or view points where much of the scene is visible, the method can become even slower than conventional view-frustum culling. This is a result of wasted queries and unnecessary state changes. This problem makes the CHC algorithm less attractive for game developers, who call for an algorithm which is reliably faster than view-frustum culling. Another problem of CHC lies in the complicated integration of the method into the rendering loop of highly optimized game engines. CHC interleaves rendering and querying of individual nodes of the spatial hierarchy which does not allow the engine to perform material sorting and leads to a higher number of engine API calls.

3.2. NOHC and its problems

The Near Optimal Hierarchical Culling algorithm proposed by Guthe et al. [GBK06] tackles the problem of wasted queries. The method uses a calibrated model of graphics hardware to estimate costs of the queries and costs of rendering. It estimates occlusion of nodes by using a simple screen coverage model and further corrections assuming temporal coherence. The occlusion estimation and hardware model are used in a cost/benefit heuristics which decides whether to apply an occlusion query on the currently processed node. This heuristic uses a sophisticated reasonability test for queries with a couple of rules.

The algorithm saves a significant number of queries, especially queries which would be applied on visible nodes. This can lead to a significant improvement over the CHC algorithm if the assumed visibility optimization proposed for CHC is not used.

The results for NOHC indicate that with a proper hardware calibration the method always performs better than view-frustum culling. In their paper, Guthe et al. [GBK06] also defined an optimal culling algorithm based on occlusion queries. The optimal algorithm is derived under the assumption that the status of every culled node has to be verified by an occlusion query. The NOHC method then closely approaches the optimal algorithm.

In our paper we show that the definition of optimality used by Guthe et al. [GBK06] still leaves significant room for improvement. In fact, the CHC++ algorithm is always clearly below the optimum defined by Guthe et al.

NOHC requires a hardware calibration step in which the hardware parameters are measured in a preprocess using artificial rendering scenarios. Measuring accurate parameters of the model requires very careful implementation. However, it turns out that even if precisely implemented, these measurements need not reflect the complex processes of state changes, pipelining, and interleaving rendering and occlusion queries during actual walkthroughs. Our new method does not rely on hardware calibration and aims to minimize its dependence on external parameters. In fact it leaves the user with loosely setting a few parameters whose influence is well predictable.

3.3. Building blocks of CHC++

The new CHC++ algorithm method addresses all previously mentioned problems, and extends CHC by including the following new components:

Queues for batching of queries. Before a node is queried, it is appended to a queue. Separate queues are used for accumulating previously visible and previously invisible nodes. We use the queues to issue batches of queries instead of individual queries. This reduces state changes by one to two orders of magnitude. The batching of queries will be described in Section 4.

Multiqueries. We compile multiqueries (Section 5.1), which are able to cover more nodes by a single occlusion query. This reduces the number of queries for previously invisible nodes up to an order of magnitude.

Randomized sampling pattern for visible nodes. We apply a temporally jittered sampling pattern (Section 5.2) for scheduling queries for previously visible nodes. This reduces the number of queries for visible nodes and while spreading them evenly over the frames of the walkthrough.

Tight bounding volumes. We use tight bounding volumes (Section 6) without the need for their explicit construction. This provides a reduction of the number of rendered triangles as well as a reduction of the number of queries.

Note that for all tests presented in the paper we used an axis-aligned bounding volume hierarchy (BVH) constructed according to the surface area heuristics [MB90]. The presented methods are however compatible with other types of spatial hierarchies [MBM*01], except for the tight bounding volumes optimization, which explicitly exploits the properties of BVH.

4. Reducing state changes

Changes of rendering state constitute a significant cost in the rendering pipeline. Previous occlusion culling methods focused mainly on scheduling the queries in a way that hides latencies and keeps the GPU occupied, as well as reducing the overall number of queries. However, even if the number of queries is reduced, every remaining query potentially leads to a state change in which at least writing to color and depth buffers is disabled and then re-enabled after the query. If complex shaders are used then this state change also involves switching the shader on and off.

It turns out that these changes of rendering state cause an even larger overhead than the query itself. The overhead may be on the hardware side (e.g., flushing caches), on the driver side or even on the application side. Thus it is highly desirable to reduce the number of state changes to an acceptable amount: game developers refer to about 200 state changes per frame as an acceptable value on current hardware.

Query batching. Our solution to this problem is based on batching occlusion queries instead of issuing queries immediately when they are requested by the algorithm. The rendering state is changed only once per batch and thus the reduction of state changes directly corresponds to the size of the query batches we issue. The batching algorithm handles visible and invisible nodes differently as described in the following sections.

4.1. Batching previously invisible queries

The invisible nodes to be queried are appended to a queue which we call *i-queue*. When the number of nodes in the i-queue reaches a user-defined batch size b, we change the rendering state for querying and issue an occlusion query for each node in the i-queue (in Section 5.1 we will see how several nodes can be combined in one occlusion query in order to reduce the number of queries).

The batch size b is tightly connected with the reduction of render state changes, giving approximately b times less state changes than the CHC algorithm. On the other hand, batching effectively delays the availability of query results for invisible nodes, which means that visibility changes could be detected later and follow-up queries spawned by them would introduce further latency if there is not enough alternative work (e.g., rendering visible nodes) left.

An optimal value for b depends on the scene geometry, material shaders, and the capabilities of the rendering engine with respect to material sorting. For our scenes and rendering engine we observed that precise tuning of this parameter is not necessary and that values between 20 and 80 give a largely sufficient reduction of render state changes while not introducing additional latency into the method.

4.2. Batching previously visible queries

Recall that the CHC algorithm issues a query for previously visible node and renders the geometry of the node without waiting for the result of the query. Similarly to CHC, our proposed method renders the geometry of previously visible nodes during the hierarchy traversal. However the queries are not issued immediately. Instead the corresponding nodes are stored in a queue which we call *v-queue*.

An important observation is that the queries for these nodes are not critical for the current frame since their result will only be used in the next frame. We exploit this observation by using nodes from the v-queue to fill up waiting time: whenever the traversal queue is empty and no outstanding query result is available, we process nodes from the v-queue.

As a result we perform adaptive batching of queries for previously visible nodes driven by the latency of the outstanding queries. At the end of the frame, when all queries for previously invisible nodes have been processed, the method just applies a single large batch for all unprocessed nodes from the v-queue.

Note that before processing a node from the v-queue, we also check whether a render state change is required. It turns out that in the vast majority of cases there is no need to change the render state at all as it was already changed by a previously issued query batch for invisible nodes. Therefore, we have basically eliminated state changes for previously visible nodes.

As a beneficial side effect, the v-queue reduces the effect of violations of the front-to-back ordering made by the original CHC algorithm. In particular if a previously hidden node occludes a previously visible node in the current frame, this effect would only be captured in the next frame, as the previously visible node would often be queried before the previously invisible node is rendered. This issue becomes apparent in situations where many visibility changes happen at the same time. Delaying the queries using the v-queue will make it more likely for such visibility changes to be detected.

4.3. Game engine integration

For easy integration of the CHC++ method into existing game engines we propose to use an additional queue in the algorithm which we call *render queue*. This queue accumulates all nodes scheduled for rendering and is processed when a batch of queries is about to be issued. When processing the render queue the engine can apply its internal material shader sorting and then render the objects stored in the queue in the new order. Another beneficial effect of the render queue is the reduction of engine API calls. These calls can be very costly and thus their reduction provides significant speedup as we experienced for example with the popular OGRE game engine.

The overview of the different queues used by the CHC++

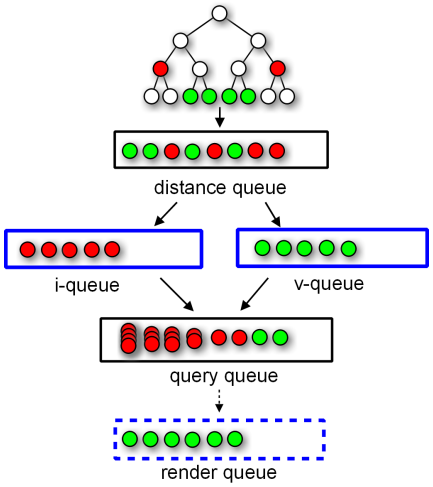


Figure 3: *Different queues used by the CHC++ algorithm. The queues which were not used by the CHC algorithm are highlighted in blue.*

algorithms is shown in Figure 3. Note that the overlaid nodes in the query queue correspond to multiqueries which will be discussed in Section 5.1.

5. Reducing the number of queries

Recent online occlusion culling methods focused on reducing the number of occlusion queries in order to reduce their overhead. In particular the method of Guthe et al. [GBK06] proposed a sophisticated approach for eliminating queries based on a cost/benefit heuristics and a calibrated model of the graphics hardware. In this section we propose two new methods which are able to reduce the number of queries even *below* the hypothetical algorithm previously defined as optimal by Guthe et al.

5.1. Multiqueries for invisible nodes

All previous techniques use one occlusion query per previously invisible primitive to be tested (node in a hierarchy, bounding volume, cell in a grid). The occlusion queries for these nodes were considered irreducible.

However, the following observation allows us to reduce the number of queries even for previously invisible nodes: If some previously invisible part of a scene remains invisible in the current frame, a single occlusion query for the whole part is sufficient to verify its visibility status. Such a query would render all bounding boxes of primitives in this scene part, and return zero if all primitives remain occluded. For example, in the extreme case of a static scene and a static view point, a single occlusion query could be used for all invisible parts of the scene.

Assuming a certain coherence of visibility, our new technique aims to identify such scene parts by forming groups of previously invisible nodes that are equally likely to remain invisible. A single occlusion query is issued for each such group, which we call a *multiquery*. If the multiquery returns zero, all nodes in the group remain invisible and their status has been updated by the single query. Otherwise the coherence was broken for this group and we issue individual queries for all nodes by reinserting them in the i-queue. Note that in the first case the number of queries is reduced by the number of primitives in the group. However, in the second case the multiquery for the batch was wasted.

We use an adaptive mechanism based on a cost/benefit heuristics to find suitable node groupings. The crucial part of the evaluation is the estimation of coherence in the visibility classification of the nodes, which is described in the next section. The actual heuristics will be described in Section 5.1.2.

5.1.1. Estimating coherence of visibility

In the vast majority of cases there is a strong coherence in visibility for most nodes in the hierarchy. Our aim is to quantify this coherence. In particular, knowing the visibility classification of a given node, we aim to estimate the probability that this node will keep its visibility classification in the next frame. Our experiments indicate that there is a strong correlation of this value with the "history" of the node, i.e., with the number of frames the node already kept the same visibility classification (we call this value *visibility persistence*). Nodes that have been invisible for a very long time are likely to stay invisible. Such nodes could be the engine block of a car, for example, that will never be visible unless the camera moves inside of the car engine. On the contrary, even in slow moving scenarios, there are always some nodes on the visible border which frequently change their classification. Hence there is a quite high chance for nodes that recently became invisible to become visible soon.

We define the desired probability as a function of the visibility persistence i, and approximate it based on the history of previous nodes:

$$p_{keep}(i) \approx \frac{n_i^{keep}}{n_i^{all}} \tag{1}$$

where n_i^{keep} is the number of already tested nodes which have been in the same state for i frames and keep their state in the $i+1$-th frame, and n_i^{all} is the total number of already tested nodes which have been in the same state for i frames. Figure 4 shows a plot of the probability p_{keep} against the visibility persistence i. The counters n_i^{keep} and n_i^{all} are accumulated over all previous frames of the walkthrough.

In the first few frames there are not enough measurements

for accurate computation of $p_{keep}(i)$, especially for larger values of i. We solve this problem by piecewise constant propagation of the already computed values to the higher values of $p_{keep}(i)$.

As a simpler alternative to evaluating $p_{keep}(i)$ by measurements, we propose an analytic formula which corresponds reasonably well to the functions we measured for our scenes and walkthroughs:

$$p_{keep}(i) \approx 0.99 - 0.7e^{-i} \tag{2}$$

Using this function does not provide as accurate estimations of p_{keep} as the measured function, but can be used to avoid implementing the evaluation of the measured function. Figure 4 illustrates the analytic function and the measured functions for two different scenes.

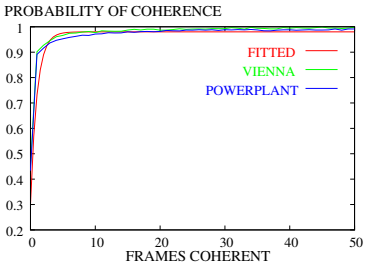


Figure 4: *$p_{keep}(i)$ in dependence on visibility persistence i. Note that the analytic function from Eq. 2 closely matches the functions measured for Powerplant and Vienna scenes.*

5.1.2. Cost/benefit model for multiqueries

Determining the optimal size of a multiquery for previously invisible nodes in a given batch (i.e., the i-queue) is a global optimization problem that requires evaluation of all possible partitions of the batch into multiqueries. Instead, we use a greedy model which maximizes a benefit/cost ratio for each multiquery.

The cost is the expected number of queries issued per one multiquery, which is expressed as:

$$C(M) = 1 + p_{fail}(M) * |M|, \tag{3}$$

where $p_{fail}(M)$ is the probability that the multiquery fails (returns visible, in which case all nodes have to be tested individually) and $|M|$ is the number of nodes in the multiquery. Note that the constant 1 represents the cost of the multiquery itself, whereas $p_{fail}(M) * |M|$ expresses the expected number of additionally issued queries for individual nodes. The probability p_{fail} is calculated from the visibility persistence values i_N of nodes in the multiquery as:

$$p_{fail}(M) = 1 - \prod_{\forall N \in M} p_{keep}(i_N), \tag{4}$$

The benefit of the multiquery is simply the number of nodes in the multiquery, i.e. $B(M) = |M|$.

Given the nodes in the i-queue, the greedy optimization algorithm maximizes the benefit at the given cost. We first sort the nodes in descending order based on their probability of staying invisible, i.e. $p_{keep}(i_N)$. Then, starting with the first node in the queue, we add the nodes to the multiquery and at each step we evaluate the value V of the multiquery as a benefit/cost ratio:

$$V(M_j) = \frac{B(M_j)}{C(M_j)} \tag{5}$$

It turns out that V reaches a maximum for a particular M_j and thus j corresponds to the optimal size of the multiquery for the nodes in the front of the i-queue. Once we find this maximum, we issue the multiquery for the corresponding nodes and repeat the process until the i-queue is used up. As a result we compile larger multiqueries for nodes with high probability of staying invisible and small multiqueries for nodes which are likely to turn visible.

5.2. Skipping tests of visible nodes

The original CHC algorithm introduced an important optimization in order to reduce the number of queries on previously visible nodes. A visible node is assumed to stay visible for n_{av} frames and it will only be tested in the frame $n_{av}+1$. This optimization effectively reduces the average number of queries for previously visible leaves by a factor of $n_{av}+1$.

This simple method however has a problem that the queries can be temporally aligned. This query alignment becomes problematic in situations when nodes tend to become visible in the same frame. For example consider the case when the view point moves from the ground level above the roof level in a typical city scene, causing many nodes to become visible in the same frame. Afterwards the queries of those nodes will be scheduled for the $n_{av}+1$-th frame, and thus most of the queries will be aligned again. The average number of queries per frame will still be reduced, but the alignment can cause observable frame rate drops.

We observed that a randomization of $n_{av}+1$ by a small random value $-r_{max} < r < r_{max}$ does not solve the problem in a satisfying manner. The problem is that if the randomization is small, the queries might still be very much aligned. On the other hand, if the randomization is big, some of the queries will be processed too late and thus the change from visible to invisible state will be captured too late.

We found that the most satisfying solution is achieved by

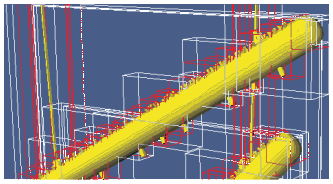


Figure 5: *Tight bounding volumes for nodes of the BVH which more closely represent the objects shown as spheres. The tight bounding volume consists of bounding boxes of the children (red) instead of the parent box (white).*

randomizing the *first invocation* of the occlusion query. After a node has turned visible, we use a random value $0 < r < n_{av}$ for determining the next frame when a query will be issued. Subsequently, if the node was already visible in the previous test, we use a regular sampling interval given by n_{av}.

We experimented with various values of n_{av}. The optimal value depends on the scene itself, inspection coherence, hardware parameters as well as the rendering engine parameters. Fortunately, our tests show that the dependence is not very strong and a value of $5-10$ has been a safe and robust choice for all tests.

6. Tighter bounding volumes

Apart from the overhead introduced by occlusion queries, the success of a culling algorithm depends strongly on how tightly the bounding volumes in the spatial hierarchy approximate the contained geometry. If the fit is not tight enough, many nodes will be classified as visible even though the contained geometry is not. There are several techniques for obtaining tight bounding volumes, mostly by replacing axis-aligned bounding boxes by more complex shapes. While these methods could directly be applied to most occlusion culling algorithms, they also constitute an overhead of calculating and maintaining these volumes. This can become costly especially for dynamic scenes.

We propose a simple method for determining tighter bounds for *inner nodes* in the context of hardware occlusion queries applied to an arbitrary bounding volume hierarchy. For a particular node we determine its *tight bounding volume* as a collection of bounding volumes of its children at a particular depth (see Figure 5).

It turns out that when using up-to-date APIs for rendering the bounding volume geometry (e.g., OpenGL vertex buffer objects), a slightly more complex geometry for the occlusion query practically does not increase its overhead. However, there might be a penalty for rasterizing the tight bounding volumes when some of the smaller bounding primitives overlap in screen space, thus increasing the fill rate compared to projecting the original bounding volume of the node. To avoid such a case, we use a simple test to ensure the usefulness of the tighter bounds. When collecting the child nodes for the tight bounding volume, we test if the sum of surface areas of the bounding volumes of the children is not larger than s_{max} times the surface area of the parent node (note that this does not depend on a particular view point). If this is the case, we terminate traversal and do not further refine the bounding representation. We terminate the search for bounding volumes if the depth from the node is greater than a specified maximal depth d_{max}. The following values gave good results in our tests: $d_{max} = 3$, $s_{max} = 1.4$.

Note that it is advantageous to determine the tight bounding volumes also for *leaves* of the hierarchy. This can be easily achieved by building a slightly deeper hierarchy and then marking interior nodes of the hierarchy containing less than a specified number of triangles as *virtual leaves*, i.e., interior nodes that are considered as leaves during traversal.

As a result, tight bounding volumes provide several benefits at almost no cost: (1) earlier culling of interior nodes of the hierarchy, (2) culling of leaves which would otherwise be classified as visible, (3) increase of coherence of visibility classification of interior nodes. The first property leads to a reduction of the number of queries. The second property provides a reduction of the number of rendered triangles. Finally, the third benefit avoids changes in visibility classification for interior nodes caused by repeated pull-up and pull-down of visibility.

7. Putting it all together

The CHC++ algorithm aims to keep the simplicity of the CHC algorithm, with several important add-ons. In this section we summarize the complete CHC++ algorithm and emphasize its main differences from the CHC algorithm. The pseudocode of the CHC++ algorithm is shown in Figure 6.

As in CHC, we use a priority queue for traversing the hierarchy. This queue provides a front-to-back order of the processed nodes. Unlike CHC, the new algorithm uses two new queues for storing nodes which should be queried (v-queue and i-queue). These two queues are the key for reduction of rendering state changes and compiling multiqueries.

The previously visible nodes are rendered immediately as for CHC. If they are scheduled for testing in the current frame, they are placed in the v-queue. The algorithm for scheduling the queries uses the discussed temporally jittered sampling pattern to reduce the number of queries and to distribute them evenly over frames. The queries for nodes stored in the v-queue are used to fill up the wait time if it should occur. At the end of the frame the remaining nodes in the v-queue form a single batch of queries.

The i-queue accumulates processed nodes which have been invisible in the previous frames. When there is a sufficient number of nodes in the queue, we apply a batch of occlusion queries for nodes in the i-queue while compiling them into the multiqueries.

When integrating the method into a game engine the vis-

```
CHC++ begin
    DistanceQueue.push(Root);
    while !DistanceQueue.Empty() || !QueryQueue.Empty() do
        while !QueryQueue.Empty() do
            if FirstQueryFinished then
                N = QueryQueue.Dequeue();
                HandleReturnedQuery(N);
++          else
++              // next prev. vis. node query;
++              IssueQuery(v-queue.pop());
        if !DistanceQueue.Empty() then
            N = DistanceQueue.DeQueue();
            if InsideViewFrustum(N) then
                if !WasVisible(N) then
                    QueryPreviouslyInvisibleNode(N);
                else
                    if N.IsLeaf && QueryReasonable(N)
                    then
++                      v-queue.push(N);
                    TraverseNode(N);
++      if DistanceQueue.Empty() then
++          // issue remaining query batch;
++          IssueMultiQueries();
++  while !v-queue.empty() do
++      // remaining prev. visible node queries;
++      IssueQuery(v-queue.pop());
end

TraverseNode(N) begin
    if IsLeaf(N) then
        Render(N);
    else
        DistanceQueue.PushChildren(N);
        N.IsVisible = false;
end
PullUpVisibility(N) begin
    while !N.IsVisible do
        N.IsVisible = true; N = N.Parent;
end
HandleReturnedQuery(Q) begin
    if Q.visiblePixels > threshold then
++      if Q.size() > 1 then
++          QueryInvididualNodes(Q); // failed multiquery
        else
            if !WasVisible(N) then
                TraverseNode(N);
            PullUpVisibility(N);
    else
        N.IsVisible = false;
end
++ QueryPreviouslyInvisibleNode(N) begin
++      i-queue.push(N) ;
++      if i-queue.size() ≥ maxPrevInvisNodesBatchSize then
++          IssueMultiQueries(); // issue the query batch
++  end
++ IssueMultiQueries() begin
++      while !i-queue.Empty() do
++          MQ = i-queue.GetNextMultiQuery();
++          IssueQuery(MQ); i-queue.PopNodes(MQ);
++  end
```

Figure 6: *Pseudo-code of the CHC++ main traversal loop and selected important functions. The differences to the original CHC are marked in blue.*

ible nodes are first accumulated in a render queue. The render queue is then processed by the engine before a batch of queries from i-queue is about to be issued.

8. Results

For all our results we used an Intel Quad Core 2.66 MHz CPU and an NVidia 8800 GTX graphics card. We tested our method on three different scenes: Vienna, a typical city scene with detailed street objects and trees (2,583,674 triangles and 10,535 BVH nodes); Pompeii, a generated city scene with detailed buildings (5,646,041 triangles and 22,468 BVH nodes), and the Powerplant model (12,748,510 triangles and 17,793 BVH nodes). In all plots we consistently use the following abbreviations: VFC for View-Frustum Culling, CHC for Coherent Hierarchical Culling, NOHC for Near Optimal Hierarchical Culling, and CHC++ for our new method. For all our measurements of CHC++ we used the following parameters: assumed visible frames $n_{av} = 10$, batch size $b = 50$, maximal depth for tighter bounds $d_{max} = 3$, and the maximal surface area increase for tighter bounds $s_{max} = 1.4$. Note that all walkthroughs shown here are included in the accompanying videos.

Figure 1, shown in the beginning of the paper, presents a frame time comparison for a walkthrough in the Powerplant. It can be seen that the CHC algorithm performs worse than view-frustum culling for some parts of the walkthrough. While NOHC is at least not worse than view-frustum culling, our algorithm performs up to two times better than NOHC.

Figure 7 shows the frame times in a walkthrough in Pompeii and studies the behavior with respect to NOHC and two artificial reference algorithms (NOHC-OPT and OPT). The NOHC-OPT method refers to the function defined as optimum by Guthe et al. [GBK06], which issues queries for all invisible nodes but only if they are feasible according to their cost model. The OPT method refers to an hypothetical algorithm that will only render the visible nodes of the hierarchy without issuing any query. OPT therefore does not depend on the cost or implementation of occlusion queries at all and is the fastest solution that can be achieved with a given hierarchy. We implemented the OPT method by recording the vis-

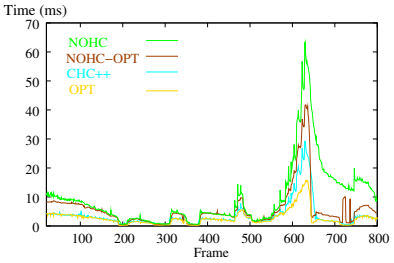


Figure 7: *Frame time comparison of NOHC, the optimal algorithm as defined by Guthe et al. (OPT-NOHC), CHC++, and an algorithm that renders only visible nodes without querying (OPT) in the Pompeii scene.*

ibility results using the exact stop-and-wait algorithm which does not make use of temporal coherence.

As claimed by the authors, NOHC is very close to NOHC-OPT algorithm, except for difficult view points with a lot of visible geometry. More notably, CHC++ is clearly significantly faster than NOHC-OPT practically everywhere. Furthermore, CHC++ is approaching the OPT curve for the moderately complex parts of the scene, which is remarkable since OPT cannot be beaten by any algorithm using occlusion queries on the given hierarchy.

There is still some noticeable overhead of CHC++ compared to OPT in the high frame time parts of the walkthrough, which correspond to views from over the houses where a lot of the scene becomes visible and we have to issue many queries to capture the change in visibility. The rest of the time difference is caused by an accumulation of minor things, like the overhead for maintaining all the queues.

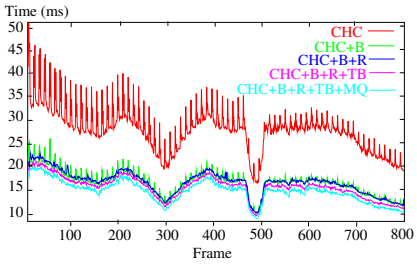


Figure 8: *The benefit of different optimizations in a walkthrough of the Powerplant. We start with CHC and add one optimization at a time. The bottom curve with all optimizations corresponds to CHC++. The abbreviations have the following meaning: B = batching of previously visible and invisible nodes, R = randomization, TB = tighter bounds, MQ = multiqueries.*

Figure 8 shows the benefit of each optimization in another walkthrough in the Powerplant. It is clearly visible that the batching brings the majority of the benefit. Query batching already removes a lot of the query overhead, otherwise the benefit of some of the other optimizations would be much more prominent. The randomization is most important in situations when many nodes become visible at once, which is well visible in the beginning of the walkthrough. The benefit of multiqueries depends on the absolute number of previously invisible nodes, which in turn depends on the properties of the hierarchy (a deeper hierarchy would mean more benefit from multiqueries). Note that the relative benefits of the different optimizations can change for different hardware architectures and rendering engines.

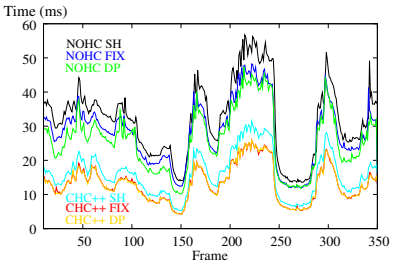


Figure 9: *Dependence of the frame time on shader complexity in a walkthrough in the Powerplant. DP refers to depth only pass, FIX to fixed pipeline shading only, SH to a shader of moderately high complexity.*

In Figure 9 we study how CHC++ and NOHC behave on a walkthrough in the Powerplant with respect to *shader complexity*. We made three different tests: In the first test we used a depth only pass (DP), in the second test we used the standard fixed pipeline material shading of the original Powerplant model (FIX). In the third test we applied a moderately complex shader to all renderable geometry (i.e., the shader has 40 texture lookups).

Note that the used walkthrough is challenging for methods that exploit coherence because it has many swift changes in visibility. As can be seen, the dependence on the shader complexity is very low for CHC++. NOHC shows a much stronger dependence, performing visibly better for the depth pass than for the shaded geometry. Still the depth pass is much slower than for CHC++. Obviously the state changes lower the performance for the depth-only pass as well, even if it only involves a switch of the depth write flag.

Figure 10 analyzes the behavior of all methods in the Vienna scene, particularly with respect to the number of *queries and state changes*. This figure shows that CHC++ fulfills the claim that it significantly reduces both queries and state changes, and that this also translates into a significant performance advantage over the other algorithms.

9. Conclusions

We proposed several modifications to the CHC algorithm [BWPP04]. These modifications provide a significant

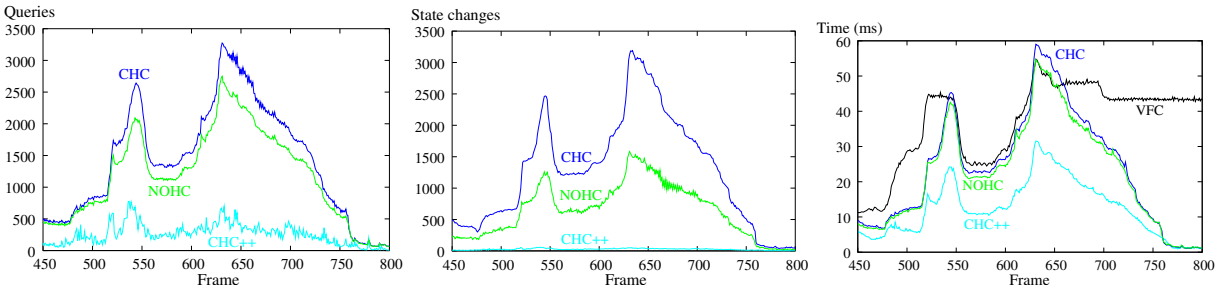


Figure 10: *Comparison of issued queries (left), state changes (middle), and the resulting frame rates for a walkthrough in Vienna. Note that VFC does not impose any additional state changes.*

reduction of state changes, number of queries, rendered triangles, and a further reduction of pipeline stalls. These benefits are achieved by batching of occlusion queries, multi-queries which cover more nodes with a single query, a randomly jittered temporal sampling pattern for queries, and tighter bounding volumes.

The results show that compared to previous methods, the new method provides up to two orders of magnitude reduction in the number of state changes and up to one order of magnitude reduction in the number of queries. These savings translate into a twofold speedup compared to CHC and about 1.5x speedup compared to NOHC [GBK06]. The proposed method is for most cases within a few percent of the "ideal" method which would know visibility classification in advance and render visible geometry only without using any occlusion queries. We believe that the new algorithm will become useful for game programmers as it is stable, easy to implement, and it integrates well with game engines.

In the future we want to study the possibility of automatic parameter adaptation during the walkthrough by exploiting the dependence of the total frame time on the number of issued queries and rendered triangles.

Acknowledgments

This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research program LC-06008 (Center for Computer Graphics), the Aktion Kontakt grant no. 48p11, and the EU under the FP6 project no. IST-014891-2 (Crossmod)

References

[AM04] AILA T., MIETTINEN V.: dpvs: An occlusion culling system for massive dynamic environments. *IEEE Comput. Graph. Appl. 24*, 2 (2004), 86–97.

[BW03] BITTNER J., WONKA P.: Visibility in computer graphics. *Environment and Planning B: Planning and Design 30*, 5 (sep 2003), 729–756.

[BWPP04] BITTNER J., WIMMER M., PIRINGER H., PURGATHOFER W.: Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3 (Sept. 2004), 615–624. Proceedings EUROGRAPHICS 2004.

[COCSD02] COHEN-OR D., CHRYSANTHOU Y., SILVA C., DURAND F.: A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics.* (2002).

[GBK06] GUTHE M., BALÁZS A., KLEIN R.: Near optimal hierarchical culling: Performance driven use of hardware occlusion queries. In *Eurographics Symposium on Rendering 2006* (June 2006), Akenine-Möller T., Heidrich W., (Eds.), The Eurographics Association.

[GSYM03] GOVINDARAJU N. K., SUD A., YOON S.-E., MANOCHA D.: Interactive visibility culling in complex environments using occlusion-switches. In *SI3D* (2003), pp. 103–112.

[HSLM02] HILLESLAND K., SALOMON B., LASTRA A., MANOCHA D.: *Fast and Simple Occlusion Culling Using Hardware-Based Depth Queries*. Tech. Rep. TR02-039, Department of Computer Science, University of North Carolina - Chapel Hill, Sept. 12 2002.

[KS01] KLOSOWSKI J. T., SILVA. C. T.: Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics 7*, 4 (Oct. 2001), 365–379.

[KS05] KOVALCIK V., SOCHOR J.: Occlusion culling with statistically optimized occlusion queries. In *WSCG (Short Papers)* (2005), pp. 109–112.

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 6 (1990), 153–65. criteria for building octree (actually BSP) efficiency structures.

[MBM*01] MEISSNER M., BARTZ D., MUELLER G., HUETTNER T., EINIGHAMMER J.: Generation of decomposition hierarchies for efficient occlusion culling of large polygonal models. In *Proceedings of the Vision Modeling and Visualization Conference 2001* (Nov. 21–23 2001), pp. 225–232.

[SBS04] STANEKER D., BARTZ D., STRASSER W.: Occlusion culling in OpenSG PLUS. *Computers and Graphics 28*, 1 (Feb. 2004), 87–92.

[ZMHH97] ZHANG H., MANOCHA D., HUDSON T., HOFF III K. E.: Visibility culling using hierarchical occlusion maps. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), Whitted T., (Ed.), Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 77–88. ISBN 0-89791-896-7.

Appendix B

Shadow Caster Culling For Efficient Shadow Mapping

Bittner, J. - Mattausch, O. - Silvennoinen, A. - Wimmer, M.: Shadow caster culling for efficient shadow mapping. In Proceedings of Symposium on Interactive 3D Graphics and Games. New York: ACM Press, 2011, p. 81-88. ISBN 978-1-4503-0565-5.

Shadow Caster Culling for Efficient Shadow Mapping

Jiří Bittner* Oliver Mattausch† Ari Silvennoinen‡ Michael Wimmer†

*Czech Technical University in Prague§ † Vienna University of Technology ‡Umbra Software

Figure 1: *(left) A view of a game scene rendered with shadows. (right) Shadow map with shadow casters rendered using a naive application of occlusion culling in the light view (gray), and shadow casters rendered using our new method (orange). The view frustum corresponding to the left image is shown in yellow. Note that for this view our shadow caster culling provided a 3x reduction in the number of rendered shadow casters, leading to a 1.5x increase in total frame rate. The scene is a part of the Left 4 Dead 2 game (courtesy of Valve Corp.).*

Abstract

We propose a novel method for efficient construction of shadow maps by culling shadow casters which do not contribute to visible shadows. The method uses a mask of potential shadow receivers to cull shadow casters using a hierarchical occlusion culling algorithm. We propose several variants of the receiver mask implementations with different culling efficiency and computational costs. For scenes with statically focused shadow maps we designed an efficient strategy to incrementally update the shadow map, which comes close to the rendering performance for unshadowed scenes. We show that our method achieves 3x-10x speedup for rendering large city like scenes and 1.5x-2x speedup for rendering an actual game scene.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism— [I.3.5]: Computer Graphics—Computational Geometry and Object Modeling

Keywords: shadow maps, occlusion culling, real time rendering

1 Introduction

Shadow mapping is a well-known technique for rendering shadows in 3D scenes [Williams 1978]. With the rapid development of graphics hardware, shadow maps became an increasingly popular technique for real-time rendering. A variety of techniques have been proposed, most of which focus on increasing the quality of the rendered image given a maximum resolution of the shadow map [Stamminger and Drettakis 2002; Wimmer et al. 2004; Lloyd et al. 2008]. Given sufficiently high resolution, these methods achieve shadows of very high quality.

For highly complex scenes, however, shadow mapping can become very slow. One problem is the rendering of the main camera view. This problem has been addressed by occlusion culling, which aims to quickly cull geometry which does not contribute to the image [Cohen-Or et al. 2003]. However, just rendering the camera view quickly does not guarantee high frame rates when shadow mapping is used. In particular, the overhead of creating the shadow map is not reduced. Thus, rendering the shadow map can easily become the bottleneck of the whole rendering pipeline as it may require rendering a huge amount of shadow casters at a very high resolution.

A naive solution to this problem would again use occlusion culling to reduce the amount of rendered shadow casters when rendering the shadow map. However it turns out that for common complex scenes like terrains or cities, the lights are set up so that they have a global influence on the whole scene (e.g., sun shining over the city). For such scenes, occlusion culling from the light view will not solve the problem, as the depth complexity of the light view is rather low and thus not much geometry will be culled. Thus even if occlusion culling is used for both the camera view and the light view, we might end up rendering practically all geometry contained in the intersection of the view frustum and the light frustum, and many rendered shadow casters will not contribute to the shadows in the final image.

In this paper, we propose a method for solving this problem by using the knowledge of visibility from the camera for culling the shadow casters. First, we use occlusion culling for the camera view to identify visible shadow receivers. Second, when rendering into the shadow map, we use only those shadow casters which cast shadows on visible shadow receivers. All other shadow casters are culled. For both the camera and the light views, we use an improved version of the coherent hierarchical culling algorithm (CHC++) [Mattausch et al. 2008], which provides efficient schedul-

*e-mail:bittner@fel.cvut.cz
†e-mail:{matt|wimmer}@cg.tuwien.ac.at
‡e-mail:ari@umbrasoftware.com
§Faculty of Electrical Engineering

ing of occlusion queries based on temporal coherence. We show that this method brings up to an order of magnitude speedup compared to naive application of occlusion culling to shadow mapping (see Figure 1). As a minor contribution we propose a simple method for incrementally updating a statically focused shadow map for scenes with moving objects.

2 Related Work

Shadow mapping was originally introduced by Williams [1978], and has meanwhile become the de-facto standard shadow algorithm in real-time applications such as computer games, due to its speed and simplicity. For a discussion of different methods to increase the quality of shadow mapping, we refer to a recent survey [Scherzer et al. 2010]. In our approach in particular, we make use of light space perspective shadow mapping (LiSPSM) [Wimmer et al. 2004] in order to warp the shadow map to provide higher resolution near the viewpoint, and cascaded shadow maps [Lloyd et al. 2006], which partition the shadow map according to the view frustum with the same aim.

There has been surprisingly little work on shadow mapping for large scenes. One obvious optimization which also greatly aids shadow quality is to focus the light frustum on the receiver frustum [Brabec et al. 2002]. As a result, many objects which do not cast shadows on objects in the view frustum are culled by frustum culling in the shadow map rendering step. Lauritzen et al. [2010] have reduced the focus region to the subset of visible view samples in the context of optimizing the splitting planes for cascaded shadow maps.

Govindaraju et al. [2003] were the first to use occlusion culling to reduce the amount of shadow computations in large scenes. Their algorithm computes object-precision shadows mostly on the CPU and used occlusion queries on a depth map to provide the tightest possible bound on the shadow polygons that have to be rendered. While their algorithm also renders shadow receivers to the stencil buffer to identify potential shadow casters, this was done *after* a depth map had already been created, and thus this approach is unsuitable for shadow mapping acceleration. In our algorithm, on the other hand, the creation of the depth map itself is accelerated, which is a significant benefit for many large scenes. Lloyd et al. [2004] later extended shadow caster culling by shadow volume clamping, which significantly reduced the fillrate when rendering shadow volumes. Their paper additionally describes a method for narrowing the stencil mask to shadow receiver fragments that are detected to lie in shadow. Décoret [2005] proposed a different method for shadow volume clamping as one of the applications for N-buffers. Unlike Lloyd et al. [2004] Décoret enhances the stencil mask by detecting receiver fragments which are visible from the camera. The methods for shadow volume culling and clamping have been extended by techniques allowing for more efficient GPU implementations[Eisemann and Décoret 2006; Engelhardt and Dachsbacher 2009].

While our method shares the idea of using visible receivers to cull shadow casters proposed in the above mentioned techniques, in these methods the shadow map served only as a proxy to facilitate shadow volume culling and clamping, and the efficient construction of the shadow map itself has not been addressed. When shadow mapping is used instead of more computationally demanding shadow volumes, the bottleneck of the computation moves to the shadow map construction itself. This bottleneck, which we address in the paper, was previously left intact.

3 Algorithm Outline

The proposed algorithm consists of four main steps:

(1) Determine shadow receivers

(2) *Create a mask of shadow receivers*

(3) *Render shadow casters using the mask for culling*

(4) Compute shading

The main contribution of our paper lies in steps (2) and (3). To give a complete picture of the method we briefly outline all four steps of the method.

Determine shadow receivers The potential shadow receivers for the current frame consist of all objects visible from the camera view, since shadows on invisible objects do not contribute to the final image. Thus we first *render* the scene from the point of view of the camera and determine the *visibility* status of the scene objects in this view.

To do both of these things efficiently, we employ hierarchical occlusion culling [Bittner et al. 2004; Guthe et al. 2006; Mattausch et al. 2008]. In particular we selected the recent CHC++ algorithm [Mattausch et al. 2008] as it is simple to implement and provides a good basis for optimizing further steps of our shadow caster culling method. CHC++ provides us with a visibility classification of all nodes of a spatial hierarchy. The potential shadow receivers correspond to visible leaves of this hierarchy.

Note that in contrast to simple shadow mapping, our method requires a render pass of the camera view before rendering the shadow map. However, such a rendering pass is implemented in many rendering frameworks anyway. For example, deferred shading, which has become popular due to recent advances in screen-space shading effects, provides such a pass, and even standard forward-renderers often utilize a depth-prepass to improve pixel-level culling.

Create a mask of shadow receivers The crucial step of our algorithm is the creation of a mask in the light view which represents shadow receivers. For a crude approximation, this mask can be formed by rendering bounding boxes of visible shadow receivers in the *stencil buffer* attached to the shadow map. In the next section, we propose several increasingly sophisticated methods for building this mask, which provide different accuracy vs. complexity tradeoffs.

Render shadow casters We use hierarchical visibility culling using hardware occlusion queries to speed up the shadow map rendering pass. In addition to depth-based culling, we use the shadow receiver mask to cull shadow casters which do not contribute to visible shadows. More precisely, we set up the *stencil test* to discard fragments outside the receiver mask. Thus, the method will only render shadow casters that are visible from the light source *and* whose projected bounding box at least partly overlaps the receiver mask.

Compute shading The shadow map is used in the final rendering pass to determine light source visibility for each shaded fragment.

4 Shadow Caster Culling

The general idea of our method is to create a mask of visible shadow receivers, i.e., those objects determined as visible in the first camera rendering pass. This mask is used in the shadow map rendering pass

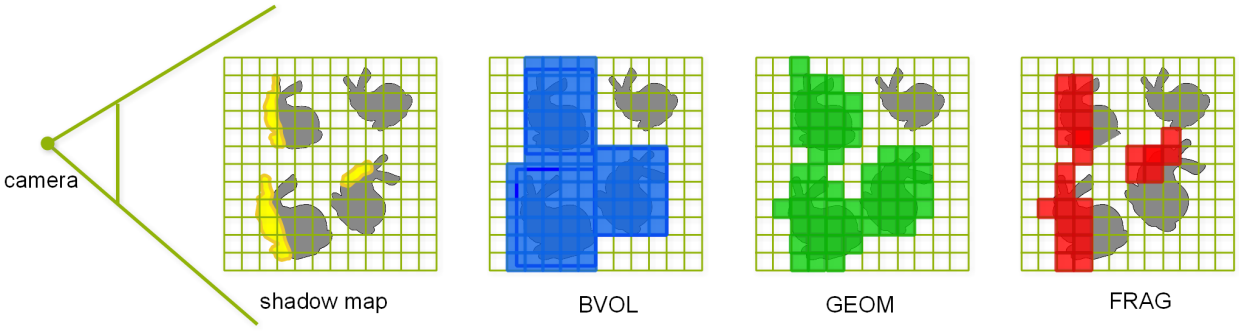


Figure 2: *Visualization of different receiver masks. On the left there is a light space view of the scene showing surfaces visible from the camera in yellow. On the right three different receiver masks are shown. Note that the invisible object on the top right does not contribute to any of the receiver masks.*

together with the depth information to cull shadow casters which do not contribute to the visible shadows. Culling becomes particularly efficient if it employs a spatial hierarchy to quickly cull large groups of shadow casters, which is the case for example in the CHC++ algorithm.

There are various options on how the receiver mask can be implemented. We describe four noteworthy variants which differ in implementation complexity as well as in culling efficiency. The creation of the mask happens in a separate pass before rendering the shadow map, but may already generate parts of the shadow map itself as well, simplifying the subsequent shadow map rendering pass.

We start our description with a simple version of the mask, which is gradually extended with more advanced culling. As we shall see later in the results section, in most cases the more accurate mask provides better results, although there might be exceptions for particular scene and hardware configurations.

4.1 Bounding Volume Mask

The most straightforward way to create the mask is to rasterize *bounding volumes* of all visible shadow receivers into the stencil buffer attached to the shadow map.

Such a mask will generally lead to more shadow casters being rendered than actually necessary. The amount of overestimation depends mostly on how fine the subdivision of scene meshes into individual objects is, and how tightly their bounding volumes fit. If an individual scene object has too large spatial extent, its projection might cover a large portion in the shadow mask even if only a small part of the mesh is actually visible. An illustration of the bounding volume mask is shown in Figure 2 (BVOL).

4.2 Geometry Mask

In order to create a tighter shadow receiver mask, we can rasterize the *actual geometry* of the visible shadow receivers instead of their bounding volumes into the stencil buffer.

While rendering the visible shadow receivers, we also write the depth values of the shadow receivers into the shadow map. This has the advantage that these objects have thus already been rendered into the shadow map and can be skipped during the subsequent shadow map rendering pass, which uses occlusion queries just for the remaining objects. An illustration of the bounding volume mask is shown in Figure 2 (GEOM).

In some scenes many shadow receivers also act as shadow casters and thus this method will provide a more accurate mask at almost no cost. However if most shadow receivers do not act as shadow casters (i.e., most of the scene is shadowed by objects outside of the camera view), the shadow mask creation can become rather expensive, since most parts of the shadow map will be replaced by other objects which act as real shadow casters. Consequently, the resources for rendering these shadow receivers into the shadow map were wasted, as they would have been culled by the occlusion culling algorithm otherwise.

4.3 Combined Geometry and Bounding Volume Mask

In order to combine the positive aspects of the two previously described methods, we propose a technique which decides whether a shadow receiver should fill the mask using its bounding box or its geometry. The decision is based on the estimation of whether the visible shadow receiver will simultaneously act as a shadow caster. Such objects are rendered using geometry (with both stencil and depth, as such objects must be rendered in the shadow map anyway), while all other visible receivers are rendered using bounding boxes (with only stencil write).

The estimation uses temporal coherence: if a shadow receiver was visible in the shadow map in the previous frame, it is likely that this receiver will stay visible in the shadow map and therefore also act as a shadow caster in the current frame.

Again, all objects that have already been rendered into the shadow map using depth writes can be skipped in the shadow map rendering pass. However, in order to estimate the receiver visibility in the subsequent frame, visibility needs to be determined even for these skipped objects. Therefore we include them in the hierarchical occlusion traversal and thus they may have their bounding volumes rasterized during the occlusion queries.

4.4 Fragment Mask

Even the most accurate of the masks described above, the geometry mask, can lead to overly conservative receiver surface approximations. This happens when a receiver object spans a large region of the shadow map even though most of the object is hidden in the camera view. An example would be a single ground plane used for a whole city, which would always fill the whole shadow receiver mask and thus prevent any culling. While such extreme cases can be avoided by subdividing large receiver meshes into smaller objects, it is still beneficial to have a tighter mask which is completely independent of the actual object subdivision.

Method	accuracy	fill rate	transform rate
BVOL	low	high	low
GEOM	medium	medium	medium
GEOM+BVOL	medium	low–medium	low
FRAG	high	medium	medium

Table 1: *Summary of the masking techniques and indicators of their main attributes. BVOL – bounding box, GEOM – geometry mask, GEOM+BVOL – combined geometry and bounding volume mask, FRAG – fragment mask. Note that the FRAG method either relies on availability of direct stencil writes, or requires slightly more effort in mask creation and culling phases as described in Section 4.4.*

Fortunately, we can further refine the mask by using a fragment level receiver visibility tests for all shadow receivers where the full geometry is used for the mask. While creating the mask by rasterizing the geometry in light space, we project each fragment back to view space and test the projected fragment visibility against the view space depth buffer. If the fragment is invisible, it is on a part of the shadow receiver that is hidden, and is thus not written to the mask, otherwise, the mask entry for this fragment is updated. The shadow map depth needs to be written in *both* cases, since even if the fragment is not visible in the camera view, it might still be a shadow caster that shadows another visible shadow receiver.

A similar mask construction approach was proposed by Décoret [2005] for culling and clamping shadow volumes of shadow casters. Décoret used a pair of *litmaps* to obtain minimum and maximum depths of visible receivers per texel in order to clamp shadow volumes. In his method all scene objects are processed twice in the mask construction phase and the method does not use the knowledge of visible shadow receivers. In contrast, we use a single binary mask of visible receiver fragments and combine the construction of the mask with simultaneous rendering of depths of objects which act both as shadow receivers and shadow casters, which is very important to reduce the bottleneck created by the shadow map construction.

The mask constructed using the fragment level visibility tests is pixel accurate, i.e., only locations that can receive a visible shadow are marked in the mask (up to the bias used for the fragment depth comparison). Note that the fragment visibility test corresponds to a "reversed" shadow test, i.e., the roles of the camera and the light are swapped: instead of testing the visibility of the camera view fragment with respect to the shadow map, we test visibility of the fragment rendered into the shadow map with respect to the camera using the camera view depth buffer. An illustration of the bounding volume mask is shown in Figure 2 (FRAG).

The implementation of this approach faces the problem that the stencil mask needs to be updated based on the outcome of the shader. This functionality is currently not widely supported in hardware. Therefore, we implement the fragment receiver mask as an additional texture render target instead of using the stencil buffer. This requires an additional texture fetch and a conditional fragment discard based on the texture fetch while rendering the occlusion queries in the shadow map rendering pass, which incurs a small performance penalty compared to a pure stencil test.

The summary of different receiver masking methods is given in Table 1. The illustration of the culling efficiency of different masks is shown in Figure 3. Note that apart from the BVOL method, all other methods initiate the depth buffer values for the light view with depths of visible receivers.

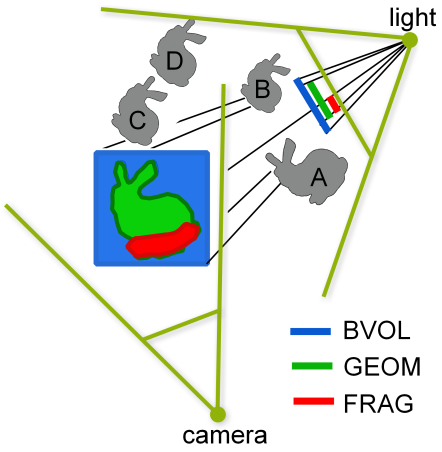


Figure 3: *2D illustration of the culling efficiency of different masking strategies. The figure shows an example of the scene with a shadow receiver object and different types of shadow receiver masks. Object A is always rendered as it intersects all masks, object B is culled only by the FRAG mask, C is culled by depth test (occluded by B) and by the FRAG and GEOM masks, and finally D is culled by all types of masks.*

5 Further Optimizations

This section contains several optimization techniques which can optionally support the proposed receiver masking in further enhancing the performance of the complete rendering algorithm.

5.1 Incremental Shadow Map Updates

If the shadow map is statically focused, we can extend the approach by restricting the shadow receiver mask only to places where a potential change in shadow can happen. We can do so by building the shadow receiver mask only from objects involved in dynamic changes. In particular we render all moving objects into the shadow receiver mask in their previous and current positions. This pass is restricted only to dynamic objects which are either visible in this or the previous frame.

If only a fraction of the scene objects is transformed, the shadow receiver mask becomes very tight and the overhead for shadow map rendering becomes practically negligible. However, this optimization is only useful if a static shadow map brings sufficient shadow quality compared to a shadow map focused on the view frustum.

Note that in order to create the stencil mask and simultaneously clear the depth buffer in a selective fashion, we reverse the depth test (only fragments with greater depth pass) and set the fragment depth to 1 in the shader while rendering the bounding boxes of moving objects.

5.2 Visibility-Aware Shadow Map Focusing

Shadow maps are usually "focused" on the view frustum in order to increase the available shadow map resolution. If visibility from the camera view is available, focusing can be improved further [Lauritzen et al. 2010]: The light frustum can then be computed by calculating the convex hull of the light source and the set of bounding boxes of the *visible* shadow receivers. Note that unlike the method of Lauritzen et al. [2010], our focusing approach does not require a view sample analysis on the GPU, but uses the readily available visibility classification from the camera rendering pass.

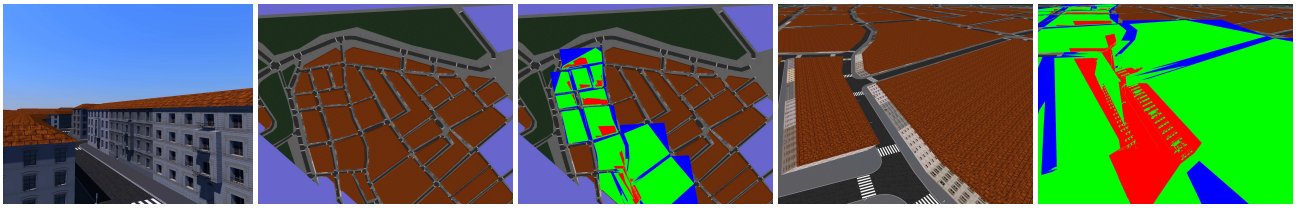


Figure 4: *This figure shows a viewpoint in the Vienna scene, the corresponding light view, and a visualization of the different versions of receiver masks where blue indicates the bounding volume mask, green the geometry mask, and red the fragment mask.*

This simple method effectively increases the resolution of the shadow map, as it is focused only on the visible portion of the scene. However this technique also has a drawback: if a large visibility change occurs in the camera view, the focus of the light view changes dramatically from one frame to the next, which can be perceived as temporal alias or popping of the shadow. This can easily happen in scenes in which a distant object suddenly pops up on the horizon such as a tall building, flying aircraft or a mountain.

5.3 CHC++ Optimizations

Our new shadow culling method works with any occlusion culling algorithm that allows taking the stencil mask into account. We use coherent hierarchical culling (CHC++ [Mattausch et al. 2008]) to implement hierarchical occlusion queries both for rendering the camera view and for rendering the shadow map. For the shadow map, occlusion queries can be set up so that they automatically take into account the shadow receiver mask stored in the stencil buffer. If the receiver mask is stored in a separate texture (as for the fragment mask), a texture lookup and conditional fragment discard is necessary in the occlusion culling shader.

We also propose a few modifications to the CHC++ algorithm, which should be useful in most applications using CHC++. Our recent experiments indicated that for highly complex views the main performance hit of CHC++ comes from the idle time of the CPU, when the algorithm has to wait for query results of previously invisible nodes. The reason is that previously invisible nodes, which sometimes generate further queries leading to further wait time, can be delayed due to batching, whereas queries for previously visible nodes, whose result is only of interest in the next frame, are issued immediately whenever the algorithm is stalled waiting for a query result.

Therefore, it turns out to be more beneficial to use waiting time to issue queries for *previously invisible* nodes instead, starting these queries as early as possible. Queries for the previously visible nodes, on the other hand, are issued *after* the hierarchy traversal is finished, and their results are fetched just before the traversal in the *next* frame. Between these two stages we apply the shading pass with shadow map lookups, which constitutes a significant amount of work to avoid any stalls.

Another very simple CHC++ modification concerns the handling of objects that enter the view frustum. If such an object has been in the view frustum before, instead of setting it to invisible, it inherits its previous classification. This brings benefits especially for the camera view for frequent rotational movements.

6 Results and Discussion

6.1 Test Setup

We implemented our algorithms in OpenGL and C++ and evaluated them using a GeForce 480 GTX GPU and a single Intel Core i7 CPU 920 with 2.67 GHz. For the camera view render pass we used a 800×600 32-bit RGBA render target (to store color and depth) and a 16-bit RGB render target (to store the normals), respectively. For the light view render pass we used a 32-bit depth-stencil texture and an 8-bit RGB color buffer with varying sizes. The application uses a deferred shading pipeline for shadowing.

City scenes exhibit the scene characteristics which our algorithm is targeted at – large, open scenes with high cost for shadow mapping. Hence we tested our algorithm in three city environments: a model of Manhattan, the ancient town of Pompeii, and the town of Vienna (refer to Table 3 for detailed information). All scenes were populated with various scene objects: Manhattan was populated with 1,600 cars, Pompeii with 200 animated characters, and Vienna with several street objects and trees. For Manhattan (Pompeii) we created 30^2 (40^2) floor tiles in order to have sufficient granularity for the object-based receiver mask methods. We measured our timings with a number of predefined walkthroughs. Figure 4 shows the different receiver masks in a real scene.

Abbreviation	Method
VFC	view frustum culling
REF	CHC++ occlusion culling (main reference)
FOCUS	REF + visibility-aware focusing
CHC++-OPT	optimized CHC++
INCR	incremental shadow map updates
UNSHAD	main render pass without shadow mapping

Table 2: *Abbreviations used in the plots (also refer to Table 1 for the variants of receiver masks).*

We compare our methods mainly against a reference method (REF) that uses the original CHC++ algorithm for *both* the camera view and the light view. We also show simple view frustum culling (VFC) and a minor modification of the REF method which uses visibility-aware focusing (FOCUS). Note that apart from the FOCUS method, all other tested methods do not use visibility-aware focusing. The tested methods are summarized in Table 2.

