Vis Comput
DOI 10.1007/s00371-013-0836-y

ORIGINAL ARTICLE

Extending a distributed virtual reality system with exchangeable rendering back-ends

Techniques, applications, experiences

Karsten Schwenk · Gerrit Voß · Johannes Behr · Yvonne Jung · Max Limper · Pasquale Herzig · Arjan Kuijper

Abstract We present an approach to integrate multiple rendering back-ends under a common application layer for distributed systems. The primary goal was to find a practical and nonintrusive way to use potentially very different renderers in heterogeneous computing environments without impairing their strengths and without burdening the back-ends or the application with details of the cluster environment. Our approach is based on a mediator layer that handles multithreading, clustering, and the synchronization between the application's and the back-end's scene. We analyze the proposed approach with an implementation for a state-of-the-art distributed VR/AR system. In particular, we present two case studies and an example application.

K. Schwenk (✉) · J. Behr · Y. Jung · M. Limper · P. Herzig · A. Kuijper
Fraunhofer IGD, Darmstadt, Germany
e-mail: karsten.schwenk@igd.fraunhofer.de

J. Behr
e-mail: johannes.behr@igd.fraunhofer.de

Y. Jung
e-mail: yvonne.jung@igd.fraunhofer.de

M. Limper
e-mail: max.limper@igd.fraunhofer.de

P. Herzig
e-mail: pasquale.herzig@igd.fraunhofer.de

A. Kuijper
e-mail: arjan.kuijper@igd.fraunhofer.de

K. Schwenk · M. Limper · A. Kuijper
TU Darmstadt, Darmstadt, Germany

G. Voß
Fraunhofer IDM@NTU, NTU, Singapore, Singapore
e-mail: vossg@camtech.ntu.edu.sg

Published online: 03 May 2013

Keywords Distributed/network graphics · Virtual reality

1 Introduction

Frameworks for distributed virtual and augmented reality applications typically have to support a wide range of use cases. Examples include visualization of large CAD models, game-like virtual environments, and photorealistic rendering. In addition, they have to support a variety of target platforms, ranging from CAVEs to mobile devices (Fig. 1). For optimal performance, such systems need highly specialized, potentially quite different, rendering back-ends. At the same time, it makes sense to use the same scene description and application layer for all scenarios, mainly to ease application development with existing tool chains.

In this paper, we describe a pragmatic and practice-proven approach to using different rendering back-ends with a common application layer in distributed systems. Key to the approach is a mediator layer between application layer and rendering back-ends. The mediators are based on OpenSG [13], an open source scene graph library with sophisticated support for clustering and multithreading. In contrast to related approaches, we focus on the needs of distributed virtual reality frameworks. In particular, we address the following two issues.

The first issue is that the back-ends can be based on extremely different architectures and paradigms. Furthermore, they often come packaged as closed source, third party libraries that define their own API and scene description. This makes it difficult to abstract these rendering back-ends with a single imperative interface and just implement this interface for each back-end.

The second issue is that we support stereoscopic rendering and rendering in a cluster (e.g., for multiprojector environments). We want to use these features with different

Fig. 1 Some use cases. *Top left*: application running on touch table and large tiled display wall. *Bottom left*: streaming from desktop to web page, viewed in browser on a tablet. *Right*: stereoscopic rendering on tiled display wall

rendering back-ends, even if the back-ends themselves do not support them. It seems reasonable to implement these features in a layer on top of the rendering back-ends. On the other hand, the application layer should be mostly agnostic to which hardware-configuration the application runs on.

We exemplarily describe our approach with the InstantReality framework [8], which we have extended with various specialized rendering back-ends. However, the ideas and design principles we describe are applicable to similar systems.

2 Related work

Many approaches to using different rendering back-ends with a common application layer have been described. Also, a lot of literature exists on rendering in clusters. But surprisingly few publications deal with the combined problem of how to efficiently support different rendering back-ends for systems that are distributed in heterogeneous computer networks.

For an overview of cluster-based rendering, we refer to the recent survey of Staadt et al. [16]. In addition to these rendering-centric approaches, there are systems whose primary concern is to distribute arbitrary computing tasks in a cluster. They contain the problem we address in this paper as a subproblem, but the top-level architecture usually just defers it to black-box rendering nodes. PaTraCo [9] and FlowVR [1] are recent examples. The PAC-C3D architectural model for collaborative virtual environments [7] works at a slightly finer granularity and uses *Representations* that have to be implemented for each rendering back-end.

Approaches for exchangeable rendering back-ends range from low-level abstractions of imperative graphics APIs to rather high-level abstractions that work with a common understanding of a "scene."

OpenGL [20] is probably the most popular imperative graphics API. However, the API strongly reflects the underlying rasterization pipeline, and to our knowledge it is not used beyond the rasterization world.

Some ray tracing frameworks take the scene abstraction to the extreme and rely on scenes that basically just have to provide an "intersect" method. For example Pharr et al. [12] do this with PBRT. While this works well for ray tracing, other back-ends may require scenes that expose more internal structure in order to process them efficiently.

Many scene graphs occupy a middle ground as far as scene abstraction is concerned and also allow custom renderers as plug-ins (e.g., [6, 13]). The typical mechanisms are custom traversals and extensible nodes (via callbacks), usually a combination of both. Döllner and Hinrichs' Virtual Rendering System [6], used for example by Steinicke et al. [17], stands out because it was specifically designed to work well with different back-ends and regards adapter components as part of the core architecture. (Döllner and Hinrichs [6] also contains a survey of previous approaches, which we will not reiterate here.)

A recent approach that is very closely related to our work is Rubinstein et al. [14] and their Real-time Scene Graph (RTSG). It allows the application to attach different rendering back-ends and provides an efficient way of propagating changes via callbacks. Another closely related approach is the Scene Graph Adapter by Berthelot et al. [4], an architecture for mixing different scene graphs in one application at runtime, without an offline conversion step. The approach consists of two standardized wrapper interfaces (Format and Renderer) that have to be implemented for each 3D format and renderer. A central Scene Graph Adapter instance then mediates between several wrapper instances of these wrappers by mapping nodes and calls.

