Spatial Data Structures For Accelerated 3D Visibility Computation To Enable Large Model Visualization On The Web.

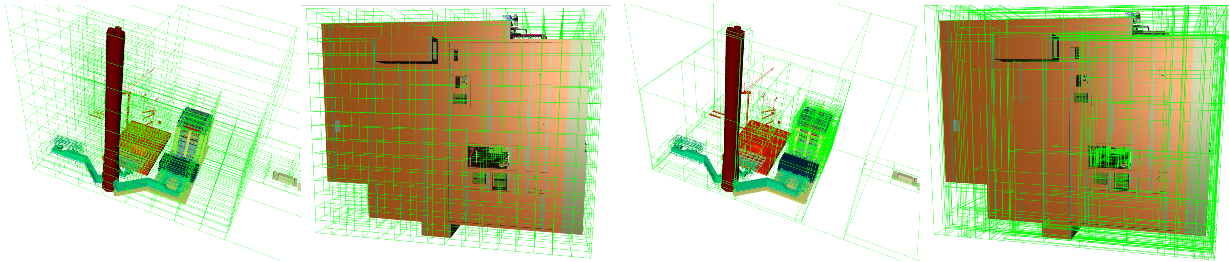
Christian Stein
christian.stein@igd.fraunhofer.de

Max Limper
max.limper@igd.fraunhofer.de

Arjan Kuijper
arjan.kuijper@igd.fraunhofer.de

Fraunhofer IGD, Darmstadt / TU Darmstadt

Figure 1: *Visualization of octree (left) and bounding interval hierarchy (right) for two powerplant models of ∼13/ ∼57 millions of triangles. Both structures are computed directly inside the browser. By accelerating the visibility determination, the data structures are the key to an interactive experience when rendering CAD data of such magnitude. Efficient hierarchies can be constructed in ∼20-30 ms already.*



Abstract

The visualization of massive 3D models is an intensively examined field of research. Due to their rapidly growing complexity of such models, visualisation them in real-time will never be possible through a higher speed of rasterization alone. Instead, a practical solution has to reduce the amount of data to be processed, using a fast visibility determination.
In recent years, the combination of Javascript and WebGL raised attention for the possibility of rendering hardware-accelerated 3D graphics directly in the browser. However, when compared to desktop applications, they are still fighting with their disadvantages of a generally slower execution speed, or a downgraded set of functionality.
We demonstrate the integration of spatial data structures, computed on the client side, using latest technology trends to mitigate the shortcomings of the 3D Web environment. We employ comparably small bounding volume hierarchies to accelerate our visibility determination, as well as to enable specific culling techniques. This allows for an interactive visualization of such massive 3D data sets. Our in-depth analysis of different data structures and environments shows which combination of data structure and visibility determination techniques are currently the best fit for the Web.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality I.3.6 [Methodology and Techniques]: Standards—Languages

Keywords: Spatial Data Structures, Culling, Visibility Determination, WebGL, X3D, X3DOM

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Web3D 2014, August 08 – 10, 2014, Vancouver, British Columbia, Canada.
2014 Copyright held by the Owner/Author. Publication rights licensed to ACM.
ACM 978-1-4503-3015-2/14/08 $15.00

1 Introduction

In the area of computer graphics, there is a long history of organizing 3D model data in hierarchical data structures. While for standardized rasterization traditionally the data is structured within scene graphs in a semantic way, a spatial structuring is able to supported the visibility determination. The set of scene elements is reduced during the visibility determination by a number of culling operations, each removing elements not contributing to the final image. For ray-tracing systems, spatial structures are typically static and constructed by a pre-processing step based on the set of triangles, while for collision detection they are based on bounding volumes and updated continuously. While various types of data structures have been intensively studied in these contexts, this knowledge has not been transferred to the Web environment so far, prevented, for example, by the comparably slow processing speed of available scripting languages. There have been efforts to use (pre-processed) server-side data structures for client-side rendering [Schwenk et al. 2013]. However, this paper aims for a server-independent, on-the-fly visualization of previously unknown model data. A concrete application scenario is the on-the-fly examination of only recently edited real-world CAD models, directly inside the browser - for example a manager reviewing the latest changes of his CAD expert group. In the last decade, JavaScript has evolved from being only the ”scripting language of the Web” to a true allrounder, used in almost any application area. This trend is driven by a widespread list of open-source libraries, frameworks or plattforms like node.js. By supporting only a very limited subset of the languages features, code conformant to the *asm.js* subset of JavaScript allows to mitigate its major drawback of slow processing speed. Thereby, asm.js enforces the progression of moving more and more logic to the client.
At the same time, the approaching arrival of WebGL 2 enhances the set of hardware features to be used in the Web, allowing for more sophisticated rendering methods and algorithms.
Hence, we propose the integration of spatial data structures for visibility determination, computed on the client side, inside the Web environment. This study covers the choice of data structure and techniques for visibility determination. Their respective trade-offs are evaluated across different environments. Our results show that, using the mentioned cutting-edge technologies, an interactive

visualization of massive 3D models on the Web becomes possible.

2 Related Work

2.1 Spatial Data Structures

The basic concept of spatial data structures is the exploitation of spatial coherence, based on the principle of recursive decomposition. On the highest level, the whole scene is contained in the root node and then gets subdivided on a space- or object-basis, depending on the chosen kind of data structure. Given a scene of *n* elements, and a branching factor of *m*, each node of the resulting tree of height $h = \lceil \log_m n \rceil$ hierarchically combines the bounding volumes of its children. That way, the spatial coherence can be exploited to limit the amount of volume-based computations, for example for intersection tests, to a number that grows only logarithmically with the number of objects inside the scene.
We categorize these hierarchies based on their characteristics, like their way of partitioning and the type of bounding volumes (boxes, spheres, etc.) used. Often these are inherently connected: for example, when using (axis-aligned) bounding boxes for the subdivision, overlaps and holes in the covered space can be easily avoided at the same time which is not possible for spheres.

Regular subdivisions The basic uniform spatial subdivision is the regular grid. In its simplest version, the space is uniformly partitioned by a raster of cells, conformable to a three dimensional array. Each cell contains a list of the scene objects with intersecting bounding volumes. The construction is based on the scene objects getting "rasterized" into the grid cells, which can be done in $O(n)$. Because multiple cells might hold the same object's reference, already visited objects can be tagged during traversal, a technique called *mailboxing*.
The octree can be thought of as the hierarchical extension of the regular grid. The use of macro cells results in additional levels of down-scaled versions of the grid [Meagher 1982]. For each node, the space is subdivided using axis-aligned planes for x-, y- and z-axis, passing through the center of the covered space, into eight subspaces, one for each child. Intersecting scene objects might be stored in every node, as well as only in the leaves. The octree can be adaptive in various ways, e.g. an adaptive maximal depth per branch or, similar to the BIH, the splitting planes could be offset in order to adapt to the scene objects' geometry. While this decreases the degree of regularity and the uniformity of the subdivision, it might trade off the adaptivity and/or impede other optimizations at the same time, which directly depend on such regularity (for example, the coherent grid traversal).