6.2 Receiver Mask Performance and Shadow Map Resolution

Table 4 shows average frame times for the tested scenes and different shadow map resolutions. As can be seen, receiver masking is faster than REF and FOCUS for all scenes and parameter combinations. There is a tendency that our algorithm brings slightly higher speedup for uniform shadow mapping than for LiSPSM. This has

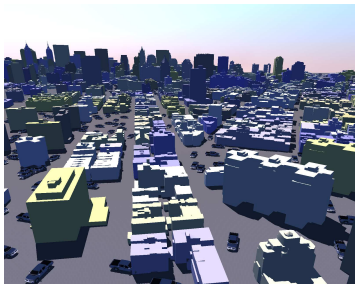
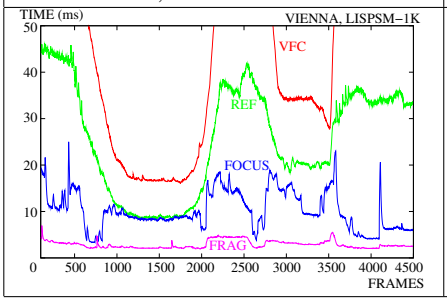
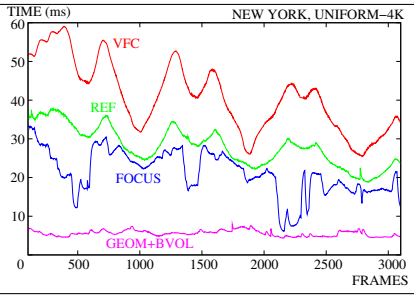
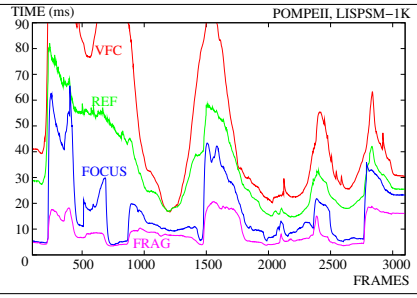
Vienna	Manhattan	Pompeii
		
12,685,693 vertices	104,960,856 vertices	63,141,390 vertices
22,464 objects	12,836 objects	11,097 objects
20,240 BVH nodes	6,471 BVH nodes	14,945 BVH nodes
		

Table 3: *Statistics for the scenes and characteristic walkthroughs.*

SM type	LISPSM			UNIFORM		
Shadow size	1K	2K	4K	1K	2K	4K
Scene	Vienna					
REF	21.6	22.3	22.4	28.7	28.7	29.2
FOCUS	9.1	9.3	9.3	10.8	11.0	11.4
GEOM+BVOL	4.9	4.9	**5.0**	4.9	5.0	**5.1**
FRAG	**2.9**	**3.5**	6.1	**2.9**	**3.4**	5.9
Scene	Manhattan					
REF	36.6	35.9	35.1	44.2	43.9	41.8
FOCUS	28.9	28.1	27.9	31.9	30.7	30.0
GEOM+BVOL	5.6	5.7	**6.6**	5.6	5.7	**6.5**
FRAG	**4.5**	**5.4**	9.0	**4.5**	**5.3**	8.6
Scene	Pompeii					
REF	34.8	34.8	39.2	40.5	40.2	44.8
FOCUS	17.4	17.4	20.6	18.2	18.3	21.2
GEOM+BVOL	11.2	11.2	13.4	11.4	11.3	13.0
FRAG	**9.7**	**10.0**	**13.2**	**9.8**	**10.2**	**12.5**

Table 4: *Average frame times for the tested scenes (in ms). The time for the best method for the given scene and shadow map resolution is marked in bold.*

two reasons: First, LiSPSM focuses much better on the visible parts of the view frustum. Second, LiSPSM trades the quality increase in the near field with quality loss in the far field regions (depending on the settings of the n parameter). Therefore it can happen that small objects in the background are simply not represented in the shadow map and will therefore be culled. Surprisingly, the frame times of the REF and FOCUS methods slightly decrease for increasing shadow map sizes in Manhattan. This unintuitive behavior might be connected with the heavy transform limitation of these algorithms for this particular scene configuration (i.e., due to many car instances).

Note that for shadow maps smaller or equal than $2K^2$, the fragment mask is consistently the fastest method in our tests. For larger shadow maps of $4K^2$ or more, it becomes highly scene dependent whether the benefit outweighs the cost of the additional texture lookups during mask creation and occlusion culling. In this case, the GEOM+BVOL method can be used, which has a smaller overhead caused only by the stencil buffer writes and the rasterization of the additional bounding boxes. On the other hand, we expect a higher speedup for the reverse shadow testing once it is possible to write the stencil buffer in the shader. There is already a specification available called *GL_ARB_STENCIL_EXPORT* [Khronos Group 2010], and we hope that it will be fully supported in hardware soon.

6.3 Walkthrough Timings

The plots in Table 3 show timings for walkthroughs selected scene and parameter combinations. For Manhattan, we applied uniform shadow mapping with $4K^2$ resolution and use the combined geometry and bounding volume receiver mask (GEOM+BVOL). Note that in this scene, the FOCUS method is often useless because skyscrapers are often visible in the distance, while most of the geometry in between can actually be culled by receiver masking. In Vienna and Pompeii, we applied LiSPSM shadow mapping with $1K^2$ resolution, and the fragment mask (FRAG). The plots show that receiver masking is the only method that provides stable frame times for the presented walkthroughs, which is an important property for a real-time rendering algorithm. The walkthroughs corresponding to these plots along with visualizations of the culled objects can be watched in the accompanying video for further analysis.

Table 5 shows the statistics and the timings for a scene taken from an actual game title (Left 4 Dead 2) using cascaded shadow mapping. This game scene was rendered using a different occlusion culling algorithm than CHC++, implemented in a different rendering engine. The receiver masking is however implemented exactly

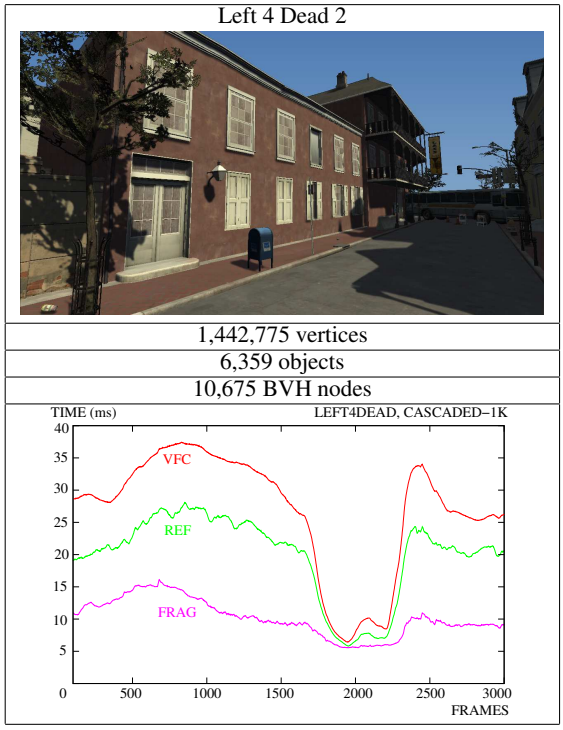


Table 5: *Statistics for the Left 4 Dead 2 scene and timings for a walkthrough using cascaded shadow mapping with four $1K^2$ shadow maps.*

as described in our paper. The fragment receiver mask and subsequent culling is applied separately for each of the four cascaded shadow maps using $1K^2$ resolution. Note that in this scene many objects are actually inside the houses and hence culled already by the REF method (depth-based culling without receiver masking). Even though this might imply a limited additional gain of our algorithm compared to REF, Table 5 shows that fragment masking still provides a significant speedup. Note that during frames 1,800-2,200, only a small fraction of objects is visible in the view frustum and almost all casters in the shadow frustum throw a visible shadow in the main view and thus there is very small potential for performance gain using any shadow caster culling method.

6.4 Geometry Analysis and CHC++ Optimization

Figure 5 analyses the actual amount of geometry rendered for different methods and contrasts that with frame times. In particular, here we include a comparison of the original CHC++ algorithm used in REF with our optimized version CHC++-OPT. The different receiver masking versions all use optimized CHC++. For all algorithms we used a $2K^2$ shadow map and LiSPSM.

Interestingly, the optimized CHC++ algorithm renders more vertices, but provides a constant speedup over the original version because of reduced CPU stalls when waiting for an occlusion query result. As can be observed from the plots, the speedup of the receiver masks corresponds closely to their level of tightness hence the number of rendered vertices.

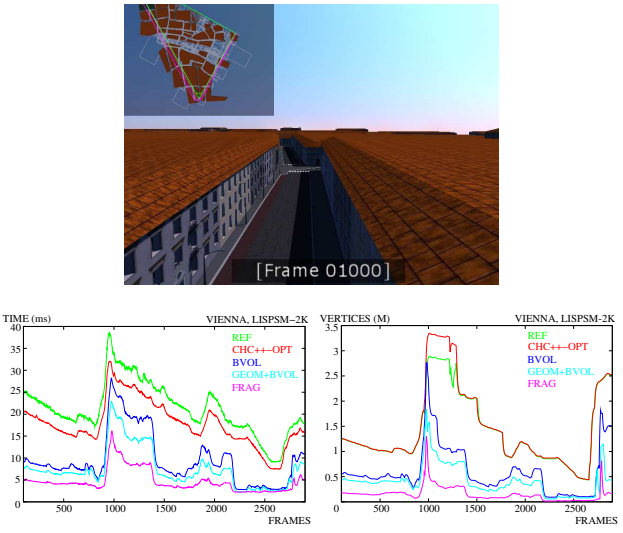


Figure 5: *Dependence of the frame time (bottom left) and number of vertices (bottom right) for a walkthrough of the Vienna scene and different shadow computation methods. Note that the walkthrouh also includes viewpoints located above the roofs. Such viewpoints see far over the city roofs (top). In this comparison the FRAG method achieves consistently the best performance followed by the the GEOM+BVOL method.*

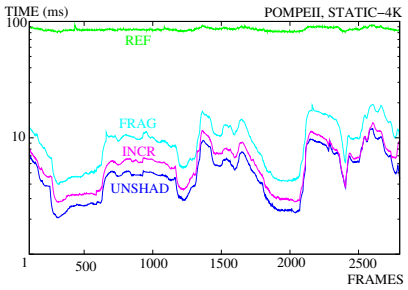


Figure 6: *Effect of incremental mask updates for a statically focused shadow map.*

6.5 Incremental Shadow Map Updates

In Figure 6, we use a static $4K^2$ shadow map for the whole Pompeii scene. Using this particular setup, REF performs over ten times worse than FRAG. Conversely, a static shadow map allows us to use the incremental shadow map updates optimization (INCR), which restricts shadow map updates to the parts of the shadow map that change, i.e., to the projections of the bounding boxes of dynamic objects. As can be seen, this technique can approach the performance of unshadowed scene rendering (UNSHAD). Note that we use a log-scale for this plot.

6.6 Dependence on Elevation Angle

In Figure 7, we show the dependence of the light source elevation angle on the (average) frame time for shadow mapping with resolution $1K^2$ in a Vienna walkthrough. For uniform shadow mapping, there is a noticeable correlation between the angle and the render times for REF and FOCUS. Interestingly, there seems to be a much weaker correlation for LiSPSM shadow mapping. The influence of the elevation angle on the performance of receiver masking is mini-

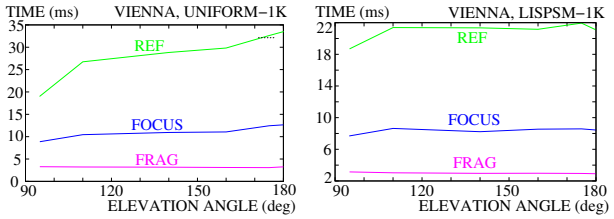


Figure 7: *Dependence of the (average) render time on the light elevation angle.*

mal in either case and our method delivers a consistent performance increase.

7 Conclusion

We proposed a method for efficient shadow caster culling for shadow mapping. In particular, our method aims at culling all shadow casters which do not contribute to visible shadow. We described several methods to create suitable receiver masks, and in particular one method which creates a practically pixel-exact mask through reverse shadow lookups.

Our method addresses the significant open problem of efficient construction of shadow maps especially in large outdoor scenes, which tend to appear more and more often in recent entertainment applications. We demonstrate speedups of 3x-10x for large city-like scenes and about 1.5x-2x for a scene taken from a recent game title. The method is also easy to implement and integrates nicely with common rendering pipelines that already have a depth-prepass.

Furthermore, we proposed a method for incremental shadow map updates in the case of statically focused shadow maps, as well as optimizations to the CHC++ occlusion culling algorithm, which are also applicable in other contexts.

In the future we want to focus on using the method in the context of different shadow mapping techniques and to study the behavior of the method in scenes with many lights.

Acknowledgements

We would like to thank Jiří Dušek for an early implementation of shadow map culling ideas; Jason Mitchell for the Left 4 Dead 2 model; Stephen Hill and Petri Häkkinen for feedback. This work has been supported by the Austrian Science Fund (FWF) contract no. P21130-N13; the Ministry of Education, Youth and Sports of the Czech Republic under research programs LC-06008 (Center for Computer Graphics) and MEB-060906 (Kontakt OE/CZ); and the Grant Agency of the Czech Republic under research program P202/11/1883.

References

BITTNER, J., WIMMER, M., PIRINGER, H., AND PURGATHOFER, W. 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3 (Sept.), 615–624. Proceedings EUROGRAPHICS 2004.

BRABEC, S., ANNEN, T., AND SEIDEL, H.-P. 2002. Practical shadow mapping. *Journal of Graphics Tools: JGT 7*, 4, 9–18.

COHEN-OR, D., CHRYSANTHOU, Y., SILVA, C., AND DURAND, F. 2003. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics. 9*, 3, 412–431.

DÉCORET, X. 2005. N-buffers for efficient depth map query. *Computer Graphics Forum 24*, 3.

EISEMANN, E., AND DÉCORET, X. 2006. Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM SIGGRAPH, 71–78.

ENGELHARDT, T., AND DACHSBACHER, C. 2009. Granular visibility queries on the gpu. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 161–167.

GOVINDARAJU, N. K., LLOYD, B., YOON, S.-E., SUD, A., AND MANOCHA, D. 2003. Interactive shadow generation in complex environments. *ACM Trans. Graph. 22*, 3, 501–510.

GUTHE, M., BALÁZS, A., AND KLEIN, R. 2006. Near optimal hierarchical culling: Performance driven use of hardware occlusion queries. In *Eurographics Symposium on Rendering 2006*, The Eurographics Association, T. Akenine-Möller and W. Heidrich, Eds.

KHRONOS GROUP, 2010. Opengl extension registry. http://www.opengl.org/registry, October.

LAURITZEN, A., SALVI, M., AND LEFOHN, A. 2010. Sample distribution shadow maps. In *Advances in Real-Time Rendering in 3D Graphics and Games II, SIGGRAPH 2010 Courses*.

LLOYD, B., WENDT, J., GOVINDARAJU, N., AND MANOCHA, D. 2004. Cc shadow volumes. In *Proceedings of EUROGRAPHICS Symposium on Rendering 2004*.

LLOYD, D. B., TUFT, D., YOON, S.-E., AND MANOCHA, D. 2006. Warping and partitioning for low error shadow maps. In *Proceedings of the Eurographics Workshop/Symposium on Rendering, EGSR*, Eurographics Association, Aire-la-Ville, Switzerland, T. Akenine-Möller and W. Heidrich, Eds., 215–226.

LLOYD, D. B., GOVINDARAJU, N. K., QUAMMEN, C., MOLNAR, S. E., AND MANOCHA, D. 2008. Logarithmic perspective shadow maps. *ACM Trans. Graph. 27*, 4, 1–32.

MATTAUSCH, O., BITTNER, J., AND WIMMER, M. 2008. Chc++: Coherent hierarchical culling revisited. *Computer Graphics Forum (Proceedings of Eurographics 2008) 27*, 3 (Apr.), 221–230.

SCHERZER, D., WIMMER, M., AND PURGATHOFER, W. 2010. A survey of real-time hard shadow mapping methods. In *State of the Art Reports Eurographics*,

STAMMINGER, M., AND DRETTAKIS, G. 2002. Perspective shadow maps. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 557–562.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings) 12*, 3 (Aug.), 270–274.

WIMMER, M., SCHERZER, D., AND PURGATHOFER, W. 2004. Light space perspective shadow maps. In *Rendering Techniques 2004 (Proceedings Eurographics Symposium on Rendering)*, Eurographics Association, A. Keller and H. W. Jensen, Eds., Eurographics, 143–151.

Appendix C

Temporally Coherent Adaptive Sampling for Imperfect Shadow Maps

Barák, T. - Bittner, J. - Havran, V.: Temporally Coherent Adaptive Sampling for Imperfect Shadow Maps. Computer Graphics Forum (EGSR '13), accepted for publication on May 19th 2013, to appear. **IF=1.636**.

Eurographics Symposium on Rendering 2013
Nicolas Holzschuch and Szymon Rusinkiewicz
(Guest Editors)

Volume 32 (2013), Number 4

Temporally Coherent Adaptive Sampling for Imperfect Shadow Maps

T. Barák, J. Bittner, V. Havran

Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

Abstract

We propose a new adaptive algorithm for determining virtual point lights (VPL) in the scope of real-time instant radiosity methods, which use a limited number of VPLs. The proposed method is based on Metropolis-Hastings sampling and exhibits better temporal coherence of VPLs, which is particularly important for real-time applications dealing with dynamic scenes. We evaluate the properties of the proposed method in the context of the algorithm based on imperfect shadow maps and compare it with the commonly used inverse transform method. The results indicate that the proposed technique can significantly reduce the temporal flickering artifacts even for scenes with complex materials and textures. Further, we propose a novel splatting scheme for imperfect shadow maps using hardware tessellation. This scheme significantly improves the rendering performance particularly for complex and deformable scenes. We thoroughly analyze the performance of the proposed techniques on test scenes with detailed materials, moving camera, and deforming geometry.

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

1. Introduction

Global illumination algorithms provide important visual cues that add on realistic appearance of rendered scenes. Ray tracing based global illumination methods can provide high quality images, but they are too slow for real-time applications such as games. Therefore we often resort to solutions that approximate global illumination up to different extents, for example they limit the number of indirect illumination bounces or use various visibility approximations. These methods often exploit rasterization, which is easy to parallelize and heavily optimized in the GPUs. Unlike ray tracing, rasterization methods do not rely on acceleration data structures and thus they easily handle dynamic scenes. On the other hand the rasterization methods are in their basic form restricted to computing visibility for coherent groups of rays enclosed by a viewing frustum.

One solution for computing global illumination, which has been designed to exploit rasterization hardware is the *instant radiosity* method [Kel97]. The main principle of the instant radiosity and the follow-up algorithms is the use of Virtual Point Lights (VPLs) that are traced from the primary light sources. The VPLs are then used for gathering of radiance at the shaded points, while visibility of VPLs is determined using shadow maps. The creation of shadow maps and subsequent visibility lookups form a bottleneck of the algorithm and even with recent hardware we cannot achieve real-time performance for moderately complex scenes. This bottleneck has been addressed by the Imperfect Shadow Maps (ISM) algorithm [RGK*08], which uses many low resolution shadow maps each computed using only a subset of the scene geometry. In contrast to the traditional shadow maps, all ISMs are stored in one texture and they are created in one rendering pass. Even though with ISMs the instant radiosity method achieves real-time performance, the number of VPLs that can be handled in real-time is still limited, which is an issue for complex scenes. To improve the quality of ISM based instant radiosity for complex scenes Ritschel et al. [REH*11] designed view adaptive method which optimizes the set of VPLs (and ISMs) for the given frame. This method estimates the importance of possible VPL positions with respect to the rendered image and then uses the inverse transform method to sample with respect to this importance.

In this paper we present a novel algorithm based on Metropolis-Hastings sampling that allows efficient selection of VPLs for complex scenes. Our main contributions are: (1) reducing temporal artifacts caused by adaptive VPL selection using parallel Independent Metropolis-Hastings sampling, (2) fast creation of ISMs using hardware tessellation.

The paper is further structured as follows. The next section presents briefly the work related to our approach. In Section 3 we describe the outline of the algorithm. In Section 4 we present a new method for temporally coherent sampling of VPLs in the context of the imperfect shadow maps algorithm. In Section 5 we describe a novel ISM creation algorithm using GPU tessellation. In Section 6 we give results from measurement for several scenes. Finally, in Section 7 we conclude the paper.

2. Related Work

Real-time global illumination. The global illumination algorithms for interactive applications were surveyed recently in the state-of-the-art report by Ritschel et al. [RDGK12]. Here we pay attention only to the most important papers directly related to our work. The proposed algorithm builds up on the idea of instant radiosity by Keller [Kel97] that introduced the concept of many virtual point lights (abbreviated to VPL). The created VPLs are used to gather the radiance from the indirect illumination in the same way as for direct illumination, while the visibility is resolved using shadow maps. Techniques which use many virtual lights for representing illumination are generally referred to as *many-light methods* and they were recently surveyed by Dachsbacher et al. [DKH*13].

Wald [WKB*02] adopted the instant radiosity method in the context of interactive ray tracing, where visibility between VPL and shaded points is computed by tracing rays on the CPU. Wald [WBS03] also developed an importance sampling approach for large scale and possibly highly occluded scenes. This method uses sparse sampling over a small set of image pixels to compute the approximation of importance of all VPLs and the most important VPLs are selected by thresholding according to the precomputed importance. Georgiev and Slusalek [GS10] propose a tunable importance sampling of VPLs, where a single parameter can be used to select a given portion of light sources. Similarly, Dammertz et al. [DKL10] in their progressive algorithm use stochastic culling of VPLs with low importance, based on the use of Halton random generator.

GPU rendering algorithms. To allow generation of VPLs on the GPU without the necessity of using ray tracing Dachsbacher and Stamminger [DS05] introduced the concept of reflective shadow maps. To generate VPL on scene surfaces the scene is rendered from primary light sources before the gathering from the generated VPLs takes place. Ritschel et al. [RGK*08] introduced imperfect shadow maps that lift restriction on the visibility computation in instant radiosity. The visibility for shadow maps is sub-sampled by coarse shadow maps of low resolution. This allows to use more VPLs for gathering, while the error of the proposed approximation is visually acceptable. In the follow-up paper Ritschel et al. [REH*11] use importance sampling based on the inverse transform method that first constructs the cumulative distribution (CDF) and then applies a binary search. The importance of all VPLs is computed for only small set of pixels (such as 0.1%) randomly selected for each light. Another alternative, which is possibly less prone to temporal artifacts and can better handle glossy scenes is to perform fast gathering of illumination [REG*09, MW11] for which visibility is evaluated with respect to gather points rather than VPLs.

Metropolis Sampling. Veach and Guibas [VG97] presented the Metropolis Light Transport (MLT) algorithm, which computes an unbiased estimate of the rendered image. The method uses mutations of light paths within the ingenious Markov chain Monte Carlo (MCMC) algorithm proposed by Metropolis et al. [MRR*53] and generalized by Hastings [HAS70]. Metropolis-Hastings sampling allows to sample from unknown possibly multidimensional distributions without the need to compute all probabilities of the underlying state space. Szirmay-Kalos et al. [SKDP99] studied the start-up bias of MLT for different ray distributions. Ashikhmin et al. [APSS01] analyzed the variance of MLT concluding the variance is inversely proportional to the number of samples taken. In context of interactive rendering and VPL-based method Segovia et al. [SIP07a] presented the use of Multiply-Try Metropolis-Hasting (MTMH) sampling of VPLs to get such sets of VPLs that are sufficiently spatially coherent and can be ray traced with fast packet based algorithms. Another method of Segovia et al. [SIP07b] exploits MTMH to provide such a set of VPLs in which each VPL contributes the same amount of power to the rendered image. Note that an open topic of research in statistics is the usage of quasi-random sequences within the MCMC methods [CDO11].

Temporally coherent rendering algorithms. A number of rendering algorithms have been specifically targeted at rendering animations and walkthroughs. In general these methods aim either to make the rendering more efficient by exploiting frame-to-frame coherence and/or to minimize the disturbing temporal artifacts usually perceived as flickering. The temporally coherent methods were surveyed by Tawara et al. [TMD*04] in the context of offline rendering algorithms and more recently by Scherzer et al. [SYM*12] in the context of real-time rendering. Directly related to our work, Laine et al. [LSK*07] presented a method that updates only a small part of all VPLs for subsequent frames. This method reduces temporal artifacts, but it also introduces a latency in handling fast illumination changes or fast camera movement for the case of adaptive VPL sampling.

3. Algorithm Overview

Our method builds on the idea of Instant Radiosity with Imperfect Shadow Maps (ISM) [RGK*08]. We use a deferred shading pipeline and thus in the first stage we create a layered framebuffer representation of the camera view, often referred to as *g-buffer* [AMHH08]. Then we create reflective shadow map (RSM) for each primary light source [DS05]. The RSM contains geometry and material information for surfaces visible from the corresponding primary light source. Similarly to the camera view the RSMs are stored as g-buffers.

The RSMs are then used to construct the set of VPLs, which then approximate the first bounce indirect illumination. We follow the adaptive sampling approach of Ritschel et al. [REH*11] who defined an importance metric for VPLs, which is used to create the importance map (IM). The IM represents the estimated contribution of the VPL at the given scene position to the rendered image. Unlike Ritschel et al. who used the inverse transform method, we use the Metropolis-Hastings sampling to sample according to the importance map, which in turn reduces the temporal artifacts caused by VPL adaptations.

When the VPLs are established, the ISMs are created by resampling the scene geometry to points and stochastically splatting these points to the ISMs. For this step we propose to use hardware tessellation, which provides significant speedup and which is also better suited to highly dynamic and deformable scenes. The ISM splatting phase is followed by the pull-push algorithm [MKC07], which fills holes caused by undersampling.

When ISMs are created, indirect illumination is gathered by summing up the contributions of the VPLs to shaded points, while using the ISMs for resolving approximate visibility of VPLs. Each pixel uses only a subset of the VPLs in order to speed up this phase of the algorithm [RGK*08]. Note, that we also use g-buffer splitting to shuffle the shaded pixels in order to improve the coherence of shadow lookups [SIMP06]. Finally, the indirect illumination is filtered using geometry aware filter and the result is summed up with the direct illumination to finalize the rendered image. The overview of the whole algorithm and its parts which will be discussed below in the paper is shown in Figure 1.

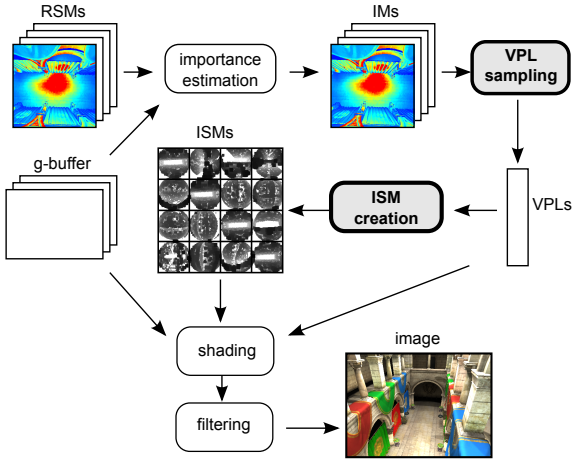


Figure 1: *Overview of the indirect illumination computation using view adaptive Imperfect Shadow Maps. The two steps of the algorithm which we address in the paper are highlighted in bold (VPL sampling, ISM creation).*

4. Temporally Coherent VPL Sampling

4.1. View Adaptive Imperfect Shadow Maps

The ISM algorithm can handle only a limited number of VPLs (and corresponding ISMs) in order to achieve real-time performance. Thus for scenes with more complex structure it is important to allocate these VPLs so that their contribution to the image is balanced.

Ritschel et al. [REH*11] proposed to treat every RSM pixel as a *potential VPL* (pVPL). Using a random subset of P scene points visible from the camera, the importance of each pVPL is estimated as follows:

$$f(\mathbf{v}) = \sum_{k=1}^{P} I(\mathbf{v}) H(\mathbf{x}_k, \omega_k, \mathbf{v}) \quad (1)$$

$$H(\mathbf{x}, \omega, \mathbf{v}) = f_r(\mathbf{x}, \omega, r(\mathbf{v}) \to \mathbf{x}) G(\mathbf{x}, r(\mathbf{v})) \quad (2)$$

where $\mathbf{v} = (l, \theta, \phi)$ is the *VPL description vector*, i.e. the vector of parameters describing a VPL (l is the index of the RSM, θ and ϕ are the coordinates of the VPL in the RSM), $I(\mathbf{v})$ is the VPL intensity, $\mathbf{x}_k$ is the world space position of image sample k, ω_k is the direction vector from the camera towards $\mathbf{x}_k$, $r(\mathbf{v})$ is the position of VPL $\mathbf{v}$, $f_r(\mathbf{x}, \omega, r(\mathbf{v}) \to \mathbf{x})$ is BRDF at point $\mathbf{x}$, $G(\mathbf{x}, r(\mathbf{v}))$ is the geometry factor between point $\mathbf{x}$ and VPL $\mathbf{v}$. Note that this function does not include visibility.

The importance computed for each pVPL forms an *importance map* (IM) of the same resolution as RSM (also called Bidirectional Reflective Shadow Map [REH*11]). This map after normalization corresponds to *probability density function*. A required number of VPLs is then established by sampling according to this density. The quality of the generated VPL pattern depends on the importance function definition and also on the actual VPL sampling algorithm. Ritschel et al. used the inverse transform method [REH*11] briefly discussed in the next section.

4.2. Inverse Transform Method

Using the inverse transform method (InvTM) the estimated contributions $f(\mathbf{v})$ in the importance map are normalized in order to create a discrete probability density $p(\mathbf{v}) \propto f(\mathbf{v})$.

The normalization this $p(\mathbf{v})$ requires the knowledge of sum of the values of $f(\mathbf{v})$ over all lights l and directions (θ, ϕ).

The cumulative distribution function CDF is calculated on the graphics hardware using multiple parallel prefix scans. For our case, the CDF is first constructed over all rows of all IMs, then over the last column of all IMs and finally, the values of all lights/IMs are cumulated. This calculation requires multiple kernel/shader launches and additional memory to store the CDF.

The successive sampling phase generates three random numbers (ξ_1, ξ_2, ξ_3) with the uniform distribution in the range $[0,1]^3$. The first random number ξ_1 is used to select the light l, ξ_2 and ξ_3 are used to select the row and column of the matching IM. This is done by using three consecutive binary searches in the CDF function (see Figure 2). Another possibility is to use two random numbers and the number used for the light selection renormalize back to the unit interval as for direct lighting [SW91]. We will show the differences between both approaches in Section 6.

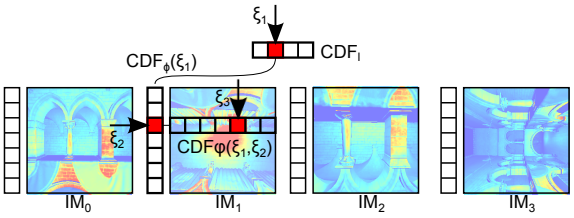


Figure 2: *Illustration of the InvTM for sampling with input of three random numbers. Each of the four IMs corresponds to one primary light source.*

For generating the random numbers (ξ_1, ξ_2, ξ_3) we use a Halton sequence evaluated directly on the GPU. Even if we use the same Halton sequence every frame, it turns out that the InvTM sampling exhibits insufficient temporal coherence. In particular even a local change in the $p(\mathbf{v})$ causes global changes in the corresponding $cdf(\mathbf{v})$s and potentially alters positions of a high number of VPLs (almost all VPLs slightly float on the surfaces). Although the VPL positions are not modified dramatically, the changes in scene geometry and materials cause sudden changes in the VPL image contribution. This is emphasized if the scene contains high geometric detail, detailed textures, or bump maps. As a result the adaptation of VPLs to the $p(\mathbf{v})$ introduces unwanted illumination flickering. In the next section we present a method which reduces the temporal artifacts, while being faster than the InvTM and simpler to implement.

4.3. Metropolis-Hastings Sampling

The Metropolis-Hastings (M-H) algorithm [HAS70] can be used to draw samples from an arbitrary probability distribution. The algorithm constructs a Markov chain and in each iteration it selects a sample that relies on the result of the previous iteration. The tentative samples are proposed using a *proposal distribution* and the algorithm decides either to accept the proposed sample or to keep the previously reported sample as the current one. After certain number of samples has been drawn the generated samples follow the target distribution. A know issue of the method is that a number of initial samples in so called *burn-in* phase will not follow the target distribution and thus they are often discarded. Alternatively we can use a compensation method which takes the accommodation phase into account. This issue is sometimes referred to as *start-up bias* [SKDP99].

A particular subclass of Metropolis-Hastings sampling are the Independent Metropolis-Hastings (IM-H) methods [Tie94]. In the IM-H methods the proposal distribution does not depend on the current state of the Markov chain. In our work we exploit the IM-H principle as this approach has several benefits for our application: (1) it is easier to control the correlation of the samples by using disjoint sets of proposals, (2) we can use a simple method to weight the samples and thus to reduce the start-up bias of the Monte Carlo estimator based on the VPLs determined by IM-H.

Traditionally the M-H algorithms are used to solve complex high dimensional integrals in cases when the InvTM method cannot be easily used. Contrary to the standard usage of M-H our primary motivation for using M-H algorithm is achieving better temporal coherence of the generated samples than the InvTM method does.

Our sampling strategy is based on parallel evaluation of N Markov chains, where N corresponds to the number of desired samples, that is the number of VPLs. From each chain we compute M samples and we only take the *last* generated sample as the representative of the target distribution. Within each chain we use a unique uniform proposal density generated using Halton sequence in the style of Wang and Hickernell [WH00]. For a chain with index j we take quasi-random numbers for seeds starting at $j + N * k$ $(0 < k < M)$, which produces disjoint proposals for each of the sequences. Together with using just one sample per chain this leads to disjoint set of generated M samples. Thus we avoid the common problem of Metropolis-Hastings sampling which is the potentially high correlation of samples, particularly the issue of one sample (VPL) being drawn multiple times.

All N chains can be evaluated in parallel using a simple algorithm, in which each chain performs M steps. The algorithm is illustrated in Figure 3 and outlined in Algorithm 1 (the use of weights $w(\mathbf{v}_j)$ will be explained in the next section). We refer to this algorithm as M-H-Our in Section 6.

4.3.1. Gathering VPL illumination

The indirect illumination at point $\mathbf{x}$ visible from the camera, which corresponds to a pixel in the image, is gathered from

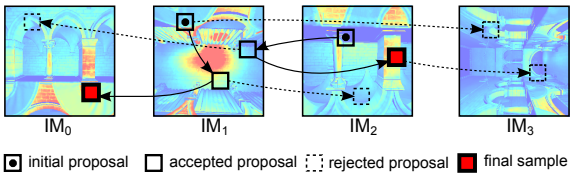


Figure 3: *Illustration of the Metropolis-Hastings VPL sampling. The figure shows two Markov chains of length $M = 5$ in four IMs. The first chain starts at IM_1 and two of the $M-1$ proposed moves were accepted. The result of this chain is a VPL sample at IM_0. The second chain starts at IM_2. Here again two of the proposed $M-1$ moves were accepted. The result of this chain is a VPL sample at IM_2.*

```
input  : j, f(), M
output: v_j, f(v_j), w(v_j)
begin
    s ⟵ 0;
    rand ⟵ initialize, use j as a seed;
    v_j ⟵ rand.nextVector3();
    for i ← 1 to M do
        v_new ⟵ rand.nextVector3();
        a ⟵ f(v_new)/f(v_j);
        s ⟵ s + f(v_new);
        if a ≥ rand.nextScalar() then
            v_j ⟵ v_new;
        end
    end
    w(v_j) ⟵ s/M;
    return v_j, f(v_j), w(v_j)
end
```

Algorithm 1: Pseudocode of the Independent Metropolis-Hastings algorithm for a single VPL that runs in parallel with different sequences of random numbers for each thread.

the VPLs using the following Monte Carlo estimator:

$$L(\mathbf{x},\omega) \sim \frac{1}{N}\sum_{j=1}^{N}\frac{1}{p(\mathbf{v}_j)}I(\mathbf{v}_j)H(\mathbf{x},\omega,\mathbf{v}_j)V(\mathbf{x},r(\mathbf{v}_j))$$

where N is the number of VPLs, $I(\mathbf{v}_j)$ is the intensity of VPL $\mathbf{v}_j$, $p(\mathbf{v}_j)$ is the probability of generating $\mathbf{v}_j$, $r(\mathbf{v}_j)$ is the position of $\mathbf{v}_j$, and $V(\mathbf{x},r(\mathbf{v}_j))$ is the binary visibility function between points $\mathbf{x}$ and $r(\mathbf{v}_j)$ and $H(\mathbf{x},\omega,\mathbf{v}_j)$ is defined according to Eq. 2.

In our case the target probability density of the IM-H algorithm is proportional to the importance function $f(\mathbf{v})$ and we use the following estimator:

$$L(\mathbf{x},\omega) \sim \frac{1}{N}\sum_{j=1}^{N}\frac{w(\mathbf{v}_j)}{f(\mathbf{v}_j)}I(\mathbf{v}_j)H(\mathbf{x},\omega,\mathbf{v}_j)V(\mathbf{x},r(\mathbf{v}_j)) \quad (3)$$

where $w(\mathbf{v}_j)$ is the weight of VPL $\mathbf{v}_j$ calculated in order to normalize the importance function $f(\mathbf{v})$ and also to reflect the below discussed issue connected with the burn-in phase of the M-H algorithm.

A common problem of M-H sampling is the burn-in phase in which the samples do not follow the target distribution. Using these samples in a Monte Carlo estimator might result in so called start-up bias of the estimator.

We use a simple strategy for reducing the start-up bias, which in the same time accounts for the normalization of the importance function $f(\mathbf{v})$. As the weight $w(\mathbf{v}_j)$ for a sample generated by chain j we use a local mean of the uniformly distributed proposals for that chain. For the case that $M = 1$ the algorithm degenerates to uniform sampling of the target distribution and the weights become $w(\mathbf{v}_j) = f(\mathbf{v}_j)$. For the case that $M \to \infty$ the weights correspond to the expected value of the importance function $f(\mathbf{v})$ and the distribution of samples $\mathbf{v}_j$ follows $f(\mathbf{v})$. Thus for very small M the generated samples $\mathbf{v}_j$ are closer to the uniform distribution, but the weights $w(\mathbf{v}_j)$ compensate for that and the estimator is normalized accordingly. Our experiments show that for $M > 4$ the samples already follow the importance function sufficiently well. It can also be shown that if $f(\mathbf{v}) \propto I(\mathbf{v}_j)H(\mathbf{x},\omega,\mathbf{v}_j)V(\mathbf{x},r(\mathbf{v}_j))$ the resulting estimator is unbiased for $M > 1$. This generally does not hold in our case, however we will show that in our target application the proposed method leads to images with error comparable to the InvTM method, while achieving lower temporal flickering.

The proposed weighting strategy resembles the method for eliminating startup bias proposed by Veach and Guibas for the Metropolis Light Transport algorithm [VG97]. Their technique draws the initial samples from a given stationary distribution and then weights the Markov chains by the image contribution function divided by the proposal distribution probability density at the initial sample points. In the results we will show that this weighting technique applied in our setting leads to higher variation of VPL intensities and thus exhibits higher temporal artifacts and higher error compared to a reference image. We refer to this variant as M-H-Single in Section 6.

Note that we use the same Halton sequence in each frame to achieve temporal coherence of the generated random numbers.

4.3.2. Performance

Unlike the InvTM which requires to store the computed CDFs the IM-H algorithm requires no additional data storage. The IM-H algorithm also avoids the CDF calculation and it does not require to calculate the total sum of importances for normalization. On the other hand the IM-H does not scale well with the length of the chains M, since $M-1$ samples are always discarded. Using the proposed method for reducing the start-up bias we however observed that very short sequences (e.g. $M > 4$) already provide high VPL adaptation to the underlying importance function (evaluation will be presented in Section 6).

5. Tessellation based ISM Rendering

5.1. Precomputed points

For the ISM creation step Ritschel et al. [RGK*08] use an approximate point representation of the scene. In the preprocessing stage, they create a point based representation of the scene by sampling the surface of the scene triangles. Later, in the ISM creation step, they use splatting of the point samples. Each point sample is splatted to a randomly chosen VPL/ISM.

This approach introduces additional complexity for moving or deforming geometry, as the point representation has to be updated with the moving or deforming triangles. This requires storing supplementary information with each point in the point-based representation, namely the index of the triangle and the barycentric coordinates of the point. To deal with deformable scenes a subset of point samples is regenerated in each frame. In the next section we propose a straightforward replacement of this approach which creates the whole point representation of the scene triangles on the fly.

5.2. Dynamic GPU tessellation

Instead of creating the point cloud off-line, we create the point representation of the scene during the ISM splatting phase by tessellation. For this task, we exploit the capabilities of modern graphics hardware that can tessellate the input geometry during the rendering stage (this functionality is called tessellation shaders in OpenGL 4.3 API).

Instead of generating finer triangle representation (which is the common use-case of the hardware tessellation), we use the *point mode tessellation* capability. During the point mode tessellation, the graphics hardware is instructed to change the primitive type from triangles to points after the tessellation stage. New point primitives are generated instead of vertices of the finer tessellated geometry. This feature allows us to transform the input triangle representation into points on the fly during the ISM splatting phase without storing the intermediate point representation.

Our ISM splatting algorithm works as follows (using the OpenGL pipeline terminology):

1. *CPU:* render the whole triangle representation of the scene with the tessellation turned on,
2. *vertex shader:* pass the vertex attributes to the next stage,
3. *tessellation control shader:* calculate the triangle surface area, set the number of points to be generated,
4. *tessellation evaluation shader:* generate a new position on the triangle, generate a random number generator seed for the next stage,
5. *geometry shader:* randomly choose VPL/ISM using the supplied seed, translate and project the splat into the selected shadow map, calculate the size of the splat,
6. *fragment shader:* store the depth into the ISM.

The process of rendering ISMs using hardware tessellation is illustrated in Figure 4.

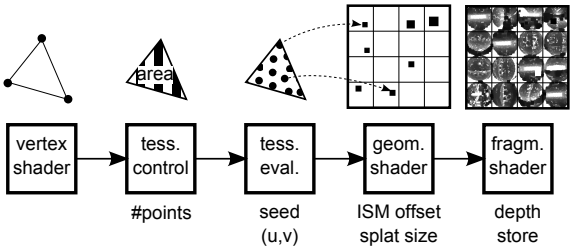


Figure 4: *Illustration of the tessellation based ISM splatting.*

The key observation behind our approach is that in common scenarios the number of required splats is much higher than the number of scene triangles. For higher quality indirect shadows, the number of required point splats becomes prohibitively high and the memory accesses can create a bottleneck in the rendering process. By omitting the need of storing and reading the large point cloud representation, we achieve a notable performance boost of this rendering stage.

Another benefit of using this approach is the additional flexibility it introduces for dynamic and deformable scenes. The proposed approach can be seamlessly connected to usual rasterization pipeline and can generate point splats directly from the deformed or moved meshes. There is also no need to maintain additional data structure containing the point splats.

6. Results

We test our implementation on a Core i7-3770, 16 GB RAM desktop computer, running 64 bit Linux operating system, equipped with GeForce GTX 470, 1280 MB GRAM graphics card with 310.32 NVIDIA driver. Rasterization uses OpenGL 4.3 API, all GPGPU tasks are implemented as GLSL compute shaders. The computation times are measured using high resolution GPU timers exported through the OpenGL timer query capability.

If it is not stated otherwise, we use the following settings in our measurements: 512×512 pixels output resolution, 2048×2048 pixels ISM resolution, 1024 VPLs are created, 128 VPLs are evaluated at each shaded pixel.

6.1. Run-time Performance

We tested the algorithm on four scenes shown in Figure 9. The timings of the individual algorithmic stages and the average frame times are shown in Table 1. The achieved frame rates are sufficient for the use in real-time applications. The time needed to create VPL by three different methods is shown in Table 2. For the InvTM method the measured times also include the CDF construction and are therefore higher

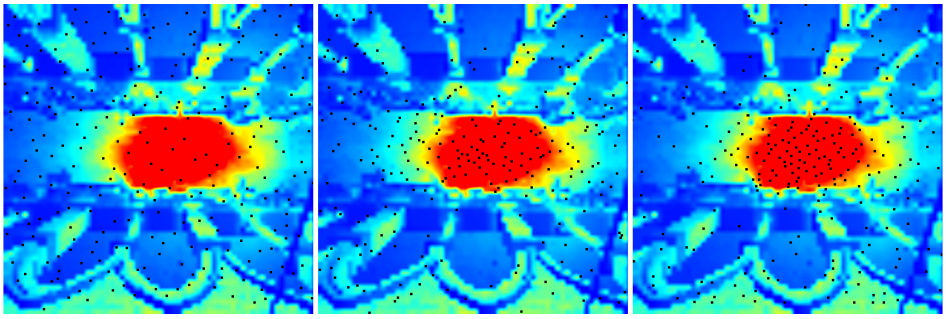


Figure 5: *The visualization of the distribution of selected VPLs in the Crytek-Sponza scene: (left) Uniform sampling, (center) Metropolis-Hastings, (right) InvTM. The intensity of indirect illumination mapped by pseudo-color from blue (low intensity) to red (high intensity). Note how both adaptive sampling methods distribute the samples according to the underlying importance function.*

than for the Uniform and M-H methods. Note that the times are almost equal for generating different number of VPLs. This follows from the fact that the GPU is capable of running more threads in parallel than the number of generated VPLs and thus in this step it is actually underutilized.

6.2. Evaluation of Image Quality and Flickering

To evaluate the frame-to-frame temporal coherence, we need a method to evaluate the amount of disturbing temporal flickering, which appears particularly during a walkthrough. Although several video-quality metrics were developed and published, e.g. [Win09, ČHM*12], they are not well suited for our evaluation for which we need a simple measure to compare the different VPL sampling techniques.

Our comparison metric is specifically designed for comparing renderings of static scenes with mostly diffuse surfaces. The radiosity solution should be temporally constant and independent of the camera movements. We therefore reproject pixels including their intensities from the previous frame to the current frame by taking their world-space coordinates from the previous frame. The *temporal image difference* (*TID*) is then given as a RMS value of differences among intensities of matching pixels for all consecutive pairs of frames in a walkthrough.

In Figure 5 we show the visualization of the VPL distribution for all three sampling algorithms being compared. We show the above described *TID* metric for the walkthrough and the RMSE against a reference walkthrough in Figure 7. The uniform sampling (VPLs are sampled uniformly from the RSMs using the same Halton sequence every frame) has the lowest temporal flickering, but the quality of the indirect illumination is lower, which leads to higher RMSE with respect to a reference. The InvTM sampling decreases RMSE, but it exhibits significant temporal flickering. Metropolis-Hastings sampling provides the good trade-off: the RMSE is slightly increased, while the temporal flickering is significantly reduced. Start-up bias depending on the M-H chain length for the tested sampling methods is shown in Figure 6.

Figure 8 shows the temporal image difference *TID* for varying number of VPLs and the number of VPLs used in gathering. The results are depicted for the sampling using either 3 (r3) or 2 (r2) random numbers from the Halton generator. In this context the temporal flickering is significantly lower when using 3 random numbers compared to the renormalization approach used for direct lighting [SW91].

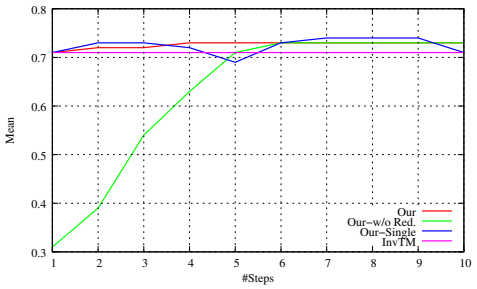


Figure 6: *Mean value of pixel intensities for various lengths of IM-H chains (i.e. number of steps) for the Crytek-Sponza scene. Note that in contrast to the IM-H without bias reduction reduction (Our − w/o − Red.), the proposed method (Our) successfully reduces the bias even for lower numbers of steps.*

6.3. Dynamic GPU Tessellation for ISMs

We measured the dynamic tessellation performance on a few scenes with various complexities. The rendered images for the tested scenes are shown in Figure 9. Table 3 shows the timings for the two evaluated ISM splatting methods in dependence on the number of splats N_{splats}. Our approach is 2 to 5 times faster than using the preprocessed points. Detailed comparison of splatting time for the two evaluated ISM splatting methods is shown in Figure 10.

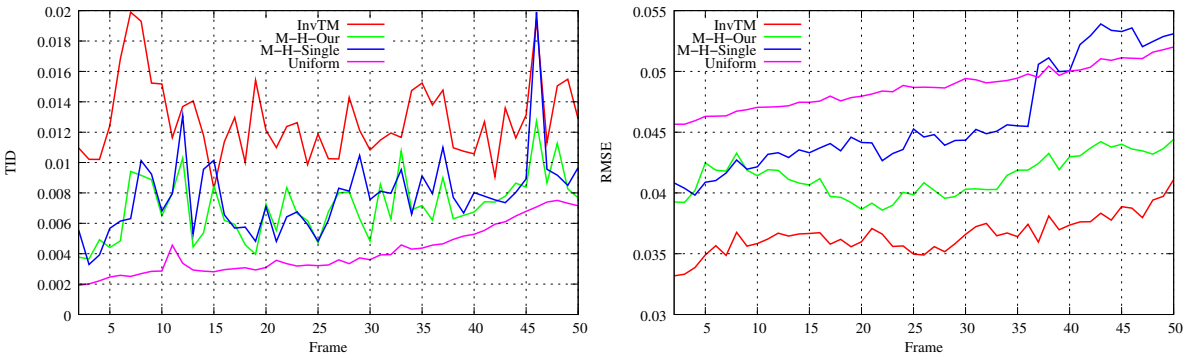


Figure 7: *Temporal image differences (left) and reference error (right) for the three compared methods (InvTM, M-H-Our, M-H-Single, Uniform). The plots show the first 50 frames of the Crytek-Sponza walkthrough using 1024 VPLs and 128 VPL evaluations per pixel.*

scene	N_L	T_{dir}	T_{ISM}	T_{indir}	T_{sum}
Conference R.	6	5.09	15.50	10.39	30.99
Sibenik Cath.	6	6.10	14.59	10.29	30.98
Crytek-Sponza	6	7.42	15.52	10.57	33.52
Armadillo	(*)	3.06	8.27	7.03	18.37

Table 1: *The timings of different algorithm phases and the total time in [ms] for four test scenes. T_{dir} is the direct illumination computed by shadow mapping, T_{ISM} covers VPL sampling and ISM splatting including the pull-push phase, T_{indir} is indirect illumination evaluation from VPLs. N_L the number of primary light sources, (*) for Armadillo the VPLs are generated on the sphere to simulate environment map lighting.*

	256 VPLs	512 VPLs	1024 VPLs
VPL Sampling algorithm	T_{sample} [ms]	T_{sample} [ms]	T_{sample} [ms]
Uniform	0.25	0.24	0.25
InvTM	0.62	0.62	0.63
M-H, 1 step	0.26	0.26	0.26
M-H, 5 steps	0.30	0.30	0.31
M-H, 10 steps	0.34	0.34	0.35
M-H, 20 steps	0.44	0.45	0.46

Table 2: *Timings for the generation of different count of VPLs. For InvTM the sampling includes the computation of CDF, for Metropolis-Hastings the different length of chain (i.e. the number of steps) is reported.*

6.4. Discussion and Limitations

Table 3 shows that our dynamic splatting approach can actually be slower if the number of generated splats N_{splats} is lower than the total number of scene triangles N_{tris}. In this case the preprocessing based approach transfers lower amount of data during the splatting phase compared to our approach which has to read and evaluate all the scene triangles. This setup is however a rare case that will unlikely appear in a real applications as the ISM algorithm requires a high number of point samples to create ISMs of sufficient quality. The method does not resolve the case, when the light sources are moving during the animation, for which other techniques such as spatio-temporal filtering are needed as discussed in [SYM*12].

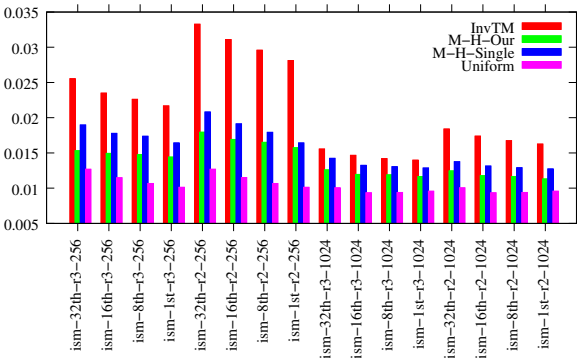
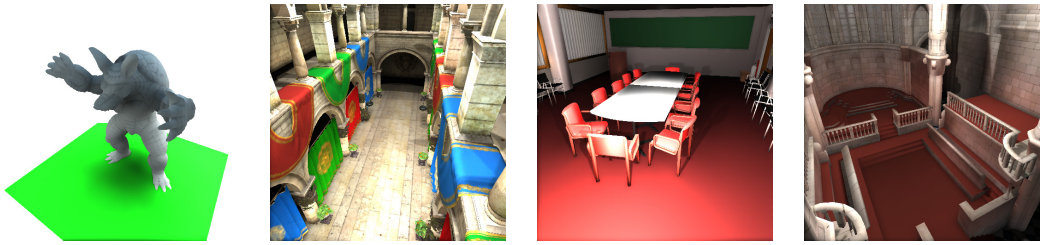


Figure 8: *The average temporal image differences for the walkthrough in Crytek-Sponza. Average sums of temporal image differences are reported for different settings referred to as ism-A-rB-C, where A denotes that every n-th VPL is used for rendering per pixel (1st for all generated VPL hence the highest computation time and quality), B denotes using either 2 or 3 dimensions of random generator and C denotes the number of VPLs generated.*

7. Conclusion

We proposed a novel algorithm for view adaptive computation of VPLs in the context of real-time instant radiosity with imperfect shadow maps. The method is based on paral-



scene	Armadillo	Crytek-Sponza	Conference	Sibenik
N_{tris}	345,946	262,267	331,179	75,284

Figure 9: *The rendered images and the count of triangles in the used test scenes.*

scene	Armadillo			Crytek-Sponza			Conference			Sibenik		
N_{SPLAT}	T_{PR} [ms]	T_{DT} [ms]	s [−]	T_{PR} [ms]	T_{DT} [ms]	s [−]	T_{PR} [ms]	T_{DT} [ms]	s [−]	T_{PR} [ms]	T_{DT} [ms]	s [−]
10,000	0.36	2.42	0.15	0.25	1.21	0.21	0.31	1.40	0.22	0.39	0.56	0.69
100,000	1.66	2.45	0.67	1.34	2.14	0.62	1.90	2.42	0.78	1.81	2.05	0.88
250,000	3.55	2.41	1.48	2.71	2.89	0.94	3.61	3.62	1.00	3.20	2.85	1.12
500,000	5.43	2.37	2.30	4.84	3.49	1.39	5.36	3.95	1.36	5.31	4.55	1.17
1,000,000	8.69	3.84	2.26	8.78	4.63	1.90	8.80	4.84	1.82	9.83	7.38	1.33
2,000,000	17.32	5.35	3.24	17.30	6.52	2.65	16.79	6.95	2.42	18.56	13.35	1.39
4,000,000	34.28	13.25	2.59	34.30	11.23	3.05	33.67	11.42	2.95	34.88	24.06	1.45
8,000,000	68.56	25.87	2.65	68.29	18.74	3.64	66.90	18.82	3.55	68.17	43.94	1.55
16,000,000	136.58	52.12	2.62	136.70	33.03	4.14	133.10	32.55	4.09	136.18	75.71	1.80
32,000,000	272.93	107.25	2.54	272.42	55.20	4.94	266.13	55.54	4.79	274.88	119.40	2.30

Table 3: *Dynamic GPU tessellation versus precomputed point-based sampling. N_{SPLAT} refers to the number of splats used to create the ISM. T_{PR} refers to the time needed to splat the precomputed point cloud [RGK*08, REH*11]. T_{DT} refers to the dynamic tessellation and splatting to ISM. Both T_{PR} and T_{DT} are reported in [ms] excluding the processing of the pull-push phase. Value $s = T_{PR}/T_{DT}$ refers to speedup of the dynamic GPU tessellation.*

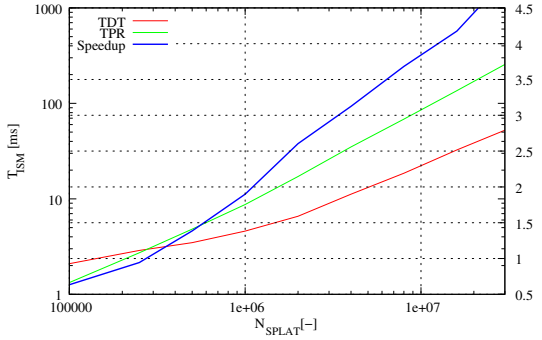


Figure 10: *ISM splatting time T_{ISM} as a function of the splat count N_{SPLAT} measured in the Crytek-Sponza scene. Our approach T_{DT} achieves better performance than the offline method T_{PR} [REH*11] for higher numbers of splats.*

lel Independent Metropolis-Hastings sampling, it is easy to implement, it is fast, and it requires no additional data storage. The results show that the method significantly reduces temporal flickering compared to the sampling based on the inverse transform method.

As a second contribution we proposed to accelerate the ISM creation by using hardware tessellation instead of precomputed point cloud. The results document significant speedup of the proposed technique and its better scalability towards large dynamic scenes.

In the future we would like to study other applications of the proposed temporally coherent Independent Metropolis-Hastings sampling. We also want to test different VPL importance metrics and evaluate them in a perceptual user experiment.

Acknowledgments

Our research was supported by the Czech Science Foundation under research programs P202/11/1883 (Argie) and P202/12/2413 (Opalis), and the Grant Agency of the Czech Technical University in Prague, grant No. SGS13/214/OHK3/3T/13, supported by Ministry of Education of the Czech Republic.

References

[AMHH08] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008. 3

[APSS01] ASHIKHMIN M., PREMOZE S., SHIRLEY P., SMITS B. E.: A Variance Analysis of the Metropolis Light Transport Algorithm. *Computers & Graphics 25*, 2 (2001), 287–294. 2

[CDO11] CHEN S., DICK J., OWEN A. B.: Consistency of markov chain quasi-monte carlo on continuous state spaces. *Annals of Statistics 2011 39*, 2 (2011), 673–701. 2

[ČHM*12] ČADÍK M., HERZOG R., MANTIUK R., MYSZKOWSKI K., SEIDEL H.-P.: New Measurements Reveal Weaknesses of Image Quality Metrics in Evaluating Graphics Artifacts. In *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* (2012), vol. 31, ACM, pp. 1–10. 7

[DKH*13] DACHSBACHER C., KŘIVÁNEK J., HAŠAN M., ARBREE A., WALTER B., NOVÁK J.: Scalable Realistic Rendering with Many-Light Methods. In *Eurographics 2013 state-of-the art report (STAR), to appear* (2013), pp. 1–16. 2

[DKL10] DAMMERTZ H., KELLER A., LENSCH H. P. A.: Progressive Point-Light-Based Global Illumination. *Computer Graphics Forum 29*, 8 (Dec 2010), 2504–2515. 2

[DS05] DACHSBACHER C., STAMMINGER M.: Reflective Shadow Maps. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2005), I3D '05, ACM, pp. 203–231. 2, 3

[GS10] GEORGIEV I., SLUSALLEK P.: Simple and Robust Iterative Importance Sampling of Virtual Point Lights. *Proceedings of Eurographics (short papers)*, 4 pages, (2010). 2

[HAS70] HASTINGS W. K.: Monte Carlo Sampling Methods using Markov Chains and their Applications. *Biometrika 57*, 1 (1970), 97–109. 2, 4

[HREB11] HOLLANDER 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 (Jun 2011), 1233–1240.

