Institute of Computer Graphics and Knowledge Visualization

Graz University of Technology

Master's Thesis

Web Service Integration Into OpenSG

Andreas Schiefer

Graz, 2010

Abstract

Scene graphs are widely used for virtual environments and other graphical applications as they provide an efficient way to manage and render big virtual scenes. But most virtual reality applications provide only limited editing capabilities and importing and exporting of data is not always feasible. Also, various different input devices may be used for controlling and editing the scene graph, but the integration of such devices often requires a lot of code.

This thesis describes a flexible software architecture for integrating a RESTful web service into applications based on the OpenSG scene graph system. The presented architecture can be integrated into any OpenSG based application with only a few lines of source code and offers access to the scene graph of the application for other applications, even over a network. Thus, using the web service, any application can access and modify the scene of the OpenSG application, which allows for many new ways of interacting with the scene graph of an OpenSG application. The web service not only allows to view and manipulate the scene graph from any webbrowser but also allows to create user interfaces for interacting with the scene graph for all kinds of web enabled devices.

To demonstrate the flexibility of the presented web service approach, an example application that provides an additional OpenSG based graphical representation for the Java based “SweetHome3D” interior design application is shown. Additionally, also a browser based client to the web service is presented, that allows to view and modify the scene graph of an OpenSG application using any modern webbrowser.

Parts of the work described in this thesis were also submitted and accepted as a paper to the *Web3D 2010 Conference*[1] with the title *Service-Oriented Scene Graph Manipulation* [20] and to the *05. Kongress Multimediatechnik Wismar 2010* with the title *Enlightened by the Web* [21].

[1]http://conferences.web3d.org/web3d2010/

Zusammenfassung

Szenengraph-Systeme werden in vielen virtuellen Umgebungen und anderen grafischen Anwendungen verwendet, da sie effizient große virtuelle Szenen verwalten und darstellen können. Aber die meisten Anwendungen für virtuelle Umgebungen stellen nur begrenzte Möglichkeiten zum Ändern der Szene zur Verfügung. Das Importieren bzw. Exportieren der Daten ist in vielen Fällen auch keine akzeptable Möglichkeit. Außerdem können viele unterschiedliche Geräte für die Kontrolle des Szenengraphen verwendet werden, welche aber oft nur mit viel Aufwand in das Programm eingebunden werden können.

Diese Arbeit beschreibt eine flexible Software Architektur für die Integration eines “RESTful” Webservice in Anwendungen, welche auf dem OpenSG Szenengraph-System basieren. Die beschriebene Architektur kann mit minimalen Änderungen am Quelltext in jede OpenSG basierte Anwendungen integriert werden und ermöglicht anderen Anwendungen den Zugriff auf den Szenengraphen der OpenSG Anwendung, auch über ein Netzwerk. Durch das Webservice kann also jede Anwendung auf den Szenengraphen der OpenSG Anwendung zugreifen und diesen auch ändern. Diese Funktionalität ermöglicht viele neue Wege um mit dem OpenSG Szenengraphen zu interagieren. Dadurch ist es nicht nur möglich den Inhalt des Szenengraphen von jedem Webbrowser aus anzusehen und zu bearbeiten, sondern auch Benutzerschnittstellen für die Interaktion mit dem Szenengraphen für alle Geräte mit Web-Unterstützung zu erstellen.

Um die Flexibilität der vorgestellten Webservice Architektur zu demonstrieren, wird eine Beispielanwendung vorgestellt, welche eine zusätzliche grafische Ausgabe basierend auf OpenSG für die in Java programmierte Innenarchitektur Anwendung “SweetHome3D” bereit stellt. Zusätzlich wird auch eine Webbrowser basierte Benutzerschnittstelle für das Webservice vorgestellt, welche es erlaubt den Inhalt des Szenengraphen mit jedem modernen Webbrowser anzusehen und zu verändern.

Die Inhalte der vorliegenden Arbeit wurden zum Teil auch unter dem Titel *Service-Oriented Scene Graph Manipulation* [20] zur *Web3D 2010 Konferenz*[2] und unter dem Titel *Enlightened by the Web* [21] beim *05. Kongress Multimediatechnik Wismar 2010* eingereicht und akzeptiert.

[2]http://conferences.web3d.org/web3d2010/

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....	....
date	(signature)

Contents

Chapter 1

Introduction

The term *web service* describes a general concept in software architecture that allows an application to share resources or functionalities with other applications that can access the shared resources or functionalities over a network using standardized protocols [3]. Integrating a web service into an application is therefore a way to allow other applications to access resources of the application over a network.

OpenSG is an open source scene graph system that, as other scene graphs, allows for a hierachical organization of the contents of a virtual scene. Scene graphs also allow application programmers to display the virtual scene on all kinds of different display devices – ranging from standard computer monitors to virtual environments that provide a three dimensional view of the scene – in a convinient and efficient way by providing built in optimization techniques and an abstraction of the underlaying graphics system.

The idea of this thesis, integrating a web service into applications that are based on the OpenSG scene graph system, therefore allows any application to access and manipulate the content of the scene graph of an OpenSG application over a network. This way, any application can change the scene graph and thus the virtual scene that gets displayed on some display device.

The software architecture described in this theses can be used for many usage scenarios as for example to provide a graphical visualization based on OpenSG for non-3D applications. Another use case is to provide editing capabilities to an OpenSG application by accessing the scene graph of the application from an editing software. Further, the described software architecture addresses some common problems with virtual reality solutions as described in "Service-Oriented Scene Graph Manipulation" [20]:

Modeling Visualization Cycle Most virtual environments offer only limited modeling capabilities. As "standard" modeling software (Maya, 3ds Max, etc.) is seldom adapted to virtual reality (VR), many VR systems are just viewers. In this case, the modeling-visualization-cycle is interrupted; i.e. a human modeler has to switch frequently between modeling environment and VR visualization. Especially during presentations where customers inspect 3D models in VR and want to apply model changes (customization), an interrupted modeling visualization cycle is not feasible.

VR Correlated UX Due to differences in VR systems (High-resolution projection walls, CAVE environments, etc.), the user interaction and user experience (UX) implement different paradigms. The interaction capabilities may vary from desktop-based mouse and keyboard interaction to multi-touch or 3D gestures. These shifting hardware situations result in a substantial adaption and implementation effort.

The flexible software architecture described in this thesis can be used to overcome this problems. Some example applications demonstrating the described software architecture and its flexibility are also presented in this thesis.

Chapter 2

Web Services

Web services provide an implementation for a service-oriented architecture (SOA) [14]. The OASIS SOA Reference Model [15] describes service-oriented architecture as

> ... a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.

As stated in "Service-Oriented Scene Graph Manipulation" [20] the main characteristic of SOA is the loose coupling of services. Because these services communicate in a very flexible way by only exchanging messages, two applications need only to agree on the message format and are independent from the actual application code.

The World Wide Web Consortium (W3C) defines web services as follows [3]:

> A web service is a software system designed to support interoperable machine-to-machine interaction over a network. ...

Beside web services there exists also a number of alternative solutions for machine-to-machine interaction like Open Network Computing Remote Procedure Call (ONC RPC) [22], Distributed Component Object Model (DCOM) [4], Common Object Request Broker Architecture (CORBA) [16], Remote Method Invocation (RMI) [24], etc.

A comparison of these systems can be found in "An Overview of Distributed Programming Techniques" [10]. While ONC RPC focuses only on remote procedure calls, the other systems mentioned use a more object-oriented approach based on distributed objects. Most of them use some sort of definition language to describe the interface of the remote procedures or distributed objects. This interface description then gets converted by a custom compiler into ready to use source code for one of the supported programming languages – this limits the use of such systems to programming languages where such an interface compiler is available. This is one of the reasons why many of these systems are only available for a limited number of platforms or programming languages.

Compared to such machine-to-machine middleware systems, web services have a number of advantages [20]:

Platform support Web services use the world wide web for communication. Every platform got already built-in support for the web. Web services can be consumed by web browsers. Since most platforms already include a web server even hosting web services can be done without additional efforts. Other module communication systems are only supported by a small number of platforms (e.g. DCOM is available only for the Windows platforms) or do not ship with the platform and need to be installed separately (e.g. CORBA).

Language support Almost every programming language provides support for internet protocols and processing of Extensible Markup Language (XML) documents (C++, Java, C#, JavaScript, Python, Ruby, etc.). Although also other middleware systems (e.g. CORBA) claim to be language independent one will find only support for C, C++, and Java.

Internet The major drawback of systems like DCOM and CORBA is that communication across the internet is nearly impossible. These middleware system use binary formats which are blocked by most firewalls. Even in the same department it can be a painful task to configure the network to successfully communicate using DCOM/CORBA.

As there are multiple possibilities of how to implement web services, the two main principles to implement web services are now described.

2.1 SOAP

SOAP [12] (formerly for Simple Object Access Protocol) is a protocol for exchanging structured information between computers based on XML. It can be used with a variety of transport protocols but most commonly it is used for remote procedure calls transported via HTTP. Because SOAP messages consist enirely of XML it is truly platform and language independent [5].

Every SOAP message consists at least of an `Envelope` and a `Body` element. The

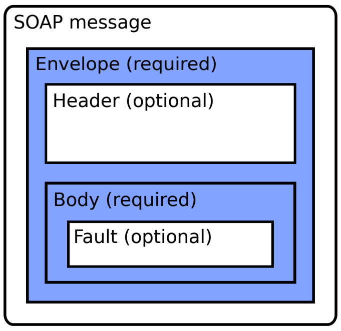


Figure 2.1: *Structure of a SOAP message. Blue parts (Envelope and Body) are required for every message sent to or from a SOAP web service. (Adapted from "Web Services Essentials" [5])*

`Envelope` specifies the SOAP version using XML namespaces and contains at least the `Body` element. The `Body` contains the "payload" – the information that is being sent with the message. Additionally the `Envelope` may also contain an optional `Header` element which can be used for application-level requirements like authentication. In case of an error, the `Body` may contain a `Fault` element with an error description. In Figure 2.1 the described structure of a SOAP message is graphically represented and Listing 2.1 shows an example XML SOAP message [5].

Usually SOAP is used together with the Web Service Description Language (WSDL) [6] which is a specification for describing web services in a common XML grammar. As listed in [5] WSDL describes the following aspects of a web service:

- Interface information about all public functions
- Information about the data types for all request and response messages
- Information about the used transport protocol
- Address information to locate a specific service

As introduced by Cerami [5] and also noted by Pautasso et al.[17], the use of SOAP together with WSDL and some other specifications is also commonly known as "Big" web services – compared to REST which is described in the next section.