Irregular subdivisions The k-d tree [Bentley 1975] basically is a binary tree which contains a hyperplane at each node, splitting the covered space in two half-spaces. Typically, these hyperplanes alternatingly align with the *k* coordinate axes, but other/further ordering-dimensions are possible. [Wald et al. 2007b] rated the k-d tree to have the most efficient traversal, opposed by the most costly build. Construction in $O(n \log n)$ is possible by presorting and maintaining the sort-order [Wald and Havran 2006]. Further variations are multi-level [Wald et al. 2003] or fuzzy [Günther et al. 2006] k-d trees.
The surface area heuristic (SAH)[MacDonald and Booth 1990] minimizes the expected traversal costs by balancing the expected costs for possible child nodes, based on their surface area, weighted by the number of objects contained. Although the required calculations can be reduced by clever approximation [Hunt et al. 2006], it is still too expensive for the real-time demands of our target applications, especially when additionally considering the limited processing speed of the Web environment.
The *Bounding Volume Hierarchy* (BVH) has been studied intensively in the context of ray-tracing in the 1980s already [Kay and Kajiya 1986; Weghorst et al. 1984; Goldsmith and Salmon 1987]. The BVH references each object only once, and its nodes might contain the same space multiple times. By this partitioning of scene objects, it naturally adapts to the geometry, which is referenced only in its leaves. Each node's bounding volume recursively encloses the bounding volumes of its children minimally. While its traversal speed is close to the k-d tree's, the BVH has the superior construction performance [Wald et al. 2007b]. It is suited for dynamic as well as incremental updates [Larsson and Akenine-Möller 2006; Kopta et al. 2012].
The *Bounding Interval Hierarchy* (BIH) was proposed independently, with only minor differences, by three groups of researchers [Havran et al. 2006; Woop et al. 2006; Wächter and Keller 2006]. Focusing on a fast construction, the spatial median split is used as building strategy. Compared to the BVH the BIH is faster to build, easier to update and brings a smaller code layout, but trades off the BVHs tighter bounding. As the fast BVH-traversal could additionally be applied to BIHs with only minor modifications [Wald et al. 2007a], Wald et al. generally recommend the BIH for ray-tracing [Wald et al. 2007b] although the best approach certainly always depends on the actual problem.

2.2 Visibility

The term culling depicts the process of removing typically not/rarely visible elements from the set of geometry to be rendered. The purpose is the speed-up, accomplished by the rasterization having less/simpler items to process, which exploits the trade-off between the time saved in rendering costs and the time needed to determine the elements to be culled. Another trade-off relates to the granularity at which the scene data is processed. Considering different levels of granularity (object, polygon or pixel level) it determines the effectiveness of each respective technique: while a higher level implies less items, and therefore a faster processing, a lower level yields finer granularity, and thus higher accurracy. There are two categories of object visibility algorithms: Conservative techniques try to determine the set of visible objects, and thereby ususally overestimate the visibility consistently, while non-conservative techniques render objects in a simpler/cheaper way and gain performance at the cost of the resulting image's correctness.
This paper evaluates two conservative culling techniques. We implemented the hierarchical view frustum culling based on a bitmask [Assarsson and Möller 2000]. Out of the variety of occlusion culling algorithms rarely any are viable with the available performance in the Web environment yet. For example, the latest advances in software occlusion culling [Barbagallo et al. 2012; Chandrasekaran (Intel) 2013] rely on SIMD-instructions, which are not available in web browsers yet (Firefox's *Nightly* got experimental support for SIMD added very recently[1]). Instead, the occlusion culling implementations evaluated within this paper are based on the CHC++ algorithm [Bittner et al. 2004; Mattausch et al. 2008] and its *LatentQuery* extension. The CHC++ algorithm makes use of hardware occlusion queries, in order to determine invisible nodes within a hierarchical spatial data structure during rendering. Targeting for game engine optimization, the LatentQuery extension evaluates queries only in the next frame after they have been issued. Thereby, for very high frame rates, it aims to completely avoid any delays originating from queries being not ready for evaluation. A further difference is the way previously invisible nodes are batched

[1] https://twitter.com/FirefoxNightly/status/418774452739252224

into queries. It aggregates nodes as long as their multiplied probability of staying invisible stays above a given threshold, where the original algorithm adds nodes to a query until a local optimum of their averaged probabilities is exceeded. This allows the user to control the average amount of batched queries by adjusting the respective threshold.

2.3 JavaScript, Typed Arrays and asm.js

JavaScript usually runs client-side in the browser, in a so-called *sandbox environment*. Recently, with the arrival of node.js[2], it is also used standalone, and especially server-side, as well. One of JavaScript's biggest drawbacks is its comparably slow execution speed when compiled just-in-Time (JIT).
Lately, there have been two opposing developments, both enabling developers to bring their C/C++ (legacy) code to the browser. On one hand, the Native Client[3], a sandboxing technology, is developed as an open-source project by Google. On the other hand, asm.js[4] consists of a strict subset of the JavaScript language. By limiting the language's features, it allows for performance improvements like ahead-of-time optimization, pushing the execution speed closer to native code. Asm.js-code is quite unpractical to be written manually by the developer, but rather intended to be cross-compiled from other languages (e.g., C++), where the developers can take advantage of the tools they already have for those ecosystems.
Emscripten[5] is one such transcompiler: it takes bitcode as input and emits JavaScript-code. Depending on the setup and optimization level, this output ranges from rather standard JavaScript to asm.js. It runs in the so-called *Module* environment, a JavaScript object maintaining its own heap-structure based on TypedArrays[6], which encapsulates the cross-compiled functionality. Hence, for these objects living inside Module, developers have to take care of their lifetime manually. In return, the cross-compiled asm.js code runs significantly faster, compared to hand-written JavaScript. Emscripten received a lot of attention in May 2013, when Epic Games and Mozilla revealed their collaboration on employing emscripten to port the Unreal Engine 3 to the Web[7]. So far, with only Mozilla's Firefox fully supporting asm.js, the performance of cross-compiled code varies quite heavily across different browsers.

