TU Bergakademie Freiberg

Faculty for Mathematics and Computer Science

Summer Semester 2015

German Title: Web-basierte Komposition von 3D-Szenen für deren serverseitige Weiterverarbeitung im Roundtrip3D Projekt

Supervisor: Prof. Jung

Matthias Lenk

Bachelor Thesis

Web-based 3D-scene composition for further server-sided processing in the Roundtrip3D project

Name:	Danny Arnold
Matriculation Number:	52315
Major:	Applied Computer Science
Email:	danny.arnold@student.tu-freiberg.de
Date:	August 13, 2015

Eidesstattliche Erklärung

Ich, **Danny Arnold**, versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen Hilfsmittel als die angegebenen benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Diese Versicherung bezieht sich auch auf die bildlichen Darstellungen.

Freiberg, August 13, 2015 _____

Acknowledgment

Contents

Contents **vi**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Scope . . . 2

2 Basics and Related Work **5**
2.1 Scene-Graph . . . 5
2.1.1 Culling . . . 5
2.1.2 Transformations . . . 6
2.1.3 Reuse . . . 6
2.2 X3D . . . 6
2.3 x3dom . . . 6
2.4 SSIML . . . 8
2.5 Roundtrip 3D . . . 8
2.6 Related Work . . . 9
2.6.1 3D Meteor . . . 9
2.6.2 Blender Plugin . . . 10
2.6.3 Gizmos . . . 10
2.6.4 Component Editor . . . 16
2.6.5 Real-Time Collaborative Scientific WebGL Visualization with Web Sockets . . . 16
2.6.6 ParaViewWeb . . . 17

3 Concept **18**
3.1 Server . . . 18
3.1.1 Node.js . . . 18
3.1.2 Koa . . . 19
3.2 Client . . . 19
3.2.1 Synchronization Process . . . 20
Terminology . . . 20
Data Binding . . . 26
3.3 Communication . . . 29

4 Implementation **32**
4.1 Server 32
4.2 Client 32
4.2.1 angular 32
Bootstrapping 32
Modularisation and Dependency Injection 32
Views 33
4.2.2 react 34
4.2.3 Synchronization Process 35

5 Results **38**
5.1 Why Scegratoo is the best since sliced Bread 38

List of Tables **39**

List of Figures **40**

List of Listings **41**

Bibliography **43**

Glossary **46**

1 Introduction

The goal of the thesis is to create a 3 Dimensional (3D) editor, using web technologies, which enables its users to post-process X3D scenes. These scenes are the product of a tool that was implemented as part of the DFG research project Roundtrip3D [26].

1.1 Motivation

As part of the Roundtrip3D project, a round-trip framework (hereafter referred to as *R3D*) was developed. This framework also includes a graphical editor for SSIML models, to describe 3D applications. Then Roundtrip3D (R3D) can be used to generate boilerplate code for multiple programming languages, such as JavaScript, Java or C++, and an X3D file describing the scene. The X3D file may contain references to other X3D files containing the actual 3D data (e.g. a car and its corresponding tires), hereafter called *inlines*. These files are created by exporting objects from a 3D computer graphics software (e.g. Blender, Maya or 3DS Max).
The problem that arises is that each object is usually in the center of its own coordinate system. So they need to be translated, rotated and maybe even scaled, to result in the desired scene (e.g. a car where the tires are in the places they belong to and not in the center of the chassis). The structure is mostly generated. Attribute values of respective nodes, such as transformation nodes, need to be adjusted in order to compose the overall 3D scene.

This can be achieved by adjusting *translate*, *rotate* and *scale* properties to arrange 3D objects. To exacerbate this problem further, the orientation the 3D artists chose for its object may be unknown, if there is no convention for 3D modeling. ~~Usually their is a convention about origin of coordinate system, scaling and orientation~~ (no there isn't, still not sure what to put here). However, we cannot assume that these conventions are always met. The main problem is, that 3D transformations, such as translation, orientation and scale of single 3D geometries, need to be adjusted. So far, there is no graphical tool that meets both of these requirements:

- User friendly and straight forward composition functionalities for X3D scenes and

- preservation of (generated) information, such as node names or comments (necessary to merge the changed files back into the source model).

Figures 1 - 4 demonstrate multiple orientatins a 3D model of a wheel can have. 1 to 3 are common orientations, since it is disputable which of these could be considered be the norm. But if one depends on art from 3rd parties, the orientation and position could be completely arbitrary like in figure 4

These properties could be added and adjusted via any text editor by opening the generated X3D file, but the resulting work-flow is not user-friendly. The following list explains the work-flow using just a text editor.

1. Model the 3D application, including 3D the scene structure.
2. Generate the 3D scene and the application code.
3. Run the application and evaluate the scene and think about what objects need to go where and whether they need to be scaled.
4. Type random translation, rotation and scale values into the transform nodes.
5. Run the application again and evaluate whether the transformations are correct (since one does not know anything about the orientation of the object).
6. Go back to 4. until all objects are placed correctly.

Tools like Maya or Blender could also be used for this, but their import and export filter discard important meta-data that is necessary for the round-trip transformation. This is what SceGraToo is meant to be. SceGraToo addresses both of these issues. It allows for loading the root X3D file and changing all transformations, containing the inline nodes, using mouse interactions. For fine grained control SceGraToo also contains a tree view that allows the user to input exact attributes for translations, rotations and scale.

1.2 Scope

This thesis addresses two issues:

1. Allowing for composing generated X3D scenes (with focus on usability). E.g., besides a 3D view, a tree view for fine grained editing should not only visualize the 3D scene's structure, but also allow for entering concrete coordinate values.

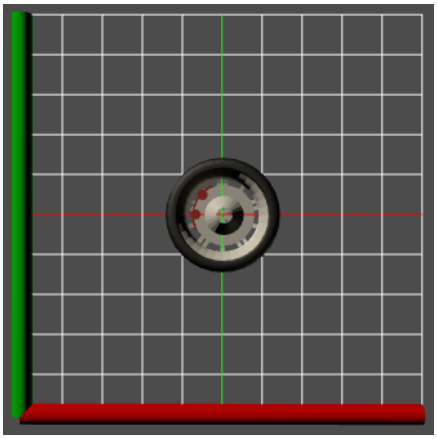


Figure 1: One possible orientation.

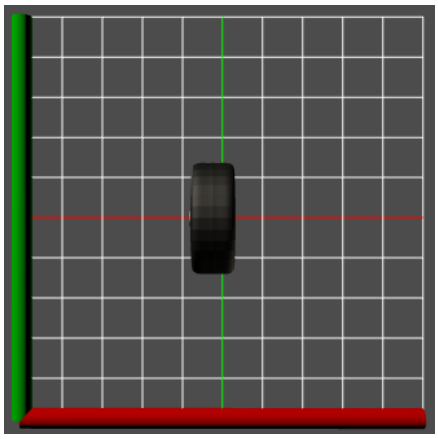


Figure 2: Another possible orientation.

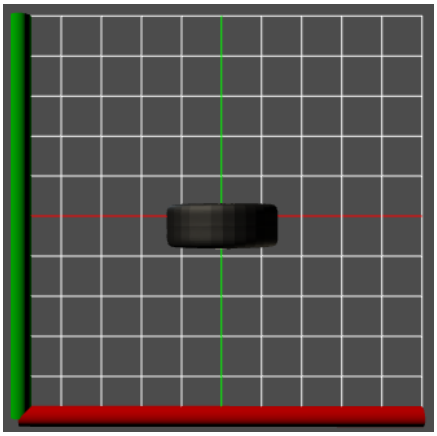


Figure 3: Yet another possible orientation.