[Kel97] KELLER A.: Instant Radiosity. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), Whitted T., (Ed.), Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 49–56. 1, 2

[LSK*07] LAINE S., SARANSAARI H., KONTKANEN J., LEHTINEN J., AILA T.: Incremental Instant Radiosity for Real-Time Indirect Illumination. In *Proceedings of the 18th Eurographics conference on Rendering Techniques* (EGSR'07), 2007, pp. 277–286. 2

[MKC07] MARROQUIM R., KRAUS M., CAVALCANTI P. R.: Efficient point-based rendering using image reconstruction. In *Proceedings Symposium on Point-Based Graphics* (2007), pp. 101–108. 3

[MRR*53] METROPOLIS N., ROSENBLUTH A. W., ROSENBLUTH M. N., TELLER A. H., TELLER E.: Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics 21*, 6 (1953), 1087–1092. 2

[MW11] MALETZ D., WANG R.: Importance Point Projection for GPU-based Final Gathering. *Computer Graphics Forum 30*, 4 (2011), 1327–1336. 2

[RDGK12] RITSCHEL T., DACHSBACHER C., GROSCH T., KAUTZ J.: The State of the Art in Interactive Global Illumination. *Computer Graphics Forum 31*, 1 (Feb 2012), 160–188. 2

[REG*09] RITSCHEL T., ENGELHARDT T., GROSCH T., SEIDEL H.-P., KAUTZ J., DACHSBACHER C.: Micro-Rendering for Scalable, Parallel Final Gathering. *ACM Transactions on Graphics 28*, 5 (2009), 132:1-132:8. 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. 1, 2, 3, 9

[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. In *ACM Transactions on Graphics 27*, 5 (2008), 129:1–129:8. 1, 2, 3, 6, 9

[SIMP06] SEGOVIA B., IEHL J. C., MITANCHEY R., PÉROCHE B.: Bidirectional Instant Radiosity. In *Proceedings of the 17th Eurographics conference on Rendering Techniques* (2006), EGSR'06, pp. 389–397. 3

[SIP07a] SEGOVIA B., IEHL J.-C., PÉROCHE B.: Coherent Metropolis Light Transport with Multiple-Try Mutations. *Tech. Rep. RR-LIRIS-2007-015*, LIRIS UMR 5205 CNRS/INSA de Lyon, Apr. 2007, pp. 1–12. 2

[SW91] SHIRLEY P., WANG C.: Direct lighting calculation by monte carlo integration. In *Eurographics Workshop on Rendering* (1991), pp. 54–59. 4, 7

[SIP07b] SEGOVIA B., IEHL J.-C., PÉROCHE B.: Metropolis Instant Radiosity. *Computer Graphics Forum 26*, 3 (Sept. 2007), 425–434. 2

[SKDP99] SZIRMAY-KALOS L., DORNBACH P., PURGATHOFER W.: On the Start-up Bias Problem of Metropolis Sampling. In proceedings *Seventh International Conference in Central Europe on Computer Graphics and Visualization (WSCG'99)* (Feb. 1999), pp. 273–280. 2, 4

[SYM*12] SCHERZER D., YANG L., MATTAUSCH O., NEHAB D., SANDER P. V., WIMMER M., EISEMANN E.: Temporal Coherence Methods in Real-Time Rendering. *Computer Graphics Forum 31*, 8 (Dec. 2012), 2378–2408. 2, 8

[Tie94] TIERNEY L.: Markov Chains for Exploring Posterior Distributions. *Annals of Statistics 22*, 4 (1994), 1701–1762. 4

[TMD*04] TAWARA T., MYSZKOWSKI K., DMITRIEV K., HAVRAN V., DAMEZ C., SEIDEL H.-P.: Exploiting Temporal Coherence in Global Illumination (an invited paper). In *Spring Conference on Computer Graphics (SCCG 2004)*, 23–33. 2

[VG97] VEACH E., GUIBAS L. J.: Metropolis Light Transport. In *SIGGRAPH 97 Conference Proceedings*, ACM SIGGRAPH, 65–76. 2, 5

[WBS03] WALD I., BENTHIN C., SLUSALLEK P.: Interactive Global Illumination in Complex and Highly Occluded Environments. In *Proceedings of the 14th Eurographics workshop on Rendering* (2003), EGWR'03, pp. 74–81. 2

[WH00] WANG X., HICKERNELL F.: Randomized Halton Sequences. *Mathematical and Computer Modelling 32*, 7-8 (2000), 887–899. 4

[Win09] WINKLER S.: Video Quality Measurement Standards: Current Status and Trends. In *Proceedings of the 7th international conference on Information, communications and signal processing* (2009), ICICS'09, IEEE Press, pp. 848–852. 7

[WKB*02] WALD I., KOLLIG T., BENTHIN C., KELLER A., SLUSALLEK P.: Interactive Global Illumination using Fast Ray Tracing. In *Proceedings of the 13th Eurographics workshop on Rendering* (2002), EGWR'02, pp. 15–24. 2

Appendix D

RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures

Bittner, J. - Havran, V.: RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures.. In Proceedings of Spring Conference on Computer Graphics 2009.. Bratislava: Comenius University, 2009, p. 61-67. ISBN 978-80-223-2644-5.

RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures

Jiří Bittner*
Faculty of Electrical Engineering
Czech Technical University in Prague

Vlastimil Havran†
Faculty of Electrical Engineering
Czech Technical University in Prague

Abstract

Surface area heuristics is currently the most popular method for view independent construction of spatial hierarchies for ray tracing. We present a method which modifies the surface area heuristics by taking into account the actual distribution of rays in the scene. This is achieved by subsampling the rays to be cast and using these rays in order to estimate the probabilities of rays traversing through nodes of the constructed hierarchy. The main aim of our paper is to analyze the potential of taking the ray distribution into account. The results indicate that we can achieve a minor speedup of ray traversal compared to standard SAH. For large densely occluded scene we can also save the construction time and memory consumption of the hierarchy by not subdividing parts of the scene where no rays are traced.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—[Ray tracing]

1 Introduction

The fundamental task of ray tracing is to determine visibility along a given ray, i.e. to find the closest intersection of the ray with the scene. In order to find the intersection as fast as possible the search must be limited only to the proximity of the ray. This can be achieved by organizing the scene in a data structure such as regular grid, kD-tree, octree, or bounding volume hierarchy.

In particular two hierarchical data structures became very popular: kD-trees and bounding volume hierarchies. The major advantage of these hierarchies is their ability to efficiently adapt to irregular distribution of objects in the scene. The best known technique for this adaptation is the *Surface Area Heuristics* (SAH) originally proposed by Goldsmith and Salmon[Goldsmith and Salmon 1987] in the context of bounding volume hierarchies and then applied to kD-trees by MacDonald and Booth [MacDonald and Booth 1990]. The SAH uses a cost model which reflects the expected number of computed intersections as well as the expected number of traversal steps. This model is then used in a greedy optimization process which aims to minimize the cost of the constructed hierarchy. The core idea of the SAH is the evaluation of probability of ray visiting nodes of the hierarchy based on their surface areas. This evaluation assumes uniform distribution of rays and no occlusion.

In this paper we relief the assumption of uniform ray distribution made by the SAH. Instead we use explicit knowledge of ray distribution for a given image or a sequence of images. We aim at two main contributions: (1) We analyze the implications of the ray uniformity assumption made by the SAH. (2) We outline possible applications of the new heuristics which exploit spatial and temporal coherence of ray distributions.

*e-mail: bittner@fel.cvut.cz
†e-mail: havran@fel.cvut.cz

2 Motivation and Related Work

The most time consuming task of ray tracing is finding a closest intersection with the scene for every ray. This task is accelerated by data structures which limit the search for the intersection only to the proximity of the given ray. Data structures for ray tracing [Whitted 1979] have been investigated intensively over last three decades. There is a large body of literature on this topic which have been summarized in several thorough surveys [Slusallek et al. 2005; Chang 2004; Havran 2000; Glassner 1989].

The common principle of efficient data structures for ray tracing is to partition the scene into spatial cells and sorting the scene objects into these cells. When a ray is cast we identify the cells intersected by the ray and only objects referenced in these cells are tested for intersection with the ray.

There are two main categories of data structures used in ray tracing: *regular data structures* and *hierarchical data structures*. The most popular regular data structure are uniform grids [Fujimoto et al. 1986]. Ray tracing with uniform grids is efficient if the distribution of scene objects is also relatively uniform. This condition is often violated for practical scenes, which severely decreases the ray tracing performance.

The hierarchical data structures are commonly represented by a tree and possibly augmented by additional data structures. The major advantage of the hierarchies is their easy adaptation to the actual distribution of the objects.

The construction of the spatial data structures is analogous to spatial sorting [Samet 2006]. In particular, a top-down construction of a hierarchical spatial data structure is *Divide and Conquer* method, which is analogous to quicksort. In order to create an interior node of our data structure we have a set of objects residing in some spatial region. Our task is to distribute the objects into two or more child nodes. We use an algorithmic rule which prescribes how the objects should be distributed to these child nodes. This algorithmic rule corresponds to the pivot used in the traditional 1D quicksort. The final performance of the data structure heavily depends of the quality on the pivot selection.

If our pivot is formed by three planes perpendicular to main axes, then we create an octree [Glassner 1984]. If we allow that the spatial cells associated with children overlap, we create a bounding volume hierarchy. If we use a single splitting plane perpendicular to one coordinate axis we construct a kD-tree [MacDonald and Booth 1990]. In particular kD-trees, have proved very efficient in practice and therefore they are often the primary choice when implementing a ray tracing application.

The major factor influencing the quality of the constructed data structure is the positioning of the splitting plane in the spatial extent defined by an interior node. One possibility is to put the splitting plane in the middle (*spatial median*), another possibility is to balance the number of objects on the left and right side of the splitting plane (*object median*). Since our query is a ray passing through the scene until it hits an object, it is important to consider the geometric probability that a ray hits a spatial cell. This probability

is proportional to the surface area of a spatial cell assuming that distribution of rays in the scene is uniform [Solomon 1978]. The *surface area heuristics* (SAH) is a cost model which combines the geometric probabilities with the estimates of the traversal and intersection costs for a node being subdivided. According to the experiments [MacDonald and Booth 1990; Havran 2000] the SAH can lead to the performance of ray tracing to be by order(s) of magnitude higher than when we use spatial and object median approach.

We would like to note that the cost model based on SAH is useful also for other data structures that use spatial cells even if these cells overlap. This includes octrees [Whang et al. 1995], bounding volume hierarchies [Wald et al. 2007], and light-weight bounding volume hierarchies [Havran et al. 2006; Woop et al. 2006].

It was shown experimentally that the kD-trees constructed with SAH cost model are already very efficient in practice [Havran 2000]. However, Havran and Bittner [Havran and Bittner 1999] showed that it is possible to further improve the SAH by lifting the assumption that rays are distributed uniformly. They assumed that the rays are formed by a perspective, orthogonal, or spherical camera. In this paper we further lift the assumption that the set of rays is generated by a camera and propose a heuristics which allows for arbitrary ray distribution. Additionally, unlike previous techniques the proposed method considers occlusion in the evaluation of the cost model.

3 Overview

The method proposed in this paper extends the surface area heuristics by using explicit knowledge of ray distribution. The ray distribution is modeled using *representative ray set*. The representative ray set is generally a subset of rays cast in the current frame. Alternatively it is a subset of rays cast in the previous frame of an animation, assuming sufficient temporal coherence between these frames exists.

The representative ray set is used during the construction of the hierarchy by tracking all rays intersecting the current leaves of the subdivision. When the position of the splitting plane is being determined for a given leaf node, we use these rays to estimate the probabilities of rays intersecting the newly established parts of the leaf. As a result the new data structure is adapted to the actual ray distribution.

The paper is organized as follows: In Section 4 we review the SAH. In Section 5 we present the new method based on the ray distribution. Section 6 describes implementation details of the new method. Section 7 presents results and their discussion. Finally, Section 8 concludes the paper.

4 Surface Area Heuristics

Surface area heuristics is a greedy optimization method which aims to minimize the cost of the constructed spatial subdivision. In particular it optimizes the position of the splitting plane by minimizing the cost of traversal steps and ray-object intersections induced by the nodes created by the split.

The SAH uses the following cost function:

$$C = c_t + c_i(p_L^{SAH}|O_L| + p_R^{SAH}|O_R|), \quad (1)$$

where $p_{\{L|R\}}^{SAH}$ is the probability of rays intersecting the left and right children, respectively, $|O_{\{L|R\}}|$ is the number of objects in the left and right children, c_t is the traversal cost of interior node of the hierarchy (containing the splitting plane) and c_i is the cost of intersection computation. The crucial part of the SAH is the estimation of $p_{\{L|R\}}^{SAH}$, which is computed as follows:

$$p_{\{L|R\}}^{SAH} = \frac{S_{\{L|R\}}}{S}, \quad (2)$$

where $S_{\{L|R\}}$ are the surface areas of the left and right child, respectively, and S is the surface area of the subdivided node (see figure 1).

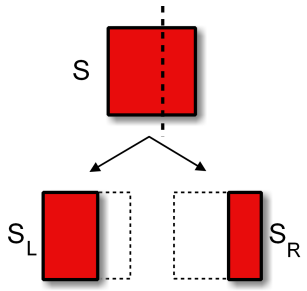


Figure 1: Subdivision of a node using a splitting plane induces two nodes with smaller bounding boxes. The SAH uses the ratio of the surface areas of the new boxes and the original box to estimate the probability of ray traversing to the corresponding nodes.

5 Ray Distribution Heuristics

The SAH uses surface areas to estimate the probability of ray traversing the left or right part of the tree. Our ray distribution heuristics evaluates these probabilities by taking into account the distribution of the actual rays which are traced.

5.1 Representing ray distribution

We represent the distribution of rays cast by the ray tracing algorithm using a *representative ray set* (RRS). The RRS can be significantly smaller than the actual set of rays to be cast as long as it describes the ray distribution reasonably well.

Let us denote the set of rays cast in frame i of an animation R^i and the set of rays used as RRS for frame i R^i_{RRS}. We consider the following possibilities for obtaining the RRS:

- $R^i_{RRS} \subseteq R^i$. The set of sample rays for frame i is a subset of rays cast in frame i. The ray distribution is described by subsampling the original ray set.
- $R^i_{RRS} \subseteq R^{i-1}$. The set of sample rays for frame i is a subset of rays cast in the previous frame $i-1$. The ray distribution is described by subsampling the ray set cast in the previous frame.
- $R^i_{RRS} \subseteq \bigcup_{\forall i} R^i$. The set of sample rays for frame i is the same for all frames and corresponds to a subset of the rays cast in all frames of the animation.

The first corresponds to undersampling in spatial domain assuming the rays cast for neighboring pixels are coherent. The second possibility also reuses the rays in temporal domain; rays cast in the previous frame are used as RRS for the current frame assuming the rays exhibit temporal coherence. The third possibility corresponds to a more aggressive undersampling in spatial and temporal domain, which however gives us a single ray set modeling the whole animation sequence.

5.2 Estimating traversal probabilities from RRS

For each splitting plane candidate we determine the rays which intersect the two fragments of the bounding box of the subdivided node (see Figure 2). Probability of a ray passing through the left and right fragments of the bounding box is then estimated as:

$$p^{RDH}_{\{L|R\}} = \frac{|R_{\{L|R\}}|}{|R|}, \tag{3}$$

where $|R_{\{L|R\}}|$ is the number of rays intersecting the left and right fragments respectively, and $|R|$ is the number of rays intersecting the whole box.

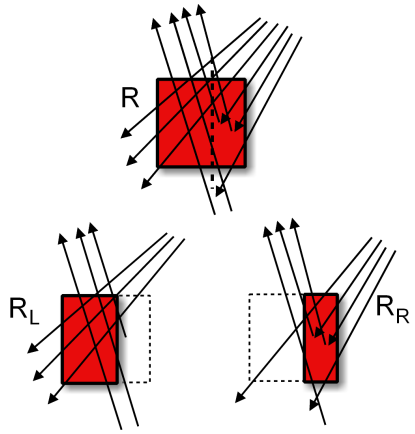


Figure 2: Illustration of the ray distribution heuristics.

From experiments we found out that using p^{RDH} directly in the cost function often leads to degenerated hierarchies. This happens due to the following two reasons: (1) the probability estimate is not perfectly accurate due to undersampling in RRS, (2) the greedy optimization based only on p^{RDH} has too strong focus towards RRS and thus is more prone to get stuck in a local minimum of the overall cost, i.e. the hierarchy degenerates. This happens for cases when majority of rays intersect the whole bounding box and hence the probabilities do not differ for wide range of a splitting plane position.

In order to solve these problems we blend the distribution of rays in *RRS* and uniform ray distribution using linear interpolation of of p^{RDH} and p^{SAH}:

$$p^{B}_{\{L|R\}} = w_r p^{RDH}_{\{L|R\}} + (1 - w_r) p^{SAH}_{\{L|R\}}, \tag{4}$$

where w_r is the weight of the ray distribution probability estimate. This weight aims to reflect the expectation that the estimate is correct for the given representative ray set. We used the following formula for computing w_r:

$$w_r = \alpha \cdot (1 - \frac{1}{1 + \beta * |R|}), \tag{5}$$

where α and β are user specified constants and $|R|$ is the number of rays intersecting the node to be split. This function gives more weight to *RDH* if $|R|$ is large and less weight if $|R|$ is small and we assume that these ray do not carry enough statistical information. For our experiments we used $\alpha = 0.9$ and $\beta = 0.1$. The weighting function of parameter $|R|$ is depicted in Figure 3.

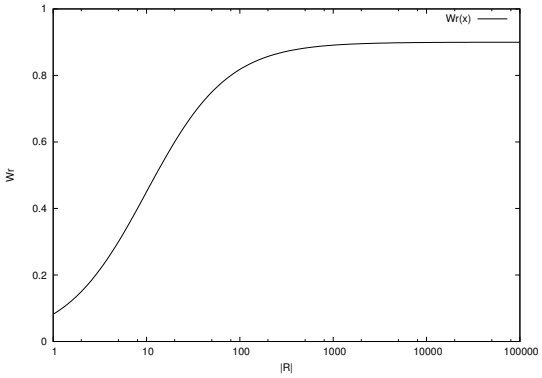


Figure 3: The weighting function w_r for $\alpha = 0.9$ and $\beta = 0.1$.

5.3 The cost function

The cost of a splitting plane candidate is computed similarly as in the SAH:

$$C = c_t + c_i(p^B_L|O_L| + p^B_R|O_R|), \tag{6}$$

where $p^B_{\{L|R\}}$ is the blended probability estimate of rays traversing the left and right child, respectively, c_t is the traversal cost and c_i is the intersection cost and $|O_{L|R}|$ is the number of objects intersecting the left and right children, respectively.

6 Implementation

This section describes several implementation details connected with the RDH.

6.1 Computing representative ray set

For testing the influence of the new heuristics we use the following strategy to compute the RRS. We first build a kD-tree using surface area heuristics. Then we subsample the actual set of rays to be cast using a regular pattern in the synthesized image. For example, using a 2x2 pattern we obtain a RRS with size of 1/4th of all rays to be cast. This subsampling method also handles secondary rays or shadow rays if these are traced. Note, that when using 1x1 pattern the RRS is equal to the actual ray set.

6.2 Efficient tracking of rays intersecting the box

For establishing the splitting plane we have to determine the set of rays intersecting left and right part of the corresponding bounding box.

We implemented a technique similar to the method for fast construction of kD-trees proposed by Wald and Havran [Wald and Havran 2006]. We use several data structures in order to make the algorithm efficient: First, we use a *ray segment*, which stores the entry and exit signed distances of a ray in the currently processed box, the reference to the whole ray, and the stack of the signed distances. Second, we use a *set of ray boundaries*, where each boundary contains a reference to the corresponding ray segment and a flag, whether the boundary is left, right, or lying on a splitting plane. Third, we use a *boundary index stack* to store which ray boundaries from the whole set of boundaries belong to the left, to the right, and to both of them. The boundary index stack hence stores 4 integers (leftmin, leftmax, rightmin, rightmax), which represent 3 intervals of ray boundaries when a splitting plane is placed: $\langle leftmin, leftmax \rangle$ for rays that belong only to the left box, $\langle leftmin, rightmax \rangle$ for rays that straddle the splitting plane, and $\langle rightmin, rightmax \rangle$ for rays that belong only to the right box. The data structures used in the algorithm are outlined in Figure 4.

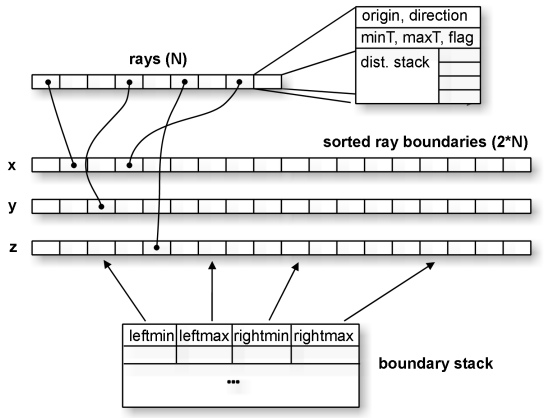


Figure 4: Data structures used for tracking the rays intersecting the currently subdivided node.

Before building the kD-tree with RDH we sort all three sets of initial ray boundaries, one set of ray boundaries for each axis. We assume that a kD-tree is constructed in a depth-first-search order, first constructing the left subtree of the current interior node. When a splitting plane position is decided and a left subtree should be constructed, we check the boundaries in the corresponding axis and mark all ray segments that have to be shortened (are cut by the splitting plane). The original ray boundaries are stored in the stack of signed distances associated with each ray segment data structure. For the marked rays we recompute the boundaries in a corresponding axis. All ray boundaries are then resorted by quicksort before we start to evaluate the RDH.

The number of rays intersecting the left and right fragments induced by the splitting plane is computed by counting the number of left and right boundaries on the left side and the right side of a splitting plane. The cost function for subsequent splitting plane candidates is evaluated incrementally by using the sweeping plane paradigm similarly to [Wald and Havran 2006].

When returning to an interior node after a left subtree has been constructed, we have to change the meaning of ray boundaries that lie on the splitting plane. These originally right boundaries for the left subtree now become the left boundaries for the right subtree to be constructed. The correct restore of boundary values is achieved by storing the depth of the node (together with the signed distance to the splitting plane) where the splitting plane has changed the boundary for each ray. The intervals of ray boundaries are also restored during return from both left child and right child.

In our experience this algorithm is relatively efficient as we store the information about original end of the ray segment only for the ray segments intersecting a splitting plane. The worst case space complexity for this algorithm is $O(N \cdot D)$, where the D is the maximum depth of a kD-tree and N is the number of rays. In practice the memory requirements are much lower since the case that all rays are always intersected by a splitting plane for all interior nodes from the root node to a leaf is very rare.

Another option how to evaluate ray distribution heuristics would be to use sampling of rays for preselected splitting plane positions similar to the approach of evaluating cost function with SAH [Hunt et al. 2006]. In that case we would need to test simultaneously both the rays from RRS and the objects. The method would have smaller accuracy as it limits the number of splitting planes tested, which was not desired for the analysis of our new heuristics.

7 Results

We implemented the ray distribution heuristics inside an optimized ray tracer. For the results we used a number of test scenes of various complexity and visibility characteristics. We tested ray casting with primary rays only and ray tracing including shadow and secondary rays. All images were computed in resolution of 513×513 pixels. The measurements were performed on a PC with 2.4GHz P4 CPU with 512KB L2 cache, 2GB RAM, and Linux OS.

7.1 Measurements overview

In our measurements we used 28 scenes in which we specified 52 different view points. For each measurement we evaluated the memory cost of the constructed kD-tree (M_{KD}), the number of intersections per ray (N_I), the average number of traversed nodes per ray (N_T), the construction time of the kD-tree (T_C), and the actual time of ray casting/ray tracing (T_R).

The tests were divided into three categories depending on the scene types and the rendering algorithm: (1) ray casting of low occlusion scenes, (2) ray tracing of low occlusion scenes, (3) ray casting of high occlusion scenes. Note that we did not test ray tracing for high occlusion scenes as we did not have meaningful definition of light sources for these scenes. In total we computed 78 measurements, where each measurement corresponds to a different image (either different scene, view point, or rendering algorithm). Several measurements are shown in Table 1. The scenes and their views correspond to snapshots shown in Figure 5. A visualization of the render cost functions for these views is shown in Figure 6.

In further tests we do not present the absolute values of the measurements, but use averages of the ratios of RDH and SAH methods for each measured parameter over all tests from the appropriate categories.

scene	method	M_{KD} [MB]	N_I [-]	N_T [-]	T_C [s]	T_R [s]
Ray casting, low occlusion scenes						
balls5	SAH	2.76	7.02	39.09	0.555	0.450
	RDH	4.04	5.60	38.67	1.677	0.444
Chevy	SAH	0.34	13.67	5.68	0.120	0.252
	RDH	2.14	3.78	8.66	0.489	0.196
jacks5	SAH	9.08	12.71	39.42	0.619	0.521
	RDH	8.10	11.17	33.65	1.569	0.488
rings17	SAH	11.14	13.87	74.56	1.006	1.027
	RDH	9.82	11.73	81.67	3.139	1.063
teapot40	SAH	8.91	4.19	29.26	0.859	0.346
	RDH	8.20	3.16	27.56	1.594	0.344
Ray tracing, low occlusion scenes						
balls5	SAH	2.76	16.27	30.91	0.465	2.974
	RDH	4.94	12.88	34.00	2.470	3.176
Chevy	SAH	0.34	37.48	8.54	0.131	2.296
	RDH	2.45	10.30	15.67	1.009	1.769
jacks5	SAH	9.08	21.95	57.33	0.619	3.620
	RDH	9.47	21.12	58.34	3.103	3.792
rings17	SAH	11.14	23.48	47.41	1.036	7.615
	RDH	11.62	21.59	56.93	5.693	8.626
teapot40	SAH	8.91	15.93	38.28	0.917	2.022
	RDH	9.36	11.82	39.36	2.579	2.002
Ray casting, high occlusion scenes						
arena v1	SAH	135.20	25.62	81.73	13.568	1.020
	RDH	126.68	19.11	61.80	17.660	0.825
arena v2	SAH	135.20	9.38	63.98	22.174	0.885
	RDH	109.61	6.51	56.27	21.267	0.494
Vienna v1	SAH	77.52	11.64	39.79	7.328	0.489
	RDH	77.98	8.64	33.95	10.691	0.419
Vienna v2	SAH	77.52	6.92	42.51	7.339	0.377
	RDH	63.43	5.14	32.12	9.773	0.315
Vienna v3	SAH	77.52	15.91	68.84	7.441	0.653
	RDH	72.13	8.44	52.81	10.026	0.489

Table 1: Experimental results for selected scenes and view points. The selection shows 5 representative scenes (views) for ray casting of low occlusion scenes, ray tracing of low occlusion scenes and ray casting of high occlusion scenes. M_{KD} is the memory cost of the constructed kD-tree, N_I is the number of intersections per ray, N_T is the average number of traversed nodes per ray, T_C is the construction time of the kD-tree and T_R is the time of ray casting/ray tracing.

7.2 Dependence on the size of RRS

In this test we analyzed the behavior of RDH in dependence on the size of RRS, i.e. number of rays used for RDH. We used different sampling patterns to obtain RRS and computed the ratios of measured values for RDH and SAH. The results are summarized in Table 2.

The results indicate that the RDH is surprisingly stable with lower number of samples. We expect that this behavior is also influenced by blending the ray distribution with the uniform ray distribution as described in Section 5.2. The information from rays is used in the higher levels of the tree whereas at the deeper levels the SAH is used with significant weight. The best results where achieved by using the actual rays to be cast as RRS (1x1 sampling pattern), but the results were quite stable even for lower sampling densities. In the remaining tests presented in the paper we used 4x4 subsampling as it exhibits reasonable performance rendering, while having a relatively low overhead on the construction time of the hierarchy.

RRS pattern	$\frac{M_{KD}^{RDH}}{M_{KD}^{SAH}}$	$\frac{N_I^{RDH}}{N_I^{SAH}}$	$\frac{N_T^{RDH}}{N_T^{SAH}}$	$\frac{T_C^{RDH}}{T_C^{SAH}}$	$\frac{T_R^{RDH}}{T_R^{SAH}}$
1×1	1.19	0.72	0.91	66.90	0.85
2×2	1.16	0.74	0.92	22.20	0.93
3×3	1.18	0.73	0.92	6.38	0.89
4×4	1.14	0.76	0.92	4.27	0.93
5×5	1.15	0.76	0.93	3.50	0.87
7×7	1.14	0.75	0.93	2.42	0.94

Table 2: Dependence of the method on the size of the RRS. The table shows a comparison of RDH and SAH for different sampling patterns of primary rays.

7.3 Behavior for different scenarios

In order to analyze the behavior of the method for different scenarios we grouped the tests according to scene types into two classes: ordinary scenes with low occlusion and complex architectural scenes with dense occlusion. In the first class we used ten common benchmark scenes for ray tracing generated (Standard Procedural Database [Haines 1987]) plus several other scenes. In the second class we tested two architectural scenes: city model and model of a sports stadium with furnished interiors. The results are summarized in Table 3.

scene	$\frac{M_{KD}^{RDH}}{M_{KD}^{SAH}}$	$\frac{N_I^{RDH}}{N_I^{SAH}}$	$\frac{N_T^{RDH}}{N_T^{SAH}}$	$\frac{T_C^{RDH}}{T_C^{SAH}}$	$\frac{T_R^{RDH}}{T_R^{SAH}}$
low occlusion	1.03	0.78	0.83	2.62	0.85
high occlusion	0.87	0.65	0.84	1.31	0.82

Table 3: Comparison of SAH and RDH on ray casting scenes with low occlusion and high occlusion.

7.4 Ray Casting vs. Ray Tracing

We compared the behavior of the method for ray casting (only primary rays) and recursive ray tracing. For the recursive ray tracing we casted shadow rays and secondary rays up to depth 3. The results of this test are summarized in Table 4.

method	$\frac{M_{KD}^{RDH}}{M_{KD}^{SAH}}$	$\frac{N_I^{RDH}}{N_I^{SAH}}$	$\frac{N_T^{RDH}}{N_T^{SAH}}$	$\frac{T_C^{RDH}}{T_C^{SAH}}$	$\frac{T_R^{RDH}}{T_R^{SAH}}$
ray casting	1.03	0.78	0.83	2.62	0.85
ray tracing	1.09	1.03	0.98	3.80	1.05

Table 4: Comparison of SAH and RDH when using ray casting and ray tracing algorithms.

The results indicate that on average we obtained 15% speedup for ray casting, but the performance of recursive ray tracing was actually decreased by 5% compared to the SAH.

In ray tracing the rays are more spread over the scene and thus they are closer to uniform distribution assumed by SAH. This explains why the RDH does not perform as good for this case as for ray casting. However, it is surprising that RDH doesn't provide greater speedup compared to SAH taking into account the fact that we consider the actual ray distribution including the occlusion. For ray casting the speedup might have some minor importance, however for ray tracing the current form of the method does not lead to practical results.

7.5 Ray based termination

In the last test we evaluated the influence of the ray based termination criterion on the constructed hierarchies. We used a threshold of one ray in order to decide whether to continue with the subdivision; i.e. if there is no ray intersecting the given box the subdivision is terminated for this node. The comparison of the RDH method and the RDH-R method which uses the ray based termination is shown in Table 5.

scenes	$\frac{M_{KD}^{RDH-R}}{M_{KD}^{RDH}}$	$\frac{N_I^{RDH-R}}{N_I^{RDH}}$	$\frac{N_T^{RDH-R}}{N_T^{RDH}}$	$\frac{T_C^{RDH-R}}{T_C^{RDH}}$	$\frac{T_R^{RDH-R}}{T_R^{RDH}}$
low occlusion	0.76	1.03	1.00	0.95	1.00
high occlusion	0.43	2.87	1.06	0.64	1.98

Table 5: Comparison of RDH-R with additional ray based termination criterion and the RDH with ordinary termination criteria.

We can observe that for low occlusion scenes the ray based termination saved about 25% of storage on average, while not increasing the average rendering time. For high occlusion scenes the memory savings where even more significant (57%), however the rendering time almost doubled on average. We can observe that we significantly increased the average number of intersections. This indicates that there have been parts of the scenes which were not covered by the sample rays, but were penetrated by significant number of actual rays that have been cast.

8 Conclusion and Future Work

We presented a method for changing the surface area heuristics by using knowledge about the actual distribution of rays in the scene. The kd-trees constructed using the proposed ray distribution heuristics achieve on average a slightly better performance than the traditional surface area heuristics for ray casting. The results indicate that for ray tracing the method actually does not provide any measurable benefit.

An important result of the paper is the observation that taking into account the actual ray distribution does not really help in the construction of the ray tracing hierarchy assuming the traditional greedy top-down construction algorithm. Thus, the actual practical application of the method to interactive ray tracing or ray tracing of animated sequences is a subject of future work. In the future we also want to study the possibility of using global optimization techniques together with the new heuristics.

Acknowledgements

This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research program MSM 6840770014 and LC-06008 (Center for Computer Graphics), and the Aktion Kontakt OE/CZ grant no. 2009/6. The Arena scene is a courtesy of Digital Media Production a.s.

References

CHANG, A. Y.-H. 2004. *Theoretical and Experimental Aspects of Ray Shooting*. PhD thesis, Politechnic University, USA.

FUJIMOTO, A., TANAKA, T., AND IWATA, K. 1986. ARTS: Accelerated Ray Tracing System. *IEEE Computer Graphics and Applications 6*, 4, 16–26.

GLASSNER, A. S. 1984. Space Subdivision For Fast Ray Tracing. *IEEE Computer Graphics and Applications 4*, 10 (Oct.), 15–22.

GLASSNER, A. 1989. *An Introduction to Ray Tracing*. Morgan Kaufmann.

GOLDSMITH, J., AND SALMON, J. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (May), 14–20.

HAINES, E. A. 1987. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications 7*, 11 (Nov.), 3–5. Available from `http://www.acm.org/pubs/tog/resources/SPD/overview.html`.

HAVRAN, V., AND BITTNER, J. 1999. Rectilinear BSP Trees for Preferred Ray Sets. In *Proceedings of SCCG'99 (Spring Conference on Computer Graphics)*, 171–179.

HAVRAN, V., HERZOG, R., AND SEIDEL, H.-P. 2006. On the Fast Construction of Spatial Data Structures for Ray Tracing. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, 71–80.

HAVRAN, V. 2000. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague.

HUNT, W., MARK, W. R., AND STOLL, G. 2006. Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *2006 IEEE Symposium on Interactive Ray Tracing*, IEEE.

MACDONALD, J. D., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 6, 153–65.

SAMET, H. 2006. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.

SLUSALLEK, P., SHIRLEY, P., WALD, I., STOLL, G., AND MARK, B. 2005. SIGGRAPH 2005 Course on Interactive Ray Tracing #38.

SOLOMON, H. 1978. *Geometric Probability*. J.W. Arrowsmith Ltd.

WALD, I., AND HAVRAN, V. 2006. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, 61–69.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1.

WHANG, K. Y., SONG, J. W., CHANG, J. W., KIM, J. Y., CHO, W. S., PARK, C. M., AND SONG, I. Y. 1995. Octree-R: an adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics 1*, 4 (Dec.), 343–349. ISSN 1077-2626.

WHITTED, T. 1979. An improved illumination model for shaded display. *Computer Graphics 13*, 2 (Aug.), 14–14.

WOOP, S., MARMITT, G., AND SLUSALLEK, P. 2006. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*.

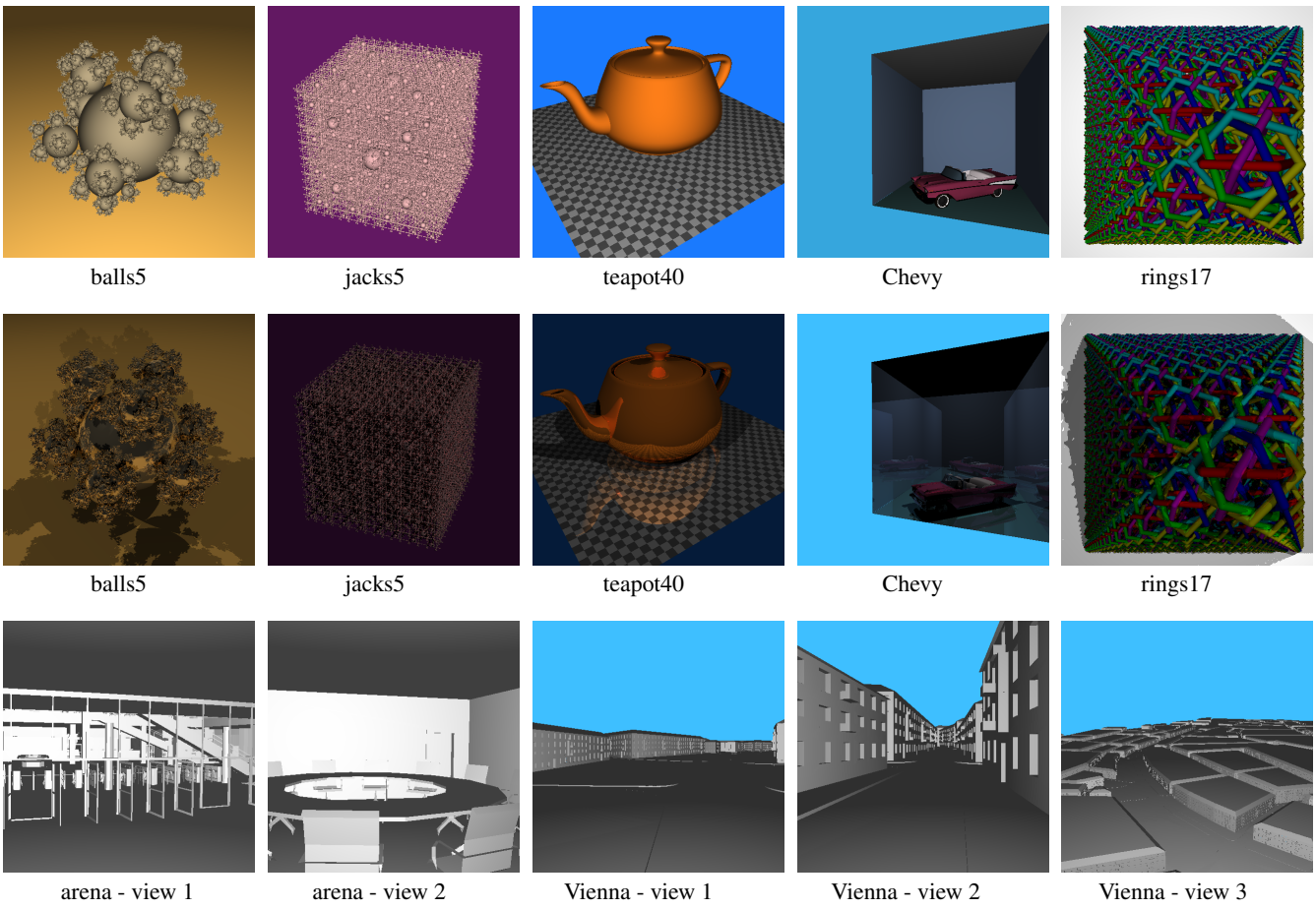


Figure 5: Snapshots of selected representative scenes and their view points. Top row: ray casting of scenes with low occlusion. Middle row: ray tracing of scenes with low occlusion. Bottom row: ray casting of scenes with high occlusion.

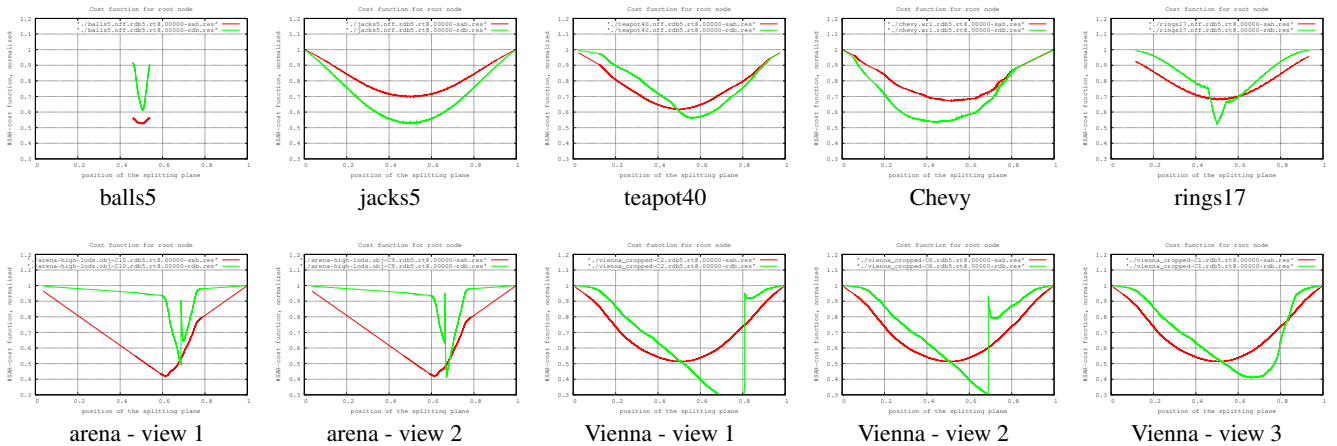


Figure 6: Comparison of the cost functions for root nodes of selected scenes. The red curve corresponds to SAH and the green curve to the RDH. Note that in most cases the RDH provides a steeper local minimum, which is slightly different from the position of the minimum determined by SAH.

Appendix E

Fast Insertion-Based Optimization of Bounding Volume Hierarchies

Bittner, J. - Hapala, M. - Havran, V.: Fast Insertion-Based Optimization of Bounding Volume Hierarchies. COMPUTER GRAPHICS FORUM. 2013, vol. 32, no. 1, p. 85-100. ISSN 0167-7055. **IF=1.636**

DOI: 10.1111/cgf.12000

COMPUTER GRAPHICS *forum*
Volume 32 (2013), number 1 pp. 85–100

Fast Insertion-Based Optimization of Bounding Volume Hierarchies

Jiří Bittner, Michal Hapala and Vlastimil Havran

Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic
{bittner, hapalmic, havran}@fel.cvut.cz

Abstract

We present an algorithm for fast optimization of bounding volume hierarchies (BVH) for efficient ray tracing. We perform selective updates of the hierarchy driven by the cost model derived from the surface area heuristic. In each step, the algorithm updates a fraction of the hierarchy nodes to minimize the overall hierarchy cost. The updates are realized by simple operations on the tree nodes: removal, search and insertion. Our method can quickly reduce the cost of the hierarchy constructed by the traditional techniques, such as the surface area heuristic. We evaluate the properties of the proposed method on fourteen test scenes of different complexity including individual objects and architectural scenes. The results show that our method can improve a BVH initially constructed with the surface area heuristic by up to 27% and a BVH constructed with the spatial median split by up to 88%.

Keywords: BVH, surface area heuristics, ray tracing.

ACM CCS: I.3.7[Computer Graphics]: Three-Dimensional Graphics and Realism-Ray Tracing.

1. Introduction

The current ray-tracing based algorithms allow to capture complex illumination effects and reach high degree of realism of the rendered images. With advances in computational power and algorithmic efficiency the ray tracing based methods have also become an alternative to rasterization for interactive and real time applications. Unlike rasterization, ray tracing relies on a highly efficient acceleration data structure which allows to trace tens or hundreds million rays per second.

Constructing such high-quality acceleration data structures is thus very important and it has received a strong attention in the last two decades. Even a relatively small improvement in performance can bring an interactive ray tracer closer to real-time or provide significant time savings when rendering many high-quality images of complex and detailed scenes in the movie industry.

There are two major classes of data structures used for ray tracing acceleration: spatial subdivisions (such as kd-trees, octrees or grids) and bounding volume hierarchies (BVH). In this paper, we propose a method which allows to optimize a given BVH beyond the current state of the art techniques. The method is based on an iterative algorithm that changes the topology of inner nodes of a BVH to improve the quality of the hierarchy initially built with arbitrary method. An example visualization of the reduction of traversal steps achieved by the proposed method is shown in Figure 1.

Our method constructs more efficient BVH in shorter time than previous approaches, whereas it allows to easily trade-off the time used for updating the BVH and the expected traversal cost. Compared to the current state of the art approach for high quality BVHs proposed by Kensler [Ken08] (simulated annealing), our algorithm is 25–147 times faster for architectural scenes, whereas achieving BVHs of comparable or even better quality. The method is applicable to a BVH built with an arbitrary technique, it is simple to implement and thus it has a potential to become a common optimization approach following the construction of the hierarchy.

The paper is further structured as follows. In Section 2, we describe the related work, Section 3 describes the proposed algorithm, Section 4 shows the results and Section 5 concludes the paper.

2. Related Work

There is a large body of literature on efficient spatial data structures for rendering. We restrict the discussion to the BVH methods for efficient rendering of static scenes.

The BVHs have a long tradition in the context of ray tracing. Rubin and Whitted used rectangular bounding volumes to create a BVH manually [RW80]. Weghorst *et al.* [WHG84] studied different types of bounding volumes for BVH and used a modelling hierarchy to build it. Kay and Kajiya [KK86] suggested to create

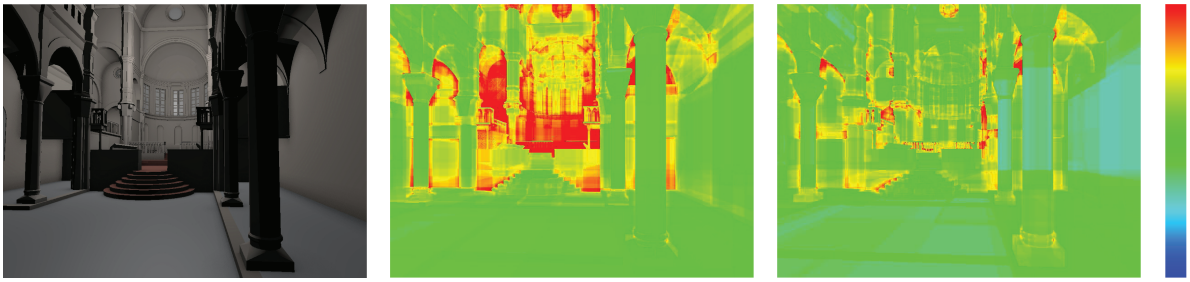


Figure 1: *Visualization of the number of traversal steps for the view of the Sibenik Cathedral scene rendered with primary and ambient occlusion rays. (a) Rendered image. (b) Number of traversal steps per ray for BVH built with SAH. (c) Number of traversal steps for the BVH optimized by the proposed method. On this image, our method gives an approximately* 16% *performance gain over the base-build algorithm with the SAH. (d) Bar-graph with pseudo colours used in the two visualization images—blue colour corresponds to 0 traversal steps, red colour to 100 or more steps.*

a BVH by an automatic top down recursive algorithm with a spatial median method and studied the efficiency of different bounding volumes. Goldsmith and Salmon [GS87] proposed the measure currently known as the *surface area heuristic* (SAH) which predicts the efficiency of the hierarchy already during the BVH construction. However, as they have built up the BVH incrementally by insertion, the BVH was usually inefficient as shown in performance study by Havran [Hav00] and later by Masso *et al.* [ML03]. The improvements for the efficiency of BVH were discussed in several articles in Ray Tracing News [RTN] and suggested in literature more than a decade ago, for example in the paper by Smits [Smi98].

BVHs are mostly used with axis-aligned bounding boxes as bounding volumes. Wächter and Keller [WK06] and Woop *et al.* [WMS06] used different bounding primitives resulting in lightweight hierarchies. Another research effort was devoted to decreasing the memory size by hierarchical encoding of the bounding volumes [MW06, KMKY10] or their better memory layout with respect to data traffic [YM06].

The precise evaluation of SAH requires sorting and thus exhibits $O(N \log N)$ complexity (N is the number of scene triangles). To reduce the constants behind the asymptotic complexity Havran *et al.* [HHS06], Wald *et al.* [Wal07, WBS07, WIP08] and Ize *et al.* [IWP07] used approximate SAH evaluation based on binning and centroids of bounding boxes of triangles. Another method to reduce the sorting demands was proposed by Hunt *et al.* [HMF07] who suggest reusing the organization of geometric data in the scene graph. They show that when certain assumptions about the scene graph hold, it is possible to achieve linear time complexity of BVH construction. Dammertz *et al.* [DHK08] proposed the variation of BVH using the branching factor of four to gain better utilization of SIMD units in modern CPUs.

Wald [Wal07] proposed a CPU-based parallel BVH construction by separating the build up process into two stages—horizontal and vertical parallel execution. Another asynchronous construction method was presented by Ize *et al.* [IWP07]. More recently, the parallel build-up of BVH has been demonstrated also on GPU by Lauterbach *et al.* [LGS*09], using 3D space-filling curve. Aila and Laine [AL09] identify the sources of inefficiency for traversing BVH on a GPU and propose a simple solution to improve the performance. Wald studied the possibility of fast rebuilds from scratch on upcoming Intel architecture with many cores [Wal12]. Pantaleoni and Luebke [PL10] and Garanzha *et al.* [GPM11] recently proposed GPU based methods for parallel BVH construction that are able to very quickly construct a BVH, however for complex scenes the BVH quality for these methods can be significantly lower than that of the full SAH builder.

Several papers have been published that relax the condition of having only one reference to each primitive. Ernst and Greiner [EG07] suggest to precompute several bounding boxes for each primitive and use them independently during top down construction. More recently, to achieve higher performance Stich *et al.* [SFD09] and Popov *et al.* [PGDS09] suggest to use spatial splits during the BVH construction, hence reintroducing some properties of kd-trees to BVHs.

Recently more interest has been devoted to methods, which are not limited to top-down BVH construction. Walter *et al.* [WBKP08], propose to use bottom-up agglomerative clustering for constructing high quality BVH. Kensler [Ken08] proposes to optimize the BVH in post-process using tree rotations with the hill climbing or simulated annealing optimization algorithms. Both these approaches allow to decrease the expected cost of BVH compared to the top down approach.

The goal of our work is the most similar to the paper of Kensler [Ken08], where the BVH is optimized beyond the common golden standard, i.e. BVHs built in top down fashion with the SAH. Compared to the method of Kensler [Ken08] our optimization procedure is significantly faster, and for complex scenes it results in hierarchies with lower costs.

3. Selective BVH Updates

This section describes our method starting with the algorithm outline, followed by a detailed description of individual steps of the algorithm, discussion of the design choices and providing further implementation details.

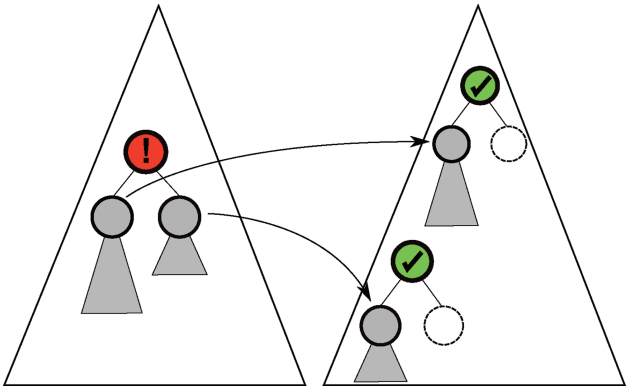


Figure 2: *The core of our method is the removal of inefficient nodes from the tree and re-insertion of their children to positions that decrease the overall cost of the tree.*

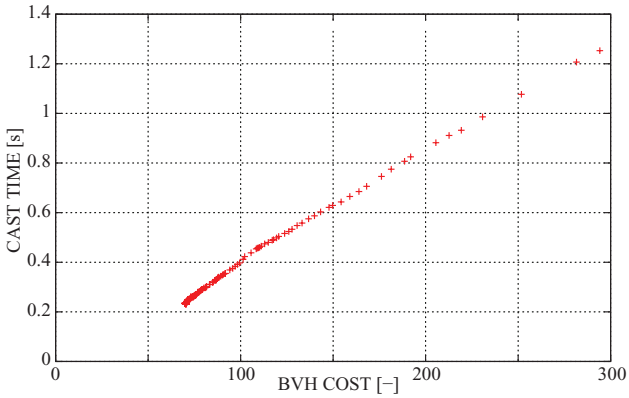


Figure 3: *Dependence of the ray casting time on the cost of the tree (measured on the Sibenik Cathedral scene) for different BVHs. The trees of different costs were obtained during the cost-optimization algorithm described in the paper.*

3.1. Algorithm overview

The BVH as an input of our algorithm can be built with various ways. We construct a BVH using a standard top down technique with the cost model based on the SAH as used for example in [Wal07, HSZ*11]. Alternatively we can use a faster BVH construction method based on object, or spatial median splits, or the method by [PL10]. Our algorithm performs the following steps (see also Figure 2):

(i) Begin **While Loop**

1) Select inner nodes for optimization.
2) For each selected node,

a) remove both its children from the tree,
b) find a position to reinsert the children using a cost-driven branch and bound search, and
c) insert each of the two children at their new positions and refit bounding volumes of all affected nodes.

(ii) End **While Loop** (until termination criteria are met).

The core of our method lies in steps (1) and (2) of the algorithm. In step (1), we select the inner nodes for optimization and in step (2) these nodes are removed from the tree and then reinserted back into the tree at more appropriate positions. In the next section, we recall the cost model behind our approach and then discuss individual steps of the algorithm in more detail.

3.2. Cost model

The SAH [MB90, Hav00] is usually described using a formula evaluating for each node the expected number of operations for processing a given ray, i.e. the cost of the node. In particular given a node N and assuming uniformly distributed unoccluded rays, which intersect the bounding volume of the node N, the expected cost of the node $C(N)$ is given as,

$$C(N) = \begin{cases} c_T + \dfrac{\mathrm{SA}(L(N)) \cdot C(L(N)) + \mathrm{SA}(R(N)) \cdot C(R(N))}{\mathrm{SA}(N)} & \text{if } N \text{ is inner} \\ c_I \cdot t_N & \text{if } N \text{ is leaf} \end{cases},$$

where c_T is the cost of traversing the inner node of the tree including the box intersection calculations, c_I is the cost for ray triangle intersection, t_N is the number of triangles in leaf N, $\mathrm{SA}(x)$ is the surface area of the bounding box associated with the node x, and $L(N)$ and $R(N)$ are the left and right children of N, respectively. Note that for hierarchies with larger branching factor than two, the traversal cost c_T would increase.

The SAH makes two assumptions: (1) the distribution of rays is uniform, (2) the rays are unoccluded thus the traversal does not terminate when a ray intersects a geometric primitive. Although these assumptions are generally not met in practice, the experiments indicate that the cost model with the SAH expresses the runtime behaviour of a ray tracing quite well (see Figure 3). Therefore, reducing the cost is directly reflected in reducing ray tracing times and as we show in Section 4.

The cost of the root node expresses the expected number of operations to process a ray intersecting the scene. By a simple derivation the recursion can be eliminated and the cost of the tree C(T) can be rewritten as:

$$\mathrm{C(T)} = \frac{1}{\mathrm{SA(T)}} \left[c_T \cdot \sum_{N \in \text{inner nodes}} \mathrm{SA}(N) + c_I \cdot \sum_{N \in \text{leaves}} \mathrm{SA}(N) \cdot t_N \right], \quad (1)$$

where SA(T) is the surface area of the bounding box of the scene. Note that the second term in the formula representing the ray triangle intersection calculations is constant for a given scene supposed there is a fixed number of primitives per leaf. Thus, the cost term which should primarily be optimized is the sum of surface areas of inner nodes in the tree which induces the traversal overhead

of the interior part of the tree ($c_T \cdot \sum \mathrm{SA}(N)$). This is exactly the core of our approach—we perform *global updates* of the tree by removing and reinserting nodes to minimize the total sum of surface areas of inner nodes. This contrasts to previous BVH optimization techniques which use local operations on the tree, such as rotations.

3.3. Updating nodes

Let us assume that we have identified a node N in the tree which causes a cost overhead. The key idea of our method is to remove the child nodes of N from the tree and reinsert both of them back at more appropriate positions. For the two child nodes we perform a global search to find the insertion positions that will minimize the tree cost. Once the insertion position is found, the node is inserted in the tree using local operations. Note that although we use a global search for the best position to reinsert the nodes, the method performs a *greedy* optimization because we always choose the positions which minimize the current tree cost.

The rest of this section describes the steps of removing, searching and reinsertion in more detail. In Section 3.4, we then describe how to actually select the nodes to be updated.

3.3.1. *Removing nodes*

When updating an inner node N, we remove N, its children L and R and its parent P from the tree. Then we update the links of the affected nodes to keep the topological consistency of the tree. We also update the bounding boxes of the affected nodes by traversing up to the root of the tree. We put the children L and R in an ordered list of nodes to be reinserted, whereas the nodes are ordered so that the node with larger surface area will be processed first. The nodes N and P are placed in a list of nodes which will be used to link the reinserted nodes with the nodes at the new positions of the tree. The removal operation is illustrated in Figure 4. Note that nodes L and R need not be leaves, but they can represent whole subtrees of the BVH.

3.3.2. *Searching for new positions*

We start the search for the new position to insert the node L at the root of the BVH and incrementally compute the total increase of the surface area. When reaching a node X, the surface area and therefore the cost increase is given by two components,

(1) The *direct cost* $C_D(L, X) = \mathrm{SA}(X \cup L)$, where $\mathrm{SA}(X \cup L)$ denotes the surface area of the box that is a union of bounding boxes of the node L and the node X.
(2) The *induced cost* $C_I(L, X)$, that is the accumulated increase of the surface area on the path from the root to the parent of the node X assuming the node L would be inserted in the subtree rooted at X. This can be defined also recursively so $C_I(L, X) = 0$ if X is the root node and $C_I(L, X) = C_I(L, \mathrm{parent}(X)) + \mathrm{SA}(\mathrm{parent}(X) \cup L) - \mathrm{SA}(\mathrm{parent}(X))$, otherwise.

The two components of the cost increase are illustrated in Figure 5. The total increase of the surface area of the tree is

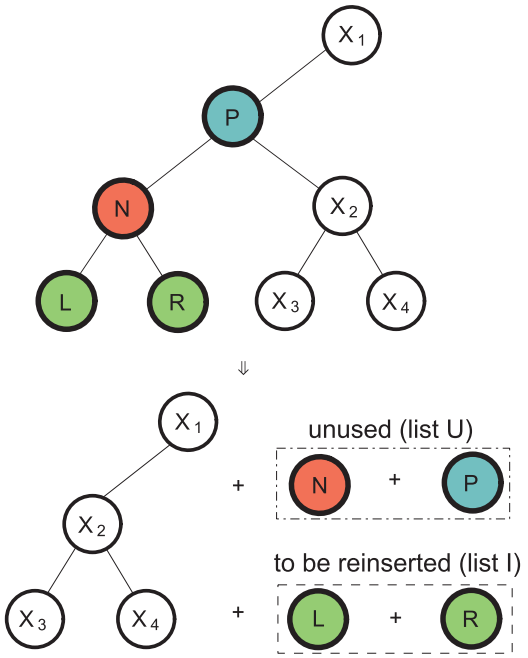


Figure 4: *Illustration of removal of node N—RemoveNode(node N, list U, list I, root P) operation. The children of N are put into the list I that contains the nodes to be inserted. Nodes N and P are put to the list U that stores the nodes that can be reused.*

considered as the cost for merging L and X, that is $C(L, X) = C_D(L, X) + C_I(L, X)$. We search for such a node X_{best} that minimizes this cost in the whole tree.