2.4 WebGL2

The first draft for the upcoming WebGL2 specification[8] was published on September 26, 2013. So far, the arrival of first official implementations is unforeseeable, yet in July 2013 already, Mozilla had released a prototypical, experimental *WebGL2* context[9]. After getting enriched continously during the following weeks, its feature set now includes, for instance, occlusion queries and transform feedback. So far, these features are only available in Firefox's *Nightly* builds, their activation depends on a set of browser and enviroment variables. Despite the experimental state of this WebGL2 implementation, the results of its evaluation can be assumend to match those of an official implementation, as it largely conforms to the specification draft.

[2] http://nodejs.org/
[3] https://developers.google.com/native-client/dev/
[4] http://asmjs.org/
[5] https://github.com/kripken/emscripten/wiki
[6] https://www.khronos.org/registry/typedarray/specs/latest/
[7] https://www.unrealengine.com/news/epic-games-releases-epic-citadel-on-the-web
[8] http://www.khronos.org/registry/webgl/specs/latest/2.0/
[9] https://wiki.mozilla.org/Platform/GFX/WebGL2

3 Testing Framework

3.1 Choice of Spatial Data Structures

Generally, spatial data structures might be using any type of bounding volume, branching with any favored factor, entailing the respective trade-offs. The selection of test candidates has to be well-considered: When targeting for on-the-fly visualization of previously unknown model data, any time-consuming pre-processing is not an option. This eliminates a vast number of bounding volume types already. Different spatial data structures can be compared most effectively if they are based on the same type of bounding volume: identical functionality can be used, rendering the evaluation indepedent of bounding volume properties. Regarding the ease of construction and the tightness of the composition, *axis-aligned bounding boxes* are best suited. Since the data structures will have to handle a set of objects, it could seem reasonable to pick structures which partition objects. In general, the geometric density of models is initially unknown and varies quite heavily. Therefore, the spatial data structures need to be adaptive.
The selected candidates are the *bounding interval hierarchy (BIH)* and the *octree*. The BIH, as a union of the k-d tree's and BVH's advantages, effectively partitions the space in an object-adaptive way. The same holds for the octree, if the maximal depth for the recursive construction adapts to the amount of drawables remaining in the respective branch.
Both have been studied intensively in the context of ray-tracing. However, this knowledge has not been transfered to the context of real-time rasterization inside the Web environment, with its specific properties and trade-offs, yet. For example, the unusual relation of processing speed between the CPU environment (built on JavaScript) and the GPU posts differing demands on the application of data structures for support the visibility determination. On the other hand, in contrast to their application in ray-tracing, a frequently repeated reconstruction might be one feasible way of achieving dynamic hierarchies. The candidates differ regarding their branching factor and subdivision strategy. Focusing on a fast construction and maximum comparability between the data structures, no construction heuristic is used. As the construction process is based on the axis aligned bounding volumes of complete geometries instead of triangles, the significance of heuristics like the SAH for this use case requires further investigation. The subdivision strategies are visualized in figure 2. The spatial median split is used for the BIH: starting with the scene's bounding volume, in every step the longest axis of the current bounding volume is chosen. All drawables of the current (sub-)set are sorted along this axis with respect to their bounding volume's center and split at the median. For both resulting subsets a plane is calculated and stored as offset along the chosen axis, splitting the original bounding volume and enclosing the subset's bounding volumes minimally. Often these subvolumes overlap. The recursion terminates as soon as the maximal depth is reached or if only a single drawable remains in the subset. In the latter case the nodes bounding volume is shrinked to the bounding volume of the drawable (not visualized in the figure).
The octree simply subdivides a node as long as more any of the drawables contained covers only a subset of the bounding volumes

Figure 2: *Subdivision scheme of octree(left) and BIH (right) in 2D. Narrower line-width means deeper in the hierarchy.*

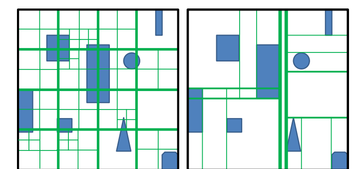
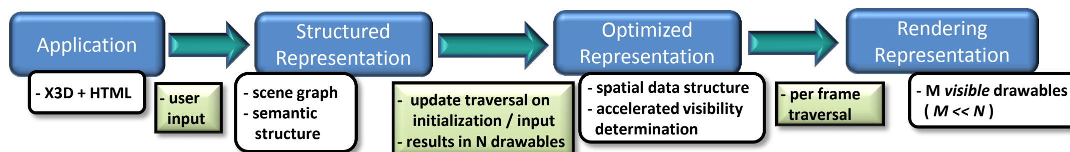


Figure 3: *Transitions between different representations of the scene data and their properties.*



of possible child nodes.
Within our study, spatial data structures are constructed and maintained based on the bounding volumes of the scene's objects. This coarse granularity compensates for the slow processing speed, being one of the major drawbacks of the Web environment. Though this leads to a lower accuracy of the visibility determination, it yields no loss in correctness of the resulting image.
As stated previously [Scherzer et al. 2010], the properties of the hierarchy, like the proper termination depth, can significantly influence the performance of any (occlusion) culling algorithm. Thus, the performance of the candidate data structures is evaluated with respect to different setups. The optimal depth levels of the given data structures for the respective test scenes are also evaluated to estimate their feasibility regarding client-side and on-the-fly construction and maintenance in the Web environment.

3.2 Implementation

The candidates were experimentally integrated in a branch of the x3dom[10] framework [Behr et al. 2009], a JavaScript library, which renders X3D[11], which was embedded into website's HTML, using WebGL. The scene objects targeted for processing in the spatial data structure are organized as subtree under a specific X3D grouping node. While the remainder is processed normally by x3dom, the list of drawables of this subtree is collected during the first scene graph traversal (cf. figure 3). The scene graph's semantic representation is broken up to create an optimized one, in form of the chosen spatial data structure. However, this structure stays connected to the scene graph, which serves as application interface. This allows x3dom to forward user input and trigger a rebuild of the spatial data structure, if required due to changes of the geometry.
With the help of a chosen spatial data structure, visibility determination is performed per frame, with respect to the enabled culling techniques. It results in the rendering representation: the set of drawables of the visible scene objects, enriched with additional information, like their current screen space coverage.
The spatial data structures, the different traversers using them, as well as the culling techniques, were implemented twice: on one hand in JavaScript, and on the other hand in C++ (cross-compiled to asm.js). From a software engineering perspective, the functionality was abstracted at various spots to allow for an efficient exchange of the used funtionality at runtime. Thus, it is possible to not only switch between cross-compiled and hand-written JavaScript, but also to freely switch between the different data structures or traversers. For an exchange of the data structure or variation of its parameters, a reconstruction of the hierarchy is inevitable so far. Particular attention has to be paid to the integration of the hardware occlusion queries. Althoug the emscripten crosscompiler is able to translate OpenGL calls to WebGL, the functionality of Mozilla's WebGL2 prototype has not been included yet. Thus, the needed WebGL2 functionality is wrapped in global JavaScript functions, which are then called from within the asm.js code. However, crossing the border between the asm.js Module enviroment and standard JavaScript is expensive, especially when occuring thousands

