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:	November 4, 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, November 4, 2015 _____

Contents

1 Introduction

The goal of the thesis is to create an editor for 3D (3 dimensional) scenes, 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 (R3D) [27].

1.1 Motivation

As part of the R3D project, a round-trip framework was developed. This framework also includes a graphical editor for SSIML (Scene Structure and Integration Modeling Language) [28] models, to describe 3D applications. Then 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 scene structure (see Listing 7) is mostly generated. Attribute values of respective nodes, such as transform 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. 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 the following 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).

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

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 the 3D 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 be used for this, but their import and export filters 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 (see Figure 9).

1.2 Scope

The provided application (SceGraToo) addresses two issues:

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

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

2 Basics and Related Work

This section examines fundamental and underlying technologies needed to comprehend the described problems and their solutions.

2.1 Scene-Graph

Scene-graphs can be used to group and organize 3D objects, their properties and concerning transformations. A Directed Acyclic Graph (DAG) can be used to represent a scene-graph. It starts with a root node that is associated with one or more child nodes. Each child node can be an object or a group, again containing more child nodes. 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]. SSIML 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 were sent to the GPU. The GPU would then test which polygons are actually in the view and had to be rendered. The problem of this approach is that this information was only known after doing a lot of calculations for every polygon.
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 hardware do less work and generally improve performance and the all-important frame-rate." [1]

2.1.2 Transformations

Another advantage of scene-graphs 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 contained by the group.

2.1.3 Reusing Nodes or Subtrees

With the ability to address nodes it is possible to reuse them (and all contained information and child nodes). For a car, it would be enough to have only one node containing the geometry for a tire, all other tires are merely referencing the tire with the geometry information (see Listing 1). That way the memory footprint and the overall complexity of an application can be reduced.

2.2 Declarative Scene Description via X3D

X3D [2] is the XML (Extensible Markup Language) representation of VRML (Virtual Reality Modeling Language) which was designed as a universal interactive 3D exchange format, much like what HTML (Hypertext Transfer Protocol) is for textual documents or SVG for vector graphics. Due to its XML structure it can be integrated in HTML documents. The Frauenhofer Institute pursued to implement an environment 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 Declarative Scene Description for the Web via 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*

```
<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` attributes . Figure 2 shows the rendered car.

up interactive 3D content for the web. On the web there are two entirely different approaches to describe 2D or 3D graphics:

- imperative
- declarative

Table 1: The matrix in this table classifies X3DOM together with other web standards concerning computer graphics [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.

2.4 3D Application Modeling via SSIML

Heterogeneous developer groups, are comprised of people with different different skills and terminologies (e.g. programmers and 3D designers), have their difficulties working together. They work in different domains and use different tools adjusted to that domain [26].

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

2.5 Roundtrip 3D

As stated above, when developing 3D applications, many different kinds of developers are involved, i.e. 3D designers, programmers and, ideally, software designers. Figure 3 shows the R3D editor with an example SSIML diagram. Figure 4 shows the proposed work-flow.

Roundtrip3D was a research project that, amongst others, resulted in a graphical editor for SSIML models. These SSIML models are used as input to generate source code, such as JavaScript, Java or C++, and an X3D file describing the scene. These code bases are further refined by the respective developers, i.e 3D designers or programmers. R3D offers an approach for merging the developers' changes back into the main model. After all working copies are merged back into the main model (dropping unwanted or conflicting changes), all code bases are regenerated, including the developers refinements and delivered to the individual developers. After this step, every developer has a copy of the project that is consistent with everyone else's. [27] It is paramount that the tools used to refine the scene must not autonomously change scene-information that is relevant for the round trip process. Such changes are for instance automatically renaming nodes, restructuring the scene-graph's by adding group nodes or removing comments or attributes. This thesis addresses issues by proposing a new 3D composition tool that retains all scene-information.

2.6 Related Work

As described in section 1.1, efforts in the following fields are essential for this work.

- Collaborative working on 3D models.
- Online 3D editors.

2.6.1 3D Meteor

This simple 3D editor allows users to add and remove colored blocks to a scene. A scene can be opened and edited by multiple users simultaneously. The synchronization is leveraging meteor's database collection subscription features. Meteor applications are comprised of a client side and a server side part. The client can subscribe to database collections and automatically gets notified of changes to that collection by the server. The only thing that is synchronized is an array of *boxes*. A box is an object with an x, y, z and a color 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, that this box was added to the *boxes* collection. These browser instances reevaluate the template that renders the X3DOM scene to the Document Object Model (DOM) and update the DOM to contain the new box, thus synchronizing all browser instances. [7]

2.6.2 Collaborative Working with Blender

As part of an asset management system, the Université du Québec à Montréal implemented a plug-in for Blender to enable 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. A set of changes is a simple list of vertices and their movement in the x, y and z direction (see Listing 2). [30]

```
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 set of changes of 4 polygons and how they where moved.