We use a *branch and bound* algorithm based on a priority queue in which the priority is inversely proportional to the induced cost. We can prune the search along the tree effectively using the smallest cost C_{best} corresponding to node X_{best} found so far. We evaluate a lower bound of the cost in the subtree of X in order to decide whether to continue the search in that subtree. The lower bound of the cost is given by the induced cost above X and the surface area of L, which is the lower bound of the direct cost in the whole subtree of X—the induced cost represents the necessary enlargement of nodes above X and the surface area of L is the minimum size of the node inserted into the tree, which joins L with a node from the tree. The subtree of X is traversed only if the lower bound of the cost is smaller than C_{best}. The whole algorithm can be terminated if the lower bound of the cost for the node on the top of the priority queue is larger than C_{best}. The pseudocode of the searching algorithm is shown in Algorithm 1.

Note that we have also experimented with using the total cost of the node X for driving the priority queue (instead of the induced cost), but in that case the pruning of the search was not as efficient as for using only the induced cost for computing the priority.

Similar cost model was used in the early work of Goldsmith and Salmon [GS87] who aimed to optimized the tree construction by minimizing the overall cost of the tree during incremental insertion of primitives. They proposed to track the 'inheritance cost' which corresponds to our induced cost. However, the actual search of the

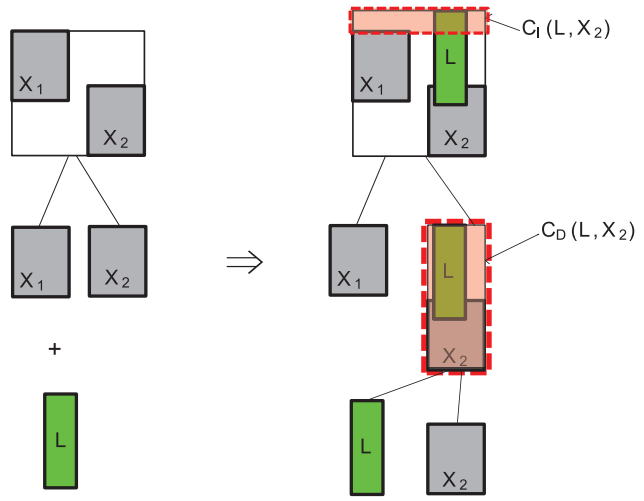


Figure 5: *A 2D example of how the total cost increase of adding the node L to the tree at node X_2 is computed. The induced cost $C_I(L, X_2)$ results from enlarging ascendants of X_2 when inserting the node L as the bounding boxes have to be refitted* (Algorithm 1:line 20). *The direct cost $C_D(L, X_2)$ is surface area of the box for the union of X_2 and L* (*Algorithm* 1:line 11).

tree was limited either to a greedy decision and a traversal of a single path or spreading the search in all subtrees for higher levels of the tree. This method was later improved in independent work by Omohundro [Omo89] who proposed to use priority queue when constructing an optimized sphere tree for proximity searching. Our method differs from the search technique of Omohundro by using different metric for priority (induced cost instead of total cost). Also our target application is different as we use the search within the tree update procedure and work with complete subtrees instead of incremental insertion of scene primitives.

3.3.3. *Reinserting nodes*

After we select the node X_{best} for insertion, which minimizes the cost increase in the whole BVH, we simply merge X_{best} and L using one of the removed nodes (N or P) as their parent. After each reinsertion we update all the bounding boxes along the path from the parent of the merged nodes to the root. The algorithm is illustrated in Figure 6.

3.4. Selecting nodes for update

The update procedure described above process arbitrarily selected nodes in the tree. An obvious choice for selecting the nodes for updates is random sampling. When using a random sampling we observe that the BVH cost is reduced until a point where it converges.

To accelerate the tree optimization we should first update those nodes that cause the highest cost overhead (surface area increase) in the tree. To achieve this we need a node inefficiency measure that would ideally correlate with the actual cost reduction when updating the node. We experimented with numerous node inefficiency measures and below we discuss the three most important ones which we finally combine together.

Algorithm 1 FindNodeForReinsertion(*node L*)—pseudo-code of finding suitable inner node or leaf for reinsertion of the candidate node L so the total cost increase is minimized. Constant ε is a small positive number (e.g. 10^{-20}), the entries with the highest priorities are removed first from the priority queue PQ.

```
1  Algorithm:FindNodeForReinsertion(node L)
2  // C_best - the smallest total cost increase found so far
   C_best = infinity;
3  // Priority queue contains pairs: (node, induced cost)
   Push (Root of BVH, 0, 1/ε) to priority queue PQ;
4  while PQ is not empty do
5      (X, C_I(L,X)) = Pop node from PQ;
6      if C_I(L,X) + SA(L) >= C_best then
7          // Early termination - not possible
8          break; // to reduce the cost C_best
9      end
10     // Compute the total cost of merging L with X
11     C_D(L,X) = SA(X ∪ L); // Direct cost
12     C(L,X) = C_I(L,X) + C_D(L,X); // Total cost
13     if C(L,X) < C_best then
14         // Merging L and X decreases the best cost
15         C_best = C(L,X);
16         // X_best is the currently best node found
17         X_best = X;
18     end
19     // Calculate the induced cost for children of X
20     C_I = C(L,X) - SA(X);
21     // Check if the cost decrease is possible in subtree
22     if C_I + SA(L) < C_best then
23         if X is not a leaf then
24             // Search in both children
25             Push (left child of X, C_I, 1/(C_I+ε)) to PQ ;
26             Push (right child of X, C_I, 1/(C_I+ε)) to PQ ;
27         end
28     end
29 end
30 return X_best
```

The first node inefficiency measure M_{SUM} corresponds to a component of the cost model used by Lauterbach *et al.* [LYTM06] and is evaluated as,

$$M_{\text{SUM}}(N) = \frac{\text{SA}(N)}{1/|\text{children of } N| \cdot \sum_{X \in \text{children of } N} \text{SA}(X)},$$

where $\text{SA}(N)$ is the surface area of the evaluated node N, $|\text{children of } N|$ is the number child nodes of N and $\sum_{X \in \text{children of } N} \text{SA}(X)$ is the sum of surface areas for these child nodes. This measure estimates the relative increase of the surface area of the node with respect to average surface areas of the children. Thus, if there will be a lot of empty space inside the node this measure will be large.

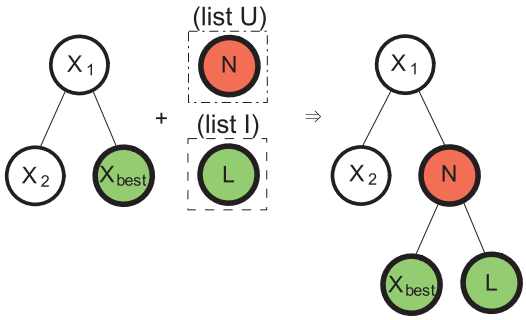


Figure 6: *Illustration of reinserting the node Lback to the tree. The node L is merged with the node X_{best} while using the node N as their new parent. The same procedure is used for inserting the node R (using P as the parent node).*

The second node inefficiency measure M_{MIN} is evaluated as,

$$M_{\text{MIN}}(N) = \frac{\text{SA}(N)}{\text{Min}_{X \in \text{children of } N}\text{SA}(X)},$$

where $\text{SA}(N)$ is the surface area of the evaluated node and $\text{Min}_{X \in children\ of\ N}\text{SA}(X)$ is the minimum of surface areas of its children. This measure aims to handle the situation when the node contains child nodes of significantly different sizes (e.g. one large node representing the whole terrain and a small node representing a particular object on the terrain). Then the M_{SUM} measure defined above might not identify such node as problematic as it takes the average surface area of the children which in this example will still be large. On the contrary the M_{MIN} measure will detect such a situation.

The third node inefficiency measure M_{AREA} directly corresponds to the surface area of the node,

$$M_{\text{AREA}}(N) = \text{SA}(N).$$

The M_{AREA} measure simple prioritizes the updates of the larger nodes of the tree because each such node has a significant contribution to the tree cost.

These three measures are able to detect some situations in which high global cost decrease can be expected. We have experimentally verified that the steepest decrease of the tree cost as a function of computation time was consistently achieved by combining these three measures together into,

$$M_{\text{COMB}}(N) = M_{\text{SUM}}(N) \cdot M_{\text{MIN}}(N) \cdot M_{\text{AREA}}(N).$$

Once the node inefficiency measure is defined, we can use it to prioritize the updates of the hierarchy nodes according to their inefficiency measure. Our method works in passes where in each pass it updates a specified number of nodes k (typically k=1% of nodes). These nodes are selected as follows: we evaluate the inefficiency measure for all inner nodes. Then we determine k nodes with the highest values of the inefficiency measure using a partial sort of the node array. These k nodes are then processed sequentially in descending order according to their inefficiency measures (the most inefficient nodes first).

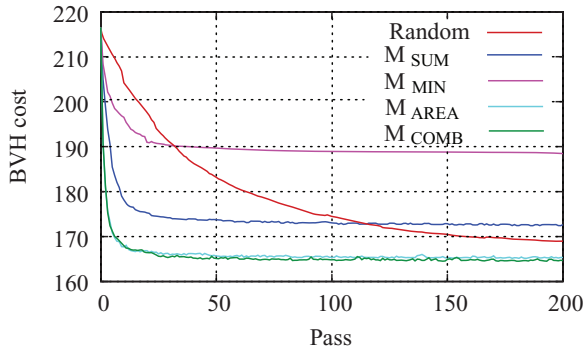


Figure 7: *Cost reduction using a random selection and the proposed node inefficiency measures on the Soda Hall scene. Note that the area measure and the combined measure achieve very fast decrease of the expected cost from 216 to 165. In one pass we update $k = 1\%$ of nodes.*

Note that an alternative would be to always update a single node with currently the highest inefficiency measure. The batch processing of nodes however speeds up the node selection procedure because the inefficiency measures are calculated only once per pass and are not updated after each change in the tree. In addition the batch processing makes the method more robust with respect to getting stuck in a local minimum for the case that the inefficiency measure of some node(s) is hard to reduce. A comparison of the decrease of the cost using the above described cost measures is shown in Figure 7.

3.5. Terminating the BVH optimization

In the beginning of the optimization, the vast majority of updates lead to reduction of the BVH cost. Because the removal operation removes two nodes from the tree and processes them sequentially (the first child is inserted in the tree while the second child is still removed), however, it is possible that after reinserting both nodes to the tree the BVH cost will increase. This behaviour becomes more apparent when the optimization converges and the BVH cost cannot be reduced anymore. Then the BVH cost oscillates in a small range near the reached minimum. Note that by a simple modification of the method which would always remove just one child from the tree we could ensure that the cost is either reduced in the given step or it remains the same. However, our experiments have shown that the BVH cost is reduced slightly more if we use the method of removing both children, although temporarily the optimization step might provide a small cost increase.

As the optimization is progressively reducing the cost we can use different termination criteria deciding when to stop the optimization such as the maximum time or the number of passes. The criteria can also be based on evaluating the convergence of the cost. We propose to terminate the computation when the cost does not improve within a given number of update passes p_{T} (recall that each pass updates certain number of nodes).

When using the combined inefficiency measure the cost might stabilize at a slightly higher value than when using random sampling

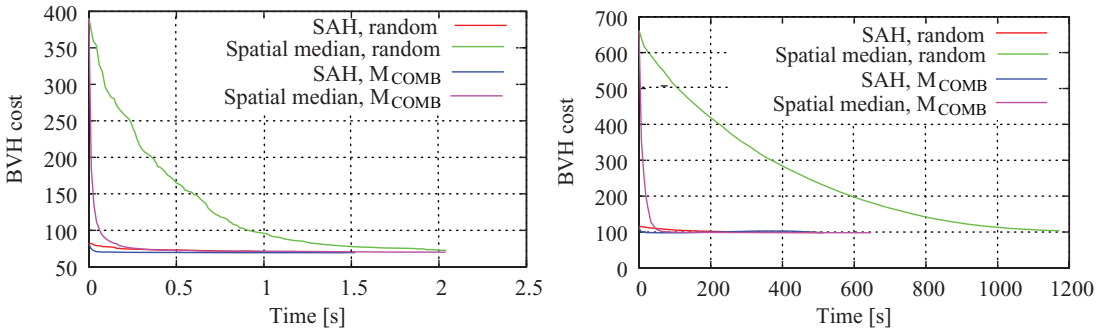


Figure 8: *BVH cost optimization for BVHs built by spatial median and SAH. (left) Sibenik Cathedral, (right) Power Plant. Our method can quickly optimize a tree, which was built either with spatial median or SAH. Note that both trees converge to very similar costs and the convergence is significantly faster for the M_{COMB} metrics (the plots show 200 update passes with 1% of updated nodes per pass).*

as there are some nodes that are never selected for optimization because their measure is low. To avoid this behaviour we switch to random sampling of nodes when we detect that the cost reduction becomes very low or even zero. Similarly to the termination of the whole computation this decision is made not for a single node but for sequence of processed nodes. We switch to the random selection if in the given number of passes p_R ($p_R \leq p_T$) the cost of the BVH does not reduce.

3.6. BVH tree compaction

We assumed that the BVH trees are constructed until each leaf contains a single triangle (or a geometric primitive in general). It is usually more beneficial to construct leaves with more triangles (e.g. 8 to 10) following the actual traversal and intersection constants (c_T and c_I) used in the SAH cost model [ML03, HHS06, Wal07, WBS07].

To get the most benefit from our optimization method we apply the method in two phases. First, we build the tree so that each leaf contains a single triangle. Second, we run the post-processing phase using a post-order traversal of the whole tree and evaluate the cost of each node using Equation (1). This requires also counting the number of triangles associated with the node during the post-order traversal. Whenever the cost of an interior node N is larger than the cost for a leaf created for the triangles contained in the leaves of the subtree rooted in the node N, we collapse the subtree to a leaf that references all the corresponding triangles. As a result the cost of the compacted tree can be significantly reduced compared to the tree with a single triangle per leaf (see the ratios between the *cost* and *ocost* in Section 4). Note that this post-processing phase happens after the optimization and before the use of BVH for ray tracing. The time needed for this phase is linear with the number of nodes and presents almost no computation time overhead.

Note that our tree optimization method can be easily applied even to BVHs with more triangles per leaf (i.e. those compacted before the optimization). In this case we have a lower number of nodes in the BVH which reduces the optimization time. On the other hand the assignment of triangles to the leaves will not be changed, although we could potentially find a tree with better distribution of triangles to leaves which reduces the tree cost.

3.7. Discussion

The idea of our optimization algorithm is to repeatedly optimize the BVH, which is similar to the paper of Kensler[Ken08]. There are two major differences in the core of the optimization procedure that result in a different behaviour of our method. First, Kensler's method *always traverses the whole tree* in a depth-first-search order to change the hierarchy around the traversed nodes. Our approach, on the other hand, selects the nodes to be optimized either randomly or using importance based decision. Second, the Kensler's method *changes the topology of the tree only locally* using one of four possible rotations. Although the sequence of rotations can principally lead to any BVH including the tree with a globally optimized cost, the changes in a BVH for one optimization step are much smaller

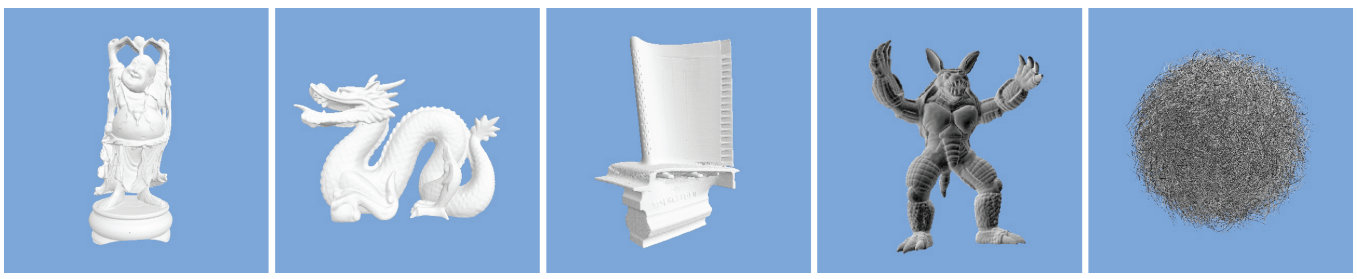


Figure 9: *Snapshots of scenes representing individual objects: Happy Buddha, Dragon, Blade, Armadillo and Hairball.*

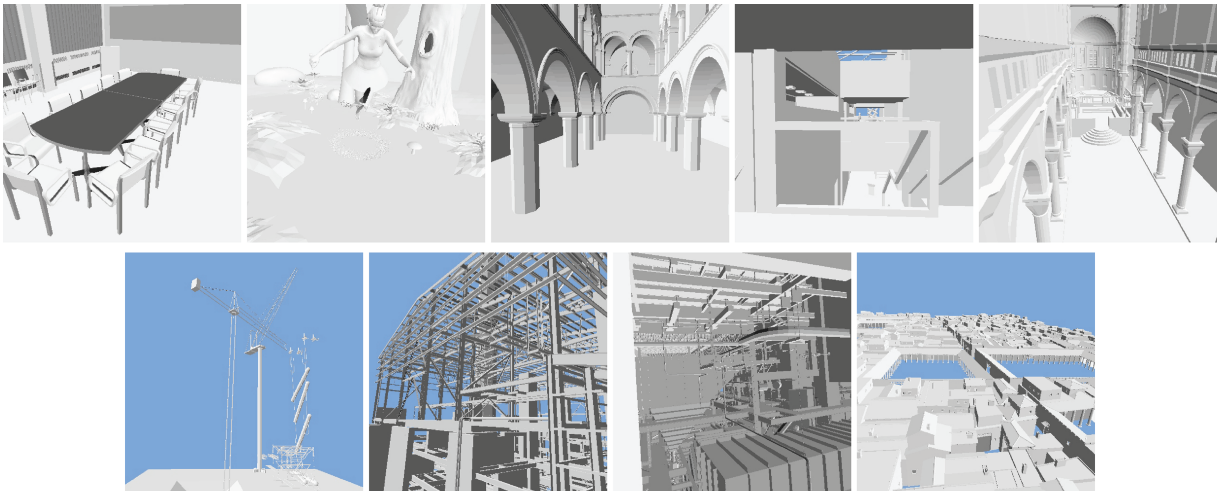


Figure 10: *Snapshots of architectural scenes: Conference, Fairy Forest, Sponza, Sodahall, Sibenik Cathedral, Power Plant section 9, Power Plant section 16, Power Plant and Pompeii Ten.*

than in our algorithm, where the subtrees are removed and then reinserted into the most appropriate position. That is, in our algorithm one optimization step can change (and often changes) the tree globally and hence has the potential of improving the cost more significantly.

Unlike Eisemann *et al.* [EGMM07] our search for the best candidate to reinsert the node intentionally starts at the root to find the best suitable position in the scope of the whole tree. Using the upwards traversal in the method of Eisemann *et al.* [EGMM07] reduces the number of traversal steps during the optimization, but does not necessarily find a good position for the inserted node.

4. Results

We have implemented the proposed algorithm in a single threaded C++ application. The results of BVH optimization and CPU ray tracing were evaluated on a PC with Linux OS, Intel(R) Xeon(R) CPU E5440 with the frequency 2.83 GHz and 40 GB of RAM and GNU C compiler version 4.40. To compute the cost according to equation (1) we used $c_T = 3.0$ and $c_I = 2.0$. As termination criteria described in Sections 3.4 and 3.5 we used $k = 1\%$, $p_T = 10$ and $p_R = 5$.

We have evaluated the method on fourteen models of different complexity which we classify into two distinct groups: (1) *individual objects* and (2) *architectural scenes*, i.e. models of larger spatial extent typically containing many objects. We have constructed the initial BVH by top down recursive procedure using a precise SAH builder which evaluates all discontinuities (two positions for each triangle) in the cost function for all three axes. The rendered images for objects and scenes are shown in Figures 9 and 10.

4.1. BVH cost reduction

We discuss separately the results for BVH built for individual objects and architectural scenes. As a reference for comparison we use our reimplementation of the state of the art BVH optimization algorithms of Kensler [Ken08] for both hill climbing and simulated annealing. For all tested methods we evaluate the BVH cost (denoted as $cost$), BVH cost optimized by compacting the tree with the method described in Section 3.6 ($ocost$), the time for optimizing the tree until predefined termination criteria ($CPUupdate$), the total build time including the optimization ($CPUbuild$) and the times for ray casting on the CPU and on the GPU for different ray types (primary rays, random rays, ambient occlusion rays).

4.1.1. *Individual objects*

As individual objects we have used five geometric data sets also used by Kensler's paper. These models exhibit relatively uniform distribution of triangles and low depth complexity. Therefore, the traditional top down build algorithm with the SAH constructs a high quality BVH that is difficult to optimize. For these scenes the updates provide practically no reduction of the BVH cost (see Table 3). It is worth to mention the 5% cost reduction for the Hairball scene, which results from its higher geometrical complexity (and hence possible source of inefficiency).

For BVHs initially constructed using spatial median splits our method optimizes the BVH cost close to the cost achieved by the top down SAH algorithm. The results are given in the Table 4. The final cost achieved by our algorithm is decreased the same or more than in the reference algorithm, the simulated annealing by Kensler [Ken08], however, the total build time achieves the great speedup, our algorithm is up to 10 to 25 times faster.

4.1.2. *Architectural scenes*

The results for nine more complex architectural scenes show our method can reduce the initial cost of the tree constructed with the traditional top down algorithm with SAH by 4–24% with an average of 17% (see Table 1). This cost reduction was achieved in time which was about three times smaller than the time of the initial tree construction (albeit the implementation of the exact SAH builder is not optimized). We can observe that the cost reduction of the

Table 1: *The results of the BVH optimization for architectural scenes. The base hierarchy for all methods (M) is created with a SAH top down driven build (method 1). For methods 2–5, the initially constructed BVH is followed by an optimization phase using different methods and settings: 2—our method with random selection, 3—our method with the combined measure of inefficiency, 4—Kensler's method, hill climbing, and 5—Kensler's method, simulated annealing. The values in brackets are the computed ratios against reference values (1.00).*

M	Cost [-]	0 Cost [-]	CPU update [s]	CPU build [s]	CPU primary [s]	CPU random [s]	GPU primary [ms]	GPU random [ms]	GPU occlusion [ms]
Conference (283k triangles)									
1	130.3 (1.00)	110.5 (1.00)	0.00	3.45 (1.00)	0.56 (1.00)	2.05 (1.00)	1.31 (1.00)	17.5 (1.00)	20.45 (1.00)
2	103.8 (0.79)	85.0 (0.76)	5.56	9.01 (2.60)	0.40 (0.72)	1.48 (0.72)	1.31 (1.00)	13.5 (0.77)	18.91 (0.92)
3	102.5 (0.78)	83.7 (0.75)	1.07	4.53 (1.31)	0.39 (0.69)	1.47 (0.71)	1.29 (0.98)	13.4 (0.77)	18.56 (0.90)
4	122.9 (0.94)	105.5 (0.95)	1.16	4.62 (1.33)	0.51 (0.91)	1.92 (0.93)	1.42 (1.08)	16.2 (0.92)	20.36 (0.99)
5	104.1 (0.79)	85.7 (0.77)	299.00	303.00 (87.50)	0.40 (0.71)	1.50 (0.72)	1.34 (1.02)	14.0 (0.80)	18.99 (0.92)
Fairy Forest (174k triangles)									
1	95.1 (1.00)	79.4 (1.00)	0.00	1.96 (1.00)	0.62 (1.00)	1.41 (1.00)	1.68 (1.00)	19.7 (1.00)	31.64 (1.00)
2	92.5 (0.97)	76.8 (0.96)	1.39	3.36 (1.70)	0.62 (1.00)	1.39 (0.98)	1.62 (0.96)	19.0 (0.96)	30.91 (0.97)
3	91.5 (0.96)	75.8 (0.95)	0.40	2.37 (1.20)	0.61 (0.98)	1.33 (0.94)	1.63 (0.97)	19.1 (0.97)	30.63 (0.96)
4	93.9 (0.98)	78.2 (0.98)	0.60	2.57 (1.30)	0.65 (1.04)	1.38 (0.97)	1.71 (1.01)	19.5 (0.99)	32.07 (1.01)
5	93.8 (0.98)	78.0 (0.98)	179.00	181.00 (91.80)	0.67 (1.07)	1.42 (1.00)	1.94 (1.15)	20.6 (1.04)	32.92 (1.04)
Sibenik Cathedral (80k triangles)									
1	82.3 (1.00)	71.5 (1.00)	0.00	0.66 (1.00)	0.55 (1.00)	1.19 (1.00)	1.73 (1.00)	8.7 (1.00)	25.59 (1.00)
2	70.2 (0.85)	61.6 (0.86)	1.19	1.85 (2.79)	0.48 (0.87)	1.00 (0.84)	1.42 (0.82)	7.1 (0.81)	20.93 (0.81)
3	69.5 (0.84)	61.1 (0.85)	0.72	1.38 (2.08)	0.47 (0.85)	0.96 (0.81)	1.41 (0.81)	7.1 (0.81)	22.32 (0.87)
4	77.9 (0.94)	68.7 (0.95)	0.22	0.88 (1.33)	0.55 (1.00)	1.15 (0.96)	1.72 (0.99)	8.5 (0.97)	25.48 (0.99)
5	70.8 (0.85)	62.0 (0.86)	79.12	79.79 (120.00)	0.52 (0.94)	1.00 (0.84)	1.73 (1.00)	8.3 (0.95)	25.49 (0.99)
Sponza (76k triangles)									
1	220.2 (1.00)	189.7 (1.00)	0.00	0.56 (1.00)	0.61 (1.00)	2.83 (1.00)	1.34 (1.00)	25.4 (1.00)	26.26 (1.00)
2	183.3 (0.83)	156.6 (0.82)	0.95	1.52 (2.70)	0.42 (0.70)	2.16 (0.76)	1.10 (0.82)	18.9 (0.74)	22.76 (0.86)
3	182.2 (0.82)	155.2 (0.81)	0.44	1.01 (1.79)	0.39 (0.65)	2.09 (0.73)	1.10 (0.82)	18.6 (0.73)	22.86 (0.87)
4	206.1 (0.93)	178.2 (0.93)	0.22	0.78 (1.39)	0.47 (0.78)	2.61 (0.92)	1.33 (0.99)	24.1 (0.94)	26.14 (0.99)
5	188.1 (0.85)	161.3 (0.85)	80.70	81.27 (144.40)	0.46 (0.76)	2.27 (0.80)	1.36 (1.01)	22.9 (0.89)	26.60 (1.01)
Soda Hall (2169k triangles)									
1	216.5 (1.00)	188.9 (1.00)	0.00	40.96 (1.00)	0.64 (1.00)	1.44 (1.00)	1.13 (1.00)	11.3 (1.00)	6.02 (1.00)
2	167.1 (0.77)	141.4 (0.74)	99.41	140.00 (3.42)	0.50 (0.77)	1.16 (0.80)	0.98 (0.86)	8.6 (0.76)	4.95 (0.82)
3	165.4 (0.76)	139.7 (0.73)	15.39	56.36 (1.37)	0.50 (0.77)	1.22 (0.84)	0.96 (0.84)	8.7 (0.76)	4.94 (0.82)
4	203.1 (0.93)	177.2 (0.93)	12.53	53.49 (1.30)	0.62 (0.97)	1.41 (0.97)	1.21 (1.07)	10.8 (0.95)	6.00 (0.99)
5	180.5 (0.83)	154.8 (0.81)	2570.00	2611.00 (63.70)	0.59 (0.92)	1.38 (0.95)	1.12 (0.99)	10.6 (0.93)	5.73 (0.95)
Power Plant, section 9 (122k triangles)									
1	57.9 (1.00)	38.7 (1.00)	0.00	1.36 (1.00)	0.16 (1.00)	0.75 (1.00)	0.86 (1.00)	4.5 (1.00)	1.33 (1.00)
2	51.8 (0.89)	32.8 (0.84)	2.78	4.14 (3.04)	0.14 (0.86)	0.64 (0.85)	1.08 (1.25)	4.2 (0.92)	1.26 (0.94)
3	51.8 (0.89)	32.6 (0.84)	0.26	1.62 (1.19)	0.15 (0.88)	0.64 (0.84)	1.12 (1.30)	4.2 (0.93)	1.26 (0.94)
4	54.9 (0.94)	36.1 (0.93)	0.59	1.96 (1.43)	0.16 (0.99)	0.72 (0.96)	1.11 (1.29)	4.3 (0.96)	1.29 (0.96)
5	51.9 (0.89)	32.8 (0.84)	124.00	125.00 (91.50)	0.15 (0.89)	0.64 (0.85)	0.98 (1.13)	4.2 (0.93)	1.28 (0.96)
Power Plant, section 16 (366k triangles)									
1	93.5 (1.00)	74.1 (1.00)	0.00	4.70 (1.00)	1.20 (1.00)	0.86 (1.00)	2.98 (1.00)	5.7 (1.00)	7.60 (1.00)
2	80.2 (0.85)	61.5 (0.82)	8.71	13.41 (2.85)	1.00 (0.83)	0.72 (0.84)	2.68 (0.89)	4.8 (0.85)	6.62 (0.87)
3	79.8 (0.85)	60.9 (0.82)	2.80	7.50 (1.59)	1.07 (0.89)	0.71 (0.82)	2.93 (0.98)	4.8 (0.84)	6.83 (0.89)
4	89.3 (0.95)	70.7 (0.95)	1.29	6.00 (1.27)	1.19 (0.99)	0.81 (0.94)	2.99 (1.00)	5.5 (0.96)	7.71 (1.01)
5	83.4 (0.89)	64.9 (0.87)	420.00	424.00 (90.20)	1.14 (0.95)	0.77 (0.90)	3.25 (1.09)	5.3 (0.92)	7.40 (0.97)
Power Plant (12 748k triangles)									
1	115.8 (1.00)	85.7 (1.00)	0.00	396 (1.00)	1.55 (1.00)	0.86 (1.00)	4.52 (1.00)	6.2 (1.00)	9.00 (1.00)
2	98.3 (0.84)	68.8 (0.80)	808.00	1204 (3.04)	1.03 (0.66)	0.68 (0.79)	4.37 (0.96)	4.9 (0.79)	6.86 (0.76)
3	97.8 (0.84)	68.3 (0.79)	85.52	481 (1.21)	1.12 (0.72)	0.68 (0.79)	4.19 (0.92)	5.0 (0.80)	6.74 (0.74)
4	110.9 (0.95)	81.5 (0.95)	72.29	468 (1.18)	1.49 (0.96)	0.82 (0.95)	4.54 (1.00)	5.8 (0.94)	8.75 (0.97)
5	102.5 (0.88)	73.1 (0.85)	13468.00	13863 (35.00)	1.38 (0.88)	0.75 (0.87)	4.48 (0.99)	5.4 (0.87)	8.20 (0.91)

(*continue*)

Table 1: *Continued.*

M	Cost [-]	0 Cost [-]	CPU update [s]	CPU build [s]	CPU primary [s]	CPU random [s]	GPU primary [ms]	GPU random [ms]	GPU occlusion [ms]
Pompeii Ten (5646k triangles)									
1	252.9 (1.00)	190.5 (1.00)	0.00	102 (1.00)	0.74 (1.00)	3.57 (1.00)	3.90 (1.00)	44.0 (1.00)	19.54 (1.00)
2	218.8 (0.86)	159.1 (0.83)	267.00	369 (3.61)	0.61 (0.81)	2.81 (0.78)	3.76 (0.96)	34.2 (0.77)	18.10 (0.92)
3	218.0 (0.86)	158.2 (0.83)	77.38	179 (1.75)	0.61 (0.81)	2.92 (0.82)	3.74 (0.95)	33.9 (0.76)	18.05 (0.92)
4	237.5 (0.93)	178.6 (0.93)	25.30	127 (1.24)	0.71 (0.95)	3.35 (0.94)	3.98 (1.02)	41.3 (0.93)	19.64 (1.00)
5	225.9 (0.89)	166.6 (0.87)	6049.00	6151 (60.30)	0.67 (0.90)	3.09 (0.86)	5.25 (1.34)	39.1 (0.88)	19.19 (0.98)

BVH optimized by our method is larger than that of the current state of the art methods for high quality BVH proposed by Kensler [Ken08]. The build time for the hill climbing reference method is slightly lower than for our method (note that this also depends on the particular termination criteria used in our method), but the hill climbing is not able to reduce the cost more than by a few percent.

The simulated annealing of Kensler [Ken08] achieves better cost reduction than the hill climbing, whereas the running time of the method is about two order of magnitude higher than for hill climbing. Our method however provides BVHs with even lower cost (up to 10% difference) than the simulated annealing by Kensler in computation time from 28 to 80 times smaller. This brings us to an observation that our proposed technique is currently able to construct the best known BVHs for the given scene and the cost model based on SAH. The BVH quality improvement over the previous state-of-the-art method is not dramatic, however the speed in which we obtain these improvements is significant (almost two orders of magnitude compared to the simulated annealing), which can actually lead to using the proposed technique in practice as the BVH build time is only slightly higher than without our method.

For BVHs built top down using a simple spatial median split the cost reduction achieved by our algorithm is very significant. The results are given in Table 2. Interestingly, the BVH cost quickly converges almost to the same cost as for the case when the BVH was built with top down build algorithm with SAH and optimized by our algorithm, but the optimization takes more computation time. This behaviour can be seen in Figure 8 where we show the optimization process using 200 passes and when updating 1% of inner nodes per pass. Again, the time needed for the build including optimization is 25–147 times smaller than the time of simulated annealing by Kensler.

To provide more compact overview of the results we have summarized all results for the architectural scenes in Table 5, where the cost without the reduction described in Section 3.6 is reported. The cost and time for the top down BVH build with SAH is taken there as a reference.

4.1.3. *BVH structure analysis*

To find out what particular changes to the BVH our algorithm does we calculated histograms of the surface areas of the nodes at different depths of the tree for the initial BVH and for the BVH optimized by our method. These histograms are shown in Figure 11.

The plots show that our optimization method reduces the sum of surface areas of inner nodes especially for the middle range depths. We can also observe that this is achieved by restructuring the tree so that certain nodes are placed deeper in the tree.

4.1.4. *BVH tree compaction*

We also evaluated the contribution of the tree compaction described in Section 3.6. The *ocost* after the compaction can be non-negligibly lower than the initial *cost* before performing the tree compaction. In particular, the *ocost* is by about 8% lower than *cost* for individual objects. For architectural scenes the difference is even more significant, for BVH top down build with SAH it reaches 20% and for the BVH built with spatial median 17% on average. Interestingly, the impact of our tree optimization algorithm is typically a few percent larger when considering the ratio of *ocost* than the ratio of *cost* before and after the optimization. For example for the Soda Hall scene the *cost* is reduced by 24% (ratio 0.76), whereas the *ocost* is reduced by 27% (ratio 0.73) (see Table 1).

4.2. Ray tracing performance

We have evaluated the optimized BVH using two different ray tracers. The first one is a CPU based ray tracer with no low level optimizations, the second one is a GPU ray tracer derived from the implementation of Karras *et al.*'s [KAL09]. For the GPU ray tracer we have used an adaptor that converts the main memory data structures to the data structures used by the GPU application CUDA kernels.

4.2.1. *CPU ray tracer results*

The CPU ray tracing performance of the constructed BVH has been evaluated for two scenarios – casting primary rays and shooting random rays. The figures computed for casting primary rays are shown in Figure 10. Note that the time for tracing rays is in strong correlation with the cost irrespective of the scenario for both primary and random rays for all reported results.

Table 2: *The results for architectural scenes where the base hierarchy for all methods is created with a spatial median top down driven build (method 1), which for methods 2–5 is followed by an update phase. These are as follows: 2—our method with random selection, 3—our method with the combined measure of inefficiency, 4—Kensler's method, hill climbing, and 5—Kensler's method, simulated annealing. The values in brackets are the computed ratios against reference values (1.00).*

M	Cost [-]	0 Cost [-]	CPU update [s]	CPU build [s]	CPU primary [s]	CPU random [s]	GPU primary [ms]	GPU random [ms]	GPU occlusion [ms]
Conference (283k triangles)									
1	842.2 (1.00)	789.5 (1.00)	0.00	0.44 (1.00)	5.56 (1.00)	16.55 (1.00)	11.00 (1.00)	198.6 (1.00)	172.03 (1.00)
2	103.4 (0.12)	84.7 (0.10)	15.05	15.50 (34.76)	0.39 (0.07)	1.49 (0.09)	1.32 (0.12)	13.8 (0.06)	19.74 (0.11)
3	102.7 (0.12)	84.1 (0.10)	7.37	7.81 (17.53)	0.38 (0.06)	1.53 (0.09)	1.38 (0.12)	13.7 (0.06)	19.27 (0.11)
4	213.7 (0.25)	197.4 (0.25)	2.74	3.19 (7.16)	1.39 (0.25)	4.43 (0.27)	2.97 (0.27)	41.7 (0.21)	47.43 (0.27)
5	105.2 (0.12)	86.6 (0.10)	307.87	308.32 (691.42)	0.39 (0.07)	1.51 (0.09)	1.27 (0.11)	13.8 (0.06)	19.41 (0.11)
Fairy Forest (174k triangles)									
1	185.0 (1.00)	171.3 (1.00)	0.00	0.24 (1.00)	1.95 (1.00)	3.16 (1.00)	4.73 (1.00)	50.1 (1.00)	93.52 (1.00)
2	93.8 (0.50)	78.1 (0.45)	4.13	4.38 (17.55)	0.66 (0.34)	1.42 (0.44)	1.76 (0.37)	20.0 (0.39)	32.83 (0.35)
3	92.9 (0.50)	77.3 (0.45)	0.99	1.24 (5.00)	0.65 (0.33)	1.39 (0.44)	2.23 (0.47)	20.2 (0.40)	38.28 (0.40)
4	123.5 (0.66)	110.5 (0.64)	0.91	1.16 (4.65)	1.04 (0.53)	2.10 (0.66)	2.91 (0.61)	30.7 (0.61)	53.06 (0.56)
5	95.6 (0.51)	80.0 (0.46)	182.62	182.87 (731.61)	0.68 (0.35)	1.48 (0.46)	1.88 (0.39)	20.7 (0.41)	33.71 (0.36)
Sibenik Cathedral (80k triangles)									
1	390.7 (1.00)	380.4 (1.00)	0.00	0.09 (1.00)	4.14 (1.00)	7.61 (1.00)	10.11 (1.00)	75.0 (1.00)	188.32 (1.00)
2	71.1 (0.18)	63.0 (0.16)	2.38	2.47 (26.92)	0.50 (0.12)	1.00 (0.13)	1.50 (0.14)	7.5 (0.09)	25.17 (0.13)
3	71.3 (0.18)	63.3 (0.16)	1.07	1.16 (12.75)	0.53 (0.12)	0.98 (0.12)	1.68 (0.16)	7.3 (0.09)	27.98 (0.14)
4	121.1 (0.31)	113.7 (0.29)	0.96	1.05 (11.51)	1.43 (0.35)	2.17 (0.28)	4.04 (0.40)	17.7 (0.23)	69.68 (0.37)
5	72.6 (0.18)	63.5 (0.16)	81.62	81.71 (888.42)	0.56 (0.13)	1.01 (0.13)	1.53 (0.15)	7.6 (0.10)	26.48 (0.14)
Sponza (76k triangles)									
1	1258 (1.00)	1205.4 (1.00)	0.00	0.07 (1.00)	7.25 (1.00)	22.45 (1.00)	14.40 (1.00)	288.2 (1.00)	295.31 (1.00)
2	185.5 (0.14)	158.8 (0.13)	3.00	3.08 (39.14)	0.45 (0.06)	2.24 (0.10)	1.20 (0.08)	20.5 (0.07)	25.08 (0.08)
3	183.4 (0.14)	156.7 (0.12)	2.03	2.11 (26.81)	0.39 (0.05)	2.17 (0.10)	1.19 (0.08)	19.7 (0.06)	24.16 (0.08)
4	498.4 (0.39)	476.5 (0.39)	0.74	0.82 (10.47)	3.25 (0.44)	8.88 (0.41)	6.29 (0.44)	112.0 (0.38)	121.37 (0.40)
5	196.5 (0.15)	169.1 (0.14)	83.73	83.81 (1061.12)	0.49 (0.06)	2.49 (0.11)	1.35 (0.09)	22.5 (0.07)	28.71 (0.09)
Soda Hall (2169k triangles)									
1	1396 (1.00)	1355.7 (1.00)	0.00	4.29 (1.00)	11.82 (1.00)	18.09 (1.00)	21.58 (1.00)	261.8 (1.00)	86.88 (1.00)
2	168.2 (0.12)	142.6 (0.10)	198.90	203.20 (47.33)	0.62 (0.05)	1.29 (0.07)	1.16 (0.05)	9.4 (0.03)	5.45 (0.06)
3	165.8 (0.11)	140.2 (0.10)	72.41	76.70 (17.91)	0.52 (0.04)	1.21 (0.06)	1.17 (0.05)	8.7 (0.03)	4.77 (0.05)
4	486.7 (0.34)	468.9 (0.34)	27.63	31.93 (7.48)	3.25 (0.28)	5.27 (0.29)	10.31 (0.48)	60.1 (0.22)	24.68 (0.28)
5	195.4 (0.13)	170.2 (0.12)	2399.95	2404.29 (560.03)	0.69 (0.05)	1.64 (0.09)	1.41 (0.06)	12.9 (0.04)	6.75 (0.07)
Power Plant, section 9 (122k triangles)									
1	188.7 (1.00)	169.6 (1.00)	0.00	0.16 (1.00)	1.03 (1.00)	3.54 (1.00)	4.95 (1.00)	22.0 (1.00)	3.28 (1.00)
2	52.1 (0.27)	32.9 (0.19)	4.04	4.21 (24.82)	0.15 (0.14)	0.65 (0.18)	1.13 (0.22)	4.3 (0.19)	1.27 (0.38)
3	52.3 (0.27)	33.1 (0.19)	2.62	2.79 (16.57)	0.15 (0.14)	0.66 (0.18)	1.21 (0.24)	4.3 (0.19)	1.28 (0.39)
4	77.8 (0.41)	60.8 (0.35)	1.05	1.22 (7.18)	0.33 (0.32)	1.43 (0.40)	1.82 (0.36)	8.3 (0.37)	1.77 (0.53)
5	51.9 (0.27)	32.8 (0.19)	128.18	128.35 (755.12)	0.15 (0.14)	0.65 (0.18)	1.21 (0.24)	4.2 (0.18)	1.27 (0.38)
Power Plant, section 16 (366k triangles)									
1	505.1 (1.00)	476.3 (1.00)	0.00	0.60 (1.00)	10.11 (1.00)	6.32 (1.00)	27.74 (1.00)	49.1 (1.00)	44.03 (1.00)
2	80.2 (0.15)	61.5 (0.12)	20.81	21.41 (35.21)	1.03 (0.10)	0.73 (0.11)	2.84 (0.10)	5.0 (0.10)	7.01 (0.15)
3	79.7 (0.15)	61.1 (0.12)	17.18	17.79 (29.28)	0.98 (0.09)	0.69 (0.11)	2.82 (0.10)	4.8 (0.09)	6.88 (0.15)
4	176.9 (0.35)	161.4 (0.33)	3.67	4.28 (7.03)	3.04 (0.30)	1.97 (0.31)	6.94 (0.25)	13.9 (0.28)	16.70 (0.37)
5	87.1 (0.17)	68.6 (0.14)	458.93	459.53 (754.25)	1.32 (0.13)	0.84 (0.13)	3.13 (0.11)	5.5 (0.11)	8.48 (0.19)
Power Plant (12 748k triangles)									
1	661.3 (1.00)	612.8 (1.00)	0	36 (1.00)	12.73 (1.00)	6.23 (1.00)	45.93 (1.00)	56.0 (1.00)	74.42 (1.00)
2	97.8 (0.14)	68.3 (0.11)	1958	1994 (55.51)	1.05 (0.08)	0.67 (0.10)	4.45 (0.09)	5.0 (0.08)	6.80 (0.09)
3	98.6 (0.14)	69.0 (0.11)	249	285 (7.92)	1.09 (0.08)	0.69 (0.11)	3.65 (0.07)	4.9 (0.08)	6.70 (0.09)
4	205.5 (0.31)	179.9 (0.29)	258	294 (8.18)	5.00 (0.38)	1.96 (0.31)	16.10 (0.35)	16.0 (0.28)	29.13 (0.39)
5	108.7 (0.16)	79.5 (0.12)	13809	13845 (385.23)	1.46 (0.11)	0.83 (0.13)	4.82 (0.10)	6.0 (0.10)	9.34 (0.12)

(*continue*)

Table 2: *Continued.*

M	Cost [-]	0 Cost [-]	CPU update [s]	CPU build [s]	CPU primary [s]	CPU random [s]	GPU primary [ms]	GPU random [ms]	GPU occlusion [ms]
Pompeii Ten (5646k triangles)									
1	766.9 (1.00)	703.8 (1.00)	0	11 (1.00)	3.00 (1.00)	17.65 (1.00)	17.79 (1.00)	276.3 (1.00)	81.87 (1.00)
2	220.3 (0.28)	160.7 (0.22)	416	427 (38.92)	0.63 (0.20)	2.92 (0.16)	4.05 (0.22)	35.3 (0.12)	18.51 (0.22)
3	220.0 (0.28)	160.3 (0.22)	105	116 (10.63)	0.63 (0.20)	2.97 (0.16)	4.27 (0.24)	34.5 (0.12)	18.98 (0.23)
4	316.5 (0.41)	268.1 (0.38)	68	79 (7.20)	1.28 (0.42)	6.08 (0.34)	7.02 (0.39)	89.1 (0.32)	31.84 (0.38)
5	229.5 (0.29)	170.4 (0.24)	6203	6214 (565.91)	0.69 (0.22)	3.20 (0.18)	4.26 (0.23)	39.7 (0.14)	19.71 (0.24)

Table 3: *Results of the BVH optimization for individual objects. The base hierarchy for all methods (M) is created with a SAH top down driven build (method 1). For methods 2–5 the initially constructed BVH is followed by an optimization phase using different methods and settings: 2—our method with random selection, 3—our method with the combined measure of inefficiency, 4—Kensler's method, hill climbing, and 5—Kensler's method, simulated annealing. The values in brackets are the computed ratios against reference values (1.00).*

M	Cost [-]	0 Cost [-]	CPU update [s]	CPU build [s]	CPU primary [s]	CPU random [s]	GPU primary [ms]	GPU random [ms]	GPU amb. occlusion [ms]
Happy Buddha (1087k triangles)									
1	165.3 (1.00)	156.5 (1.00)	0.00	15.78 (1.00)	0.20 (1.00)	2.71 (1.00)	0.94 (1.00)	165.5 (1.00)	6.45 (1.00)
2	161.9 (0.97)	154.2 (0.98)	26.35	42.14 (2.66)	0.20 (1.01)	2.66 (0.98)	0.97 (1.03)	163.3 (0.98)	6.49 (1.00)
3	163.2 (0.98)	155.3 (0.99)	15.04	30.82 (1.95)	0.20 (1.02)	2.72 (1.00)	0.95 (1.01)	163.0 (0.98)	6.48 (1.00)
4	164.2 (0.99)	156.3 (0.99)	2.58	18.37 (1.16)	0.20 (1.00)	2.71 (0.99)	0.94 (1.00)	164.7 (0.99)	6.51 (1.00)
5	165.3 (1.00)	156.5 (1.00)	1135.00	1151.00 (72.92)	0.20 (1.00)	2.71 (1.00)	0.94 (1.00)	165.5 (1.00)	6.45 (1.00)
Dragon (871k triangles)									
1	145.4 (1.00)	138.1 (1.00)	0.00	11.96 (1.00)	0.26 (1.00)	2.37 (1.00)	1.00 (1.00)	156.8 (1.00)	8.80 (1.00)
2	144.5 (0.99)	137.9 (0.99)	6.21	18.18 (1.51)	0.27 (1.03)	2.41 (1.01)	1.07 (1.07)	155.9 (0.99)	8.92 (1.01)
3	145.4 (1.00)	138.1 (1.00)	1.21	13.18 (1.10)	0.26 (1.00)	2.38 (1.00)	1.08 (1.08)	155.6 (0.99)	8.88 (1.00)
4	144.7 (0.99)	138.0 (0.99)	1.80	13.77 (1.15)	0.26 (1.00)	2.37 (0.99)	1.01 (1.01)	156.4 (0.99)	8.85 (1.00)
5	145.4 (1.00)	138.1 (1.00)	911.00	923.00 (77.14)	0.26 (1.00)	2.37 (1.00)	1.00 (1.00)	156.8 (1.00)	8.80 (1.00)
Blade (1.765k triangles)									
1	190.3 (1.00)	178.8 (1.00)	0.00	27.37 (1.00)	0.21 (1.00)	2.35 (1.00)	0.97 (1.00)	10.4 (1.00)	4.01 (1.00)
2	190.3 (1.00)	178.8 (1.00)	3.24	30.61 (1.11)	0.22 (1.00)	2.38 (1.01)	0.97 (1.00)	10.8 (1.03)	4.03 (1.00)
3	190.3 (1.00)	178.8 (1.00)	2.35	29.72 (1.08)	0.22 (1.00)	2.38 (1.01)	0.99 (1.02)	10.7 (1.02)	4.03 (1.00)
4	190.1 (0.99)	178.7 (0.99)	4.65	32.02 (1.16)	0.21 (1.00)	2.34 (0.99)	0.97 (1.00)	10.4 (1.00)	4.00 (0.99)
5	190.3 (1.00)	178.8 (1.00)	1835.00	1863.00 (68.05)	0.21 (1.00)	2.35 (1.00)	0.97 (1.00)	10.4 (1.00)	4.01 (1.00)
Armadillo (307k triangles)									
1	86.3 (1.00)	82.5 (1.00)	0.00	3.35 (1.00)	0.22 (1.00)	1.53 (1.00)	0.99 (1.00)	56.7 (1.00)	6.81 (1.00)
2	86.3 (0.99)	82.5 (1.00)	0.33	3.69 (1.10)	0.22 (0.99)	1.52 (0.99)	0.97 (0.97)	57.7 (1.01)	6.94 (1.01)
3	86.3 (1.00)	82.5 (1.00)	0.34	3.70 (1.10)	0.22 (0.99)	1.52 (0.99)	0.94 (0.94)	57.1 (1.00)	6.91 (1.01)
4	86.2 (0.99)	82.5 (1.00)	0.52	3.87 (1.15)	0.22 (0.99)	1.52 (0.99)	0.97 (0.97)	56.8 (1.00)	6.81 (1.00)
5	86.3 (1.00)	82.5 (1.00)	316.00	319.00 (95.13)	0.22 (1.00)	1.53 (1.00)	0.99 (1.00)	56.7 (1.00)	6.81 (1.00)
Hairball (2850k triangles)									
1	1415.2 (1.00)	1057.1 (1.00)	0.00	48.75 (1.00)	1.01 (1.00)	7.30 (1.00)	5.55 (1.00)	180.1 (1.00)	43.70 (1.00)
2	1345.3 (0.95)	985.9 (0.93)	143.00	192.00 (3.93)	0.97 (0.96)	6.61 (0.90)	5.09 (0.91)	165.0 (0.91)	40.72 (0.93)
3	1345.7 (0.95)	986.0 (0.93)	131.00	180.00 (3.69)	0.99 (0.98)	6.82 (0.93)	5.01 (0.90)	165.1 (0.91)	40.89 (0.93)
4	1408.0 (0.99)	1052.2 (0.99)	10.09	58.85 (1.20)	1.05 (1.04)	8.99 (1.23)	5.54 (0.99)	179.8 (0.99)	43.70 (1.00)
5	1409.9 (0.99)	1052.8 (0.99)	2948.00	2996.00 (61.57)	1.05 (1.04)	8.76 (1.20)	5.73 (1.03)	185.0 (1.02)	45.07 (1.03)

4.2.2. *GPU ray tracer results*

The GPU ray tracing results were evaluated using a modified version of the Karras *et al.*'s [KAL09] GPU ray tracer. For measurements we have used a PC with Intel Core i7-2600K 3.40 GHz, NVIDIA GeForce GTX 580 3 GB GDDR5 and Windows 7 OS. The application was built using Microsoft Visual Studio 2010, version 64-bit, with CUDA Toolkit v3.2.

We have tested the performance of primary rays, random rays and ambient occlusion rays. Karras *et al.*'s primary rays and ambient occlusion rays generation and general tracing kernels were used

Table 4: *The results for individual objects where the base hierarchy for all methods is created with a spatial median top down driven build (method 1), which for methods 2–5 is followed by an update phase. These are as follows: 2—our method with random selection, 3—our method with the combined measure of inefficiency, 4—Kensler's method, hill climbing, and 5—Kensler's method, simulated annealing. The values in brackets are the computed ratios against reference values (1.00).*

M	Cost [-]	0 Cost [-]	CPU update [s]	CPU build [s]	CPU primary [s]	CPU random [s]	GPU primary [ms]	GPU random [ms]	GPU amb. occlusion [ms]
Happy Buddha (1087k triangles)									
1	275.6 (1.00)	269.1 (1.00)	0.00	1.89 (1.00)	0.34 (1.00)	4.83 (1.00)	1.64 (1.00)	278 (1.00)	12.37 (1.00)
2	166.4 (0.60)	158.7 (0.58)	37.82	39.71 (20.97)	0.21 (0.62)	2.78 (0.57)	0.97 (0.59)	167 (0.60)	6.59 (0.53)
3	168.9 (0.61)	161.4 (0.59)	43.62	45.51 (24.03)	0.21 (0.63)	2.83 (0.58)	1.08 (0.65)	170 (0.61)	6.83 (0.55)
4	217.5 (0.78)	211.4 (0.78)	7.88	9.77 (5.16)	0.27 (0.79)	3.79 (0.78)	1.34 (0.81)	220 (0.79)	9.07 (0.73)
5	179.4 (0.65)	171.8 (0.63)	1152.00	1154.00 (609.33)	0.24 (0.70)	3.14 (0.65)	1.14 (0.69)	181 (0.65)	7.18 (0.58)
Dragon (871k triangles)									
1	232.9 (1.00)	227.6 (1.00)	0.00	1.49 (1.00)	0.46 (1.00)	3.95 (1.00)	2.05 (1.00)	246 (1.00)	16.01 (1.00)
2	146.2 (0.62)	139.6 (0.61)	32.58	34.07 (22.85)	0.29 (0.62)	2.44 (0.61)	1.17 (0.57)	158 (0.63)	9.06 (0.56)
3	149.3 (0.64)	143.0 (0.62)	39.50	40.99 (27.49)	0.29 (0.63)	2.53 (0.64)	1.19 (0.58)	160 (0.65)	9.26 (0.57)
4	189.3 (0.81)	184.2 (0.80)	7.14	8.63 (5.79)	0.38 (0.82)	3.31 (0.83)	1.57 (0.76)	207 (0.83)	12.44 (0.77)
5	157.1 (0.67)	150.5 (0.66)	927.02	928.51 (622.84)	0.33 (0.70)	2.73 (0.69)	1.30 (0.63)	173 (0.70)	10.05 (0.62)
Blade (1765k triangles)									
1	345.9 (1.00)	337.6 (1.00)	0.00	3.03 (1.00)	0.40 (1.00)	4.53 (1.00)	2.02 (1.00)	21.5 (1.00)	7.95 (1.00)
2	196.3 (0.56)	185.2 (0.54)	82.97	86.00 (28.34)	0.24 (0.60)	2.45 (0.54)	1.10 (0.54)	11.4 (0.53)	4.22 (0.53)
3	202.4 (0.58)	191.5 (0.56)	100.46	103.49 (34.10)	0.25 (0.61)	2.52 (0.55)	1.22 (0.60)	11.8 (0.54)	4.41 (0.55)
4	272.6 (0.78)	264.6 (0.78)	11.64	14.67 (4.83)	0.32 (0.80)	3.55 (0.78)	1.59 (0.78)	15.6 (0.72)	5.97 (0.75)
5	214.3 (0.61)	203.3 (0.60)	1872.49	1875.52 (618.05)	0.29 (0.73)	2.87 (0.63)	1.45 (0.71)	13.3 (0.61)	4.66 (0.58)
Armadillo (307k triangles)									
1	143.5 (1.00)	140.73 (1.00)	0.00	0.47 (1.00)	0.40 (1.00)	2.63 (1.00)	1.88 (1.00)	95.8 (1.00)	12.23 (1.00)
2	89.7 (0.62)	85.87 (0.61)	6.21	6.69 (14.06)	0.25 (0.62)	1.64 (0.62)	1.03 (0.54)	60.1 (0.62)	7.24 (0.59)
3	91.4 (0.63)	87.82 (0.62)	7.06	7.54 (15.85)	0.25 (0.62)	1.67 (0.63)	1.10 (0.58)	61.5 (0.64)	7.53 (0.61)
4	116.5 (0.81)	113.67 (0.80)	1.68	2.15 (4.53)	0.32 (0.79)	2.18 (0.82)	1.48 (0.78)	77.5 (0.80)	9.53 (0.77)
5	95.2 (0.66)	91.39 (0.64)	321.23	321.71 (675.96)	0.28 (0.70)	1.80 (0.68)	1.13 (0.60)	65.3 (0.68)	7.78 (0.63)
Hairball (2850k triangles)									
1	2448 (1.00)	2114.84 (1.00)	0.00	5.24 (1.00)	2.04 (1.00)	17.00 (1.00)	15.23 (1.00)	476 (1.00)	110.03 (1.00)
2	1347 (0.55)	987.64 (0.46)	231.92	237.16 (45.22)	0.97 (0.47)	6.63 (0.39)	5.18 (0.34)	166 (0.34)	41.30 (0.37)
3	1362 (0.55)	1003.56 (0.47)	285.32	290.56 (55.40)	0.98 (0.48)	6.90 (0.41)	5.28 (0.35)	174 (0.36)	43.33 (0.39)
4	1892 (0.77)	1576.44 (0.74)	26.78	32.02 (6.10)	1.57 (0.76)	12.58 (0.74)	9.35 (0.62)	318 (0.66)	72.19 (0.65)
5	1473 (0.60)	1119.17 (0.52)	3053.79	3059.04 (583.31)	1.20 (0.58)	8.34 (0.50)	6.28 (0.42)	202 (0.42)	48.47 (0.43)

Table 5: *Summary results for our algorithm used for architectural scenes, where the reference method is SAH-based build method in top down fashion without optimization of the tree (100%). The data are compacted from Tables 3 and 4.*

Scene	Cost[-] SAH build	Cost, our optimized SAH build	Cost, spatial median	Cost, our optimized sp. median	Time[s], SAH build	Time, our optimized SAH build	Time, spatial median	Time, our optimized sp. median
Conference	130.30 (100%)	78.66%	646.38%	78.83%	3.45 (100%)	131.30%	12.75%	226.38%
Fairy Forest	95.10 (100%)	96.21%	194.52%	97.70%	1.96 (100%)	120.92%	12.24%	63.27%
Sibenik Cathedral	82.30 (100%)	84.45%	474.74%	86.60%	0.66 (100%)	209.09%	13.64%	175.76%
Sponza	220.20 (100%)	82.74%	571.25%	83.30%	0.56 (100%)	180.36%	12.50%	376.79%
Soda Hall	216.50 (100%)	76.40%	644.96%	76.61%	40.96 (100%)	137.60%	10.47%	187.26%
Power Plant, sec. 9	57.90 (100%)	89.46%	325.87%	90.33%	1.36 (100%)	119.12%	11.76%	205.15%
Power Plant, sec. 16	93.50 (100%)	85.35%	540.16%	85.24%	4.70 (100%)	159.57%	12.77%	378.51%
Power Plant	115.80 (100%)	84.46%	571.10%	85.16%	396.00 (100%)	121.46%	9.08%	71.90%
Pompeii Ten	252.90 (100%)	86.20%	303.24%	86.98%	102.00 (100%)	175.49%	10.76%	114.02%

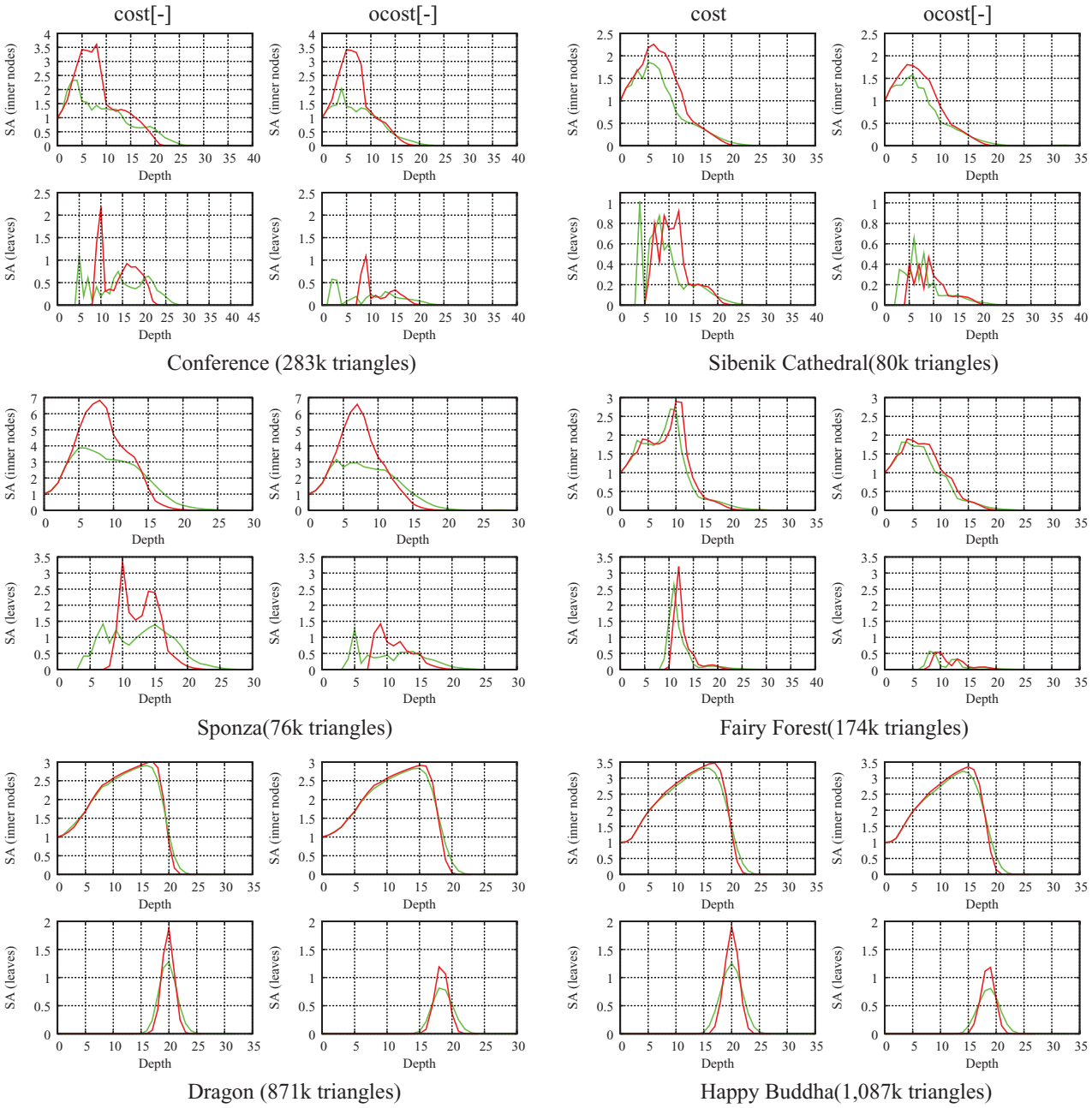


Figure 11: *Histograms showing the sums of surface areas of inner nodes and leaves at different depths of the tree. The figures in the odd columns correspond to a tree with leaves representing individual triangles* (cost *in Table 1–4*), *the figures in even columns to compacted trees with more triangles per leaf* (ocost *in Table 1–4*). *The odd rows show the sum of costs for interior nodes and the even rows the sum of costs for leaves in the dependence on the depth. The results for trees constructed by top down building algorithm with surface area heuristic are in red colour, the results for the trees after optimization by our algorithm are in green colour.*

without changes. We have implemented a random rays generation routine, casting 8 million rays for each scene, where the rays are defined by generating two uniformly distributed points in the scene bounding box. For the ambient occlusion test we spawn ten rays at each hit of the primary ray.