[10]http://x3dom.org
[11]http://www.web3d.org/realtime-3d/x3d/what-x3d

Figure 4: *Hardware specifications and browser versions of the testing machine.*

Property	**Value**
CPU	Intel Core i7-4770 @3.4 GHz
RAM	32 GB
Gfx	GeForce GTX 770
OS	Windows 7 64 bit SP1
Browser	**Version**
Chrome	33.0.1750.154 m
Firefox(Nightly)	31.0a1

of times during a traversal including occlusion culling.
Another important aspect is the efficient application hardware occlusion queries. After the construction of the spatial data structure, all bounding volumes are known, for its (inner) nodes as well as for the set of drawables. The difficulties arise because the CHC++ algorithm batches entirely different subsets of bounding volumes for each and every occlusion query. Thus, in order to draw those bounding boxes, different subsets of vertices have to be rendered up to a multiple of thousand times per frame. In a native application, updating and passing a client-side index array, stored in main memory, seems to be a promising approach to draw such highly frequent changing data, yet this feature is not available in WebGL. Both options, the deletion and recreation of *vertex buffer objects*, as well as overwriting their content with *glBufferSubData*, came out not to be viable on a per frame basis, at least required in the magnitude. Thus, although draw calls are a huge performance bottleneck for WebGL, for each and every bounding volume a single draw call has to be issued. Thus, the same index buffer and vertex buffer, representing a single unit-size axis-aligned bounding box, is positioned, scaled and oriented by a transformation matrix uniform, for the rendering of each bounding box.

4 Evaluation

The performance is evaluated for two different browsers: The key aspect of this evaluation are the differences in performance for the four possible combinations of the standard JavaScript or asm.js enviroment, executed by the Browsers *Chrome* or *Firefox (Nightly)*.
Note that Chrome does not directly support asm.js. However, based on internal optimizations, Chrome is able to execute such code significantly faster compared to standard JavaScript[12].
For evaluations which are not directly related to rendering, the test scene contains five thousands randomly positioned utah teapots. A scene with that many elements is needed to force the BIH construction to result in a hierarchy consisting of roughly the same amount of nodes as an octree of depth five or higher. For the evaluation of the culling techniques, two powerplant models (see. figure 1) with individual geometric properties are used. The first well-known powerplant (left side), supplied by the University of North Carolina

[12]https://blog.mozilla.org/futurereleases/2013/11/26/chrome-and-opera-optimize-for-mozilla-pioneered-asm-js/

at Chapel Hill[13] (ca. 13.6 millions of triangles) delivers a moderate til high geometric density, whereas the powerplant model from EDF (Électricité de France, right side), consisting of roughly 57 millions of triangles, represents a very demanding real-world model.
The geometry is loaded in form of x3dom's BinaryGeometry [Behr et al. 2012], which provides a great loading and rendering performance. However, during conversion to this format, the geometry chunks that represent the mesh data are optimized, to reduce draw calls. This has a negative impact on the occlusion properties, a fact detailed in section 4.3.
All measures are based on the JavaScript-function *performance.now()*. Its implementation is browser-dependent: While it accurately returns fractions of milliseconds in Firefox, Chrome returns millisecond values only. This introduces errors not negligible, especially when measuring very short time intervals. We will refer to this later on.

4.1 Construction Time

The BIH's node count completely depends on the overall amount of drawables. If every leaf node contains only one drawable, it simply can not grow deeper. In contrast, the octree's maximal depth, when using the given strategy, depends on the positioning and size of the bounding volumes. Compared to the BIH, the octree grows that much faster, that its subdivision usually only stops when reaching the *maximal depth*.
The construction time was measured as the average time consumed over a series of 50 constructions for the teapot test scene, processed in consecutive frames. The results for the four environments are listed in figure 5. The maximal depth parameters are impossible to relate for spatial data structures with differing branching factors and construction algorithms. Hence, construction times are contrasted with respect to the resulting node count. Generally, the BIH's construction performance for similar node counts is a trifle better compared to the octree, as the results of the js-environment show. This is reasoned in the construction algorithms: while for the BIH with increasing depth a decreasing number of drawables is processed, depending on the spatial distribution this number might stay rather constant for the octree.
For node counts above one thousand a re-construction is already too expensive to be done per frame. Firefox delivers the better performance.
Both spatial data structures were chosen because of their ability to adapt to the geometry. Figure 1 shows this similarity of the resulting hierarchies.

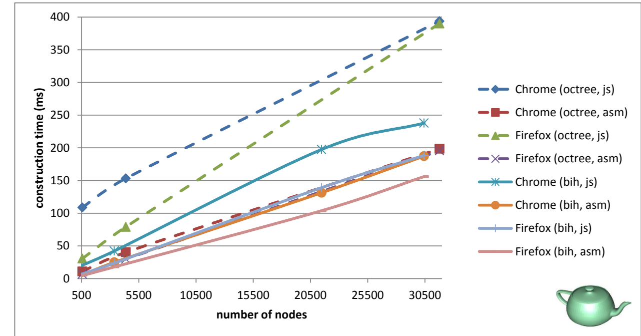
4.2 Hierachical Traverse Performance

The hierarchical traversal of the spatial data structure simply collects the drawable. It does not distinguish the different structures: for the traversal of a certain node count it needs the same amount of time, if no additional calculations, e.g. with respect to culling techniques, are performed. Initially, we compare this traversal performance for the different environments to find an estimate for the maximally manageable node count if all culling techniques are deactivated (cf. figure 6). Because of the generally unstable JavaScript performance, the traversal times are averaged over a number of subsequent renderings of the teapot scene. It is important to note that, for the asm.js-environment, the costs of cross-environment calls add up to the traverse time. While these calls did not matter for the construction (one environment switch), during traversal there is one cross-border-call for each visible drawable. The traverse times for Chrome are not as accurate as for Firefox, because of the lower precision of its implementation of *performance.now()*. Hence, these results have a margin of uncertainty of at least 0.5 milliseconds.