These sets are saved on the server. Other users, working on the same object, can apply them to their working copies. They can actually be applied to any object that has the same number of vertices. That is also one of the

shortcomings. Adding or removing vertices cannot be handled by the plug-in. And the synchronization is not in real time. As a result it is more comparable to version control systems, like git, but 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 being dragged (Figures 6 to 8). [8]

In X3D, gizmos can be realized with specialized X3DDragSensorNodes [9], like:

SphereSensor "SphereSensor converts pointing device motion into a spherical rotation around the origin of the local coordinate system [10]."

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" [11].

PlaneSensor "`PlaneSensor` converts pointing device motion into 2D translation, parallel to the local Z=0 plane. [12]."

The sensors track drag events on their siblings. In the example in Figure 8, 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.

```
<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: `PlaneSensor` node to register drag event on its siblings. This is part of the scene depicted in Figure 8.

2.6.4 Component Editor

The X3DOM maintainers released their Component Editor [13]. It was released after the the work on SceGraToo started. Its development took about a year and three people working on it part-time [14]. Although it does offer all the wanted scene composition features, it lacks the ability to:

- Load an existing X3D file,
- serialize 3D scenes to X3D files and
- upload X3D files that can be included as inline nodes.

Scenes can only be deserialized and serialized as JavaScript Object Notation (JSON) representations (see Listing 4). Conversion between the formats may be possible, but meta information like Identifiers (IDs) in comments would be lost. And these are important for the round-trip process (see 2.5)

```
{
  "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 a promising approach performance wise [29]. 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 its complexity. In the end, the differences between SceGraToo's requirements

and theirs outweigh the similarities. Besides visualizing a scene, SceGraToo also has to manipulate it. So, 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 a challenging task. They visualize a specific dataset in a threejs's specific JSON format [15]. 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 is pretty good at this.

2.6.6 ParaViewWeb

It is 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 X3D importer is written. The visualization is mainly meant to explore data sets. There is no straight forward way to manipulate the data. This can only happen by extending the visualization pipeline via python scripts on the server. [16]

3 Concept

SceGraToo uses a client server architecture. The scene data is stored on the server and is to be extended by X3D files that are uploaded using a web browser. In the following sections the initial design concept of SceGraToo is illustrated.

3.1 Server

In the following I'll state the server requirements and the technology that were considered for the implementation.

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." [17]

Node.js allows for the straight forward implementation of web servers. Listing 5 shows a server that looks up users from the database and returns them as JSON to the browser.

```
1  const http = require('http')
2  const db = require('db')
3
4  http.createServer((request, response) => {
5    db.getuser(function(error, user) {
6      if (error) {
7        return res.status(404).send(error);
8      } else {
9        response.writeHead(200, {'Content-Type':
       ↪  'application/json'})
10       response.send(user)
11     }
12   })
13 }).listen(1337, '127.0.0.1')
```

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

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." [18]

Koa is used to make writing the server, and using asynchronouse functions in request handlers simpler and clearer. See Listing 6 for the same example from Listing 5 written with koajs. It is not only more concise, but also more straight forward.

```
1 const db = require('db')
2 const koa = require('koa')
3 const app = koa()
4
5 app.use(function * (){
6   this.body = yield db.getUser()
7 })
8
9 app.listen(3000)
```

Listing 6: An example server utilizing the Koa 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 structure rather than the rendered representation. Different off-the-shelf solutions, like angular or JQuery plug-ins, 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 10 from the Chrome DevTools.

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

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 9 and Listing 8 (HTML output shortened and simplified).

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 12 and Figure 13.

```
<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: Generated X3D example scene.

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

```
<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: Simplified tree-view structure.

to compose a scene of `inlines`, not to change something inside the `inlines`. Also not all attributes are shown, but only the ones the user may be inter-

ested in (such as `DEF`, `translate` or `rotate`).

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 a consistent 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.

In the following, two main issues 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 is added to the tree-view, like checkboxes for specific properties or saving state in the tree-view that is not part of the scene-graph, e.g the possibility to collapse parts of the tree.

Problem 2: implementation effort For every new feature four pieces of code have to be written:

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

This can be simplified if only the functionality for parsing the scene-graph, and for creating the DOM elements that represent the scene-graph, is implemented and, on every change, the whole tree-view is recreated by repeating these steps.

