RTSG: Ray Tracing for X3D via a Flexible Rendering Framework

Dmitri Rubinstein* Saarland University DFKI Saarbrücken

Iliyan Georgiev* Saarland University

Benjamin Schug* Saarland University

Philipp Slusallek* Saarland University DFKI Saarbrücken

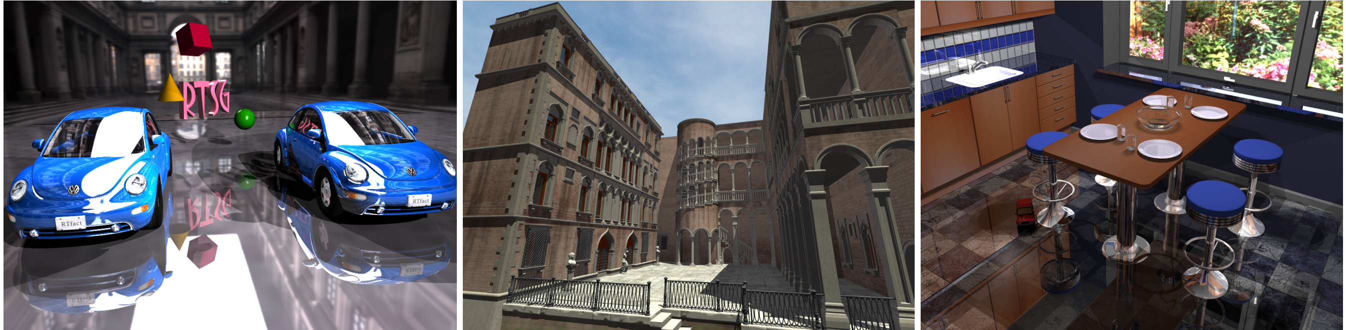

Figure 1: *Scenes rendered interactively with RTSG using ray tracing with shadows, reflection, and refraction.* ***Left****: Beetles (2.4 mln triangles, 4.1 FPS).* ***Middle****: Venice (1.2 mln triangles, 5.6 FPS).* ***Right****: BART Kitchen (108,000 triangles, 1.9 FPS).*

Abstract

VRML and X3D are the most widely adopted standards for interactive 3D content interchange. However, they are both designed around the common restricted functionality available in hardware graphics processors. Thus, most existing scene graph implementations are tightly integrated with rasterization APIs, which have difficulties simulating advanced global lighting effects.

Conversely, complex photo-realistic effects are naturally supported by ray tracing based rendering algorithms [Glassner 1989]. Due to recent research advances and the constantly increasing computing power of commodity PCs, ray tracing is emerging as an interesting alternative for interactive applications.

In this paper we present RTSG (Real-Time Scene Graph), a flexible scene management and rendering system. RTSG is X3D-compliant and has been designed to efficiently support both ray tracing and rasterization using a backend-independent rendering infrastructure. We describe two ray tracing and one rasterization backends and demonstrate that they achieve real-time rendering performance.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing; I.3.4 [Computer Graphics]: Graphics Utilities—Application packages, Graphics packages;

Keywords: X3D, VRML, scene graph, ray tracing, interactive

1 Introduction

Since its first publication in 1994, the VRML standard has undergone numerous enhancements to eventually become X3D. What started as an idea of a third dimension for the world wide web limited to static scenes, is today a powerful platform for creation and deployment of highly dynamic three-dimensional applications. X3D is used for 3D scene interchange, interactive user manuals, simulation, and other applications. Its ability to deliver 3D content over the internet is now just one of its many features.

This is a result of the tremendous improvements in technology over the past decade. Today's commodity PCs have powerful multi-core CPUs with gigabytes of memory and have broadband connection to the internet. Most of the limitations of the original VRML standard have lost their justification over time and have pushed the development of the X3D standard to its current form.

Surprisingly, on the other hand, the visual appearance model of the X3D standard has not changed much since VRML 1.0. The only added feature has been the support for user-defined shaders supported by currently available rasterization APIs. Using shaders, designers can implement custom lightning algorithms to tweak the visual appearance of scenes. However, producing photo-realistic images is quite hard with rasterization, which is the basis of the rendering infrastructure in the X3D standard. This is further complicated by many incompatible shading languages and technical issues, such as the need for multi-pass rendering.

Because rasterization algorithms render each primitive independently, only local illumination information is available in a single rendering pass. As a result, even basic optical effects, such as reflection or shadows, require multiple rendering passes. For example, in order to simulate reflections, one must first render the complete scene from the view of each reflective surface and then from the camera view. While it is possible to support this in X3D, it violates its descriptive approach and causes numerous complications. Furthermore, such technique is not accurate for curved reflective surfaces, ignores inter-reflections, and must be performed each animation frame in the case of object changes.

For producing images with a high degree of photo-realism, ray tracing based algorithms are often used. This is due to the fact that ray tracing closely models physical light transport by simulating light propagation in the scene. This approach naturally supports secondary lighting effects, such as reflection and refraction. However, ray tracing usually requires much more processing time than

*e-mail: {rubinstein, georgiev, bschug, slusallek}@cs.uni-sb.de

rasterization to produce an image. Thus, it has been traditionally used mostly for offline rendering where image quality is crucial and longer rendering times are more tolerable.

The above mentioned technological advances have also influenced ray tracing. Recent advances in research and hardware have enabled ray tracing to achieve real-time performance [Wald et al. 2001; Reshetov et al. 2005; Wald et al. 2007]. These works have made ray tracing an interesting alternative to rasterization for interactive applications, due to its superior image quality, physical realism, and ease of use. Ray tracing and its descriptive approach to scene rendering are a perfect match for X3D. The renderer can automatically generate accurate images, without further help by the designer.