2.2 REST

Representational State Transfer (REST) is an architectural style introduced by Roy Thomas Fielding [9] that focuses on resources instead of methods. A resource

```xml
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <SOAP-ENV:Body>

    <ns1:getTemp
    xmlns:ns1="urn:xmethods-Temperature"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

      <zipcode xsi:type="xsd:string">10016</zipcode>

    </ns1:getTemp>

  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Listing 2.1: *Simple SOAP message describing a request for calling a method* `getTemp` *of the web service with the zipcode* `10016` *as a paramter to this method. (from "Web Services Essentials" [5])*

can be any information that can be named, e.g. a document, an image, a collection of other resources, a person, the current weather in a city, etc. Every resource has at least one Uniform Resource Identifier (URI) by which it can be accessed. When a resource is accessed, a representation of that resource may be returned to the caller. The data format of the resource is not specified by REST, it can be any sequence of bytes [9].

REST does not invent new protocols to access resources, instead every access happens through the standard HTTP interface using methods like `GET`, `POST`, `PUT` and `DELETE`. Also a resource is not limited to one representation. Every resource may have multiple representations in different data formats or for different languages, etc. Commonly used representation formats include XML documents, objects in JavaScript Object Notation (JSON) [7] or image files, among others [19].

Another important property of REST is that the server is stateless. The session state is therefore stored entirely on the client and every request needs to contain the whole information that is necessary for the server to understand the request. The disadvantage of this approach is that the network performance may decrease because of repetetive data sent in a series of requests. But as described in "Architectural Styles and the Design of Network-based Software Architectures" [9] the stateless server property improves visibility, reliability, and scalability. Also it simplifies implementation of the server.

Web services complying with the principles of REST described in this section are typically called *RESTful* web services [19].

Chapter 3

Scene Graph Systems

As described in “Service-Oriented Scene Graph Manipulation” [20] a scene graph is typically “. . . a hierarchical collection of nodes expressing logical and spatial relations of 3D objects. Each node may have arbitrary children but only one parent node. The first node of the hierarchy is called root node from which all of the nodes in the scene can be accessed by traversing through the graph. Nodes are typically parts of the geometry of the scene. Other functionality like transformations and groups can also be expressed by nodes.”

Everything affecting a node also affects the node’s children. This way spatial relations can be expressed easily. Consider, for example, a room node with a chair as child node – every transformation applied to the room node also affects the chair node. When the room moves, the chair moves along with it. Figure 3.1 shows a schematic representation of how such a scene graph may look like.

Also scene graphs may be used for high level optimizations like visibility culling. Such optimizations are hard to achieve using a typical state machine renderer, because the renderer is not aware of the whole scene. Using a scene graph in combination with a bounding volume hierarchy, visibility culling can be efficiently performed [23]. Every bounding volume of a node is defined so that it contains all bounding volumes of its child nodes. If a node is not visible when testing for visibility, its child nodes need not to be tested, because they are also not visible then.

An additional feature of many scene graph systems is, that they make it possible to reuse one geometry in different locations to save memory. This feature is called *instancing* and is also often available for other resources like materials, shaders, transformations, etc [20].

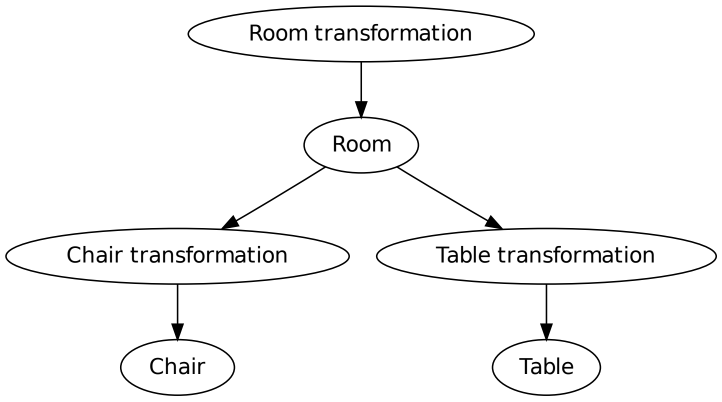


Figure 3.1: *Simple scene graph describing a room with a chair and a table. The transformation of the room also affects the transformations of the chair and the table.*

3.1 OpenSG

OpenSG[1] is an open source C++ scene graph system for high quality real time 3D graphics [18]. OpenSG was born after the Fahrenheit scene graph project of Microsoft and SGI was not continued in 1999. Initially, it was designed and developed by Dirk Reiners, Gerrit Voss and Johannes Behr at IGD Fraunhofer in Darmstadt. OpenSG is based on the cross platform graphics API standard OpenGL. The source code of OpenSG is available under the GNU Lesser General Public License (LGPL) and runs on Windows, Linux and Mac OS.

OpenSG is designed to be a scene graph system that serves as a general basis for a wide variety of applications. The latest version at the time of writing is OpenSG 2.0 with many improvements in usability and supported features.

In "Service-Oriented Scene Graph Manipulation" [20] the following key features of OpenSG are listed:

Performance From the beginning it was designed to make optimal use of the graphics hardware. To obtain optimal performance OpenSG provides optimization algorithms. For example it provides functions to transform a model consisting of triangles to connected triangle stripes, a material sorting optimizer which minimizes the number of state changes for rendering the scene. Also many high level optimizations are included in the OpenSG renderer

[1]http://www.opensg.org

like visibility culling. The processing of scene parts which are behind the viewer or occluded by other parts can be skipped by OpenSG to speed up the rendering of large scenes.

Multi-Threading and Clustering OpenSG allows for multiple threads accessing and manipulating the scene graph in an efficient way [25]. Data structures are organized in multi-thread safe buffers called aspects. Each thread can be assigned to its own aspect meaning it can have its own copy of the buffer. To reduce memory overhead the copy is created only on demand. Remote aspects are copies of the buffers which are send over the network. They are used to synchronize the scene graph within a cluster of multiple machines. In this way the rendering of complex geometry can be handled interactively. Also virtual environments consisting of many screens and machines like tiled displays and CAVEs are implemented with remote aspects.

Dynamically Extensible Not only by making the source code available but also by using highly dynamic and flexible structures OpenSG can easily be extended or adapted to specific needs. Custom render nodes can be added to extend the render capabilities. Reflective interfaces make it possible to integrate arbitrary types of custom data to the whole system. New application-specific data which is not known by the system at compile time can however be used in the internal loaders and writers and synchronized to a cluster.

One concept that is different in OpenSG compared to other scene graph systems is the distinction between nodes and node-cores. While most other scene graph systems have different types of nodes for all kind of functionality (eg. transform nodes, geometry nodes, etc.), OpenSG has exactly one node class. The only purpose of nodes in OpenSG is to describe the hierachy of the scene graph. In OpenSG, the functionality is provided by so called node-cores. Similar to the different node types in other scene graph systems, OpenSG provides different types of node-cores for different functionalities. Every node in an OpenSG scene graph can contain exactly one node-core that defines the functionality of that node. Instancing of node-cores is also possible, so that multiple nodes can share one common node-core.

Chapter 4

Web Services and Scene Graphs

The integration of a web service into an OpenSG application offers runtime access to the scene graph and also makes it possible to modify the contents of the scene graph during runtime without recompilation of the application. Many new ways in interacting with the OpenSG application and the scene graph are possible with these features, some of them are discussed later. Figure 4.1 shows a system overview of the integration of a web service into an OpenSG application that is described now. [20]

As described in “Service-Oriented Scene Graph Manipulation” [20] our web service has the following main features:

- Querying the contents of the scene graph
- Adding and deleting of nodes
- Changing properties of nodes
- Referencing of node data – to use it in other nodes (instancing)
- Download the scene or sub graphs of the scene graph as a single file

4.1 RESTful web service API

As mentioned in “Service-Oriented Scene Graph Manipulation” [20] the API to access the web service is designed in terms of the Representational State Transfer (REST) architectural style as described by R. Fielding [9]. The REST style was choosen because of its simplicity and because the hierarchical nature of a scene graph fits nicely to the resource oriented URI style of RESTful web services.

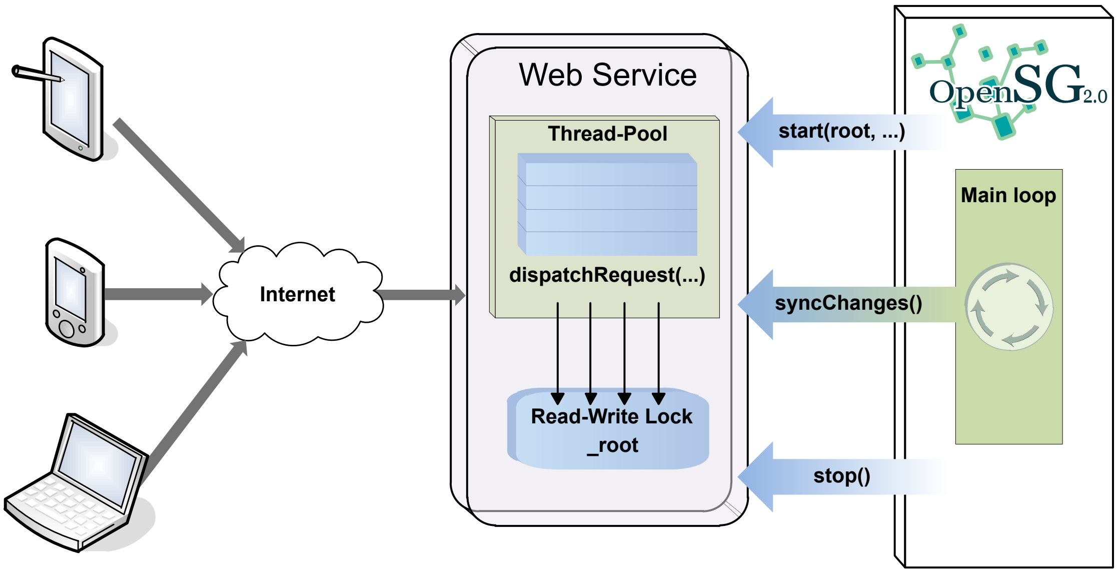


Figure 4.1: *Integrating a service oriented architecture via a web service into the scene graph system OpenSG offers many new possibilities. To export a scene graph of an arbitrary OpenSG application, three method calls to the web service are sufficient. With this minimally invasive change of the application, a closed visualization can be transformed into an open service oriented platform. (Adapted from "Service-Oriented Scene Graph Manipulation" [20])*

Another reason is that RESTful web services use standard HTTP methods for accessing resources. The four most important HTTP methods GET, POST, PUT and DELETE map very well to the most common operations on the scene graph: reading, creating, updating and deleting nodes and values [20].

Because the REST architectural style does not specify how the protocol for accessing the web service must look like, the designer of such a protocol is not limited at all. But this also offers the possibility for badly designed protocols. Because of that, one should follow commonly accepted best practices for designing REST protocols as described in "How to Create a REST Protocol" [11] and "RESTful Web Services" [19].

According to "How to Create a REST Protocol" [11] four questions should be answered when designing a REST protocol (in that order):

1. What are the URIs? (resources)
2. What's the format? (representations)
3. What methods are supported at each URI?

4. What status codes could be returned?

To answer the first question, we need to consider what our web service should be able to do. It should be able to access all nodes of the scene graph – so the first identified resources are the scene graph nodes. Because we also want to access and change the property fields of the scene graph nodes, all fields of those nodes are resources too.

Because the children of a node are part of the node's fields, the whole scene graph can be traversed by only accessing fields, starting at the root node.

The URIs of the web service follow directly from this observation. The main URI of the web service for accessing the scene graph is defined as `/scene`. It provides access to all fields of the root node as defined by it's OpenSG class. By accessing the root node's fields, all other nodes and their fields can be accessed through the web service.

The data interchange and resource representation format used by our web service is the JavaScript Object Notation (JSON)[1]. In "RFC4627: JavaScript Object Notation" [7] the following description of JSON can be found:

> "JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript Programming Language Standard. JSON defines a small set of formatting rules for the portable representation of structured data."

For the defined resources (nodes and fields), three different types of representations are used by our web service: `FieldContainer`, `List` and `Data`.

The three representation types are defined as follows [20]:

Data `Data` resources contain the value of primitive OpenSG data-types like integers, floats, vectors or matrices as a string. Additionally the data-type itself is also specified as the name of its OpenSG class. In the following example the field `travMask` from the `FieldContainer` example below is represented:

```
{
  "content": "Data",
  "type": "UInt32",
  "value": "4294967295"
}
```

[1]http://www.json.org

List `Lists` are ordered collections of resources that are accessible by index. The `List` representation contains the type of its items and a list of locations to the item resources. The following example shows the JSON representation of the `children` field, again from the `FieldContainer` example below:

```
{
  "content": "List",
  "type": "NodePtr",
  "items": [
    "/scene/root/children/0",
    "/scene/root/children/1"
  ]
}
```

FieldContainer `FieldContainers` are resources that itself contain locations of other resources which are accessible by name. Every `FieldContainer` has an unique id and a type, which is the name of its OpenSG class. All sub-resources of the `FieldContainer` are accessible through the `fields` attribute of the JSON object, which contains a mapping from the field name to the location of the corresponding resource. Here is an example of a `FieldContainer` that represents an OpenSG `Node` object located at the root of the scene graph – it contains locations to both the `travMask` and the `children` resources of the above examples:

```
{
  "content": "FieldContainer",
  "id": 292,
  "type": "Node",
  "fields": {
    "attachments": "/scene/root/attachments",
    "travMask": "/scene/root/travMask",
    "core": "/scene/root/core",
    "children": "/scene/root/children"
  }
}
```

This three basic representation types are sufficient to access all OpenSG data structures accessible in the scene graph. For accessing the resources, as mentioned earlier, the four HTTP methods GET, POST, PUT and DELETE are used, but not all methods are allowed for all resources. Also, the method OPTIONS is partially supported, to allow AJAX web applications on different domains to access the web service. As described in the W3C Cross-Origin Resource Sharing working

Status code	*Description*
`200 - OK`	The request has been processed without errors.
`201 - Created`	A new node has successfully been created.
`400 - Bad request`	The request sent from the client has errors (e.g. URI query missing/invalid, content is not a valid JSON object).
`404 - Location not found`	The URI of the request does not point to an valid resource of the web service.
`405 - Method not allowed`	The HTTP method used for the request is not allowed for the resource.
`500 - Internal Server Error`	The web service cannot process the request, because the OpenSG data type does not support the requested operation.

Table 4.1: *Summary of all status codes that the web service may return within the response messages to requests.*

draft[2], web browsers use this method to authenticate cross site requests and web services must respond correctly to the OPTIONS request to allow the cross site request [20].

The last two questions for the design of the REST protocol are now answered together with a description of the semantic of the different HTTP methods when accessing resources of the web service. Also the different status codes of the response messages of the web service are defined and additionally summarized in Table 4.1.

4.1.1 GET

As stated in "Service-Oriented Scene Graph Manipulation" [20], GET requests are used for getting a representation of a resource. The GET method can be used for all valid URIs of the web service. It normally returns one of the three representation types presented above. An important property of GET requests is that they never change any data in the scene graph.

If the GET request is made with just the URI of the resource, one of the three

[2]http://www.w3.org/TR/cors/

JSON representations, depending on the type of the specified resource, is returned. The URI may additionally contain a query with a file extension. In that case the specified resource can be downloaded – including sub-resources – in the specified file format (if supported by OpenSG). The filetype query is only processed for `FieldContainer` resources with the type `Node`, for all other resources the query is ignored.

If the given URI cannot be found by the web service in the scene graph, an error message with the HTTP status code `404 - Location not found` is returned by the web service. Else, if there is no error, the response has the status code `200 - OK` [20].

The following examples show the usage of GET [20]:

GET /scene/root/core Get the JSON representation of the core of the root node

GET /scene/root?filetype=osb Download the root node (and all its children) as OpenSG-Binary file

4.1.2 POST

The method POST is either used to append new nodes to resources of type `List` or to update the value of `Data` resources. Operations like appending new nodes are only allowed using the POST method, because according to the HTTP/1.1 specification [8], the POST method is the only non-idempotent method. This means that only for POST it is allowed that multiple identical requests lead to different results (as when appending to a list).

POST is also used for updating the value of `Data` resources because the POST method can handle big amounts of upload data which may occur when updating for example textures or vertex arrays.

When appending a new node to a `List` resource, the required `type` URI query specifies the OpenSG type of the new node. In case the specified type is a OpenSG node, an empty `Group` core is created and automatically added to the node, to prevent nodes without a core. Otherwise, if the specified type is a OpenSG node-core, an empty node is created to which a core of the specified type is added. After appending the new node, the response to the request is returned, which contains a JSON string with the location of the new node and additionally also a HTTP header field `Location` with the full URI to the newly created node.

For the other use case of POST, updating the value of a `Data` resource, the new value of the resource must be sent as the entity body of the HTTP request in the JSON string format. After the value of the resource is successfully updated, the JSON representation of the updated resource is returned – the same as a GET to that resource, but already with the updated value.

Because the POST method is used for multiple purposes and its operations depend on user defined input there are multiple possible (error-) status codes that may be returned. In case a new node is successfully appended to a `List` resource, the returned response has the status code `201 - Created`. If updating the value of a `Data` resource was successful, the status code `200 - OK` is returned. If either the `type` URI query for appending a new node is missing or the given type is unknown, or if the entity body for updating a value of a `Data` resource is not a valid JSON string the response has the status code `400 - Bad request`. In case the location specified by the URI is not found in the scene graph, the status code `404 - Location not found` is returned. Finally, if the URI was found but does not point to a `List` or `Data` resource, the response has the status code `405 - Method not allowed` [20].

Examples for POST requests as listed in “Service-Oriented Scene Graph Manipulation” [20]:

POST /scene/root/children?type=Geometry Appends a new node with a `Geometry` core to the children of the root node.

POST /scene/root/children/1/core/matrix With `"1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1"` as the entity body, sets the matrix of the second child's core to the identity matrix.

4.1.3 PUT

Requests with the method PUT can be used to create new resources, similar to POST requests. The difference to POST requests is, that instead of appending a new resource to a list a PUT requests replaces an already existing resource or inserts a new resource at a specific index in a list.

Replacing an existing `FieldContainer` resource works either with a `type` URI query that specifies the type of the new `FieldContainer` to create – similar to POST – or with an `location` URI query with the location of another `FieldContainer` resource. When using the `location` URI query, the specified location gets referenced from the URI of the PUT request. This makes it possible to share geometry, materials, etc. between multiple nodes, which is also known as instancing.

As POST requests, a PUT request may return one of several status codes indicating either success or different errors. If the request is processed without error, the response has the status code `200 - OK`. The status code `400 - Bad request` is returned if the `type` resprectivly `location` URI query is not valid. This is the case if either the specified location is not found in the scene graph or the given type is unknown. Also the same status code `400 - Bad request` is used if the index specified in the URI is not valid when inserting a new resource into a list. Like requests with other methods, a PUT request returns the `404 - Location not found` status code if the called URI is not found in the scene graph. Another error case is if the called URI does not point to a `FieldContainer` resource or an item of a `List` resource. In that case, the status code `405 - Method not allowed` is returned by the web service. Inserting an item at a specific index is not supported by all OpenSG containers. If the web service detects that the item cannot be inserted at a specific index, a response with the status code `500 - Internal Server Error` is returned [20].

In "Service-Oriented Scene Graph Manipulation" [20] the following examples for PUT requests are shown:

PUT /scene/root/core?type=Geometry Replaces the core of the root node with a new `Geometry` core.

PUT /scene/root/children/0/core?location=/scene/root/core Sets the core of the first child to the root's core. Both nodes share the same core after this request.

PUT /scene/root/core/properties/8?type=GeoVec2fProperty Creates a new `GeoVec2fProperty` and assigns it to the ninth item in the root's core properties.

4.1.4 DELETE

The last method used by our web service to manipulate the scene graph, DELETE, is used to delete items of a `List` resource. In case the DELETE request is made to the URI of the `List` resource itself, all items of that `List` are deleted. If the request is instead made directly to one of the items in a `List` resource, only that item is removed from the `List`.

If the deletion of either the single item or all items of the `List` resource is successful, the response has the status code `200 - OK`. Similar to the other methods, a response with the status code `404 - Location not found` is returned if the called URI is not found in the scene graph. Also because DELETE is only allowed for `List` resources or items of such a `List` resource, the status code `405 - Method not allowed` is returned if the URI points to any other type of resource [20].

Examples for DELETE requests [20]:

DELETE /scene/root/children/0 Removes the first child of the root node.

DELETE /scene/root/children Deletes all children from the root node of the scene graph.

4.2 Implementation

This section describes the actual integration of a web service into OpenSG based applications. Different possible designs for the implementation and libraries for implementing them are discussed. Finally the current implementation is presented in detail.

4.2.1 Design alternatives

Integrating a web service into an application can be achieved in many different ways. For the specific case of integrating a web service in an OpenSG application, the only requirement is that the web service must somehow have access to the OpenSG scene graph data. Because the data of the OpenSG scene graph is only available to the process of the OpenSG application itself, at least a part of the web service must run within the OpenSG application's process to provide access to the data.

Attributes of OpenSG data structures are usually accessed by calling a method of the object. But in the case of the web service, that means that all possible OpenSG data structures must be known in advance, including the attributes and methods they support. For situations like this, OpenSG provides a string based reflection mechanism. Using the built-in reflection mechanism of OpenSG it is possible to query the fields and manipulate the field data of any OpenSG data structure that supports that mechanism at runtime without knowing the type at compile time. This also makes it possible to access fields of user defined node types that are not part of OpenSG, if they support the reflection mechanism.

One idea for integrating the web service into OpenSG based applications was to integrate some scripting language into the OpenSG application and provide access to the generic, string based reflection API of OpenSG to that scripting language. This would not only allow a web service to access the data of the OpenSG scene graph, but would provide a generic scripting environment for every OpenSG based application. Also, because the web service would be implemented in the scripting language, the web service could easily be changed later without recompilation of the OpenSG application by simply changing the script files, for example for adding new features.

A good canditate for an embeddable scripting language to implement the web service in is Python[3]. Python is a dynamic, object oriented programming language with a very clear and easy to read syntax. The main Python implementation CPython is available for all major platforms like Linux, Mac OS X and Windows. Additionally Python has been ported to many other operating systems, devices and runtime environments like the Java Virtual Machine. Many Linux distributions ship Python in their standard configuration. What makes Python particularly suited for implementing the web service is its rich standard library (it comes with "batteries included") and the wide variety of available web frameworks/servers for Python such as

- CherryPy (http://www.cherrypy.org)
- Django (http://www.djangoproject.com)
- Pylons (http://pylonshq.com)
- TurboGears (http://www.turbogears.org)

just to name a few. The Python standard library even has a basic web server built in and also has support for processing JSON data.

An alternative to Python is Mono[4], the open-source implementation of Microsoft's .NET Framework. The .NET Framework is a software platform consisting of a runtime environment, a standard library, APIs and Services. Unlike the .NET Framework, which is only available for Windows, the Mono runtime is available for pretty every major operating system. According to the Mono homepage, supported platforms of Mono include Linux, Microsoft Windows, Mac OS X, BSD and even game consoles like Nintendo Wii or Sony PlayStation 3. Mono is also designed to be easily embeddable into other applications and also has a very good support for web services. Because Mono implements most of the .NET Framework it also has a great standard library with many things built in. Integrating the Mono runtime additionally allows for many different languages to be used for implementing scripts, like C#, VisualBasic.NET, Java or even Python.

The second approach to embedd a web service into an OpenSG based application

[3]http://www.python.org
[4]http://mono-project.com

is to just implement the web service in C++. The disadvantage of that is of course that you can not use higher level language features like garbage collection and have to recompile the application for every change to the web service. On the other hand, C++ provides a better performance than embedded scripting languages.

There are some big C/C++ web frameworks available like Apache Axis2[5] or the gSOAP Toolkit[6], but they primarly target SOAP web services. Also they are probably overkill for the use case of an embedded web service, because they are more all-in-one solutions that contain everything needed to build (SOAP) web services in C/C++.

Instead, small embedded web server libraries like Mongoose[7] or GNU libmicrohttpd[8] are better suited for the task of integrating a RESTful web service into any OpenSG application. They don't provide unneeded features like SOAP support or XML processing, instead they just provide an API to start a web server and to handle incoming HTTP requests – and that's everything needed to implement a RESTful web service.

Mongoose is a small, easy to use web server that can be either used standalone or embedded within another application. It is available for all major operating systems like Linux, Mac OS X and Windows. The implementation of Mongoose is written in C and consists of only one header and one source file, what makes it very easy to integrate it in other applications. Beside support for C/C++ Mongoose also has support for additional programming languages like Python and C#.

The GNU libmicrohttpd is also a small web server library similar to Mongoose. It is not available as a standalone web server but instead only designed to be integrated in other applications. The library is also written in C and is available for most platforms (BSD, Linux, Mac OS X, Windows). For many Linux distributions libmicrohttpd is already available packaged in their repositories. The implementation of libmicrohttpd is HTTP 1.1 compliant, has support for incremental processing of POST data and can be used with three different, flexible threading models.

All in all the two libraries Mongoose and libmicrohttpd offer comparable features and seem equally good for implementing a RESTful web service in C++.

Both approaches for integrating a web service into OpenSG based applications have their advantages and disadvantages. While the scripting approach offers the greatest possible flexibility, it probably needs more work to implement it, because

[5]http://ws.apache.org/axis2/c/

[6]http://gsoap2.sourceforge.net/

[7]http://code.google.com/p/mongoose/

[8]http://www.gnu.org/software/libmicrohttpd/

first the interface from OpenSG to the scripting language has to be implemented in addition to the web service itself. But with the scripting interface in place, the implementation of the web service would be easier because it can be done in an higher level language. On the other side, implementing the web service directly in C++ has the advantage that processing the requests runs with the speed of C++. Also there is no level of indirection when accessing the data of the scene graph. But a C++ implementation of the web service would not offer the additional possibilities of the scripting interface and also would not be very flexible. Changing the web service always would require to recompile the OpenSG application.

The selected approach for implementing the web service and the reasons for choosing it are now described together with implementation details in the next section.

4.2.2 Current implementation

In the current implementation the web service is implemented directly in C++. This approach was choosen because the additional possibilities of the scripting interface are not really needed. If needed, the OpenSG application can still be scripted by accessing the web service from a scripting language. Also the web service should be fast at processing requests because it may be used in different real time scenarios.

One design decision made when implementing the web service in C++ was to only provide a minimal but powerful interface (see API in section 4.1). So the API of the web service provides only a very low level access to the scene graph and does not offer high level features like for example to load and insert a new 3D model with a single request. If such features are needed through a web service, an additional standalone web service can easily be created that offers such features on top of the existing web service. Figure 4.2 shows the idea of multiple layered web services that provide additional convenient features on top of the low level RESTful web service API.

For the implementation of the web service, the GNU libmicrohttpd library is used. The biggest advantage over Mongoose is the flexible threading model that eases the integration into the – possible threaded – OpenSG application. Also the fact that it is already available in many Linux distributions is an advantage, because that makes using the web service a lot easier on Linux.

The disadvantage of the C++ approach, that the OpenSG application always has to be rebuild if the web service needs to be changed, is mainly a disadvantage

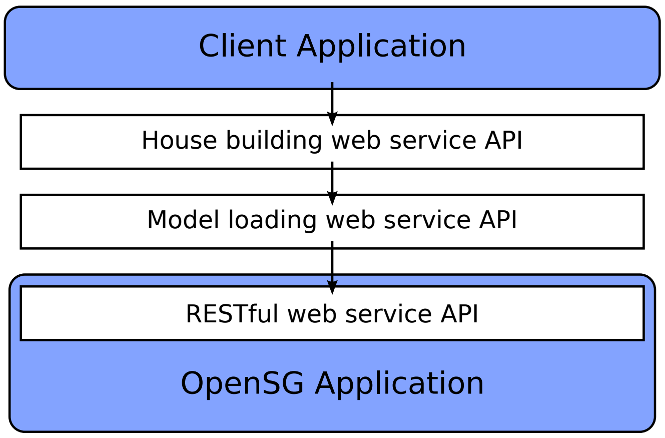


Figure 4.2: *Multiple web services can be stacked on top of the low level API to provide an higher level interface to the client application. Additional web services use the low level interface to provide new features. Additionally, low level requests to those web services may just be forwarded to the underlaying web service.*

during the development of the web service only. Once the web service is stable only few changes will be made to it, so the problem does not exist anymore then.

The actual implementation of the web service has a clear structure. The libmicrohttpd web server is started with a threading model that uses a thread pool. The thread pool is used because all threads that change OpenSG data structures must be registered at OpenSG before and using the thread pool limits the threads that must be registered to a few. For every request to the web service, one of the threads in the thread pool starts and processes the request. Depending on the prefix of the accessed URI (`/scene` or `/actions`) and on the used HTTP method (GET, PUT, POST, DELETE) the appropriate action is performed. Because multiple threads may simulataneously change the OpenSG scene graph, all access to the scene graph is protected by a Read-Write lock that guarantees that multiple threads can read at the same time but if one thread writes, no other thread can access the scene graph.

4.3 Integration overview

As noted in “Service-Oriented Scene Graph Manipulation” [20] the implemented web service is designed to be easily integrable into every OpenSG application with only a few changes to the source code. The public interface of the class

`OSGWebservice`, which is the main class for using the web service, consists of only three methods that are needed to get the web service running in any OpenSG based application. This three methods are now presented in detail.