[13] http://gamma.cs.unc.edu/POWERPLANT/

Figure 5: *Construction times for octree and BIH for a scene with 5k elements within different environments in ms.*

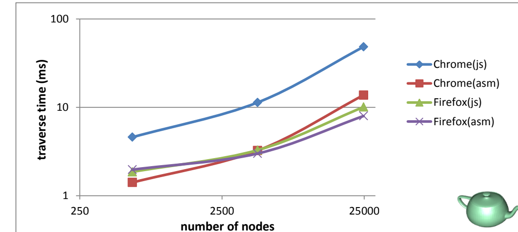
			Chrome		Firefox	
			js	asm	js	asm
depth	#nodes	#leaves	octree			
3	577	505	108.7	10.9	30.5	**6.7**
4	4377	3830	153.2	40.2	79.1	**29.9**
5	31777	27805	393.7	198.5	390.1	**195.4**
			BIH			
7	483	242	19.7	4.7	6.6	**4.1**
10	3367	1684	42.2	24.9	23.6	**17.8**
30	21469	10735	197.4	131.3	139.0	**103.8**
50	30429	15215	237.8	187.2	188.2	**156.0**



The results are similar to those of the construction evaluation: generally, Firefox performs better, but only Chrome's js-variant is really falling behind, while its asm.js-variant can keep up. In contrast to the asm-environment, standard JavaScript struggles with the garbage collection, uncontrollably kicking in and decreasing performance considerably then.
Defining a threshold of 10 ms as maximum for the hierarchical traversal excluding any further calculations, a node count of up to 25k is feasible when using the asm.js environment.

Figure 6: *Average traverse performance for octrees of varying depth and different browsers/environments in ms.*

		Chrome		Firefox	
depth	#nodes	js	asm	js	asm
3	585	4.59	**1.41**	1.86	1.97
4	4441	11.34	3.24	3.28	**3.00**
5	24745	48.38	13.75	10.09	**7.98**



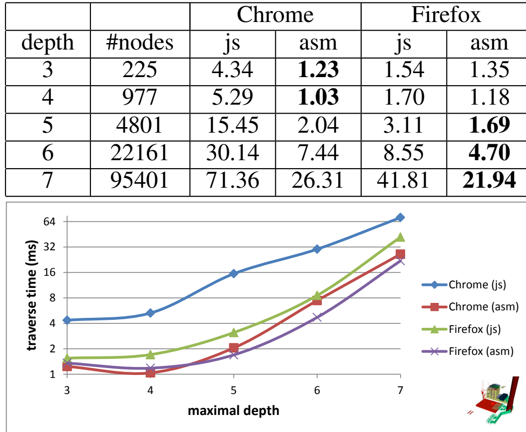
4.3 Visibility Determination

The visibility determination reduces not only the amount of rendered drawables, but also the amount of traversed nodes. When counterbalancing the costs of the required calculations against the

saved rendering costs, conservative culling techniques can be evaluated quite reasonably. This information is implicitly contained in the frame time.
The culling techniques are evaluated in combination with each spatial data structure, and a reasonable range of maximal depth values. We use the powerplant models for these analyses to most strongly conform to real-life-applications.

Figure 7: *Traverse times using view frustum culling for octrees of varying depth in ms.*

		Chrome		Firefox	
depth	#nodes	js	asm	js	asm
3	225	4.34	**1.23**	1.54	1.35
4	977	5.29	**1.03**	1.70	1.18
5	4801	15.45	2.04	3.11	**1.69**
6	22161	30.14	7.44	8.55	**4.70**
7	95401	71.36	26.31	41.81	**21.94**



View frustum culling Figure 7 shows the average traverse times (rendering times excluded) for a walkthrough of the smaller powerplant (13.6M triangles), using view frustum culling: only Chrome with standard JavaScript falls considerably behind. Up to a maximum depth level of six (ca. 20k nodes), the traverse times are low enough to allow for an interative experience. An interesting detail are the higher traverse times for 255 nodes than for 977 nodes when using asm.js. This directly depends on the less efficient view frustum culling, resulting from a comparably bad granularity of the spatial subdivision as a result of the node count being too low. This gets exacerbated by a higher number of cross-environment calls, as more drawables are returned as being visible than actually should.
This comparison indicates increasing benefits for the utilization of asm.js for a rising node count. As Firefox with asm.js showed the best performance, it is used to analyse the effects of varying spatial data structures and depth levels on view frustum culling. By using the same scene for all measurements, the resulting frame time can be used as performance indicator. Figure 8 lists the view frustum culling results for BIH and octree of varying depth levels. The plot visualizes the direct correlation between the number of traversed nodes (complement of the culled nodes) and the frame time (smaller = better). We see the octree generally performing better than the BIH, the optimal node count (corresponding to the highest frame rate) for both of them is located somewhere between five and fifteen thousand nodes. A higher node count of the BIH does not generally introduce additional calculations for view frustum culling: most of the additional nodes form subtrees under (previous) leaves which might already have been completely in- or outside the frustum. The frame time of the BIH stays almost constant for higher node counts, while the percentage of traversed nodes is falling. In contrast, for the octree the percentage of traversed nodes and most notably the frame time are rising distinctively.
As mentioned previously, on one hand, for very low node counts the hierarchies' granularity does not allow efficient view frustum culling which results in unnecessary rendering costs. On the other hand, for very high node counts, the increase in accuracy of the culling does not have to pay off anymore. The immense traversal effort dominates the spared rendering costs, as illustrated by the strong decrease of the render time percentage. Certainly, this "barrier" totally depends on the details of the scene and their relation to the viewing area.
Figure 9 further depicts the different view frustum culling be-

Figure 8: *View frustum culling costs in ms for Firefox with asm.js employing BIH and octree, indicating a correlation between the amount of nodes traversed and the resulting frame time as result of the best fitting data structure granularity for the given scene.*

BIH					
depth	10	30	50	100	400
# nodes	183	4635	9319	21019	91219
# traversed	153.9	3441.5	5139.3	11946.7	48759.2
% traversed	84.1	74.2	**55.1**	56.8	53.4
frame time	37.1	30.1	**24.4**	25.8	27.3
render time	35.2	28.0	**22.5**	23.3	21.9
% rend. time	94.7	93.3	92.3	90.1	80.3
octree					
depth	3	4	5	6	7
# nodes	225	977	4801	22161	95401
# traversed	124.7	457.9	2042.5	9866.5	46438.1
% traversed	55.4	46.8	**42.5**	44.5	48.6
frame time	26.7	19.5	**18.2**	21.3	39.4
render time	25.1	18.1	**16.3**	16.4	17.3
% rend. time	94.1	93.0	89.7	77.1	44.0

