Eurographics Symposium on Parallel Graphics and Visualization (2016)
W. Bethel, E. Gobbetti (Editors)

Web-enabled server-based and distributed real-time Ray-Tracing

G. Tamm[1,2] and P. Slusallek[1,2]

[1]DFKI Saarbrücken, Germany
[2]Saarland University, Germany

Abstract

As browsers expand their functionality, they continuously act as a platform for portable application development within a web page. To bring interactive 3D graphics closer to the web developer, frameworks allowing a declarative scene description in line with the HTML markup exist. However, these approaches utilize client-side rendering and are thus limited in the scene complexity and rendering algorithms they can provide on a given device. We present the extension of the declarative 3D framework XML3D to support server-based rendering. The server back-end enables distributed rendering with an arbitrary hierarchy of cluster nodes. In the back-end, we deploy a custom real-time ray-tracer. To distribute the ray-tracer, we present a load balancing method which exploits frame-to-frame coherence in a real-time context. The load balancer achieves strong scalability without inducing communication overhead during rendering to coordinate the workers.

Categories and Subject Descriptors (according to ACM CCS): C.2.4 [Computer-Communication Networks]: Distributed Systems—Client/Server; I.3.2 [Computer Graphics]: Graphic Systems—Distributed/Network Graphics, Load Balancing; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction Techniques, Web-based/Browser Interaction; I.3.7 [Computer Graphics]: Three-dimensional Graphics and Realism—Ray-Tracing, Real-time

1. Introduction

Modern browsers continuously expand the functionality they provide, and thus establish themselves as a platform for a wide range of applications. The tendency is further reflected in the restriction and ultimately removal of plugin-based approaches in recent and upcoming browser versions. These plugins are a stability risk as they run with full privileges on the client system, may contain platform- and OS specific code, and therefore require a user dialog for installation. In contrast, an application within the browser's bounds enables cross-platform development, and user access via a open web page on any capable device.

One application area in the browser is interactive graphics. The widely adopted WebGL allows the development of GPU-accelerated 3D applications within a web page. However, WebGL is a low-level API. While higher level libraries like three.js exist, they are still separate from HTML5 and the Document Object Model (DOM), apart from the integration with the HTML5 canvas element for display. Therefore, graphics or library-specific programming knowledge is required to develop proprietary WebGL applications.

To make graphics content creation more accessible for the web developer, approaches for the declarative description of 3D scenes, tightly coupled with the web page, have been developed [BEJZ09] [SKR*10]. Especially XML3D has been designed as a HTML5 extension, and utilizes the DOM directly for scene hierarchy building and manipulation.

An aspect not addressed by above client-side WebGL libraries is server-based rendering. Not every device may have the capabilities required to store and interactively render a scene at the desired frame rate. Specific to the browser environment, persistent storage for large binary data is limited, and only a subset of the OpenGL features is available in WebGL. Implementations are further limited by the reduced features and performance of JavaScript (JS) compared to native code. Moving the rendering workload to a dedicated server back-end can overcome these restrictions.

DOI: 10.2312/pgv.20161182

In this paper, we present the extension of XML3D to support server-based rendering. We decided to use XML3D on the client-side due to its HTML5-embedded, generic approach for 3D content creation accessible for the common web developer. Further, XML3D is an established framework. This allows us to make the server back-end available to a range of already existing and upcoming applications.

The server back-end currently provides a rasterizer and a real-time ray-tracer, which supports additional features and material properties. Real-time ray-tracing has been a topic of research for over a decade [PMS*99]. Being an embarrassingly parallel problem, the key for high performance is a careful utilization of parallelism on modern processors.Still, enabling advanced diffuse effects like ambient occlusion or area lights in real-time at a high resolution is barely possible on a single commodity machine. Therefore, running a ray-tracer distributed on multiple machines is the consequence.

Our server back-end can operate in a distributed fashion, allowing an arbitrary hierarchy of servers in a standard or InfiniBand network. How well a ray-tracer scales on such an architecture is determined by the load balancing process. Ray-tracing workload can be highly heterogeneous. Some areas in an image may be more expensive to compute than others. To achieve optimal scaling, work must be balanced to keep all processing units busy until the frame concludes.

We present a load balancing approach that exploits frame-to-frame coherence in a real-time scenario. Based on cost measurements for the previous frame, we demonstrate that an accurate balance can be achieved for the next frame with negligible overhead. There is no communication between the master node, which accumulates the final rendering result, and the rendering nodes during a frame, and no communication between the rendering nodes at all. The approach is thus especially suitable when the connection between some nodes can be a bottleneck or is not possible.

The following section outlines related work for server-based rendering in the browser, real-time ray-tracing and load balancing. The paper next focuses on the client-side extension of XML3D to support server-based rendering. It then describes the server back-end and the load balancing method for distributed ray-tracing. The results section provides measurements and an analysis of the distributed ray-tracer.

2. Related Work

2.1. Web3D and server-based Rendering

There are two initiatives to embed declarative 3D content into a web page, and thus make it accessible for the web developer without requiring domain-specific or graphics programming knowledge. X3DOM [BEJZ09] [JRS*13] utilizes the XML-based X3D format to describe 3D content within a web page. In contrast, XML3D [SKR*10] [KSSS14] is an extension of HTML5. A XML3D scene is part of the DOM, and can be manipulated using the existing JS API developers are accustomed to.

Tamm et al. [TS15] describe the state-of-the-art methods for plugin free server-based rendering in the browser. One approach is to receive images via a WebSocket (WS) connection [WPJR11] [MPJ*13]. Motion JPEG (MJPG) over HTTP is another widely supported technique used by several systems [KPS10] [JBDW12]. Cloud gaming providers prominently use more bandwidth-efficient video streaming. To date, there are two options for plugin free real-time video streaming. In Chrome, a video receiver can be implemented with NativeClient (NaCl) [YSD*09], which allows to run native code within the browser's secured sandbox environment. Further, WebRTC [LR12] can be exploited to display a video stream of rendered data in the browser.

Behr et al. [BMP*15] describe a service infrastructure for visualization applications in the browser. The framework includes client-side rendering with hare3d [SLTB15] in addition to a server-side rendering component.

The above server-based rendering solutions are domain-specific and require proprietary libraries to operate from a web page. Further, there is no specific support for a distributed server back-end to facilitate high quality and performance rendering. In contrast, we enable server-based rendering in the declarative 3D library XML3D, which allows to specify generic 3D content in actual HTML5. Custom application logic can be built on top of XML3D with JS. Using this approach, we make the distributed rendering back-end available to a wide array of potential applications.

2.2. Real-time Ray-Tracing and Load Balancing

With the advances in parallel computing architectures, real-time ray-tracing is a topic of increasing interest. Today, such parallelism is available even on commodity multi-processor machines. We give a brief overview of real-time ray-tracing, and then focus on load balancing.

Parker et al. [PMS*99] describe an early interactive ray-tracer. Wald et al. [WS01] thoroughly outline the research area and its challenges, and present their own distributed ray-tracer. A generic, template-based interactive ray-tracing framework is presented by Georgiev et al. [GS08]. OptiX [PBD*10], a ray-tracing framework running on the GPU, enables a range of applications including real-time usage. More recently, a collection of optimized CPU kernels has been made available with Embree [WWB*14].