```
bool start(FieldContainerMTRecPtr root,
           UInt32 aspect,
           UInt32 port);
```

The `start` method as shown above is used, as the name suggests, for starting the web service. Usually this method gets called once at the beginning of the program, just after creation of the initial OpenSG datastructures like the root node. Everything accessible through the FieldContainer given as the paramter `root` is exported through the web service. The object specified as `root` is accessible trough the web service at the location `/scene`. Typically this is the root node of the OpenSG scene graph, but it may also be any other FieldContainer like for example an OpenSG `Viewport`. In case it is a `Viewport` some additional usefull properties of the OpenSG application are accessible through the web service in addition to the normal scene graph data, like the camera of the specified `Viewport`.

For multi-threaded operations on the scene graph, OpenSG provides so called *aspects*. Multiple threads can operate independent from each other on different aspects of the scene graph without interfering each other. At regular intervalls the aspects can be synchronized and every thread then sees the changes of the other threads. The `aspect` parameter of the `start` method specifies on which apsect the web service operates. This parameter should most of the time be set to some unused aspect which is then exlusively used for the changes of the web service [20]. Per default this paramter is set to the aspect `0`, which is the main aspect of an OpenSG application. If the application itself also modifies the scene graph, this value must be changed!

The last parameter `port` specifies the network port on which the web services listens for incoming HTTP connections. The default value for the `port` is `8080` (which may already be used by some other web server) [20].

```
void syncChanges();
```

This method synchronizes the changes of the main application and the changes made through the web service and should therefore be called at a regular interval. Typically this method is called within the render function of the main OpenSG application every frame [20].

It is important that the changelist of the main OpenSG aspect doesn't get cleared until this method gets called. Otherwise, no changes from the main OpenSG aspect can be synchronized to the web service. But after calling this method, and possibly synchronizing other threads, the changelist of the main OpenSG aspect should be cleared, to prevent synchronizing the same changes again in the next call to this method.

```
void stop();
```

Using this method, the web service can finally be stopped when it is not needed anymore [20].

4.4 Extensibility

So far, the described web service provides access to everything in the scene graph of an OpenSG based application. But for some use cases, the users of the web service, the application programmers, may want to provide additional functionality for their application through the web service. To allow such special use cases, an additional API is designed, which allows the application programmers to map function calls of their application to user defined URIs accessible through the web service. For this to work, the application programmer just needs to register a callback function at the web service together with a specific user defined URI. The web service stores the URI and the associated callback funciton and every time the URI gets accessed through the web service, the stored function gets called [20].

The interface to this additional API consists of two methods to register and unregister user defined methods which are now presented.

```
void registerAction(std::string name,
                    WebserviceAction callback);
```

This method registers a new callback for a new user defined URI. The paramter `name` specifies the URI for which the callback is registered. All registered URIs are accessible under the location `/actions`. The `callback` parameter is a function pointer to the application programmers function that gets called every time the URI is accessed. Functions to be registered as a callback for an URI must have the following signature:

```
std::string function_name(FieldContainerMTRecPtr web_root,
                          std::string name,
                          std::string method,
                          std::string input);
```

As the first paramter `web_root`, the callback functions gets the pointer to the root FieldContainer that is exported by the web service. All changes made in the callback to the scene graph must be performed through this FieldContainer, because the callback is executed in the thread and aspect of the web service.

The `name` paramter contains the name under which the callback was registered at the web service. This can be used to use the same callback function for different URIs and decide based on the name which action to perform.

To have the ability to use different HTTP methods, the third paramter `method` contains the HTTP method used to call the URI that triggered the execution of the callback function. Using this paramter, different operations for the HTTP methods like GET, PUT, POST or DELETE can be implemented in the callback.

Finally, the entity body of the request – the "POST data" – is given as the `input` paramter to the callback. This data is not processed in any ways by the web service, only directly passed through to the callback function. Using this data, the callback can recieve additional arguments to process the request.

The return value of the callback function is used by the web service to create the response to the request. The response is created with just the string returned by the web service, without further processing by the web service. The content type of the response is `text/plain` and the status code is always `200 - OK`.