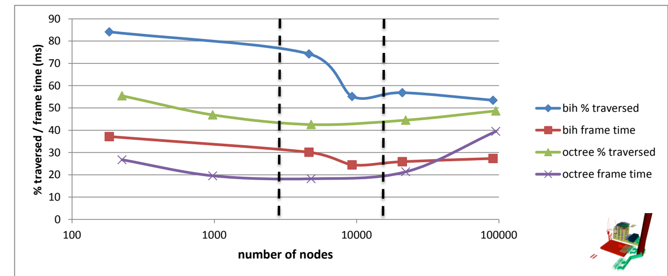


Figure 9: *View frustum culling frame time and percentage of traversed nodes during a walkthrough for BIH and octree.*

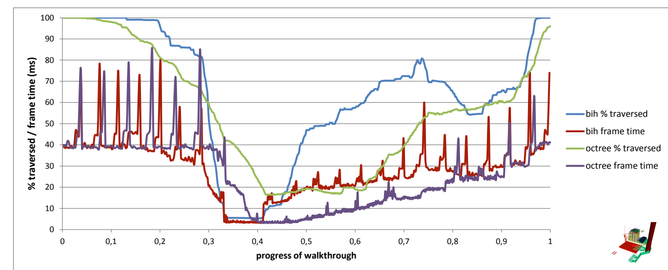
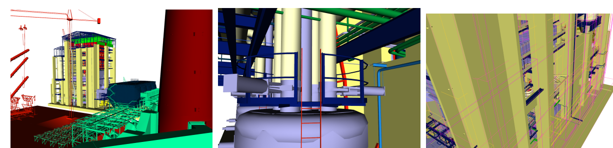


Figure 10: *Different points of view during a walkthrough: outside (left) and inside (middle). Bounding volumes reaching out to far, thus interfering with occlusion culling.(right)*



haviour of BIH and octree for the best performing maximal depths of 5 and 50. During a short period of the walkthrough, the camera is positioned outside of the geometry (cf. figure 10 - left): frametime- as well as culling-wise the BIH slightly outperforms the octree temporarily here. Yet, throughout the majority of the walkthrough, the camera moves completely inside the data structure (cf. figure 10 - middle): here, the octree performs better.

Occlusion culling Occlusion culling ist not a conservative culling technique per se, a visibility threshold greater than one pixel (any geometry occupying less pixels is culled) makes it a non-conservative one. However, the current Firefox WebGL2 prototype does not support other/user-defined thresholds. Therefore, we evaluate it by comparing the frame times, as well. Up to this point, Chrome does not support the required WebGL2 features, hence all measurements were taken in Firefox.
During conversion to BinaryGeometry, using the AOPT tool[14], the optimization for reduced draw calls batches geometry sharing the same material. Sometimes, this yields a bad fitting of the resulting bounding volumes. Figure 10 (right) visualizes this problem: Because of small (possibly visible) parts, the bounding volumes of mostly hidden geometries are reaching out too far and thereby interfere with the occlusion culling algorithm. A query, rendering a hierarchy node which contains these "empty" parts of bounding volumes, will falsely determine the corresponding geometry as visible. All occlusion culling measurements were taken from walkthroughs of the EDF powerplant model.
The benefit of combining view frustum culling with occlusion culling is shown in figure 13. For an octree of depth 4 (4681 nodes), there is already a notable benefit. Yet, for a depth of 5 (37097 nodes), the results are most impressive, as the frame times for activated occlusion culling are very close to those of depth 4, while for a node count that high, with only view frustum culling activated, interactive rendering had not been possible anymore.
Octree and BIH were contrasted by comparing the results for hierarchies of comparable granularities. Again, as illustrated in figure 14, the octree outperforms the BIH for the majority of the walkthrough while the camera is moving deeply inside the spatial data structure. For this part, the query time for the octree is lower as well. Yet, for the last 20 percent of the walkthrough, when the camera is looking from the outside, the BIH shows better results considering both, query time and frame time.
Both occlusion culling traversers, CHC++ and LatentQuery, process the hierarchy nodes in order of their distance to the camera position. Therefore, the nodes are managed in a queue, sorted by their distance. When a node is traversed and its children are not culled, they are inserted into this queue. Figure 11 breaks down the average times spent on different tasks per frame during an occlusion culling traversal. It visualizes the relations as fractions normalized to the frame time using an octree of varying maximal depth. For the first levels, the increasing node count delivers better results, based on the finer granularity of the occlusion tests. Yet, later on, the joined effort of traversal and occlusion queries for the rapidly increasing node count gets too high, resulting in a performance collapse. The sort time corresponds to the time spend to keep this queue sorted. The traverse time consists of the time for the traversal of the hierarchy including view frustum culling, as well as the calculations for the query batching.
Figure 12 compares the average costs of the different tasks for occlusion culling with an octree of depth 5 (37097 nodes) using js/asm.js. The time spend on tasks involving CPU calculations gets reduced significantly by the use of asm.js. In figure 15, the query and frame time are compared for the two occlusion culling variants. The evaluation of queries within the same frame allows the CHC++ to instantly perceive and react to changes like camera movement and thus deliver a more accurate occlusion determination. However, the time elapsing while the traverser waits for a query result to become retrievable is costly and weights even heavier when aiming for very high frame rates. If such a high frame rate, for example in games, has been achieved, the one-frame-delay before the algorithm reacts on changes is probably neglegible.

Figure 11: *Breakdown of average times spent during a frame by the CHC++ in ms normalized to the frame time.*

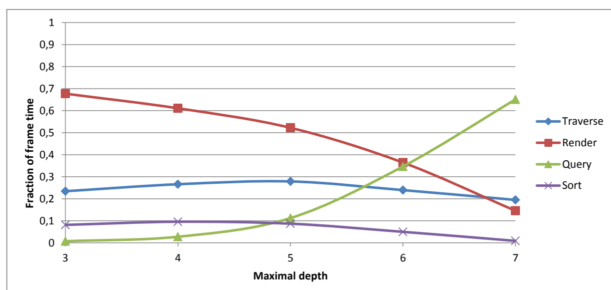
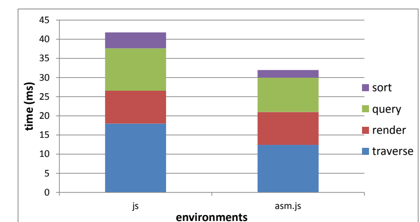