Load balancing is the process to distribute the potentially heterogeneous ray-tracing tasks to the processing units, with the goal to achieve maximum utilization. We distinguish between dynamic and static load balancing [CDR02].

2.2.1. Dynamic Load Balancing

A dynamic load balancer assigns initial tasks of potentially varying cost to the workers. If a worker becomes idle, it is as-

signed still outstanding tasks on demand. The first approach is to manage a central task queue. Workers request new tasks from the queue as they finish their current work [Pla02]. Ize et al. [IBH11] describe an out-of-core ray-tracing system which uses a queue both locally to schedule tasks on threads, and globally for the nodes in a cluster. The queue manager described by Wald et al. [WS01] attempts to assign previously rendered tasks to nodes, facilitating good cache locality assuming temporal coherence in interactive ray-tracing.

The second approach is work stealing [BL99]. Workers attempt to steal tasks from others instead of relying on a central queue. This effectively removes the queue manager as a possible communication bottleneck, as different node pairs can communicate in parallel. Tzeng et al. [TPO10] use work stealing to assign irregular workload to the GPU, giving ray-tracing as one application. DeMarle et al. [DGP04] initially assign previously rendered tasks to exploit temporal coherence in their distributed, shared-memory ray-tracing system. This is crucial to minimize fetching missing data from another node. After the initial assignment, work stealing is used.

Dynamic load balancers are generically applicable to parallel problems, and naturally scale well even with heterogeneous computing resources. However, they can suffer from communication overhead which increases with the number of workers. A low-latency connection between the master and the workers, or in case of work stealing between all workers, is mandatory.

2.2.2. Static Load Balancing

A static load balancer determines fixed tasks before rendering a frame, and thus avoids task management and communication overhead during rendering. Our distributed rendering back-end supports any hierarchy of nodes, and a dedicated network setup is not mandatory. We do not assume a fast or any link between the rendering nodes. Not even the master is necessarily directly connected to a rendering node. Therefore, we employ static load balancing. However, a static approach can not react to imbalance by shifting tasks to workers which become idle. Thus, determining a task distribution to accurately equalize the rendering cost on the workers decides about the effectiveness.

Heirich et al. [HA98] discuss several load balancing strategies for ray-tracing, including a randomized static assignment of pixels among workers. While such highly granular scattering can achieve an even cost distribution, it facilitates bad cache locality since each worker operates across the whole image [WPSB03]. Scattering pixels does further not fare well with a modern ray-tracer which traces packets of coherent rays.

More recent approaches attempt an estimation of the cost distribution. From the cost predicate, a tiling into tasks of equal cost can be derived. Moloney et al. [MWMS07] calculate a per-pixel cost estimate for their direct volume rendering system. Gillibrand et al. [GDC05] propose to time profiling rays at a lower resolution, and then applying the resulting cost map to the full resolution image. They tested the approach only with primary rays. Though, producing a representative cost map by profiling can cause major overhead, especially when considering secondary rays.

Similar to our approach, Cosenza et al. [CCDC*08] assume temporal coherence in a real-time ray-tracing system. Timings obtained for the previous frame are considered representative for the next one. However, along with Gillibrand et al. [GDC05], their approach suffers from inaccuracy as timings are obtained in a lower resolution than the one of the renderer. Each node only measures each task assigned to it. Cost differences within a task can cause the average to be largely off for contained rays or ray packets, which can lead to unbalanced scheduling decisions. Consequently, they use a task queue in addition to account for possible imbalance.

Cosenza et al. [CDE13] approximate the cost map for the next frame by rendering a rasterized preview on the GPU. The load balancer uses a summed area table based tiling algorithm to derive tasks of equal cost from the map. Though, there is a substantial loss in accuracy which prevents scaling similar to a dynamic approach. Therefore, they ultimately propose a work stealer which is optimized through sorting of the initial tasks by the approximated cost.

In our method, each node obtains a cost map for its task in the packet space of the renderer using high-resolution timings. The load balancer can thus achieve a strong accuracy while still only generating one static task per node and frame, effectively minimizing communication and tiling overhead. In contrast, Cosenza et al. [CCDC*08] [CDE13] end up with variants of existing dynamic approaches due to the limitation of their static attempts, nullifying the advantage of not requiring communication during rendering. They do not consider non power of two node counts and heterogeneous nodes. We extend the tiling of Cosenza et al. [CDE13] to support any amount of and heterogeneous workers.

3. System Architecture

Figure 1 introduces the system architecture on a high level. From the user's perspective, accessing an application is as simple as opening a standard web page. Embedded into the page is a XML3D scene, as well as the application logic on top of XML3D. If feasible and desirable, the client-side WebGL renderer can process the scene.

In addition, XML3D may offload rendering to a native server back-end. There are two independent connections between the client and the master node, and between each pair of connected nodes in the cluster. The first transfers the scene updates from the client to the master. Each node forwards the data to its child nodes. The second transfers the output from the resident renderer and the child nodes down the pipeline.

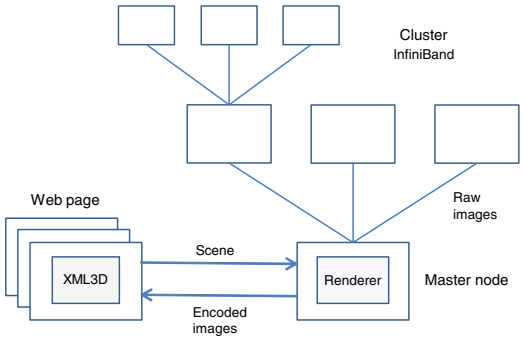

Figure 1: *Exemplary architecture of the distributed rendering system. XML3D clients connect to a rendering cluster with a hierarchy of nodes connected via InfiniBand.*

As there may be bandwidth restrictions between client and master, this includes an encoding step on the master. Within the cluster, the nodes send raw pixel data. Therefore, we support InfiniBand in addition to a standard network. The separate connections enable an asynchronous pipeline where the client already prepares and synchronizes the updates for the next frame, while the current one is still in progress of rendering or transfer to the client.

4. Client Side

A web page may contain a statically embedded XML3D scene. However, the web developer can build application logic on top of XML3D, manipulating the existing scene or adding new elements or the entire scene dynamically. With the server-based rendering extension, we want to keep and utilize this flexibility to create arbitrary applications. We therefore followed a minimal invasive approach, which exposes the server-side functionality with a manageable set of attributes. The attributes are an optional addition to existing XML3D elements. There are no new elements the developer needs to get accustomed to.

Consequently, existing applications can immediately be used, and new ones created as before. The approach enables a hybrid architecture where the client executes the application logic in parallel to the server-side rendering. The server does not need to adopt XML3D-specific features, which avoids maintaining redundant functionality and makes the server easily portable to another client.

The disadvantage is that the client still needs access to the scene as XML3D allows the manipulation of resources like buffers and textures. Also, there is overhead in synchronizing resources with the server. Though, the procedure is progressive, so rendering can already commence and provide the user with intermediate results quickly while part of the scene is still loading.