We can see that the GPU results correlate with the results from the CPU raytracer (see Tables 1 and 3). However, there are scenes, where the GPU version does not provide a performance gain comparable to the CPU implementation particularly for primary rays (e.g. Conference—CPU primary 69%, GPU primary 98%), though

in these cases the reference method of Kensler exhibits similar behaviour. On the contrary on a few tested scenes the optimized BVH provides slightly higher performance gain for the GPU ray tracer than for the CPU version (e.g. Sibenik - CPU primary 85%, GPU primary 81%). The analysis why this happens is a matter of future work when convenient profiling tools on a GPU will become available.

5. Conclusion and Future Work

We proposed an algorithm for building a high quality BVH by incremental updates of the BVH initially constructed by a top down method with SAH. The algorithm is based on performing selective updates of the BVH by identifying problematic nodes and reinserting them back in appropriate positions to minimize the total BVH cost. The updates are prioritized and the resulting method is highly flexible in terms of the update time with respect to the quality of the hierarchy.

We have shown that for complex scenes our method achieves very good cost reduction in much shorter time than previous methods. In fact the results indicate that the method constructs the best currently known BVHs under the SAH cost model and thus it has a potential to become a common optimization technique, which further reduces the cost of the SAH builders used in practical applications.

Currently, we work on a parallel version of the algorithm on modern GPUs in CUDA, where both the update and the rendering algorithms run solely on the GPU. We also want to study the properties of the hierarchy for other visibility computations such as occlusion and view-frustum culling for large scenes. Another possible future work is to study other types of BVH such as those with explicit spatial splits or with higher branching factor.

Acknowledgements

We would like to thank the contributors of the scenes used in our paper, Prof. C. Sequin for Soda Hall model, the University of North Carolina for the Power Plant model, Marko Dabrovic for the Sponza and Sibenik models, Ingo Wald for Fairy Forest, Greg Ward for the Conference model, Samuli Laine and Tero Karras for Hairball model and Stanford repository for other models.

Further, we want to thank Andrew Kensler for providing us with the source code for his paper [Ken08] that allowed to reproduce the reference method exactly. We would also like to thank Tero Karras, Timo Aila and Samuli Laine for releasing their GPU ray tracing framework. Our research was supported by the Czech Science Foundation under research programs P202/11/1883 (Argie) and P202/12/2413 (Opalis), and the Grant Agency of the Czech Technical University in Prague, grant No. SGS10/289/OHK3/3T/13, supported by Ministry of Education of the Czech Republic.

References

[AL09] Aila T., Laine S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High-Performance Graphics (HPG'09)* (Aug2009), pp. 145–149.

[DHK08] Dammertz H., Hanika J., Keller A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum 27*, 9 (June 2008), 1225–1233.

[EG07] Ernst M., Greiner G.: Early split clipping for bounding volume hierarchies early split clipping for bounding volume hierarchies. In *IEEE Symposium on Interactive Ray Tracing (RT'2007)* (September 2007), pp. 73–78.

[EGMM07] Eisemann M., Grosch T., Magnor M., Mueller S.: Automatic creation of object hierarchies for ray tracing dynamic scenes. In *WSCG'2007 Short Papers Post-Conference Proceedings* (Pilsen, Bohemia, Czech Republic, January 2007), V. Skala, (Ed.), WSCG, Pilsen, Bohemia, Czech Republic, pp. 57–64.

[GPM11] Garanzha K., Pantaleoni J., Mcallister D.: Simpler and faster HLBVH with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG'11)* (August 2011), pp. 59–64.

[GS87] Goldsmith J., Salmon J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (May 1987), 14–20.

[Hav00] Havran V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Nov 2000.

[HHS06] Havran V., Herzog R., Seidel H.-P.: On the fast construction of spatial data structures for ray tracing. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006* (September 2006), pp. 71–80.

[HMF07] Hunt W., Mark W. R., Fussell D.: Fast and lazy build of acceleration structures from scene hierarchies. In *IEEE/EG Symposium on Interactive Ray Tracing 2007* (September 2007), pp. 47–54.

[HSZ*11] Hou Q., Sun X., Zhou K., Lauterbach C., Manocha D.: Memory-scalable GPU spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics 17*, 4 (April 2011), 466–474.

[IWP07] Ize T., Wald I., Parker S. G.: Asynchronous BVH Construction for ray tracing dynamic scenes on parallel multi-core architectures. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization 2007 (EGPGV'07)* (May 2007), pp. 101–108.

[KAL09] Karras T., Aila T., Laine S.: Understanding the efficiency of ray traversal on GPUs; Google Code, 2009.

[Ken08] Kensler A.: Tree rotations for improving bounding volume hierarchies. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing* (August 2008), pp. 73–76.

[KK86] Kay T. L., Kajiya J. T.: Ray tracing complex scenes. In *SIGGRAPH '86 Proceedings* (New York, NY, USA, August 1986), D. C. Evans, R. J. Athay (Eds.), vol. 20, pp. 269–278.

[KMKY10] Kim T.-J., Moon B., Kim D., Yoon S.-E.: RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies. *IEEE Transactions on Visualization and Computer Graphics 16*, 2 (March–April 2010), 273–286.

[LGS*09] Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D.: Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384.

[LYTM06] Lauterbach C., Yoon S.-E., Tuft D., Manocha D.: RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *IEEE Symposium on Interactive Ray Tracing (RT'06)* (September 2006), pp. 39–46.

[MB90] Macdonald J. D., Booth K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 6 (1990), 153–165.

[ML03] Masso J. P. M., Lopez P. G.: Automatic hybrid hierarchy creation: A cost-model based approach. *Computer Graphics Forum 22*, 1 (2003), 5–13.

[MW06] Mahovsky J., Wyvill B.: Memory-conserving bounding volume hierarchies with coherent raytracing. *Computer Graphics Forum 25*, 10 (June 2006), 173–182.

[Omo89] Omohundro S. M.: *Five Balltree Construction Algorithms*. Tech. Rep. TR-89-063, International Computer Science Institute, Berkeley, 1989.

[PGDS09] Popov S., Georgiev I., Dimov R., Slusallek P.: Object partitioning considered harmful: space subdivision for BVHs. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)* (August 2009), pp. 15–22.

[PL10] Pantaleoni J., Luebke D.: HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics (HPG'10)* (June 2010), pp. 87–95.

[RTN] Ray Tracing News, internet publications, put together and edited by Eric Haines, since 1987. http://tog.acm.org/resources/RTNews/html. Accessed on 13 April 2012.

[RW80] Rubin S. M., Whitted T.: A 3-Dimensional Representation for Fast Rendering of Complex Scenes. In *SIGGRAPH '80 Proceedings* (July 1980), vol. 14, pp. 110–116.

[SFD09] Stich M., Friedrich H., Dietrich A.: Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG'09)* (August 2009), pp. 7–13.

[Smi98] Smits B.: Efficiency issues for ray tracing. *Journal of Graphics Tools 3*, 2 (1998), 1–14.

[Wal07] Wald I.: On fast Construction of SAH based bounding volume hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing* (September 2007), pp. 33–40.

[Wal12] Wald I.: Fast construction of SAH BVHs on the Intel Many Integrated Core (MIC) architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 1 (Jan 2012), 47–57.

[WBKP08] Walter B., Bala K., Kulkarni M., Pingali K.: Fast agglomerative clustering for rendering. In *IEEE Symposium on Interactive Ray Tracing (RT'2008)* (August 2008), pp. 81–86.

[WBS07] Wald I., Boulos S., Shirley P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (Jan 2007), DOI 10.1145/1189762.1206075.

[WHG84] Weghorst H., Hooper G., Greenberg D. P.: Improved computational methods for ray tracing. *ACM Transactions on Graphics 3*, 1 (January 1984), 52–69.

[WIP08] Wald I., Ize T., Parker S. G.: Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Computers and Graphics (Special Section: Parallel Graphics and Visualization) 32*, 1 (2008), 3–13.

[WK06] Wächter C., Keller A.: Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the 17th Eurographics Symposium On Rendering (EGSR'06)* (June 2006), pp. 139–149.

[WMS06] Woop S., Marmitt G., Slusallek P.: B-KD trees for hardware accelerated ray tracing of dynamic scenes. In *Proceedings of Conference on Graphics Hardware 2006* (September 2006), pp. 67–77.

[YM06] Yoon S.-E., Manocha D.: Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum 25*, (June 2006), 507–516.

Appendix F

Massively Parallel Hierarchical Scene Processing with Applications in Rendering

Vinkler, M. - Bittner, J. - Havran, V.: Massively Parallel Hierarchical Scene Sorting with Applications in Rendering. Computer Graphics Forum, accepted for publication on Aprill 11th 2013, to appear. **IF=1.636**

Volume 0 (1981), Number 0 pp. 1–13 COMPUTER GRAPHICS *forum*

Massively Parallel Hierarchical Scene Processing with Applications in Rendering

Marek Vinkler[1] Jiří Bittner[2] Vlastimil Havran[2] Michal Hapala[2]

[1]Masaryk University, Brno
[2]Faculty of Electrical Engineering, Czech Technical University in Prague

Abstract

We present a novel method for massively parallel hierarchical scene processing on the GPU, which is based on sequential decomposition of the given hierarchical algorithm into small functional blocks. The computation is fully managed by the GPU using a specialized task pool which facilitates synchronization and communication of processing units. We present two applications of the proposed approach: construction of the bounding volume hierarchies and collision detection based on divide-and-conquer ray tracing. The results indicate that using our approach we achieve high utilization of the GPU even for complex hierarchical problems which pose a challenge for massive parallelization.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—[Graphics data structures and data types] I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—[Ray tracing]

1. Introduction

Hierarchical algorithms and data structures are powerful tools for efficient processing of computationally intense problems. Hierarchical data structures such as bounding volume hierarchies or kd-trees have become standard methods for rendering acceleration particularly when targeting ray tracing based techniques. Apart from the established methods based on spatial hierarchies, some new techniques such as the divide-and-conquer ray tracing [WK09,Mor11,Áfr12] work with an implicit hierarchy stored in a simple index array. Such methods may become an interesting alternative for ray tracing highly dynamic scenes.

While the hierarchical techniques have their provable benefits in terms of algorithmic efficiency, the general drawback is their difficult mapping to the massively parallel computational model of the GPU. While a number of clever solutions for this mapping have already been designed, most of the proposed techniques rely on management of the computation from the CPU side, invoking specialized computational kernels at different stages of the computation. This is due to the fact that different computational stages of the hierarchical techniques exhibit different levels of parallelism and it is not easy to reflect this using the currently available frameworks for GPU computation such as CUDA or OpenCL. Therefore, the GPU may get underutilized if for any kernel there is not enough work for each processing unit. As a result the scalability of the CPU managed method might be reduced when targeting massively parallel systems with tens of thousands of processing units, which are likely to become available in the future.

In this paper we propose an innovative method which moves the whole computation to the GPU and it requires no management from the CPU side. The method handles all important aspects of hierarchical techniques as it is able to perform complex evaluation of the given task, spawn new tasks, and handle the dependencies among the tasks. We provide two applications that justify the concept for our method: Bounding Volume Hierarchy (BVH) construction and divide-and-conquer ray tracing. The results indicate that the implementations based on our method perform comparable or even superior to existing solutions.

2. Related Work

We review here in short the relevant background knowledge in GPU algorithms and spatial sorting with focus on the building of hierarchical data structures on GPUs.

GPUs and Load Balancing. Load balancing and scheduling for GPU architectures is an active research area which relates to the method proposed in our paper. Tsigas and Zhang [TZ01] proposed a simple non-blocking concurrent queue for FIFO processing for shared memory multiprocessor systems that utilized compare-and-swap operations (CAS). With the availability of atomic operations on GPUs Cederman and Tsigas [CT08] compared four approaches for dynamic load balancing on GPUs and concluded that blocking queues perform the worst. Tzang et al. [TPO10] studied efficiency of load balancing methods for irregular workloads on the GPU and they concluded that task-stealing and task-donation are the most efficient. Chen et al. [CVKG10] proposed a task-based dynamic load balancing approach for single and multi GPU computer systems. They used a persistent kernel running on a *device(s)* (a GPU or several GPUs) where the task queue is generated on a *host* (CPU). The recent work by Sundell et al. [SGPT11] proposed a lock-free algorithm for distributing work on concurrent hardware without the restriction of work producers and consumers. Independently of our work Steinberger et al. [SKK*12] designed a flexible GPU framework, which also builds on the idea of persistent threads. Compared to their work our method is more specific to hierarchical scene processing and it provides dependencies among the tasks and better data level parallelism (more units can cooperate on solving the same task). We also want to point out a recent paper by Lee et al. [Lee10] who rigorously analyzed the performance of an NVIDIA GTX280 and an Intel Core i7 960 processor for fourteen different computational problems with carefully optimized implementations. They showed that the GPU-CPU performance gap narrows from the mythical 10-100 times to only 2.5 times on average.

GPU Rendering and Hierarchical Data Structures. There has been number of approaches dealing with the hierarchical data structures used in computer graphics for ray tracing, general visibility computations such as occlusion culling, collision detection etc. We focus our discussion on the bounding volume hierarchies (BVH) and kd-trees with the stress on the algorithms implemented on the GPU. In particular we focus not only on those that efficiently utilize the GPU for performing computations with the help of these data structures, but also on those that use the GPU for actually building these data structures. The first technique that used the GPU for ray tracing was proposed by Purcell et al. [PBMH02] who utilized a shading language and remapped a uniform grid into textures. This approach was followed by other methods which are surveyed by Wald et al. [WMG*09]. However, the data structures were typically prepared on the CPU and the memory footprint was transferred to the GPU to allow for parallel traversal operations. The building of data structures on the GPU have become possible with the introduction of CUDA [NBGS08] and OpenCL [SGS10].

Kd-trees. Zhou et al. [ZHWG08] presented an algorithm to build kd-trees on the GPU, restricting the approach to a spatial median and cutting off empty space. This approach was extended by Hou et al. [HSZ*11] using partial breadth-first-search to afford for limited memory consumption. Danilewski et al. [DPS10] presented a scalable GPU algorithm with binning for kd-trees that improves on the quality of constructed kd-trees following the method of Shevtsov et al. [SSK07]. Wu et al. [WZL11] proposed an algorithm running on the GPU as a sequence of kernels that construct kd-trees in a breadth-first search manner, but for all boundary positions in the fashion of the serial approach by Wald and Havran [WH06]. This algorithm was also parallelized for multi-core CPUs by Choi et al. [CKL*10].

Bounding Volume Hierarchies. Lauterbach et al. [LGS*08] presented an algorithm to build the Linear BVH (LBVH) using Morton codes, where the speed is moderately penalized by the quality of the built BVH. Aila and Laine [AL09] studied different possibilities to organize the traversal code on GPU architectures to get the highest performance. Pantaleoni and Luebke [PL10] presented a more efficient version of the LBVH algorithm with Morton codes and compress-sort-decompress strategy, together with improved memory management. They call it the Hierarchical LBVH (HLBVH). Further, they presented a hybrid algorithm with a two-level BVH, where top levels are built with an exact algorithm with a surface area heuristics (SAH) [Wal07] and bottom levels with a Morton curve based algorithm. Garanzha et al. [GPM11] simplified the HLBVH algorithm using binary search and work queues. They achieved both memory savings and lower build times than the paper by Pantaleoni and Luebke [PL10]. Wald described a parallel version of a BVH based builder with SAH using binning on a many-core architecture (MIC) [Wal12]. Sopin et al. [SBU11] studied binned SAH BVH construction on the GPU with focus on efficient division of data between computational units.

Grids. Kalojanov and Slusallek [KS09] presented a parallel algorithm for building uniform grids, followed by another paper by Kalojanov et al. [KBS11] for hierarchical grids.

Implicit Hierarchies. Wächter and Keller [WK09] presented an approach for ray tracing which simultaneously subdivides rays and triangles and can be computed without explicit spatial data structures. A similar approach was independently developed and implemented on a single-core CPU with SSE instructions by Mora [Mor11]. Mora also utilized bounding cones for primary rays to improve the performance.

3. Hierarchical Scene Processing

In this section we first present the terminology and an overview of our algorithm and then propose a novel general methodology of mapping a hierarchical algorithm to the GPU framework. We limit our discussion to CUDA [NBGS08] based implementations and use terminology and constants associated with the currently available CUDA platforms.

3.1. Terminology

Prior to introducing the algorithm we briefly define the basic terms used in the paper.

- *Task* is a computational job which is associated with the given range of scene data (geometry, ray queries, etc. stored in the linear array in contiguous block of memory). The whole computation is initiated using a single task associated with the whole scene. After a task is processed, it is either finished or spawns one or more child tasks. Every child task processes the associated range of data. The task is characterized by its state.
- *Phase* is a logical algorithmic block of the task, such as finding the splitting plane, sorting triangles, computing a tight bounding box, etc.
- *Step* is an algorithmic block of the phase. A phase might consist of a single step, but some phases need more steps. The number of required steps may depend on the size of the given data range. If the phase consists of more steps, the results of one step are processed by the further steps in order to compute an aggregated result of the whole phase. An example when more steps are needed is a parallel reduction computation used for computing the new AABB of child nodes, which requires a logarithmic number of steps.
- *Work chunk* is a data range associated with a particular step processed by a single warp. The work chunk is the smallest unit of work in our method. Note that while the phase and the step represent a subdivision into smaller algorithmic blocks (i.e. in the time domain), the work chunk represents a subdivision of the data associated with the step (i.e. in the contiguous block in memory address space). The work chunk consists of 32 data items and each thread in a warp processes a single item.
- *Task pool* is a data structure used for managing the execution of tasks. In our method the task pool is not working as a queue nor stack. This is implied by required computational efficiency as well as computational dependencies among the tasks. The details on the task pool will be given in section 3.4.

Apart from the above defined terms we recall the basic terminology associated with CUDA: *kernel* is a program executed on the CUDA device and *warp* is a group of 32 threads, which execute the same instruction at a time.

3.2. Algorithm Overview

Our algorithm follows the divide-and-conquer principle of hierarchical methods: when the current task is too large to be solved directly it is further subdivided until it is small enough to be terminated or solved in a trivial way.

Each task holds the information about its data range (e.g. interval in the triangle index array) and the state of the task. The task also holds information describing the current phase, the current step, the number of available work chunks, and auxiliary information such as the bounding box of the given geometry data.

A typical task dealing with 3D primitives can be divided into two major phases which are processed sequentially: determining a quicksort-like pivot for one phase of sorting (e.g. a splitting plane) and sorting the primitives according to this pivot into two parts using the index array. Note that this sorting can take place more times to create multiple subsets. After these phases the algorithm continues with subsets of these primitives in a given number of branches. The algorithm can also contain other phases, which evaluate data needed for further invocation of the algorithm such as bounding boxes. An example of the computational phases for the BVH construction is illustrated in Figure 1.

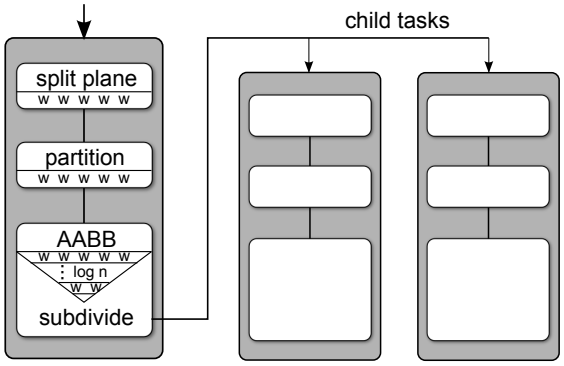


Figure 1: *Overview of the task and its phases in an algorithm for BVH construction. The figure also shows the relative number of warps cooperating on solving the particular steps of the task phases.*

Our algorithm is built on the concept of *persistent warps* [AL09]. Using persistent warps instead of threads or blocks has several advantages. First, it is easier to manage memory access coherency and branching as they are resolved on this level in hardware. Second, there are several horizontal warp-level functions that can accelerate the processing.

We aim at maximizing the parallelism of the computation on two different levels. First, we aim to process a given step of the computation using as many threads as possible (fine grain spatial parallelism), i.e. the number of threads working

on the given step corresponds to the size of the data range associated with the step. Second, we aim to compute different tasks in different computational phases in parallel (coarse grain temporal parallelism). For example, we want to determine a splitting plane for one node in the hierarchy using a number of warps and at the same time we perform sorting of the triangles in some other node using the remaining warps.

For some algorithmic problems it is possible that several tasks may need to work on the shared data range. For example when constructing a kd-tree, the subsets of triangles associated with the left and right children of a node generally overlap and the data ranges of the associated child tasks overlap as well. In such a case it is necessary to enforce an order on the task execution, and we mark some tasks as dependent on other active tasks. These dependent tasks must wait to be activated upon the completion of active tasks. A dependent task holds the counter on how many tasks have to finish before it is activated. An active task contains pointers (indices) to tasks it is responsible for activating.

3.3. Managing the parallel computation

We launch a single kernel with as many persistent warps as can be run simultaneously on the GPU. After launching this kernel there is no further management from the CPU and the work flow takes place completely on the GPU. The crucial component in our system is the *task pool* stored in the global memory: all warps are synchronized and take their work from the task pool. The task pool holds all the information about the current state of the computation.

When the kernel is launched the task pool is filled with a single task. This task encapsulates all the geometry (e.g. triangles). When this task is finished it can spawn its child task(s) until the whole task pool is empty, signalling that the computation is done.

Since warps are independent in CUDA, each warp can process a different task with a different state. However, in our method warps also participate in computation of the same task. This is in contrast with the previous approaches where communication and synchronization between warps was either limited or not possible at all. Each warp takes work chunks from an arbitrary active task based on the current distribution of work in the task pool. The pseudocode of the method is shown in Algorithm 1.

The pseudocode shows that the warps are constantly searching for arbitrary work chunks that they can handle. When they succeed in retrieving the work chunks they perform the work according to that particular task and its phase and step. The overview of the main data structures used in our method is depicted in Figure 2.

3.4. Task Pool

The task pool is represented by two arrays, one for holding all the information necessary for computing the task (*task data array*) and the second compact one for defining the amount of work to be done in the current step (*task header array*). Because of this decomposition the task header array contains a single integer for each task. This gives a very small memory footprint that can fit easily into the cache on modern GPU architectures.

```
In serial: Insert the first task into the task pool;
In parallel: while Task pool is not empty do
    if retrieve(taskIdx, work chunks) was successful
    then
        read task from task data array;
        switch task phase do
            repeat
                process_task (step, work chunk);
            until no more work chunks;
            memory fence;
            advance the state or finish the task;
        endsw
    end
end
Computation is done;
```

Algorithm 1: Main loop of the algorithm.

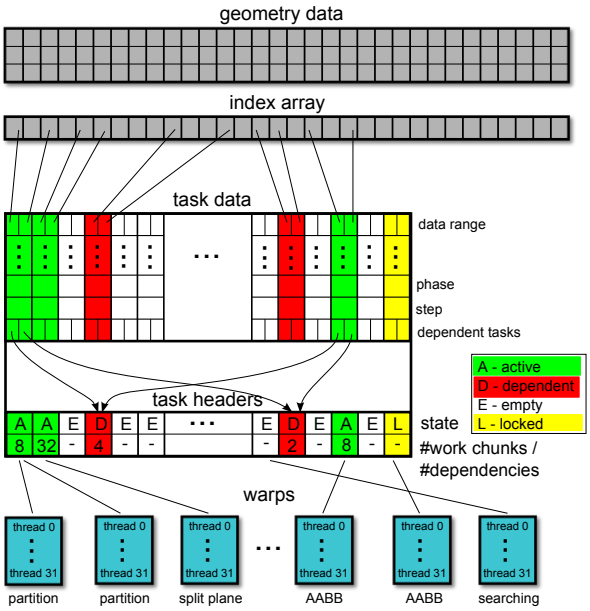


Figure 2: *Overview of the main data structures used in our method.*

For each task the header array encodes the task state in an integer value. Apart from the task state we also encode additional quantitative information in this integer value, the meaning of which depends on the task state. This additional information allows us to use efficient mechanism to retrieve work chunks and handle task dependencies. The task can ex-

hibit one of the following four states in its header by the integer value I:

- $I > 0$: *Active* state. The task is ready to be processed and there are I work chunks to be done on this task for the current phase and step. Note that the phase and the step is stored in the task data array. Below we call a task in active state an active task.
- $I < 0$: *Dependent* state. The task waits for $-I$ other tasks to finish before it is activated.
- $I = 0$: *Locked* state. The task is locked, which means that its data entry is just being created or modified.
- $I <= -BIG\_INT$: *Empty*. This entry in the task pool is currently not used and it can be populated with a new task.

Retrieving work. When warps are trying to find a work chunk to process they loop through the task header array searching for an active task. To promote parallelism each warp starts at a different index in the pool, based on its warp ID and the number of entries in the pool. Each thread in the warp then reads the state of one consecutive entry from the task header array. As multiple entries can be active, it has to choose one to take work from. To prevent all warps from choosing the same active task introducing conflicts of atomic operations, we compute the prefix sum on the states of the entries within each warp and choose the i-th active task, where i is based on the warp ID. When the active tasks are chosen, the warps atomically decrement the tasks' header. Each warp may decrement the value by any number i.e. retrieve as many work chunks from a single entry as it desires. It is often beneficial to retrieve multiple work chunks in one atomic operation because the overhead of retrieving the work chunk is not negligible. We use the following function for determining the number of work chunks retrieved by the warp:

$$N_w = max(\frac{S}{W} + 1, K), \quad (1)$$
$$S = \sum_{i=1}^{32} N_i,$$

where N_i is the number of work chunks corresponding to the task entry sampled by thread i of the warp; W is the number of warps launched on the GPU, and K is a constant preventing the retrieval of too few work chunks. Note that N_i represents the number of work chunks the current task step was created with and $N_i = 0$ for all inactive tasks.

The first term $\frac{S}{W} + 1$ aims to distribute the available work among other warps, while the constant K prevents the fragmentation of work and in turn it bounds the overhead connected with the task pool management, especially in the later stages of the computation when the processed tasks consist of smaller amount of data. Note that $K = 14$ was experimentally verified to be a reasonable choice in practice for contemporary GPU architectures.

It may happen that the value of the entry is decremented below the value representing the *Lock* state by multiple warps concurrently trying to retrieve work from the same task. This is not a problem as long as the counter is not decremented to the value representing an empty task. If the warp did not succeed in retrieving the work chunks the value returned by the atomic decrement is not positive. In that case the same process is repeated on a different task.

Finishing work. When warp finishes the retrieved work it has to communicate this fact to the other warps. In particular the last finished warp has to be aware that it is responsible for advancing the task state or issuing new tasks. To accomplish this we use another counter of unfinished work chunks stored in the task's data. This counter is atomically decremented by each finished warp. The warp that decrements it to zero is the last finished warp. This warp can then interpret the results of the step and progress the computation further to the next step or phase for the given task. We cannot use the value obtained from the task's header for this purpose since the warp that last retrieved work from the task need not be the warp that finishes it last.

Storing work. In order to create a new task the warp loops through the task header array searching for an empty entry. When it finds one it atomically compares-and-swaps its value with the value representing a lock. If it succeeds, it fills the corresponding entry in task data array with the child task. As the last step, it sets the header array entry with the number of work chunks required to process the first step of the first phase of the child task to unlock it, or it sets the entry with the number of tasks this task is dependent on to mark a dependent task. Note that a memory fence operation must be issued before the task is unlocked to make sure valid data are visible to other warps.

Handling dependent tasks. Working with dependent tasks is straightforward in our framework. Since their task header value is less than the *Lock* state they are ignored by the warps during the retrieving or storing of work. The active tasks that point to this dependent task increase the dependent task's header value upon their finish, eventually increasing it to the *Lock* state. This signals all dependencies are resolved and the header value can be overwritten by the number of work chunks in the task, signalling the *Active* state.

Minimizing pool overhead. We use two improvements that accelerate the computation of tasks. They are both targeting steps with little parallelism. First, when some phase requires zero work chunks to compute it is immediately skipped. Second, if some step requires less than K work chunks, this step is processed immediately by the given warp without writing the task into the pool (K is the constant used in Eq. 1). This is often the case with the reduction in the *AABB* phase.

3.5. Comparison to Standard Kernel Launching

Performing the entire computation and management on the GPU has several advantages compared to the standard method based on serial kernel launching and synchronization. First, the intermediate results need not be saved to global memory (on a GPU) between consecutive kernel launches or transferred over an even slower PCI-E bus to the main memory. Our approach is in an agreement with the GPU evolution which places more computation on the GPU side to limit the communication. Moreover, during these data transfers and kernel launch preparations the GPU is idle (if there is no concurrently running kernel). Also the kernel launch is a high overhead operation as stated by several authors [ZHWG08, GPM11].

Managing the computation on the GPU has other advantages besides limiting overhead. While the available parallelism is fixed for the kernel launching approach e.g. to a single level of the hierarchy (spatial parallelism), in our approach nodes from different levels can be processed simultaneously (temporal parallelism). This increases the available parallelism and limits the computation stalls due to underutilized GPU. For the kernel launching these stalls often happen when processing top levels of hierarchies where there is not enough data to process or when some warps have already finished their work and are waiting for other warps to terminate the kernel.

In the rest of the paper we discuss two applications of the proposed framework for parallelization of hierarchical algorithms: BVH construction and divide-and-conquer ray tracing.

4. Constructing Bounding Volume Hierarchies

Bounding volume hierarchies are common data structures used for rendering acceleration. They became particularly popular for ray tracing acceleration of dynamic scenes since they are relatively fast to construct and update, and have predictable memory footprint.

The algorithm for constructing a BVH can be easily mapped to our parallel framework as we describe in the next sections. The ease of mapping the BVH build comes mainly from the fact that each task is completely independent of other tasks. For the rest of the paper we assume that the scene consists of triangles although the method can generally handle other scene primitives as well.

4.1. Defining Phases and Steps

The computation starts with a single task associated with all scene triangles. Each task then needs to subdivide the given set of triangles into two disjoint subsets (assuming a binary hierarchy). The formation of these subsets is typically based on spatial criteria such as the spatial median or the more involved surface area heuristics (SAH). The subdivision can be easily implemented by sorting the triangle indices into two disjoint groups in the index array. If the given triangle subset is large enough, a new task is created. Otherwise, the current branch of the computation is terminated and a leaf is created.

For each task we define three different phases:

1. *SplitPlane*: Splitting plane computation (spatial median or cost model with SAH).
2. *Partition*: Partitioning of a triangle range into the left and right sub-ranges in the double buffered index array.
3. *AABB*: Computation of the two bounding boxes for the child tasks.

Note that some of these phases represent parallel divide-and-conquer algorithms on their own (*AABB*) and, therefore, require a logarithmic number of steps to complete. The illustration of the phases is shown in Figure 1. Note that the figure also shows the relative number of work chunks required by different steps of the tasks (indicated as w). The number of work chunks per step corresponds to the number of warps which perform the work according to Eq. 1. Below we describe the particular phases of the algorithm in more detail.

SplitPlane. Currently we support two splitting strategies: the spatial median and the SAH. The spatial median cycles the splitting plane in the round-robin fashion, where for the first task the longest axis is chosen. The SAH chooses the best plane from equal number of candidates in each axis, where the evaluation of the SAH cost is similar to [HHS06]. For the SAH strategy we select 32 candidate planes and evaluate their cost using the SAH in parallel. Each warp processes a distinct sub-range of a task's triangles from the index array and each thread computes their position with respect to one of the candidate planes. Then the number of triangles to the left and to the right of the splitting plane, as well as the bounding boxes are atomically updated in global memory. The warp that has finished its work last loads these data from the global memory and chooses the best splitting plane. As there are exactly 32 (warp size) candidates this is done in parallel as well. For the spatial median strategy the splitting plane is evaluated directly when the task is enqueued in the pool and this phase is skipped.

Partition. In this phase the triangles are divided into the left and right subsets based on the position of their centroid to the splitting plane. The method reads 32 consecutive triangles from the input index array and appends left triangles to the start of the output range and prepends right triangles to the end of the output range. Since the order of triangles in the left and right subsets is not important they can be written in arbitrary order, in our case the write offset is computed by *atomicAdd* to the start of the range and *atomicSub* to the end of the range. This atomic operation is done by a single thread in the warp and the returned value is used by all threads of the warp to compute their write offset using prefix scan. To prevent overwriting the input range, a new output array has to be used. We are using two triangle index arrays with each

task holding a pointer to either of the two arrays with the valid data.

AABB. Segmented parallel reduction is computed on the range in the triangle index array. The bounding boxes for the triangle ranges corresponding to left and right triangle subsets are computed using double-buffered array for storing the reduction tree. The computation requires $\log_2(\#tris)$ steps [HSO07]. This phase is only needed for the median splitting as the SAH evaluation already gives us the bounding boxes.

FullSAH. When the number of triangles in a task drops bellow the warp size it is possible to process the task more efficiently. In such a case we move it to a distinct phase that builds its subtree in one step using a single warp. The subtree is built using a full SAH computation that requires triangle sorting in all three axes. To make this operation efficient all the data are stored in registers and shared between the threads of a warp using the shuffle instruction introduced in the Kepler generation of NVIDIA GPUs.

4.2. Handling Tasks

Since the algorithm subdivides the current data range into disjoint subsets there are no data dependencies among different tasks and the tasks can be processed fully in parallel.

The two child tasks are created by the last phase, more precisely at the last step of the *AABB* phase for the spatial median splitting or *Partition* phase for the SAH based splitting. The algorithm first checks whether the termination criteria are met for the given subset of triangles. If this is the case (the number of triangles is below a threshold, a maximum depth is reached or the SAH termination takes place), a BVH leaf is created. Otherwise, a new task is stored to the task pool and it is initiated to the *SplitPlane* phase. When creating new tasks the method reuses the task pool entry for the current task and then it searches for an empty spot in the task pool to allocate the other child task.

5. Divide-And-Conquer Ray Tracing

In this section we describe the application of our method to the parallelization of the divide-and-conquer ray tracing algorithm proposed by Mora [Mor11], Keller and Wächter [WK09] and Áfra [Áfr12]. We first present a brief overview of this method and describe the phases and steps needed to cover the method in our framework.

5.1. Algorithm overview

The divide-and-conquer ray tracing is based on an idea of avoiding the explicit construction of a spatial data structure. Instead, the method performs a hierarchical computation in which an implicit spatial subdivision is used and maintained in an array of indices for both triangles and rays.

The method starts with all scene triangles and the set of rays to be cast. Then it picks up a splitting plane which subdivides the current bounding box into two smaller boxes. The method then sorts the triangle and ray arrays and recursively evaluates the triangles and rays intersecting one of the smaller boxes. When the recursion returns it resorts the rays and triangles to obtain those that intersect the other bounding box and performs recursion. The recursion is terminated if the number of rays or triangles is below a specified threshold. Then the intersections of rays and triangles are computed using a naive algorithm, computing the intersection among all pairs of rays and triangles.

While the recursive formulation of this method is simple, its parallelization is rather involved. The main problem is that the sets of rays and triangles intersecting the bounding boxes of the implicit spatial subdivision can overlap. We cannot evaluate all the child tasks in parallel using a single array of indices since different tasks would compete for sorting the ray and triangle ranges and storing the results. Therefore, we need to establish dependencies for the computation and handle them appropriately in the parallel version of the algorithm.

When a splitting plane is selected for the given bounding box all rays and triangles associated with the given task are classified as either lying left, right, or straddling the splitting plane. We aim to create child tasks which would cover all sub-ranges at which an intersection of rays and triangles can happen. As we have three ranges for both rays and triangles we obtain nine pairs of different ray/triangle ranges to process. Out of the nine pairs for two pairs of ranges no intersection can happen: (1) triangles lying left of the splitting plane and rays lying right of the plane and (2) triangles lying right of the plane and rays lying left of the plane. For the remaining seven range pairs we create child tasks and proceed with the computation. The subdivision into child tasks is illustrated in Figure 3.

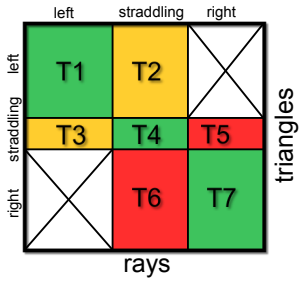


Figure 3: *Matrix representing a subdivision of the task into its child tasks for the divide-and-conquer ray tracing.*

There are clear computation dependencies among the child tasks shown in Figure 3: the tasks cannot be executed

simultaneously if they share some triangle or ray data (they are in the same row or column). There are several ways to execute and synchronize the tasks in order to avoid different tasks competing for the access to the same data. For example, the execution of the tasks can proceed as follows. We first activate three independent tasks T1, T4, and T7. Tasks T2 and T3 wait for execution as they depend on finalizing T1 and T4. Task T5 depends on T3 and T7 and task T6 depends on T2 and T7. More details about the task dependencies will be discussed in Section 5.3.

5.2. Defining Phases and Steps

For the divide-and-conquer ray tracing there are two types of tasks that can be created in the task pool: the intersection tasks and the subdivision tasks. The intersection task consists of one step with a number of work chunks which are set in a way that each warp processes 32 ray-triangle intersections in parallel (one intersection per thread). The closest intersection for each ray, if any, is then written to the global memory. Note that our implementation does not explicitly identify the type of the task. Instead for the intersection task we initiate it into a phase which implies a different task type (intersection phase).

The subdivision task is more complex. It consists of four phases that are computed sequentially (see Figure 4). Some of these phases are only a minor modification of the phases described for the BVH construction in Section 4.1. Since the rays are divided into four groups: left, straddling, right and clipped and triangles into three groups: left, straddling and right, the partition phase is executed twice, each time with a different pivot. Below we describe these phases in more detail.

SplitPlane. Again we support two splitting strategies: spatial median and cost model based splitting. A different cost model than SAH is used which is explained in this paragraph. Instead of using the SAH or a spatial median as proposed by Mora we use the Ray Distribution Heuristic (RDH) cost model by Bittner and Havran [BH09] which also takes the distribution of rays into account and achieves higher performance. The termination criteria are derived using this cost model; the intersection task is created when

$$\#tris \cdot \#rays \cdot C_{INTERS} < (\#tris + \#rays) \cdot C_{SORT}, \qquad (2)$$

where C_{INTERS} is the expected cost for one ray-triangle intersection and C_{SORT} is the expected cost for one sorting operation. We use 32 candidate planes that cover all three axes. The number of candidates is the same in each axis and the candidate positions are uniformly distributed. We do not use all triangles and rays associated with the given task for the evaluation of cost, but only their smaller subsets. The number of triangle samples N_T and ray samples N_R are computed as: $N_T = \sqrt{\#tris}$, $N_R = \sqrt{\#rays}$. The median splitting strategy is the same as for the BVH construction. Either strategy, this phase needs only one step.

Partition1. The partition is computed in parallel on both ray and triangle index arrays and runs in a single step. During the ray classification the rays that do not hit the bounding box of the current node are marked as clipped. These rays are treated as lying to the right of the splitting plane in this phase. Other than that this phase is exactly the same as the *Partition* phase in building BVH. As the result of this operation the rays are divided into two groups: left+straddling versus right+clipped and triangles are divided into left+straddling versus right. Also the sizes of the groups are known afterwards.

Partition2. This phase is done the same way as *Partition1* but for two ranges (the left+straddling and right[+clipped]) in parallel. After this phase the task's ray range is fully sorted into left, straddling, right, and clipped rays and triangle range into left, straddling, and right ranges with respect to the selected splitting plane.

AABB. This phase works similarly as for the BVH construction and requires $\log_2(\#tris)$ steps. The only difference is that we have to compute three new bounding boxes instead of two, since we compute the bounding box of the triangles straddling the splitting plane (tasks T2, T4, and T6).

5.3. Handling Tasks

As mentioned in Section 5.1 the child tasks resulting from the subdivision of a given task have certain dependencies and cannot be processed fully independently. Certain groups of child tasks are, however, independent. For the divide-and-conquer ray tracing we can formalize the dependencies among the tasks in a way that each task is responsible for activating at most two other dependent tasks. Initially we mark three child tasks as active and the remaining four tasks wait for being activated. Note that some of the child tasks need to inherit the activation pointers of the task being subdivided, as some other tasks may depend on it.

We propose to use the following subdivision into three independent task groups: (T1, T4, T7), (T2, T3), (T5, T6), which implies the following dependencies (TX→TY: TX activates TY, i.e. TY depends on TX):

- T1→T2, T1→T3,
- T4→T2, T4→T3,
- T7→T5, T7→T6,
- T3→T5,
- T2→T6,
- T5→P1, T5→P2,
- T6→P1, T6→P2,

where P1 and P2 are the dependencies inherited from the parent task. A task TY which should be activated by task TX has to be inserted first into the task pool. This is due to the fact that the task TX needs to know the index of the entry for the task TY in the task pool.

When an active task finishes, it updates the task header

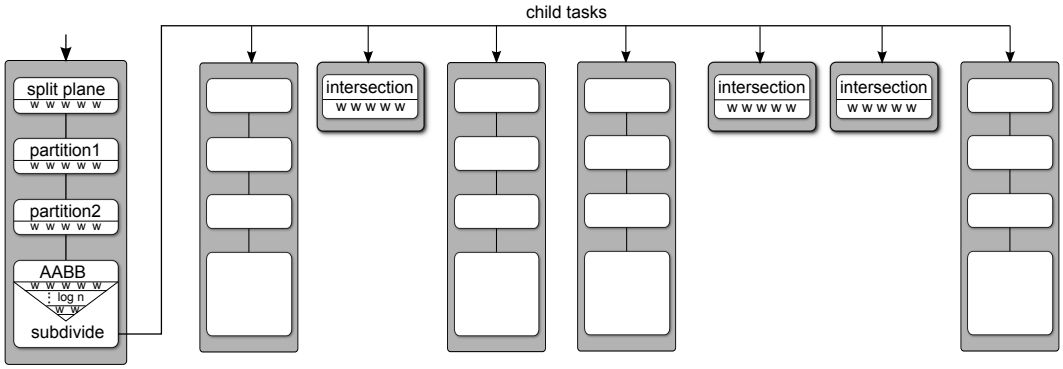


Figure 4: *Overview of the task phases and steps for the divide-and-conquer ray tracing algorithm.*

array for the dependent tasks. If this was the last dependence for that task the active task activates the dependent task by setting its entry in the header array to the number of its unfinished work chunks.

Often some of the child tasks T1-7 contain no rays or triangles in which case it is useless to add them into the pool. To prevent this the division of tasks into groups is defined by a look-up table. This table is queried when a parent task is divided into its child tasks. The index into the table is a binary array flag describing which ranges (left, right, straddling) are empty and which are nonempty. The table contains the number of child tasks to generate, number of child tasks in the last dependency group, the order in which the child tasks should be added into the task pool, the dependencies among the child tasks and the activity flag for each child task.

6. Results

We have implemented the proposed framework in C++ and CUDA [NBGS08]. For testing we have used a PC with Intel Core i7-2600, 16GB of RAM and NVIDIA GeForce GTX 680 running on Windows 7 64-bit. We have used two types of scenes for testing, individual objects and more complex architectural scenes. The images for the test scenes are shown in Figure 5.

6.1. Constructing BVHs

First, we tested the time for building BVHs using the parallel algorithm described in Section 4 and the traversal performance of these BVHs. We used three different ray distributions: primary rays, incoherent rays corresponding to ambient occlusion (AO) and diffuse rays shot from the hit points of the primary rays (seven AO or diffuse rays per primary ray). For reporting the SAH cost of the BVH we used the following traversal and intersection costs: $c_t = 3$, $c_i = 2$.

In order to show the quality of our method we compare it to our implementation of the state-of-the-art method of Garanzha et al. [GPM11]. According to the original paper we are using 30bit Morton codes of the triangle centroids, where $30 - 3k$ most significant bits are used for creating clusters of triangles falling inside the same voxel of the hypothetical grid. These clusters are the leaves of the top part of the tree built with SAH. The bottom part of the tree (each cluster) is built with fast HLBVH method using the least significant 3k bits of the Morton codes. This means that up to $3k$ bottom levels of the BVH are built using HLBVH (the constant k relates to constant m used in the original paper in this way: $k = 10 - m$). We denote the method with $k = 10$ as $HLBVH_M$ because it builds the entire tree with *median splitting*. Since the behaviour of the HLBVH method is strongly dependent on the number of bits used, we are always reporting the value of k. In the results we are using maximum of four triangles per leaf as termination criteria. The SAH termination cannot be used in this method as the SAH part of the tree is always built down to a single cluster per leaf and during the HLBVH construction bounding boxes are not known.

Our implementation of the HLBVH method does not feature multiple GPU queues mentioned in the original paper and has moderately slower build times. Nevertheless, the traversal performance of the HLBVH method should not be impacted and may be even superior as we are using the traversal kernels of Aila and Laine [AL09] with compact BVH layout and Woop triangle representation [Woo04].

For our method a leaf is created when the number of triangles in a node is four or less or when the SAH termination criteria are met. The building and traversal results are given in Table 1 where the HLBVH method with $k = 4$ (as proposed by Garanzha et al.) is compared to our method. The HLBVH build times start to be lower than the ones of our method for scene *Crytek Sponza*, for smaller scenes our method is in fact faster. This can be explained by the different complexity of the two algorithms: after sorting the Morton codes the complexity of HLBVH build is $O(N)$, while our method following the standard top-down scheme ex-

Figure 5: *Snapshots for more complex architectural models: Sponza, Sibenik Cathedral, Fairy Forest, Crytek Sponza, Conference, Soda Hall, San Miguel, Power Plant, and for single geometric object models: Happy Buddha, Blade, Hairball rendered using diffuse rays.*

Scene	$HLBVH_4$			$OurBVH$		
	T_{GPU} $[ms]$	T_{CPU} $[ms]$	R $[-]$	T_{GPU} $[ms]$	T_{CPU} $[ms]$	R $[-]$
Sponza	7.5	26.5	0.22	10.5	13.5	0.43
Sibenik Cathedral	6.9	27.3	0.20	12.9	13.7	0.48
Fairy Forest	11.2	27.0	0.29	19.7	14.6	0.57
Crytek Sponza	11.7	28.9	0.29	27.8	14.4	0.65
Conference	10.6	27.3	0.28	29.6	14.6	0.66
Happy Buddha	37.4	34.7	0.51	113.4	20.1	0.84
Blade	54.9	36.9	0.59	173.8	20.7	0.89
Soda Hall	56.0	37.8	0.59	228.7	20.5	0.91
Hairball	103.8	47.9	0.68	298.9	24.8	0.92
San Miguel	179.3	50.1	0.78	911.6	39.4	0.95
Power Plant	252.7	52.9	0.82	1452.1	49.0	0.96

Table 2: *Absolute GPU kernel time (T_{GPU}), CPU management time (T_{CPU}) and relative CPU idle time ($R = \frac{T_{GPU}}{T_{GPU}+T_{CPU}}$, higher is better).*

hibits $O(N \log N)$ complexity, but lower CPU management overhead. The traversal performance, on the other hand, is almost always higher for our method. This is not only because SAH splitting is used down to the leaves but also because the SAH evaluation in the top part of the tree is allowed to separate triangles that would fall into a single cluster for the HLBVH method.

Given that our method typically has slower build but faster traversal there is a crossover point where using our method leads to a lower rendering time. These points are evaluated in Table 3. Notice that on scenes with uniform size and distribution of triangles, such as Happy Buddha and Blade the tree quality cannot be improved much by the SAH and the crossover point lies very far ($> 100 MRays$). We believe the cases where the traversal for our method is slower are caused by the SAH being only approximate measure of performance.

The behaviour of the HLBVH method with varying number of bits used, given by the parameter k is shown in Table 4. Generally, the build times decrease with increasing value of k, while the traversal performance also decreases. This is in agreement with the increase of the SAH cost as more levels are built with the HLBVH. For smaller scenes the relations are not as straightforward, since the clusterization for low values of k may force creation of very small leaves, which hampers traversal performance.

Table 2 shows the build times on both the GPU and the CPU, and the relative GPU utility R. The value of R shows how much of the time needed for the data structure build is spent on the GPU in relation to the total build time. Note that value $(1-R)$ represents the time spent by copying the data to the GPU and managing the computation. With increasing build times, this management overhead is relatively less significant but still important when targeting realtime or interactive applications. Our method clearly features less CPU management overhead. Moreover, the CPU can perform some meaningful computations during the entire run of our kernel, while for the standard method the intervals between kernel launches are very short and the CPU must frequently interrupt its computation to keep the GPU busy.

6.2. Divide-And-Conquer Ray Tracing

Second, we have tested the divide-and-conquer ray tracing described in Section 5. We provide results for the spatial median subdivision and for the cost model based on the RDH compared to HLBVH with $k = 4$ and our BVH for tracing collision detection rays.

For the collision detection test we assume that the scene contains a number of moving agents (corresponding for example to characters in a game). The movement of the agents is given by randomly selected line segments within the bounding box of the scene. Next, we take 20 equidistant points on each segment that we use as the agents' positions in a simulation consisting of 20 frames. For each frame we shoot 128 ray segments into the sphere around the agents' position, where the length of the ray segment is the distance between the current position of the agent and the next one. The approximation of collision detection between moving agents and the scene is then computed as the intersections of the ray segments with the scene. Note, that since the line segments are random, the agent speed (corresponding to the length of the ray segments) varies among the agents.

Figure 7 shows the comparison of the four introduced methods for computing the intersections of the collision detection rays. The number of agents is denoted by #agents. While the computational times for the $HLBVH_4$ and $OurBVH$ are dominated by the build time, the computational time of the divide-and-conquer methods ($DACRT$) is dominated by the number and length of rays. RDH is usually faster than the median splitting and more so on complex scenes, which comes from the extra knowledge during

		SAH cost [-]		Build [ms]		Trace performance [MRays/s]					
						Primary		Ambient Occlusion		Diffuse	
Scene	#tris	$HLBVH_4$	$OurBVH$	$HLBVH_4$	$OurBVH$	$HLBVH_4$	$OurBVH$	$HLBVH_4$	$OurBVH$	$HLBVH_4$	$OurBVH$
Sponza	76k	188.6	200.8	34.0	24.0	306.5	259.5	182.4	173.4	54.4	51.2
Sibenik Cathedral	80k	80.6	75.7	34.2	26.6	203.0	231.2	167.7	197.2	42.0	43.2
Fairy Forest	174k	82.6	82.5	38.2	34.3	95.5	167.9	63.7	78.3	46.5	54.9
Crytek Sponza	262k	199.9	193.0	40.6	42.2	159.4	172.7	84.4	87.0	36.3	34.4
Conference	282k	122.3	117.5	37.9	44.2	274.1	303.4	134.0	150.4	63.6	70.2
Happy Buddha	1,087k	194.9	172.1	72.1	133.5	266.9	332.6	291.5	335.4	252.5	288.8
Blade	1,765k	222.6	201.7	91.8	194.5	266.5	323.8	326.9	365.4	243.9	272.3
Soda Hall	2,169k	212.9	203.9	93.8	249.2	279.5	343.5	299.1	324.2	88.3	99.8
Hairball	2,880k	1352.4	1145.1	151.7	323.7	38.8	53.6	36.8	47.3	30.4	39.5
San Miguel	7,880k	215.4	194.0	229.4	951.0	35.9	60.3	25.0	33.3	13.0	18.2
Power Plant	12,748k	171.3	123.4	305.6	1501.1	26.3	55.6	77.1	122.9	9.1	13.7
average	-	-	-	102.7	320.4	84.2	129.7	80.7	100.3	31.9	40.2

Table 1: *Results for HLBVH with $k = 4$ compared to our SAH BVH building algorithm. For the primary rays 1M rays are shot while for the Ambient Occlusion and Diffuse rays 7M rays are shot. The average MRays/s are computed from averaged ray tracing times.*

Scene	C_1		C_4		C_M	
	N_r [MRays]	R [−]	N_r [MRays]	R [−]	N_r [MRays]	R [−]
Sponza	+	1.19	10.08	0.95	2.12	1.42
Sibenik Cathedral	+	1.00	+	1.03	1.56	1.74
Fairy Forest	+	1.24	+	1.17	4.23	1.26
Crytek Sponza	74.99	0.98	-	0.95	2.76	1.40
Conference	+	1.18	5.61	1.10	3.37	1.65
Happy Buddha	+	1.07	141.17	1.13	150.19	1.18
Blade	+	1.02	235.86	1.10	230.27	1.13
Soda Hall	43.54	1.17	120.18	1.12	22.90	1.76
Hairball	+	1.28	28.67	1.29	37.02	1.30
San Miguel	26.12	1.43	36.13	1.39	18.07	1.80
Power Plant	56.04	1.29	35.54	1.49	17.31	2.03

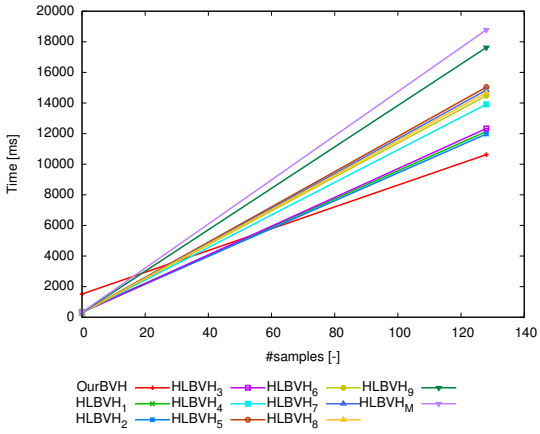


Table 3 & Figure 6: *The table shows the crossover of the time to image including build times for our BVH method and the HLBVH methods with various number of bits used for the HLBVH. The crossover point is computed as an intersection of lines given by two points: (0, T_b) and (128, T_i), where the first coordinate is the number of diffuse samples per pixel (for 1MPixel image), T_b is the build time of the BVH and T_i is the time to image (build time + primary rays time + diffuse rays time). C_1 gives the crossover point with $HLBVH_1$, similarly C_4 gives the crossover point with the method $k = 4$ and C_M is the crossover point with the fully median built HLBVH. N_r columns give the crossover points; the number of rays for which both methods have the same time to image. R columns give the ratio of traversal times for HLBVH and our method. The + sign is for cases where the time to image for our method is always lower than for the HLBVH method and the − sign vice versa. The right figure shows our BVH method compared to all of the HLBVH methods on the Power Plant model.*

Stat	$HLBVH_1$	$HLBVH_2$	$HLBVH_3$	$HLBVH_4$	$HLBVH_5$	$HLBVH_6$	$HLBVH_7$	$HLBVH_8$	$HLBVH_9$	$HLBVH_M$
SAH cost [-]	163.3	161.6	165.1	171.3	176.2	179.8	181.5	181.2	183.3	184.4
Build GPU [ms]	310.2	283.0	259.9	252.7	252.9	254.7	253.6	262.0	261.0	239.0
Build total [ms]	382.0	345.7	317.2	305.6	303.5	303.3	299.1	307.0	303.2	331.2
Primary [MRays/s]	36.5	40.7	29.5	26.3	23.9	26.7	22.4	23.1	14.9	13.9
Ambient Occlusion [MRays/s]	95.1	93.7	85.4	77.1	73.6	70.6	70.8	68.4	69.0	71.1
Diffuse [MRays/s]	10.5	10.7	10.4	9.1	8.5	8.8	8.6	8.7	7.2	6.8

Table 4: *Comparison of the HLBVH method based on the value of k for the Power Plant model. For the primary rays 1M rays are shot while for the Ambient Occlusion and Diffuse rays 7M rays are shot.*

the node splitting. The constants used in the formula deciding when to stop the subdivision and invoke the intersection tasks defined in Equation 2 were set as follows: $C_{INTERS} = 1$, $C_{SORT} = 80$. An intersection task is also invoked when the number of rays drops below 32 or the number of triangles drops below 16.