```
void removeAction(std::string name);
```

This method presented above can be used to remove previously registered callbacks.

The only parameter `name` specifies the name of the callback that should be removed.

Using this callback mechanism an application programmer can for example map a function like `buildHouse(...)` to the URI `/actions/build_house`. Whenever the URI `/actions/build_house` gets accessed, the specified callback function gets called by the web service.

This features allows the application programmer to easily extend the functionality of the web service specifically for their applications's needs [20].

4.5 Threading issues

As described before, the web service may use mutliple threads to handle incoming HTTP requests. The number of threads can be changed by the application programmer when creating the web service. All of the threads work on a single OpenSG aspect, which is specified when starting the web service, so they need to synchronize their access to the data structures of the aspect. To synchronize all threads accessing the same aspect, the access to the shared aspect is protected by a Read-Write lock as already noted before. This ensures that multiple threads can read, but if one thread writes to the aspect, no other thread can access the aspect at that time. Every subsequent access to the aspect waits until the current write operation finishes.

But even with this synchronization in place, it is possible that threads overwrite each others changes to the aspect before the changes get synchronized to the main thread of the application. To ensure reliability of the web service, only one writing change is therefore allowed for every call to `syncChanges()`. Requests that only read data from the web service are not affected by this limitation [20].

This limitation does also not apply if the web service is started with only one worker thread that handles requests, because in this case no changes can be overwritten by another thread. As a result, a web service started with only one thread may be faster at processing many write requests (it does not need to wait for `syncChanges()` to get called), but it will be slower when many parallel read requests are made.

Due to some problems with the used libraries on the Windows platform – libmicrohttpd and pthreads-win32[9], which is used for threading on Windows – the maximum number of threads in the thread pool is currently limited to one when running on Windows. But as described before, this is not really a problem because the special case of only one worker thread is optimized and can perform better than multiple threads in some use cases. Also the limitation may be easily removed if the problems are solved.

[9]http://sourceware.org/pthreads-win32/

Chapter 5

Advantages of Web-Based Linking

The web service approach as presented in the previous chapter offers a variety of possibilities. Most importantly, the presented approach realizes a simple integration of any application based on the scene graph system OpenSG into already existing applications. It is of course also possible to realize such an integration via static or dynamic linking, but a web-based linking via web services additionally also enables a simple integration across different platforms and languages. The common denominator of web services are HTTP reuests, which are understood by almost all programming languages on nearly every platform. Because of this, the integration of a visualization based on OpenSG into a Java application is no problem anymore. This approach just offers the possibility to use the right tool respectivly programming language to get the job done.

Also, the web-based integration of the OpenSG scene graph system offers new and simple interaction possibilities. Because the camera used for the OpenSG viewport may be a part of the scene graph that is exported via the web service, this camera can also be edited and modified via a webbrowser, which makes new human-computer interactions possible. One example for this technique would be to control and explore a CAVE environment with a smart-phone [20].

Another important feature is that web services can also be used and controlled by other web services. This makes it possible, for example, to create independent web services that provide additional or higher level features like presented in 4.2.2. Also possible are scenarios where such additional web services allow different users to control different parts of the scene graph. One web service may also link to multiple instances of via web service exported OpenSG scene graphs and can then control all instances simultaneously.

As noted in “Service-Oriented Scene Graph Manipulation” [20] such things were possible before, but only with specialized, adopted applications. With the presented approach this is possible a lot easier now: only a few lines of code are needed to start the web server, the web server takes any node of the scene graph and automatically exports everything that is accessible from that node.

Chapter 6

Example Applications

6.1 Interior Design Visualization

As stated in [20] a plug-in to the open-source interior design application SweetHome3D[1] (see Fig. 6.1) has been developed to test and demonstrate the web service.

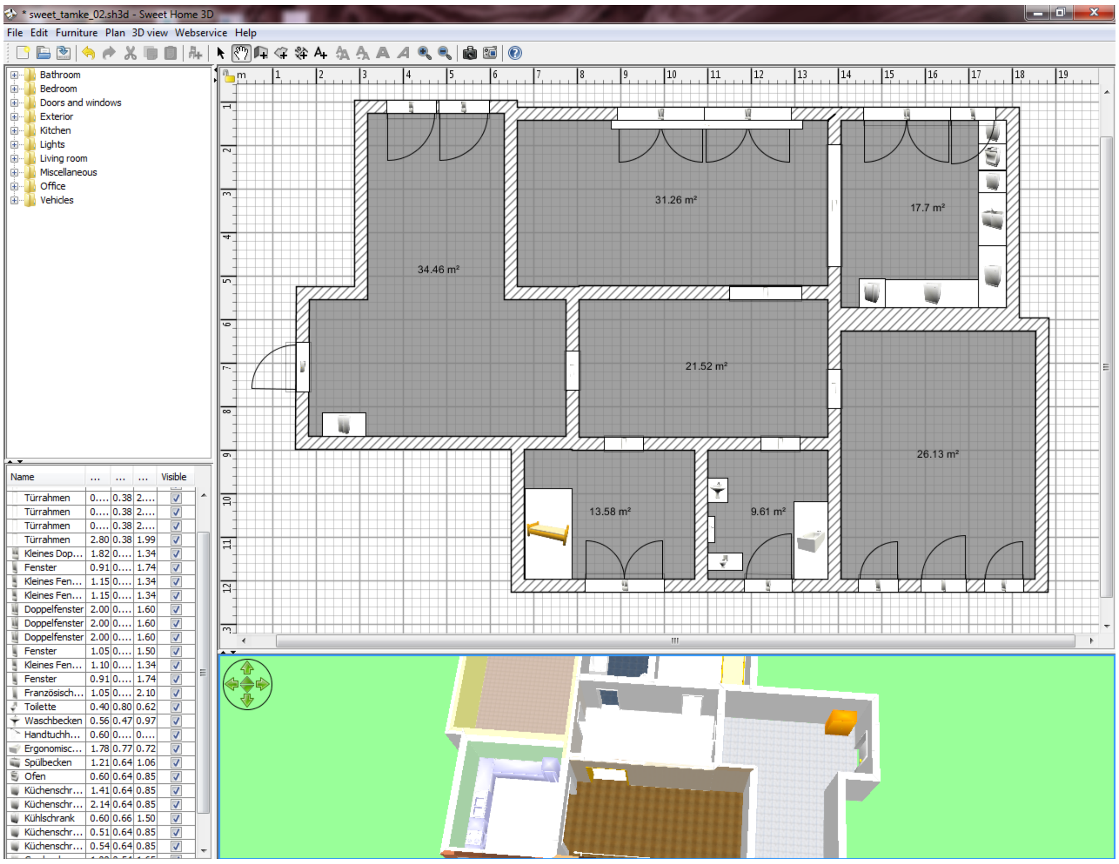
SweetHome3D is implemented in Java and therefore runs on a variety of platforms. The main features are the ability to draw a 2D house plan and to place arbitrary 3D objects like furniture or plants on that plan. Included in the user interface of SweetHome3D is also a 3D preview of the geometry created from the house plan and the placed objects. A plug-in API is available which makes it possible to track changes made to the 2D house plan and to get data like position and size of walls, materials, etc. Using this plug-in API the web service plug-in now described was developed [20].

6.1.1 Integration of the web service plug-in

The web service plug-in integrates itself into SweetHome3D with a common menu entry through wich several items for starting and controlling the web service plugin are available.

Before connecting to an OpenSG application, several settings can be made like to specify a location in the scene graph of the OpenSG application under which all contents of SweetHome3D will be added. It is also possible to set a transformation

[1]http://www.sweethome3d.eu



Name	...	...	...	Visible
Türrahmen	0....	0.38	2....	✓
Türrahmen	0....	0.38	2....	✓
Türrahmen	0....	0.38	2....	✓
Türrahmen	2.80	0.38	1.99	✓
Kleines Dop...	1.82	0....	1.34	✓
Fenster	0.91	0....	1.74	✓
Kleines Fen...	1.15	0....	1.34	✓
Kleines Fen...	1.15	0....	1.34	✓
Doppelfenster	2.00	0....	1.60	✓
Doppelfenster	2.00	0....	1.60	✓
Doppelfenster	2.00	0....	1.60	✓
Fenster	1.05	0....	1.50	✓
Kleines Fen...	1.10	0....	1.34	✓
Fenster	0.91	0....	1.74	✓
Französisch...	1.05	0....	2.10	✓
Toilette	0.40	0.80	0.62	✓
Waschbecken	0.56	0.47	0.97	✓
Handtuchh...	0.60	0....	0....	✓
Ergonomisc...	1.78	0.77	0.72	✓
Spülbecken	1.21	0.64	1.06	✓
Ofen	0.60	0.64	0.85	✓
Küchenschr...	1.41	0.64	0.85	✓
Küchenschr...	2.14	0.64	0.85	✓
Kühlschrank	0.60	0.66	1.50	✓
Küchenschr...	0.51	0.64	0.85	✓
Küchenschr...	0.54	0.64	0.85	✓

Figure 6.1: *The SweetHome3D interior design application connected to an OpenSG application via the web service plug-in.*

matrix that can be used to translate, rotate or scale the SweetHome3D content in the OpenSG application. This is important if there are any differences in the coordinate systems used by SweetHome3D and the OpenSG application. All these settings can also be specified via a configuration file that is read by the plug-in.

Finally the plug-in can be connected to an OpenSG application which has the web service integrated by just specifying the IP adress and port of the OpenSG application (which also can be stored in the configuration file). Once connected, the plug-in sets up the environment for the SweetHome3D content in the OpenSG application. This includes setting up a `Transform` node with the specified transformation matrix and creating `Group` nodes for every type of object that is transferred from SweetHome3D, like rooms, walls and furniture, just below the `Transform` node. After everything is set up, all changes made to the 2D house plan in SweetHome3D are transferred by the plug-in in real time through the web service to the OpenSG application where they are added to the scene graph.

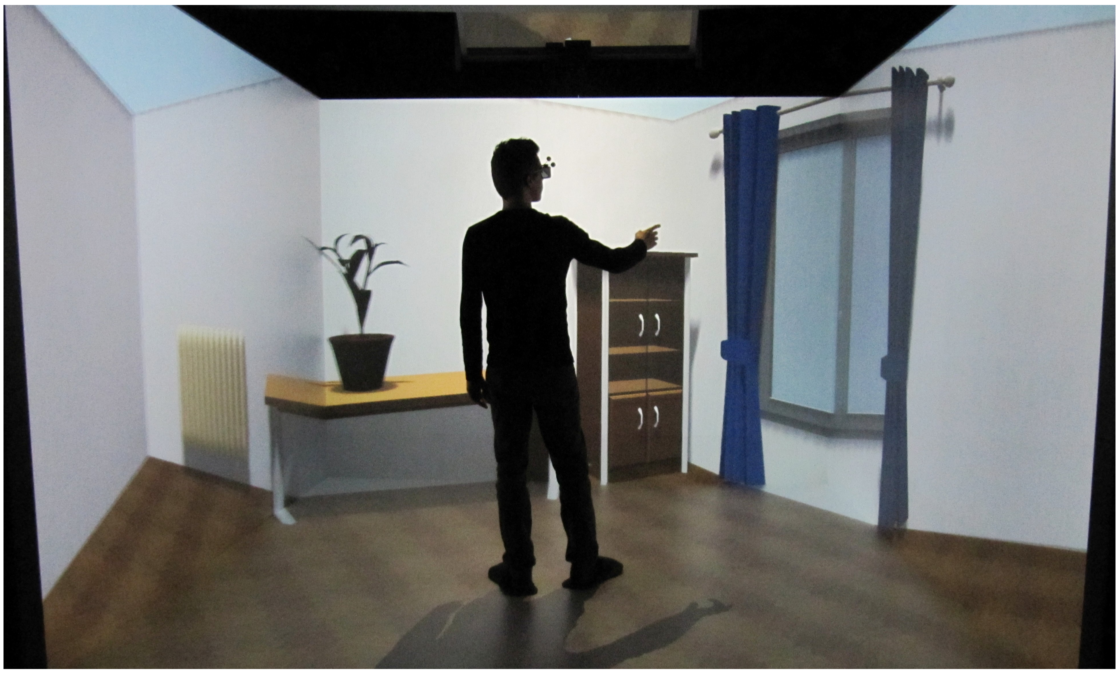


Figure 6.2: *The SweetHome3D output is transferred with our web service to an OpenSG CAVE application which lets the user walk through a 3D representation of the house plan [20].*

The only requirements for the OpenSG application to display the 3D representation of the SweetHome3D house plan are to start the web service and to export one node in which the plug-in creates the house geometry.

All features of SweetHome3D like the geometry, the colors, the materials and textures as well as the imported geometry representing the interior are transferred correctly and displayed the same way as planned in SweetHome3D. Additionally, displaying the scene using OpenSG allows the graphical representation to be much improved over the simple 3D preview in SweetHome3D. The preview in SweetHome3D does not feature any advanced graphical features like shadows, etc. which can be easily included in an OpenSG application displaying the 3D representation of the house plan.

The biggest advantage of displaying the scene in an OpenSG application over the the included 3D preview is the flexibility of OpenSG. Using OpenSG makes it possible to display the 3D representation of the house plan in a virtual environment like a CAVE or on a tiled display as shown in Figures 6.2 and 6.3. At the same time, the house plan can still be modified in its semantically enriched form of the SweetHome3D 2D house plan editor [20].

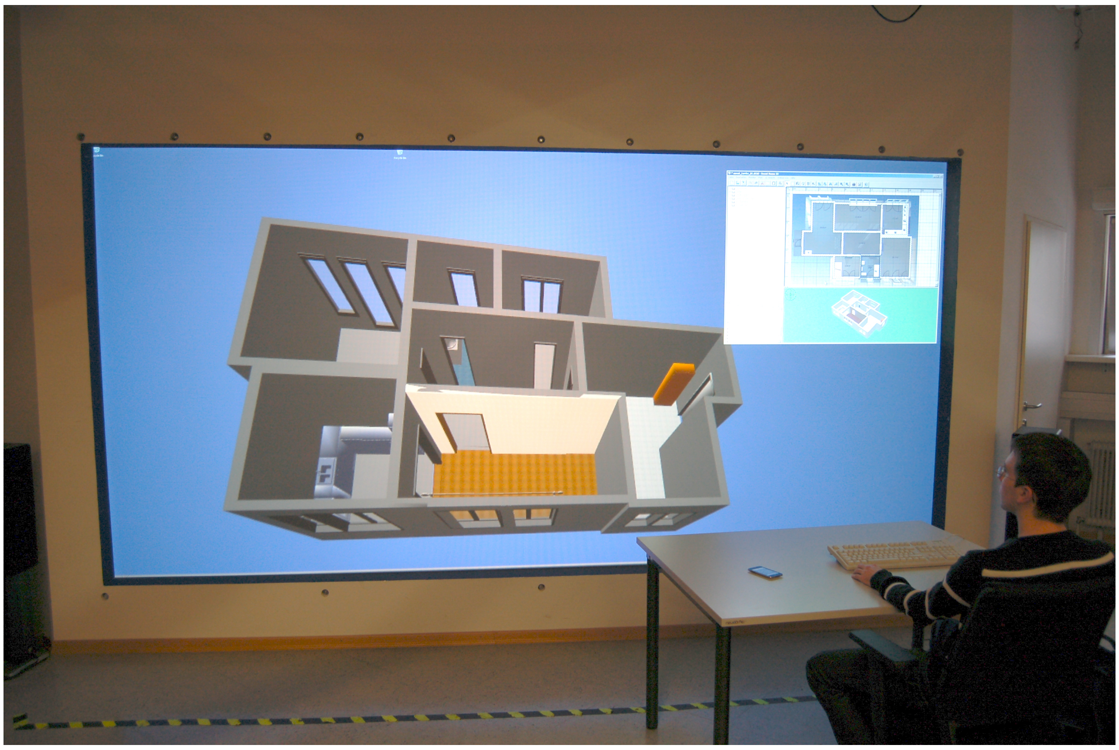


Figure 6.3: *A plug-in to the interior design application SweetHome3D transfers all changes made to the 2D house plan through our web service to an OpenSG application which displays a 3D representation of the house plan on a big tiled display [20].*

6.1.2 Processing changes

Two mechanisms of the plug-in API of SweetHome3D are used by the plug-in to keep track of the changes made to the 2D house plan. First, a `CollectionListener` is added to the collection of every type the web service plug-in is interessted in. The observed types are currently rooms, walls and furniture. With this listeners in place, the web service gets informed whenever an object of the specific type is added or deleted. When a new object of a specific type is created, a wrapper object for that specific object and type is created wich holds the second type of listener used by the web service plug-in. The second type of listener is the `PropertyChangeListener` that gets added to every newly created object of the observed types. Whenever a property of an observed object is changed, the wrapper object that holds the listener for the observed object gets notified about the change and can then process it.

On creation of such a wrapper object, the necessary nodes in the OpenSG scene graph are created and initialized to represent the object in its current state. Whenever a property changes, the representation in the OpenSG scene graph is updated according to the change. The number of nodes and how they are updated differs slightly between the three types of objects.

Room objects

Every room object consists of one `Group` node that in turn contains two `Geometry` nodes, one for the floor and one for the ceiling. Every `Geometry` node has properties for vertices, normals and texture coordinates. Additionally, every `Geometry` node has its own material.

Whenever the points of the polygon that makes up the room changes, the wrapper object for that specific room gets notified. From the associated room object the two dimensional (x, y) points can be queried, that define the room on the 2D house plan. From that points, and general properties of the house like wall height, the 3D geometry for the floor and the ceiling can be calculated. The algorithm for calculating the geometry is basically the same as the one used by the SweetHome3D 3D preview, to be as compatible as possible. All the algorithm does is checking for self intersection of the polygon – and split it if that is the case – and assigning normals and texture coordinates. All vertex data is then transferred via the web service to the OpenSG scene graph, where it is rendered as a single `GL_POLYGON`.

The room wrapper object also gets notified if either the material of the floor or that of the ceiling changes. In that case, either a color has been set as the new material or a texture. If the new material is a color, that color is simply set as the ambient and diffuse color of the material of the `Geometry` node for the floor respectivly the ceiling. In case the new material is a texture, the pixels of the texture are transferred as raw RGBA values via the web service and stored as the pixels of the image property of the `Geometry` node's material. Additionally, properties like width and height of the texture are set in the OpenSG image object.

SweetHome3D also allows the user to change visibility of the floor respectivly the ceiling. This changes are also transferred correctly to the OpenSG scene graph by the wrapper object for the room. To set the visibility in the OpenSG scene graph, the `travMask` property of the `Geometry` node for either the floor or the ceiling is set accordingly, so that the node is not traversed if it is not visible.