4.1. Connection Setup

To enable server-based rendering, the *xml3d* HTML5 element must contain a *server* attribute pointing to the address and port of a rendering server. Otherwise, the scene will be processed by the client-side WebGL renderer. The client first establishes a WS connection to synchronize the scene data. This synchronization channel is also used to send a handshake to the server.

The handshake tells the method to encode and transfer the image data. According to the selection, XML3D creates the display channel which establishes the connection for incoming images, and provides the HTML5 element to display the images in the web page. The client places the display element behind the transparent *canvas* element otherwise used for local rendering. The canvas is still required to capture user input, e.g. to select objects or move the camera. The architecture is modular and enables to integrate several display channel types (Section 4.3).

4.2. Synchronization

The data to be sent over the synchronization channel includes the resolution, camera and the lights. It also includes a collection of meshes. A mesh does not store data other than a transformation, but references buffers, textures, samplers and material properties. Meshes may share references, enabling the reuse of data (e.g. for geometry instancing). The separation offers a lot of flexibility to compile meshes. The client can compress buffers with the deflate algorithm.

XML3D loads resources asynchronously and progressively. An event notifying the initialization, change or deletion of a data entry can be generated anytime. Instead of synchronizing in-place with the event callback, the client schedules the update on the main run loop. This loop runs at a selectable rate to pass outstanding updates to the synchronization channel. If at least one update occurred during an iteration, the client sends a special message to the server requesting the rendering of a new frame.

The scheduling allows the client to postpone updates to prevent a WS send buffer overflow. Especially the initial loading of the scene may trigger heavy traffic for buffers and textures, making a rate control necessary. Also, updates may be triggered by the application logic at a higher rate than the one of the run loop. The scheduling prevents excessive rendering requests or redundant updates between requests.

4.3. Display

Independent of sending scene updates to the server, rendering results may arrive. We support several plugin free methods for transfer and display, which allows the application to choose the most appropriate one given the conditions and requirements. In a closed scenario, the focus may be on maximum en- and decoding performance. In a best-effort network, bandwidth-efficiency may be of more concern.

The server can transfer JPEG images over WS, and MJPG over HTTP. To trade compression-ratio for more en- and decoding speed, there is support for S3 texture compression (S3TC). S3TC enables fast parallel encoding. Using a commonly supported WebGL extension, the client can decode S3TC images directly on the GPU. For more bandwidth-efficiency, we further implemented a NaCl module to receive H.264 video. Tamm et al. [TS15] present measurements regarding latency and bandwidth for all methods.

4.4. DOM Integration

We expose server-based rendering to the developer with a set of attributes which can be added to the *xml3d* HTML5 element. With the exception of the *server* attribute, the attributes are optional.

- **server**: Address and port of a rendering server.
- **renderer**: The renderer to use, currently supporting the reference rasterizer and the real-time ray-tracer (defaults to the rasterizer).
- **display**: Method to transfer and display images, currently supporting JPG and S3TC via WS, MJPG via HTTP, and H.264 via NaCl (defaults to JPG).
- **nodes**: The maximum number of nodes to use for distributed rendering. It may be desirable to only use a subset or single node to increase the back-ends client capacity. A renderer may not require or benefit from several nodes.

Further, we extended XML3D with a set of new material properties to reflect the capability of the server-side ray-tracer. Refraction and reflection coefficients and the refraction index can now be specified for any material.

Figure 2 demonstrates the simple changes to port a scene to server-based rendering. By simply removing or renaming the *server* attribute, XML3D will fall back to the WebGL renderer which silently ignores unsupported features.

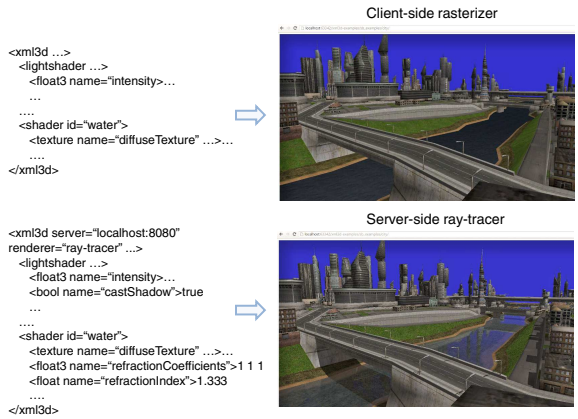

Figure 2: *The simplified original version of a scene (top) and the adapted declaration for the server-side ray-tracer.*

5. Server Side

Figure 3 illustrates the main components of the server back-end. Analogous to the client, the server manages a synchronization and a display channel. The display channel is responsible for sending the rendering output to the client, and thus interfaces with the local renderer. It applies scene updates from the synchronization channel directly, or caches them if the renderer is active. We call the last receiver in the pipeline the display client. The master encodes the final image to be sent to the display client.

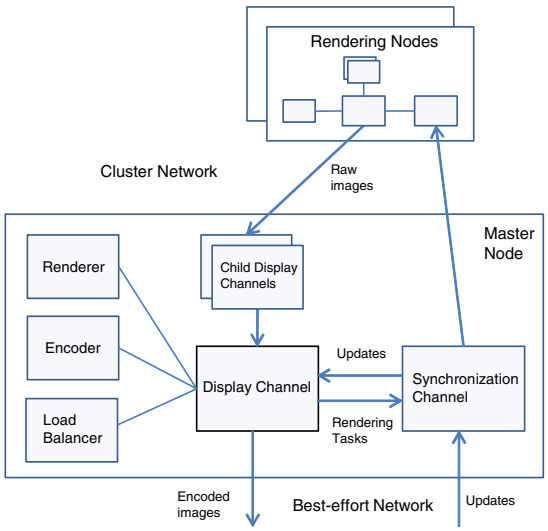

Figure 3: *The components running a distributed rendering session in the server back-end.*

For distributed rendering, synchronization and display channel also act as a client by establishing a connection to each of the participating child nodes. The corresponding handshake requests a dedicated display channel for raw pixel transport in the cluster network. The child nodes receive updates from, and send their rendering output to their parents. The master runs a renderer-specific load balancer to determine the rendering tasks for the next frame (Section 5.2). Every node has the same capabilities, and can assume the role of the master. This results in a flexible architecture, which allows to chain an arbitrary hierarchy of nodes.

The server facilitates asynchronous execution through its pipeline. The components in Figure 3 run in separate threads. While the local renderer executes, rendering results from the child nodes may arrive and be forwarded to the parent. Concurrently, scene updates for the next frame may arrive to be cached and forwarded to the child nodes. The server can immediately issue a new frame from cached updates, keeping the renderer occupied. The encoder can run in parallel to the rendering of the next frame. Therefore, we achieve strong parallel utilization of the computing and network resources.

5.1. Renderers

The server provides an abstract API which developers can implement to plug in their renderers. So far, we have integrated two renderers. The first is the reference rasterizer which mimics the functionality of the client-side WebGL renderer. The second is a custom CPU ray-tracer on top of the Embree [WWB*14] ray-tracing kernels. The ray-tracer is optimized for real-time performance. It operates on packets of rays for both tracing and shading, capable to utilize the SSE, AVX and AVX2 instruction sets. To run the renderer locally on multiple cores, we utilize the Cilk-Plus multi-threading language (originating from Blumofe et al. [BJK*95]) and its internal work stealer.