The most important aspect that sets our work apart from related approaches is our "fat" mediator layer based on OpenSG [13]. In fact, OpenSG can be regarded as the front-end toward the application layer, providing a common interface for the mediator and the back-end.

Compared to approaches based on imperative APIs [20] our approach provides a higher abstraction of the rendering back-end behind a declarative interface (a stripped-down scene graph). The loose coupling allows more generic back-ends; the scene graph adaption allows each back-end to work with a suitable scene representation for high performance.

Approaches that treat the scene as a black box (or at least a very opaque box) [12] have the complementary problem: They cannot query enough information to efficiently map a scene. In contrast, our scene-graph-based approach allows an efficient mapping of the scene.

Approaches based on scene graphs that use custom callbacks during traversal [6, 13] are usually quite close to a

sweet spot as far as scene abstraction is concerned. However, purely relying on this approach leaves the concerns of multithreading and clustering to the application layer or the rendering back-end. Another issue not sufficiently addressed by the traversal approach is that the adapter has no efficient way of detecting and propagating changes (without a full traversal). RTSG [14] solves this problem via callbacks, we use a mechanism based on OpenSG's ChangeLists. We also provide a cleaner separation of application layer, mediator layer, and back-end. The Scene Graph Adapter [4] leaves the multithreading and clustering issues unattended and focuses solely on scene mapping. We use an adapter that is similar in spirit in our mediators, but we base it on a low-level OpenSG scene. This results in a cleaner interface and takes the burden of multithreading and clustering off the shoulders of application and back-end. The downside is that we have to pay the memory overhead of the OpenSG scene as an intermediate representation.

3 Our approach

In this section, we first lay down the most important requirements that guided our design, then we give an overview of the architecture itself, and finally we briefly discuss two particularly interesting aspects of our approach in more depth: how we handle incremental changes and clustering support. Since the focus of this paper is on a practice-oriented discussion, we kept the description of the architecture relatively short and put more emphasis on the case studies in Sect. 4. Details of the system architecture appear elsewhere [15]. Furthermore, our approach is based on OpenSG and we assume a basic familiarity with concepts like Fields, FieldContainers, Aspects, and ChangeLists [13, 18].

3.1 Requirements

From our experiences with past projects and the objectives of current projects, we have derived the following requirements for the extended system.

Extensibility and generality. The system should be able to integrate new rendering back-ends relatively painlessly. It should also be general enough to handle back-ends coming from very different application areas and following different design paradigms.

Nonintrusiveness. Neither the application layer nor the rendering back-end should need any changes or extensions in order to work together. (Back-ends may extend the application layer to expose specialized functionality, as described below, but basic functionality should be possible without touching both.) This is important because we want to support commercial libraries as back-ends that usually come with an unalterable interface.

Clustering and stereo. The system should provide (at least basic) support for rendering in a computer-cluster (tile-based and cooperative) and stereoscopic rendering, even if a back-end itself does not support it.

Rendering performance. Of course, the system should allow each renderer to play out its strengths—after all, that is why we want multiple, specialized rendering back-ends. Integrating a back-end into the system should hamper its rendering performance as little as possible.

Efficient incremental updates. Not only the raw rendering performance is important, but also how updates are propagated from the application layer to the rendering back-end (and sometimes the other way around). The system should provide an efficient (in terms of runtime and usability) solution to this problem.

Ability to extend application layer. While the system should rely as much as possible on a common low-level abstraction of a scene, sometimes it is practical to expose attributes that are specific to a certain back-end in the application layer. An example are extended material attributes for a ray tracer. The system should provide a mechanism to pass on such data.

Mixed (hybrid) rendering. The system should provide (at least basic) support for mixing different renderers during the generation of one image. For example, it should be possible to render large static geometry with a back-end optimized for that purpose and to render dynamic 3D GUI elements in the same scene with another back-end.

3.2 Basic design

Figure 2 shows a schematic overview of our architecture. The application layer (in our concrete case an X3D-browser [2]) manages the high-level application logic and

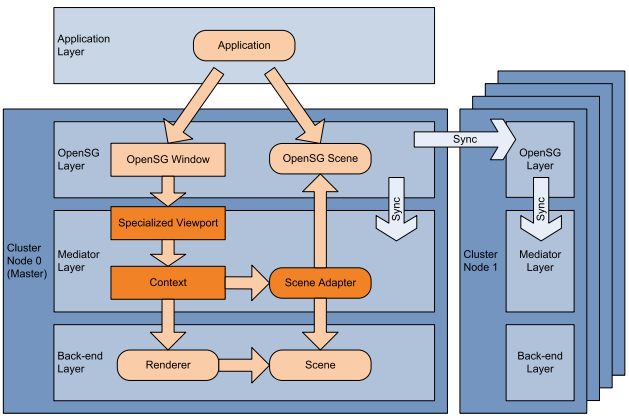


Fig. 2 Schematic overview of our architecture

mirrors the state of the X3D scene in a low-level OpenSG scene graph. The OpenSG layer (and all layers below) are only concerned with the current state of components (transforms, materials, geometries, etc.) and not with procedures that change this state (animation, physics, I/O, etc.).

The mediator layer is the main subject of this paper. It has to be implemented for each rendering back-end (although parts can be reused as shown in Sect. 4.1). It usually consists of a specialized OpenSG-Viewport, a scene adapter, and a context. With the Viewport the mediator can hook itself into OpenSG's rendering infrastructure. The scene adapter translates the OpenSG scene into the renderers internal representation and keeps it up-to-date. The context manages instances of the back-end and allows multiple Viewports to share these instances.