In this paper we describe the design of our X3D-based scene graph library RTSG. It has been built to support interactive rendering using ray tracing. The flexible infrastructure for partial scene updates allows RTSG to efficiently handle large scenes, which is crucial for ray tracing applications. Furthermore, a backend-independent rendering infrastructure also allows for fast vizualization using traditional rasterization. We describe two ray tracing and one rasterization backends and demonstrate interactive performance competitive to or even surpassing existing VRML/X3D browsers.

2 Related Work

The X3D [X3D 2004b] standard defines a hierarchical description containing all objects in a scene and their parameters, such as location or appearance. It represents a descriptive data model and can be used to exchange 3D models between different applications. The standard also specifies a run-time environment that defines how objects respond to user input and how they interact with each other. The run-time environment and the scene description together allow to interactively display the X3D scene. The standard does not specify how the scene should be rendered but only defines the appearance and the behavior of the objects in it at an abstract level.

X3D implementations typically map the standard to a scene graph library such as OpenInventor [Wernecke 1994], Performer [Rohlf and Helman 1994], OpenSG [Reiners et al. 2002], or OpenSceneGraph [OSG 1999]. Such scene graphs use a rasterization-based rendering pipeline, which typically consists of following three stages [Rohlf and Helman 1994; Akenine-Möller and Haines 2002]:

- **APP**
  In this stage the application logic is executed, which may result in modifying, adding, deleting, and moving objects, processing sensors, etc.
- **CULL**
  The scene graph is traversed and all potentially visible portions of the geometry are prepared for rendering.
- **DRAW**
  The geometry prepared in the previous stage is visualized on the screen by issuing graphics API commands.

Often the **CULL** and **DRAW** stages are performed in a single combined traversal step after the application logic for one frame has been executed. Even in this case the scene graph needs to be traversed at least once for each frame in order to visit all nodes that may have been changed. This full traversal is necessary when immediate drawing mode is used for the underlying low-level graphics API. In this approach, the full state of the scene graph is sent to the graphics card every frame, even when the changes are minimal.

However, the preferred way of using current graphics hardware is in retained mode, where a partial or even the full state of the scene is stored in GPU memory. This removes the need for sending static geometry to the GPU each frame, which can greatly reduce the bandwidth requirements. In retained mode, only the minimal set of changes per frame needs to be uploaded to the GPU in order to synchronize the X3D scene with the data on the graphics card.

Most existing X3D systems implement the actual rendering in the **DRAW** stage by adding a virtual rendering method to every scene graph node. This method is called during scene graph traversal and issues low-level graphics API commands that create or update the geometry or the appearance of objects. Such approach works only when a single rendering algorithm is used, respectively a single rendering API, and rendering optimizations in the scene graph often take into account specific properties of both. However, optimizations that make sense for rasterization can be meaningless for ray tracing. For example, state sorting, which is often performed by scene graph implementations, is not needed for ray tracing.

One possible solution to the problem above is to use a double dispatch mechanism, which selects the rendering code not only by the type of the node but also by the type of the rendering method. OpenInventor [Wernecke 1994] provides such functionality by storing a list of callback functions per traversal type, which allows for registering functions for new node types and for overriding the existing ones. Thus, all rendering functionality can be moved entirely into the `DRAW` traversal. Functionality for rendering new nodes can still be added by simply registering their types with the `DRAW` traversal. However, OpenInventor still performs a full traversal of the scene graph from top to bottom for every frame, which is problematic for complex scenes and especially for ray tracing, which needs to maintain optimized prebuilt acceleration structures.

Another alternative to adding a rendering method to every X3D node is the approach used by Avalon [Behr and Fröhlich 1998], where the X3D scene graph is not used directly for rendering. Instead, an OpenSG [Reiners et al. 2002] backend scene graph is created which also performs rendering-depended optimizations. However, OpenSG has exactly the same issues as just discussed.

VrmlRT [Dietrich et al. 2004] was the first implementation of a VRML scene graph with support for ray tracing. It took the same approach as rasterization-based scene graphs and thus was also restricted to one rendering algorithm, namely ray tracing. VrmlRT suffered from too much runtime overhead when animations were present in the scene, which resulted in low rendering performance. The reason for this overhead was that like all existing scene graph implementations VrmlRT was optimized to use a specific rendering API and was not designed to allow incremental scene updates, when small or local changes happened somewhere in the graph. It performed complete re-traversals of the entire graph at each frame to find changes, which can be quite costly for more complex scenes.

3 RTSG

RTSG has been designed with three major concepts in mind: modularity, rendering backend independence, and optimal speed. Figure 2 gives an overview of the system components. The main component of RTSG is the core library, which implements the scene graph, event management, and prototypes, as specified by the X3D standard. Additionally, it implements an extensibility infrastructure. Based on the core, different sets of nodes implement the scene representation. X3D and its extensions are one such set of nodes.

The application has access to renderer plugins. Each plugin consists of a library which connects our X3D scene graph with an actual rendering API. We have developed two ray tracing plugins based on the OpenRT [Dietrich et al. 2003] and RTfact [Georgiev and Slusallek 2008] ray tracing frameworks. We have also implemented a rasterization plugin based on th OGRE [OGR 2000] engine.