Wall objects

Similar to room objects, every wall object consists of a `Group` node that holds two `Geometry` nodes. The `Geometry` nodes correspond to the left respectivly the right side of a wall object in SweetHome3D, or in other words to the inside respectivly the outside of a wall as seen from within a room. The wall geometry is splitted in half because that makes it possible to set different materials for each side.

The geometry of a wall must be updated if any of these properties of the wall changes:

- start location
- end location
- height
- height at end
- thickness

Every time one of these properties changes, the two geometries for the wall (left and right side) get rebuild using the properties listed above. Again, the algorithm for creating the wall geometries is taken mostly from the renderer of the 3D preview in SweetHome3D. The biggest challenge when creating the wall geometries is, that doors and windows intersecting the wall must be "cut" out of the wall geometry. The algorithm does that by normally creating the parts of the wall to the left and to the right of the door or window and additionally adding polygons above the door/window and below a window. Also, normals and texture coordinates are calculated and together with the vertices transferred to the OpenSG scene graph where multiple `GL_POLYGON`s – depending on the number of doors and windows in the wall – are stored for rendering.

The material of a wall may change for the left and the right side independently. Similar to room objects, the new material may either be a solid color or a texture image. The properties of the new material are transferred to the OpenSG scene graph the same way as they are for room objects.

Visibility is also handled similar to room objects, but for walls the visibility of the left and the right side cannot be set seperatly. Because of that, the `travMask` property is only changed on the common `Group` node of the wall according to the visibility of the whole wall.

Furniture objects

A furniture object in SweetHome3D can be every 3D model that can be imported. SweetHome3D itself supports multiple file formats such as OBJ, DAE, 3DS or LWS. But because SweetHome3D only provides access to the file stream of the loaded 3D model, the parsing and processing of the file has to be done by the plug-in. Because of that, the web service plug-in currently only supports loading OBJ files, but that is not a big problem because all models included in SweetHome3D are in the OBJ format. In the OpenSG scene graph, every furniture object consists of a `Transform` node which in turn contains one `Geometry` node for every material used in the imported 3D model.

As the geometry of an imported 3D model never changes, only the `Transform` node is changed when the furniture objects is moved, rotated or scaled on the 2D house plan. Additionally, SweetHome3D allows the user to mirror the 3D model, which is also handled by changing the `Transform` node.

Altough the imported 3D model has at least one (default) material set, it is possible to override the imported materials and set a custom solid color or a texture image as the material for the furniture object. In this case, all the `Geometry` nodes for the different materials are changed to either the same solid color for the whole model or to the specified texture.

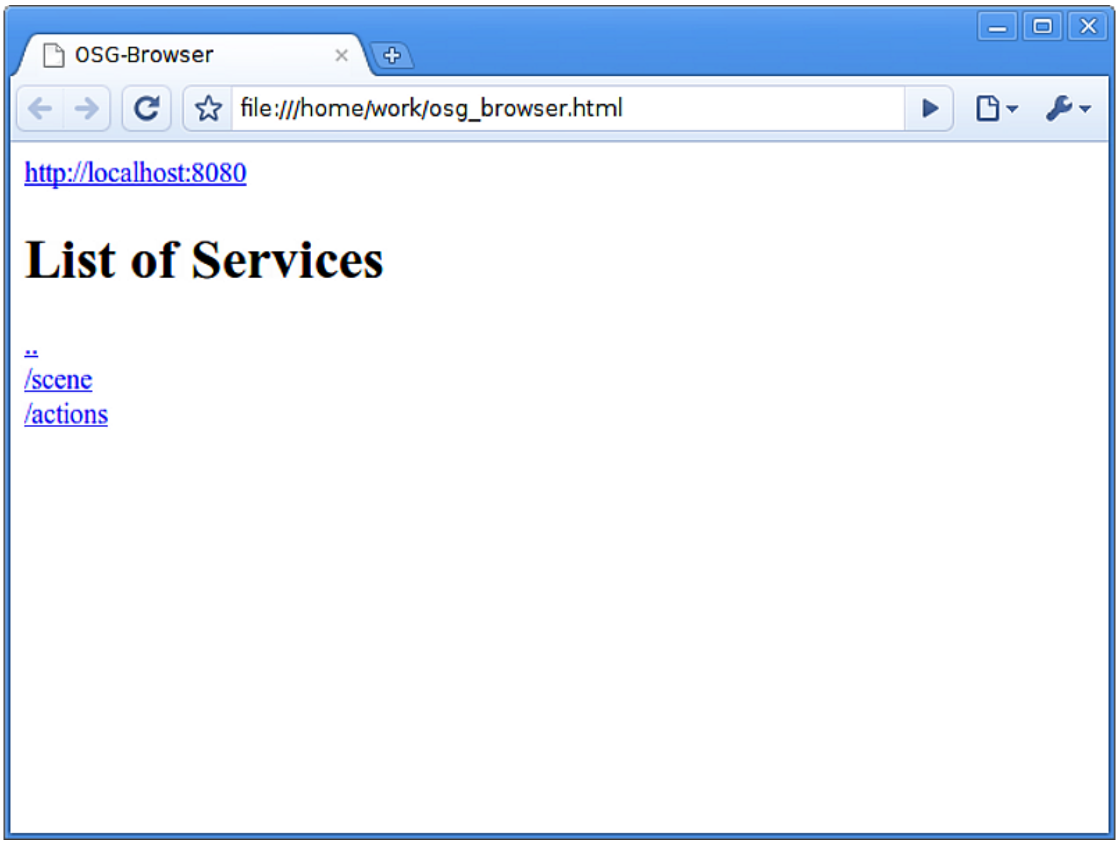
Visibility is handled similar to the other object types, by changing the `travMask` property of the `Transform` node representing the furniture object in the scene graph.

6.2 Browser-Based Web Service Interface

For debugging purposes an additional tool that makes use of the web service of an OpenSG application was created. It is a single web page that makes it possible to browse through the exported scene graph of the OpenSG application using any modern webbrowser. Besides browsing through the scene graph it also allows for adding, deleting and editing nodes and values of the scene graph. As shown in Figure 6.4, besides the scene graph also the actions defined in the web service are available from the browser-based interface.

The web page is entirely written in the Hypertext Markup Language (HTML) and JavaScript and connects to the web service of the OpenSG application using standard features of modern webbrowsers like `XMLHTTPRequest` and the ability to process JSON data. The source code of the web page can be found in Appendix A.

Developers of clients to the web service of an OpenSG application can use the described web page both as a reference that shows how to access the web service and as a debugging tool for their own web service client. Using the browser-based interface to the web service allows the developer to inspect changes made to the



http://localhost:8080

List of Services

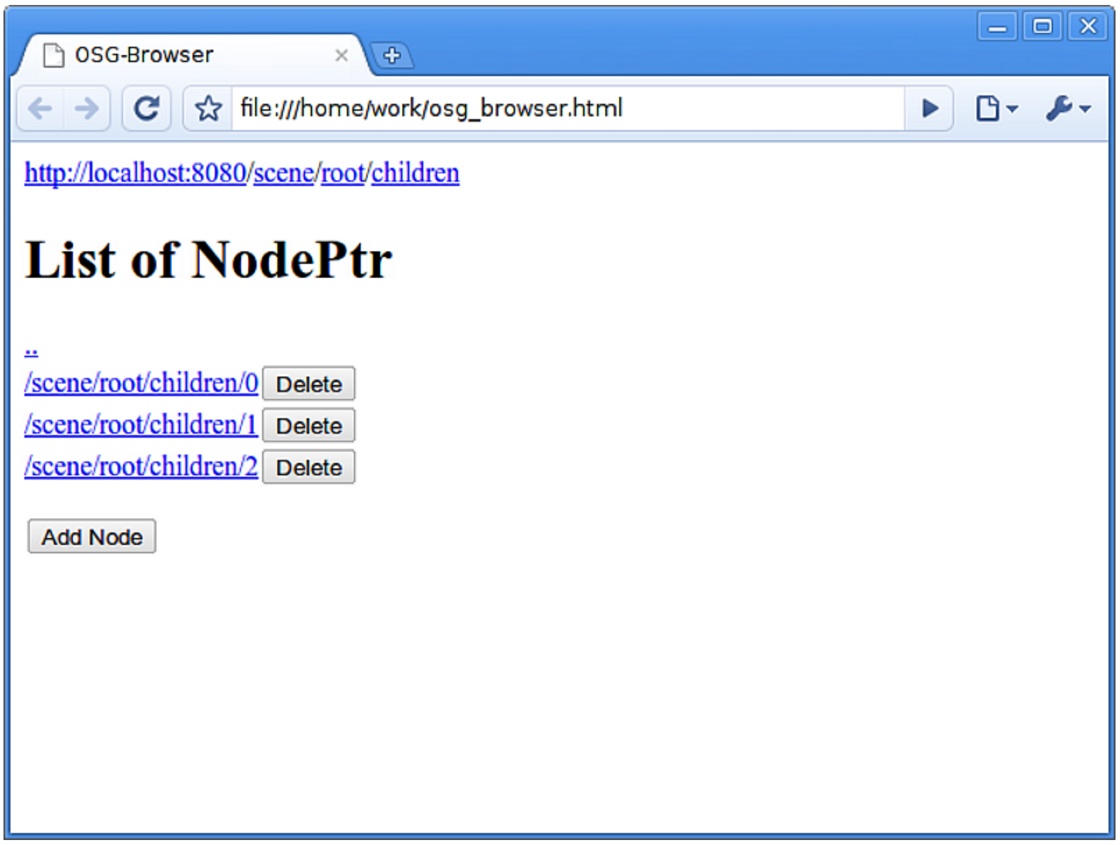
..
/scene
/actions

Figure 6.4: *The main screen of the browser-based web service interface, offering links to either the scene graph or the actions offered through the web service.*

web service from other clients, which can be used to find errors easily. Figure 6.5, for example, shows how all child nodes of a specific node in the scene graph can be listed.

Also it is possible to try things out using the browser-based interface before writing own code that accesses the web service. For example, it might not be obvious to the developer of an web service client which fields are accessible through the web service or which data types they have. Because the described web page lists all accessible fields and their data types, it is a very good reference for developers to look up things about the scene graph and its nodes, as shown in Figure 6.6.

The use of the presented web page is not limited to debugging tasks, instead it is also useful for many common tasks during the modeling-visualization-cycle. Often the developer of an OpenSG application just wants to test a new camera position or wants to move an object in the displayed scene. Usually this is only possible if the OpenSG application has specific code that offers this kind of manipulations



http://localhost:8080/scene/root/children

List of NodePtr

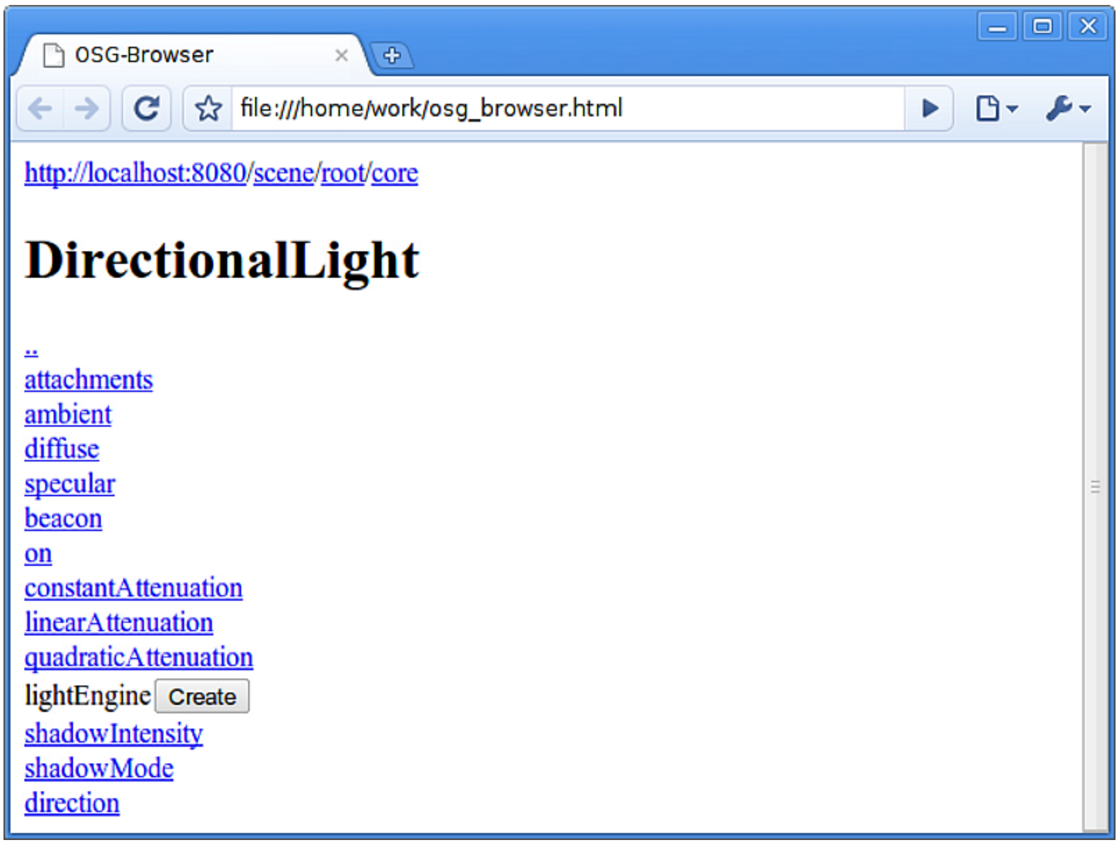
..
/scene/root/children/0 Delete
/scene/root/children/1 Delete
/scene/root/children/2 Delete

Add Node

Figure 6.5: *Screenshot of the browser-based interface to the web service listing all child nodes of the root node of the scene graph. Children can be added and deleted using the provided buttons.*

through the user interface of the application. But often the user interface of the application is too limited to make such manipulations possible, for example in a VR setting. Also it is very time consuming to program all these interaction possibilities into the OpenSG application. The alternative is often to stop the application, change the desired objects in the scene and then restart the application to display the changes. This is of course not feasible, as it takes a very long time for only simple changes.

Using the presented web page, such changes can be easily made through every device that can run a webbrowser. All properties of all the objects in the scene graph can be manipulated through the web page while the application is running, so common tasks like moving or deleting objects or changing the camera during runtime are no problem anymore. Figure 6.7 shows how a transformation matrix can be changed to move an object in the scene or to change the camera used to



http://localhost:8080/scene/root/core

DirectionalLight

..
attachments
ambient
diffuse
specular
beacon
on
constantAttenuation
linearAttenuation
quadraticAttenuation
lightEngine Create
shadowIntensity
shadowMode
direction

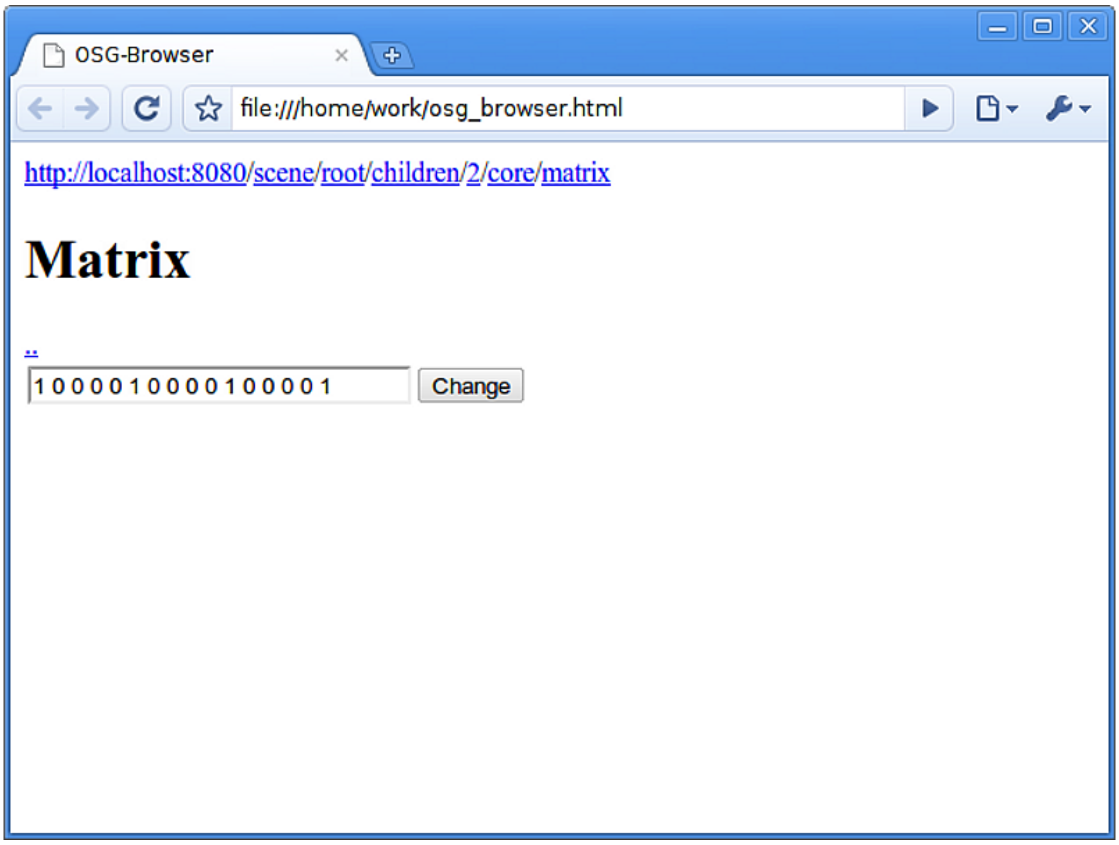
Figure 6.6: *All accessible fields of every node in the scene graph can be inspected using the browser-based web service interface. Fields that are currently* `null` *can be created through the webbrowser.*

display the scene.

6.3 Additional Usage Scenarios

As shown in section 6.1 it is possible to use the web service to provide additional 3D views of a scene by writing a small plugin. This is not limited to modelling software or 3D software in general. The same principle could also be used to provide a 3D view of the data of some non-3D application, or even to provide a visualization for a software that has no graphical user interface at all. For example the calculations of a computation process running on a server could be visualized in real time on another computer while the computation is still in progress.

Also, as many 3D modelling applications provide ways to write plugins for them,



http://localhost:8080/scene/root/children/2/core/matrix

Matrix

..

1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1

Figure 6.7: *Fields containing primitive data types like numbers, vectors or matrices can be changed through the browser-based web service interface. The screenshot shows how a transformation matrix can be changed using the webbrowser.*

the same princible as described for SweetHome3D can also be applied to all kinds of other modelling software. Writing a similar plugin as the one described in section 6.1 for a general purpose modelling software like Maya or 3ds Max would for example make it possible to display the currently modelled 3D model in real time in the target environment while still working on the model. This way, the final appearance of the 3D model in the target application with all materials, shaders, etc. applied can be checked in real time. Time consuming intermediate steps like exporting the 3D model from the modelling application and importing it in the visualization application can be skipped.

This concept could even be taken further as the web service also has the ability to read all data contained in the scene graph. A plugin to a modelling application could therefore also load geometry from the scene graph of an OpenSG application into the modelling application and display it there. In the modelling application, the geometry imported from the scene graph can then be modified in a convinient way. All changes made to the geometry in the modelling application can then be transferred back into the visualization application through the web service. This way, the geometry displayed in an OpenSG application could be changed in a convinient way while the application keeps running. No manual importing or exporting is needed. Also the visualization application does not need any code that allows manipulation of its scene – the only requirement is the integration of the web service, which usually takes only a couple lines of source code.

Chapter 7

Comparison & Conclusion

7.1 Web-based Architectures

There is currently an ongoing discussion in the SOA world regarding "REST vs. SOAP". An overview and comparison of RESTful and "big" web services can be found in "Restful web services vs. "big" web services: making the right architectural decision." [17].

REST and SOAP services should not be seen as different implementation alternatives for the same solution, as noted in "Web Services Platform Architecture" [26]:

> ...As a rule of thumb, REST is preferable in problem domains that are query intense or that require exchange of large grain chunks of data. SOA in general and Web service technology ... in particular is preferable in areas that require asynchrony and various qualities of services ...

While the strengths of REST are its simplicity and the ease of implementation, SOAP may provide more reliability. Because REST uses only standard web technologies, no additional libraries are needed for implementing a RESTful web service or a client to such a web service. Furthermore, as REST does not specifiy the data interchange format or the structure of messages, the developer of a RESTful web service has a high degree of freedom for the actual implementation of the service. For SOAP web services on the other hand, the data interchange format and the message structure are standardized and thus are the same for all SOAP compliant web services. As can be seen, depending on the requirements, both approaches have their own advantages and disadvantages and the decision between REST and SOAP has to be made on a case by case basis.

7.2 Distributed Graphics

A render cluster using the Chromium[1] software is described by Bacu et al. [1]. The described solution renders a scene graph on multiple clients by transferring the low level OpenGL commands of the network. The render results are streamed back to the server using a video streaming format. The Chromium software makes it possible to distribute OpenGL based applications without any code changes, but transfering all OpenGL commands gererally results in a much greater amount of data to be transfered over the network compared to using OpenSG. Expecially for a CAVE setup a customized application is esential [20].

7.3 Integrated Approaches

A framework using web services to combine multiple input sources into one 3D portal application is proposed by Zhang and Gračanin [27].

Hagedorn and Döllner describe using web services in combination with a 3D city visualization [13]. The web service is used to provide high quality rendered images independent of the client devices used. The described framework is specialized on using high-level geoinformation services without cluster support.