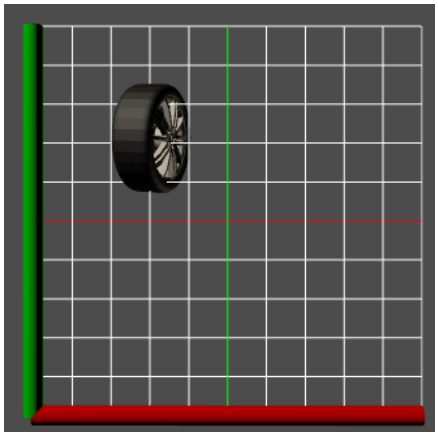


Figure 4: An improbable orientation, unless the 3D artist was very confused.

Four possible orientations the 3D model of a wheel could have.

2. During the editing process, the preservation of all information/metadata must be guaranteed.

2 Basics and Related Work

2.1 Scene-Graph

Scene-graphs can be used to group and organize 3D objects, their properties and concerning transformations. A glsDAG can be used to represent a scene-graph. It starts with a root node that is associated with one or more children. Each child can be an object or a group, again containing more children. A group can contain associated transformation information, like *translation*, *rotation* or *scaling*. This structure has certain advantages compared to applying all transformations to the raw meshes and sending everything to the Graphical Processing Unit (GPU). [1] Scene Structure and Integration Modeling Language (SSIML) [27] scene-graphs differ from the above definition and there are three types of nodes:

- Transform nodes,
- geometry nodes and
- group nodes.

2.1.1 Culling

Before using structures like scene-graphs, all polygons would be sent to the GPU and the GPU would need to test which polygons are actually in the view and thus need to be rendered. The problem of this approach is that this information was only known after doing a lot of calculations for every polygon already.
With a scene-graph it is possible to start from the root and traverse the graph, testing the bounding box of each group and only sending it to the GPU if it is completely visible. If it is not, the whole sub-tree is not sent to the GPU. If it is partially visible the same process is applied to the sub-tree. By using a structure that retains more information about what it represents it is possible to let the CPU do more of the heavy lifting and unburden the GPU.

> "Rather than do the heavy work at the OpenGL and polygon level, scene-graph architects realized they could better perform culling at higher level abstractions for greater efficiency. If we can remove the invisible objects first, then we make the hard-

> ware do less work and generally improve performance and the all-important frame-rate." [1]

2.1.2 Transformations

Another advantage is the way transformations work. Instead of applying them to the meshes directly, and keeping track of what meshes belong to the same object (like the chassis, the tires and the windows of a car), they can simply be nested under the same transformation group. The transformation thus applies to all objects associated with that group.

2.1.3 Reuse

With the ability to address nodes it is possible to reuse their information. If you have a car, it would be enough to have only one node containing the geometry for a tire, all other tires are merely addressing the tire with the geometry information (see listing 1). That way the memory footprint of an application can be reduced.

2.2 X3D

X3D [2] is the Extensible Markup Language (XML) representation of Virtual Reality Modeling Language (VRML) which was designed as a universal interactive 3D exchange format, much like Hypertext Transfer Protocol (HTML) is for written documents or SVG for vector graphics. Due to its XML structure it can be integrated in HTML documents, thus the Frauenhofer Institute pursued to implement a runtime that could interpret and visualize X3D in the browser, by using a WebGL context. It is called x3dom [3] and it is extensively used by SceGraToo, the tool that arose from this thesis.

2.3 x3dom

As said in the previous section, x3dom was developed by the Frauenhofer Institute to realize the vision that started VRML in the first place: *mark up interactive 3D content for the web.* On the web there are two entirely different approaches to describe the same thing:

- imperative
- declarative

```
<Group>
  <Transform>
    <Shape def="chassis">
        <Appearance>
            <Material></Material>
        </Appearance>
        <Box></Box>
    </Shape>
  </Transform>
  <Transform>
    <Shape def="tire">
        <Appearance>
             <Material></Material>
        </Appearance>
        <Torus></Torus>
    </Shape>
  </Transform>
  <Transform>
    <Shape use="tire"/>
  </Transform>
  <Transform>
    <Shape use="tire"/>
  </Transform>
  <Transform>
    <Shape use="tire"/>
  </Transform>
</Group>
```

Listing 1: Example X3D group showing the use of `def` and `use`.

Table 1: The matrix in this table classifies x3dom together with other common web technologies [3].

	2D	3D
Declarative	Scalable Vector Graphics (SVG) [4]	x3dom [3]
Imperative	Canvas [5]	WebGL [6]

As can be seen in table 1 x3dom complements the already existing technologies perfectly.

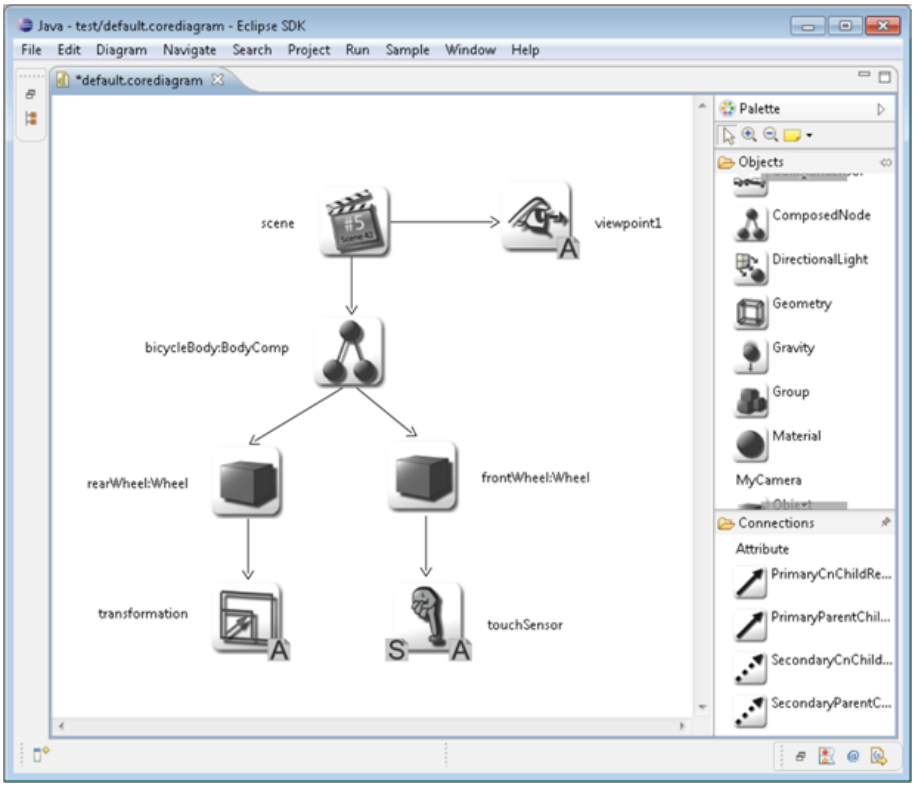


Figure 5: This figure shows the graphical editor for SSIML. In particular it shows a scene comprising a bike. [7]

2.4 SSIML

Heterogeneous developer groups, groups that are comprised of people from different backgrounds (programmers, 3D designers) have their difficulties working together. They work in different domains and use different tools adjusted to that domain. [25]

SSIML is a graphical approach to unify the scene-graph model and the the application model, thus making the communication between the different parties easier.

It also serves as a code generation template.

2.5 Roundtrip 3D

As stated above, when developing 3D applications, many different developers are involved, i.e. 3D designers, programmers and, ideally, also software designers (see figure 5).

Roundtrip3D was a research project that, amongst others, resulted in a graphical editor for SSIML models. It offers an approach for merging a developer's changes back into the main model. After all working copies are merged back into the main model (dropping unwanted or conflicting changes), all working copies are regenerated and delivered to the individual developers. After each round-trip every developer has a copy of the project that is consistent with everyone else's.

2.6 Related Work

The next section examines efforts and applications in the field of

- collaborative remote working on 3D models and
- online 3D editors.