Note that there is no direct dependency from the application/OpenSG layer into the mediator layer (only via the default OpenSG-Viewport interface) and no dependency of the mediator into the application layer (only into the OpenSG layer, a mediator potentially works with all OpenSG applications). Also, the mediator depends on the back-end, but not the other way around. This takes care of our requirements of *extensibility and generality* and *nonintrusiveness*. Since the back-end is fed with its own scene representation and simply asked to fill a Viewport, it usually can maintain a near-optimal *rendering performance*. Relying on Viewports also enables a simple (but usually sufficient) way to do *mixed (hybrid) rendering*. Viewports can be layered on top of each other in order to combine the images of different back-ends using z-buffering and alpha-blending. The *ability to extend the application layer* is granted by OpenSG's attachment mechanism [13], which allows attaching arbitrary data to nodes. The application layer only needs to pack data for extensions into attachments, which are then interpreted by mediators that understand the extension and ignored by others. How the system meets the requirements *efficient incremental updates* and *clustering and stereo* is particularly interesting and described in Sects. 3.2.3 and 3.2.4.

3.2.1 Viewport

Each mediator exposes a specialized OpenSG-Viewport which internally maps to the underlying renderer. So, every time OpenSG (on behalf of the application) wants a Viewport to be rendered, the back-end is invoked. The target is usually an OpenGL back buffer, but it can also be another render target. For example, one can implement a Viewport that renders an image to disk or streams a video to a website.

The back-ends are invoked exclusively through this specialized Viewport class. If the application wants to use a certain back-end, it just creates the corresponding Viewport and attaches it to a OpenSG-Window. The Viewport then creates the infrastructure necessary to convert the scene and instantiates the underlying renderer.

3.2.2 Scene conversion

The OpenSG scene in Fig. 2 is a stripped-down scene graph with only a few node types. We intentionally did not strictly define which nodes a mediator has to understand, unknown nodes can simply be ignored by the scene adapter. Node types that are supported by all our renderers are Transform, Geometry, two basic Materials, Lights, Camera, and Background. This seems to be the minimal set a renderer needs to produce meaningful pictures.

The scene adapter is responsible for mapping the OpenSG scene to a representation the back-end can use. This is usually where the bulk of work has to be done when implementing a new mediator layer. In some cases, the renderer may be able to use the OpenSG scene directly, or at least parts of it (e.g., an OpenGL-based deferred renderer), but usually a conversion of the scene will be necessary. In this case, the adapter will usually traverse the whole OpenSG scene graph once during initialization and build a shadow scene by converting objects such as geometries, materials, and lights into suitable representations. Note that this does not have to be (and usually is not) a one-to-one mapping, the case studies in Sect. 4 contain examples. However, the handling of incremental updates (described in the following section) requires to quickly identify representatives that need updating as a result of a change. Therefore, usually several maps are build during the initial conversion, which serve as a scene dictionary.

3.2.3 Handling changes

OpenSG automatically keeps track of all changes that are made to the attributes of an object (the Fields of a FieldContainer) in so-called ChangeLists [18]. A ChangleList's primary purpose is to allow synchronization between threads and over the network in OpenSG's multi-buffered threading model. A side effect of this mechanism is that at each render-call on the Viewport, we have a ChangeList available that contains all changes made to the scene in this frame. Syncing the OpenSG scene with the back-end scene is now almost trivial: In each call to render, the Viewport invokes the scene adapter's sync method, which parses through the current ChangeList and updates the representatives for all relevant changes. ChangeLists also contain entries for newly created and deleted objects, so these events can be handled as well.

This approach has several advantages over registered listeners. First of all, we process a changed field only once. If, for example, a transform is changed three times in one frame, only one update propagates to the back-end, namely the last. Second, the mechanism works well in multithreaded and clustered environments in conjunction with OpenSG's Aspect concept. It is possible for the back-end to access

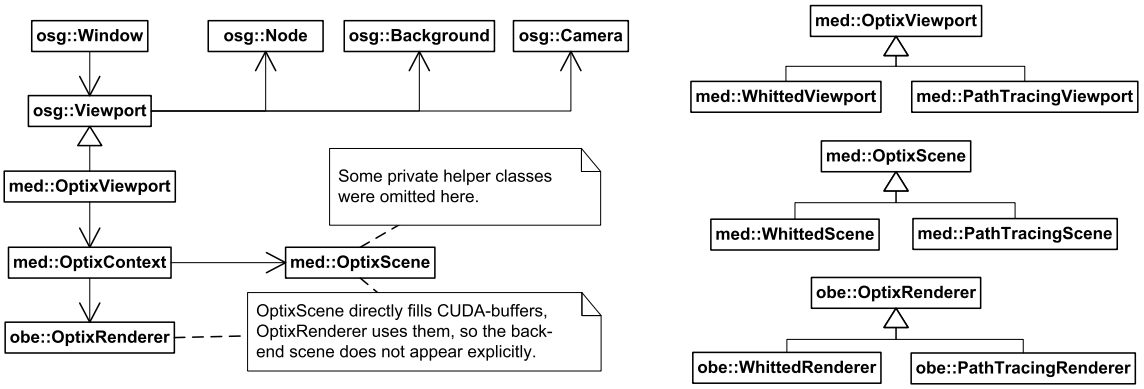


Fig. 3 Static structure of the Optix mediator. Namespace-tags refer to the layers in Fig. 2: osg = OpenSG layer, med = mediator layer, obe = Optix back-end. Some helper classes and the coverage renderer (Sect. 4.3) were omitted for a more compact representation

a consistent OpenSG-state for the current scene, while the application already updates the OpenSG scene for the next frame. Similarly, it is absolutely irrelevant for the mediator whether the ChangeList came from the same machine or over the network. This way the mediator is completely agnostic toward the clustering setup it is running in.

3.2.4 Multithreading, clustering, and stereo

Multithreading and clustering have been mentioned before, and our mediator design allows us again to use large parts of what OpenSG has to offer here. The details and inner workings of OpenSG's multithreading and clustering concept are described by Voß et al. [18], we will only describe how our approach integrates into this framework.