Behr et al. [2] describe a similar combination of web service and scene graph for the instant reality framework[2]. Their solution also uses OpenSG as the rendering back end and X3D for the scene description. Unlike the solution presented in this thesis, they use SOAP instead of REST. The instant reality framework can also be used in CAVE environments and offer a great render performance. One important difference is, that it is not possible to integrate the instant reality framwork into existing applications on the C++ level.

7.4 Conclusion

In this thesis an integrated approach of a web-based scene graph application interface is presented. The most important properties of the presented solution compared to previous work are [20]:

[1]http://chromium.sourceforge.net
[2]http://www.instant-reality.org

multi-threaded All components of our system are multi-threaded. The scene graph system OpenSG supports multiple threads and the web service can handle requests in parallel.

multi-user capable The implemented synchronization mechanism ensures a consistent scene graph, even if multiple users send several HTTP requests at once.

platform independent OpenSG as well as the web service implementation (libmicrohttpd) are platform independent; i.e. all major platforms (MS Windows, Linux, Mac OSX) are supported.

minimally invasive Any OpenSG application can be upgraded to a web-based service-oriented application by adding a few lines of code. Due to the clean design of OpenSG only the web server's start-up method and its synchronization routine have to be inserted into the existing application. Therefore, the number of changes to an existing application normally involves less than ten lines of code. This minimally invasive modification transforms an OpenSG application into a web server.

adaptable / accessible On the client side only minimal adaptions are needed. The effort required to build a client to a RESTful service – the technique used to transform OpenSG into a service-oriented architecture – is very small as developers can begin testing such services from an ordinary web browser. Due to web support on all platforms and in all languages (Java, C/C++, ...), the integration of an OpenSG-based visualization is no problem anymore. Wrapper and interface code to overcome differences in language and communication is not needed.

All in all, the solution presented in this thesis enriches web-based interaction and visualization. As stated in "Service-Oriented Scene Graph Manipulation" [20] the described combination of OpenSG and an integrated web server has beneficial effects on service-oriented scene graph systems, on the integration of immersive visualization environments into existing applications, and on browser-based and web-based user interfaces in virtual and mixed reality.

Appendix A

Source Code for Browser-Based Web Service Client

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" >
  <head>
    <title>OSG-Browser</title>
    <script type="text/javascript">

      // Try to get a XMLHTTPRequest object
      // for all common browsers
      function getHTTPObject() {
        if (typeof XMLHttpRequest != 'undefined') {
          return new XMLHttpRequest();
        }
        try {
          return new ActiveXObject("Msxml2.XMLHTTP");
        } catch (e1) {
          try {
            return new ActiveXObject("Microsoft.XMLHTTP");
          } catch (e2) {}
        }
        return false;
      }

      var http = null;         // Global XMLHTTPRequest object
      var server = "";         // Current server address/IP
      var working = false;     // Is there an ongoing request?
      var current_url = "";    // Last loaded URL
```

```
//Forward declaration of some functions
var urlOpener, valueChanger, fcCreator, loadResult;


// Writes the header of the page consisting of a
// clickable URL, a heading describing the content
// and a "one-level-up" link
// caption: String that gets displayed in the heading
function writeHeader(caption) {
  // Get header DIV and clear it
  var header_div = document.getElementById("header");
  header_div.innerHTML = "";
  // Create link to root of server
  var server_link = document.createElement("a");
  server_link.href = "javascript:void(0)";
  server_link.onclick = urlOpener("/");
  server_link.innerHTML = server;
  header_div.appendChild(server_link);
  // Split current URL and create link to every
  // "directory" level
  var path = "";
  var parts = current_url.split("/");
  for (var i in parts) {
    if (parts[i].length > 0) {
      var part_link = document.createElement("a");
      part_link.href = "javascript:void(0)";
      part_link.onclick = urlOpener(path + "/" + parts[i]);
      part_link.innerHTML = parts[i];
      header_div.appendChild(document.createTextNode("/"));
      header_div.appendChild(part_link);
      path += "/" + parts[i];
    }
  }
  // Create heading with caption string
  var header = document.createElement("h1");
  header.innerHTML = caption;
  header_div.appendChild(header);
  var up = document.createElement("a");
  up.href = "javascript:void(0)";
  up.onclick = urlOpener(current_url.slice(0,
      current_url.lastIndexOf("/")));
  //Create "one-level-up" link
  up.innerHTML = "..";
  header_div.appendChild(up);
  header_div.appendChild(document.createElement("br"));
}
```

```
// Callback that displays all fields of the returned
// JSON object when the request finishes
function displayFields() {
  // Check if request has finished
  if (http.readyState == 4) {
    // Request has finished
    working = false;
    var field_div = document.getElementById("fields");
    field_div.innerHTML = "";
    if (http.status == 0) {
      // The request is not valid
      return;
    }
    // Parse JSON object and process it depending on
    // the "content" field that specifies the type
    var response = JSON.parse(http.responseText);
    if (response.content == "FieldContainer") {
      writeHeader(response.type);
      for (var field in response.fields) {
        if (response.fields[field]) {
          // Field is valid, create a link to it
          var field_link = document.createElement("a");
          field_link.href = "javascript:void(0)";
          field_link.onclick = urlOpener(response.fields[field]);
          field_link.innerHTML = field;
          field_div.appendChild(field_link);
        } else {
          // Field is NULL, add button to create it
          var fieldname = document.createTextNode(field);
          var new_fc = document.createElement("input");
          new_fc.type = "button";
          new_fc.value = "Create";
          new_fc.onclick = fcCreator(current_url + "/" + field,
                                     "PUT", displayFields);
          field_div.appendChild(fieldname);
          field_div.appendChild(new_fc);
        }
        field_div.appendChild(document.createElement("br"));
      }
    } else if (response.content == "List") {
      writeHeader("List of " + response.type);
      for (var i in response.items) {
        if (response.items[i]) {
          // List item is valid, add link to it
          var link = document.createElement("a");
          link.href = "javascript:void(0)";
          link.onclick = urlOpener(response.items[i]);
          link.innerHTML = response.items[i];
          field_div.appendChild(link);
        } else {
          // List item is NULL, add only text
          field_div.appendChild(document.createTextNode("null"));
        }
        if (current_url.length > 1 && current_url != "/actions") {
          // If current list item is not an action, add a
          // button to delete that item from the list
          delete_btn = document.createElement("input");
          delete_btn.type = "button";
          delete_btn.value = "Delete";
          delete_btn.onclick = deleter(response.items[i]);
          field_div.appendChild(delete_btn);
        }
        field_div.appendChild(document.createElement("br"));
      }
      if (current_url.length > 1 && current_url != "/actions") {
        // If current list doesn't contain actions, add buttons
        // to create list items
        var buttons = document.createElement("p");
        var add_fc = document.createElement("input");
        add_fc.type = "button";
        add_fc.value = "Add Node";
        add_fc.onclick = fcCreator(current_url, "POST",
                                   loadResult);
        buttons.appendChild(add_fc);
        if (!current_url.match(/children$/)) {
          // The children list doesn't allow adding by index
          // For other lists, add a button to add item by index
          var set_fc = document.createElement("input");
          set_fc.type = "button";
          set_fc.value = "Set new Node at index";
          set_fc.onclick = fcCreator(current_url, "PUT",
                                     displayFields, true);
          buttons.appendChild(set_fc);
        }
        field_div.appendChild(buttons);
      }
    } else if (response.content == "Data") {
      // For data values, add an input box to allow
      // changing the value
      writeHeader(response.type);
      var input_box = document.createElement("input");
      input_box.type = "text";
      input_box.value = response.value;
      input_box.size = response.value.length;
      var change_value = document.createElement("input");
      change_value.type = "button";
      change_value.value = "Change";
      change_value.onclick = valueChanger(input_box);
```

```
      field_div.appendChild(input_box);
      field_div.appendChild(change_value);
    }
  }
}


// Start a new request that loads and displays the
// fields of the specified url
// url: URL from which to load the fields
function fetchFields(url) {
  if (!working) {
    working = true;
    current_url = url;
    http = getHTTPObject();
    http.open("GET", server + url, true);
    http.onreadystatechange = displayFields;
    http.send(null);
  }
}


// Callback that loads the fields of the URL that is
// specified in the response from the request
// when the request is finished
function loadResult() {
  if (http.readyState == 4) {
    working = false;
    new_url = JSON.parse(http.responseText);
    fetchFields(new_url);
  }
}


// Returns a function that loads the fields of a given
// URL with a given HTTP method
// url: URL which is loaded by the returned function
// method: HTTP method used for loading the URL (optional)
function urlOpener(url, method) {
  var url_to_open = url;
  var http_method = typeof(method) != 'undefined' ?
                    method : "GET";
  var opener = function() {
    fetchFields(url_to_open, http_method);
  };
  return opener;
}
```

```
// Returns a function that changes the value of
// the current field to the current value of
// the given input-box
// input: input-box which will contain the new value
function valueChanger(input) {
  var input_box = input;
  var change_func = function() {
    if (!working) {
      working = true;
      http = getHTTPObject();
      http.open("POST", server + current_url, true);
      http.onreadystatechange = displayFields;
      http.send('"' + input_box.value + '"');
    }
  };
  return change_func;
}


// Returns a function that creates a new FieldContainer
// (for null fields, or as list items)
// url: URL where the FieldContainer should be created
// method: HTTP method used for creating the
//         FieldContainer (PUT, POST)
// display_func: Callback function that displays the
//               the results of the request
// ask_index: Should the returned function ask for an
//            list index at which the FieldContainer
//            gets created (optional)
function fcCreator(url, method, display_func, ask_index) {
  var post_url = url;
  var http_method = method;
  var data_display_func = display_func;
  var ask_for_index = typeof(ask_index) != 'undefined' ?
                      ask_index : false;
  var add_func = function() {
    var index;
    if (ask_for_index) {
      index = prompt("Index:");
    }
    var node_type = prompt("Node-Type:");
    if (!working) {
      working = true;
      current_url = post_url;
      http = getHTTPObject();
      full_url = server + current_url;
      if (ask_for_index) {
        full_url += "/" + index;
        current_url += "/" + index;
```

```
          }
          full_url += "?type=" + node_type;
          http.open(http_method, full_url, true);
          http.onreadystatechange = data_display_func;
          http.send(null);
        }
      };
      return add_func;
    }


    // Returns a function that deletes a list item
    // url: URL of the list item to delete
    function deleter(url) {
      var delete_url = url;
      var delete_func = function() {
        if (!working) {
          working = true;
          http = getHTTPObject();
          http.open("DELETE", server + delete_url, true);
          http.onreadystatechange = displayFields;
          http.send(null);
        }
      };
      return delete_func;
    }


    // Asks for the server to connect to and loads the
    // root resource of that server
    // Gets called when this page is initially loaded
    function init() {
      server = prompt("OSGWebservice URL:",
                      "http://localhost:8080");
      fetchFields("/");
    }

  </script>
</head>
<!--
  Ask for server address/IP and load root resource when
  body is initially loaded
-->
<body onload="init()">
  <!--
    Container for the header
    (clickable URL, heading, "one-level-up"-link)
  -->
  <div id="header"></div>
```

```
    <!--
      Container for the actual fields of the current URL
    -->
    <div id="fields"></div>
  </body>
</html>
```