2.6.1 3D Meteor

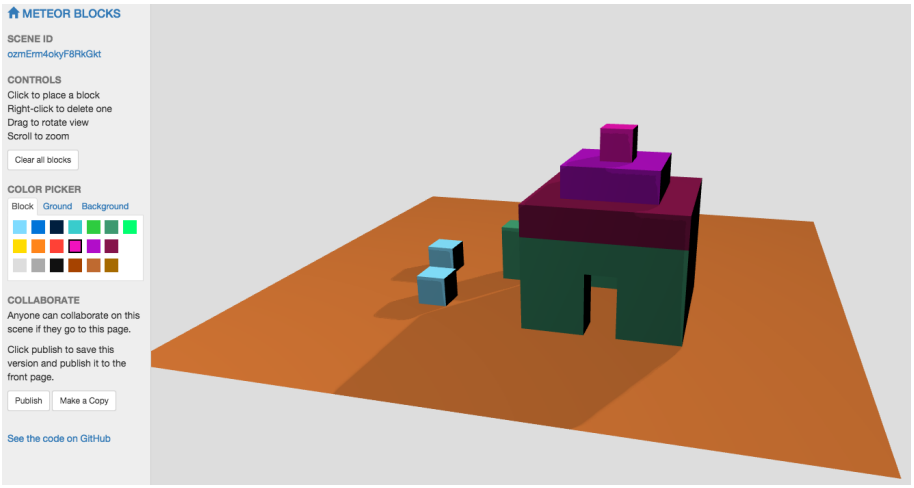


Figure 6: This is a scene in 3d Meteor showing a house and a garden of cubes.

This simple 3D editor allows the user to add and remove colored blocks to a scene. The synchronization is leveraging meteor's database collection subscription features. Meteor applications are comprised of a client side and a server side. The client can subscribe to database collections and gets automatically notified of changes to that collection by the server. The only thing that actually is synchronized is an array of *boxes*. A box is an object

with an x, y and z property describing its position. When a box is added to the collection the collection is synchronized with the server. The server informs all other browser instances that show this scene of the new box. These browser instances reevaluate the template that renders the x3dom scene to the Document Object Model (DOM) and the the DOM is updated to contain the new box, thus synchronizing the scene with the browser instance. [8]

2.6.2 Blender Plugin

As part of an asset management system the Université du Québec à Montréal implemented a plugin for Blender for collaborative working. An artist can record changes to wire-meshes and store them on a server. Another artist can download these changes and apply them to his working copy. The change sets are a simple list of vertices and their movement in the x, y and z space (see listing 2). [29]

```
95 [0.0000, 0.0000, 0.0000]
295 [0.0027, 0.0013, 0.0000]
309 [0.2123, 0.1001, 0.0000]
311 [0.3029, 0.1429, 0.0000]
```

Listing 2: This shows a change set of 4 polygons and how they where moved.

These are saved on the server and another user working on the same object can apply them to his working copy. They can actually be applied to any object that has the same number of vertices. That is also a shortcoming. Adding or removing vertices cannot be handled by the plugin. It is also not in real time, so it is more comparable to version control system like git just for 3D models.

2.6.3 Gizmos

Gizmos, also called manipulators, are handles or bounding boxes with handles that manipulate their containing objects in a predefined way when dragged. [9]

In X3D gizmos can be realized with specialized X3DDragSensorNodes [10], like:

SphereSensor SphereSensor converts pointing device motion into a spherical rotation around the origin of the local coordinate system. [11]

CylinderSensor The CylinderSensor node converts pointer motion (for example, from a mouse) into rotation values, using an invisible cylinder of infinite height, aligned with local Y-axis. [12]

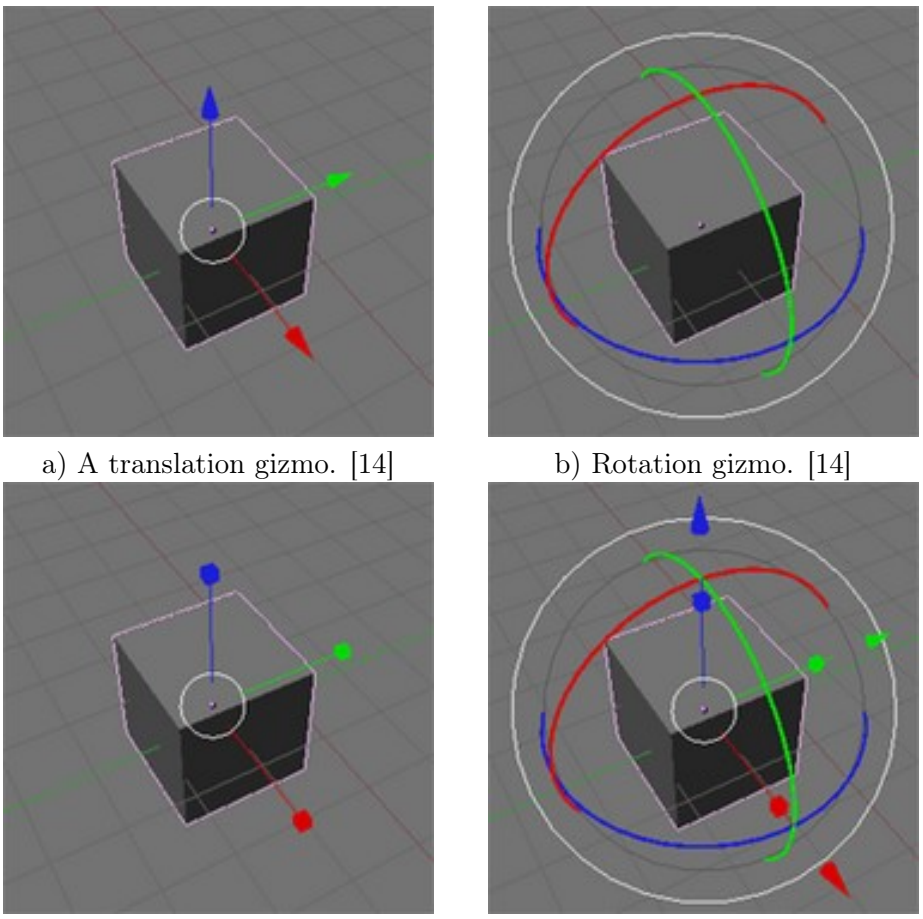
PlaneSensor PlaneSensor converts pointing device motion into 2D translation, parallel to the local Z=0 plane. Hint: You can constrain translation output to one axis by setting the respective minPosition and maxPosition members to equal values for that axis. [13]

The sensors track drag events on their siblings. In the example in figure 9 (which is taken directly from the x3dom website) the PlaneSensor tracks drag events on the cones and the cylinder that make up the cyan handle. Part of the structure of the scene can be seen in listing 3.

Every time it detects a drag event it converts it into a 2D transformation and raises an `onOutPutChange` event.
The callback `processTranslationGizmoEvent` is registered as an event handler. In this function the position of the handle is adjusted to make it follow the drag movement, also the position of the teapot is adjusted.

Having the handles being 3D objects within the scene, that look touchable and interactable, make it easier for users to find their way around the application. Instead of having to learn keyboard shortcuts users simply use their intuition and knowledge about how to interact with objects in the real world.

The pictures in 7 to 9 show different gizmos from different applications.



a) A translation gizmo. [14]

b) Rotation gizmo. [14]

c) Scale gizmo. [14]

d) All gizmos together. [14]

Figure 7: The same cube in Blender with different gizmos/transformers enabled.