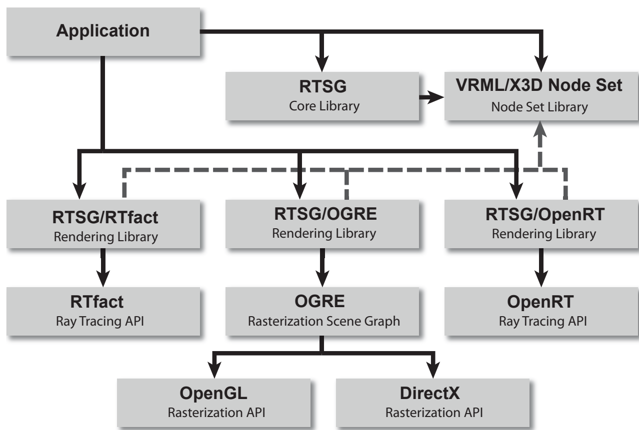

Figure 2: *Overview of RTSG: The application uses the RTSG core and specific nodes to describe the scene. Independently, it creates, configures, and attaches one or more renderers to the scene that read the nodes' data and render it to a device or memory buffer.*

3.1 RTSG Core

The basis of our scene graph implementation is the RTSG core library. This library implements general concepts needed for the implementation of VRML and X3D and provides most of the functionality described in the "Concepts", "Field Type Reference", and "Core Component" chapters of the X3D standard. The idea behind this organization is reducing the VRML and X3D standards to a minimal set of required functionality, and then implementing remaining parts of the standards as extensions and plugins.

The RTSG core defines a high-level concept of a scene graph and provides only the basic features of the X3D standard, such as nodes, single and multi-value fields, event routing, and prototypes. In contrast to many other X3D implementations, the core library does not define any specific node hierarchy. It provides only a basic node class and a tool for generating the code for node class skeletons from the VRML and X3D specifications. Additionally, the user can define new field types not included in the standard. This functionality is inspired by OpenInventor [Wernecke 1994], but is often not available in other X3D implementations. This allows the developer to easily extend library with application specific functionality, and quickly support the growing X3D standard.

For input and output of X3D scenes the core library provides an I/O infrastructure which is inspired by the Xj3D [Xj3 2006] X3D browser and is similar to the SAX API for XML parsing. Scene reader components produce a sequence of commands, such as `beginDocument`, `beginNode`, and `endNode`, instead of directly constructing a scene graph. Scene writer classes produce from this sequence a document in arbitrary format (e.g. VRML, X3D, or compressed binary encoding). The scene constructor component constructs an actual scene from the command sequence, and the scene exporter creates a command stream from an already constructed scene. Also available are scene processing and filtering components which can modify the stream by e.g. adding, deleting and modifying nodes, their names, types, or parameters.

By separating input, output, and scene construction operations into different components, we are able to configure them into various scene processing pipelines. For example, we can convert one X3D representation to another by simply connecting the scene reader from one format to the scene writer that outputs a different format. For this operation no actual scene graph is constructed and only a minimal fixed amount of memory is required. Additionally, this approach allows us to easily support new file format readers and writers in our system without modifying the core library.

The RTSG core library also provides a well defined C++ API, modeled according to the X3D SAI specification [x3d 2004a]. We support functionality for registering listeners specified by the `registerFieldInterest` service defined in the X3D SAI. Additionally to listeners for node fields, RTSG allows registering multiple listeners with every node. These listeners will be reported changes in the fields of the nodes, as well as when a node is removed from or added to the scene graph. This feature is crucial for our optimized rendering support.

3.2 VRML/X3D Node Set

The VRML extension library implements a node hierarchy set according to the "Architecture and base components" part of ISO X3D standard. However, the node classes do not contain any rendering-dependent code. Instead, they only provide code responsible for the non-rendering semantics. This structure allows the VRML/X3D extension library to be used with any rendering algorithm.

4 Rendering with RTSG

As already discussed in Section 2, the traditional rendering pipeline, which performs full scene traversal at each frame, is often not efficient for graphics libraries that operate in retained mode. For such backends, frequent traversal of big parts of the graph may become a bottleneck, especially for large scenes where only small parts change over time. This particularly holds for all ray tracing based renderers, as they always work in retained mode. The geometry and material database of ray tracers is usually created only once and is then continuously updated.

The core of RTSG does not define a fixed rendering pipeline, but instead relies on plugins to control the rendering process. The motivation behind this is that the plugin for some specific rendering algorithm knows the most efficient way to handle changes in the scene graph. Usually, in RTSG a renderer plugin is notified for each change taking place in the **APP** simulation stage. The renderer may directly propagate the changes to the rendering backend or collect and process them in one step only when the next frame is to be rendered.

The traditional pipeline can be implemented in RTSG simply as a renderer plugin that does not register any listeners, but instead performs full scene graph traversal after the **APP** simulation stage.

In the following subsections we describe the rendering infrastructure of RTSG in more detail.