Figure 12: *Average times per task for js and asm.js in ms for CHC++ and octree.*



5 Conclusion and Future Work

In this paper, client-side computed spatial data structures were shown to be feasible for the Web. The shortcomings of the Web environment were mitigated through the use of the latest technology trends. Both, asm.js and the WebGL2 features, contributed significantly to allow for an interactive visualization of massive 3D models. Besides the speedup, the utilization of asm.js prevents performance break downs due to JavaScript garbage collection. However, the integration still has room for improvement, for example direct read/write on asm.js's TypedArray heap from both environments, to decrease the amount of cross-border calls. Further investigation needs to be invested on the memory footprint of these techniques and its optimization, being a particularly important aspect for both, large model visualization and the web environment. A considerable improvement in stabilty and conformance of the WebGL2 features is likely to happen within the near future, which will further improve the gain.
The evaluated data structures proved to be efficient in not only accelerating the visibility determination, but also allowing for more complex algorithms like CHC++ to be performed. View frustum culling profited notably when combined with a spatial data structure. An efficient occlusion culling, the key for the interactive visualization of massive 3D data sets, is achievable by employing comparably small hierarchies already. The octree showed better results when the camera was completely inside the spatial data structure, while the BIH performed better for the rest. We believe this to be a result of the BIH being more affected by problematic bounding volumes than the octree. The selected culling techniques were a good choice to cover scenarios of different geometric

[14] www.instantreality.org

Figure 13: *Comparison of simple* ***view frustum culling*** *with* ***CHC++*** *(combined of* ***occlusion and view frustum culling****) employing the* ***octree*** *of maximal depth levels 4 and 5. The CHC++ with the octree of level 4 performs best. The application of occlusion culling immensly improves the performance for the previously unmanageable node count of the octree of level 5 (37097 nodes). It outperforms the level 4 octree with only view frustum culling for the majority of the walkthrough. Compared to the level 4 octree, the higher granularity for occlusion culling never improves the frame time for a longer period of the walkthrough.*

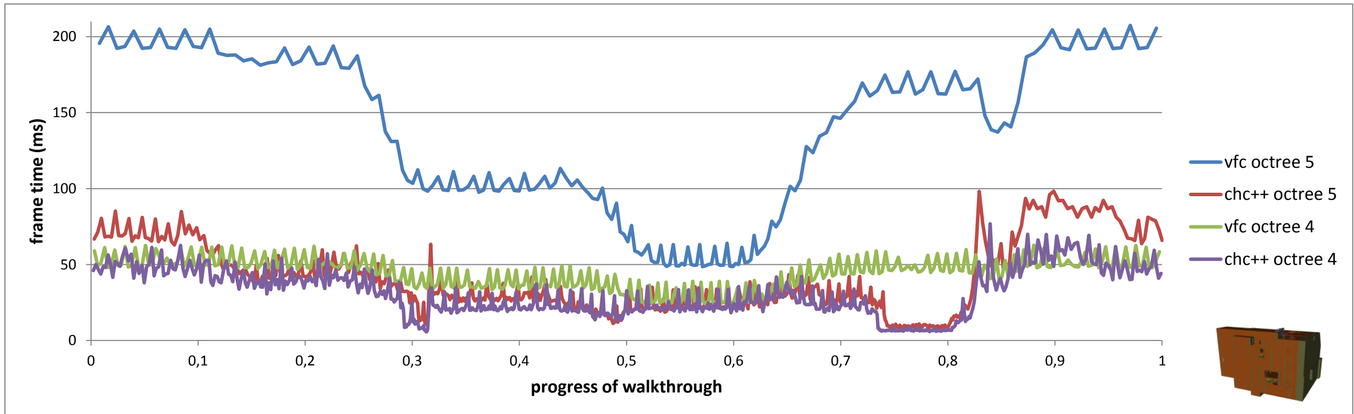


Figure 14: *Comparison of* ***octree****(maximal depth 4, 4861 nodes) and* ***BIH*** *(maximal depth 50, 8101 nodes) with the* ***CHC++*** *traverser (view frustum and occlusion culling) activated: The* ***octree*** *invests less time for queries, while delivering a better culling, resulting in a lower frame time, for the majority of the walkthrough.*

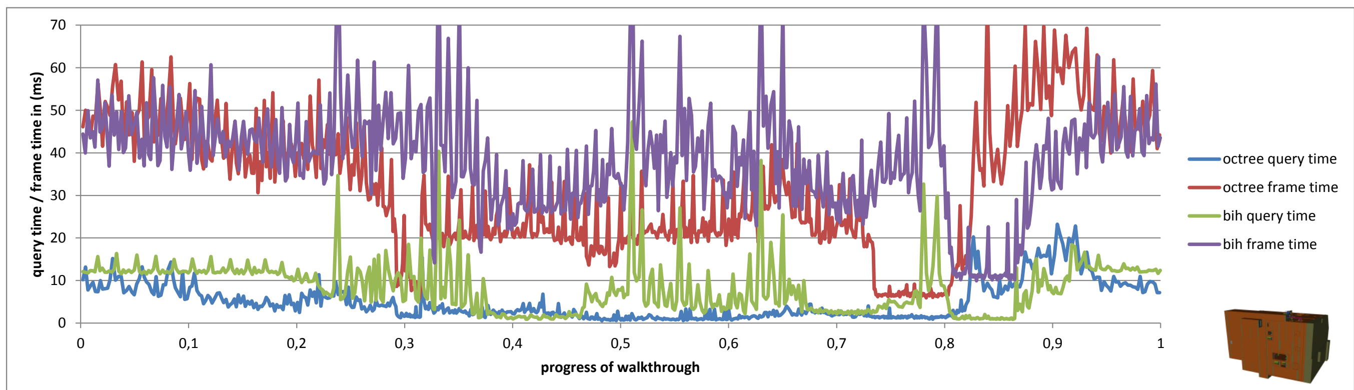
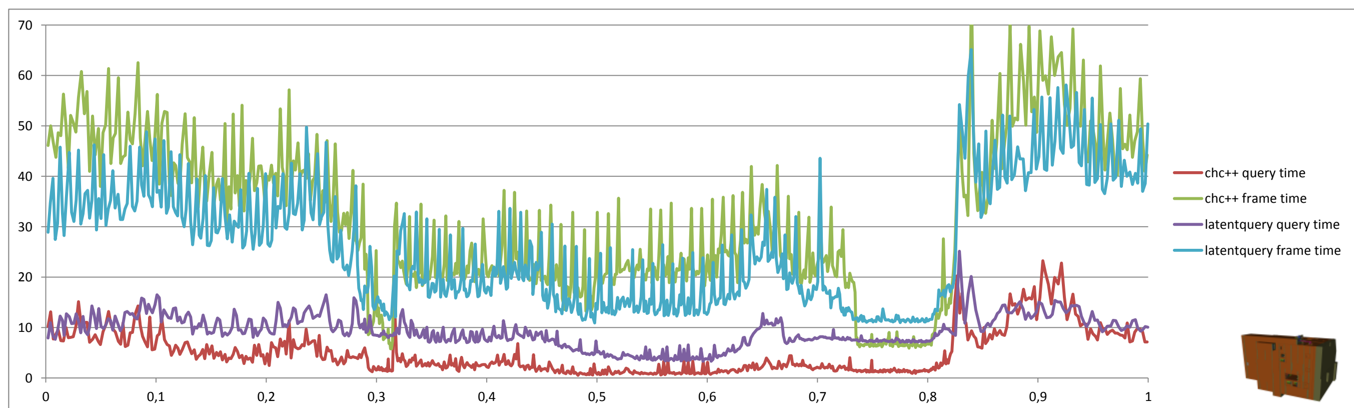