In our system, the application layer exists only once on a single host, but the OpenSG scene may be mirrored on multiple machines (Fig. 2). The copies of the scene are kept in sync by ChangeLists sent over the network. As already mentioned, the fact that the mediators only work on the OpenSG scene and ChangeLists allows them to function properly in clustered or multithreaded setups without knowing anything about the application layer or the cluster environment. Rendering in a cluster is managed by the ClusterWindow class, which can be envisioned as a virtual window that exists on a remote host; or, in the case of tiled rendering and load balancing, multiple hosts. Since a mediator only interacts with OpenSG via instances of its specialized Viewport class, one just has to add the specialized Viewports to a ClusterWindow like ordinary OpenSG Viewports to use them in any OpenSG-compatible cluster setup.

Using a Viewport as the primary interface of the mediator allows us to elegantly implement stereo rendering. This is done by simply using two Viewports, one for each eye (layered on top of each other, side-by-side, or even on different machines). In order to prevent wasting resources in such a setup, the Viewports usually share the underlying converted scenes and other resources via reference counted contexts. Since Viewports can be deactivated and activated on-the-fly, layered Viewports can also be used to quickly switch from one back-end to another. For example, one can use rasterization for navigation and then seamlessly switch to ray tracing once a interesting viewpoint has been reached. Also, the possibility to selectively use different back-ends on different machines opens up interesting new possibilities. For example, one could use a fast, low-quality renderer an a touchtable or mobile device to navigate through a scene, while a powerful cluster renders the same scene on a large tiled display wall using load balancing with a progressive, photorealistic algorithm like path tracing.

4 Case studies

In this section, we discuss two case studies: an Optix-based ray tracing back-end and a visibility guided renderer for very large data sets. We also describe a concrete example application that uses these two mediators.

4.1 Optix back-end

Optix [11] is a ray tracing engine based on CUDA [10]. One interesting aspect of our Optix mediator is that the same mediator supports three renderers: a simple Whitted-style real-time ray tracer, an interactive progressive path tracer, and a special coverage renderer (described in Sect. 4.3). Figure 3 shows the static structure of our Optix mediator. The renderers differ mainly in some CUDA programs (e.g., camera and material programs), while most other parts (e.g., intersection programs, most of the scene adapters) are identical. Therefore, we implemented the key components of the mediator layer, OptixScene, and OptixViewport as "fat" base classes that contain most of the functionality and specialized them where needed.

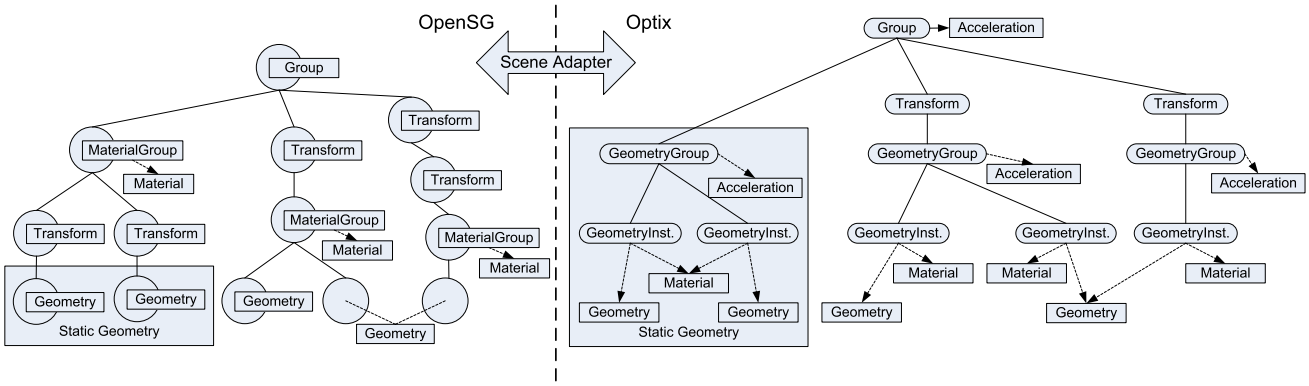


Fig. 4 The scene adapter maps the OpenSG scene to an Optix scene. In order to optimize for ray tracing, this is not a one-to-one mapping: GeometryGroups that provide Acceleration structures are introduced, Tranforms are collapsed, static geometry is placed under one common Acceleration. The figure also illustrates sharing of Materials and Geometries (instancing)