4.1 EXTRA : EXTensible Rendering Architecture

The rendering loop used by RTSG-based applications looks similar to the traditional rendering pipeline and is illustrated in Listing 1. After the scene has been loaded, one or more renderers are created and attached to it. The call to `newFrame` of the `Scene` class corresponds to the **APP** stage, described in Section 2, and performs the propagation of X3D events. The **CULL** stage is specific to the traditional rasterization pipelines and is thus part of a concrete renderer implementation and not of the application. The same also holds for the **DRAW** stage, as the renderer itself decides when to actually send commands to the rendering backend. The application only notifies the renderer with a call to `renderFrame` when the simulation stage has finished, so that the current state of the scene graph can be presented to the user.

```
// load and initialize the VRML/X3D scene
Scene* scene = new Scene;
scene->readFromURL(url);
scene->initialize(getCurrentTime());

// create and attach a renderer to the scene
Renderer* renderer = new MyRenderer();
renderer->attachToScene(scene);

// rendering loop
while (doRender)
{
    // perform VRML/X3D event propagation (APP stage)
    scene->newFrame(getCurrentTime());

    // display the current frame to the output device
    renderer->renderFrame();
}
```

Listing 1: *Pseudo code for scene creation and rendering with RTSG.*

Note that we use the term "rendering" in a very broad sense here. While it typically refers only to visual rendering, exactly the same approach can be used for acoustic rendering, haptics, or even general processing of the scene data. This approach is also ideally suited for synchronizing one X3D scene with another by streaming the changes to another machine/scene.

4.1.1 Customization of the Rendering Code

When implementing a custom renderer plugin, one can override the `attachToScene` and `renderFrame` methods of the base `Renderer` class. Both methods can initiate a full or partial scene graph traversal using the default implementations of the `renderScene`, `renderNode`, or `renderNodes` methods of the `Renderer` class. The first argument to these three methods specifies the requested target – the whole scene, a single node, or a list of nodes. The second argument is a so-called *rendering operation*, which describes how the target should be processed. The rendering operation is always specific to the renderer implementation and can be used to perform different tasks on a single or multiple nodes. For simplifying the implementation of rendering operations, we provide utility classes that can generate vertex, normal, color and other arrays from the standard geometry nodes, such as `IndexedFaceSet`, `Box`, `Sphere`, etc.

When called with the scene or a node list, the rendering method applies the request to each node contained in the list, respectively to each scene root node. When called with a single node, the rendering method tries to find a so-called *node renderer* that is responsible for handling the specific node type. If such node renderer has been registered with the rendering plugin, the request is delegated to it. Otherwise, the node's own `render` method is invoked. This approach is similar to the one used by OpenInventor [Wernecke 1994].

The described customization of the rendering functionality can be viewed as a message passing model, with the rendering operation being the message. One can also define operations for tasks other than rendering. Note that an actual partial or full traversal is performed only as a side effect when rendering methods propagate the rendering operation down the scene graph.

By overriding the rendering methods one can have single or double dispatch strategies depending on both the node and renderer types:

- **Single dispatch on the type of the node** is done by overloading the virtual `render` method of the `Node` base class. This can be done for nodes where only a single rendering scheme makes sense, e.g. rendering architecture dependent nodes.
- **Single dispatch on the type of the renderer class** can be achieved by including the necessary code directly into the custom renderer plugin class by overriding its `renderFrame` and `renderNode` methods of the base `Renderer` class. This is useful when some common operations need to be performed either for each frame or for all node types.
- **Double dispatch on the type of the node and the renderer class** is the most flexible way of rendering a single or multiple nodes specific to one rendering backend. Node renderers can be loaded and registered at run-time and override existing rendering functionality inside the node classes.

4.1.2 Event Driven Handling of Scene Changes

In the previous subsection we described how the rendering loop in Listing 1 can be customized by overriding the `attachToScene` and `renderFrame` methods of the `Renderer` class. However, because `attachToScene` is called before any changes have taken place and `renderFrame` is called always after the scene has changed, full scene graph traversal would still need to be performed in order to detect and propagate all changes. In order to avoid this, we use the infrastructure of the RTSG core library for registering *listeners* for possible changes in the scene. These listeners are called directly upon changes in the `newFrame` method (i.e. during the **APP** stage) or any other manipulation of the scene graph by the application. They can either directly update the state in the rendering backend by sending rendering operations or record the changes for later processing.

For ray tracing and other retained mode rendering backends, as well as backends that maintain internal scene graphs of their own (e.g. OpenSG or OGRE), one would create the backend scene graph in the renderer's `attachToScene` method. Additionally, one would register listeners with all nodes which may be modified in the X3D scene, potentially all. In order to detect modifiable nodes, the routing graph and the `Script` nodes of the scene can be analyzed.

An important feature of EXTRA is that it allows using multiple rendering techniques simultaneously. For example, one could have multiple types of renderer plugins (visual, audio, haptics) that output to different devices. As another example, a hybrid rendering approach may combine rasterization and ray tracing, which may require multiple rendering passes, where the different rendering algorithms need to inter-communicate. This would not be possible if all rendering code would be placed inside the node classes.

5 Applications

In this section we present specific applications of RTSG for real-time rendering using different rendering algorithms and backends. We have implemented renderer plugins for two ray tracing and one rasterization based graphics libraries.

5.1 Ray tracing

Ray tracing has been originally introduced as a method for realistic rendering of three-dimensional scenes [Appel 1968]. In the classic recursive ray tracing [Whitted 1980], primary rays are first traced through the camera to determine the directly visible geometry. In order to shade the found hit points, shadow rays are traced to check the light sources in the scene for occlusion, while secondary rays are recursively cast in the case of reflective or refractive geometry.

The biggest advantage of ray tracing over rasterization is that most real-world effects are straightforward to simulate. These include depth of field, motion blur, soft shadows [Cook et al. 1984], as well as full global illumination [Pharr and Humphreys 2004].