The ray-tracer supports ambient occlusion, which is a monte-carlo technique requiring a good amount of sample rays to avoid noisy results. The cost can snowball quickly with more samples, especially considering materials triggering secondary rays. The feature is barely adequate for a single commodity machine in real-time, thus motivating the scaling in a rendering cluster.

5.2. Distributed Rendering

Nodes participating in rendering, which may include the master, are called rendering nodes. For each type of renderer, a rendering node stores a coefficient indicating its performance relative to the other nodes in the cluster. In a cluster of homogenous machines, all nodes have the same coefficient. It is up to the operator to determine coefficients reflecting the heterogeneous nodes in the cluster, for example with benchmark tests. Section 6.3 describes our calculation for the ray-tracer.

When the display client connects, the master requests the coefficients of the renderers available in each child's sub-hierarchy. A child node will add its own renderer to the list, and further traverse the tree by requesting the renderer list from its children. Effectively, this process flattens the node hierarchy, and the master ends up with the complete list of coefficients. The master then selects the renderers to use in this session, prioritizing stronger nodes if only a subset of the available nodes should take part.

For the communication between two nodes, we support TCP over Ethernet and InfiniBand. TCP enables the deployment in a commodity setup, but may be limited in bandwidth. Since there is raw pixel data transport, we recommend the use of at least a 10 Gigabit network to minimize delay.

To determine the screen-space rendering task for each node in the upcoming frame, the master asks a renderer-specific load balancer. The server provides an abstract API for static load balancing, which a renderer implements to benefit from distributed execution. A renderer may generate custom cost data for the current frame, which the node transports to its parent along with the pixel data. The master gathers the data from all nodes, and passes it to the load balancer to generate the task distribution. The load balancer may also consider the renderer coefficients to weigh nodes.

Concluding, our architecture enables a flexible setup of possibly heterogeneous nodes with different roles. Nodes not suitable for rendering may still contribute as a master dedicated for encoding, or as a network hub giving access to a set of rendering nodes otherwise not reachable. Since each node can be the master, the definition of sub-clusters is possible, allowing different kinds of clients access at a certain level. A ray-tracing client may access the whole hierarchy, while it is enough to restrict rasterization to a small branch. Due to the static nature of the load balancing, rendering nodes must not be connected with low-latency to each other.

6. Load Balancing

This section describes the static load balancing for the distributed real-time ray-tracer. The foundation is the observation that in a real-time scenario, view changes between frames are likely small. Thus, timings for one frame are representative for the following frame. This maps to our server back-end and the ray-tracer, which are explicitly designed for real-time operation.

The ray-tracer operates in packets of neighboring pixels. During rendering, it measures the cost to determine the colors for each packet, effectively producing a cost map in packet space. The renderer transforms the cost map into a summed area table (SAT), which the display channel transfers to the master in addition to the pixel space image produced for the current task. The SAT allows to determine the cost of any rectangular region in constant time. Once the SATs from the rendering nodes have been accumulated, the load balancer uses them as input to determine a tiling into tasks of balanced rendering cost. Figure 4 outlines the steps to acquire the task distribution for the upcoming frame.

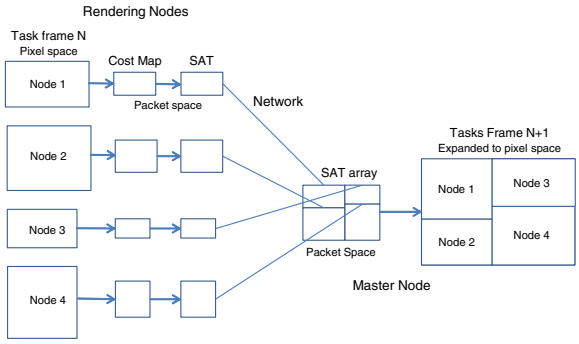

Figure 4: *The static load balancing for real-time ray-tracing. In this example, the ray-tracer measures the cost for packets of 2x2 pixels. Each node outputs a SAT to the master, which runs the tiling algorithm on the array of SATs to generate tasks of balanced cost for the next frame.*

Due to the high timing granularity in the packet space of the ray-tracer and the frame-to-frame coherence present in our real-time system, the load balancer can achieve a strong accuracy and thus scalability (Section 7). There is no communication during rendering, and no communication between the rendering nodes at all, making a basic deployment with any setup of machines and network possible.

The load balancer requires each node to generate a cost map during rendering, convert it to a SAT, and send the SAT to the master. The overall overhead is constant, and depends on the packet space resolution of the image. More nodes effectively reduce the impact as they process continuously smaller parts of the image in parallel. In contrast, the communication overhead of a dynamic load balancer increases with the number of nodes and tasks. More cores on a node reduce the cost map generation overhead as they acquire timings in parallel. As rendering cost increases, the overhead becomes increasingly negligible. In contrast, a dynamic approach may require a finer task granularity in response to a high rendering time of individual tasks to avoid stalling on a single worker and task in the end, in return increasing the communication overhead.

6.1. Cost Map

The mechanism to measure the cost for the pixel packets must be fine-grained and induce little overhead. We support two techniques reflecting the requirements. The processor time stamp counter (TSC) register stores the number of clock cycles since the last reset. To acquire values which are consistent across heterogeneous nodes, the ray-tracer divides by the maximum core frequency in Kilohertz, assuming all cores on a node run at this rate under the ray-tracing load.

In our tests, the TSC produced reliable results. Still, it may suffer from issues which can reduce the timing accuracy. The counters on different cores may not be tightly synchronized. While the OS may attempt the synchronization on booting, there is no guarantee. A thread switching the core between two measurements can thus result in distorted values. Also, processors with out-of-order execution support may shift the execution order of instructions, which can cause a slightly misplaced read of the counter. The processor switching its frequency can cause further inconsistencies.

To account for the potential issues with the TSC, we alternatively support the performance counter provided by the Windows OS. If possible, the performance counter relies on the TSC internally, thus also providing high precision and speed. It adds logic to handle the TSC issues, and can be considered as portable and reliable across recent systems.

6.2. Summed Area Table Generation

Hensley et al. [HSC*05] describe fast SAT generation on the GPU. However, since our cost map resides on the CPU, we implemented a multi-threaded CPU version which induces minimal overhead (Section 7). Part of the overhead is hidden by parallel execution as a rendering node sends its image data asynchronously to the SAT generation. Also, this enables the master to start the encoding of the accumulated image while the SAT array is still incomplete.

Cost map and SAT store 32-bit values. In a bandwidth setup of 10 GBit/s, this only accounts for around 0.737 ms of constant transfer overhead for a 720p image and packets of four pixels. The overhead is further mitigated through the distributed sending, the master participating in the rendering and possibly already ongoing encoding.