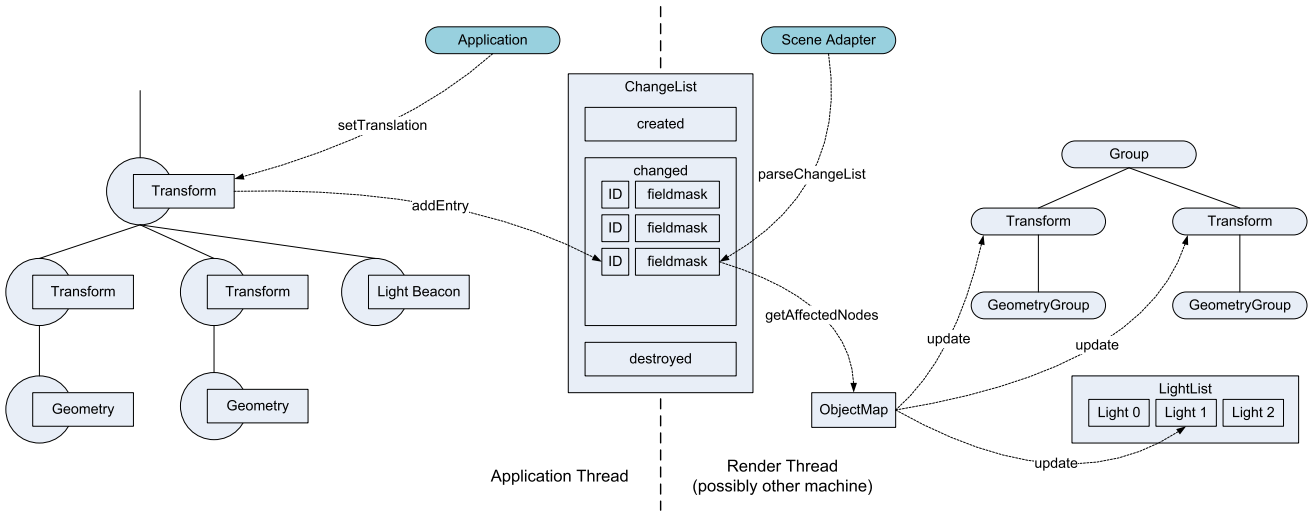


Fig. 5 Snooping on ChangeLists is our way to propagate incremental changes. When the application changes a Field, a corresponding entry is created (or updated) in the current ChangeList. When the mediator syncs the back-end (usually before rendering), it parses through the ChangeList and carries out the necessary updates on the back-end scene. ObjectMap maps the FieldContainer IDs in the ChangeList to objects in the back-end scene. Because of OpenSG's Aspect mechanism, this sync is thread-safe and works between cluster nodes. In this particular example (Optix back-end), lights are not part of the back-end scene graph and managed in a separate LightList

The most interesting component of most mediators is the scene adapter; this is also the case here (OptixScene). One important point is that we do not really mirror the OpenSG scene as an Optix scene. The OpenSG scene graph is usually quite deep (as it mirrors the X3D/VRML-graph of the application layer), and we collapse it into a flat graph that has at most one transform above each geometry instance (Fig. 4). A mapping from the original transforms to the collapsed ones is established, so we can quickly find the representatives that need updating for a given ChangeList entry (Fig. 5). We assume geometry instances below the same transform node do not move independently and combine them into the same acceleration structure. Geometry that has been explicitly flagged as static is combined into one large static geometry chunk. Geometry can also be flagged as animated, in which case a special acceleration structure is used that can be quickly updated, but has slightly lower rendering performance. These optimizations are necessary to retain good ray tracing performance while still allowing fast updates for frequently occurring changes (changing transforms, lights, animated meshes, etc.). Changes on the graph structure become more expensive with this mapping, but we assume these occur relatively seldom. As the build times for the acceleration structures can be quite high, they can be cached on disk, so one does not have to pay the costs every time a scene is loaded. Geometry data is written directly into CUDA-buffers, as are image textures, in order to prevent unnecessary duplication of large data.

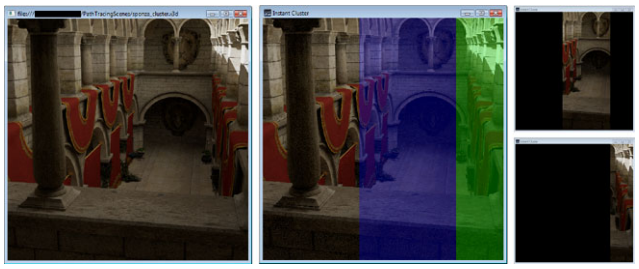


Fig. 6 Cooperative path tracing with dynamic load balancing (local host with two supporting machines). The *colored overlay* in the *middle image* shows the parts offloaded to the two supporting machines; the *two images* to the *right* show their viewports

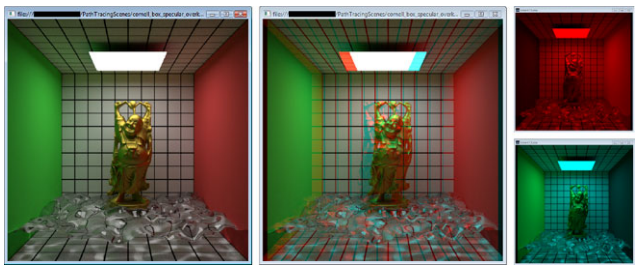


Fig. 7 Path tracing in a stereo setup (anaglyph with one client and two servers, one for each eye)

Area lights are another interesting aspect of this mediator. They are a simple example of how a mediator can freely interpret the scene in order to play out the strengths of its back-end without exposing extensions in the application layer. Instead of exposing a specialized node in the application layer, area lights are simply geometries with a non-zero emission component in their material. The mediator detects these geometries and converts them into area lights for the ray tracers.

Figures 6 and 7 show the Optix back-end in cluster setups (cooperative and stereo). It is important to note that the back-end is not aware of the fact that it is working in a cluster or stereo setup, it just fills its viewport.

4.2 VGR back-end

We have successfully integrated 3D Interactive's Interviews3D platform [5] into our system, a visibility guided renderer (VGR) for large datasets. Besides the default renderer, the VGR mediator also supports a coverage renderer (Sect. 4.3).

In contrast to Optix, VGR is a very closed package that gives you much less freedom. Optix is a relatively low-level API (often advertised as OpenGL for ray tracing) and constitutes a framework on top of which you have to implement your rendering algorithms. VGR, on the other hand, has a closed renderer and a very strictly defined scene authoring interface centered around a scene database. This allows the library to play out its strengths in regard to out-of-core rendering of huge data. The scene adapter for this mediator converts the OpenSG scene into such a database. Here, we had to compromise and provide two options. The first option converts the OpenSG scene and keeps it intact and in memory. This is the default. This option is best for interactive use, since the application layer is kept intact and everything that relies on the OpenSG scene still works (navigation, picking, animation). The downside is that it hampers VGR's out-of-core abilities, because the whole OpenSG scene is kept in memory. Therefore, we also provide the option to use a (possibly preconverted) database that does not need the OpenSG scene in memory. This is most useful for visualizing very large static scenes without much interaction (the ChangeList mechanism will only work on elements that are present in the OpenSG layer). For some applications, however, this may not be a problem. An example is the web-based visualization application described in the following section, where all the interaction takes place on the client-side.