Traditionally, ray tracing has been used for high quality off-line image generation, as its performance has been too low compared to hardware rasterization. However, recent technological advances have enabled ray tracing to achieve real-time performance, and have made it an interesting alternative to rasterization for interactive applications. State-of-the-art ray casting algorithms operate on bundles of rays to efficiently amortize intersection costs and rely on highly optimized acceleration structures built over the scene geometry [Wald et al. 2001; Reshetov et al. 2005; Wald et al. 2007].

The versatility of ray tracing as a visibility sampling technique makes it useful not only for rendering, but also for other tasks performed in interactive applications, such as collision detection or object picking. While such tasks traditionally utilize different data structures, in an interactive ray tracing setting the same structures can be used for accelerating both rendering and object interaction.

5.1.1 Using EXTRA for Ray Tracing Backends

The X3D scene graph structure is not directly optimized for ray tracing based rendering. This is due to the fact that X3D, like its predecessor VRML, have been originally developed with rasterization as a target rendering technology.

When using rasterization, any *potentially* visible geometry is usually processed at each rendering pass, in order to minimize the upload to the GPU. In ray tracing, on the other hand, geometry is usually created once and cached, and only updated by changing e.g. transformation matrices or shading parameters.

Because of this difference, a second optimized scene graph version is required for ray tracing based rendering (called *speed-up graph* in the following). The primary task of this speed-up graph is to keep the ray tracing objects synchronized with their corresponding X3D nodes. Thus, X3D node changes must be propagated to the corresponding ray tracing object(s). For this purpose speed-up graphs use RTSG's listener API.

The speed-up graph introduces some additional memory overhead, due to the replication of the scene graph data. The replication of the scene graph structure can be minimized by keeping only the non-static nodes separately in the speed-up graph. The static X3D nodes corresponding to one subgraph can be collapsed into a single speed-up graph node. The replication of the node data can be minimized by using the same representation for the X3D and the rendering backend, e.g. to share vertex arrays. However, due to the possible differences between the memory layout of X3D fields and the backend data structures, this might not always be possible.

5.1.2 OpenRT

Our first ray tracing rendering backend implementation for RTSG is based on the OpenRT API [Dietrich et al. 2003]. OpenRT has a syntax similar to OpenGL, but there are major differences in the semantics. As ray tracing is not efficient for immediate drawing, OpenRT based programs must define the entire geometry before rendering begins. Geometry is encapsulated in objects, which are similar to OpenGL's display lists, and can be instantiated multiple times in the scene. As soon as an object has been defined, OpenRT constructs a spatial index structure for it. While redefining a large object can be time consuming, rigid animation of instances is efficiently supported via changes to their transformation matrices without modifying the corresponding spatial index structure [Wald et al. 2003]. OpenRT maintains a top-level acceleration structure over all instances, which is rebuilt when instances have changed.

Two major requirements for our OpenRT based RTSG rendering backend need to be fulfilled, in order to achieve maximum performance with X3D scenes. First, the multi-level X3D scene graph has to be optimally mapped to the two-level OpenRT scene graph. Second, we want to avoid traversing the whole X3D scene graph every frame, and instead propagate changes directly using listeners.

One straightforward way to map the X3D scene graph to OpenRT would be to put all geometry into a single OpenRT object. However, this is only efficient for fully static scenes, as objects would have to be recreated on every animation frame. Another straightforward approach would be to create one OpenRT object for each `Shape` node. However, in this case OpenRT would also not perform well, as there would be too many instances, which would impact ray casting performance. Therefore, we have taken a hybrid approach, which tries to find an optimal number of OpenRT objects and instances. We analyze the X3D graph, locate its largest static subgraphs, and merge dynamic nodes that transform coherently.

In order to avoid full traversal of the X3D scene graph, we use a speed-up graph, which is synchronized with scene changes. The speed-up graph is constructed after the X3D scene has been created and contains only nodes that influence the transformation stack. It is designed to minimize the number of OpenRT objects and instances.

As already mentioned, we additionally take care of `Script` nodes, as they may also modify the scene state. For scripts with deactivated "direct output", all nodes that they can modify are assumed to be dynamic. When "direct output" is activated for a script, we conservatively assume that all X3D namespaces accessible by the script are fully dynamic, as such scripts can directly modify the scene graph via the SAI API. Our algorithm tries to detect the largest static subgraphs in the scene which cannot be modified neither by routes nor by scripts and merges them into a single OpenRT object.

Objects' appearance changes are directly propagated to OpenRT. When a change modifies the transformation hierarchy, we traverse the speed-up graph in order to compute accumulated transformation matrices for the influenced OpenRT instances. As each node in the speed-up graph stores the accumulated matrices of all its parents, we directly start traversing from the modified speed-up graph node.

The speed-up graph additionally manages non-geometric objects, such as light sources and view points, which might be transformed by the graph. This is required because OpenRT only manages such objects in world coordinate space.

5.1.3 RTfact

We have also developed a module for RTSG which uses RTfact [Georgiev and Slusallek 2008] as a rendering backend. RTfact is a modern real-time ray tracing library that utilizes generic programming concepts to deliver both performance and flexibility. It consists of a collection of loosely coupled components, which allows easy integration of new algorithms and data structures with maximum runtime performance, while leveraging as much of the existing code base as possible. The efficiency of C++ templates allows RTfact to achieve fine component granularity, to incorporate multiple shading models, and to generically handle arbitrary rays.

A key feature of RTfact is that it can robustly handle multiple rendering configurations simultaneously, in contrast to OpenRT. This allows us to use the best suitable algorithms and data structures for a specific application in order to achieve best performance. RTfact also directly exposes ray tracing functionality to the application, which for example enables object picking or collision testing.