6.3. Tiling

The master gathers the SATs from the rendering nodes in the SAT array data structure. The array is the input to the tiling algorithm, which determines the tasks for the next frame. The tiling executes asynchronously to possibly still ongoing encoding, mitigating the already low overhead of the algorithm. The array behaves like a single SAT in the overall packet space resolution, providing the accumulated cost for a rectangular region from the origin to any packet. Several SATs may contribute to the cost, which Figure 5 illustrates. The load balancer first sorts all SATs by their offset on the x-axis, which then allows to quickly reject SATs, which start beyond the requested region, with a binary search.

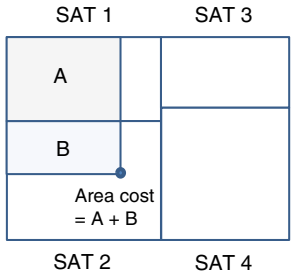

Figure 5: *Sampling two SATs in the SAT array to obtain the rendering cost for the pixel packets in an area.*

The load balancer starts with a packet space resolution tile with all the nodes attributed to it. Consulting the SAT array, the balancer uses a binary search to split the tile into two child sides with the cost balanced according to the nodes attributed to each side. It will recursively split the children, switching the axis on each level, until a leaf tile representing the task for a single node has been reached. For an even count of homogeneous nodes, balancing means finding the split which evens out the cost on both sides. However, the load balancer also considers an uneven node count attributed to a tile, and the presence of heterogeneous nodes. It weighs nodes according to their renderer coefficient (Section 5.2).

A timing represents the cost to render a packet of pixels

on a single core. But the performance may vary on heterogeneous nodes. We therefore statically assign a performance coefficient to each node, which indicates the performance increase for a single core relative to the node with the weakest cores. The load balancer normalizes the cost values retrieved from a SAT by multiplying with the performance coefficient of the originating node. The coefficient is an empirical factor which has to be chosen by the cluster operator. If all nodes share the same processor family, we set the values proportional to the node's core frequencies.

Each node locally distributes the renderer to the number of logical cores with a work stealing load balancer. The static balancer in the cluster thus assumes a linear scaling of the ray-tracing performance to the number of cores on each node. It therefore calculates a node's renderer coefficient as the product of the number of logical cores and the single-core performance coefficient.

When splitting a tile, the algorithm balances the normalized cost based on the ratio between the sum of renderer coefficients attributed to the first and the second child, thus accounting for any node count and heterogeneous nodes. For an uneven count, the balancer assigns the additional node to the side which brings the sums on both sides closest together. This facilitates producing child tiles of similar cost.

The balancer assigns tasks in a fixed order using a depth-first traversal of the tile tree, resulting in each node sticking to roughly the same image area. This facilitates good cache locality, which can improve the rendering performance.

7. Results

This section demonstrates the performance of the server back-end and the distributed real-time ray-tracer. The cluster consists of 20 rendering nodes. Each node is equipped with two Intel Xeon X5650 six-core processors running at 2.66 GHz. The processors do not support AVX instructions. Consequently, the ray-tracer falls back to SSE with packets of 2x2 pixels. We compared the performance of the ray-tracer in SSE and AVX mode on a modern machine, and measured an average performance increase of 88.8% with AVX.

The nodes are connected with 1 GBit/s Ethernet. Moreover, there is a 10 GBit/s InfiniBand link between ten of the nodes. The rendering nodes send RGBA output with 32 bits per pixel. The master uses the S3TC encoder. The image resolution is 1280x720.

We used the example scenes shown in Figure 6 to produce the results. The scenes are textured with diffuse and specular maps. The city has 65960, the tavern 1382164 and the hacienda 7691995 triangles. All scenes contain parts where there is heterogeneity in the rendering cost. The background is the cheapest area. In addition, the city contains a river causing secondary rays due to refraction. The tavern contains a wet reflective table. The most demanding scene is the hacienda with refraction for the glasses and the fountain, and a large amount of alpha mapped leafs. Each scene has a single light. There are 16 ambient occlusion rays per hit for the city and eight for the tavern and hacienda scene.

For reproducible results, the master automatically replays a recorded camera interaction loop for each scene. The view changes between frames are small as expected in a real-time scenario. The camera creates different viewing angles, effectively shifting the rendering cost distribution. The results build on the following per-frame measurements.

Rendering node

- **Kernel**: The total cost spent in the ray-tracing kernel across all threads to determine the colors for the pixel packets. This is equivalent to the bottom-right entry of the SAT generated from the cost map. The load balancer aims to equalize the kernel cost on the nodes. Therefore, this is the core measurement to show the scalability.
- **Rendering**: The time to determine the colors for the pixel packets in the multi-threaded setup. The kernel executes on the logical cores using a work stealing scheduler. This value includes the threading and timing overhead.
- **SAT Generation**: The time to generate the SAT for the rendering task from the cost map.

Master node

- **Tiling**: The time to determine the tasks for the next frame.
- **Pipeline**: The time spent in addition to the rendering to send the final image to the display client, which includes the encoding and in case of distributed execution, SAT generation, image and SAT transfer, and the tiling.

7.1. Scalability

Table 1 outlines the single node performance for each scene to set the benchmark. For the kernel and rendering measurements, the table showcases the strong scaling efficiency in the cluster. The values are the averages across all frames. Figure 7 illustrates the performance increase as nodes are added.

The kernel cost exhibits a super linear scalability. With more nodes, the load balancer assigns increasingly smaller tasks to the nodes in a fixed order. This can result in an increased cache locality, which we attribute the super linear effect to. Also, the foundation is the accurate rendering cost balance which the algorithm can derive from the SAT array generated for the previous frame. The scalability remains stable over time with occasional minor fluctuation and outliers as Figure 8 illustrates.

Along with the kernel, we observe a strong scalability of the rendering time. The rendering includes the threading and work stealing overhead, which stays about constant with more nodes. There is also the per-thread overhead to iterate over and time the assigned pixel packets. This overhead depends on the task size and does thus not necessarily decrease comparatively to the kernel cost with more nodes. Therefore,

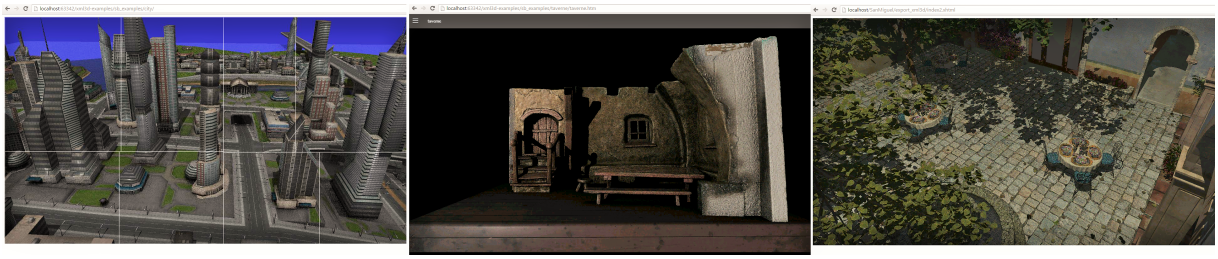

Figure 6: *The city (left), the tavern (middle) and the San Miguel hacienda scene. For the city, the tiling into rendering node tasks is visualized.*