The collision detection rays were chosen because they have desirable properties for our parallel divide-and-conquer ray tracer: they are relatively short compared to the scene diagonal and relatively few rays are sufficient to compute the solution. Keeping the ray count low is important since the algorithm also needs to partition the rays, and global memory accesses are not cached in L1 in current generations of GPU, leading to excessive memory bus traffic. The ray length influences the dependencies between individual tasks. Since the introduction of dependent task limits the available parallelism and there are more dependencies with increasing depth of tasks the ray length directly influences the amount of parallelism and, thus, the performance of the method. For many infinitely long rays such as the diffuse rays the method is actually slower than when computed on the CPU.

7. Conclusion

We have proposed a novel method for massively parallel processing in the context of hierarchical algorithms dealing with 3D geometrical data. Our method runs entirely on the GPU and requires no management of the computation from the CPU side. We propose a methodology of subdividing a given hierarchical algorithm into tasks, phases, steps, and work chunks in order to map the algorithm to the parallel framework. We show two applications of our method: construction of the BVH and divide-and-conquer ray tracing on the GPU. We evaluated two proof of concept applications, which indicate that our approach has a good potential for massive parallelization of complex hierarchical problems.

In the future we would like to apply our method to other problems in computer graphics such as SBVH/kd-tree construction or GPU path tracing.

Acknowledgement

We would like to thank the contributors of the scenes used in our paper, Prof. C. Sequin for Soda Hall model, the University of North Carolina for the Power Plant model, Marko Dabrovic for the Sponza and Sibenik models, Ingo Wald for Fairy Forest, Greg Ward for the Conference model, Samuli Laine and Tero Karras for Hairball model, and Stanford repository for the other models. We would also like to thank Tero Karras, Timo Aila, and Samuli Laine for releasing their GPU ray tracing framework. Our research was supported by the Czech Science Foundation under research programs P202/11/1883 (Argie), P202/12/2413 (Opalis) and P202/10/1435.

References

[Áfr12] ÁFRA A. T.: Incoherent ray tracing without acceleration structures. In *Eurographics (Short Papers)* (2012), Andújar C., Puppo E., (Eds.), Eurographics Association, pp. 97–100. 1, 7

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG'09, ACM, pp. 145–149. 2, 3

[BH09] BITTNER J., HAVRAN V.: RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. In *25th Spring Conference on Computer Graphics (SCCG 2009)* (Budmerice, Slovakia, May 2009), Hauser H., (Ed.), pp. 61–67. 8

[CKL*10] CHOI B., KOMURAVELLI R., LU V., SUNG H., BOCCHINO R. L., ADVE S. V., HART J. C.: Parallel SAH k-D tree construction. In *Proceedings of the Conference on High Performance Graphics* (2010), HPG'10, Eurographics, pp. 77–86. 2

[CT08] CEDERMAN D., TSIGAS P.: On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2008), GH '08, Eurographics, pp. 57–64. 2

[CVKG10] CHEN L., VILLA O., KRISHNAMOORTHY S., GAO G. R.: Dynamic Load Balancing on Single- and Multi-GPU systems. In *Proceeding of IPDPS'10 conference* (2010), pp. 1–12. 2

[DPS10] DANILEWSKI P., POPOV S., SLUSALLEK P.: *Binned SAH Kd-Tree Construction on a GPU*. Tech. rep., Computer Graphics Group, Saarland University, June 2010. 2

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and Faster HLBVH with Work Queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG'11, ACM, pp. 59–64. 2, 6, 9

[HHS06] HAVRAN V., HERZOG R., SEIDEL H.-P.: On the Fast Construction of Spatial Data Structures for Ray Tracing. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006* (Sept. 2006), pp. 71–80. 6

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, August 2007, ch. 39, pp. 851–876. 7

[HSZ*11] HOU Q., SUN X., ZHOU K., LAUTERBACH C., MANOCHA D.: Memory-Scalable GPU Spatial Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics 17*, 4 (April 2011), 466–474. 2

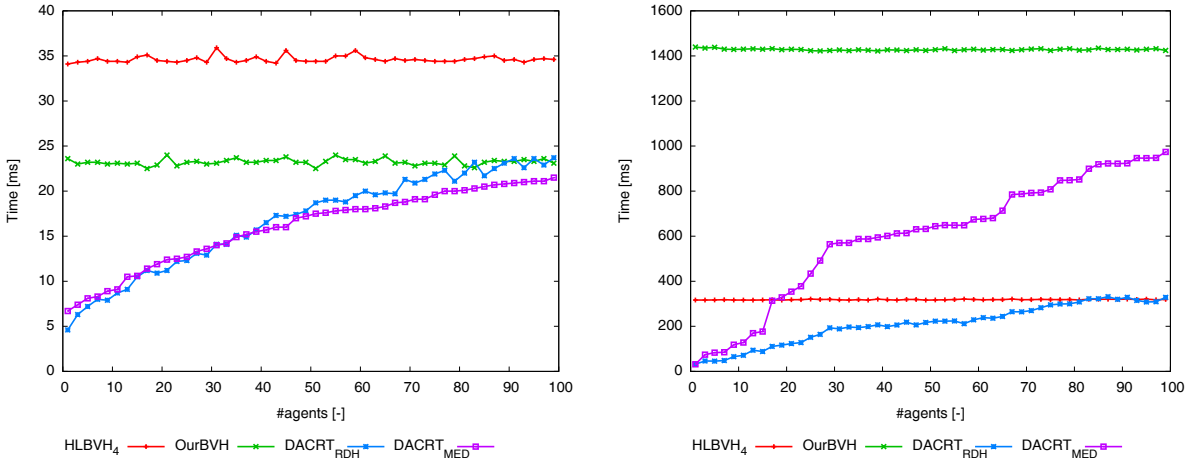
[KBS11] KALOJANOV J., BILLETER M., SLUSALLEK P.: Two-Level Grids for Ray Tracing on GPUs. *Computer Graphics Forum 30*, 2 (2011), 307–314. 2

[KS09] KALOJANOV J., SLUSALLEK P.: A Parallel Algorithm for Construction of Uniform Grids. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG'09, ACM, pp. 23–28. 2

[Lee10] LEE V. E. A.: Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 451–460. 2

[LGS*08] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2 (2008), 375–384. 2

[Mor11] MORA B.: Naive ray-tracing: A divide-and-conquer approach. *ACM Trans. Graph. 30*, 5 (Oct. 2011), 117:1–117:12. 1, 2, 7



(a) *Sponza* **(b)** *Power Plant*

Figure 7: *Comparison of various methods for computing collision detection rays on the Sponza and Power Plant models in dependence on the number of moving agents. The computation times are for fully computing (build+trace) one batch of rays and are averaged over 20 batches simulating agent's movement in the scene. Each agent is checked for collision with the scene using 128 rays uniformly shot into the sphere at the agents' positions and the ray length is set as the distance between current position and the position in the next step.*

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable Parallel Programming with CUDA. *Queue 6*, 2 (Mar. 2008), 40–53. 2, 3, 9

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray Tracing on Programmable Graphics Hardware. *ACM Trans. Graph. 21*, 3 (July 2002), 703–712. 2

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry. In *Proceedings of the Conference on High Performance Graphics* (2010), HPG'10, Eurographics, pp. 87–95. 2

[SBU11] SOPIN D., BOGOLEPOV D., ULYANOV D.: Real-Time SAH BVH Construction for Ray Tracing Dynamic Scenes. In *21th International Conference on Computer Graphics and Vision (GraphiCon)* (2011), pp. 74–77. 2

[SGPT11] SUNDELL H., GIDENSTAM A., PAPATRIANTAFILOU M., TSIGAS P.: A Lock-Free Algorithm for Concurrent Bags. In *SPAA* (2011), Rajaraman R., Meyer auf der Heide F., (Eds.), ACM, pp. 335–344. 2

[SGS10] STONE J. E., GOHARA D., SHI G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test 12*, 3 (May 2010), 66–73. 2

[SKK*12] STEINBERGER M., KAINZ B., KERBL B., HAUSWIESNER S., KENZEL M., SCHMALSTIEG D.: Softshell: Dynamic Scheduling on GPUs. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 161:1–161:11. 2

[SSK07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN A.: Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. *Computer Graphics Forum 26*, 3 (2007), 395–404. 2

[TPO10] TZENG S., PATNEY A., OWENS J. D.: Task Management for Irregular-Parallel Workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics* (2010), HPG'10, Eurographics, pp. 29–37. 2

[TZ01] TSIGAS P., ZHANG Y.: A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. In *Proc. of the 13th ACM Symposium on Parallel Algorithms and Architectures* (2001), ACM, pp. 134–143. 2

[Wal07] WALD I.: On fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), RT '07, IEEE Computer Society, pp. 33–40. 2

[Wal12] WALD I.: Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 1 (January 2012), 47–57. 2

[WH06] WALD I., HAVRAN V.: On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006* (Sept. 2006), pp. 61–69. 2

[WK09] WÄCHTER C., KELLER A.: Efficient Ray Tracing Without Acceleration Data Structure, 2009. U.S. Patent Applications Publication No. US2009/02225081 A1. 1, 2, 7

[WMG*09] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the Art in Ray Tracing Animated Scenes. *Computer Graphics Forum 28*, 6 (2009), 1691–1722. 2

[Woo04] WOOP S.: *A Ray Tracing Hardware Architecture for Dynamic Scenes*. Tech. rep., Saarland University, 2004. 9

[WZL11] WU Z., ZHAO F., LIU X.: SAH KD-tree Construction on GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG'11, ACM, pp. 71–78. 2

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time KD-tree Construction on Graphics Hardware. *ACM Trans. Graph. 27*, 5 (Dec. 2008), 126:1–126:11. 2, 6

Appendix G

Adaptive Global Visibility Sampling

Bittner, J. - Mattausch, O. - Wonka, P. - Havran, V. - Wimmer, M.: Adaptive Global Visibility Sampling. ACM Transactions on Graphics (TOG). 2009, vol. 28, no. 3, p. 94:1-94:10. ISSN 0730- 0301. **IF=3.489**

Adaptive Global Visibility Sampling

Jiří Bittner* Oliver Mattausch† Peter Wonka‡ Vlastimil Havran* Michael Wimmer†

*Czech Technical University in Prague§ † Vienna University of Technology ‡Arizona State University

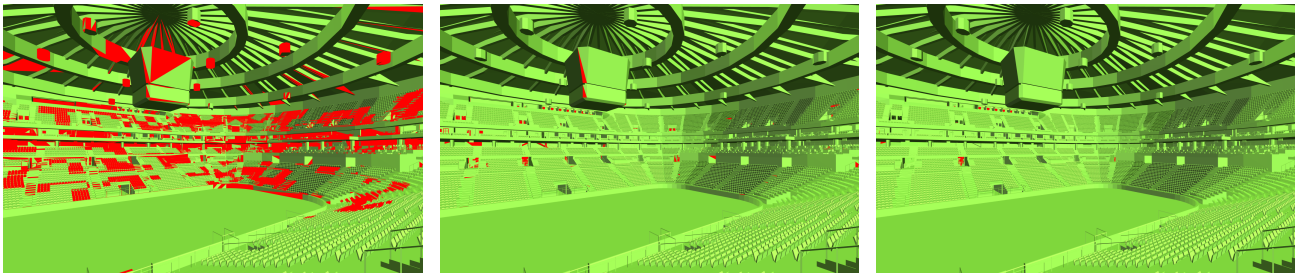


Figure 1: *Results of visibility computations after 1 minute of sampling. Visibility errors are marked in red. Left: Traditional per-view cell sampling. Middle: Adaptive Global Visibility Sampling. Right: Adaptive Global Visibility Sampling with visibility filter. Observe the severe underestimation of visibility in the left image. The visibility computed by our method in the middle produces significantly less visible artifacts. To the right, our method with a visibility filter applied is practically artifact-free. Note that during this minute, the potentially visible sets for* all *8,192 view cells in this example model have been generated.*

Abstract

In this paper we propose a global visibility algorithm which computes from-region visibility for all view cells simultaneously in a progressive manner. We cast rays to sample visibility interactions and use the information carried by a ray for all view cells it intersects. The main contribution of the paper is a set of adaptive sampling strategies based on ray mutations that exploit the spatial coherence of visibility. Our method achieves more than an order of magnitude speedup compared to per-view cell sampling. This provides a practical solution to visibility preprocessing and also enables a new type of interactive visibility analysis application, where it is possible to quickly inspect and modify a coarse global visibility solution that is constantly refined.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible line/surface algorithms

Keywords: Visibility preprocessing, from-region visibility

1 Introduction

This paper addresses the problem of computing Potentially Visible Sets (PVS) for a set of view cells. PVS computation is typically carried out as a preprocess in applications that need to know visibility information in advance and where online visibility cannot be used. These applications include polygon budget computations, level design for computer games, line-of-sight analysis, planning geometry and texture transmissions for networked virtual environments, acquisition planning for object scanning, or path planning for artificial intelligence in computer games.

Recently it has been shown that sampling is a robust solution to PVS computation. In particular the Guided Visibility Sampling (GVS) algorithm introduced by Wonka et al. [Wonka et al. 2006] efficiently computes a PVS for a view cell using ray casting. While GVS is very efficient at sampling from a single view cell, it does not exploit the coherence among different view cells and does not work in progressive fashion with respect to all view cells.

In this paper we propose a method which computes from-region visibility for all view cells simultaneously in a progressive manner. We build on global sampling strategies [Mattausch et al. 2006] and use visibility information from one ray for all view cells it intersects. The main contributions of the paper are: (1) We introduce a set of sampling strategies for global visibility computation which adapt to the scene geometry and explore the visibility coherence. (2) We show the first progressively refined global visibility solution, in which PVS estimates for all view cells are obtained within seconds or minutes (see Figure 1).

The method achieves more than an order of magnitude speedup compared to per-view cell sampling. This provides a practical solution to visibility preprocessing and also enables a new type of interactive visibility analysis applications (see Figure 2), where it is possible to quickly inspect and modify a global visibility solution. The proposed algorithm consumes little memory, is easy to implement, and works on arbitrary input scenes consisting of view cells and objects intersectable with a ray tracer.

2 Related work

Visibility and occlusion is essential to a wide area of graphics problems and we refer the reader to recent surveys for a broader discussion of visibility [Cohen-Or et al. 2003; Bittner 2003]. In the following we will focus on visibility algorithms that compute visi-

*e-mail:{bittner|havran}@fel.cvut.cz
†e-mail:{matt|wimmer}@cg.tuwien.ac.at
‡e-mail:wonka@asu.edu
§Faculty of Electrical Engineering

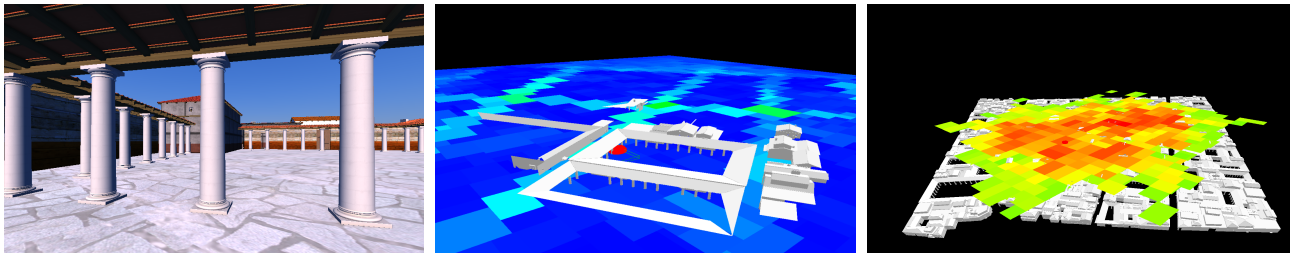


Figure 2: *Interactive visibility analysis. Left: A view of the Pompeii model. Middle: The PVS of the corresponding view cell depicted together with PVS costs of the view cells at the same horizontal level (warmer colors correspond to higher costs). Right: Interactive exploration of visibility hot spots. Note that in this scene the hot spots correspond to view points above the roofs in the central part of the scene.*

bility from one or multiple regions in space as opposed to calculating visibility from a single point.

Geometric solutions. First, it is important to study exact solutions to the visibility problem [Duguet and Drettakis 2002; Nirenstein et al. 2002; Bittner 2003; Haumont et al. 2005]. These algorithms are an important inspiration, but it is currently unclear if exact algorithms can be extended to handle large scenes robustly. Therefore, many authors set out with simplifying assumptions to make the problem more tractable. Interesting simplifications are 2.5D visibility [Wonka et al. 2000; Bittner et al. 2001; Koltun et al. 2001], architectural scenes [Airey et al. 1990; Teller and Séquin 1991], the restriction to volumetric occluders [Schaufler et al. 2000], or the restriction to larger occluders close to the view cell [Durand et al. 2000; Leyvand et al. 2003].

Visibility Sampling. A popular concept in visibility computation is to sample visibility interactions on a regular grid using graphics hardware, giving an approximate solution. Conservative algorithms attempt to interpret a pixel as a small subset of ray space [Durand et al. 2000; Wonka et al. 2000; Koltun et al. 2001; Leyvand et al. 2003] and record occlusion information only if all rays corresponding to the ray space subset are blocked. These algorithms are not able to handle complex visibility configurations within the subset of ray space defined by a pixel and basically require a single occluder to block all rays [Cohen-Or et al. 1998]. As computing power increased, it became feasible to avoid conservative algorithms and just sample ray space very densely. A conceptually elegant idea is to shoot either random rays from a view cell [Airey et al. 1990], or to first sample the boundary of the view cell with points and then sample visibility from each of these points [Levoy and Hanrahan 1996]. Recent algorithms try to address the question on how to best position new samples based on visibility information from previous samples [Gotsman et al. 1999; Pito 1999; Wilson and Manocha 2003; Nirenstein and Blake 2004; Wonka et al. 2006].

Sampling in other fields. Sampling has been widely used in global illumination to calculate the illumination integral [Dutré et al. 2003]. In contrast to global illumination, it is not an integral that is evaluated in visibility computations, but the *maximum set* of different values of the visibility function. Related to our technique are the VEGAS algorithm [Lepage 1980], which adapts the sampling distribution based on an estimation of the integrand from previous samples, and Metropolis sampling [Veach and Guibas 1997], where a new sample is generated by displacing the previous one randomly. A similar strategy is known as *mutation* in genetic algorithms [Goldberg 1989]. Analogous to genetic algorithms, we start with a random sampling step and then generate new samples by mutation. In sampling theory, this strategy is also called *adaptive cluster sampling* [Thompson and Seber 1996], and it is used to sample rare populations.

3 Overview

3.1 Problem Statement

We consider visibility problems of the following form: given a view space as a set V of view cells $v \in V$, and a set O of objects $o \in O$, we are interested in which objects can be seen from which view cell. More formally, let a ray $r = (x_r, d_r)$ be defined by a ray origin x_r and a ray direction d_r. The ray casting function $h(r) \in O$ assigns each ray the first object hit by the ray. Then the desired visibility solution is the *exact visible set* EVS_v for each view cell. It is defined as the actual set of objects that can be seen along some ray from v: $EVS_v = \{h(r) | x_r \in v\}$.

The potentially visible set PVS_v of a view cell is an approximation to EVS_v determined by a particular visibility algorithm. Conservative algorithms overestimate the visible set ($PVS_v \supseteq EVS_v$), while aggressive algorithms underestimate it ($PVS_v \subseteq EVS_v$). Sampling algorithms such as the one described in this paper provide an aggressive solution. We also describe a visibility filter to extend PVSs, which makes the solution approximate ($PVS_v \sim EVS_v$).

A *global visibility solution* is simply the set $\{PVS_v | v \in V\}$. A global visibility algorithm is *progressive* if at each step i it computes a solution $\{PVS_v^i | v \in V\}$ with $PVS_v^{i-1} \subseteq PVS_v^i \subseteq EVS_v$. Loosely speaking, we only call a solution progressive if it is so globally, i.e., the ratio $|PVS_v^i|/|EVS_v|$ increases for all view cells v simultaneously during the runtime of the algorithm (in particular, sequential view cell evaluation is not a progressive solution).

3.2 Algorithm Overview

The Adaptive Global Visibility Sampling (AGVS) algorithm uses *sampling* to determine visibility. We assume the availability of a ray tracing algorithm [Shirley et al. 2006] that can evaluate $h(r)$, and in addition $r \cap v$ for any view cell v.

At the core of our algorithm is a *global visibility sampler* (Section 4.1), which casts bidirectional rays to determine maximal free line segments in the scene, and then determines their contribution to all view cells. Thus a single visibility sample can contribute to many view cells.

The second part of the algorithm generates the samples. The 5D sampling domain is too large to quickly capture all important rays by regular sampling. Therefore we use different heuristical distributions which are combined in an *adaptive mixture distribution* (Section 4.2) that takes into account their success in discovering new PVS entries. The sampling uses several ray distributions that are

suitable for a global visibility algorithm. *Stationary distributions* (Section 4.3) sweep visibility globally and seed the *mutation-based distributions* (Section 4.4), which focus the sampling at places of visibility changes and allow adapting the sampling rate to the distance of visible objects.

We present *visibility filtering* (Section 5), which counteracts errors due to undersampling in early stages of the algorithm by including in the PVS also objects that are likely to be visible based on the sampling density. We also describe a quick algorithm to discover areas affected by scene edits, thus allowing *dynamic edits* to the scene without having to recompute the whole visibility solution (Section 6).

4 Adaptive Global Visibility Sampling

In this section we describe novel sampling strategies that are well suited to the global computation of visibility.

4.1 Global Visibility Sampling

Visibility sampling relies on *ray tracing* to obtain visibility information. One main reason for the efficiency of our approach is the use of spatial coherence in visibility. In contrast to a per-view cell algorithm, where each ray can only contribute to one single view cell, we determine all view cells the ray encounters, as was proposed in the context of view-cell optimization [Mattausch et al. 2006], and calculate the contribution to those view cells. For this, we use *visibility samples* created from rays.

Visibility sample definition. In visibility sampling algorithms, a sample ray r is usually defined to start in a view cell ($x_r \in v$) and with a direction d_r. A visibility contribution is then determined by shooting the ray using a standard ray tracer and determining the closest object $h(r)$. We define the contribution of a ray as

$$\begin{aligned} H(r) &= \{h(r)\} \quad \text{if } h(r) \in O \text{ (object hit)} \\ &= \{\} \quad \text{otherwise (no object hit)} \end{aligned} \tag{1}$$

In global visibility, however, we are not only interested in the visibility contribution of the hit object to the view cell at the ray origin, but to all view cells pierced by the ray along an unobstructed path from the hitpoint. This path is equivalent to the *maximal free line segment* defined by r. To obtain this line segment, we not only shoot the original ray, but also the ray in the opposite direction, i.e., $-r = (x_r, -d_r)$, and obtain a second hit $h(-r)$. Our sample s is thus a line segment associated with zero, one or two visible objects at its endpoints, with contribution $H(s) = H(r) \cup H(-r)$. A sample with one or two visible objects ($|H(s)| > 0$) is a *valid sample*.

Updating view cells. The view cells affected by a valid sample are determined by intersecting the line segment with the data structure containing the view cells. The objects associated with the sample are then added to *all* view cells pierced by the line segment.

Sample contribution. During this process, the *contribution* of the sample is evaluated in order to distinguish between samples which give us valuable information about visibility and samples which either hit no object or hit objects already discovered as visible. This measure will be used to drive the sampling process (Section 4.2).

The local contribution of the sample to a particular view cell equals the number of objects associated with the sample that are added to the PVS of the view cell (and have not already been included in the PVS in previous steps of the algorithm): $\bar{c}(s,v) = |H(s) \backslash PVS(v)|$.

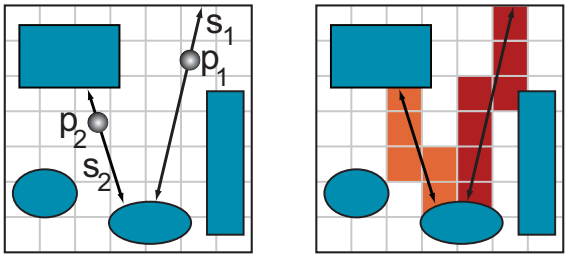


Figure 3: *Left: This figure shows the creation of visibility samples. The scene objects are shown in blue. A sample point (see p_1 and p_2) is generated together with a direction vector. A ray is cast from the sample point in the generated direction as well as into the opposite direction. Right: A ray associates all its points with visibility of object(s) on its endpoints. These objects are added to all PVSs of view cells pierced by the ray.*

Note that $\bar{c}(s,v)$ yields a value of 0, 1 or 2. The total contribution $c(s)$ of the sample is simply the sum of contributions to all view cells $V(s)$ which are pierced by the sample:

$$c(s) = \sum_{v \in V(s)} \bar{c}(s,v) \tag{2}$$

4.2 Adaptive Mixture Distribution

In addition to the explicit use of spatial visibility coherence, the most important novel contribution of our global visibility algorithm is the ability to adapt to the visibility structure in the scene. This is achieved in two ways: first, by choosing sample distributions according to previous visibility contributions, and second by using a ray mutation strategy which places samples near visibility events and thus adapts to the required sampling density in ray space (described in Section 4.4).

We have found that no single sampling strategy is efficient for all types of scenes, as the efficiency of a sampling strategy depends on the scene properties and its visibility characteristics. We therefore employ a probabilistic approach based on a distribution mixture: we allow the use of several different distributions to generate visibility samples. The success of previous samples generated by each distribution is used to drive its selection probability, thus automatically adapting to the scene visibility properties. This strategy is reminiscent of VEGAS importance sampling [Lepage 1980], although we do not try to compute an integral here.

More specifically, the contribution $C(D)$ of a sample distribution D at a specific point in time is defined as the sum of contributions $c(s)$ of all samples s in a reference set $S(D)$. The reference set is typically the set of rays generated by the distribution D in a certain time window:

$$C(D) = \sum_{s \in S(D)} c(s). \tag{3}$$

Next we compute the average contribution per sample from D as $c_s(D) = C(D)/|S(D)|$. To account for the fact that samples from different distributions have different processing costs (e.g. a distribution need not be very succesful per ray, but the rays are generated and cast very quickly) we compute the *weight* $w(D)$ of the

distribution D as its average contribution per time unit: $w(D) = c_s(D)/t_s(D)$, where $t_s(D)$ is the average time for processing a sample from the distribution D. $t_s(D)$ is measured for each distribution in a *calibration pass* by generating and processing a sufficiently large number of samples (e.g. 100k) from each distribution. Note that $w(D)$ can slightly change over time, so we repeat the calibration passes during the computation (e.g. after each 100M samples).

The probability for drawing a sample from distribution D_i in the mixture distribution is then set to:

$$p(D_i) = \frac{w(D_i)}{\sum_D w(D)}. \quad (4)$$

The following two subsections describe the distributions which we use in the algorithm. Three of these distributions are stationary, while two mutation-based strategies adapt to the actual visibility contributions.

4.3 Stationary Ray Distributions

A distribution is stationary if each sample is independent of previously drawn samples. These distributions are used to provide an initial efficient covering of ray space and to seed the more adaptive mutation-based strategies described below. The stationary distributions explore the whole ray space and therefore guarantee the progressivity of the algorithm.

In contrast to per-view cell visibility, there is no obvious "uniform" sample distribution. Instead, the best strategy to discover new visible objects depends on the visibility configuration of the scene. We list three distributions which we have found to work well. All rely on low-discrepancy series like the Halton sequence to generate samples. We use independent random variables $\psi_1, \psi_2, \ldots$ in the range $[0,1)$.

View space-direction distribution. This distribution makes sure that all view cells are sampled in all directions. We generate a random point x in view space using a uniform distribution and a random direction d using a uniform distribution in the directional space with spherical coordinates $\phi = 2\pi\psi_1, \quad \theta = \arccos(1-2\psi_2)$.

Object-direction distribution. This distribution makes sure that all objects are sampled in all directions (note that this can be inefficient if the view space does not fully include the object space). We generate a random point x on the surface of a randomly chosen object, and a random direction d using a cosine-weighted uniform distribution on the hemisphere erected over the tangent plane of x. The spherical coordinates of d are: $\phi = 2\pi\psi_1, \quad \theta = \arccos\sqrt{\psi_2}$.

Two-point distribution. This distribution focuses on the most probable visibility interactions between objects and view cells. It can be seen as a combination of the previous two strategies which considers only the points generated by them and not the directions. So from a point o on a random object and a point v in view space, the ray generating the visibility sample is $r = (v, o-v)$. The most important feature of the two-point distribution is that it adapts to the shape of the scene. For example, typical urban scenes are much wider than they are high. This fact is taken into account in the two-point distribution so that most samples are cast in a roughly horizontal direction. For this reason, the two-point distribution is typically the most successful stationary sampling strategy.

4.4 Mutation-Based Distributions

Even using an optimal mix of stationary sample distributions, the size of ray space that needs to be sampled is prohibitively high. The required sampling density is very non-uniform: for a particular view cell, far away regions need to be sampled more densely than near regions. Furthermore, Wonka et al. [2006] have shown that the efficiency of visibility sampling can be vastly improved by trying to sample near possible changes in visibility. We exploit these two observations using mutation-based sampling distributions: a *two-point mutation* distribution and a *silhouette mutation* distribution.

Mutation candidate maintenance. Both strategies are based on the following principle: During the sampling process, each sample with non-zero contribution $c(s)$ (regardless of which distribution generated it) is stored as a candidate for mutation. In case the visibility sample has two objects associated with it ($|H(s)| = 2$), and both objects actually generated a contribution in at least one cell, we generate two mutation candidates from it, as we distinguish between the segment termination point (where the object under consideration is hit) and the segment origin (either the intersection with an object or a view cell). These two points and the object id of the object are stored with the sample.

All mutation candidates for a strategy are collected in a buffer. When a new sample is requested from a mutation-based strategy, a candidate is chosen from the buffer. However, instead of choosing randomly from the buffer, we observe that the history of mutations conveys important information about the possible importance of a candidate: If a candidate has received a large number of mutations already, it is likely that its neighborhood is already well explored. On the other hand, new mutation candidates have not received any mutations so far, and especially after sampling has been running for some time, new mutation candidates represent more "difficult" cases of visibility. We therefore count the number of mutations generated from each candidate. This mutation count is used to sort the buffer and we always select the candidate with the lowest mutation count value. If the buffer is full, the candidate with the highest value is dropped.

Two-point mutation. The aim of the two-point mutation is to adapt the sampling rate of visibility to the complexity of ray space. More specifically, given a segment $s = (s_o, s_t)$, the segment termination point s_t is mutated so as to discover nearby *objects*, while the segment origin s_o is mutated so as to discover nearby *view cells* (note that it is entirely due to the global nature of the algorithm that a strategy for discovering new view cells is possible at all).

Let $o(x)$ be the object or view cell (in case the reverse ray generating the segment didn't hit an object) associated with point $x \in \{s_o, s_t\}$, and $r(o)$ the radius of the bounding volume of object or view cell o. Then we construct a plane perpendicular to the segment, and mutate x by drawing a new point x' from a two-dimensional Gaussian distribution on this plane centered at x with standard deviation $\sigma = r(o(x))$ (see Figure 4, left).

Note that for both mutation-based strategies, once we have obtained the new segment (s'_o, s'_t), a visibility sample is generated from this segment by creating a forward ray $r' = (x_{r'}, d_{r'})$ together with the reverse ray $-r'$ with $x_{r'} = (s'_o + s'_t)/2$ and $d_{r'} = s'_t - s'_o$. The new ray origin is chosen at the center of the new segment so as to avoid local occlusion at the start or termination points of the segment.

Silhouette mutation. The silhouette mutation adapts the deterministic adaptive border sampling strategy proposed by Wonka et al. [2006] for a progressive setting. This strategy places samples

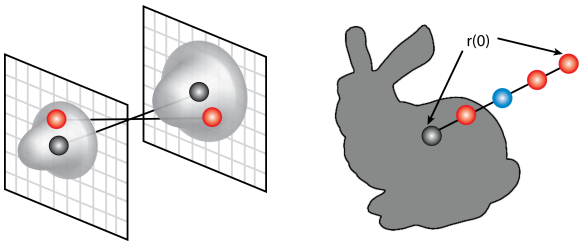


Figure 4: *Left: The two-point mutation moves both start and endpoints according to Gaussian distributions. Right: Silhouette search.*

near the silhouettes of newly discovered objects, where changes in visibility are most likely to occur. However, deterministically sampling the whole silhouette of an object is not only computationally expensive, but would also quickly saturate the list of mutation candidates.

Therefore we chose a probabilistic approach that randomly selects *one* silhouette point. As in the two-point mutation, a plane perpendicular to the segment is placed at s_t. On this plane, we choose a random direction d. Then we shoot "discovery rays" with endpoints s'_t along the segment $(s_t, r(o) \cdot d)$. The closest discovery ray that does not intersect the object is chosen as a silhouette ray and the mutation point s'_t is used to construct a new visibility sample (see Figure 4, right).

Note that for most ray tracers, these discovery rays can be evaluated much faster than ordinary rays. First, the region of interest can be restricted to the bounding volume of the object so that the ray does not need to be intersected with the whole scene. Second, the segment origin remains fixed for all rays, so that ray packet optimizations can be exploited. For example, shooting packets of 4 rays, a quaternary search of depth 3 provides a ray very close to the object silhouette.

Important visibility events also appear at possible *depth discontinuities* discovered by the silhouette sample. For this, we adapt the reverse sampling strategy described by Wonka et al. [2006]. When the silhouette ray $r = (s_o, s'_t - s_o)$ is cast, a depth discontinuity is reported if the distance between mutated segment endpoint s'_t and the actual hitpoint $h(r)$ is larger than k times the original object radius: $s'_t - h(r) > k \cdot r(o)$, where k is a user supplied constant (we used $k = 3$). This margin is intended to avoid situations where a closer hitpoint on the same object could be interpreted as a depth discontinuity. Now instead of looking for the exact depth discontinuity as in reverse sampling, we do a new silhouette mutation as described above using the segment $(s'_t, h(r))$, and store the resulting ray along with the original silhouette ray.

4.5 Termination of the computation

An important problem of previous visibility preprocessing techniques has been choosing the parameters of the method in order to achieve a good solution. For conservative methods, the parameters need to be set so that the PVS is not too overestimated, whereas for aggressive methods the final pixel error needs to be controlled.

The fact that our method is based on sampling allows us to devise a very elegant way to estimate an average pixel error on the fly. The estimated pixel error serves as a measure of the quality of the current solution and also as an intuitive termination criterion.

We define the average pixel error as the average number of rays from a randomly positioned and oriented camera with a given resolution (number of rays) which hit a different object when using the PVS related to the camera position compared to the situation when using all scene objects.

The distribution of rays covered by randomly positioned cameras corresponds exactly to the view space-direction distribution of rays described above. Thus we evaluate the ratio of contributing rays generated by the view space-direction distribution. As a contributing ray we count a ray which generated a new PVS entry with respect to the view cell contaning the origin of the ray. The average pixel error ε is given as $\varepsilon = resolution \cdot N_c/N$, where N is the number of rays generated by view space-direction distribution and N_c is the number of contributing rays.

In order to obtain a sufficiently large N we use a time window consisting of rays recently generated using the view space-direction distribution. We adapt the size of the window following the ratio of N_c to N: in the beginning of the computation a small window is sufficient, whereas in the nearly converged state, when the pixel error is small (1 pixel / 1M pixel image), we need to collect many rays in order to estimate the pixel error with reasonable precision. By following the weak law of large numbers for binomial distributions we get:

$$N \geq \frac{1-\varepsilon}{k^2 \varepsilon (1-P)}, \tag{5}$$

where P and k are constants such that P represents the desired probability with which the absolute difference between the estimated error and the real error is smaller than $k\varepsilon$. We observed that a good tradeoff between the size of the window and the accuracy of the estimation is achieved using $P = 0.9$ and $k = 0.5$.

4.6 Putting It All Together

The complete Adaptive Global Visibility Sampling algorithm works in a loop as follows:

```
while (!terminate)
{
  select distribution
  draw a sample from selected distribution
  cast forward and reverse rays
  if (hit)
  {
    update view cells
    if (contribution > 0)
      store sample as mutation candidate
  }
  update distribution probabilities
}
```

Note that since the solution is progressive, it can be inspected at any time, and if it is not satisfactory, the algorithm can easily be resumed.

Loop optimization. In practice the loop is not carried out for individual rays, but for larger batches of rays (e.g., 1M). This allows reordering the samples so as to achieve better coherence for the ray tracer. In addition, if the window used for calculating ray distributions is chosen to be exactly one batch, the update of the distribution probabilities $C(D)$ (see Section 4.2) needs to be carried out only once per batch.

5 Visibility Filtering

While Adaptive Global Visibility Sampling is an aggressive algorithm (i.e., each PVS only contains a subset of the exact visible set), the sampling strategies described in the previous section aim to approach the exact solution as quickly as possible. However, especially in the initial phases of the algorithm, visible errors will appear.

To cope with this problem, we introduce the so-called *visibility filter*, which extends the computed PVSs by additional objects which are likely to be visible. The main novelty is that we take into account the visibility error expected from the sampling density. This means that PVSs for regions which have been sampled densely will not be extended significantly, whereas undersampled regions will have more objects added. The visibility filter fills gaps in "visible fronts" that appear when the sampling rate in a region is lower than the object density. It cannot discover objects that are visible in an isolated manner (i.e., objects smaller than the sampling density that have no already discovered visible neighbors).

The visibility filter can be applied as a postprocess to all PVSs after running the main algorithm, or it can be evaluated lazily during walkthrough for the current view cell. We present an *object space filter*, which adds objects in proximity of already visible objects, and a *view space filter*, which merges visible objects from neighboring view cells.

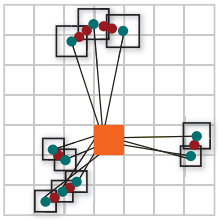


Figure 5: *Illustration of the object space visibility filter. Objects originally in the PVS are shown in blue. The extended bounding volumes are shown in black. Objects added to the PVS by the filter are shown in red.*

Object space filter. The object space filter works on a view cell v and its potentially visible set PVS_v. For each object $o \in PVS_v$, the size of *extension* $e(o)$ is calculated as the estimated distance to a visibility sample in the vicinity of the object. The object space filter is then applied by extending the bounding volume of each object o by $e(o)$ (e.g., the radius of a bounding volume or the axes of a bounding box), and adding to PVS_v all scene objects that intersect one of the extended bounding volumes (see Figure 5). The extension $e(o)$ can be calculated either from the density of all visibility samples intersecting the view cell (global extension $e_g(o)$), or from the density of samples that hit the object (local extension $e_l(o)$).

To estimate the *global extension* $e_g(o)$, we assume that n_v visibility samples are uniformly distributed on a sphere of radius $d(o)$, which is the distance of the object from the view cell. As extension we use half the approximated distance between two neighboring samples on this sphere: $e_g(o) = 2d(o)/\sqrt{n_v}$.

To estimate the *local extension* $e_l(o)$, we assume that the $n_v(o)$ visibility samples which hit the object are uniformly distributed on a disk of radius $r(o)$, which is the radius of the object bounding volume. As extension we use half the approximated distance between two neighboring samples on this disk: $e_l(o) = r(o)/\sqrt{n_v(o)}$.

Note that both estimations are not accurate: the global estimation ignores that rays due to the mutation-based strategy are not distributed uniformly, while the local estimation ignores that parts of the object may be occluded from the view cell, leading to an overestimation of e_l. In practice we therefore choose the minimum of the two estimations, and allow the user to increase or decrease the filter size with a constant k for more conservative or more aggressive results: $e(o) = k \cdot min(e_g(o), e_l(o))$.

View space filter. The view space filter is useful if the size of the view cells is relatively small (i.e., comparable to size of the objects). This filter is very simple: it merges the PVS of the given view cell with the PVSs of neighboring view cells, for example those with the most similar PVSs.

6 Dynamic Updates

Previous PVS-based methods were not applicable in interactive scenarios in which the scene gets manipulated. Our method can work in interactive sessions and so we also propose a method for updating the global visibility solution after scene edits.

Scene edits are implemented as follows: (1) When deleting an object, we remove it from the PVSs of all view cells. (2) When inserting an object O, we need to remove all objects from the current visibility solution that might be hidden by O: For each view cell v we construct a penumbra shadow volume of O and remove all objects of PVS_v which intersect the shadow volume (see Figure 6). (3) When editing an object (e.g. scaling, translation, mesh modification), we perform subsequent deletion and insertion of the object.

Note that before we continue with the visibility computation, we also need to update the ray casting data structures in order to reflect the scene edits.

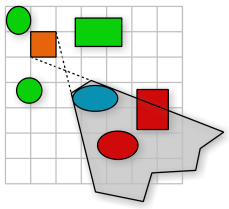


Figure 6: *Dynamic visibility updates: One object has been inserted (in blue). For a given view cell (in orange) the PVS objects intersecting the penumbra of the inserted object are removed from the PVS (in red). The remaining PVS objects (in green) can not be affected by the insertion operation.*

AGVS is suitable for dynamic edits because the visibility solution gets constantly refined, even without explicit treatment of the edits in the sampling process. The removed entries are quickly reinserted by our visibility sampling strategies (if they are still visible); in particular the mutation-based strategy will automatically focus on places in which the entries have been invalidated.

Dynamic updates are an important tool in global visibility analysis: a user can tentatively insert or edit an object (e.g., a wall or other blocker) and observe the effect this object has on the visibility solution and on render cost. This can be used to remove visibility hotspots in large scenes.

7 Results

We have evaluated the proposed method on five different scenes, which are depicted in Figure 8 (top row). Statistics including the number of triangles, objects, and view cells are shown in Table 1. The view cells are represented as kd-trees, the objects as bounding volume hierarchies (BVH). The view space kd-tree is generated according to the optimized construction described in [Mattausch et al. 2006]. The object BVH is built using the surface-area heuristics. The batch size for the computation was set to 1M samples, the buffer size for the mutation candidates was 2M samples.

The results were measured on a server with two Intel Xeon E5440 2.83GHz quad-core CPUs with 32GB of RAM. We used a custom BVH-based ray tracer which provides between 0.1M and 1M rays/s on the tested scenes. The ray tracer is about 30% slower than the state-of-the-art [Reshetov et al. 2005], but has very fast setup times (a few seconds for building a kd-tree for Arena, about 1-2 minutes for Powerplant), which is important for visibility analysis applications. Note that in the plots in this section, one sample always corresponds to two rays being cast, the forward and the reverse ray.

Scene	triangles	objects	view cells
Vienna	3,609,675	6,156	8,192
Arena	4,528,160	7,804	8,192
Pompeii	5,646,041	12,288	8,192
PowerPlant	12,748,510	10,150	8,192
Boeing 777	337,000,000	130,000	8,192

Table 1: *Statistics for all scenes.*

We have implemented the presented method in a multi-threaded application. Visibility is computed in the background by the visibility computation thread, while the GUI thread allows interactive manipulation/walkthrough using the current visibility solution. Figure 7 shows a small subset of the samples generated by the AGVS algorithm and demonstrates how the samples adapt to the visibility structure of the scene. Figure 2 shows an example how AGVS could be used for interactive visibility analysis. The accompanying video shows a real-time capture of such an analysis using our tool.

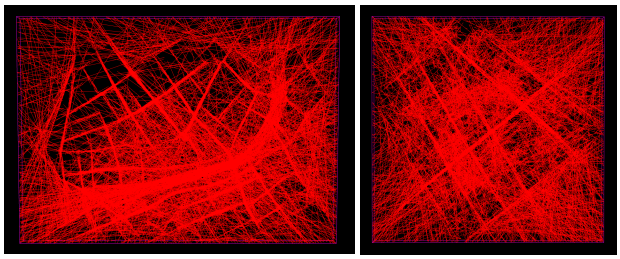


Figure 7: *Subset of rays generated by the AGVS algorithm after about 20M samples have been cast. Left: Vienna, Right: Pompeii.*

Progressive sampling efficiency. We are not aware of any published method that is able to compute global visibility progressively. Therefore as a basic reference technique (REF) we use a method which randomly selects a view cell and then casts a ray from a random point inside the view cell into a random direction. If an object is hit, we add this object to the PVS of the view cell. Similar to our new method, REF is fully progressive; as we cast more and more rays the PVSs become more accurate.

As a second reference we implemented a GPU-based method (GPU-REF), which is similar to the techniques used in industry [Hastings 2007]. This progressive method selects a random view point, renders the scene into a z-buffer of $1{,}024\times1{,}024$ pixels for six directions of the surrounding cube, and uses occlusion queries to discover visibility of the objects. NVIDIA's depth clamping functionality was used to avoid problems with the near clip plane. We evaluated the GPU-REF method on all but the Boeing 777 scene, as our GPU renderer is in-core and we can only run it on desktops, which do not have sufficient memory to store the Boeing 777 scene.

As a third reference, we use a global method that uses only the view space-direction distribution for sampling (SGVS for stationary global visibility sampling), similar to [Mattausch et al. 2006]. SGVS shows how much of the benefit of our method is due to the global view cell evaluation (when comparing to REF), and how much additional benefit is gained by the adaptive mixture distribution and the mutation-based sampling strategies (when comparing to AGVS).

Convergence. Figure 8 compares the convergence behavior of the four methods on our test scenes. We show the progressive evolution of the PVS size (measured as the average number of triangles in a PVS) with respect to the running time. Using the PVS size to compare the methods is a compromise: ideally, one would like to compare to the exact solution, and plot $|PVS|/|EVS|$. However, we are not aware of any algorithm that can provide an exact solution for the scenes we tested in reasonable time, and as shown by Wonka et al. [Wonka et al. 2006], the exact solution suffers from numerical errors similarly to the ray tracing solution. Therefore we believe that PVS sizes are a good measure for comparing the relative efficiencies of algorithms. As an alternative evaluation, we provide a comparison of pixel error later on. Note that we do not use visibility filtering in this test, so the AGVS curve does not overestimate the PVS.

The most important observation from these plots is that our proposed AGVS method is much more efficient in determining the PVS than the other methods. The AGVS method provides a PVS significantly larger than SGVS and GPU-REF, and 2-4 times larger than the one provided by REF. Note however that the factor between the PVSs provided by the method is *not* an indicator of the speedup or benefit of AGVS. Instead, one has to compare how much time it takes the methods to achieve the *same PVS size*. The curves clearly show that SGVS consistently takes more than one order of magnitude longer than AGVS to obtain the same average PVS size. When compared to REF and GPU-REF, AGVS provides up to two orders of magnitude speedup. Note that the positions marked by dashed lines in the plots correspond to a state where AGVS is still in a phase of steep increase, so if we extrapolated the curves to show the same comparison for a later phase, the benefit of AGVS over REF and GPU-REF would probably exceed two orders of magnitude. The figure shows clearly that convergence of the reference method compared to AGVS is so slow that it is almost imperceptible in the duration we have run the tests.

An interesting observation follows from the comparison to GPU-REF. Although in the same time the GPU-REF method is able to process more than one order of magnitude more samples than AGVS, the overall convergence of GPU-REF is significantly slower. The samples produced by GPU-REF are highly correlated (6M samples always intersect at a common view point) and thus they do not easily discover some difficult visibility interactions such as objects occluded from a view cell by nearby occluders. This can also be observed in the pixel error analysis shown later.

Influence of distributions. Figure 9, left, shows the mixture of distributions actually selected by the adaptive mixture distribution algorithm (Section 4.2) according to previous success rates (Vienna

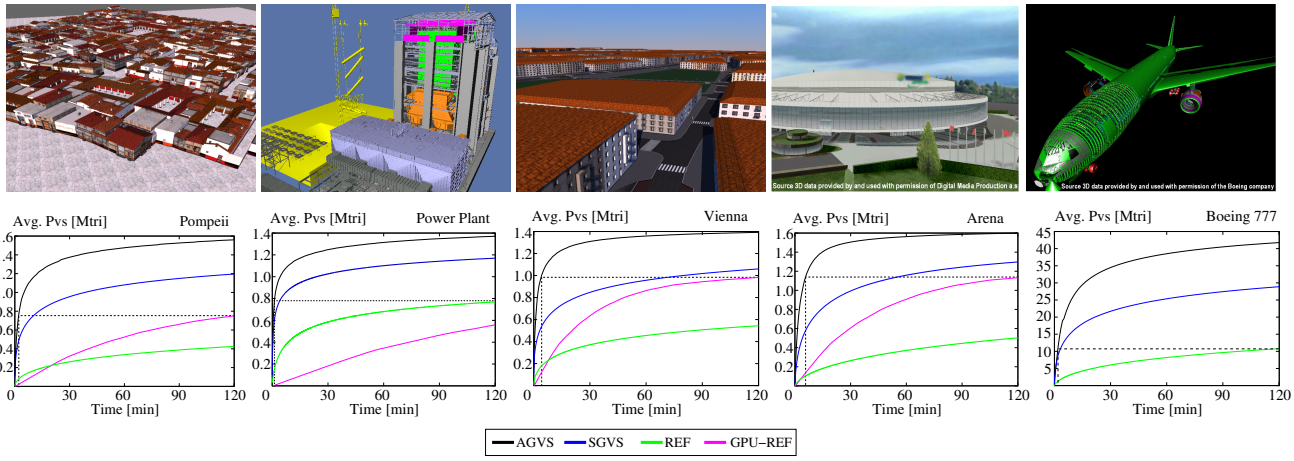


Figure 8: *The tested scenes (top row) and the analysis of the convergenece of the tested methods (bottom row). The dashed lines show reference points for comparing the benefit of AGVS over the better reference method (REF or GPU-REF), i.e., the factor of time needed to achieve the same PVS. Visibility filtering is* not *used in this test.*

model). The mutation-based strategies (shown as one curve) are the most important, closely followed by the stationary two-point distribution, which works well because it adapts to the shape of the scene.

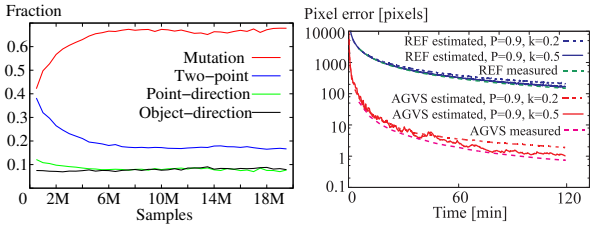


Figure 9: *Left: Ratio of distributions actually selected by the adaptive mixture distribution. Right: Pixel error estimation. The plot shows the estimated pixel error in a city scene in comparison with pixel error measured using 20,000 random views (1,024×1,024 pixels each).*

Pixel error estimator. A typical curve corresponding to the pixel error estimator described in Section 4.5 is depicted in Figure 9, right. For comparison we also show the reference error curve measured using offline pixel error evaluation using 20,000 cameras. Please note that the proposed estimator comes practically for free already during the computation, whereas the camera-based evaluation is a costly computation applied as a postprocess.

The pixel error estimator for the view point-direction distribution closely follows the measured behavior. The pixel error estimator for AGVS tends to be conservative as larger time windows must be used (the number of uniform samples cast is low as these samples are not very succesful in discovering new PVS entries). Note that using smaller k smooths the estimator, but makes it even more conservative as larger time windows extending to the past are used.

Practical behavior. For demonstrating the practical influence of the average PVS sizes shown above, we have measured the average and maximum pixel errors (i.e., number of incorrect pixels) for a subset of viewpoints (more specifically, an actual walkthrough). Note that there is as yet no feasible way to generate the "actual" PVS as a reference: exact visibility solvers do not work on scenes of this complexity, while the convergence of a brute-force reference solution is simply too slow to be feasible.

Figure 10 shows this for the Vienna scene, with pixel error evaluated using a walkthrough consisting of 1,282 view points. The first part of the walkthrough corresponds to walking on a street, the second part is a flyover sequence. After two hours of sampling, the average pixel error among all tested view points drops to 10 pixels on a 1,024×1,024 screen, while the corresponding maximum pixel error for the whole scene drops to 501 pixels. Note that these values were reached even without the visibility filter. The application of the visibility filter would typically reduce the error by an order of magnitude at the cost of increasing the PVS size by 50 to 150%. The increase of PVS size depends on the scene and the object representation. Although the visibility filter adapts to the sampling density (it uses a smaller kernel when more samples have already been cast), it typically does not converge to the unfiltered solution: if the objects overlap in space, filtering even with no extension at all adds overlapping objects, including those which need not be visible. Since the PVS increase can reduce the performance of the target application, we suggest to use the visibility filter only in the early stages of the computation to compensate for larger visibility errors. The decision for using/not-using the visibility filter can also be based on the estimated pixel error.

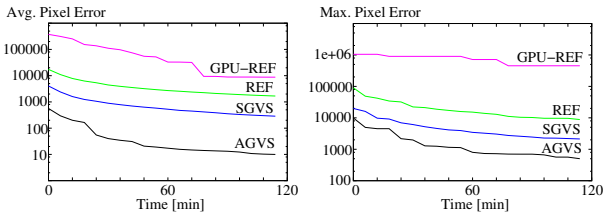


Figure 10: *Pixel error measurement for Vienna scene on a resolution of 1,024×1,024.*

Comparison with Guided Visibility Sampling. Guided Visibility Sampling (GVS) [Wonka et al. 2006] is an aggressive visibility algorithm that also uses ray tracing to sample visibility. A comparison between AGVS and GVS is difficult because GVS does not

work on object granularity and AGVS does not work on triangle granularity.

We applied GVS on the view cells corresponding to the walkthrough sequence used for the AGVS pixel error evaluation and converted the resulting triangle-based PVSs to the object-based representation used by AGVS. Note that the conversion of GVS results to object-based PVSs inflates the number of triangles in the PVS by about 50% to 100%. There were 33 view cells visited by the walkthrough. GVS took 17 minutes to calculate these 33 view cells using about 500M samples. The termination criterion in GVS was set to a threshold of 50 triangles per 1M rays [Wonka et al. 2006]. The error evaluation of the GVS solution lead to an average pixel error of 5.4 and a maximum pixel error of 424. Extrapolated to 8,192 view cells, GVS would have taken 62 hours to calculate visibility for the whole scene.

For pure AGVS, we have higher pixel errors after 17 minutes of computation (average 123 and maximum 4,519 – see Figure 10), but at this time the PVSs for all 8,192 view cells are already available. After two hours of computation the average pixel error of AGVS drops down to 10 and the maximum pixel error to 501. This shows that AGVS can also compete with GVS for applications where a saturated PVS is required.

Comparison with online occlusion culling. We implemented support for PVSs in a rendering engine. For each view point we tag objects in the corresponding PVS and use hierarchical view frustum culling to remove objects outside of the view frustum. As a reference for the comparison we used the CHC++ algorithm [Mattausch et al. 2008] – a state of the art online occlusion culling algorithm based on hardware occlusion queries. A render time comparison between View Frustum Culling (VFC), View Frustum Culling + PVSs (VFC+PVS), and CHC++ can be seen in Figure 11, for a walkthrough in Vienna and a walkthrough in Powerplant.

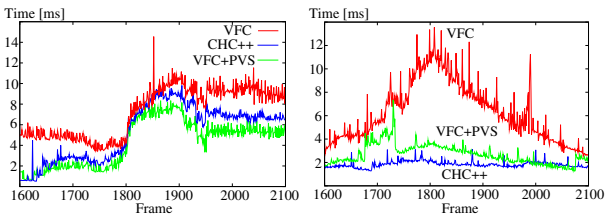


Figure 11: *Comparison of View Frustum Culling (VFC), CHC++, and View Frustum Culling + PVSs (VFC + PVS) in a walkthrough in Vienna (left) and Powerplant (right).*

We tested the algorithms on an Intel Core 2 2.66GHZ quad-core CPU (using only one core) and an NVIDIA GeForce 280 GTX GPU. For all tested scenarios both VFC+PVS and CHC++ are generally faster than plain VFC, while there are passages where either CHC++ or VFC+PVS performs better. Note that CHC++ usually renders fewer triangles since it computes visibility for a set of image samples from a given view point, whereas a PVS is computed for all view points in a view cell, thus overestimating each individual view point. It is an interesting topic for future work to analyze whether splitting view cells on demand during visibility sampling could improve this overestimation. On the other hand there is an additional cost associated with CHC++, which for some view points becomes more important than visibility overestimation associated with the view cell. For a reasonable number of view cells, we observed that rendering using precomputed PVSs is competitive to state-of-the-art online culling algorithms if a slight pixel error is tolerable. Visibility preprocessing is usually the best choice when visibility has to be known beforehand or a bound on visibility has to be guaranteed. We are also interested in combining precomputed visibility with occlusion queries in future work.

8 Discussion

Memory consumption. One advantage of our algorithm is that it maintains only a small and easily controllable state. In contrast to conservative and exact methods, no ray space or scene data structure is created to represent visibility. The memory consumption therefore depends on two main factors: 1) The total memory required for the complete visibility solution (i.e., all PVS entries for all view cells). This memory is influenced by the visibility structure of the scene (which cannot be changed) and by the view cell subdivision and object clustering (which can be changed). For example, in a scene with 4,000 view cells and 1,000 objects visible per view cell on average, the data structure would require 16MB of main memory at convergence (using 32 bit object identifiers). There are also algorithms for compressing PVS data if this becomes an issue [van de Panne and Stewart 1999]. 2) The memory required by the geometry of the scene and the ray tracer acceleration data structure. This is the only factor that limits the type of scene our algorithm can be applied to, and depends on the ray tracer being used. The Boeing 777 model, for example, requires 28GB of main memory including the BVH. For larger models, out-of-core ray tracing is an option that needs to be evaluated further.