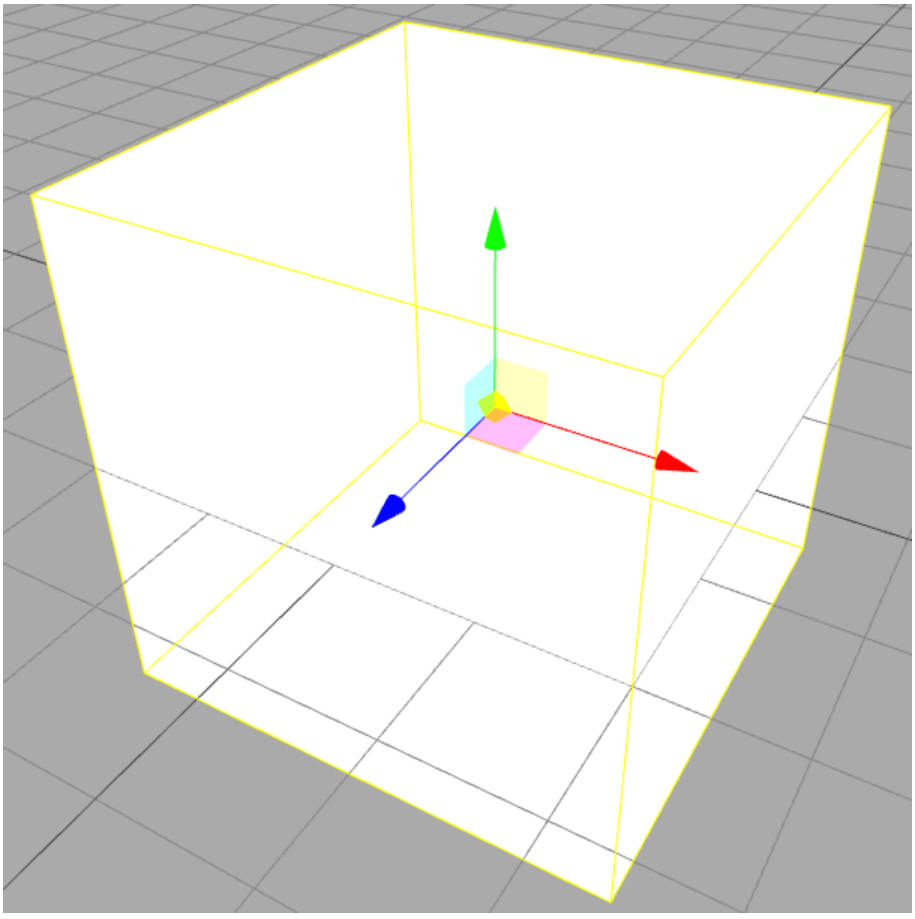


Figure 8: Shows translate gizmos along the x, y and z axis as well as gizmos that translate the cube along the xy, xz, yz and frustum plane. [15]

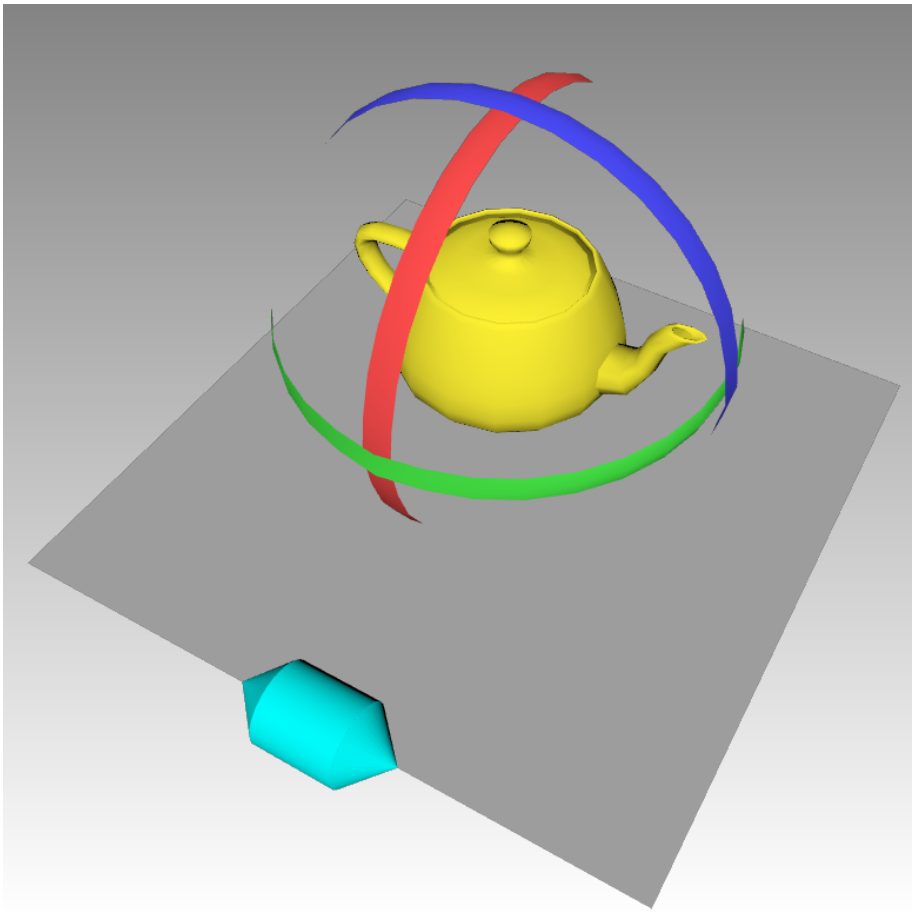


Figure 9: Shows an official x3dom tutorial for using sensors to create gizmos. [16]

```
<group>
  <planeSensor autoOffset='true' axisRotation='1 0 0 -1.57'
  ↪  minPosition='-6 0' maxPosition='6 0'
  ↪  onoutputchange='processTranslationGizmoEvent(event)'>
  </planeSensor>

  <transform id='translationHandleTransform'>
    <transform translation='0 -5.5 8' rotation='0 1 0 1.57'>
      <transform translation='0 0 1.5' rotation='1 0 0 1.57'>
        <shape DEF='CONE_CAP'>
          <appearance DEF='CYAN_MAT'><material diffuseColor='0
          ↪  1 1'></material></appearance>
          <cone height='1'></cone>
        </shape>
      </transform>
      <transform rotation='1 0 0 -1.57'>
        <shape>
          <appearance USE='CYAN_MAT'></appearance>
          <cylinder></cylinder>
        </shape>
      </transform>
      <transform translation='0 0 -1.5' rotation='1 0 0
      ↪  -1.57'>
        <shape USE='CONE_CAP'></shape>
      </transform>
    </transform>
  </transform>
</group>
```

Listing 3: This shows a group containing a planeSensor. It shows a part of the scene depicted in figure 9.

2.6.4 Component Editor

On the 12th of June 2015 the x3dom maintainers released their Component Editor. [17] It was released after the the work on SceGraToo started. Its development took about a year and three people working part-time on it. [18] Although it does offer all the wanted scene composition features, it lacks the ability to:

- Load an existing X3D file,
- save the scene as an X3D file and
- upload X3D files that can be included as inlines.

Scenes can only be loaded and saved as a JavaScript Object Notation (JSON) representation (see listing 4). Conversion between the formats may be possible, but meta information like Identifiers (IDs) in comments would be lost.

```
{
  "0": {
    "type": "Box",
    "transform": "1.000000, 0.000000, 0.000000, 0.000000,
    ↪  \n0.000000, 1.000000, 0.000000, 0.000000, \n0.000000,
    ↪  0.000000, 1.000000, 0.000000, \n0.000000, 0.000000,
    ↪  0.000000, 1.000000",
    "referencePoints": ["p1", "p2", "p3", "p4", "p5", "p6"],
    "parameters": {
      "Size": [1, 1, 1],
      "Positive Element": "true"
    }
  }
}
```

Listing 4: The JSON format used by the component editor to save scenes.

2.6.5 Real-Time Collaborative Scientific WebGL Visualization with Web Sockets

Using web sockets instead of AJAX is an interesting approach. [28] Especially the cut down on latency. It is over all very similar to the approach that was considered for SceGraToo, but not implemented due to time constraints. In the end the differences between SceGraToo's requirements and

theirs outweigh the similarities. SceGraToo not only has to visualize a scene, but also manipulate it. And to achieve a spectator mode like in this work not only the viewpoint would have to be synchronized but also the whole scene. This can either be done by sending the whole scene to the other clients on every change, or sending change sets, which poses an interesting challenge. They visualize a specific dataset in a threejs's specific JSON format [19]. SceGraToo only needs to render X3D scenes. Converting the scene into another format and having it rendered by another scene graph framework is unnecessary, since x3dom does a pretty good job doing this.