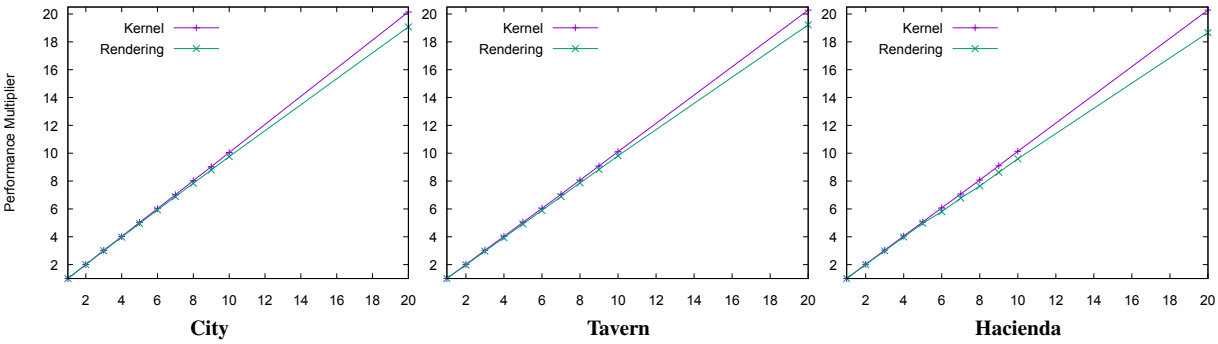

Figure 7: *The kernel and rendering performance increase as a function of the node count.*

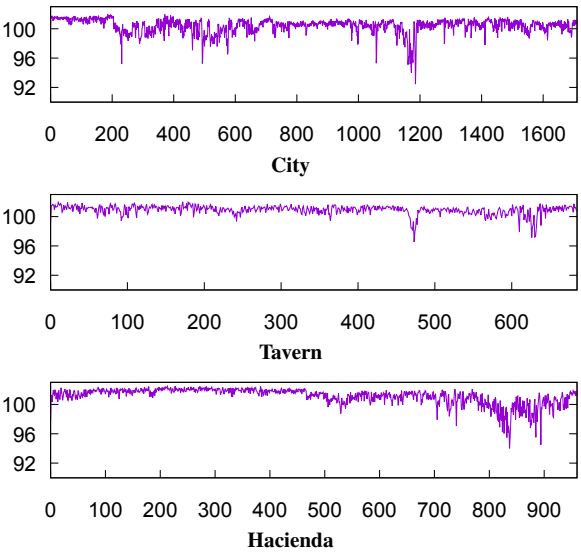

Figure 8: *The kernel scaling efficiency for eight nodes as a function of the frame number.*

the rendering time efficiency is naturally below the kernel efficiency. Further, as the per-node ray-tracing cost decreases with more nodes, the constant overhead has a higher impact, which can cause the efficiency gradually falling as nodes are added. We observed an almost identical single node rendering time with en- and disabled cost map generation. This shows the timing overhead is minimal, and fades into obscurity with increasing per-packet cost.

In addition, due to the execution on multiple cores, the rendering time is more susceptible to fluctuation and outliers caused by outside interference, like the OS occupying a core for a different task. Such occurrences can temporary reduce the scaling efficiency of the local work stealer. The more nodes, the more likely a disruption on any node occurs. Also, the efficiency of the work stealer can fluctuate by itself. Local imbalance negatively reflects on our top level load balancer in the cluster, which must assume a consistent scaling to the number of cores across the nodes. Therefore, we increased the process and rendering thread priority, which substantially reduced the appearance of outliers.

7.2. Pipeline Time

The tendency is the increase of the pipeline time with more nodes. Since the master participates in the rendering, the network load is reduced proportional to the task size assigned to the master. The transfer speed for 20 nodes drops greatly as we switched from InfiniBand to Ethernet.

We observe a higher pipeline time for the city than for the

Table 1: *The scaling efficiency and pipeline time for different node counts given the single node (SN) performance.*

City (**SN Rendering**: 384.1 ms, **Pipeline**: 0.884 ms)

	2	3	4	5	6
Kernel	100.8%	100.9%	100.5%	100.7%	100.4%
Rendering	100%	99.6%	99.1%	98.7%	98.6%
Pipeline	2.645	2.275	2.393	2.801	3.358
	7	8	9	10	20
Kernel	100.3%	100.4%	100.4%	100.6%	100.8%
Rendering	98.1%	97.7%	97.4%	97.3%	95.1%
Pipeline	3.714	4.022	4.29	4.495	36.98

Tavern (**SN Rendering**: 823.5 ms, **Pipeline**: 0.862 ms)

	2	3	4	5	6
Kernel	100.9%	101.3%	100.9%	101%	100.7%
Rendering	97.9%	98.2%	97.8%	97.9%	97.9%
Pipeline	1.376	1.979	1.615	2.033	2.046
	7	8	9	10	20
Kernel	100.8%	100.9%	101%	101.2%	101.5%
Rendering	98%	98%	98.1%	98%	95.9%
Pipeline	2.315	2.637	2.363	2.892	35.23

Hacienda (**SN Rendering**: 818.7 ms, **Pipeline**: 0.861 ms)

	2	3	4	5	6
Kernel	101.9%	102%	101.7%	101.5%	101.5%
Rendering	100.7%	99.1%	99.4%	99.2%	96.1%
Pipeline	2.4	2.259	2.098	2.307	2.187
	7	8	9	10	20
Kernel	101.2%	101.1%	101.3%	101.5%	101.5%
Rendering	96.3%	94.8%	95.3%	95.5%	92.9%
Pipeline	2.266	2.112	2.322	2.331	33.5

other scenes in Table 1. All scenes show a strong scaling efficiency of the renderer. However, the tavern's and hacienda's higher rendering cost causes the nodes to send their results with a higher absolute time offset to each other. Therefore, when the last node finishes its task, a larger part of the overall transfer already happened. This effectively relieves the network interface on the master, since there is less overlap of the incoming results. The hacienda shows a slightly reduced rendering scaling efficiency compared to the tavern from node count six onwards. Once more, the result is a higher offset between transfer operations, which avoids a pipeline time increase like in the other scenes.

The SAT generation overhead is low even on a single node with around 0.244 ms across all runs. Due to the distributed SAT array generation, we measured an additional average performance gain of up to 34.2% with 20 nodes. As the generation of each SAT is multi-threaded, the threading overhead naturally prevents a higher gain percentage-wise.

The tiling time increases with more nodes as the master must split more tiles to find a task for each node. Most crucially, the sampling of the SAT array becomes increasingly expensive. However, the cost stays low with around 0.236 ms across the 20 node runs. For much higher node counts, we plan to extend the pipeline so that the master can also accumulate cost maps to generate one overall SAT on its own. For test purposes, we extended the tiling with multi-threading, and ran it with 50000 tasks on one SAT. The average time is 0.67 ms. For high node counts, the heavily accelerated tiling outweighs the distributed SAT generation benefit.