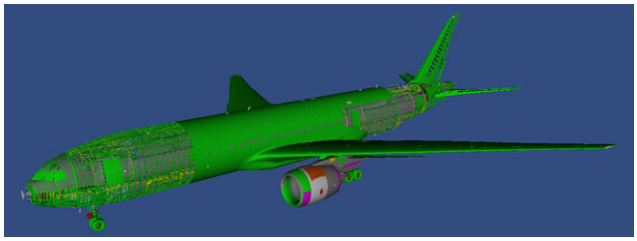


Fig. 8 Boeing 777 CAD model rendered with the VGR back-end. The model consists of over 300 million polygons and is rendered at 100 Hz on a GeForce GTX 470 (with progressive rendering enabled)

With VGR, mixed rendering became very important, because the VGR renderer is quite limited when it comes to (3D-)GUI-elements. A lot of our applications need to display huge models efficiently, but also need the possibility to combine them with dynamic annotations and markers. Here, the Viewport interface of mediator and the thoughtful design of the VGR system helped us a lot. Even though the system is closed (it even creates its own OpenGL context internally), VGR's output is basically a rendered frame buffer (color + z-buffer), which can be combined with other frame buffers (stemming from other Viewports, rendered by other back-ends).

Figure 8 shows a Boeing 777 CAD model rendered in real-time with the VGR back-end. Again, the back-end knows nothing about the clustering setup.

4.3 Application: visualizing large CAD-models in the web browser

In this section, we describe a concrete application built on our system architecture and the mediators explained above. It is a distributed visualization application for large models.

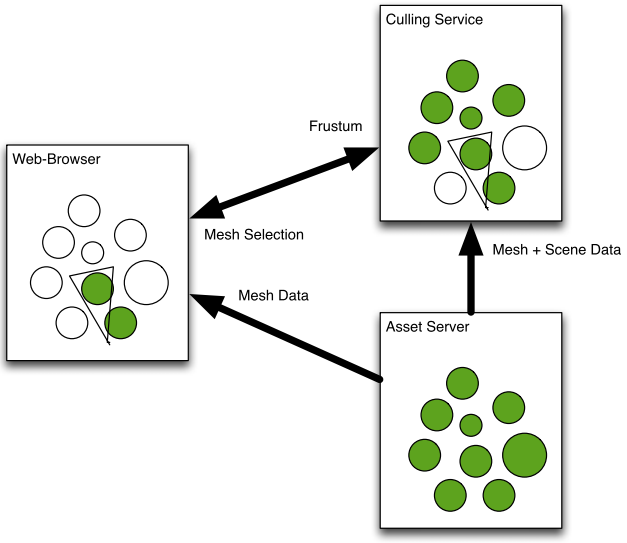


Fig. 9 Schematic data flow between browser, culling server, and asset server

The front-end is simply a WebGL-enabled web browser, rendering a HTML5-page with X3DOM [3]. However, current web-technology is not capable of handling large models efficiently. ("Large models" here means large for web applications, in the order of tens or hundreds of millions of polygons.) Therefore, we use a novel out-of-core approach to minimize the workload in the browser. The key idea is to use an asynchronous, remote culling service. Figure 9 shows the basic data flow. The browser (actually the X3DOM runtime) sends its current view frustum to the culling service, which determines the objects with the largest screen coverage and sends back a list of IDs for these objects. The browser then only fetches these "most important" objects from the asset server. This keeps memory consumption and rendering time manageable on weak devices, which would otherwise not be able to render such complex models. On the other hand, the approach consumes less bandwidth between culling-service and browser than full server-side rendering with video streams. This allows us to maintain high quality and interactivity even in weaker networks, where video streaming does not work well.

The culling service is an InstantReality instance running a special rendering back-end. This back-end does not render a traditional image, but calculates which objects have which coverage in the final rendering (including occlusion). From this information the sorted list of object-IDs is generated, which allows the browser to prioritize important objects. We have implemented the culling service as an Optix-based back-end (as a ray tracer) and as an VGR-based back-end (as a rasterizer). Both cases use a minimalistic scene adapter that basically only converts geometry and establishes a mapping of IDs to objects. The geometry conversion is shared

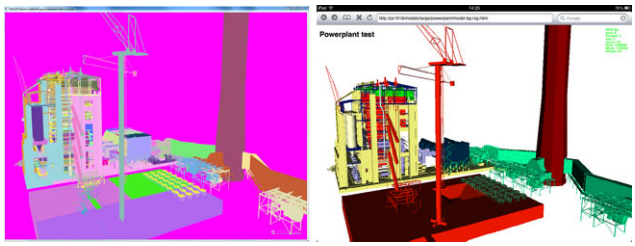


Fig. 10 The browser application with Optix back-end. *Left*: the culling service identifies large objects (in terms of screen coverage). *Right*: the web application running on an iPad only loads and renders the most important objects. The full powerplant model has 14 million triangles, of which the web-application only renders 1.8. Note that the *left image* is only a debug visualization, the culling service does not have to generate an image

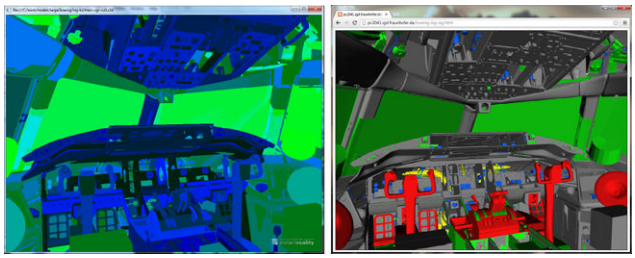


Fig. 11 The browser application with VGR back-end. *Left*: culling service. *Right*: client-side WebGL-rendering. The image shows the cockpit of a Boeing 777 CAD model. The full model has over 350 million polygons, the web application renders 4.2 million

with the other renderers in the Optix/VGR mediator. Material information (apart from transparency) is not necessary.