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

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

Rerendering the whole part of the DOM on every change usually is inefficient. Removing a big part of the DOM and replacing it would lead to 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. [19]

React is used to minimize the possibility of a reflow. React calculates a lightweight representation of what the DOM (called the virtual DOM) should be like and compares that to the present DOM. 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. From the browser's point of view only a minimal set of changes, only those that are required

to transform the DOM into the desired state, are applied, thus enormously 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: Virtual DOM with newly created `li` node.

```
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 to parse the scene-graph and to generate the lightweight representation of the tree-view is correct, the tree-view will represent the current state of the scene-graph.

Data Binding Another approach 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 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 rendered structure 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, that 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.

This is an minimal example to demonstrate how angular directives work from a programmer's point of view. The scene-graph is traversed and, for each eligible child node, a new `treenode` is created. The double braces are angular's syntax to denote data-binding in templates. Data from the elements scope is automatically inserted and kept up to date. Then, the data-binding would ensure, that the tree-view and the model are kept in sync when the `treenode`'s attributes are changed and the other way around.

3.3 Communication

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

4 Implementation

In the following sections the technology that was eventually used to implement SceGraToo is dissected and explained in detail.

4.1 Server

The server has a small interface, meaning it does not have a lot of routes it serves. When it receives a `GET` request from a client it tries to match the request to one of its routes. If no route matches it tries to find a static files that matches the requested route URL. If it finds the file, the file is served to the client. If not a 404 HTTP code is returned, which means `Not Found`. A `POST` route also exists for uploading content. `POST` and `GET` are different HTTP methods. An HTTP request to the same route can happen via different methods to evoke different action from the server [20].

Some of the routes the server 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

The client uses to frameworks to solve different problems. Angular is used for managing and organizing most of the application, while react is used for highly dynamic views, like the tree-view.

4.2.1 Angular

AngularJS is used for:

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

Bootstrapping Web applications need to be bootstrapped. There has to be one entry-point for starting the the application. And this entry-point needs to be called after all scripts, that are referenced in the HTML document, have been downloaded from the server. Angular accomplishes this by subscribing to the `DOMContentLoaded` event. Angular will then look for DOM elements with an `ng-app` attribute. That attribute is set by the developer and tells angular, that the corresponding DOM element is supposed to contain an application and the application's name. The bootstrapping process is described in Figure 14.

The code from Listing 17 initializes SceGraToo. In the `config` function, routes are defined. When a link is clicked, the browser will not issue 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 perceives immediate feedback, while navigating. The application can render the view and react to user input, while it is still waiting for some requested resources from the server (like a list of all available projects). This approach is called single-page application [31].

Modularisation and Dependency Injection Each controller, view or service is contained in its own module and does not pollute the global namespace. An angular service is a lazily instantiated singleton. In a browser's JavaScript context, the global namespace refers to the namespace that belongs to the `window` object. Defining variables in a script defines these variables 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 an `Array` 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 templates are mostly logic-less. Listing 20 shows a template that renders all projects the corresponding controller retrieved from the server.

```
1  window.angular.module('scegratooApp')
2    .config(function ($routeProvider) {
3      $routeProvider
4        .when('/', {
5          redirectTo: '/projects'
6        })
7        .when('/projects', {
8          templateUrl: 'views/projects.html',
9          controller: 'ProjectsCtrl'
10       })
11       .when('/projects/:project', {
12         templateUrl: 'views/project.html',
13         controller: 'ProjectCtrl'
14       })
15       .when('/projects/:project/:file*', {
16         templateUrl: 'views/projects/:project/x3d/:file.html',
17         controller: 'ProjectsProjectX3dFileCtrl'
18       })
19   })
```

Listing 17: This is how SceGraToo is initialized. It also shows how the routing is defined. E.g. in line 4 to 6 for the index route (the one that is requested when the request contains only the domain: *www.example.com*) is defined to redirect to `/projects` and in line 7 to 10 it is defined that the `/project` rout is controlled by the `ProjectCtrl` and rendered with the `views/projects.html` template.

```
1  window.angular.module('scegratooApp')
2    .service('moveables', function () {
3      return new Array()
4    })
```

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

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

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

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

```
<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

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 inside the JavaScript code 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.

```
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.

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

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

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 process is repeated. Listing 24 show a check box component. It receives a property called `owner`. That is a scene-graph node. Nodes in X3DOM have the render attribute. If a attribute is true, that node and all its children, with their render attribute sent to true, are rendered, if not, they are not visible. The component is showing the state of the the `owner`'s render attribute's state. When the user clicks the 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.

```
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 check box that show the `owner` render property's state. Clicking the check box changes the `owner`'s property's state.

5 Conclusion

The presented work explores the planning and the implementation of a 3D composition tool. The created scene composition tool, SceGraToo, enables the user to:

- Upload a R3D project,
- visualize the scene,
- translate, rotate and scale any 3D object,
- remove 3D objects,
- reorder the the 3D objects in the the tree-view,
- add new objects via dragging them over the tree-view and dropping them on any group or transform node,
- save the changed scene on the server and
- download it again.

And all of that is possible from within the browser. The downloaded project can be further processed with he R3D framework. What couldn't be implemented, due to time constraints, is a way to synchronize a scene between browser sessions and allow multiple users to change one scene.