	Beetles	Venice	Conference	Campus
inView	3.5	4.1	8.1	2.9
RTSG/OpenRT	3.8	4.4	8.2	2.9
RTSG/RTfact	**4.1**	**5.6**	**17.4**	**4.8**

Table 1: *Performance in FPS of RTSG with OpenRT and RTfact on static scenes compared to inView.*

For managing dynamic scenes, RTfact maintains its own internal data structure, which contains only geometry and appearance information. It supports multiple levels of instantiation which allows it to mirror the RTSG scene graph, without the need for an intermediate speed-up graph, like with our OpenRT plugin. Such organization has another two advantages. First, it simplifies the RTSG integration module, as it now only needs to propagate scene changes to the RTfact backend. Second, it ideally allows for best optimization, as the underlying rendering backend now contains enough information about the scene organization, which enables efficient building of acceleration structures for ray tracing using the scene hierarchy.

The RTfact renderer plugin attaches listeners to the RTSG scene graph nodes at initialization time. During event propagation in RTSG, each event is translated and immediately sent to the RTfact backend. There, the modified nodes are updated and pushed into a priority queue, which sorts the events according to the depth of the modified nodes in the scene graph. As soon as all scene modifications have been performed, the queue is flushed which results in updating the internal ray tracing acceleration structures.

RTfact also allows attaching non-geometry objects to nodes, which makes light source transformations, for example, straightforward. However, RTfact currently does not optimize well static and dynamic parts of the scene. As a result, in the presence of animations a separate node is created in its internal scene graph for each X3D shape node. Our RTfact plugin also currently does not support coordinate interpolation from corresponding X3D nodes.

5.2 Rasterization

In order to demonstrate that RTSG is as well suited for rasterization as for ray tracing, we have implemented a rasterization-based renderer using the OGRE [OGR 2000] rendering engine.

OGRE maintains its own internal graph structure for the scene geometry, which has some differences to X3D. First, OGRE's scene graph is actually a tree, i.e. each node (except for the root node) has exactly one parent node. Second, not all X3D nodes have a direct counterpart in OGRE. A `SceneNode` node in OGRE is actually a `Transform` node in X3D. Also, geometry, textures, and material properties are stored outside the OGRE scene graph in `Mesh` and `Material` objects. Geometry instances correspond to `Entity` nodes in OGRE, and must be attached to `SceneNodes`. `Entity` nodes contain references to `Mesh` and `Material` objects.

Similarly to the ray tracing rendering plugins, our OGRE plugin also uses the event notification infrastructure of RTSG. When the plugin is attached to an RTSG scene, it creates a corresponding scene graph in OGRE. X3D `Transform` nodes are directly mapped to `SceneNodes` in OGRE. As OGRE's scene graph must be a tree, every X3D `Transform` node with multiple parents is replicated as multiple `SceneNodes` in OGRE. The same holds for X3D `Shape` nodes, for which OGRE `Entity` nodes are created.

The renderer attaches listeners to the RTSG scene graph nodes that may change during the lifetime of the scene. Whenever a listener is triggered, the change is immediately propagated to the corresponding OGRE nodes/objects.

	Troopers	Kitchen	Conference	Campus
inView	N/A	1.5	6.9	2.4
RTSG/OpenRT	12.0	1.6	8.2	**2.9**
RTSG/RTfact	**16.7**	**1.9**	**11.2**	2.7

Table 2: *Performance in FPS of RTSG with OpenRT and RTfact on dynamic scenes compared to inView. The Troopers animation does not work with inView.*

6 Results

In this section, we present performance results of RTSG using our three rendering plugins, tested on a number of VRML/X3D scenes.

The Campus scene (Figure 3 right) is a massive architectural model consisting of 4.3 million triangles. It tests the ability of a renderer to handle complex static scenes.

The Beetles scene (Figure 1 left) has a lot of detailed static geometry consisting of 2.7 million triangles.

The Venice scene (Figure 1 middle) is fully static and consists of 1.2 million triangles. It contains three complex statues on the ground, and the buildings have reflective and refractive windows. The scene also has very large textures which require rasterization based renderers to scale them down in order to fit them into the GPU memory.

The Troopers scene (Figure 4 number 2) consists of 48 rigid body animated figures adding up to about 19,000 triangles. With its many Routes and changing transformations, this scene is a good performance test for the event processing of a VRML/X3D browser.

The Escherknot scene (Figure 4 number 4) contains 4,600 triangles and a CoordinateInterpolator with two key frames. It tests how efficiently a browser implements key-frame animation.

The Conference Room [Ward 1996] (Figure 3 left) and BART Kitchen [Lext et al. 2000] (Figure 1 right) are popular ray tracing benchmark scenes. The Conference Room contains static detailed geometry made up of 280,000 triangles. The BART Kitchen (ca. 108,000 triangles) contains a lot of reflective and refractive geometry and a toy vehicle driving around the floor.

We additionally put a rotating teapot in the Conference and Campus scenes, in order to measure how different rendering plugins behave when a simple animation takes place in a complex scene.

All numbers in the performance tables are in frames per second (FPS). Ray tracing performance was measured on an Intel Q9400 CPU with 4GB RAM, while rasterization performance was measured on a machine with an AMD Phenom 9850 CPU, 2GB RAM, and a Radeon HD4850 graphics card. The best results in each table are shown in bold.