Figure 10 shows the Optix-based implementation, Figure 11 the VGR-based implementation. In both cases the navigation is smooth in the browser.[1] We believe this show case nicely demonstrates how the freedom obtained by our approach for flexible rendering back-ends can be used to build innovative distributed applications.

5 Discussion

Apart from the points discussed in the case studies, we have made the following observations:

Scene adapter. Converting a stripped-down OpenSG scene into the back-ends preferred representation works very well in general. The mediator design allows us to easily extend OpenSG's well-designed and practice-proven support for incremental updates, multithreading, and clustering to back-ends that never were designed to work with OpenSG or in

[1] http://www.youtube.com/watch?v=zIHV3yC3IYo, http://www.youtube.com/watch?v=h0SUWqJfQsE.

cluster setups. The fact that a scene adapter can (and usually will) change the structure of the scene graph can make it hard to track which changes imply updates to which representatives, but that is the price one has to pay if one wants to feed the back-end with an optimized representation.

Conversion speed. Another issue related to the scene adapter is that the conversion of the scene can be slow if complex operations are necessary (e.g., changing a texture format or converting surface patches into triangles). Performing parts of the conversion only once and caching the result can alleviate this problem. The cached acceleration structures and pre-converted databases described in Sect. 4 are examples of this approach.

Viewport interface. The fact that a rendering back-end only shows a specialized Viewport to OpenSG and the application layer is a mixed blessing. On the one hand, it is a very slim interface that allows us to plug in the back-ends at the most important places. On the other hand, it can be limiting for advanced use cases, because it fails to separate three concerns: *what* to render (scene, camera) and *how* to render it (the back-end), and *where* to render it to (render target). For example, a Viewport that streams to a website cannot be freely combined with each back-end, but would have to be implemented multiple times. Of course, there can still be code-reuse, but a design with clear separation of concerns, as sketched in Sect. 6, would be preferable.

Memory consumption. Building the mediators on OpenSG scenes seems like a waste of memory at first sight. In the worst case, the scene can be represented three times: in the application layer, the OpenSG layer, and in the mediator or back-end. While this can be a problem sometimes (e.g., in the VGR-case), most of the time memory consumption is not excessive and acceptable. The reason is that the scene adapter usually does not duplicate large data (e.g., vertex buffers and images) in main-memory, but translates them directly into the back-ends representation (e.g., CUDA-buffers and OpenGL textures)—an operation that has to occur anyway. Also, the application layer can usually directly use OpenSG data structures, which removes the duplication between application layer and OpenSG layer. (Although this is currently not done in InstantReality.) This leaves only the OpenSG scene as the central scene representation. Even this copy can be eliminated by feeding the mediator directly from the application layer. We provide this option for the VGR back-end, but it should be the exception, because it circumvents our original design and loses two of its strong points: interactive, thread-safe updates, and clustering support provided by OpenSG's ChangeList mechanism.

6 Outlook

We are currently extending the approach described in this paper. The new system will be based on OpenSG 2.0 (our current implementation uses 1.8). The most important extensions are:

General clustering. In the future, we want to use OpenSG more as general data management layer, not only as a scene graph. The goal is to be able to build more general clustered applications. Currently, the whole scene graph (and a few associated things like viewports) is simply mirrored on each cluster node in a client-server cluster [16]. Moving away from the rendering-centric scene graph and ClusterWindow concepts would allow a directed distribution of arbitrary data in a cluster with more specialized cluster nodes while keeping the benefits of OpenSG's sophisticated synchronization mechanism.

Not only rendering. OpenSG as a general data management layer would also make it easier to extend our mediator approach to semantics other than rendering. For example an application scene (interaction), a physics scene (simulation), and a graphics scene (rendering) could coexist and could be kept in sync almost automatically. And these components could even be moved on different cluster nodes without major changes to the application.

Decouple viewport from back-end. To gain more flexibility, we plan to break the tight coupling of a mediator and its specialized Viewport. We want to use OpenSG 2.0's Stage concept [19] to plug in mediator layers (at least for rendering). There will be only one specialized Viewport to which different Stages (i.e., different back-ends) can be attached. The Viewport defines *what* is to be rendered, the Stage *how* it should be rendered.

7 Conclusion

We have described a pragmatic, practice-proven approach to using different rendering back-ends with a common application layer in distributed systems. The approach is based on a mediator layer that can be plugged into the OpenSG infrastructure. This design allows us to naturally extend OpenSG's multithreading and clustering capabilities to use cases that the original OpenSG system cannot handle: different, possibly distributed rendering back-ends. We have focused on the practical aspects of our approach and described a concrete implementation for a commercial VR/AR system. In particular, we have presented two case studies with several rendering back-end and an example application.

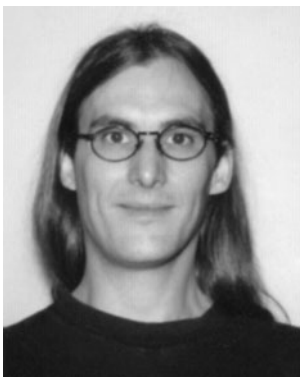
A weakness of our approach is the high memory consumption in some cases. Another issue that we want to address with future work is support for a more general (less rendering-centric) clustering approach.

Acknowledgements The Buddha model was provided by the Stanford 3D Scanning Repository; the Sponza scene was provided by Crytek GmbH and Marko Dabrovic; the Powerplant model was provided by the GAMMA research group at UNC; the 777 model was provided by Boeing. We would also like to thank Jens Keil for the photographs in Fig. 1.