5.1 Example

In the following I'll demonstrate an example. Listing 25 shows an X3D scene with 3 cubes. All 3 cubes lie in the origin of the coordinate system (see Figures 15 and 16). After moving the cubes via the 3D scene and refining their translations via the tree view (Figure 18) the cubes appear stacked, as can be seen in Figure 17. In Listing 26 the changes transform nodes can be seen.

```
<group>
  <transform translation="0 0 0">
    <inline>
      <Shape>
        <Appearance>
          <Material></Material>
        </Appearance>
        <Box></Box>
      </Shape>
    </inline>
    <transform translation="0,0,0">
      <inline>
        <Shape>
          <Appearance>
            <Material></Material>
          </Appearance>
          <Box></Box>
        </Shape>
      </inline>
      <transform translation="0,0,0">
        <inline>
          <Shape>
            <Appearance>
              <Material></Material>
            </Appearance>
            <Box></Box>
          </Shape>
          <Shape>
            <Appearance>
              <Material></Material>
            </Appearance>
            <Box></Box>
          </Shape>
        </inline>
      </transform>
    </transform>
  </transform>
</group>
```

Listing 25: A group with 3 nodes with all transform's translation attributes set to `0,0,0`.

```
<group>
  <transform translation="0 0 0">
    <inline>
      <Shape>
        <Appearance>
          <Material></Material>
        </Appearance>
        <Box></Box>
      </Shape>
    </inline>
    <transform translation="0,3,0">
      <inline>
        <Shape>
          <Appearance>
            <Material></Material>
          </Appearance>
          <Box></Box>
        </Shape>
      </inline>
      <transform translation="0,3,0">
        <inline>
          <Shape>
            <Appearance>
              <Material></Material>
            </Appearance>
            <Box></Box>
          </Shape>
          <Shape>
            <Appearance>
              <Material></Material>
            </Appearance>
            <Box></Box>
          </Shape>
        </inline>
      </transform>
    </transform>
  </transform>
</group>
```

Listing 26: A group with 3 nodes where two transforms' translation attributes are set to `0,3,0`, thus stacking the cubes.

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://3d.meteor.com/.

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

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

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

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

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

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

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

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

[16] URL http://paraviewweb.kitware.com/.

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

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

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

[20] URL https://developer.mozilla.org/en-US/docs/Web/HTTP#HTTP_request_methods.

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

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

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

[24] URL https://docs.angularjs.org/guide/bootstrap.

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

[26] 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.

[27] 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.

[28] 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.

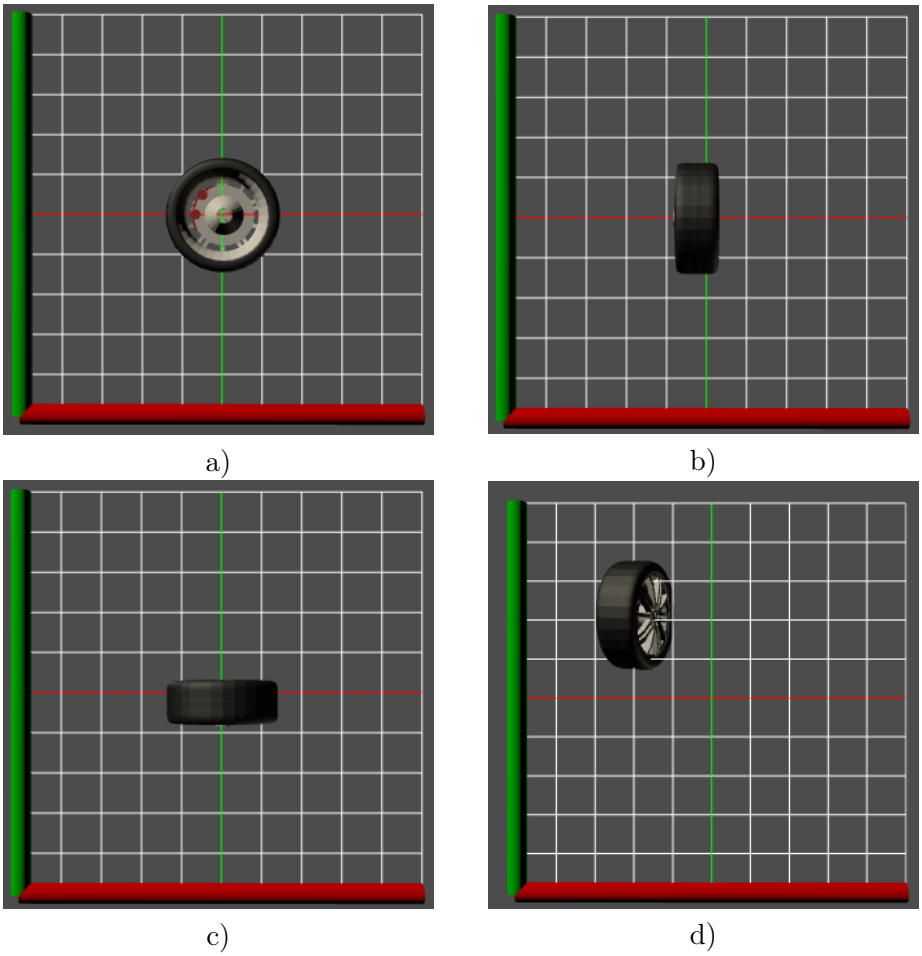
[29] Marion, Charles und Julien Jomier. Real-time collaborative scientific webgl visualization with websocket. In *Proceedings of the 17th International Conference on 3D Web Technology*, Web3D '12, Seite 47–50, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1432-9. URL http://doi.acm.org/10.1145/2338714.2338721.