2.6.6 ParaViewWeb

Simple to use out of the box, but needs a paraview server instance and paraview does not support X3D as an input format. So using this is unfortunately not possible unless an import filter is written. The visualization is mainly meant to explore data sets. There is no easy way to manipulate the input data. This can only happen by extending the visualization pipeline via python scripts on the server.

3 Concept

In the following sections the initial design concept of SceGraToo is illustrated.

3.1 Server

3.1.1 Node.js

> "Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices." [20]

Node is used because it is inherently easy writing small servers with it, and SceGraToo's server is not going to be complex. [1]

Listing 5 shows a server that is looking up a user from the database and returning it as JSON to the browser.

```
const http = require('http')
const db = require('db')

http.createServer((request, response) => {
  db.getuser(function(error, user) {
    if (error) {
      return res.status(404).send(error);
    } else {
      response.writeHead(200, {'Content-Type':
      ↪  'application/json'})
      response.send(user)
    }
  })
}).listen(1337, '127.0.0.1')
```

Listing 5: an example server in Node.js, using the http module in its standard library

[1] Complex servers can be written in Node.js as well, it is just not encouraged, since most people tend to refactor their infrastructure into micro services.

3.1.2 Koa

> "Koa is a new web framework designed by the team behind Express, which aims to be a smaller, more expressive, and more robust foundation for web applications and APIs. Through leveraging generators Koa allows you to ditch callbacks and greatly increase error-handling. Koa does not bundle any middleware within core, and provides an elegant suite of methods that make writing servers fast and enjoyable." [21]

Koa is used to make writing the server and using asynchronouse functions in request handlers even simpler and clearer. See listing 6 for the same example from listing 5 written with koajs. It is not only shorter, but also simplier and clearer.

```
const db = require('db')
const koa = require('koa')
const app = koa()

app.use(function * (){
  this.body = yield db.getUser()
})

app.listen(3000)
```

Listing 6: an example server utilizing the koajs framework

3.2 Client

> I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.
>
> — Hoare, Turing Award Lecture 1980

The tree-view is the most important part of SceGraToo, it shows the the structure rather than the visual representation. Different off-the-shelf solutions, like angular or JQuery plugins, were tested against the following requirements:

1. Custom HTML elements as part of tree nodes (e.g. multiple checkboxes or multiple inputs),
2. ability to observe the tree node's state changing,
3. binding to an arbitrary model and
4. detecting inconsistencies between the model and the view and recovering from them.

Partially not met requirements:

- Custom elements as part of tree nodes and
- ability to listen to changes to the tree node.

Requirements none of the tested tools met:

- Binding to an arbitrary model and
- detecting inconsistencies between the model and the view and recovering from them.

None of the off-the-shelf solutions could satisfy all expectations. After evaluating a couple of solutions it was clear that the problem space was too specific and a custom solution is required.

3.2.1 Synchronization Process

Of all requirements, the most complicated part is keeping the tree-view in sync with the scene-graph while the scene-graph is being modified and vice versa.

Terminology

scene-graph The X3D representation of the scene as part of the DOM, see listing 7 and the screenshot in figure 11 from the Chrome DevTools.

scene-graph-node A specific scene graph node (e.g. `inline`, `transform` or `scene`).

tree-view-component Comprises all functionality related to parsing the scene-graph and creating the tree-view out of individual tree-view-node-components.

tree-view The HTML representation of the tree-view-component as part of the DOM, see figure 10 and listing 8 (HTML output shortened and simplified).

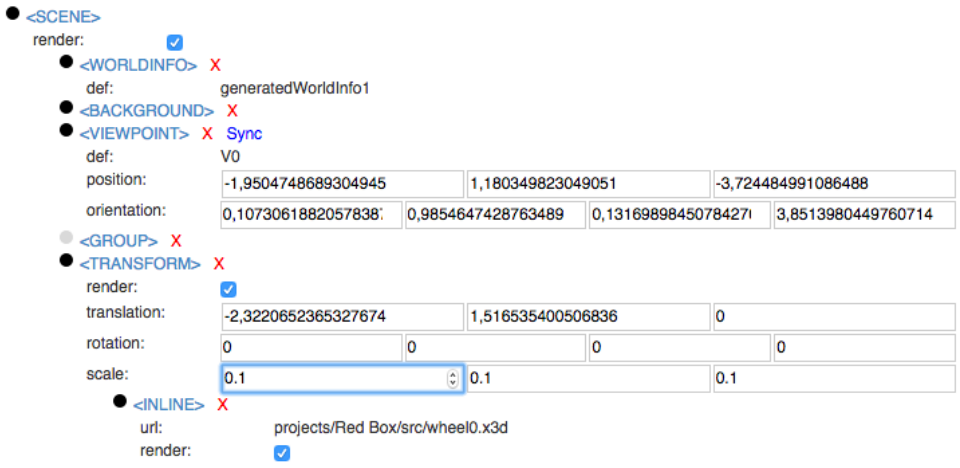


Figure 10: The rendered tree-view.

Figure 11: The X3D scene inside the DOM.

tree-view-node-component Comprises all functionality related to synchronizing changes from a scene-graph-node to the corresponding tree-view-node and vice-versa.

tree-view-node The HTML representation of the tree-view-node-component as part of the DOM, see figure 13 and figure 14.

The aim is to keep the tree-view a consistent representation of the scene-graph. The tree-view filters some nodes and attributes. As an example nodes contained in an `inline` are not shown, since SceGraToo's task is only to compose a scene of `inlines`, not to change anything inside the `inlines`. Also not all attributes are shown, but only the ones the user may be interested in (such as `DEF`, `translate` or `rotate`).

Figure 12: About half of the DOM elements that make up a tree-view of only three tree-view-nodes: a `scene`, a `transform` and an `inline`.

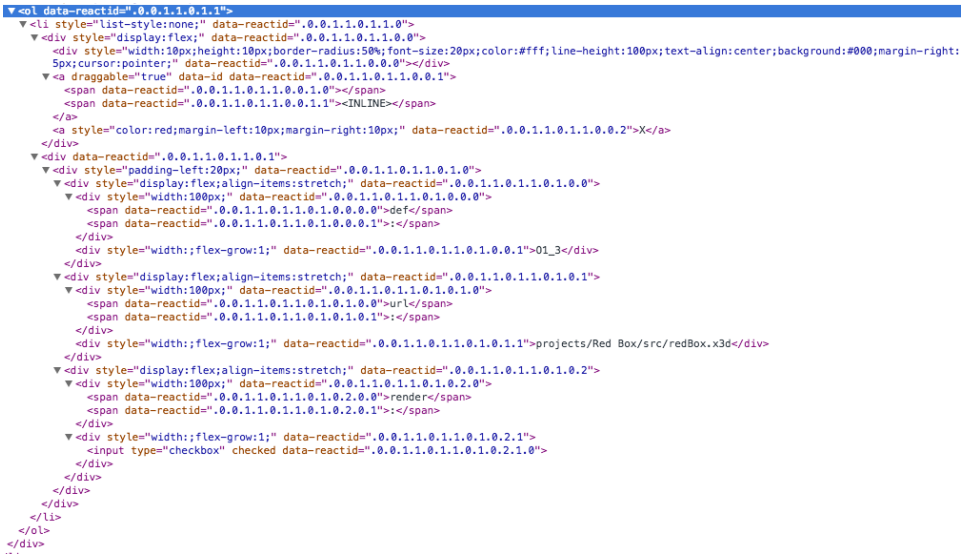


Figure 13: The DOM elements that make up a tree-view-node for an `inline`.