Visibility coherence. Some recent algorithms have attempted to exploit the spatial coherence between objects [Nirenstein and Blake 2004; Laine 2005]. The fundamental difference is that these algorithms propagate occlusion, whereas our algorithm propagates visibility. However, neither hierarchical [Nirenstein and Blake 2004] nor sequential [Laine 2005] propagation of occlusion information lends itself to progressive computation, as this requires a complete visibility solution for a particular region to establish occlusion, whereas a single ray suffices to establish visibility. Note that algorithms that construct the PVS top-down proceed in a depth-first manner and are thus not progressive in our sense, while breadth-first traversal is most likely infeasible due to memory requirements.

Object-level visibility. Since AGVS calculates all view cells simultaneously, it is sensitive to the total number of objects in the scene, both in memory consumption for all PVSs, and in convergence speed. Therefore, a triangle-level solution as provided by Guided Visibility Sampling is typically not feasible. However, any application that requires a global visibility solution will need to compress triangle-level visibility into object-level visibility because of the high memory overhead of triangle-level PVSs. For real-time rendering applications, object-level visibility is also required because current graphics hardware works best on batches of triangles and not on individual triangles.

Accuracy. Wonka et al. [2006] discussed the issue of accuracy in the context of visibility processing and claim that accuracy is limited by numerical precision even for exact algorithms. Specific limitations concern the tendency of current ray tracers to shoot "through" an object if a ray pierces an edge that does not lie on the silhouette. We have found that in scenes consisting of closed objects, this artifact can be reduced by discarding rays that hit a backfacing triangle, but the development and evaluation of robust ray casters remains a topic of future work.

9 Conclusion

We described a new visibility algorithm that computes global visibility in the scene by calculating PVSs for all view cells simultaneously in a progressive fashion. The main contribution of the paper is a set of adaptive sampling strategies based on ray mutations that exploit the spatial coherence of visibility. The mutation based distributions are mixed with other heuristic distributions using the adaptive mixture distribution technique. We have shown that our algorithm achieves more than an order of magnitude speedup compared to sequential per-view cell visibility computation. We believe that the Adaptive Global Visibility Sampling algorithm breaks new grounds in the applicability of visibility algorithms. In addition to making preprocessed visibility feasible for everyday use, it also enables new applications like visibility analysis for level design.

Acknowledgments

This work has been supported by the Austrian Science Fund (FWF) contract P21130-N13, Czech MŠMT programs no. LC-06008 and MSM 6840770014, the Aktion Kontakt 2009/6, and the NSF.

References

AIREY, J. M., ROHLF, J. H., AND BROOKS, JR., F. P. 1990. Towards image realism with interactive update rates in complex virtual building environments. In *Computer Graphics (1990 Symposium on Interactive 3D Graphics) 24*, 2, 41–50.

BITTNER, J., WONKA, P., AND WIMMER, M. 2001. Visibility preprocessing for urban scenes using line space subdivision. In *Proc. of Pacific Graphics '01*, 276–284.

BITTNER, J. 2003. *Hierarchical Techniques for Visibility Computations*. PhD thesis, Czech Technical University in Prague.

COHEN-OR, D., FIBICH, G., HALPERIN, D., AND ZADICARIO, E. 1998. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum (Eurographics '98) 17*, 3, 243–254.

COHEN-OR, D., CHRYSANTHOU, Y. L., SILVA, C. T., AND DURAND, F. 2003. A survey of visibility for walkthrough applications. *IEEE Trans. on Visualization and Computer Graphics 9*, 3, 412–431.

DUGUET, F., AND DRETTAKIS, G. 2002. Robust epsilon visibility. *ACM Transactions on Graphics 21*, 3, 567–575.

DURAND, F., DRETTAKIS, G., THOLLOT, J., AND PUECH, C. 2000. Conservative visibility preprocessing using extended projections. In *Proc. of SIGGRAPH '00*, 239–248.

DUTRÉ, P., BALA, K., AND BEKAERT, P. 2003. *Advanced Global Illumination*. AK Peters.

GOLDBERG, D. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

GOTSMAN, C., SUDARSKY, O., AND FAYMAN, J. A. 1999. Optimized occlusion culling using five-dimensional subdivision. *Computers and Graphics 23*, 5, 645–654.

HASTINGS, A., 2007. Occlusion systems. Insomniac Games Tech Presentation. http://www.insomniacgames.com/tech/articles/1107/occlusion.php.

HAUMONT, D., MÄKINEN, O., AND NIRENSTEIN, S. 2005. A low dimensional framework for exact polygon-to-polygon occlusion queries. In *Rendering Techniques '05*, 211–222.

KOLTUN, V., CHRYSANTHOU, Y., AND COHEN-OR, C.-O. 2001. Hardware-accelerated from-region visibility using a dual ray space. In *Rendering Techniques '01*, 205–216.

LAINE, S. 2005. A general algorithm for output-sensitive visibility preprocessing. In *Proc. of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 31–40.

LEPAGE, G. 1980. Vegas: An adaptive multidimensional integration program. Tech. Rep. CLNS-80/447, Cornell University.

LEVOY, M., AND HANRAHAN, P. 1996. Light field rendering. In *Proc. of SIGGRAPH '96*, 31–42.

LEYVAND, T., SORKINE, O., AND COHEN-OR, D. 2003. Ray space factorization for from-region visibility. *ACM Transactions on Graphics 22*, 3, 595–604.

MATTAUSCH, O., BITTNER, J., AND WIMMER, M. 2006. Adaptive visibility-driven view cell construction. In *Rendering Techniques '06*, 195–206.

MATTAUSCH, O., BITTNER, J., AND WIMMER, M. 2008. CHC++: Coherent hierarchical culling revisited. *Computer Graphics Forum (Eurographics '08) 27*, 3 (Apr.), 221–230.

NIRENSTEIN, S., AND BLAKE, E. 2004. Hardware accelerated visibility preprocessing using adaptive sampling. In *Rendering Techniques '04*, 207–216.

NIRENSTEIN, S., BLAKE, E., AND GAIN, J. 2002. Exact from-region visibility culling. In *Rendering Techniques '02*, 191–202.

PITO, R. 1999. A solution to the next best view problem for automated surface acquisition. *IEEE Trans. Pattern Anal. Mach. Intell. 21*, 10, 1016–1030.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. *ACM Transactions on Graphics 24*, 3, 1176–1185.

SCHAUFLER, G., DORSEY, J., DECORET, X., AND SILLION, F. 2000. Conservative volumetric visibility with occluder fusion. In *Proc. of SIGGRAPH '00*, 229–238.

SHIRLEY, P., SLUSALLEK, P., WALD, I., MARK, B., STOLL, G., AND MANOCHA, D. 2006. SIGGRAPH 2006 course 4, State of the art in interactive ray tracing.

TELLER, S. J., AND SÉQUIN, C. H. 1991. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics (Proc. of SIGGRAPH '91)*, 61–69.

THOMPSON, S. K., AND SEBER, G. A. F. 1996. *Adaptive Sampling*. Wiley.

VAN DE PANNE, M., AND STEWART, A. J. 1999. Effective compression techniques for precomputed visibility. In *Rendering Techniques '99*, 305–316.

VEACH, E., AND GUIBAS, L. J. 1997. Metropolis light transport. In *Proc. of SIGGRAPH '97*, 65–76.

WILSON, A., AND MANOCHA, D. 2003. Simplifying complex environments using incremental textured depth meshes. *ACM Transactions on Graphics 22*, 3, 678–688.

WONKA, P., WIMMER, M., AND SCHMALSTIEG, D. 2000. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques '00*, 71–82.

WONKA, P., WIMMER, M., ZHOU, K., MAIERHOFER, S., HESINA, G., AND RESHETOV, A. 2006. Guided visibility sampling. *ACM Transactions on Graphics 25*, 3, 494–502.

Appendix H

Visibility-driven Mesh Analysis and Visualization through Graph Cuts

Zhou, K. - Zhang, E. - Bittner, J. - Wonka, P.: Visibility-driven Mesh Analysis and Visualization through Graph Cuts. IEEE Transactions on Visualization and Computer Graphics. 2008, vol. 14, no. 6, p. 1667-1674. ISSN 1077-2626. **IF=2.215**

Visibility-driven Mesh Analysis and Visualization through Graph Cuts

Kaichi Zhou, Eugene Zhang, Jiří Bittner, and Peter Wonka

Abstract— In this paper we present an algorithm that operates on a triangular mesh and classifies each face of a triangle as either inside or outside. We present three example applications of this core algorithm: normal orientation, inside removal, and layer-based visualization. The distinguishing feature of our algorithm is its robustness even if a difficult input model that includes holes, coplanar triangles, intersecting triangles, and lost connectivity is given. Our algorithm works with the original triangles of the input model and uses sampling to construct a visibility graph that is then segmented using graph cut.

Index Terms—Interior/Exterior Classification, Normal Orientation, Layer Classification, Inside Removal, Graph Cut.

1 Introduction

We address the following problem: given a model as a potentially unstructured set of triangles $t_i \in T$ where each triangle t_i consists of two faces $t_{i,1}$ and $t_{i,2}$. As output we want to compute a classification of all triangle faces $t_{i,j}$ into either inside or outside. Such an approach has several applications and we will demonstrate three in this paper: the inside removal of architectural and mechanical models for faster visualization, normal orientation and layer-based visualization of multiple layers of geometry using transparency.

We believe that previous work does not use visibility to its full potential so that classification errors are likely in more difficult models. There are two existing approaches to using visibility for inside outside classification. 1) Rays are sampled from an outside bounding volume to classify geometry as outside (e.g. [4, 16], see Fig. 1 left). This approach has difficulties with cracks and with the fact that parts of the outside surface might not be visible from an enclosing bounding volume. 2) Rays are sampled from the outside to stab the whole model. The inside-outside classification changes with each intersection (e.g. [16], see Fig. 1 middle). This approach has also some difficulties with cracks, double sided triangles, self intersecting triangles, and coplanar triangles. In contrast, we observe that any ray path can be used to propagate inside-outside classifications, see Fig. 1 right.

Our solution is to use visibility analysis to establish connections between entities that we call half-space nodes. A half-space node can correspond to a single half-space point or a (possibly infinite) set of half-space points. A half-space point is an oriented sample on a triangle and consists of a point p_i in R^3 and hemisphere centered at p_i. The orientation of the hemisphere is decided by the normal of a triangle face at p_i. Visibility relationships between half space points are established through sampling using ray casting. The details of our algorithm include a solution on how and where to create half space nodes and how to sample rays to establish visibility relations. The second part of our approach is a classification using iterative graph cut. The main contributions of our work are the following:

- We propose a model preprocessing algorithm that can classify triangle faces into inside or outside even if the input contains coplanar polygons, self-intersections, two-sided triangles and cracks.
- We are the first to propose semi-automatic extensions to inside outside classification to increase robustness in case the automatic method is unspecified or the errors in the model are above a user defined tolerance.

- *Kaichi Zhou graduated from Arizona State University and is now with NVIDIA, E-mail: kaichi.zhou@gmail.com.*
- *Eugene Zhang is with Oregon State University, E-mail: zhange@eecs.oregonstate.edu*
- *Jiří Bittner is with Czech Technical University in Prague, E-mail: bittner@fel.cvut.cz*
- *Peter Wonka is with Arizona State University: E-mail: pwonka@gmail.com*

Manuscript received 31 March 2008; accepted 1 August 2008; posted online 19 October 2008; mailed on 13 October 2008.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

1.1 Challenges

There are several geometric configurations that we want to consider in this paper as explained in the following:

One sided and two sided triangles: A one sided triangle has one side on the inside and one side on the outside while a two sided triangle has both faces on the outside (or inside). The main difficulty is to ensure that the algorithm can detect holes and does not classify triangle faces as outside that are only visible through an unwanted crack or hole in the model (see Fig. 2 for an example illustration).

Intersections and coplanar triangles: The algorithm should be robust in the presence of self intersections and coplanar triangles (see Fig. 3 left and right).

User input: It is unlikely that all models can be handled without some user input. In many cases, such as with terrains the question of what is inside and what is outside is actually a modeling problem that requires user input (see Fig. 3 right). We want to make use of minimal user input to clarify ambiguous configurations and to improve robustness in difficult cases.

Inside-outside definitions: There are two fundamentally different definitions of an inside-outside classification: *view-based* and *object-based*. The *view-based* definition considers all triangle faces as outside that are seen along a straight line from a bounding region, e.g. sphere, around the object. The *object-based* definition considers all triangle faces as outside that can be seen along a poly-line from the bounding region. The line or poly-line cannot intersect other triangles or pass through cracks in the model. Please note that only the *object-based* definition establishes an inherent property of the object, while the *view-based* definition produces different results for different viewing regions. In this paper we focus on the *object-based* definition. The *view-based* definition is a from-region visibility problem that could be addressed with our previous work [21, 20] as a starting point.

1.2 Related Work

Surface based model repair: Surface based model repair algorithms operate directly on the input mesh and fix model errors by local modifications to the mesh. A large class of methods fixes the model errors by either stitching the boundaries of the patches together or filling the holes by inserting additional geometry [3, 1, 9, 12]. Murali and Funkhouser [15] construct a BSP tree using the input triangles and then use linear programming in order to classify the cells of the BSP either inside or outside, remove cracks and fill holes. However due

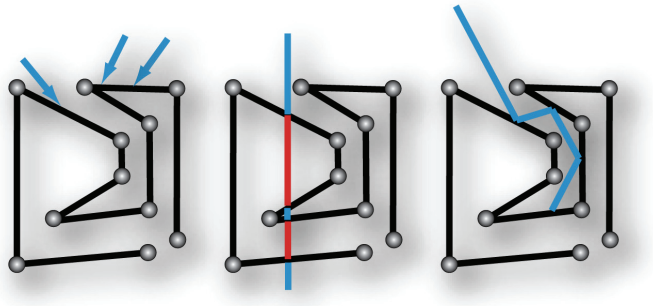


Fig. 1. Triangles (line segments) are shown in black, vertices as grey spheres, an outside classification is shown in blue and an inside classification is shown in red. Left: Classification of outside geometry by sampling visibility from a bounding sphere around the scene. Middle: stabbing the object along straight lines and alternating outside inside classification. Right: propagating inside and outside classification along an arbitrary path.

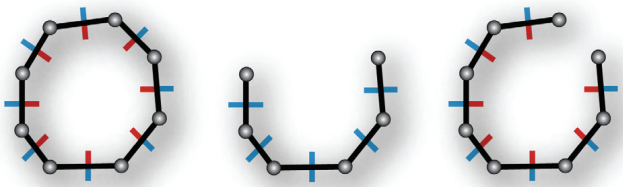


Fig. 2. Triangles (line segments) are shown in black, vertices as grey spheres, an outside classification is shown in blue and an inside classification is shown in red. Left: a typical example of a one sided object. Middle: a cup where each triangle has two outside faces. Right: this is a tricky example that could have multiple interpretations. We show the interpretation as one sided object with a hole.

to the BSP and linear programming the application of this approach to large models is costly and memory demanding. Our algorithm is closely related and partially inspired by the paper of Borodin et al. [4]. This paper also proposes to use visibility sampling for mesh analysis, but there are three issues that we want to improve upon: 1) The algorithm is not able to cope with coplanar polygons and intersections. 2) the algorithm computes neither a view-based nor an object-based inside-outside classification, but some mixed form. 3) The algorithm does not have a mechanism to correct sampling errors due to the ray tracer. This includes the omission of cosine-based sampling suggested by Zhang and Turk [21].

Volumetric model repair: In recent years a significant attention has been paid to techniques which repair a polygonal model by using an intermediate volumetric representation [16, 10, 2]. Nooruddin and Turk [16] convert the model into a volumetric grid. They perform inside outside classification of the cells of the grid by using ray casting. The grid is processed using morphological operations. Finally, the model is converted back to a polygonal representation. Ju [10] extends this approach by using an octree instead of a regular grid. This method uses a more efficient local approach for the inside outside classification of the cells of the octree. For the conversion to the polygonal model he suggests to use dual contouring which is able to better preserve sharp features of the model. Bischoff et al. [2] proposed a method which is able to robustly handle interpenetrating geometry or dangling triangles. This method also uses octree based voxelization, which is followed by hole filling using morphological operations and an extension of the dual contouring [11]. The common problem of volumetric methods is that they perform low pass filtering of the input geometry. Therefore they have problems with preserving sharp features and thin structures of the model.

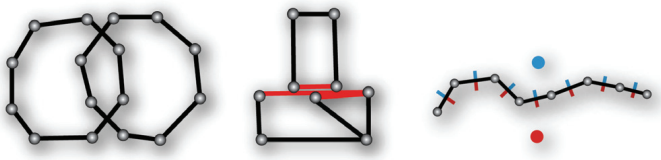
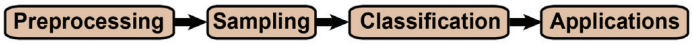


Fig. 3. Left: intersecting lines segments (triangles). Middle: a simple object showing three coplanar line segments (triangles). Right: A terrain has no inherent orientation. The user can specify the classification of two points in space to define inside and outside. Outside points are shown in blue and inside points in red. The algorithm suggested in this paper can correctly classify the triangles that are non-intersecting and non-coplanar. On the intersecting and coplanar triangles there would be a number of seed points (depending on the setting) that are classified as inside or outside.

1.3 Overview

The pipeline of our algorithm consists of four stages (see below):



Preprocessing: In the preprocessing step we load a model as triangle soup. In the basic version of the algorithm we mainly use three preprocessing steps. We build a kd-tree, we mark triangles that intersect other triangles, and we mark triangles that are coplanar with other triangles. Alternatively, we also use three more sophisticated preprocessing techniques that are optional. We can make use of connectivity and geometry information to construct clusters of triangles, we can compute exact triangle intersections, and we can remove coplanar triangles. The latter two computations are fairly difficult and not as robust as the other parts of the pipeline. For example, triangle intersections can often lead to many small additional triangles that are a disadvantage for many applications. See section 2 for details.

Sampling: In the sampling stage we create half-space nodes on triangle surfaces and shoot rays to sample visibility connections to other half space nodes. The methodology combines ideas from visibility, geometry, and global illumination to obtain a robust sampling strategy for the creation of half space nodes and the generation of rays. See section 3 for details.

Classification: The classification step analyzes the graph using a max-flow, min-cut algorithm to classify half space nodes as either inside or outside. Additionally, we provide a user input to set some global parameters of the classification, or to locally refine the computation in case of difficult geometric configurations. See section 4 for details.

Application and Results: The algorithm can be used for normal orientation, inside removal, or layer classification. We present a few more details on how to fine tune the pipeline for these applications in section 5 including a variety of results on selected example models.

2 Preprocessing

The input to our algorithm is a set of n_0 triangles t_i, with $1 \leq i \leq n_0$. In this section we explain six preprocessing steps: 1) kd-tree construction, 2) intersection testing, 3) coplanar testing, 4) intersection retriangulation, 5) coplanar triangle removal, and 6) patch clustering. The first three steps are required for all models and the second three are optional.

Our main philosophy is to establish a conservative and robust model processing algorithm. We found that the two optional preprocessing steps intersection retriangulation and coplanar triangle removal give undesirable results in many cases. The main focus of this paper is therefore to establish robustness despite geometric errors and inaccuracies rather then to fix these problems. A general problem with intersection retriangulation is the long implementation time for a correct algorithm and the many (often very thin) triangles that can be generated in the process (see Fig. 4 for an example intersection retriangulation). Coplanar triangle removal shares the same problems of intersection retriangulation. Additionally, it is unclear how coplanar triangles can be removed in textured models, because there are multiple textures or

texture coordinates to choose from. As default we will assume that only the required steps are performed and will note additional optional preprocessing steps for each model.

Kd-tree construction: All triangles in the model are sorted into a kd-tree to accelerate ray casting. The kd-tree construction and the ray tracer are an improved version of multi-level ray tracing [17]. The kd-tree construction takes under two minutes for the largest model in the paper.

Intersection Test: The intersection test is performed for each triangle $t_i \in T$ and outputs a flag $inters_i \in \{true, false\}$ that is *true* if the triangle intersects another triangle and is $false$ if the triangle does not intersect another triangle. The algorithm has to be conservative, i.e. two non-intersecting triangles can be incorrectly classify as intersecting but not the other way around. The reason for this conservative strategy is that the subsequent sampling and classification steps more carefully analyze intersecting triangles. Therefore, the first type of misclassification does not lead to any problems. We proceed as follows. We first use the kd-tree to select a list of intersection candidate triangles. We can find a set of candidate triangles $CSet_i$ by computing the bounding box B_i of triangle t_i and then taking all triangles that are stored with the leaf nodes of the kd-tree that intersect the bounding box B_i. For each triangle $t_j \in CSet_i$ we compute an intersection using the algorithm proposed by Moeller [13]. Alternate intersection routines are described in [14] chapter 13. We choose the first algorithm, because we found it easier to modify to make it conservative. The main idea of the conservative intersection tests is to introduce ε thresholds. We omit a very detailed description because it would be very lengthy due to many special cases and the ε intersection algorithm is not a contribution of our paper. Some of the special cases arise due to degenerate intersections and overcoming floating point limitations. Please note that we do not consider triangles sharing a vertex or an edge as intersecting. These cases are explicitly excluded in the implementation.

Coplanarity Test: The conservative coplanarity test for triangles is performed for each triangle t_i and outputs a flag $cpl_i \in \{true, false\}$ that is *true* if the triangle is coplanar to another triangle and is $false$ otherwise. The test is conservative because we use ε thresholds as in the intersection computation. The algorithm proceeds as follows. We reuse the candidate set $CSet_i$ to test each triangle $t_j \in CSet_i$. We compute the angle α between the plane containing t_i and the plane containing t_j with a normal vector dot product. If α indicates parallel plane orientation we proceed to test the distance d_{ij} between the two closets points on t_i and t_j. If the distance is within an ε threshold we finally project t_i into the plane of t_j and use a two-dimensional version of the triangle intersection test [13]. If the triangles intersect they are both marked as coplanar.

Triangle Clustering: The idea of this stage is to cluster triangles that can be classified together. The clustering algorithm assumes that all connected triangles are part of orientable surface patches. The clustering algorithm is a simple greedy algorithm that needs two main parts: the clustering algorithm itself and a preprocessing algorithm that marks each triangle edge with a flag $\in true, false$ that indicates if triangles that share this edge can be clustered. In the following we first explain how to compute the edge flag and then give the algorithm outline. The edge flag is set to *true* by default and then we set the flag to $false$ in the following cases: 1) the edge belongs to a triangle with $inters_i = true$ or 2) $cpl_i = true$; 3) the edge is non-manifold, i.e. it is shared by more then two triangles, and 4) the edge belongs to a triangle that is too close to another non-adjacent triangle.

The clustering algorithm marks all triangles of the model as not visited using a boolean flag per triangle. Iteratively, the first not visited triangle is selected to start a new cluster. The cluster is expanded by adding other triangles that share edges marked *true* using breath first search. If there is only one triangle in a cluster we assign the cluster id $cluster_i = -1$ and otherwise we set the cluster id of all triangles in the cluster using a counter to ensure unique cluster ids.

Intersection Re-triangulation: The idea of intersection re-triangulation is to re-triangulate triangles so that no triangle is intersected by another one. While the output of this algorithm is a very helpful simplification of a general input, we believe that the actual use of the algorithm is controversial, because it can create many additional triangles and it is difficult to implement correctly. The triangle intersection algorithm for a triangle t_i proceeds similar to the intersection test and first computes intersection candidates and then triangle intersections. For each triangle intersection we store the intersection lines l_j in a set $LineS_i$. After all intersection lines are computed we retriangulate t_i using constrained Delauney triangulation [18]. More robust triangulations can be computed with the CGAL library [6]. See Fig. 4 for an examples.

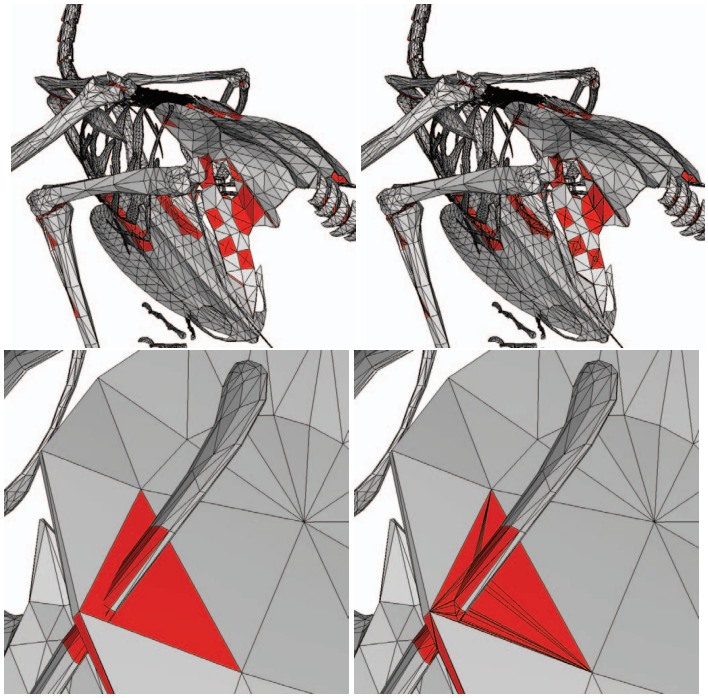


Fig. 4. The images on the left show the bird model before triangle splitting. The images on the right show the bird model after splitting. All triangles considered for splitting are shown in red. Please note the high number of thin polygons in the closeup view.

Coplanar Removal: The coplanar triangle removal uses the result of the coplanarity test. Coplanar triangles are projected into one plane and the resulting polygon is retriangulated.

Output of Preprocessing: At the end of preprocessing we obtain a set of n triangles t_i, with $1 \leq i \leq n$. If steps four and five are not performed $n = n_0$. Additionally, each triangle i has a flag $cpl_i \in \{true, false\}$ to denote if the triangle has other coplanar triangles and a flag $inters_i \in \{true, false\}$ to denote if the triangle is intersecting other triangles. If triangle clustering is used we store a cluster identifier $cluster_i$ with each triangle.

3 Sampling

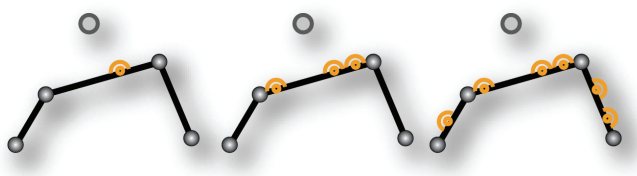


Fig. 5. Three cases of half-space nodes. Left: a single half-space point. Middle: the set of half-space points on one triangle face. Right: all half-space points on a cluster of triangle faces.

The goal of the sampling stage is to generate a visibility graph $G(V,E)$ consisting of *half-space nodes* $hsn_i \in V$, *virtual nodes* $\in V$, and edges $\in E$ that encode visibility connections between the nodes.

We divide the process into two steps: geometry sampling for node generation and visibility sampling. In principal the nodes in the graph correspond to geometry. We want to classify the nodes into inside and outside and then later transfer the node classification back to the geometry. These are then our two fundamental ideas on how to design such a graph:

1. The goal for the node design is to create nodes that correspond to geometry that can receive a consistent classification. This will result in a solution where more nodes are placed in difficult parts and fewer nodes are placed in easy parts of the model. For example, intersecting and coplanar triangles are difficult, because the visibility classification is expected to change within a triangle and we cannot assume that we can classify the faces of such a triangle consistently. On the other extreme, triangles within a triangle cluster will have two sets of faces so that the classification within each of the two sets is consistent.

2. The goal for edge design is to generate edge weights that measure how strong the visibility between two nodes is. Therefore, we propose to use a measure in ray space that corresponds to how many visible rays exist between two nodes.

3.1 Geometry Sampling

We define a *half-space point* as a point in R^3 and a set of viewing directions on a hemisphere Ω. Typically, the half-space point is a sample on a triangle and the hemisphere is defined by the normal vector of one of the two triangle faces.

A *half-space node* corresponds to either a single half-space point or a set of half-space points (see Fig. 5). We use different methods to generate half-space nodes for triangle clusters and the coplanar and intersected triangles:

Triangle Clusters: All the triangles belonging to a triangle cluster give rise to two sets of triangle faces fs_1 and fs_2 that will have the same inside-outside classification. Only two half space nodes are generated for a triangle cluster. One for all half space points on triangle faces in fs_1 and one for all half-space points on triangle faces in fs_2. Note that a single triangle is just a special case of triangle cluster.

Intersected and Coplanar Triangles: A predefined number of sample points are distributed over each intersected triangle. Each point contributes two half space nodes in the graph. We propose three sampling schemes to distribute the points: 1) Uniformly sample the triangle. 2) Uniformly sample the polygonal face, plus sample along the polygon's edges. 3) Compute all the intersection edges on the face (for coplanar faces, first project all the coplanars onto the interested face's plane), triangulate the face and uniformly sample the sub-faces.

The sampling quality increases from the first method to the third, but the time complexity also increases. In our experiments, we found that the second scheme generates good results for most of the models. To avoid potential numerical issues of ray casting, we need to detect whether the selected sample point is within an epsilon proximity of other triangles. For each of the other triangles intersected or coplanar with the current triangle, the distance from the point to its plane is computed and also the point is projected onto its plane to see if the point really falls inside the triangle. The half space point is offset along the normal of the furthest triangle that satisfies the proximity test above. Fig. 6 illustrates the adjustment.

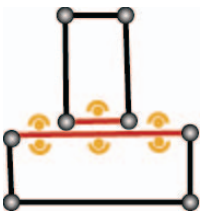


Fig. 6. Coplanar and intersected samples are adjusted to the surface of the furthest triangle that satisfies the proximity test.

Virtual Nodes: The graph always contains at least two virtual nodes, one of which represents the outside and the other represents the inside. Later, we will compute a cut in the graph to compute two disjunct sets of nodes. The set of nodes that contains the *outside node* (*inside node*) will be classified as outside (inside).

The sampling stage first creates the half-space nodes and then stochastically samples visibility between half-space nodes using ray casting. During this process a number of half-space points is generated for each half-space node. The details will be described in the next section.

3.2 Visibility Sampling

The constructed graph of halfspace nodes contains edges with weights. The weight of an edge connecting two halfspace nodes should reflect the amount of visibility between the corresponding patches. In the next section we derive the weight assignment and then we describe how the weights are established using ray casting.

3.2.1 Weight Assignment

We desire that the weight of connection of two patches in the scene reflects the amount of their mutual visibility. Mutual visibility of two patches has been studied intensively in the context of radiosity methods which aim to simulate illumination of diffuse environments. In radiosity methods the visibility between patches drives the amount of power exchanged between them.

Inspired by these methods, we define the weight of an edge connecting two patches (half-space nodes) as the average amount of power transferred between these patches. The power transfer from patch i to patch j is:

$$P_{ij} = A_i B_i F_{ij} \tag{1}$$

where A_i is the area of patch i and F_{ij} is the form factor between patches i and j. Assuming all patches have unit radiosity ($B_i = 1$) we get $P_{ij} = A_i F_{ij}$. The form factor F_{ij} is given by the following equations [7]:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} F(x,y) dA_j dA_i \tag{2}$$

where x and y are points on patches A_i and A_j respectively, dA_i and dA_j are differential surface areas, and $F(x,y)$ is the point-to-point form factor defined as:

$$F(x,y) = \frac{(\Theta_{xy} \odot N_x)(-\Theta_{xy} \odot N_y)}{\pi r_{xy}^2} V(x,y) \tag{3}$$

where Θ_{xy} is a unit direction vector from x to y, $N_x(N_y)$ is the patch normal vector at $x(y)$, r_{xy} is the distance between x and y, and $\odot$ is the inner product. $V(x,y)$ is the visibility function given as:

$$V(x,y) = \begin{cases} 1 & \text{if a ray from point } x \text{ hits point } y \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

From the definition of the form factor, we can see that $A_i F_{ij} = A_j F_{ji}$ and thus $P_{ij} = P_{ji}$.

The form factors can be computed using ray casting as follows: If we originate a ray from a uniformly chosen location on patch i and shoot into a cosine-biased direction (i.e. the probability of the direction depends on the cosine between the direction and the patch normal), the probability of such a ray lands on patch j is equal to the form factor F_{ij} [7]. Thus we can estimate for sufficiently large number of rays: $F_{ij} \approx \widetilde{F_{ij}} = n_{ij}/n_i$ where n_{ij} is the number of rays shot from patch i and landed on patch j and n_i is the total number of rays shot from patch i. From $\widetilde{F_{ij}}$ and $\widetilde{F_{ji}}$ we can compute estimates of power exchange $\widetilde{P_{ij}}$ and $\widetilde{P_{ji}}$. The weight of the link between patches i and j is then defined as the average of the two power exchange estimates:

$$w(i,j) = \frac{1}{2}(\widetilde{P_{ij}} + \widetilde{P_{ji}}) = \frac{1}{2}(A_i \frac{n_{ij}}{n_i} + A_j \frac{n_{ji}}{n_j}) \tag{5}$$

The weight $w(i,j)$ can also be interpreted as the measure of rays which connect the two patches. In order to support this intuition we

provide the following substitution: Assuming uniformly distributed rays we know that $n_i \approx n\frac{A_i}{A}, n_j \approx n\frac{A_j}{A}$, where n is the total number of rays cast and A is the total area of patches. By substitution we can rewrite Eq.5 as:

$$w(i,j) \approx \frac{A}{2}\frac{n_{ij}+n_{ji}}{n} \tag{6}$$

Thus the defined weight corresponds to the ratio of the number of rays connecting the two patches with respect to all rays cast with a constant scale factor $\frac{A}{2}$.

Note that the area of a patch is the sum of the areas of all the triangles belonging to the patch. If a half space node represents a single half space point on a triangle the patch corresponds to a voronoi region on the triangle (the details will be discussed in section 3.2.2). Since the samples are mostly uniformly distributed on the triangles, it is reasonable to assume that each such voronoi region has the same area, which is the triangle's area divided by the number of samples on the triangle.

The rays hitting void from a half space node i contribute to the connection between node i and the outside node. The geometrical meaning of the outside node is the bounding sphere of the scene. The weight is defined as:

$$w_o(i) = A_i(1-\sum_j \frac{n_{ij}}{n_i}) \tag{7}$$

Note that this weight is not reciprocal, since we do not shoot rays from the bounding sphere. Based on geometrical probability, the form factor computation stays valid if we shoot rays from the bounding sphere and count the number of hits landed on each node. Nevertheless, it is inefficient and unnecessary. To embody the *inside node* in a geometric sense is problematic. One of the workarounds is that the user assigns a region on the surface or a volume in space as inside. Then we can shoot rays from such regions to build the connection between half space nodes and the inside node. Without users' guidance, we contribute one half space node's outside weight to its opposite half space node's inside weight: $w_i(opp(i)) = w_o(i)$ where opp() is an operator to find out node i's opposite half space node.

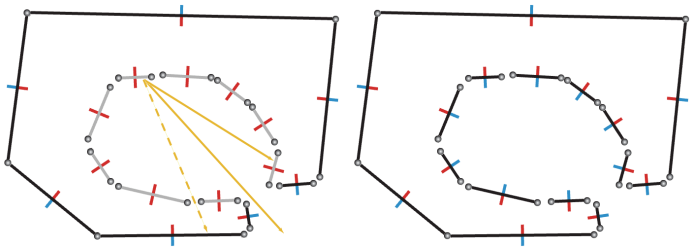


Fig. 7. A 2D illustration of visibility filtering for a cave model. Thicker line segments represent triangle faces in 3D. Thinner blue lines represent outside normals of the faces and red ones represent inside normals. Left: An undesirable classification occurs because rays pass through small cracks between faces. These rays are shown as dotted line. The visibility propagated through the small cracks outweighs the visibility propagated through the main entrance of the cave (shown as straight line). Right: The weights are filtered so that the algorithm gives less weight to smaller cracks resulting in the desired classification.

Due to the imperfection of the geometry, the sampling may incorporate some amount of errors (see Fig. 7). Thus we apply a cos filter to dampen the error fraction. For each node i,

$$w_{max}(i) = max\{w(i,j), w_o(i), w_i(i)\}, \forall j \tag{8}$$

Our directed filter function is defined as:

$$filter(w(i,j)) = \frac{1}{2}(1+\cos(\pi - \pi\frac{w(i,j)}{w_{max}(i)}))w(i,j) \tag{9}$$

3.2.2 Ray Casting

The start locations of the sampling rays are generated uniformly in the region each half space node possesses. Specifically, for a cluster half space node, one triangle of the cluster is selected according to its area and the location is uniformly distributed on the triangle. The directions are sampled on the corresponding hemisphere with a cosine-distributed probability. If a sampling ray shot by node i hits a triangle belonging to a cluster, then a connection is built between the corresponding cluster node and node i.

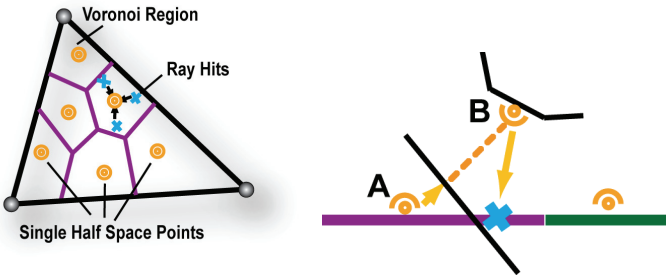


Fig. 8. Left: Voronoi regions formed by single half space nodes in a triangle. Right: A ray hit from Node B is snapped onto Node A, which is not visible from Node B.

If the hit triangle is either an intersected one or a coplanar one, we aim at estimating visibility in such a manner that a single half space point represents a voronoi region on the triangle instead of a single point (see Fig. 8). As we want to avoid complex computations we only try to compute a fast estimate that results in robust visibility classifications. We therefore snap the ray to a half space node j on the triangle closest to the hit point. Since we only originate rays from one location of such node, which is the center of its voronoi region, the form factor integral is biased. Due to the fact that $P_{ij} = A_i F_{ij} = A_j F_{ji} = P_{ji}$, we reduce the variance by using P_{ji} only as the weight between node i and node j instead of $\frac{P_{ij}+P_{ji}}{2}$, if node j is not a single half space point.

Another important special case with snapping occurs when visibility changes between the ray hit and the snapped half space point. In order to discover this case we test reverse visibility by shooting a reverse ray from the node snapped on to the sampling ray's origin. The right of Fig. 8 shows a case where visibility changes in a voronoi region. Node B's sampling ray hits a triangle and the hit is snapped onto Node A, but a reverse ray from Node A to Node B is blocked by other geometry in between. Therefore the ray hit from Node B to A is discarded. Please note that while visibility is symmetric in general, in this case visibility is not symmetric. This is because the starting point of the reverse ray is not the end point of the original ray.

4 Classification

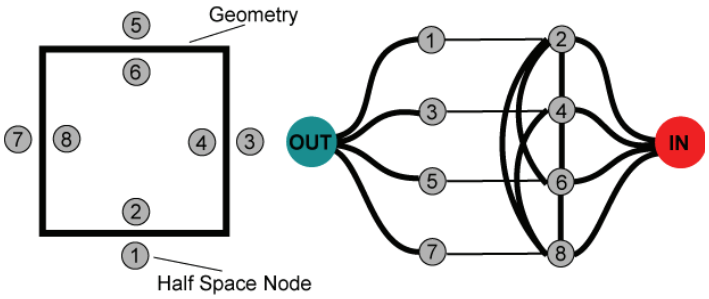


Fig. 9. The visibility graph built from the input geometry on the left.

This section describes how the visibility graph can be processed to obtain an inside and outside classification of each node in the graph. The algorithm is basically a graph clustering or segmentation algorithm. There are several older but powerful algorithms, such as k-

means, single-link, agglomerative, or Ward's algorithm that could be transferred to our problem. See Tan et al.'s book [19] for a review of these techniques. While these algorithms can be fast, they are not guaranteed to produce optimal results. In contrast, many new techniques rely on spectral matrix decompositions.

Our problem is given as follows. The sampling stage generates a visibility graph $G = \langle V, E \rangle$ (see Fig. 9). An inside-outside classification of N half space nodes can be denoted by a binary vector $X = (x_1, x_2, \ldots, x_N)$, where $x_i = 1$ if node i is outside, otherwise $x_i = 0$. We define an energy function for the classification proportional to the visibility connections that have to be cut:

$$E(X) = \sum_{(i,j) \in E} w_{ij} \delta_{ij} \tag{10}$$

where w_{ij} is the weight between node i and j, and

$$\delta_{ij} = \begin{cases} 1 & \text{if } x_i \neq x_j \\ 0 & \text{if } x_i = x_j \end{cases} \tag{11}$$

The minimization of this energy function defines an inside-outside classification with minimal visibility errors. The optimal solution can be computed by the algorithm of Boykov and Jolly [5]. This algorithm was proposed in the context of image processing and it was designed to work with graphs where each node has only a few incident edges. Our experiments show that the algorithm still performs well for graphs with denser (visibility) connections. This is partially due to the fact that we do not shoot a very large numbers of rays to estimate visibility. Fig. 10 shows a visualization of an example graph. In complex situations the algorithm does slow down and can take several minutes. This classification algorithm can be computed once or iteratively depending on the application. In the next section we will show several example applications that we implemented and explain how they setup the classification.

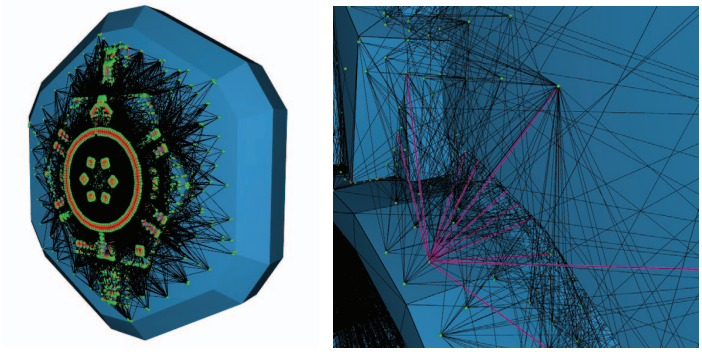


Fig. 10. Left: An example visibility graph. Nodes are shown in green and red and graph-edges in black. Right: All graph-edges connected to a selected node are highlighted in pink.

5 Applications

Inside Removal: Inside removal of models can be computed by one iteration of the sampling and classification step. All triangles that are associated with an outside half space node are considered to be in layer one. All other triangles can be removed. See Fig. 11 and 22 for an example.

Normal Orientation: Normal orientation of a model is computed by one iteration of the sampling and classification step. The user can decide the *front face vertex order*, i.e. if triangles should be oriented in clockwise or counterclockwise order. All triangle faces associated with exactly one outside half-space node, or triangles with multiple agreeing half space nodes are oriented according to the user setting. Triangles associated with multiple half space nodes that indicate a conflicting inside-outside classification are typically duplicated and two triangles with the two possible orientations are created. We found that to be the most intuitive output of normal orientation for general models (see Fig. 13). If the application requires one triangle face to be outside and one triangle be face to be inside, i.e. one normal per triangle, we can pick the orientation where the sum of edge weights to the outside is bigger.

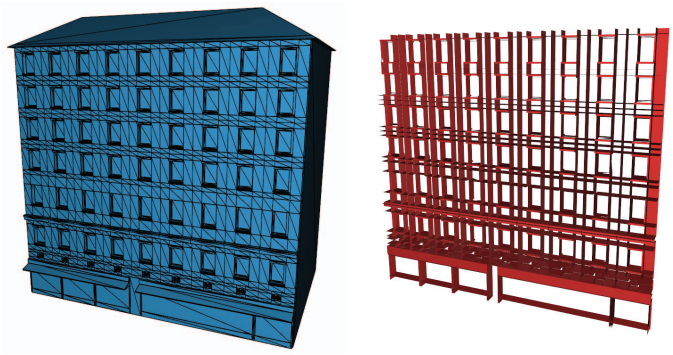


Fig. 11. Left: The outside of the *house* model. The model was created by stacking boxes, so that almost all triangle-edges are non-manifold, and most faces are coplanar. Right: The interior geometry that was removed.

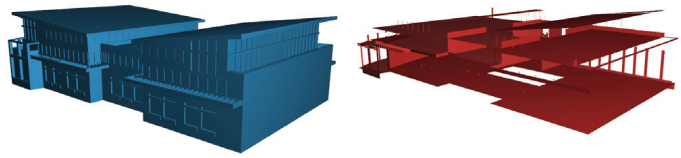


Fig. 12. Left: the outer layer of the *blgd2* model. Right: the inside polygons that were removed.

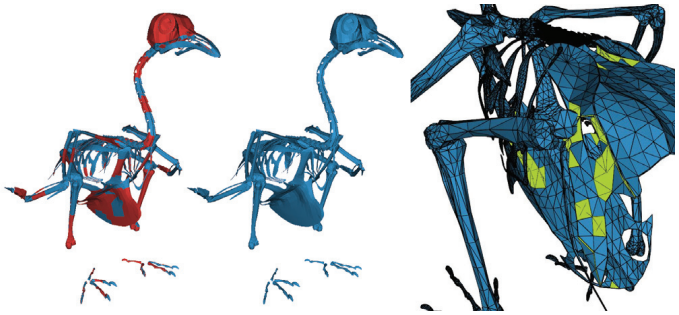


Fig. 13. Left: the *bird* model with front facing (outside) triangles in blue and back facing (inside) triangles in red. Middle: reoriented normals. Right: The model is difficult to process due to the thin (coplanar) structures shown in yellow.

Layer Computation and Visualization: Layer computation can help to gain insight into the internal structure of a model. Layers can be computed by multiple iterations of the classification step and an edge reweighing step. The first iteration computes the first layer. After the first iteration the edges connected to inside nodes associated with triangles in the first layer are connected to the source (the outside). Then the classification step is repeated. The layer computation can be used to assign different transparency values to different layers. See Fig. 14 and Fig. 16 for examples where we render transparency based on an Nvidia whitepaper [8]. While this application has been suggested by previous authors our contribution is the improved definition and computation of inside and outside layers. See Fig. 15 for the separate layers.

Interactive Editing and Mesh Analysis: The framework also includes the ability to interact with the visibility graph. A user can define additional nodes on the model or in free space and manually set weights of edges and classifications of nodes. We do not use interac-

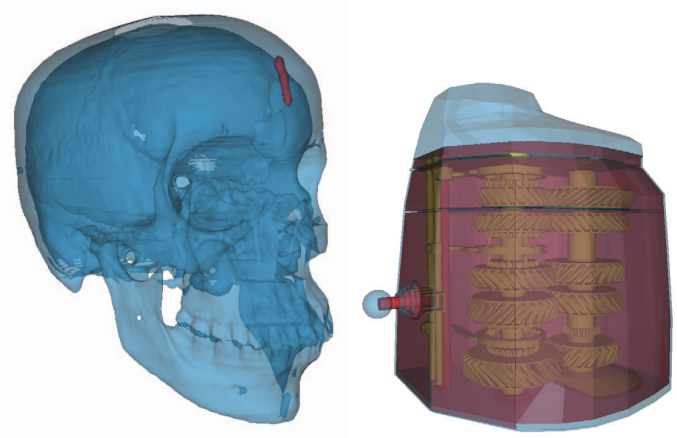


Fig. 14. Left: The *skull* model with the first layer in blue and the second layer in red. Right: layer visualization in the *motor* model.

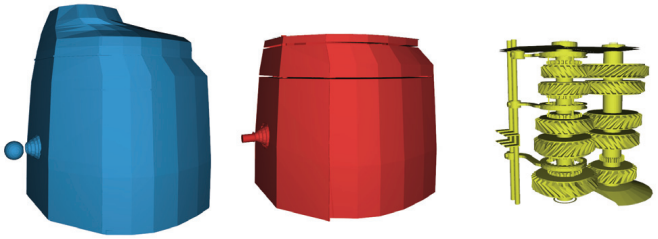


Fig. 15. Three layers of the top of the motor model.

tive editing for the results in this paper, because that would skew the comparison with other approaches.

6 Results

We present the running times of our algorithm on a small set of models. The models were selected to include some variety of how difficult the models are and how many sharp features they possess. For each model we run one inside-outside classification for inside removal. We list the number of polygons of the input model, the number of polygons of the output, as well as the complete running time (See Table 1). We also include an informal comparison with Ju's PolyMender software [10] available at his web page[1]. PolyMender needs to scan the input models into a hierarchical grid. One parameter *maxd* allows the user to control the maximal depth of the octree. In our experiments, we tune the parameters for PolyMender to capture the details of different models. Polymender generates an order of magnitude more triangles for the models house, building 2, and mechanic 1. A main reason is that coplanar faces seem to force a very detailed voxelization. Most of the models are well reconstructed with $maxd = 9$. The bird skeleton contains many detailed structures and requires $maxd = 10$. For simpler models such as the Turbine and Skull we set $maxd = 8$. We use the version *dc-clean* for all models except for the bird which uses *dc*. While our algorithm cannot compete with Polymender's hole filling functionality, we believe that the results underline that our proposed system can complement volumetric methods well. We also list a breakdown of our running times for the three major steps clustering, sampling, and graph cut in Table 2. In the same table we list statistics about the graph and memory consumption. The main parameters of our algorithm are the number of random samples, the number of border samples, the number of rays per node, the threshold for the intersection computation and the threshold for the coplanarity computation. For these tests we made a binary correct or not correct decision based on visual inspection. See Table 3 and Table 4 for the results. We measured the influence of clustering on three models (see Table 5). Even though we do not perform clustering in the second column, we still have to run the intersection and coplanarity tests (I[s]). Note that the skull is an extreme case that benefits a lot from clustering, as it has very large connected components; the others are just normal cases where both connected mesh and intersected triangles both exist. The third column simulates a triangle soup case by randomly messing up selected triangles. The clustering algorithm typically improves the overall running time because it reduces the number of nodes and edges in the visibility graph processed by the graph cut algorithm.

[1] http://www.cse.wustl.edu/ taoju/code/polymender.htm

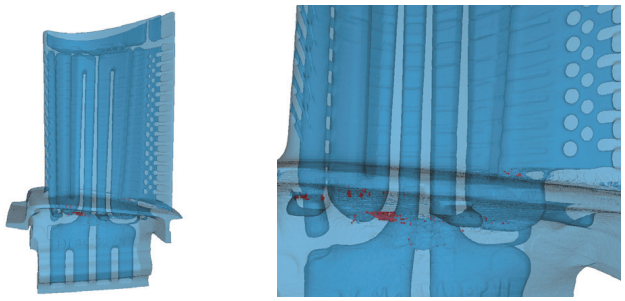


Fig. 16. Left: The *turb* model with the first layer in blue and second layer in red. This model is an illustration of the difference of the *view-based* and *object-based* inside-outside classification. Please note that the interior pipe structure is correctly classified as outside because it is reachable through several smaller pipes. Right: a closeup.

Model		Alg.	Tri.In#	Tri.Out#	Time [s]
house		VGC	24K	11K	89.1s
		JU	24K	1.9M	14.97s
bird		VGC	111K	92K	17.6s
		JU	111K	1.1M	8.37s
bldg1		VGC	93K	85K	43.9s
		JU	93K	791K	17.812s
bldg2		VGC	33K	22K	22.7s
		JU	33K	897K	6.687s
mech1		VGC	1.04K	1.03K	5.52s
		JU	1.04K	861K	3.062s
mech2		VGC	25K	23K	3.75s
		JU	25K	742K	2.34s
turb		VGC	1.76M	1.76M	66.5s
		JU	1.76M	843K	46.18s
skull		VGC	1.16M	1.16M	47.89s
		JU	1.16M	1.61M	44.31s
motor		VGS	140k	44k	74.14s
		JU	140k	1.5M	17.016s

Table 1. The table lists the number of input triangles (Tri.In) and output triangles (Tri.Out) and total computation time in seconds. We compare our results (VGC: Visibility Graph Cut) against Ju's PolyMender (version 1.7).

7 Discussion

In this section we want to compare to previous work, identify contributions and open problems that are of interest for future research.

Mesh Repair: Our algorithm performs several steps of a mesh repair framework, but we do not currently address hole filling, a major challenge that is implemented in some previous mesh repair systems, e.g. [2, 10]. However, the problem of hole filling is not solved by previous work and it remains an inherently difficult topic. Many cases require user input to be resolved. Our major avenue for future work is to determine how the visibility graph can be used to let a user specify hints for hole filling.

Inside Outside Classification: We believe that our algorithm significantly improves the state of the art, because we make better use of visibility information. Previous work, especially Borodin et al. [4]

Model	C[s]	S[s]	G[s]	Rays	Node	Edge	Mem
house	2.1	73	14	20M	1.2M	8.7M	316M
bird	7.3	9.1	1.2	3.5M	224K	1.2M	46M
bldg1	6.4	36	1.5	11M	691K	5.3M	183M
bldg2	4.8	16.7	1.2	6.9M	432K	1.9M	74M
mech1	0.1	5.3	0.1	2.4M	77K	825K	28M
mech2	0.7	2.1	0.95	1.2M	80K	156K	7M
turb	66	0.1	1ms	9.4K	590	739	40K
skull	46	1.4	0.06	55K	3.4K	1.5K	338K
motor	5.1	58	11	26M	1.6M	12.4M	443M

Table 2. The break-down of computation times for each model and running times in seconds for clustering (C) , sampling (S), and graph cut (G). Further we list the number of rays used in the sampling stage (Rays), the number of nodes (Node) and edges (Edge) of the visibility graph, and the memory consumption (Mem).

RS = 5, BS = 30, RperN =	8	16	32	48	64
house, time=	49s	89s	206s	327s	382s
RS = 5, RperN = 16, BS =	6	15	24	30	36
house, time=	23s	44s	68s	89s	121s
BS = 30, RperN = 16, RS =	0	5	10	15	20
house, time=	75s	89s	102s	121s	143s
RS = 10, BS = 0, RperN =	8	16	32	48	64
bird, time=	7s	17s	22s	32s	42s
RS = 10, RperN = 16, BS =	0	6	15	24	30
bird, time=	17s	35s	59s	108s	146s
BS = 0, RperN = 16, RS =	5	10	15	20	25
bird, time=	8s	17s	33s	52s	79s

Table 3. We evaluate the parameters number of random samples (RS), number of border samples (BS), and the number of rays per node (RperN) on two selected models. We always keep two parameters the same and vary the other one. As result we report the running time. The lowest running time that produces a correct result is highlighted in red.

and Murali et al. [15] make many assumptions about the model and are therefore not robust to intersecting and coplanar triangles. On the other hand volumetric methods [2, 10] can deal with a larger number of inputs, but they are not able to classify the original geometry. As a result a significant increase in triangles is likely for all models that do not have a nice uniform triangulation. Furthermore, our experience with the Polymender software [10] shows that there are some robustness issues that would have to be resolved. We therefore argue that our algorithm is the best available choice for inside-outside classification and related applications.

8 Conclusion

We presented a robust visibility based geometry analysis algorithm that takes a triangular model as input and computes an inside and outside classification of oriented triangle faces. We show how to use this classification for three example applications: inside removal, normal orientation, layer-based visualization. The core idea of out framework is to propagate visibility using ray casting and to compute a classification using graph cut. We believe that our algorithm is a significant technical improvement over previous techniques.

Model	x-size	y-size	z-size	I-thresh	C-thresh
house	204	216	157	$10^{-6} - 10^{-2}$	$10^{-5} - 10^{-2}$
mech1	26	25	30	$10^{-6} - 10^{-3}$	$10^{-5} - 10^{-2}$
motor	5	21	20	$10^{-6} - 10^{-2}$	$10^{-5} - 10^{-2}$

Table 4. We list the setting for three models for the threshold parameters that produce correct results: threshold for the intersection detection (I-thresh) and threshold for the coplanar detection (C-thresh). The parameter range was determined by running a large number of tests with different thresholds and subsequent visual inspection of the results.

	Clustering (C[s]/S[s]/G[s])	No Clustering (I[s]/S[s]/G[s])	Triangle Soup (S[s]/G[s])
bird	18 (7.3/9.1/1.2)	25 (3.5/18.7/3)	11 (10.1/1.1)
skull	47 (46/1.4/0.06)	194 (21.4/145/28)	161 (135.2/25.9)
motor	25 (3.5/20/1.5)	48 (2.3/43.2/2.5)	11 (10.1/0.5)

Table 5. Running times for clustering(C), sampling(S), intersection(I), and graph cut(G)

Acknowledgements

The authors acknowledge the contribution of all the reviewers and financial support from NSF IIS-0612269, NSF CCF-0643822, NSF CCF-0546881, and grant LC-06008 from the Czech Ministry of Education, Youth and Sports. We thank Will Schroeder, Ken Martin, Bill Lorensen, Bruce Teeter, Terry Yoo, Mark Levoy and the Stanford Graphics Group for the 3D models in this paper.

References

[1] G. Barequet and M. Sharir. Filling gaps in the boundary of a polyhedron. *Computer Aided Geometric Design*, 12(2):207–229, 1995.
[2] S. Bischoff, D. Pavic, and L. Kobbelt. Automatic restoration of polygon models. *ACM Transactions on Graphics*, 24(4):1332–1352, Oct. 2005.
[3] J. H. Bøhn and M. J. Wozny. A topology-based approach for sell-closure. In *Geometric Modeling for Product Realization*, volume B-8, pages 297–319, 1992.
[4] P. Borodin, G. Zachmann, and R. Klein. Consistent normal orientation for polygonal meshes. In *Computer Graphics International*, pages 18–25. IEEE Computer Society, 2004.
[5] Y. Boykov and M.-P. Jolly. Interactive graph cuts for optimal boundary and region segmentation of objects in n-d images. In *ICCV*, pages 105–112, 2001.
[6] CGAL. Cgal: Computational geometry algorithms library. www.cgal.org, 2008.
[7] P. Dutre, K. Bala, and P. Bekaert. *Advanced Global Illumination*. AK Peters, 2006.
[8] C. Everitt. Interactive order-independent transparency, 2001.
[9] A. Guéziec, G. Taubin, F. Lazarus, and W. Horn. Cutting and stitching: Converting sets of polygons to manifold surfaces. *IEEE Trans. Vis. Comput. Graph*, 7(2):136–151, 2001.
[10] T. Ju. Robust repair of polygonal models. *ACM Transactions on Graphics*, 23(3):888–895, Aug. 2004.
[11] T. Ju, F. Losasso, S. Schaefer, and J. D. Warren. Dual contouring of hermite data. *ACM Trans. Graph*, 21(3):339–346, 2002.
[12] P. Liepa. Filling holes in meshes. In *Symposium on Geometry Processing*, volume 43, pages 200–206, 2003.
[13] T. Möler. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997.
[14] T. Möller and E. Haines. *Real-Time Rendering, Second Edition*. A. K. Peters Limited, 2002. ISBN 1568811829.
[15] T. M. Murali and T. A. Funkhouser. Consistent solid and boundary representations from arbitrary polygonal data. In *1997 Symposium on Interactive 3D Graphics*, pages 155–162. ACM SIGGRAPH, 1997.
[16] F. S. Nooruddin and G. Turk. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):191–205, Apr./June 2003.
[17] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Trans. on Graphics*, 24(3):1176–1185, 2005.
[18] Shewchuk. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In *WACG: 1st Workshop on Applied Computational Geometry: Towards Geometric Engineering, WACG*, 1996.
[19] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addiso Wesley, 2006.
[20] P. Wonka, M. Wimmer, K. Zhou, S. Maierhofer, G. Hesina, and A. Reshetov. Guided visibility sampling. *ACM Trans. Graph.*, 25(3):494–502, 2006.
[21] E. Zhang and G. Turk. Visibility-guided simplification. In *IEEE Visualization*, 2002.