Listing A.1: *Source code of the single HTML document of the browser based client to the web service described in section 6.2.*

Bibliography

[1] Victor Bacu, Lucian Muresan, and Dorian Gorgan. Cluster Based Modeling and Remote Visualization of Virtual Geographical Space. *Proceedings of the 2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 10:416–421, 2008.

[2] Johannes Behr, Patrick Dähne, and Marcus Roth. Utilizing X3D for immersive environments. *Proceedings of the ninth international conference on 3D Web technology*, 9:71–78, 2004.

[3] David Booth, Hugo Haas, Francis Mccabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture. *World Wide Web Consortium*, 20040211:1–98, 2004.

[4] N Brown and C. Kindel. Distributed Component Object Model Protocol – DCOM/1.0, 1998.

[5] Ethan Cerami. *Web Services Essentials*. O'Reilly, 2002.

[6] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web service definition language (wsdl). Technical Report NOTE-wsdl-20010315, World Wide Web Consortium, March 2001.

[7] Douglas Crockford. RFC4627: JavaScript Object Notation, 2006.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC Editor, 1999.

[9] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Univeristy of California, Irvine, 2000.

[10] M. Golub, D. Jakobović, and I. Janeš. An overview of distributed programming techniques.

[11] Joe Gregorio. How to Create a REST Protocol. http://www.xml.com/pub/a/2004/12/01/restful-web.html, 2004.

[12] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. Soap version 1.2 part 1: Messaging framework. W3C Recommendation, June 2003.

[13] Benjamin Hagedorn and Jürgen Döllner. High-level web service for 3D building information visualization and analysis. *Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, 15:8:1–8, 2007.

[14] Nicolai M. Josuttis. *SOA in Practice: The Art of Distributed System Design.* O'Reilly, Beijing, 2007.

[15] C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F Brown, and Rebekah Metz. Reference Model for Service Oriented Architecture 1.0. *OASIS Standard*, 2006.

[16] Object Management Group. Common Object Request Broker Architecture: Core Specification.

[17] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big" web services: making the right architectural decision. *Proceeding of the 17th international conference on World Wide Web*, 17:805–814, 2008.

[18] Dirk Reiners, Gerrit Voss, and Johannes Behr. OpenSG: Basic concepts. *Proceedings of OpenSG Symposium 2002*, 1:1–7, 2002.

[19] Leonard Richardson and Sam Ruby. *RESTful Web Services.* O'Reilly, 2007.

[20] Andreas Schiefer, René Berndt, Torsten Ullrich, Volker Settgast, and Dieter W. Fellner. Service-Oriented Scene Graph Manipulation. 2010.

[21] Thomas Schiffer, Andreas Schiefer, René Berndt, Torsten Ullrich, Volker Settgast, and Dieter W. Fellner. Enlightened by the Web. 2010.

[22] R. Srinivasan. Remote Procedure Call Protocol Specification Version 2, 1995.

[23] Dirk Staneker. *Hardware-assisted Occlusion Culling for Scene Graph Systems.* PhD thesis, Eberhard-Karls-Universität Tübingen, 2005.

[24] Sun Microsystems Inc. Java Remote Method Invocation Specification. http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf, 2004.

[25] Gerrit Voß, Johannes Behr, Dirk Reiners, and Marcus Roth. A multi-thread safe foundation for scene graphs and its extension to clusters. *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, 4:33–37, 2002.

[26] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, march 2005.

[27] Xiaoyu Zhang and Denis Gracanin. Streaming web services for 3D portal applications. *Proceedings of the 13th international symposium on 3D web technology*, 13:23–26, 2008.