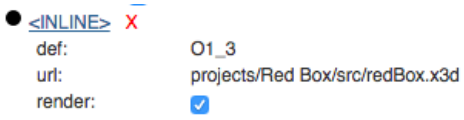


Figure 14: The rendered tree-view-node of an `inline`.

One approach is to instantiate the tree-view-component with a scene-graph-node as root node. For all child nodes the tree-view-component instatiates new tree-view-node-components. These tree-view-node-components create the corresponding tree-view-nodes while further traversing the scene-graph. For each scene-graph-node a corresponding tree-view-node-component is created. If there are no child nodes left the tree-view creation is done. Each tree-view-node-component creates all DOM elements necessary to represent the corresponding scene-graph-node in the DOM. Also each tree-view-node-component observes its corresponding scene-graph-node for attribute mutations and added or removed child nodes and acts accordingly.

Depending on how the scene-graph is mutated three main cases can be differentiated:

a scene-graph-node is added A new tree-view-node-component is instantiated, adding all DOM elements making up the tree-view-node to the DOM.

a scene-graph-node is deleted The corresponding tree-view-node-component is destroyed and all DOM elements making up that tree-view-node are removed from the DOM.

a scene-graph-node is mutated The corresponding DOM elements that make up the tree-view-node are altered to reflect the mutation.

Tree-view-nodes can also be used to edit scene-graph nodes' properties. When an input element that contains the x value of a transformation is edited its tree-view-node-component is notified of the change, by firing a change event to which the component subscribed, and applies the new value to the corresponding scene-graph-node.

It is assumed that the updates will always lead to consistent a state, where the scene-graph and the tree node converge. It is also assumed that an application may be buggy and in that case the synchronization process has no ability to detect if updates lead to an inconsistent state. It also has

no ability to recover from an inconsistent state, though without the ability to detect inconsistencies this does not really matter.

Following, the two main problems are described.

Problem 1: keeping the tree-view consistent with the scene-graph The difficulty to make sure that incremental updates are error-free exacerbates even more when further functionality - like checkboxes for specific properties or saving state in the tree-view that is not part of the scene-graph, like the possibility to collapse parts of the tree - is added to the tree-view.

Problem 2: implementation effort For every new feature four things have to implemented:

1. Code for parsing the scene-graph
2. Code to generate the tree-view-node
3. Code to synchronize changes to a scene-graph-node to the corresponding tree-view-node
4. Code to synchronize changes to a tree-view-node to the corresponding scene-graph-node

This can be greatly simplified if only the functionality for parsing the scene-graph and creating DOM elements to represent the parts of the scene-graph is implemented and on every change the whole tree-view is recreated by doing this again.

Problem 1 is disappears completely, because the incremental updates are gone and Problem 2 is reduced to the following two steps:

1. Code for parsing the scene-graph and generating the tree-view
2. Code to synchronize changes to a tree-view-node to the corresponding scene-graph-node

Rerendering everything on every change is usually inefficient. Removing a big part of the DOM and replacing it would kick of a *reflow*, which is the browser's process of laying out the content. This process is blocking, meaning the user can't scroll or otherwise interact with the application. [22]

React is used to minimize the possibility of a reflow happening. React calculates a lightweight representation of what the DOM should be like and compares that to what the DOM is already. It calculates a set of patches and only applies these to the DOM.

From a developer's point of view the application is programmed like it is completlely rerendered everytime something changes. But from the browser's point of view only the most minimal set of changes that are required to transform the DOM into disired state are applied, thus greatly reducing the risk of a reflow.

The code below (listings 9, 10 and 11) is for explanitary purposes to describe how react works. It does not resemble react's implementation in any way:

```
<ol data-reactid=".0">
  <li data-reactid=".0.0">scene
    <ol data-reactid=".0.0.0">
      <li data-reactid=".0.0.0.0">transform
        <ol data-reactid=".0.0.0.0.0">
          <li data-reactid=".0.0.0.0.0.0">inline</li>
        </ol>
      </li>
    </ol>
  </li>
</ol>
```

Listing 9: Old Virtual DOM

```
<ol data-reactid=".0">
  <li data-reactid=".0.0">scene
    <ol data-reactid=".0.0.0">
      <li data-reactid=".0.0.0.0">transform
        <ol data-reactid=".0.0.0.0.0">
          <li data-reactid=".0.0.0.0.0.0">inline</li>
        </ol>
      </li>
      <li>group</li>
    </ol>
  </li>
</ol>
```

Listing 10: New Virtual DOM

```
var li = document.createElement('li')
li.innerText = 'group'
document.querySelector('[data-reactid=".0.0.0"]')
  .appendChild(li)
```

Listing 11: Patch

That means as long as the code that parses the scene-graph and generates the lightweight representation of the tree-view is correct, the tree-view will represent the current state of the scene-graph.

Data Binding Another idea is to utilize templates and data binding. Frameworks like angular or web components implementations like polymer support templates and two way data binding. Following only angular directives are examined, but the same should be possible with web components.

An angular directive consists of a mostly logic less template and some javascript containing logic for creating the directive or reacting to events.

For each node a directive is instantiated which creates a template rendering the node. Also for each child it creates a new instance of itself.

Example:

The structure that is rendered is shown in listing 12. The `treenode`

(listing 13) expands into the node name and a `nodelist` (listing 14), that then expands into a list of tree-nodes for each child node (listing 15), that further expand again (listing 16). This recursive expanding stops when a `treenode` is childless.

```
node: {
  name: "scene",
  children: [
    {
      name: "viewpoint"
    },
    {
      name: "worldinfo"
    }
  ]
}
```

Listing 12: Example input data.

```
<treenode node="node">
</treenode>
```

Listing 13: The initial template, node is the node from the data in listing 12.

```
<treenode node="node">
  <span>{{node.name}}</span>
  <nodelist ng-repeat='node in children' children='children'>
  </nodelist>
</treenode>
```

Listing 14: the template expands itself, putting the node's name into a span and adding a nodelist directive the expands the node's children

```
<treenode node="node">
  <span>{{node.name}}</span>
  <nodelist ng-repeat='node in children' children='children'>
    <treenode node="children[0]">
    </treenode>
    <treenode node="children[1]">
    </treenode>
  </nodelist>
</treenode>
```

Listing 15: the nodelist expands the children array and renders a treenode for every child

```
<treenode node="node">
  <span>{{node.name}}</span>
  <nodelist ng-repeat='node in children' children='children'>
    <treenode node="children[0]">
      <span>{{node.name}}</span>
    </treenode>
    <treenode node="children[1]">
      <span>{{node.name}}</span>
    </treenode>
  </nodelist>
</treenode>
```

Listing 16: The treenode directive expands the nodes and renders their names, since there are no nodes left to render they stop.

Again, this is not an accurate depiction of how angular works, it is just for illustration purposes.

The scene-graph is traversed and for each eligible child node a new `treenode` is created. The double braces are angular's way to denote data-binding in templates. The data from the elements scope is automatically inserted and kept up to date. This data binding would then ensure that when the scene-graphs attributes are changed the model is kept in sync and the other way around.