7.3. Comparison

We observe a similar scaling efficiency compared to Cosenza et al. [CDE13]. They utilize a cluster-level work stealing scheduler, which makes a low-latency network between the rendering nodes mandatory. In contrast, we achieve the results with a static load balancer allowing a flexible network setup. Further, we repeated the run for the city and eight nodes, but this time only measured the overall cost of a task like Cosenza et al. [CCDC*08]. The scaling efficiency of the kernel drops substantially to 49.8%. To achieve competitive scalability, they incorporate a task queue to compensate the inaccuracy. In contrast, our method remains static by relying on the fine-grained timing mechanism.

We further performed a comparison with prevalent dynamic approaches on the thread-level. We repeated the runs on a single node with disabled ambient occlusion, and used three different load balancing methods to distribute the tasks among the threads: our static load balancer, a task queue, and work stealing. Table 2 shows the rendering performance of the static method in competition with the dynamic ones.

The static method performs almost on a par with the dynamic schedulers. Dynamic load balancers are ideally suitable locally due to the direct link between a moderate amount of cores. However, within a cluster, network communication and the coordination of many nodes can decrease the efficiency of these approaches. The task queue on the master can become a synchronization bottleneck if there are many simultaneous requests. A low-latency network is mandatory, and must be available between all nodes in case of work stealing. In contrast, our method can scale independent of the latency, and utilize nodes which are not connected. Only the tiling overhead increases with the number of nodes, but stays at a negligible level.

7.4. Reduced Frame-to-Frame Coherence

The load balancer relies on a strong coherence between consecutive frames in real-time rendering. To test the method under restricted conditions, we repeated the runs with the city scene, but used a new interaction loop with coarser view changes this time. The new loop contains only every fourth view of the original loop. As expected, the accuracy of

Table 2: *The rendering performance of the static load balancer on the thread-level relative to prevalent dynamic approaches: a task queue using OpenMP's dynamic scheduler, and work stealing using CilkPlus (Section 5.1).*

	City	Tavern	Hacienda
Task Queue	-3.6%	-3.4%	-1.4%
Work Stealing	-0.9%	-2.1%	-1.4%

the load balancer drops with the larger discrepancy between frames, which Figure 9 illustrates. Though, the scaling efficiency is still strong. While the load balancer will break if view changes become arbitrary, the results demonstrate the method is feasible in an interactive environment with continuous camera movement.

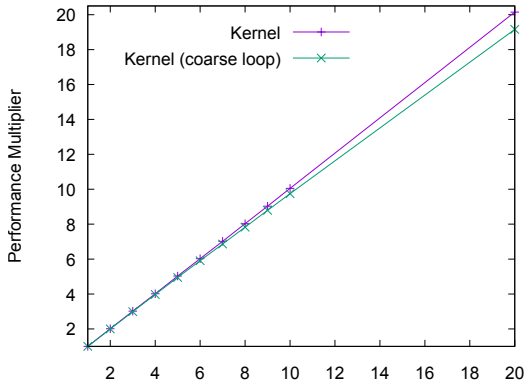

Figure 9: *The city scene kernel performance increase as a function of the node count for the original and the coarser interaction loop.*

7.5. Multiple Clients

To test the system under more pressure, we repeated the run with the city and eight nodes. This time, we connected three clients simultaneously. For each client, we measured a rendering time comparable to a single client using three nodes. Due to a minor offset between the connections, there is a short span in the beginning and end where not all renderers are active. This brings the performance closer to what we would expect from using nine nodes.

For each client, the load balancer still achieves a strong kernel scalability which only drops by around 2.6 percent compared to a single client. Due to the fine-grained timing mechanism in packet space, the OS unlikely switches to another thread during a measurement. Therefore, the measurements within a rendering session are mostly unaffected by the other clients, and remain stable. This allows the load balancer to operate each session accurately, which ultimately results in an equally smooth execution for all clients.

8. Conclusion and Future Work

The contribution in this paper is twofold. We presented the extension of the XML3D framework, which enables declarative 3D content in the web, with server-based rendering. The minimally invasive integration keeps the application logic untouched in the XML3D front-end, enabling arbitrary existing and upcoming applications to harness the back-end's power. The back-end is capable to run different renderers in a cluster hierarchy. We presented a static load balancing method to distribute a real-time ray-tracer in this architecture. The load balancer exploits temporal coherence between adjacent frames in the real-time scenario. Based on high-resolution timings gathered for the previous frame, it derives rendering tasks of balanced cost for the potentially heterogeneous nodes in the cluster. We demonstrated the strong scalability and low overhead the approach can achieve.

The combination of XML3D, which enables generic and portable graphics applications in the browser, and the dedicated server back-end, which gives these applications access to a selection of high-performance and possibly distributed renderers, makes our architecture accessible to both the common web developer and the expert user in a closed scenario.

The main limitation of the current architecture is the necessity to synchronize the scene data, which the client-side application logic may change at any time. We therefore plan to investigate the execution of the XML3D page in a headless, server-side browser environment. This would enable us to interface with the rendering back-end directly, and also remove potentially expensive XML3D features like data processing and animations from a less capable client.

Acknowledgments

This work was supported by the European Union funded "Dreamspace" project.

References

[BEJZ09] BEHR J., ESCHLER P., JUNG Y., ZÖLLNER M.: X3dom: A dom-based html5/x3d integration model. In *Proceedings of the 14th International Conference on 3D Web Technology* (New York, NY, USA, 2009), Web3D '09, ACM, pp. 127–135. doi:10.1145/1559764.1559784. 1, 2

[BJK*95] BLUMOFE R. D., JOERG C. F., KUSZMAUL B. C., LEISERSON C. E., RANDALL K. H., ZHOU Y.: Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 1995), PPOPP '95, ACM, pp. 207–216. doi:10.1145/209936.209958. 6

[BL99] BLUMOFE R. D., LEISERSON C. E.: Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (Sept. 1999), 720–748. doi:10.1145/324133.324234. 3

[BMP*15] BEHR J., MOUTON C., PARFOURU S., CHAMPEAU J., JEULIN C., THÖNER M., STEIN C., SCHMITT M., LIMPER M., DE SOUSA M., FRANKE T. A., VOSS G.: webvis/instant3dhub: Visual computing as a service infrastructure

to deliver adaptive, secure and scalable user centric data visualisation. In *Proceedings of the 20th International Conference on 3D Web Technology* (New York, NY, USA, 2015), Web3D '15, ACM, pp. 39–47. doi:10.1145/2775292.2775299. 2

[CCDC*08] COSENZA B., CORDASCO G., DE CHIARA R., ERRA U., SCARANO V.: Load balancing in mesh-like computations using prediction binary trees. In *Parallel and Distributed Computing, 2008. ISPDC '08. International Symposium on* (July 2008), pp. 139–146. doi:10.1109/ISPDC.2008.24. 3, 10

[CDE13] COSENZA B., DACHSBACHER C., ERRA U.: Gpu cost estimation for load balancing in parallel ray tracing. In *International Conference on Computer Graphics Theory and Applications (GRAPP)* (2013), pp. 139–151. URL: http://www.dps.uibk.ac.at/~cosenza/papers/CostMap. 3, 10