Figure 15: *Comparison of* ***CHC++*** *and* ***LatentQuery*** *traversers's query time versus overall frame time for an* ***octree*** *(max. depth 4). By evaluating queries in the same frame they have been issued in, the CHC++ traverser is able to perceive changes of the visibility immediately. In return for the image error, resulting from evaluating its queries only in the next frame, and although spending more time on queries, the LatentQuery traverser delivers the overall lower frame times and thus better performance.*



distribution and density. The optimal depth for a data structure is influenced by a wide range of properties of both, the scenario and the hardware. This is especially true for the latter, as any wasted processing power is the more significant the less performance the environment provides. The optimal depth of the data structures is particularly dependent on the respective scene properties. At best, such a system would initially classify the performance of its current environment to act on this basis.
So far, only static scenes are fully supported, dynamic changes force a rebuild of the data structures. The construction time for moderate node counts is already too high to allow for per-frame rebuilds. While the effort to support dynamic scenes is comparably low for the octree, most easily done by only re-inserting the respective geometry into the hierarchy, for the BIH more sophisticated methods like (re-)balancing are the strategy of choice. Such update strategies simply adjust the modified nodes' bounding volumes, which push the changes up to their parents. After a certain amount of degradation, the hierarchy gets "repaired", a task to be possibly executed in the background using web workers.

References

ASSARSSON, U., AND MÖLLER, T. 2000. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools 5*, 9–22.

BARBAGALLO, L. R., LEONE, M. N., BANQUIERO, M. M., AGROMAYOR, D., AND BURSZTYNR, A. 2012. Techniques for an image based occlusion culling engine. *Argentine Congress on Computer Sciences (CACIC)*.

BEHR, J., ESCHLER, P., JUNG, Y., AND ZÖLLNER, M. 2009. X3dom: A dom-based html5/x3d integration model. In *Proceedings of the 14th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '09, 127–135.

BEHR, J., JUNG, Y., FRANKE, T., AND STURM, T. 2012. Using images and explicit binary container for efficient and incremental delivery of declarative 3d scenes on the web. In *Proceedings of the 17th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '12, 17–25.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (Sept.), 509–517.

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

CHANDRASEKARAN (INTEL), C., 2013. Software occlusion culling update. http://software.intel.com/en-us/blogs/2013/09/06/software-occlusion-culling-update-2, Last Visited: 31.03.14.

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

GÜNTHER, J., FRIEDRICH, H., WALD, I., SEIDEL, H. P., AND SLUSALLEK, P. 2006. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum 25*, 3 (sep), 517–525. (Proceedings of Eurographics).

HAVRAN, V., HERZOG, R., AND SEIDEL, H. P. 2006. On the fast construction of spatial hierarchies for ray tracing. In *Interactive Ray Tracing 2006, IEEE Symposium on*, 71–80.

HUNT, W., MARK, W., AND STOLL, G. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. In *Interactive Ray Tracing 2006, IEEE Symposium on*, 81–88.

KAY, T. L., AND KAJIYA, J. T. 1986. Ray tracing complex scenes. *ACM SIGGRAPH Computer Graphics 20*, 4 (Aug.), 269–278.

KOPTA, D., IZE, T., SPJUT, J., BRUNVAND, E., DAVIS, A., AND KENSLER, A. 2012. Fast, effective bvh updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, 197–204.

LARSSON, T., AND AKENINE-MÖLLER, T. 2006. A dynamic bounding volume hierarchy for generalized collision detection. *Computer Graphics 30*, 3 (June), 450–459.

MACDONALD, D. J., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer: International Journal of Computer Graphics 6*, 3 (May), 153–166.

MATTAUSCH, O., BITTNER, J., AND WIMMER, M. 2008. Chc++: Coherent hierarchical culling revisited. *Computer Graphics Forum (Proceedings Eurographics 2008) 27*, 2, 221–230.

MEAGHER, D. 1982. Geometric modeling using octree encoding. *Computer Graphics and Image Processing 19*, 2, 129 – 147.

SCHERZER, D., YANG, L., AND MATTAUSCH, O. 2010. Exploiting temporal coherence in real-time rendering. In *ACM SIGGRAPH ASIA 2010 Courses*, ACM, New York, NY, USA, SA '10, 24:1–24:26.

SCHWENK, K., VOSS, G., BEHR, J., JUNG, Y., LIMPER, M., HERZIG, P., AND KUIJPER, A. 2013. Extending a distributed virtual reality system with exchangeable rendering back-ends. *The Visual Computer 29*, 10, 1039–1049.

WÄCHTER, C., AND KELLER, A. 2006. Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006 Proceedings of the 17th Eurographics symposium on Rendering*, 139–149.

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 the 2006 IEEE symposium on interactive ray tracing*, 61–70.

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Distributed interactive ray tracing of dynamic scenes. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, IEEE Computer Society, PVG '03, 11–.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (Jan.).

WALD, I., MARK, W. R., GÜNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. 2007. State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics 2007*, D. Schmalstieg and J. Bittner, Eds., 89–116.

WEGHORST, H., HOOPER, G., AND GREENBERG, D. P. 1984. Improved computational methods for ray tracing. *ACM Transactions on Graphics 3*, 1 (Jan.), 52–69.

WOOP, S., MARMITT, G., AND SLUSALLEK, P. 2006. B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *Proceedings of the 21st ACM SIGGRAPH/Eurographics symposium on Graphics hardware*, ACM, GH '06, 67–77.