Appendix I

Layout-aware optimization for interactive labeling of 3D models

Čmolík, L. - Bittner, J.: Layout-aware optimization for interactive labeling of 3D models. Computers & Graphics. 2010, vol. 34, no. 4, p. 378-387. ISSN 0097-8493. **IF=1.0**

Computers & Graphics 34 (2010) 378–387

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical Section

Layout-aware optimization for interactive labeling of 3D models

Ladislav Čmolík *, Jiří Bittner

Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

ARTICLE INFO

Keywords:
Illustration
Labeling
Interaction

ABSTRACT

We propose a novel method for computing labeling of 3D illustrations in real-time. We solve a multiple criteria optimization problem in which we consider the desired layout already in the stage of searching for salient points of the labeled areas. In the solution we employ fuzzy logic combined with greedy optimization. The method runs on the GPU and achieves interactive rates on medium sized models. The results indicate that the method compares favorably to the state-of-the-art interactive labeling techniques.

1. Introduction

Illustrations are an important visual component of communication. They are used to visually expound various objects and support their textual description. In the latter case the reader needs to link the terms contained in the text with the illustration. The relation between textual and visual representations of information is mediated through *labeling*, i.e. assigning textual labels to various parts of the illustration. Digital media offer new possibilities for illustrations, such as 3D models, which the reader can manipulate interactively.

In this paper we present a novel labeling method which is targeted at interactive illustration of 3D models. The three main contributions of the paper are: (1) We formulate the labeling as multiple criteria optimization problem which considers the desired layout already in the stage of searching for salient points of labeled areas. This improves the resulting labeling compared to previous methods especially in the areas with many labels (see Fig. 1). (2) We use fuzzy logic and greedy optimization to solve the multiple criteria optimization problem. (3) We describe a GPU implementation of the method, which achieves interactive rates on medium sized models. Since the labeling is recomputed every frame, our method supports arbitrary manipulations of the model as well as interactive modifications of the model and of the labels.

The paper is organized as follows: Section 2 introduces terms used in the area of labeling. Section 3 summarizes state-of-the-art in the area of labeling. Section 4 formally describes the problem of external labeling. Section 5 presents our solution to the problem, which is summarized once more in Section 6. Section 7 presents results and comparisons and finally Section 8 concludes the paper.

* Corresponding author.
E-mail addresses: cmolikl@fel.cvut.cz (L. Čmolík), bittner@fel.cvut.cz (J. Bittner).

0097-8493/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2010.05.002

2. The labeling problem

In this section we describe the labeling problem and define terminology used later in the paper.

2.1. Basic terminology

We assume that the model consists of n objects $O_i, 1 \leq i \leq n$ and each object O_i is assigned a unique label. After projection of the model to the screen, object O_i becomes visible in the screen area A_i. Note that if O_i is invisible then $A_i = \emptyset$.

The interior area A_I is a superset of the union of A_i over all objects. In our case we deal with a convex A_I, which is constructed to include a small boundary area around the model. The exterior area A_E is the complement of A_I with respect to the total screen area A_S ($A_E = A_S - A_I$). If the labels are placed in the interior area we call the labeling *internal*. If the labels are placed in the exterior area we call the labeling *external*. Our method deals with external labeling and thus we describe it in more detail in the next section.

2.2. External labeling

In external labeling a label is associated with the *anchor*, the *leader line*, and the *label box*. The anchor $\mathbf{a}_i$ is a point inside the area A_i. The label box L_i is a rectangle containing the label typically in the form of a short text string. Leader line $\mathbf{l}_i$ is a line segment connecting the anchor $\mathbf{a}_i$ and the label box. The endpoint of the leader line is denoted $\mathbf{e}_i$ (see Fig. 3).

The label boxes in external labeling can be either floating or fixed. A floating label box can be placed at any position in the external area while a fixed label box can be placed only at several fixed positions (the number of these positions is typically the same as the number of label boxes).

Similarly, the anchors and endpoints of leader lines can be floating or fixed. A floating anchor can be placed at any position inside the corresponding area while the fixed anchor has a one or several fixed positions. A floating endpoint can be placed at any position on the boundary of the label box while a fixed endpoint has only one or a few fixed positions. We call a labeling method *automatic* if it deals with both floating anchors and floating endpoints.

With floating label boxes, a set of *principal* directions D can be used to specify the desired layout of the leader lines. Then each leader line l_i should be parallel to some principal direction $d \in D$. However, this is not always possible for all leader lines without introducing overlaps of the leader lines or the label boxes. If this happens there are two commonly used solutions:

- The leader lines remain straight lines, but some of them are no longer parallel to any principal direction.
- Some of the leader lines are split into two orthogonal lines with one bend, where the segment from the anchor to the bend is orthogonal to $d \in D$ and the segment from the bend to the endpoint is parallel with d.

Examples of layouts with different sets of principal directions and type of leader lines are shown in Fig. 2.

3. Related work

The labeling problem has first received attention in the cartographic domain for assigning labels to static features. A comprehensive bibliography of these labeling techniques can was presented by Wolff and Strijk [19].

Although we deal with external labeling we identified several methods for internal labeling that are related to our work: The method of Götzelmann et al. [8] determines the positions of internal labels using a multiple criteria optimization. In the method of Ropinski et al. [15] the labels indicate the shape of the overlaid part of the 3D object. The method of Maass and Döllner [13] integrates the labels into a virtual reality environment.

In the case of external labeling we split the discussion of the related work into four parts according to the flexibility of anchors and label boxes (fixed vs. floating).

Fixed anchors and fixed label boxes: Bekos et al. [3] defined the boundary labeling problem where the label boxes are arranged on the rectangle enclosing a set of anchors. They study various types of leader lines, arrangements of label boxes and sizes of label boxes. Their primary focus is on efficient labeling algorithms that calculate leader lines whose combined length is minimal. Later, Benkert et al. [5] formulated the boundary labeling problem as a multiple criteria optimization problem where the length of leader lines, the number of bends, and the distance of anchors to leader lines are used to find an optimal solution of one-sided labeling (all label boxes are on one side of the enclosing rectangle).

Floating anchors and fixed label boxes: Bekos et al. [2] extended the boundary labeling problem. Each anchor can float within a polygonal area. They propose efficient labeling algorithms for various types of leader lines under some restrictions on the polygonal area with the aim of minimizing the combined length of leader lines.

Fixed anchors and floating label boxes: Stein and Décoret [17] presented a greedy algorithm for the labeling of fixed anchors attached to 3D objects. The occlusion of the 3D objects is minimized by placing label boxes in empty areas. Shadow regions and a summed area table [11] are used to prevent the crossing of leader lines and overlaps of label boxes.

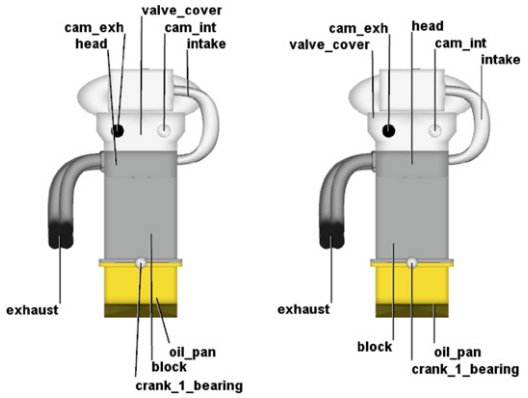


Fig. 1. A comparison of our method with the method of Ali et al. [1] on an engine model using top–bottom layout. (left) Ali et al. [1], (right) the proposed technique. Note that in our method the leader lines are distributed more evenly over the model, which according to our opinion increases their saliency and leads to more aesthetic labeling.

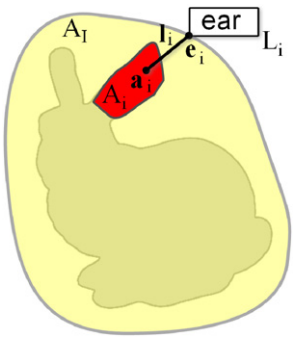


Fig. 3. Illustration of the basic terms used in external labeling.

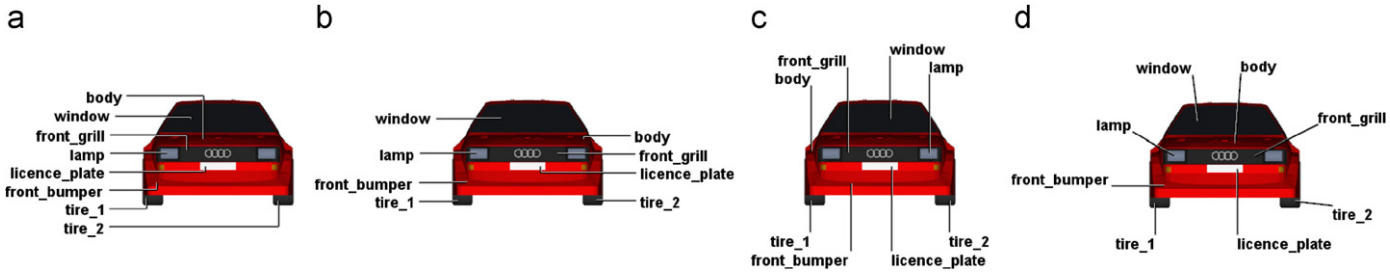


Fig. 2. Examples of different label layouts: (a) left layout with orthogonal lines, (b) left–right layout with orthogonal lines, (c) top–bottom layout with straight lines, (d) silhouette based layout with straight lines.

Floating anchors and floating label boxes: Hartmann et al. [9] introduced a method to determine the labeling of 2D and 3D objects based on dynamic potential fields. The problem is split into finding anchors for the objects and labeling those anchors. The labeling is obtained as an equilibrium between attractive and compulsive forces established for the label boxes and the objects. Ali et al. [1] presented a real-time labeling pipeline, allowing to produce various labeling styles of 3D objects. The problem is again split into finding anchors of 3D objects and labeling those anchors. This method is able to calculate the labeling of 3D models with frame-to-frame coherence at interactive frame rates. Götzelmann et al. [6] presented an agent-based labeling system, allowing the integration of internal and external labels. Also here the problem is split into finding anchors for 3D objects and labeling those anchors. Agents are assigned to initial labels and they compete and/or cooperate to meet metrics for functional and aesthetic label layouts [10], extracted from handmade illustrations.

There are also methods that do not fit into the classification according to anchor and label box properties, such as the method of Götzelmann et al. [7] for labeling animated objects or the method of Vollick et al. [18], which is able to learn a specific labeling style from given examples and to apply the style to new illustrations.

4. External labeling as optimization problem

In this section we review the criteria for finding a good external labeling.

4.1. Criteria for external labeling

A labeling of an illustration should exhibit four general properties: *readability*, *unambiguity*, *aesthetics*, and *compactness*. In order to allow an automatic computation of the labeling these properties are usually transformed into a number of criteria, which deal with the positions of the leader lines, label boxes, and anchors [9,10]:

(I) *Leader line crossing*: Leader lines do not cross.
(II) *Leader line distance*: Leader lines are not too near to each other.
(III) *Leader line alignment*: Leader lines are aligned with principal directions.
(IV) *Leader line length*: Leader lines are as short as possible.
(V) *Label box distance*: Label boxes are near to the corresponding objects.
(VI) *Label box overlap*: Label boxes do not overlap.
(VII) *Anchor salience*: Anchors are salient points of the corresponding areas.
(VIII) *Anchor distance*: Anchors $\mathbf{a}_i$ are not too near to each other.
(IX) *Endpoint distance*: Endpoints $\mathbf{e}_i$ are not too near to each other.
(X) *Label box coherence*: Label boxes in frame t are close to their positions in frame $t-1$.
(XI) *Anchor coherence*: Anchors in frame t are close to their positions in frame $t-1$.

Note that the last two criteria are important for interactive applications, where the labeling of the model should exhibit temporal coherence, i.e., the leader lines and the labels should not jump from one frame to the next. Note that in static applications these two criteria can be neglected. Some of the criteria contradict each other (e.g., criterion (IV) and criterion (VII)) and thus we have to find a balance between the contradicting criteria. Finding the importance of the contradicting criteria is an issue dealing with human perception and in general there is no unique optimal solution to the external labeling problem.

5. Computing optimized external labeling

If we analyze the methods suitable for automatic interactive external labeling of 3D objects [1,6], we observe that these methods proceed in three steps: (1) calculate the anchors, (2) calculate the initial layout for the anchors, (3) correct the initial layout. This approach, however, has one drawback: As the calculation of anchor points in the first step is not using any information about the layout, the distribution of these points may lead to situations in which the labeling computed in the second step contains clusters of overlapping labels. In the third step these clusters can be resolved by repositioning the labels, but in general this leads to undesired bends or rotations of the leader lines. In our approach, we improve these approaches by combining the optimization of anchor determination with optimizing the layout of the labels.

5.1. Overview of our approach

We use a greedy optimization to determine the positions of the anchors, the leader lines, and the label boxes. As a leader line connects the anchor with a label box, we only deal with finding a good set of leader lines. Leader lines are calculated and placed over the illustration by repeating the following two steps:

1. Select an unlabeled area.
2. Calculate the leader line of the area.

In the first step we select an area which has not been labeled so far. We process first the areas where there is less freedom for placing the labels (i.e., small areas in dense regions of the model). The area selection is described in detail in Section 5.4. The second step forms the core of the method. We search for a good leader line considering all feasible anchor points and principal directions. The leader line computation is described in Section 5.3. In the next section we discuss the criteria which we later use for both the area selection and the leader line computation.

5.2. Simplifying the criteria for optimization

Our method performs the determination of the anchors and the calculation of the label layout together, which allows to improve the results in some difficult situations with a high density of labels. However, optimizing both the positions of the anchors and the label boxes is more complicated than optimizing the position of just one of these features as done by previous interactive methods.

In particular when placing a single floating leader line over the illustration we have four degrees of freedom (two for the anchor point and two for the end point) and so we deal with a 4D problem. However, we can reduce the dimensionality of the problem by making use of the labeling criteria dealing with principal directions by defining a unique leader line with respect to each anchor point as follows: According to the above listed criteria (III) and (IV) this line is determined as the shortest line with respect to the silhouette of A_I which follows a principal direction. More precisely, for a leader line $\mathbf{l}_i = (\mathbf{a}_i, \mathbf{e}_i)$, $\mathbf{e}_i$ is the point on the silhouette of A_I closest to $\mathbf{a}_i$ (criterion (IV)) such

that $\mathbf{a}_i - \mathbf{e}_i$ is parallel to a principal direction (criterion (III)). In this way we reduce the number of possible directions that the leader line can follow to a single line (see Fig. 4(a)). In turn we reduce the dimensionality of the problem from 4D to 2D which allows to us to represent the measures associated with the quality of every leader line candidate directly in the labeled image.

Following this mapping we can simplify the criteria for labeling as follows: The leader lines defined using the described mapping will never cross (assuming A_l is convex), therefore the criterion (I) (leader line crossing) can be omitted.

Criterion (II) (leader line distance) can be replaced by considering the distance between the anchors and between the endpoints of the leader lines.

We can also replace the criteria for the label boxes by criteria for the endpoints and anchors. If leader lines are short (criterion (IV)) then each label box is near to its corresponding object. Therefore the criterion (V) can be omitted. The criterion (IV) is replaced as follows: The leader lines do not cross and they are not even near each other if the distance between each pair of endpoints $\mathbf{e}_i,\mathbf{e}_j, i \neq j$, is greater than a threshold $t > 0$. If the threshold t is greater than the height of the label box and the internal area A_l is convex, then the label boxes can be stacked around silhouette of A_l without overlap. See Fig. 4(b) for details.

In summary, by using the above described mapping and a convex shape for describing the internal area A_l we can simplify the labeling criteria to the following seven criteria (we use Arabic numbers in order to distinguish between the original and the simplified criteria):

1. *Leader line alignment*: Leader lines are aligned with principal directions.
2. *Leader line length*: Leader lines are as short as possible.
3. *Anchor salience*: Anchors are salient points of the corresponding areas.
4. *Anchor distance*: Anchors $\mathbf{a}_i$ are not too near to each other.
5. *Endpoint distance*: Endpoints $\mathbf{e}_i$ are not too near to each other.
6. *Label box coherence*: Label boxes in frame t are close to their positions in frame $t-1$.
7. *Anchor coherence*: Anchors in frame t are close to their positions in frame $t-1$.

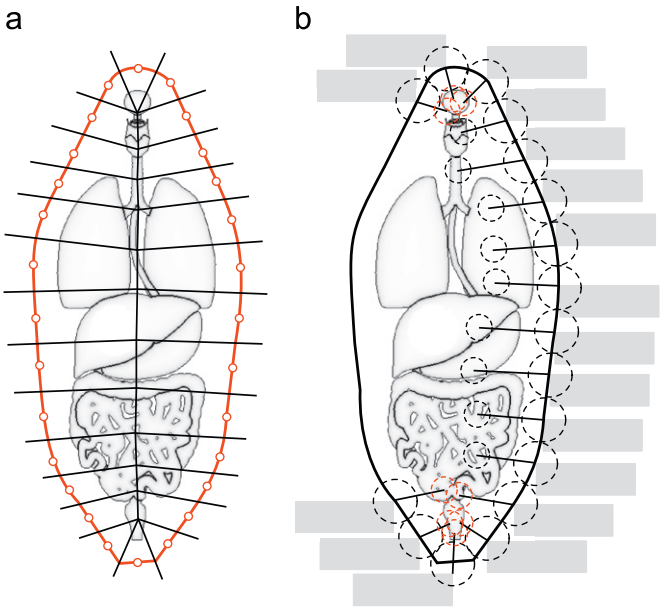


Fig. 4. (a) Voronoi diagram of points on the silhouette of A_l. Note that as we add more points on the silhouette the cells become thinner. When we use all points on the silhouette the cells will collapse into lines and thus we get a leader line candidate for each point in A_l. (b) Stacking of label boxes proposed by Ali et al. [1]. Label boxes can be stacked around the silhouette of a convex A_l if there is space around each leader line endpoint equal or bigger than the height of the label box. Note that in the upper and lower parts some leader lines have to be extruded (e.g., the leader line of the lowest label box). If we demand a larger distance between the anchors we have to choose only one leader line from the two overlapping ones (marked by a red circle around the anchor). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

These seven criteria are used in the optimization framework for finding the most suitable labeling as described in the following sections.

5.3. Leader line calculation

Given an area A_i we need to find a leader line which follows the criteria listed in the previous section. For every point in A_i we have a unique *leader line candidate*. From these candidates the algorithm selects the leader line which provides the best match for the labeling criteria.

In order to do so we use fuzzy optimization [21] based on fuzzy decision making by Bellman and Zadeh [4]. A process where simultaneous satisfaction of several criteria is sought.

We see the following benefits in using the fuzzy optimization. It can handle conflicting criteria due to the simultaneous satisfaction of all criteria. It can handle uncertainty in the criteria due to its basis in the fuzzy set theory and fuzzy logic.

Fuzzy set theory is an extension of set theory. In fuzzy set theory we can express a partial membership of an element to the set. A fuzzy set is commonly described by its membership function that maps each element to values in the range [0,1] which indicates the membership of the element to the set, 0 means that the element is not in the set and 1 means that the element is entirely in the set. Fuzzy logic defines operations on the fuzzy sets that are equivalents of Boolean logic operations (e.g., negation, conjunction, and disjunction).

In the fuzzy optimization we consider a solution space X. Each criterion C_i is modeled as a fuzzy set on X and its membership function f_i describes the satisfaction of the criterion by the solutions $x \in X$. The membership functions are aggregated together using an intersection operator (e.g., fuzzy conjunction), thus we obtain the aggregated membership function f for the criteria:

$$f(x) = \bigcap_{1 \le i \le 7} f_i(x). \tag{1}$$

We denote the value returned by $f(x)$ as feasibility of solution x. Note that the intersection operator guaranties the simultaneous satisfaction of the criteria. In other words, satisfaction of one criterion cannot compensate dissatisfaction of another criterion. The last step is to find the most feasible solution, that is the global maximum of the aggregated membership function $f(x)$.

In the following, we model the satisfaction of our labeling criteria as fuzzy sets and define their membership functions. In our case the solution space are the leader line candidates of area A_i for which we search the leader line, which provides best match for all criteria. The criterion 1 is satisfied implicitly since we consider only leader line candidates which are aligned with principal directions. The fuzzy membership functions for a leader line candidate $\mathbf{l} = (\mathbf{a}, \mathbf{e})$ of area A_i for criteria 2–7 are evaluated as follows:

Leader line length: To express the leader line length we calculate the distance of the anchor from the endpoint and

normalize it using the length of the longest leader line candidate:

$$f_2(\mathbf{l}) = 1 - \frac{|\mathbf{a}-\mathbf{e}|}{d_{max}}, \tag{2}$$

where $|\mathbf{a}-\mathbf{e}|$ is the distance of points $\mathbf{a}$ and $\mathbf{e}$ and d_{max} is the length of the longest leader line candidate (see Fig. 5(b)).

Anchor salience: To express the anchor salience we compute the distance of the anchor from the silhouette of A_i and normalize it using the length of the longest leader line candidate:

$$f_3(\mathbf{l}) = \frac{dist_{sil}(\mathbf{a})}{d_{max}}, \tag{3}$$

where $dist_{sil}$ is a distance of the anchor to the silhouette of area A_i and d_{max} is length of the longest leader line candidate (see Fig. 5(c)).

Anchor distance: To express the distance from other anchors we use the minimal distance of the anchor to already placed leader lines $\mathbf{p} \in P$. The distance is normalized using the threshold d_a, which reflects the desired minimal distance between the anchors. For a given leader line candidate and an already placed leader line $\mathbf{p}$ we get

$$dist_a(\mathbf{l},\mathbf{p}) = \min\left(\frac{|\mathbf{a}-\mathbf{a_p}|}{d_a}, 1\right), \tag{4}$$

where $|\mathbf{a}-\mathbf{a_p}|$ is the distance of the anchors. To compute membership function f_4 with respect to all already placed leader lines P we use a conjunction of $dist_a$:

$$f_4(\mathbf{l}) = \bigwedge_{\mathbf{p} \in P} dist_a(\mathbf{l},\mathbf{p}). \tag{5}$$

Fig. 5(d) shows a visualization of this membership function.

Endpoint distance: To express the distance from the endpoints we use the minimal distance of the endpoint of the leader line to endpoints of already placed leader lines $\mathbf{p} \in P$. The distance is normalized using the threshold d_e, which reflects the desired minimal distance between the anchors. For a given leader line candidate and an already placed leader line $\mathbf{p}$ we get

$$dist_e(\mathbf{l},\mathbf{p}) = \min\left(\frac{|\mathbf{e}-\mathbf{e_p}|}{d_e}, 1\right). \tag{6}$$

To compute membership function f_5 with respect to all already placed leader lines P we use a conjunction of $dist_e$:

$$f_5(\mathbf{l}) = \bigwedge_{\mathbf{p} \in P} dist_e(\mathbf{l},\mathbf{p}). \tag{7}$$

Fig. 5(e) shows a visualization of this membership function.

Anchor coherence: The anchor coherence is expressed by calculating the distance of the corresponding anchors in two consecutive frames and normalizing it using the diagonal of the bounding sphere d_S:

$$f_6(\mathbf{l}) = 1 - \frac{|\mathbf{a}_t-\mathbf{a}_{t-1}|}{d_S}, \tag{8}$$

where $|\mathbf{a}_t-\mathbf{a}_{t-1}|$ is the distance of the corresponding anchor points in frames t and $t-1$.

Endpoint coherence: To express the endpoint coherence we calculate the distance of the corresponding endpoints in two consecutive frames and normalize it using the diagonal of the bounding sphere d_S:

$$f_7(\mathbf{l}) = 1 - \frac{|\mathbf{e}_t-\mathbf{e}_{t-1}|}{d_S}, \tag{9}$$

where $|\mathbf{e}_t-\mathbf{e}_{t-1}|$ is the distance of the corresponding endpoints in frames t and $t-1$.

To obtain the aggregated membership function of all criteria we use the natural fuzzy conjunction [21] as the intersection operator in Eq. (1). Thus, we get

$$f(\mathbf{l}) = \bigwedge_{2 \le i \le 7} f_i(\mathbf{l}). \tag{10}$$

Natural fuzzy conjunction corresponds to a simple multiplication. Note that the use of multiplication is robust with respect to scaling one, several, or all membership functions, i.e., the most feasible solution does not change.

If we want to tune the behavior of the labeling we can define a weight for each criterion. Here we use the weighted modification

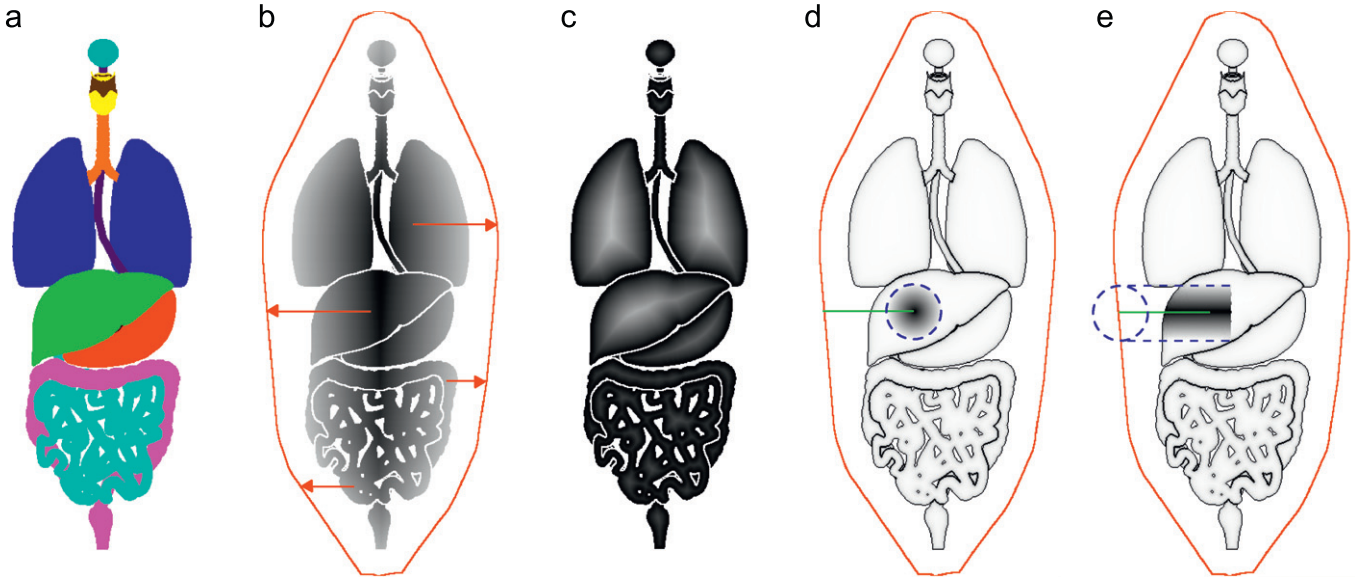


Fig. 5. (a) Colour coded areas A_i. (b–e) Visualizations of membership functions: (b) f_2—leader line length, (c) f_3—anchor salience, (d) f_4—anchor distance, (e) f_5—endpoint distance. A darker pixel corresponds to a less feasible leader line candidate. We show the membership functions for left–right layout and therefore only the distances to the convex hull in left and right direction are considered in figure (b). Figures (d) and (e) show the corresponding membership functions after one leader line has been placed over the illustration (depicted in green colour). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

of fuzzy optimization introduced by Yager [20], who proposed to modify the membership function of each criterion prior to the aggregation. The modified function F_i is expressed as

$$F_i(\mathbf{l},w_i) = f_i(\mathbf{l})^{w_i}, \tag{11}$$

where f_i is the membership function and w_i is the weight in range [0, 1]. Eq. (1) is modified to

$$F(\mathbf{l}) = \bigwedge_{2 \le i \le 7} F_i(\mathbf{l},w_i). \tag{12}$$

Note that if $f_i'(\mathbf{l}) = c \cdot f_i(\mathbf{l})$ where c is a constant, then $F_i'(\mathbf{l},w_i) = f_i'(\mathbf{l})^{w_i} = c^{w_i} \cdot f_i(\mathbf{l})^{w_i} = c^{w_i} \cdot F_i(\mathbf{l},w_i)$ where c^{w_i} is also a constant. Therefore the weighted fuzzy optimization is resistant to scaling.

Note that all described functions are evaluated on a discrete grid corresponding to the image pixels (see Fig. 5). In the evaluation we make heavy use of the jump flooding algorithm for computing distance fields. We provide more details about the implementation in Section 6.2.

5.4. Processing order of labeled areas

We use a greedy optimization technique without backtracking and therefore the quality of the final labeling depends on the order in which the leader lines are placed over the illustration. Each leader line placed over the illustration potentially reduces the possibilities for placing leader lines for other areas.

We use the following strategy for determining the processing order of the labeled areas: we first process the areas that could be heavily influenced by leader lines of other areas. The presumption is that the area with a low number of feasible leader line candidates can get influenced much more than an area with high number of feasible candidates and therefore it should be processed first.

Thus for each area we sum the feasibility of all corresponding leader lines obtained with Eq. (12) and define the priority p_i of the area A_i as

$$p_i = \sum_{\mathbf{l} \in A_i} (1 - f(\mathbf{l})). \tag{13}$$

For the weighted conjunction we get

$$P_i = \sum_{\mathbf{l} \in A_i} (1 - F(\mathbf{l})). \tag{14}$$

Initially we evaluate priorities for all areas and select the area with the highest priority as the next area for placing a label. After the leader line is calculated we reevaluate the priorities of all unlabeled areas and proceed again by selecting the area with the highest priority.

5.5. Corrections of label layout

Note that in some cases not all labeling criteria can be satisfied at the same time. Especially the satisfaction of the criterion 5 (endpoint distance) is crucial, which, if violated, would result in overlapping label boxes. Fortunately, this can happen only if there is no space left to place new leader lines and the violation can be easily detected and corrected by repositioning the label boxes to avoid overlaps and relieving the criterion 1, i.e., splitting or rotating the leader lines [1]. This in turn may cause intersections of leader lines (the criterion 1 is waived). If this happens, the intersections have to be detected and resolved by swapping label box positions [1,2].

5.6. Defining the layout types

The layout of the label boxes is determined by two factors: the shape of the internal area A_I, and the set of principal directions D. A feasible leader line is the shortest line from an anchor to the silhouette of internal area A_I that is parallel with one of principal directions $\mathbf{d} \in D$. The shape of the internal area A_I then determines the length of leader line in the possible directions and the shortest one is chosen. In case that the set of principal directions is not restricted the leader line is simply the shortest line from the anchor to the silhouette of A_I.

If all directions are used as principal directions then both the directions of leader lines and the positions of the label boxes are only given by the shape of the internal area A_I. If only certain directions are used as principal directions then the directions of leader lines are given by the principal directions and the positions of label boxes are given by the shape of the internal area A_I.

6. Putting it together

In this section we briefly review the complete algorithm and describe some implementation details, particularly those connected with the GPU implementation of the method.

Table 1
Look up directions used in the jump flooding algorithm for different layout types.

Layout type	Principal directions	Jump flooding look up directions
Left	West	West
Right	East	East
Left-right	West, East	West, East
Top	North	North
Bottom	South	South
Top–bottom	North, South	North, South
Silhouette-based	All directions	West, East, North, South, Northwest, Northeast, Southwest, Southeast

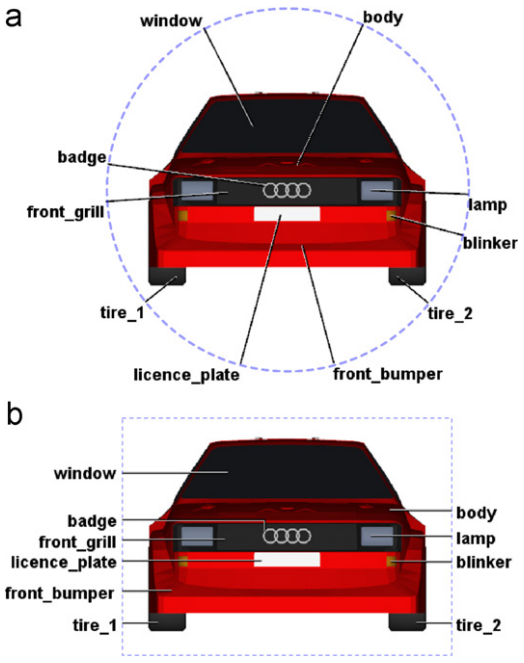


Fig. 6. Label layouts obtained by using: (a) a circle enclosing the 3D model, (b) a rectangle enclosing the 3D model instead of convex hull of the 3D model.

6.1. Summary of the algorithm

The proposed algorithm takes as input a 3D scene S consisting of n 3D objects $O_i, i \in \{1,\ldots,n\}$, the set of principal directions D, the parameters d_a and d_e, and the weights $w_1, \ldots, w_7$. The result of the algorithm is a set of leader lines $\mathbb{L} = \{\mathbf{l}_1 = (\mathbf{a}_1,\mathbf{e}_1), \ldots, \mathbf{l}_n = (\mathbf{a}_n,\mathbf{e}_n)\}$. The algorithm proceeds as follows:

1. Obtain the set of areas $\mathbb{A} = \{A_1,\ldots,A_n\}$ by computing the visibility of each object in scene S.
2. For each point $\mathbf{a}$ in internal area A_I find the closest point $\mathbf{e}$ on the silhouette of A_I such that $\mathbf{a} - \mathbf{e}$ is collinear with a direction $\mathbf{d} \in D$. Thus the leader line candidate $\mathbf{l} = (\mathbf{a}, \mathbf{e})$ is established and function $f_2(\mathbf{l})$ is calculated for each point $\mathbf{a}$ in A_I.
3. Calculate the length of the longest leader line candidate d_{max}.
4. For each point $\mathbf{a}$ in area $A_i, i \in \{1,\ldots,n\}$, calculate the function $f_3(\mathbf{l})$. That is, the distance of $\mathbf{a}$ to the nearest point on the silhouette of A_i.
5. Establish the set of leader lines $\mathbb{L} = \{\}$.
6. Calculate the feasibility $F(\mathbf{l})$ of each leader line candidate $\mathbf{l} = (\mathbf{a}, \mathbf{e})$.
7. While the set of areas is not empty, do
   (a) Calculate the priority P_i for each area A_i.
   (b) Select the area A_{max} with the highest priority.
   (c) Select the most feasible leader line candidate $\mathbf{l}_{max}$ with maximal $F(\mathbf{l})$.
   (d) Put the leader line candidate $\mathbf{l}_{max}$ into the set of leader lines $\mathbb{L}$.
   (e) Remove the area A_{max} from the set of areas $\mathbb{A}$.
   (f) Update functions f_3 and f_4 using $\mathbf{l}_{max}$ and reevaluate $F(\mathbf{l})$ for all leader line candidates in non-processed areas $\mathbb{A}$.
8. If necessary correct the label layout.

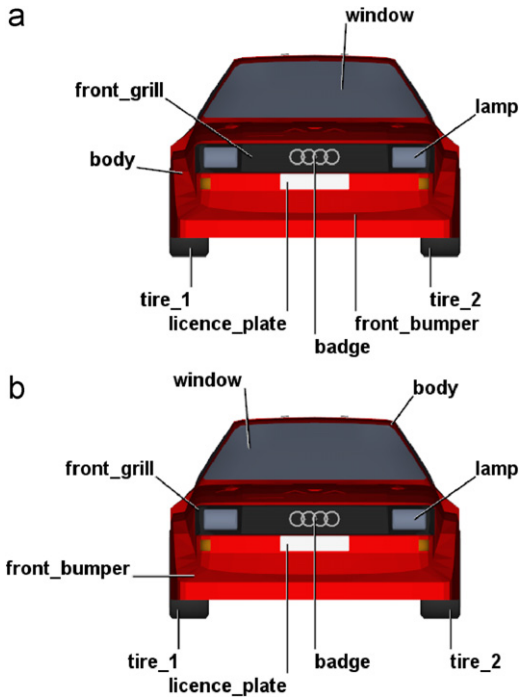


Fig. 7. The influence of the parameter d_a on the final labeling: (a) $d_a = 0.1$, (b) $d_a = 0.4$. We used $d_e = 0$ for both figures.

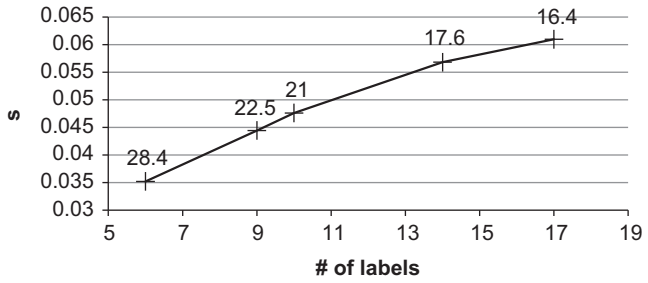


Fig. 8. The average time per frame in dependency on the number of labels. The number above each sample is the corresponding FPS rate.

6.2. Implementation details

We have implemented the presented algorithm as multi-pass rendering algorithm written in Java using OpenGL and JOGL [12].

Step 2 is calculated with jump flooding [14]. Jump flooding is a fast method, suitable for the GPU, for obtaining a Voronoi diagram of the seeds and the Euclidean distance of each pixel to the nearest seed. The pixels on the silhouette of A_I are used as the seeds. Note that for each pixel we need to find the closest point along one of the principal directions on the silhouette of A_I. Traditional jump flooding computes the Euclidean distance along all possible directions which is suitable for the silhouette-based layout only. For layouts with a restricted set of principal directions (e.g., left–right layout) we have modified the jump flooding algorithm to compute the Euclidean distance only along these directions. In Table 1 we show side-by-side the principal directions for major layout types and the corresponding look up directions used to evaluate minimal distances in the jump flooding algorithm. Note that apart from the metric used in distance evaluation the resulting layout can also be modified by changing the shape of the internal area A_I (as discussed

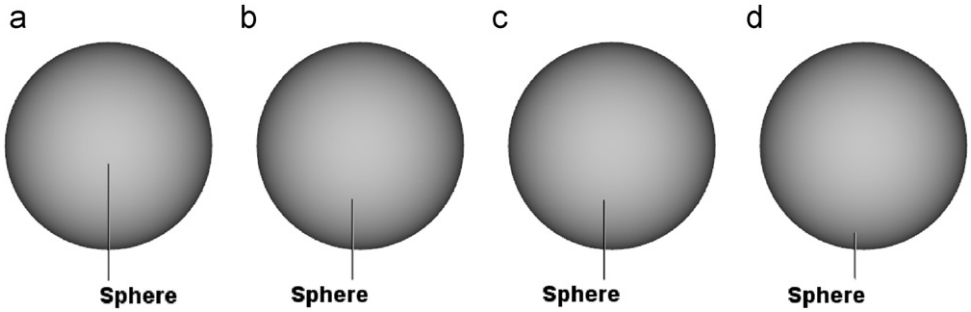


Fig. 9. A simple example showing the influence of the weights w_2 and w_3 on the labeling: (a) $w_2 = 0.2, w_3 = 1.0$, (b) $w_2 = 1.0, w_3 = 1.0$, (c) $w_2 = 0.2, w_3 = 0.2$, (d) $w_2 = 1.0, w_3 = 0.2$.

in Sections 5.6 and 7.1). An important aspect of jump flooding is that the texture in which we compute the distance has to be rectangular. Note that for representing the distances, the membership functions and their aggregations we use 16 bit floating point textures.

In step 3 we use scattering [16] to find the longest leader line candidate among all areas.

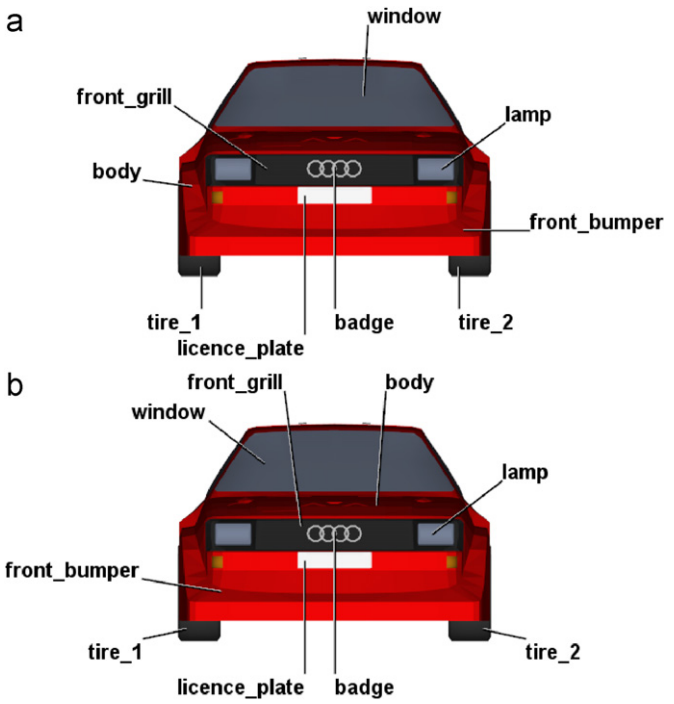


Fig. 10. The influence of the parameter d_e on the final labeling: (a) $d_e = 0.1$, (b) $d_e = 0.4$. We used $d_a = 0$ for both figures.

Step 4 is also calculated with jump flooding, where the seeds are pixels on the silhouettes of areas $A_1, \ldots, A_n$. For each pixel in area A_i, $i \in \{1, \ldots, n\}$, we calculate the Euclidean distance (along all possible directions) to the closest pixel on the silhouette of A_i. Note that the areas do not overlap each other, therefore we can process all areas at once.

Step 6 is implemented as a fragment shader working on a screen aligned quad. In the shader we calculate the feasibility $F(l)$ of each leader line candidate using Eq. (12).

In steps (a)–(c) we use scattering to compute the priorities P_i, selecting the area with the highest priority and selecting the most feasible leader line candidate.

In step 7(f) the functions $f_4(\mathbf{l})$ and $f_5(\mathbf{l})$ are calculated and multiplied with $F(\mathbf{l})$ using again a fragment shader operating on the screen aligned quad.

7. Results

In this section we present results of our solution of the external labeling problem and discuss the impact of the input parameters on the results. Further we compare our results with the method of Ali et al. [1].

7.1. Supported layout types

In order to demonstrate the flexibility of the labeling layouts produced by our method, we used our method with different shapes of internal area A_I and different sets of principal directions. When we use a convex hull of the 3D scene as internal area A_I and consider the distances only in certain principal directions we can obtain layouts presented in Figs. 1, 2, 7, 10, 11 and 12. When we use other shapes of the internal area we can obtain layouts as those presented in Fig. 6. In Fig. 6(a) principal directions correspond to all possible directions. In Fig. 6(b) only the east and west directions are used.

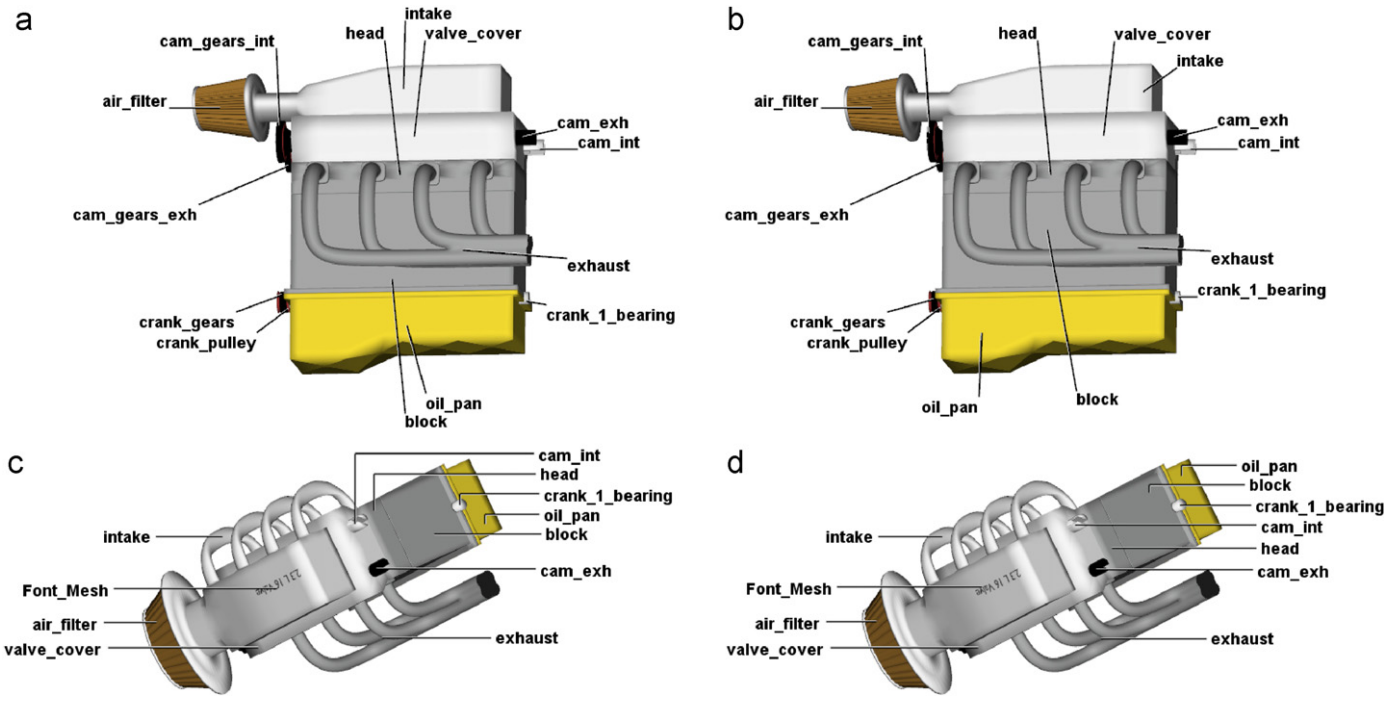


Fig. 11. A comparison of our method with the method of Ali et al. [1] on an engine model: (a) a silhouette-based layout produced using the method of Ali et al. ($d_a = 0.05$), (b) a silhouette-based layout using our method ($d_a = 0.05$, $d_e = 0.15$), (c) a left–right layout using the method of Ali et al. ($d_a = 0.15$), (d) a left–right layout using our method ($d_a = 0.15$, $d_e = 0.06$).

7.2. Distance of leader lines and distance of anchors

The input parameter d_a is used to specify the desired minimal distance between anchors. We express the distance as a scale of the bounding sphere diameter d_S. The impact of parameter d_a on the final labeling can be seen in Fig. 7.

The input parameter d_e is used to specify the desired minimal distance between the leader lines. Also here we express the distance as a scale of the bounding sphere diameter d_S. Our experiments have shown that superior values of d_e for horizontal layouts (left, right, and left–right) are similar to the height of the label box. For vertical layouts (top, bottom, and top–bottom) and radial layouts (silhouette-based) the superior values are similar to the average width of the label boxes. The impact of parameter d_e on the final labeling can be seen in Fig. 10.

Note that with increasing parameter d_a we can produce labeling where labels are more evenly distributed. However, certain anchors may become less salient (e.g. anchor to which label *body* is attached in Fig. 7(b)). When we also increase parameter d_e then the anchors are more resistant to this effect.

7.3. Influence of the weights

In this section we discus the impact of the weights on the final labeling. The weights are used to reduce the impact of the membership functions. In other words we use the weights to suppress significance of certain criteria and emphasize significance of the other criteria. As we use the same criteria for each type of layout we also use the same weights for all layout types. Note that we cannot change the type of layout by using different weights. We demonstrate the impact of the weights on a simple example in Fig. 9. In the example we use only criterion 1 (Leader line length) and criterion 2 (Anchor salience) and their respective weights.

The weights w_2 and w_3 influence the length of the leader lines and the salience of the anchors. The corresponding two criteria contradict each other and therefore a balance between them has to be found. Our experiments have shown that good values for w_2 and w_3 are: $w_2 = 0.2$ and $w_3 = 1$. In other words we prefer the salience of anchors over the length of the leader lines.

The weights w_4 and w_5 influence the distance between the anchors and the distance between the endpoints of leader lines. A similar effect can be achieved by using the parameters d_a and d_e.

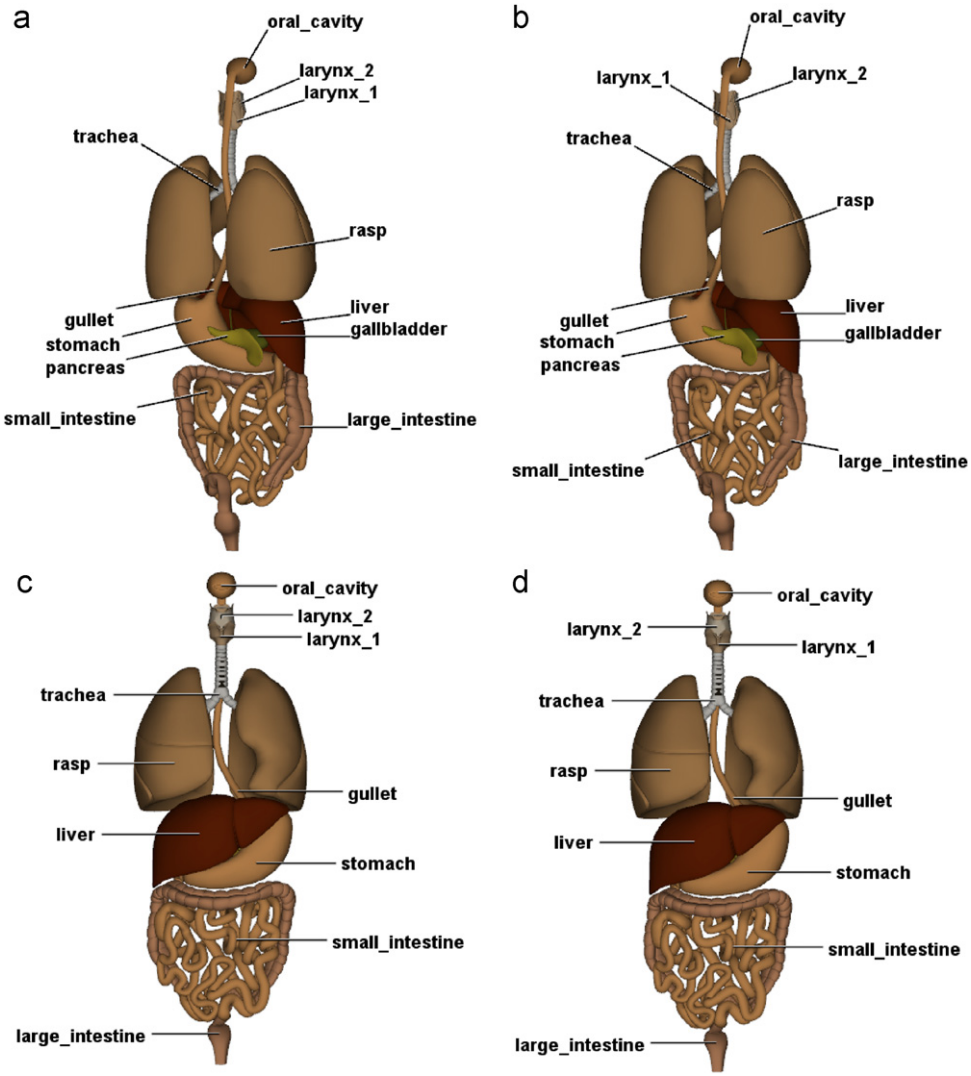


Fig. 12. A comparison of our method with the method of Ali et al. [1] on an anatomy model: (a) a silhouette-based layout produced using the method of Ali et al. ($d_a = 0.05$), (b) a silhouette-based layout using our method ($d_a = 0.05, d_e = 0.2$), (c) a left–right layout using the method of Ali et al. ($d_a = 0.05$), (d) a left–right layout using our method ($d_a = 0.05, d_e = 0.15$). Note particularly the difference in placing the larnix_1 and larnix_2 labels. Due to the endpoint distance criterion used in our method these labels are more uniformly distributed around the model.

These parameters provide more intuitive control over the labeling and therefore we recommend to set the weight w_4 and w_5 to 1 and use d_a and d_e to control the distance between the leader lines.

The weights w_6 and w_7 are used to reduce the impact of membership functions f_6 and f_7 that provide frame-to-frame coherence in interactive environments. Our experiments have shown that good values of w_6 and w_7 are: $w_6 = 0.7$ and $w_7 = 0.7$. These values appeared as best compromise between too glued anchors and too unstable leader lines.

Note that all figures in this paper obtained by our method were generated using weights with the above described values.

7.4. Comparison with state-of-the-art

In this section we compare the proposed method with our implementation of the method of Ali et al. [1]. A side by side comparison is shown in Figs. 11 and 12. As the method of Ali et al. does not consider criteria applying to the length of leader lines, the endpoint distance and the endpoint coherence it gives less aesthetic results for views where the most salient points of several areas are collinear. Note that for silhouette-based layout Ali et al. apply compulsive forces between the label boxes to reduce their uneven distribution. This step is omitted in our implementation of both their method and our method.

7.5. Performance

Fig. 8 shows the performance of our method in dependency on the number of labels. We can see that our method scales almost linearly with the number of labels. The measurements were done on a computer equipped with NVIDIA Geforce 8800 GT with 512 MB of RAM in resolution 512×512 pixels. Note that the number of labels depends on the number of visible objects. Typically the 3D model contains many more objects and only some of them are visible from a given viewpoint.

8. Conclusion

In this paper we have proposed a novel method for computing the labeling of 3D illustrations in real-time. We have shown how to reduce the dimensionality of the problem and how to formulate the labeling problem as a multiple criteria optimization problem. We solve the optimization problem using fuzzy logic combined with greedy optimization. We have implemented the presented method almost entirely on the GPU and the resulting implementation achieves interactive rates on medium-sized models. The results show that the method compares favorably to the state-of-the-art techniques for interactive external labeling. In particular, according to our opinion the method provides aesthetic labeling for various layout types and it is easy to fine tune the labeling by using a few intuitive parameters.

Acknowledgements

This research has been supported by the MSMT under the research programs LC-06008 (Center for Computer Graphics) and MEB060906 (Kontakt OE/CZ).

Appendix A. Supplementary material

Supplementary data associated with this article can be found in the online version of 10.1016/j.cag.2010.05.002.

References

[1] Ali K, Hartmann K, Strothotte T. Label layout for interactive 3D illustrations. Journal of the WSCG 2005;13(1):1–8.
[2] Bekos MA, Kaufmann M, Potika K, Symvonis A. Polygon labelling of minimum leader length. In: APVis '06: proceedings of the 2006 Asia-Pacific symposium on information visualisation. Darlinghurst, Australia: Australian Computer Society, Inc.; 2006. p. 15–21.
[3] Bekos MA, Kaufmann M, Symvonis A, Wolff A. Boundary labeling: models and efficient algorithms for rectangular maps. Computational Geometry 2007;36(3):215–36.
[4] Bellman RE, Zadeh LA. Decision-making in a fuzzy environment. Management Science 1970;17(4):B-141–64.
[5] Benkert M, Haverkort H, Kroll M, Nöllenburg M. Algorithms for multi-criteria one-sided boundary labeling. In: 15th international symposium on graph drawing, vol. 4875/2008. Berlin, Germany: Springer; 2008. p. 243–54.
[6] Götzelmann T, Hartmann K, Strothotte T. Agent-based annotation of interactive 3D visualizations. In: 6th international symposium on smart graphics, vol. 4073/2006. Berlin, Germany: Springer; 2006. p. 24–35.
[7] Götzelmann T, Hartmann K, Strothotte T. Annotation of animated 3D objects. In: Simulation und Visualisierung 2007 (SimVis 2007), Erlangen, Germany: SCS Publishing House; 2007. p. 209–22.
[8] Götzelmann T, Ali K, Hartmann K, Strothotte T. Form follows function: aesthetic interactive labels. In: Computational aesthetics 2005: eurographics workshop on computational aesthetics in graphics, visualization and imaging. Natick, MA, USA: A K Peters; 2005. p. 193–200.
[9] Hartmann K, Ali K, Strothotte T. Floating labels: applying dynamic potential fields for label layout. In: 4th international symposium on smart graphics, vol. 3031/2004. Berlin, Germany: Springer; 2004. p. 101–13.
[10] Hartmann K, Götzelmann T, Ali K, Strothotte T. Metrics for functional and aesthetic label layouts. In: 5th international symposium on smart graphics; vol. 3638/2005. Berlin, Germany: Springer; 2005. p. 115–26.
[11] Hensley J, Scheuermann T, Coombe G, Singh M, Lastra A. Fast summed-area table generation and its applications. Computer Graphics Forum 2005;24: 547–555.
[12] Java Binding for the OpenGL 〈http://kenai.com/projects/jogl/pages/Home〉.
[13] Maass S, Döllner J. Seamless integration of labels into interactive virtual 3D environments using parameterized hulls. In: Proceedings of the 4th international symposium on computational aesthetics in graphics, visualization, and imaging. Lisbon, Portugal: Eurographics Association; 2008. p. 33–40.
[14] Rong G, Tan T-S. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In: I3D '06: proceedings of the 2006 symposium on interactive 3D graphics and games. New York, NY, USA: ACM; 2006. p. 109–16.
[15] Ropinski T, Prassni J-S, Roters J, Hinrichs KH. Internal labels as shape cues for medical illustration. In: Proceedings of the 12th international fall workshop on vision, modeling, and visualization; 2007. p. 203–12.
[16] Scheuermann T, Hensley J. Efficient histogram generation using scattering on GPUs. In: I3D '07: proceedings of the 2007 symposium on interactive 3D graphics and games. New York, NY, USA: ACM; 2007. p. 33–7.
[17] Stein T, Décoret X. Dynamic label placement for improved interactive exploration. In: NPAR '08: proceedings of the 6th international symposium on non-photorealistic animation and rendering. New York, NY, USA: ACM; 2008. p. 15–21.
[18] Vollick I, Vogel D, Agrawala M, Hertzmann A. Specifying label layout style by example. In: UIST '07: proceedings of the 20th annual ACM symposium on user interface software and technology. New York, NY, USA: ACM; 2007. p. 221–30.
[19] Wolff A, Strijk T. The map-labeling bibliography 〈http://i11www.iti.uni-karlsruhe.de/~awolff/map-labeling/bibliography/〉.
[20] Yager RR. Fuzzy decision making including unequal objectives. Fuzzy Sets and Systems 1978;1(2):87–95.
[21] Zimmermann H-J. Description and optimization of fuzzy systems. International Journal of General Systems 1975;2(1):209–15.