[CDR02] CHALMERS A., DAVIS T., REINHARD E. (Eds.): *Practical Parallel Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2002. 2

[DGP04] DEMARLE D. E., GRIBBLE C. P., PARKER S. G.: Memory-savvy distributed interactive ray tracing. In *Proceedings of the 5th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2004), EGPGV '04, Eurographics Association, pp. 93–100. doi:10.2312/EGPGV/EGPGV04/093-100. 3

[GDC05] GILLIBRAND R., DEBATTISTA K., CHALMERS A.: Cost Prediction Maps for Global Illumination. In *EG UK Theory and Practice of Computer Graphics* (2005), Lever L. M., McDerby M., (Eds.), The Eurographics Association. doi:10.2312/LocalChapterEvents/TPCG/TPCGUK05/097-104. 3

[GS08] GEORGIEV I., SLUSALLEK P.: Rtfact: Generic concepts for flexible and high performance ray tracing. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (Aug 2008), pp. 115–122. doi:10.1109/RT.2008.4634631. 2

[HA98] HEIRICH A., ARVO J.: A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing 12*, 1-2 (1998), 57–68. doi:10.1023/A:1007977326603. 3

[HSC*05] HENSLEY J., SCHEUERMANN T., COOMBE G., SINGH M., LASTRA A.: Fast summed-area table generation and its applications. *Computer Graphics Forum 24* (2005), 547–555. URL: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.90.8836. 7

[IBH11] IZE T., BROWNLEE C., HANSEN C. D.: Real-time ray tracer for visualizing massive models on a cluster. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2011), EGPGV '11, Eurographics Association, pp. 61–69. doi:10.2312/EGPGV/EGPGV11/061-069. 3

[JBDW12] JUNG Y., BEHR J., DREVENSEK T., WAGNER S.: Declarative 3d approaches for distributed web-based scientific visualization services. In *Dec3D* (2012), Behr J., Brutzman D. P., Herman I., Jankowski J., Sons K., (Eds.), vol. 869 of *CEUR Workshop Proceedings*, CEUR-WS.org. URL: http://dblp.uni-trier.de/db/conf/www/dec3d2012.html. 2

[JRS*13] JANKOWSKI J., RESSLER S., SONS K., JUNG Y., BEHR J., SLUSALLEK P.: Declarative integration of interactive 3d graphics into the world-wide web: Principles, current approaches, and research agenda. In *Proceedings of the 18th International Conference on 3D Web Technology* (New York, NY, USA, 2013), Web3D '13, ACM, pp. 39–45. doi:10.1145/2466533.2466547. 2

[KPS10] KASPAR M., PARSAD N. M., SILVERSTEIN J. C.: Cowebviz: interactive collaborative sharing of 3d stereoscopic visualization among browsers with no added software. In *Proceedings of the 1st ACM International Health Informatics Symposium* (New York, NY, USA, 2010), IHI '10, ACM, pp. 809–816. doi:10.1145/1882992.1883113. 2

[KSSS14] KLEIN F., SPIELDENNER T., SONS K., SLUSALLEK P.: Configurable instances of 3d models for declarative 3d in the web. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies* (New York, NY, USA, 2014), Web3D '14, ACM, pp. 71–79. doi:10.1145/2628588.2628594. 2

[LR12] LORETO S., ROMANO S. P.: Real-time communications in the web: Issues, achievements, and ongoing standardization efforts. *Internet Computing, IEEE 16*, 5 (2012), 68–73. doi:10.1109/MIC.2012.115. 2

[MPJ*13] MARION C., POUDEROUX J., JOMIER J., JOURDAIN S., HANWELL M., AYACHIT U.: A hybrid visualization system for molecular models. In *Proceedings of the 18th International Conference on 3D Web Technology* (New York, NY, USA, 2013), Web3D '13, ACM, pp. 117–120. doi:10.1145/2466533.2466558. 2

[MWMS07] MOLONEY B., WEISKOPF D., MÖLLER T., STRENGERT M.: Scalable sort-first parallel direct volume rendering with dynamic load balancing. In *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2007), EGPGV '07, Eurographics Association, pp. 45–52. doi:10.2312/EGPGV/EGPGV07/045-052. 3

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: Optix: A general purpose ray tracing engine. In *ACM SIGGRAPH 2010 Papers* (New York, NY, USA, 2010), SIGGRAPH '10, ACM, pp. 66:1–66:13. doi:10.1145/1833349.1778803. 2

[Pla02] PLACHETKA T.: Perfect load balancing for demand-driven parallel ray tracing. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing* (London, UK, UK, 2002), Euro-Par '02, Springer-Verlag, pp. 410–419. URL: http://dl.acm.org/citation.cfm?id=646667.700319. 3

[PMS*99] PARKER S., MARTIN W., SLOAN P.-P. J., SHIRLEY P., SMITS B., HANSEN C.: Interactive ray tracing. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics* (New York, NY, USA, 1999), I3D '99, ACM, pp. 119–126. doi:10.1145/300523.300537. 2

[SKR*10] SONS K., KLEIN F., RUBINSTEIN D., BYELOZYOROV S., SLUSALLEK P.: Xml3d: Interactive 3d graphics for the web. In *Proceedings of the 15th International Conference on Web 3D Technology* (New York, NY, USA, 2010), Web3D '10, ACM, pp. 175–184. doi:10.1145/1836049.1836076. 1, 2

[SLTB15] STEIN C., LIMPER M., THÖNER M., BEHR J.: hare3d - rendering large models in the browser. *WebGL Insights* (2015), 317–332. doi:10.1201/b18564-27. 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* (Aire-la-Ville, Switzerland, Switzerland, 2010), HPG '10, Eurographics Association, pp. 29–37. URL: http://dl.acm.org/citation.cfm?id=1921479.1921485. 3

[TS15] TAMM G., SLUSALLEK P.: Plugin free remote visualization in the browser. In *Proc. SPIE, Visualization and Data Analysis* (2015), vol. 9397. doi:10.1117/12.2077761. 2, 5

[WPJR11] WESSELS A., PURVIS M., JACKSON J., RAHMAN S. S.: Remote data visualization through websockets. In *Proceedings of the 2011 Eighth International Conference on Information Technology: New Generations* (Washington, DC, USA, 2011), ITNG '11, IEEE Computer Society, pp. 1050–1051. doi:10.1109/ITNG.2011.182. 2

[WPSB03] WALD I., PURCELL T. J., SCHMITTLER J., BENTHIN C.: Realtime ray tracing and its use for interactive global illumination. In *In Eurographics State of the Art Reports* (2003). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.500.8528. 3

[WS01] WALD I., SLUSALLEK P.: State of the art in interactive ray tracing. In *Eurographics* (2001), pp. 21–42. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.6266. 2, 3

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph. 33*, 4 (July 2014), 143:1–143:8. doi:10.1145/2601097.2601199. 2, 6

[YSD*09] YEE B., SEHR D., DARDYK G., CHEN J., MUTH R., ORMANDY T., OKASAKA S., NARULA N., FULLAGAR N.: Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), pp. 79–93. doi:10.1109/SP.2009.25. 2