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

[31] 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.

List of Listings

List of Figures



a) b)

c) d)

Figure 1: Figures 1a-c show common orientations the 3D model of a wheel could have. Figure 1d shoes the worst case.

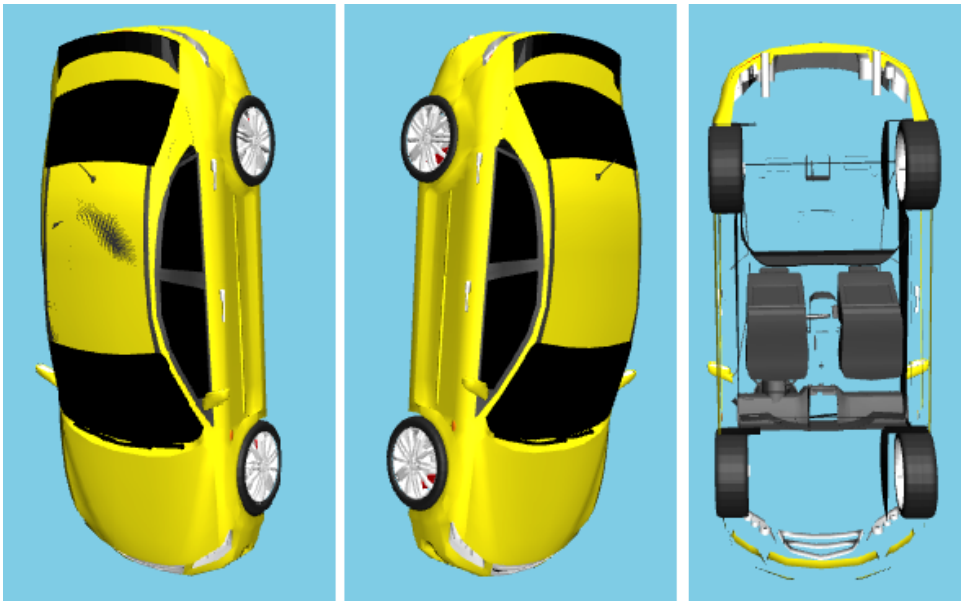


Figure 2: The rendered car from Listing 1.

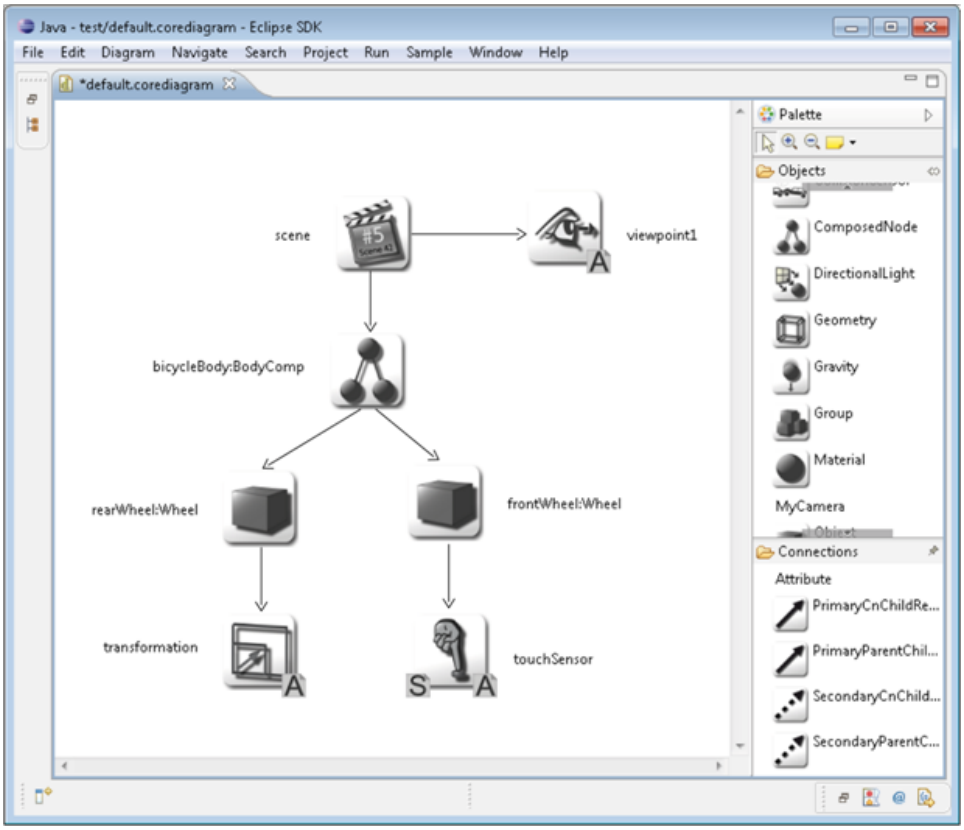


Figure 3: This Figure shows the graphical editor for SSIML. In particular it shows a scene describing a bike [21]

.