3.3 Communication

The client comunicates with the server via a small HTTP API returning JSON. Making a GET request to `/projects` returns all projects stored on the server. Making a GET request to `/projects/unicorn` returns all data about unicorn project.

```
<x3d version="3.0" profile="Interaction" width="708px"
 ↪  height="354px">
<!-- id=69b81d54-6e7a-4967-acca-b8c89ba90782 -->
<scene render="true" bboxcenter="0,0,0" bboxsize="-1,-1,-1"
 ↪  pickmode="idBuf" dopickpass="true">
<worldinfo>
</worldinfo>
<background skycolor="0.3 0.3 0.3"></background>
<viewpoint fieldofview="0.7" position="1 1 3" orientation="0.1
 ↪  0.9 0.13 3.8">
</viewpoint>
<!-- id=8d3f0a8a-b6d7-4acc-922b-ea59364443fa -->
<group render="true" bboxcenter="0,0,0" bboxsize="-1,-1,-1">
  <transform render="true">
    <!-- id=8459b736-5a9d-4688-b624-e519857a92fd -->
    <inline url="projects/Red Box/src/redBox.x3d"
     ↪  render="true" load="true">
      <Shape render="true" isPickable="true">
        <Appearance sortType="auto" alphaClipThreshold="0.1">
          <Material diffuseColor="1 0 0"
           ↪  ambientIntensity="0.2"
           ↪  shininess="0.2"></Material>
        </Appearance>
        <Box solid="true" size="2,2,2"></Box>
      </Shape>
    </inline>
  </transform>
</group>
</scene>
</x3d>
```

Listing 7: X3D example scene.

```
<div>
  <li>
    <div>
      <a> <span>SCENE</span> </a>
    </div>
    <div>
      <div>
        <div>
          <div>
            <span>render:</span>
          </div>
          <div>
            <input type="checkbox">
          </div>
        </div>
      </div>
      <ol>
        <li>
          <div>
            <a> <span>WORLDINFO</span> </a>
            <a>X</a>
          </div>
          <div>
            <div>
              <div>
                <div>
                  <span>def:</span>
                </div>
                <div>generatedWorldInfo1</div>
              </div>
            </div>
          </div>
...
      </ol>
    </div>
  </li>
</div>
```

Listing 8: Example tree view structure, structure is simplified.

4 Implementation

4.1 Server

The server has a really small interface. When it gets a request it tries to to match the request to one of its routes, otherwise tries to serve the requested file from the file system (assuming it is a static file), otherwise answers with a 404, which means `Not Found`.

Some of the routes it answers to are:

POST /projects/:project/src/:file saving files being uploaded
GET /projects return all projects on the server
GET /projects/:project return all information for the project `:project`

4.2 Client

4.2.1 angular

AngularJS is used for:

- Bootstrapping the application
- Modularization
- Dependency management
- Resource management
- Routing
- Less dynamic views

Bootstrapping Listing 17 initializes an SceGraToo. In the `config` function routes are defined. When a link is clicked the browser will not make a request to the server and load that page instead a new controller takes over and renders a different template. The advantage of this approach is that the user sees immediate feedback while navigating. The application can render the view and react to user input while its still waiting for some requested resources from the server (like a list of all available projects). This approach is called single-page application [30].

Modularisation and Dependency Injection Each controller, view or service is contained in its own module and does not pollute the global name space. In a browser's javascript context the global name space refers to

```
window.angular.module('scegratooApp')
  .config(function ($routeProvider) {
    $routeProvider
      .when('/', {
        redirectTo: '/projects'
      })
      .when('/projects', {
        templateUrl: 'views/projects.html',
        controller: 'ProjectsCtrl'
      })
      .when('/projects/:project', {
        templateUrl: 'views/project.html',
        controller: 'ProjectCtrl'
      })
      .when('/projects/:project/:file*', {
        templateUrl: 'views/projects/:project/x3d/:file.html',
        controller: 'ProjectsProjectX3dFileCtrl'
      })
  })
```

Listing 17: This is how SceGraToo is initialized. It also shows how the routing is defined.

the name space that belongs to the `window` object. Defining variables in scripts defines this variable on the window object. Angular modules prevent this. Modules are registered on a specific angular application (thus one website could also accommodate multiple angular applications). The defined variables are contained by creating a function that returns whatever the module is supposed to contain, thus creating a closure. Listing 18 shows a module that creates a WeakMap and returns it. Modules can denote that they depend on other modules, this can be seen in listing 19. The `MoveableUtils` request that the `moveable` module is injected into it when initializing. Angular creates a dependency graph and resolves dependencies automatically.

Views Angular tempates are mostly logicless (except for ng-repeat iterators and ng-if statements). Listing 20 shows a template that renders all prejects the corresponding controller retrieved from the server.

```
window.angular.module('scegratooApp')
  .service('moveables', function () {
    return new WeakMap()
  })
```

Listing 18: This module creates a WeakMap that can be injected in multiple other modules. These modules all share the same WeakMap since services are singletons. `service`'s first argument is the `service`'s name, that can be used by other modules by importing it.

```
angular.module('scegratooApp')
  .service('MoveableUtils', function (moveables) {
    return {
      logMoveables: () => console.log(moveables)
    }
  })
```

Listing 19: This module requests the `moveables` module to be injected.

4.2.2 react

React is utilized by SceGraToo to render the tree-view that gives a more structured view of the scene-graph than the rendered scene does.

React works by creating components and nesting them. Listing 21 shows the `TreeView` component. The `TreeNode` is another component that handles a specific tree-view-node, components keep instantiating and returning components until the whole scene-graph is traversed. In listing 22 it is shown how it is rendered to the DOM. The HTML syntax is simply syntactic sugar and is transpiled into normal javascript before being evaluated (the transpiled equivalent of listing 22 is shown in listing 23).

Using react the view virtually becomes a function of its input. The input is the root node of the scene-graph, the X3D node.

The parsing and rendering process can be described as follows:

1. Choose a graph node as the root
2. call the node component with that graph node,
3. instantiate corresponding components for each of the nodes attribute,
4. if the graph node has child nodes call the node component again with each child node and return their return values or
5. if the graph node has no children return an empty element.

```
<sgt-navigation-bar>
</sgt-navigation-bar>
<div class="sash">
  <h3>
    Editable files for {{project.name}}
  </h3>
  <div ng-repeat="file in project.files | orderBy:'view'">
    <div ng-show="file.view">
      {{file.view}} -
      <a
        href="#/projects/{{projectName}}/
        ↪  {{file.view}}/{{file.path}}">
        {{file.path}}
      </a>
    </div>
  </div>
</div>
```

Listing 20: A template that renders projects that the controller retrieved from the server

4.2.3 Synchronization Process

Synchronizing the tree-view when the scene-graph changes is done by calling `React.render` again, just like in listing 22. React calculates the changes that need to be done to update the DOM and applies these.

If the tree-view changes the scene-graph, the same thing happens. Listing 24 show a check box component. It receives an property called `owner`. That is a scene-graph node. Nodes in x3dom have the render property. If the property is true, that node and all its children are rendered, if not, they are not visible. The component is showing the state of the the `owner`'s render property's state. When the user clicks that check box the `owner`'s attribute is changed. The component does not have to update the DOM node, react is doing it the next time `React.render` is called. This is a simple example but the concept holds up for more complicated interactions like adding new nodes or moving via drag and drop.

```
React.createClass({
  displayName: 'TreeView',
  propTypes: {
    data: React.PropTypes.object.isRequired
  },
  render: function () {
    if (this.props.data.runtime) {
      return (
        <TreeNode
          data={this.props.data.querySelector('scene')}
          runtime={this.props.data.runtime}
        />
      )
    } else {
      return <div/>
    }
  }
})
```

Listing 21: The TreeView component is instantiated with a node. Its render function returns an instantiated TreeNode unless the given node has no runtime property, in that case it just returns an empty div.

```
const treeViewContainer = document.querySelector('#container')
const x3dNode = document.querySelector('x3d')
React.render(<TreeView data={x3dNode} />, treeViewContainer)
```

Listing 22: Shows how react renders to the DOM. The `treeViewContainer` is the the DOM element react will render into. `x3dNode` is the scene-graph in the DOM.

```
const treeViewContainer = document.querySelector('#container')
const x3dNode = document.querySelector('x3d')
React.render(React.createElement(TreeView, { data: x3dNode }),
↪  treeViewContainer)
```

Listing 23: Shows the transpilation output of listing 22. This is standard compliant javascript.

```
const TreeNodeAttributeRender = React.createClass({
  displayName: 'TreeNodeAttributeRender',
  propTypes: {
    owner: React.PropTypes.object.isRequired,
  },
  changeHandler: function (event) {
    if (event.currentTarget.checked) {
      this.props.owner.setAttribute('render', true)
    } else {
      this.props.owner.setAttribute('render', false)
    }
  },
  render: function () {
    const attribute = this.props.owner.getAttribute('render')
    const checked = render === 'true'

    return <input type='checkbox' checked={checked}
     ↪  onChange={this.changeHandler} />
  }
})
```

Listing 24: A component that renders a checkbox that show the `owner` render property's state. Clicking the checkbox changes the `owner`'s property's state.