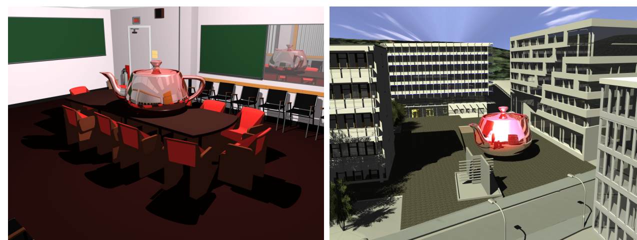

Figure 3: *Two of our ray tracing test scenes.* ***Left****: Conference Room w/ teapot (17.4 FPS);* ***Right****: Campus w/ teapot (6.3 FPS).*

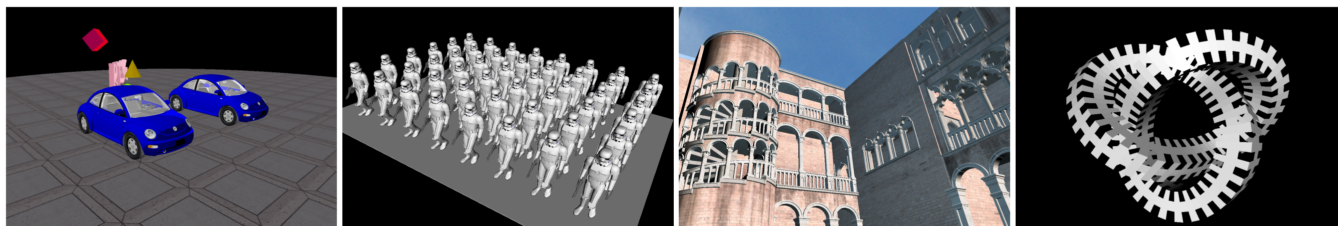

Figure 4: *Test scenes used with OGRE for rasterization performance comparison. From left to right: Beetles, Troopers, Venice, Escherknot.*

6.1 Ray Tracing

Tables 1 and 2 compare the performance of our RTfact and OpenRT renderer plugins for RTSG against the inView VRML browser. inView is an improved version of VrmlRT and uses OpenRT as a rendering backend. In the presence of animations, it performs full scene graph traversal for each frame, in order to detect changes and to update OpenRT's acceleration structures.

For the static scenes, inView and RTSG/OpenRT perform almost identically. RTSG/OpenRT is slightly faster because of its ability to optimize objects and instances better. RTSG/RTfact consistently outperforms the other two, mostly due to RTfact's superior rendering performance.

Even though our RTfact integration does not currently optimize static and dynamic nodes in the presence of animations, RTSG/RTfact again outperforms the other two renderers on most dynamic scenes. This is also due to the fast build-from-hierarchy algorithm for RTfact's internal acceleration structures.

On the Conference and Campus scenes RTSG/OpenRT performs most consistently, as can be noted by comparing the results in Table 1 and Table 2. Switching on the rotating teapot animation does not impact its performance, as only one node of the speed-up graph is traversed for each frame. The performance RTSG/RTfact drops significantly on both scenes, because of the lack of optimizations in dynamic scenes. As a result, the internal ray tracing acceleration structures contain a lot of geometry instances, which hurts rendering performance.

inView also experiences performance loss in the presence of animations. This is due to the overhead introduced by the full scene graph traversal for each frame, which amounts to 14ms on the Conference scene and 66ms on Campus. As a serial task, this traversal overhead can become a rendering bottleneck if the visualization is faster, i.e. on many-core machines or with rasterization, or on more complex scenes.

6.2 Rasterization

We also compared our OGRE rendering plugin against a number of existing rasterization based VRML/X3D browsers, specifically BS Contact (v7.1), Octaga Player (v2.2.0.10), Flux Player (v2.1), InstantReality (public beta 6), and Xj3D (v1.0).

We gathered performance results on Windows, as it is the operating system supported by most of the publicly available browsers. We used Fraps [Beepa 2009] to measure the frame rates.

Most browsers (including our own) had problems with the head-up display with clickable text in the Troopers scene. Only InstantReality and Octaga Player managed to render it correctly. In all other browsers, it was either ignored altogether, probably due to missing support, or rendered at a completely wrong position.

	Beetles	Troopers	Venice	Escherknot
BS Contact (GL)	14.8	11.6$^{\diamond}$	151.6$^{\otimes}$	**631.8**
Octaga Player	38.2	14.0	64.2	64.2
Flux Player	30.4	9.2$^{\diamond}$	24.7	32.0
InstantReality	64.2	32.3	11.6	64.1
Xj3D	26.8	27.2$^{\diamond}$	123.4$^{\otimes}$	256.5
RTSG/OGRE	**113.7**	**51.6**$^{\diamond}$	**631.2**	192.5

Table 3: *Performance in FPS of RTSG with OGRE compared to other VRML/X3D browsers, measured with Fraps. Incorrectly rendered scenes are marked with* $^{\otimes}$*, and* $^{\diamond}$ *indicates missing head-up display with user interface on the Troopers scene.*

The Venice scene was problematic as well. Xj3D suffered from rendering artifacts and BS Contact loaded only part of the textures, keeping large parts of the scene untextured. InstantReality displayed the scene correctly, but was slowed down considerably by the large textures. Some frames took up to 6 seconds to render.

As can be seen from Table 3, RTSG/OGRE consistently outperforms the other browsers by a considerable margin, often by more than an order of magnitude. This is mainly OGRE' merit, as we simply propagate scene changes without performing any node optimizations. OGRE's rasterization engine is well optimized for dynamic game environments, and performs efficient render state management and spatial culling. RTSG/OGRE is not the fastest renderer only on the Escherknot scene. This is due to our implementation of `CoordinateInterpolators` which replaces the complete mesh on each frame instead of performing the interpolation on the graphics card. As a result our performance is currently suboptimal for such scenes.