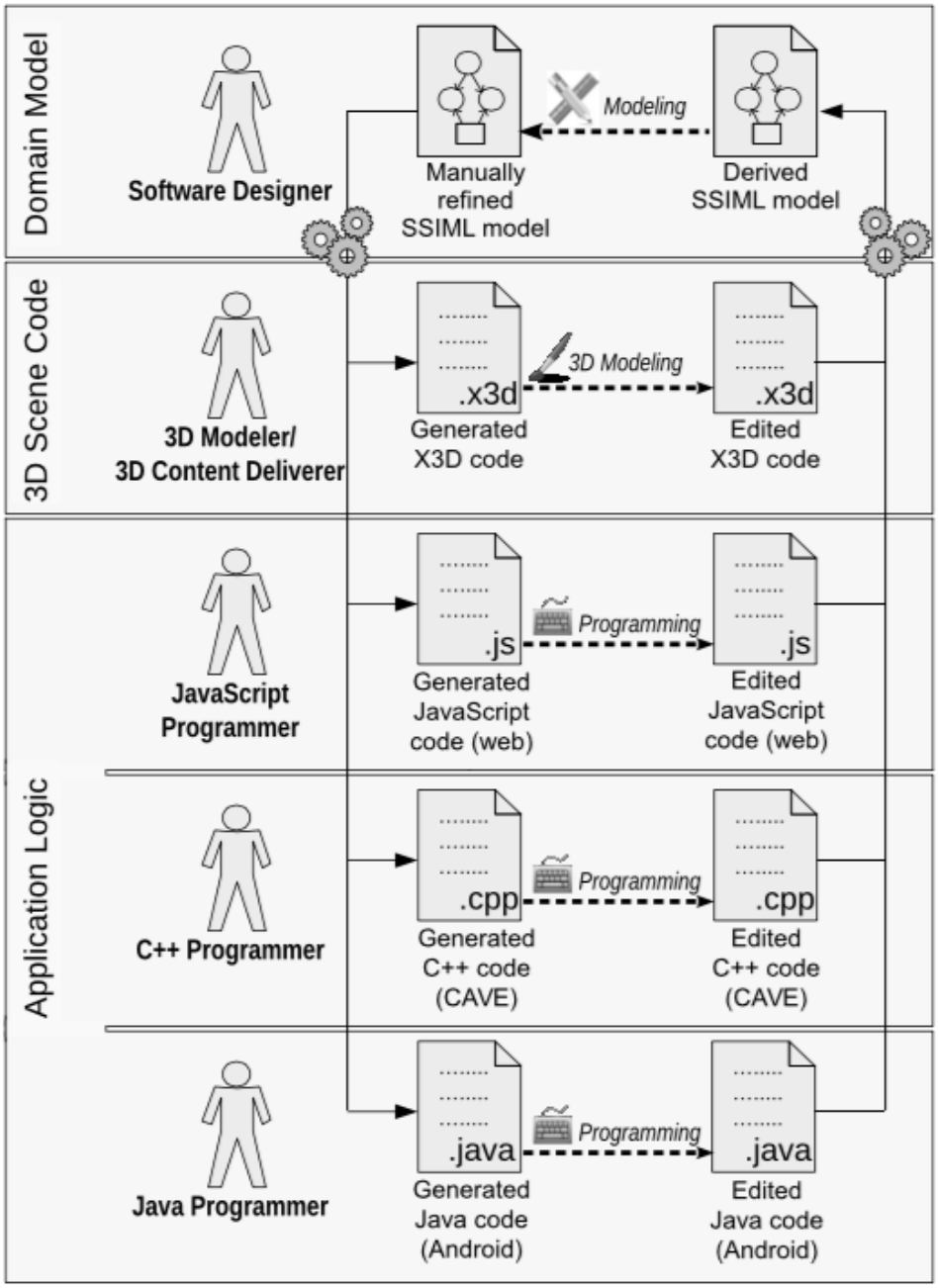


Figure 4: The development work-flow described proposed by R3D [27].