5 Results

5.1 Why Scegratoo is the best since sliced Bread

List of Tables

1 The matrix in this table classifies x3dom together with other common web technologies [3]. 7

List of Figures

1 One possible orientation. 3
2 Another possible orientation. 3
3 Yet another possible orientation. 3
4 An improbable orientation, unless the 3D artist was very confused. 3
5 This figure shows the graphical editor for SSIML. In particular it shows a scene comprising a bike. [7] 8
6 This is a scene in 3d Meteor showing a house and a garden of cubes. 9
7 The same cube in Blender with different gizmos/transformers enabled. 12
8 Shows translate gizmos along the x, y and z axis as well as gizmos that translate the cube along the xy, xz, yz and frustum plane. [15] 13
9 Shows an official x3dom tutorial for using sensors to create gizmos. [16] 14
10 The rendered tree-view. 21
11 The X3D scene inside the DOM. 21
12 About half of the DOM elements that make up a tree-view of only three tree-view-nodes: a `scene`, a `transform` and an `inline`. 22
13 The DOM elements that make up a tree-view-node for an `inline`. 22
14 The rendered tree-view-node of an `inline`. 23

List of Listings

1 Example X3D group showing the use of `def` and `use`. 7
2 This shows a change set of 4 polygons and how they where moved. 10
3 This shows a group containing a planeSensor. It shows a part of the scene depicted in figure 9. 15
4 The JSON format used by the component editor to save scenes. 16
5 an example server in Node.js, using the http module in its standard library 18
6 an example server utilizing the koajs framework 19
9 Old Virtual DOM 25
10 New Virtual DOM 26
11 Patch 26
12 Example input data. 27
13 The initial template, node is the node from the data in listing 12. 27
14 the template expands itself, putting the node's name into a span and adding a nodelist directive the expands the node's children 27
15 the nodelist expands the children array and renders a treenode for every child 28
16 The treenode directive expands the nodes and renders their names, since there are no nodes left to render they stop. . . . 28
7 X3D example scene. 30
8 Example tree view structure, structure is simplified. 31
17 This is how SceGraToo is initialized. It also shows how the routing is defined. 33
18 This module creates a WeakMap that can be injected in multiple other modules. These modules all share the same WeakMap since services are singletons. `service`'s first argument is the `service`'s name, that can be used by other modules by importing it. 34
19 This module requests the `moveables` module to be injected. . 34
20 A template that renders projects that the controller retrieved from the server 35

21 The TreeView component is instantiated with a node. Its render function returns an instantiated TreeNode unless the given node has no runtime property, in that case it just returns an empty div. 36
22 Shows how react renders to the DOM. The `treeViewContainer` is the the DOM element react will render into. `x3dNode` is the scene-graph in the DOM. 36
23 Shows the transpilation output of listing 22. This is standard compliant javascript. 36
24 A component that renders a checkbox that show the `owner` render property's state. Clicking the checkbox changes the `owner`'s property's state. 37

Bibliography

[1] URL http://www.realityprime.com/blog/2007/06/scenegraphs-past-present-and-future/.

[2] URL http://www.web3d.org/x3d/what-x3d.

[3] URL http://www.x3dom.org.

[4] URL http://www.w3.org/Graphics/SVG/.

[5] URL http://www.w3.org/TR/html5/scripting-1.html#the-canvas-element.

[6] URL https://www.khronos.org/webgl/.

[7] URL http://tu-freiberg.de/fakult1/inf/professuren/virtuelle-realitaet-und-multimedia/forschung-jung/roundtrip3d.

[8] URL http://3d.meteor.com/.

[9] URL https://en.wikipedia.org/wiki/Gizmo.

[10] URL http://doc.x3dom.org/author/PointingDeviceSensor/X3DDragSensorNode.html.

[11] URL http://doc.x3dom.org/author/PointingDeviceSensor/SphereSensor.html.

[12] URL http://doc.x3dom.org/author/PointingDeviceSensor/CylinderSensor.html.

[13] URL http://doc.x3dom.org/author/PointingDeviceSensor/PlaneSensor.html.

[14] URL http://wiki.blender.org/index.php/Doc:2.4/Manual/3D_interaction/Transform_Control/Manipulators.

[15] URL http://threejs.org/editor/.

[16] URL http://doc.x3dom.org/tutorials/animationInteraction/transformations/example.html.

[17] URL https://github.com/x3dom/component-editor.

[18] URL https://github.com/x3dom/component-editor/issues/1.

[19] URL https://github.com/mrdoob/three.js/wiki/JSON-Geometry-format-4.

[20] URL https://nodejs.org/.

[21] URL http://koajs.com/.

[22] URL https://developers.google.com/speed/articles/reflow.

[23] URL https://git-scm.com/.

[24] URL https://www.opengl.org/about/.

[25] Glinz, Martin und Samuel A. Fricker. On shared understanding in software engineering: An essay. *Comput. Sci.*, 30(3-4):363–376, August 2015. ISSN 1865-2034. URL http://dx.doi.org/10.1007/s00450-014-0256-x.

[26] Jung, Bernhard, Matthias Lenk und Arnd Vitzthum. Structured development of 3d applications: Round-trip engineering in interdisciplinary teams. *Comput. Sci.*, 30(3-4):285–301, August 2015. ISSN 1865-2034. URL http://dx.doi.org/10.1007/s00450-014-0258-8.

[27] Lenk, Matthias, Arnd Vitzthum und Bernhard Jung. Model-driven iterative development of 3d web-applications using ssiml, x3d and javascript. In *Proceedings of the 17th International Conference on 3D Web Technology*, Web3D '12, Seite 161–169, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1432-9. URL http://doi.acm.org/10.1145/2338714.2338742.

[28] Marion, Charles und Julien Jomier. Real-time collaborative scientific webgl visualization with websocket. In *Proceedings of the 17th International Conference on 3D Web Technology*, Web3D '12, Seite 47–50, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1432-9. URL http://doi.acm.org/10.1145/2338714.2338721.

[29] Martin Lesage, Martin Lesage, Gilles Raîche. A blender plugin for collaborative work on the articiel platform. 2007.

[30] Mikowski, Michael und Josh Powell. *Single Page Web Applications: JavaScript End-to-end*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2013. ISBN 1617290750, 9781617290756.

Glossary

3D 3 Dimensional. 1–3, 5–11, 41

DOM Document Object Model. 10, 20–23, 27, 28, 35–37, 41, 43

git A free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. [23]. 10

GPU Graphical Processing Unit. 5

HTML Hypertext Transfer Protocol. 6, 20, 21, 35

ID Identifier. 16

JSON JavaScript Object Notation. 16–18, 32, 42

OpenGL Open Graphics Library. An environment for developing portable, interactive 2D and 3D graphics applications. [24]. 5

R3D Roundtrip3D. 1

Roundtrip3D Aim to extend SSIML, a MDD approach for 3D development to full round-trip engineering. [7]. 1, 2, 16–18, 34, 35, 42

SSIML Scene Structure and Integration Modeling Language. 5

SVG Scalable Vector Graphics. 6, 7

VRML Virtual Reality Modeling Language. 6, 47

WebGL A cross-platform, royalty-free web standard for a low-level 3D graphics API based on OpenGL ES 2.0. [6]. 6, 7

X3D An XML based scene description language and the successor of VRML. 1, 2, 6, 7, 10, 16, 17, 20, 21, 25, 35, 41, 42

XML Extensible Markup Language. 6