References

1. Arcila, T., Allard, J., Ménier, C., Boyer, E., Raffin, B.: FlowVR: a framework for distributed virtual reality applications. In: Journees de l'AFRV (2006)
2. Behr, J., Dähne, P., Roth, M.: Utilizing X3D for immersive environments. In: Web3D '04 Proceedings, pp. 71–78. ACM, New York (2004)
3. Behr, J., Jung, Y., Keil, J., Drevensek, T., Eschler, P., Zöllner, M., Fellner, D.W.: A scalable architecture for the HTML5/X3D integration model X3DOM. In: Web3D '10 Proceedings, pp. 185–193. ACM, New York (2010)
4. Berthelot, R.B., Royan, J., Duval, T., Arnaldi, B.: Scene graph adapter: an efficient architecture to improve interoperability between 3D formats and 3D applications engines. In: Web3D '11 Proceedings, pp. 21–29. ACM, New York (2011)
5. Brüderlin, B., Heyer, M., Pfützner, S.: Interviews3D: a platform for interactive handling of massive data sets. IEEE Comput. Graph. Appl. **27**, 48–59 (2007)
6. Döllner, J., Hinrichs, K.: A generic rendering system. IEEE Trans. Vis. Comput. Graph. **8**(2), 99–118 (2002)
7. Duval, T., Fleury, C.: PAC-C3D: a new software architectural model for designing 3D collaborative virtual environments. In: ICAT 2011. VRSJ (2011)
8. Fraunhofer IGD: Instant Reality Framework (2012). http://www.instantreality.org
9. Frey, S., Ertl, T.: PaTraCo: a framework enabling the transparent and efficient programming of heterogeneous compute networks. In: EGPGV '10 Proceedings, pp. 131–140. Eurographics Association, Aire-la-Ville, Geneva (2010)
10. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2), 40–53 (2008)
11. 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. ACM Trans. Graph. **29**, 66:1–66:13 (2010)
12. Pharr, M., Humphreys, G.: Physically Based Rendering: from Theory to Implementation. Morgan Kaufmann, San Mateo (2004)
13. Reiners, D., Voß, G., Behr, J.: OpenSG: basic concepts. In: 1. OpenSG Symposium (2002). http://www.opensg.org/
14. Rubinstein, D., Georgiev, I., Schug, B., Slusallek, P.: RTSG: Ray tracing for X3D via a flexible rendering framework. In: Web3D '09 Proceedings, pp. 43–50. ACM, New York (2009)
15. Schwenk, K., Voß, G., Behr, J.: A system architecture for flexible rendering back-ends in distributed virtual reality applications. In: Proceedings of Cyberworlds 2012, pp. 7–14 (2012)
16. Staadt, O.G., Walker, J., Nuber, C., Hamann, B.: A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering. In: SIGGRAPH ASIA 2008 Courses, pp. 41:1–41:10. ACM, New York (2008)
17. Steinicke, F., Ropinski, T., Hinrichs, K.: A generic virtual reality software system's architecture and application. In: ICAT '05 Proceedings, pp. 220–227. ACM, New York (2005)
18. Voß, G., Behr, J., Reiners, D., Roth, M.: A multi-thread safe foundation for scene graphs and its extension to clusters. In: EGPGV '02 Proceedings, pp. 33–37. Eurographics Association, Aire-la-Ville, Geneva (2002)
19. Voß, G., Reiners, D.: Towards a flexible back-end for scenegraph-based rendering systems. In: GRAPHITE '06 Proceedings, pp. 303–309. ACM, New York (2006)
20. Woo, M., Neider, J., Davis, T., Shreiner, D.: OpenGL Programming Guide: the Official Guide to Learning OpenGL, 3rd edn. Addison-Wesley, Reading (1999)

Karsten Schwenk is a software engineer at Fraunhofer IGD's Visual Computing System Technologies department and a doctoral student at TU Darmstadt. His main research interests are interactive global illumination and local reflectance models. Schwenk received his Diploma in Computer Science from the University of Kaiserslautern in 2006.



Gerrit Voß is a senior staff member at the Fraunhofer Project Centre for Interactive Digital Media, NTU Singapore. Prior to this, he worked at the Centre for Advanced Media Technology, NTU Singapore, and at Fraunhofer IGD's Visualization and Virtual Reality department. He is also a member of the core development team of OpenSG. Mr. Voss received his diploma degree in Computer Science from TU Darmstadt in 1997. His research interests are real time rendering, virtual reality, augmented reality, and GPGPU computation.

Johannes Behr heads Fraunhofer IGD's Visual Computing System Technologies department. From 2008 to 2010, he was leading the VR-group of the Virtual and Augmented Reality department. His research interests are virtual reality, computer vision, stereo vision, and 3D interaction techniques. Behr received his M.Sc. diploma in Advanced Software Engineering from the University of Wolverhampton in 1996 and a doctorate from TU Darmstadt in 2005.

Yvonne Jung is a senior researcher at Fraunhofer IGD's Visual Computing System Technologies department. Between 2004 and 2010, she was a member of the Virtual and Augmented Reality group at Fraunhofer IGD. Her research interests include multimodal interaction techniques, virtual characters, and Web3D technologies. In 2011, she received a Ph.D. from TU Darmstadt.

Max Limper is a Ph.D. student in the Interactive Graphics Systems Group at TU Darmstadt, and also works for the Visual Computing System Technologies group at Fraunhofer IGD. His research interests include Web3D, Mesh Compression, Scene Graphs, and Virtual Reality. He holds a M.Sc. in Computer Science from the University of Siegen.

Pasquale Herzig works for the Visual Computing System Technologies group at Fraunhofer IGD. His research interests include Mesh and Scene Graph optimization, Big Data visualization and Web3D. He holds a M.Sc. in Computer Science from the University of Applied Sciences in Darmstadt.

Arjan Kuijper is lecturer at TU Darmstadt and TU Graz, as well as a staff member of Fraunhofer IGD. His research interests cover all aspects of mathematics-based methods for computer vision and graphics. He received a Ph.D. in 2002 from the Department of Computer Science and Mathematics, Utrecht University. In 2009, he finished his habilitation at TU Graz's Institute for Computer Graphics and Vision.