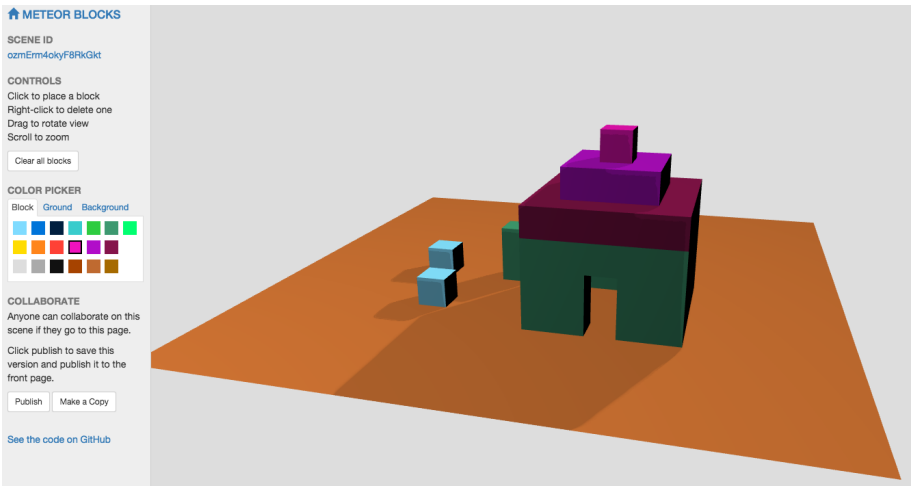
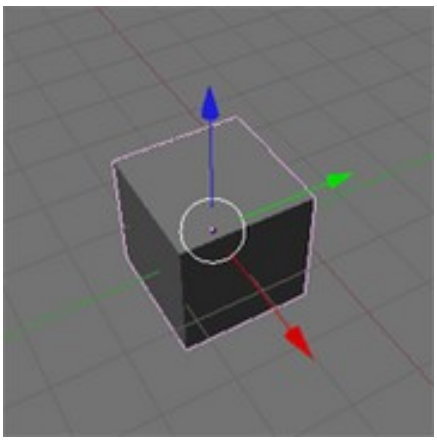
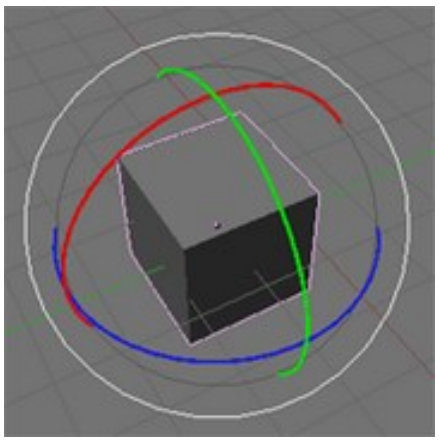


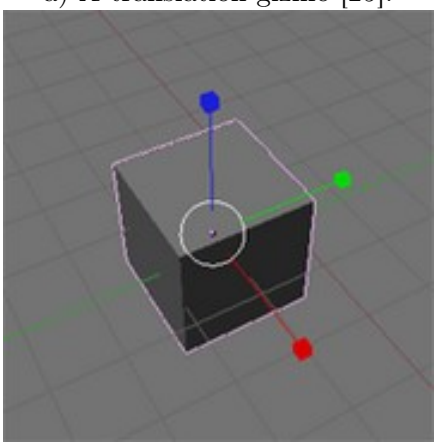
Figure 5: This is a scene in 3D Meteor showing, a house and a garden of cubes.



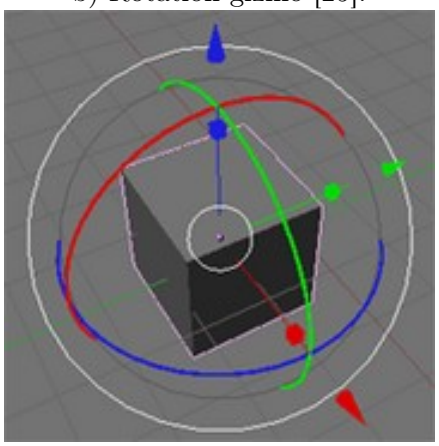
a) A translation gizmo [25].



b) Rotation gizmo [25].



c) Scale gizmo [25].



d) All gizmos together [25].

Figure 6: The same cube in Blender with different gizmos/transformers enabled.