7 Limitations and Discussion

One of the problems when using ray tracing with X3D is the missing support in the appearance model for basic material parameters, such as reflection and refraction. We added support for such parameters by using the OpenRT-specific `ORTAppearance` node which can support arbitrary material parameters. As a result, only our system can play the modified scenes.

As already mentioned, the original VRML material definition was designed at a time when GPUs had a fixed shading model. However, complex lighting effects, such as shadows, reflections, and programmable materials, are supported by many of today's GPUs. Instead of extending the VRML lighting model in the X3D standard, only direct support for shading languages has been added. While this makes shading more flexible, adding imperative code to appearance nodes violates the descriptive nature of X3D and limits the use of different rendering algorithms, such as ray tracing.

It is difficult to visualize materials in the same or even similar way when using different rendering algorithms. X3D (and computer

graphics in general) is missing a portable, unified, declarative shading language that can be efficiently mapped to both ray tracing and rasterization. One solution to this problem might be to use external prototypes. In the X3D scene, we would use an external prototype to specify custom material parameters, such as reflection and refraction coefficients. This prototype would be then mapped to a custom material node if the browser recognizes and supports it, or to a fallback standard material node otherwise. Still, X3D has no standard way of describing rendering semantics. Thus, although standard-compliant, such an approach would not be fully portable.

Another problem that we have experienced is the implementation overhead for avoiding scene graph traversal. Situations where the scene topology changes, e.g. when adding or removing nodes from the scene graph, are hard to map to the backend scene graph without full re-traversal of the scene. Additionally, the performance improvement gained by avoiding full traversal is more noticable only on dynamic scenes with a large number of static nodes, or when visualization is very fast.

8 Conclusions and Future Work

In this paper we presented RTSG, a flexible X3D-compliant scene management and rendering system. Its highly customizable rendering pipeline enables the use of various visualization algorithms and rendering backends, such as ray tracing and rasterization.

There are still opportunities for improving the performance of our renderer plugins. For example, our RTfact and OGRE plugins do not currently analyze the routes in the scene in order to optimize static and dynamic geometry, as done for OpenRT. Doing so would allow the rendering backends to group geometry more efficiently.

Another current limitation in our system is the missing multi-threading support. While the RTfact and OpenRT ray tracing engines are multi-threaded, RTSG event processing is still serial.

In future, we plan to extend RTSG to a full-featured virtual reality system. We are already working on a sound sumulation sub-system, which uses the RTSG's rendering plugin infrastructure, and we are looking into haptics integration.

Another interesting topic is hybrid rendering, incorporating both rasterization and ray tracing. For this we would need to improve the messaging mechanisms between our rendering components, as it might require more information than simple integer-encoded rendering operations.

Finally, we plan to release RTSG as an open source library.

References

AKENINE-MÖLLER, T., AND HAINES, E. 2002. *Realtime Rendering (2nd edition)*. A K Peters, July. ISBN: 1568811829.

APPEL, A. 1968. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ACM, New York, NY, USA, 37–45.

BEEPA, 2009. Fraps: Real-time video capture and benchmarking. http://www.fraps.com/.

BEHR, J., AND FRÖHLICH, A. 1998. Avalon, an Open VRML VR-AR system for Dynamic Application.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 137–145.

DIETRICH, A., WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, 23–31.

DIETRICH, A., WALD, I., WAGNER, M., AND SLUSALLEK, P. 2004. VRML Scene Graphs on an Interactive Ray Tracing Engine. In *Proceedings of IEEE VR 2004*, 109–116.

GEORGIEV, I., AND SLUSALLEK, P. 2008. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *IEEE/Eurographics Symposium on Interactive Ray Tracing 2008*.

GLASSNER, A. 1989. *An Introduction to Ray Tracing*. Morgan Kaufmann. ISBN 0-12286-160-4.

LEXT, J., ASSARSSON, U., AND MÖLLER, T. 2000. BART: A Benchmark for Animated Ray Tracing. Tech. rep., Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, May. Available at http://www.ce.chalmers.se/BART/.

2000. OGRE Object Oriented Graphics Rendering Engine. http://www.ogre3d.org.

1999. OpenSceneGraph. http://www.openscenegraph.org.

PHARR, M., AND HUMPHREYS, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Elsevier Science & Technology Books.

REINERS, D., VOSS, G., AND BEHR, J. 2002. OpenSG: Basic Concepts. In *First OpenSG Symposium*.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics 24*, 3.

ROHLF, J., AND HELMAN, J. 1994. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Computer Graphics 28*, Annual Conference Series, 381–394.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*.

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*.

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

WARD, G., 1996. MGF, Materials and Geometry Format Package. http://radsite.lbl.gov/mgf/HOME.html.

WERNECKE, J. 1994. *The Inventor Mentor*. Addison-Wesley. ISBN 0-201-62495-8.

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM 23*, 6, 343–349.

2004. ISO/IEC 19775-1:200x, Extensible 3D (X3D), Part 2: Application programmer interfaces. http://www.web3d.org/x3d/specifications/x3d_specification.html.

2004. ISO/IEC 19775:200x, Extensible 3D (X3D). http://www.web3d.org/x3d/specifications/x3d_specification.html.

2006. Xj3D, a toolkit for VRML97 and X3D content written completely in Java. http://www.xj3d.org/.