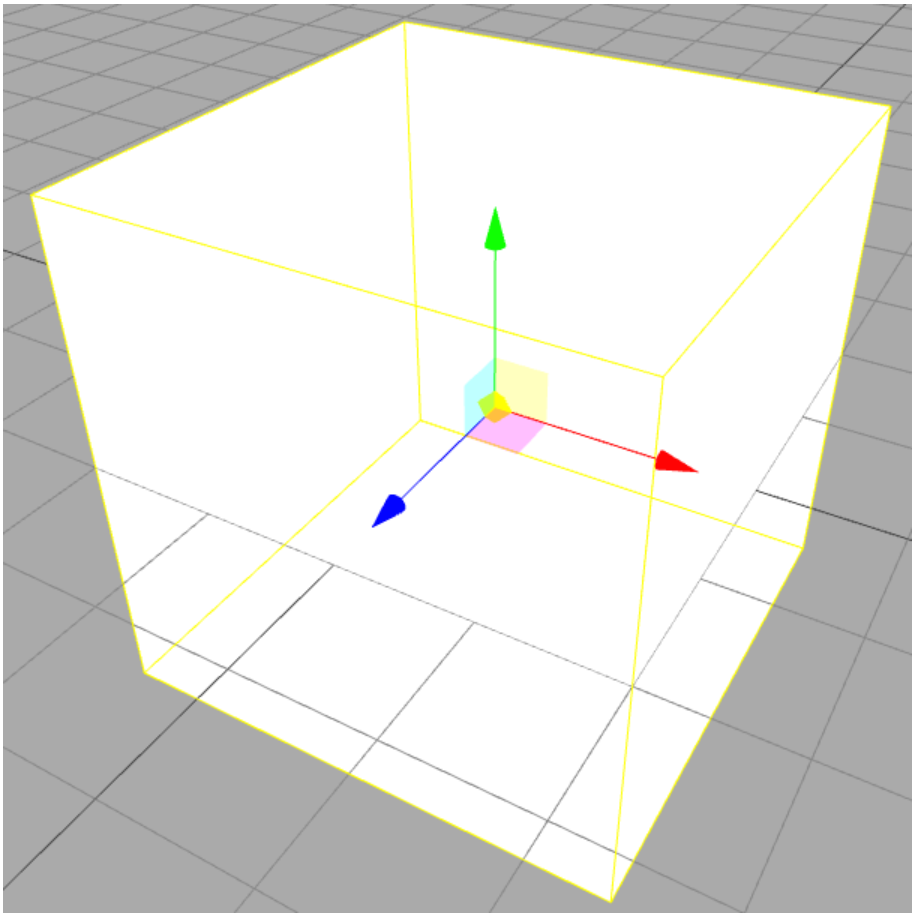


Figure 7: Translate gizmos along x, y and z axes as well as gizmos that translate the cube along the xy, xz, yz and frustum planes [22].

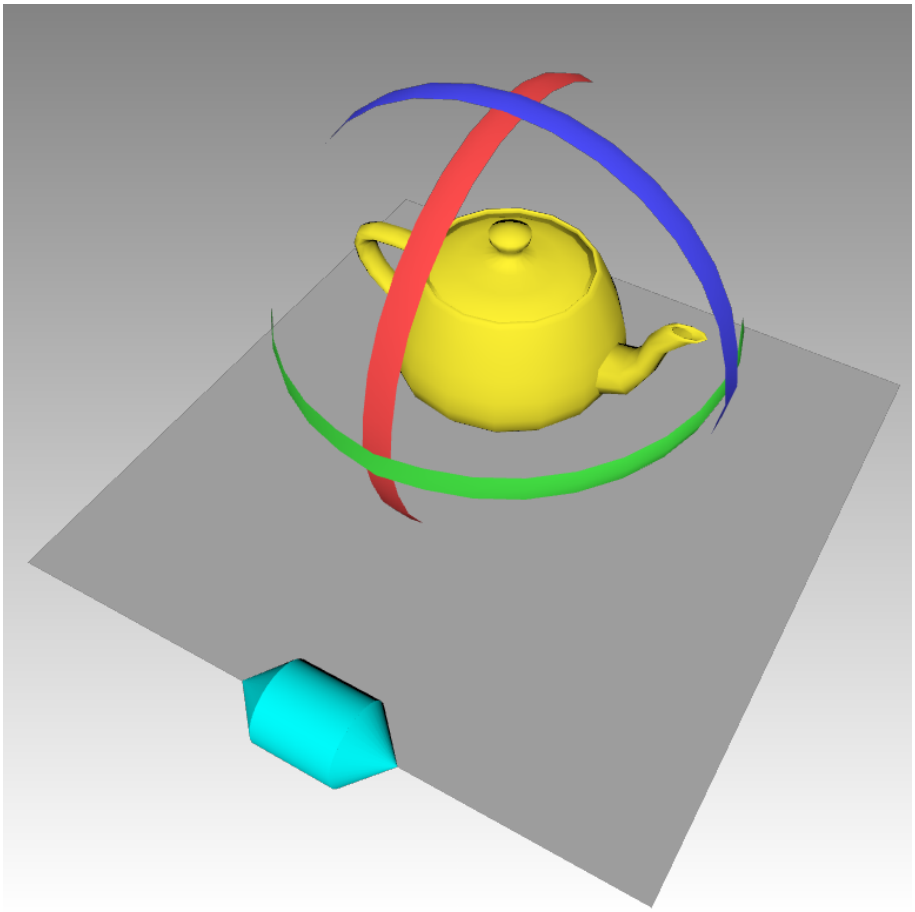


Figure 8: An official X3DOM tutorial for using X3D sensors to create gizmos [23].

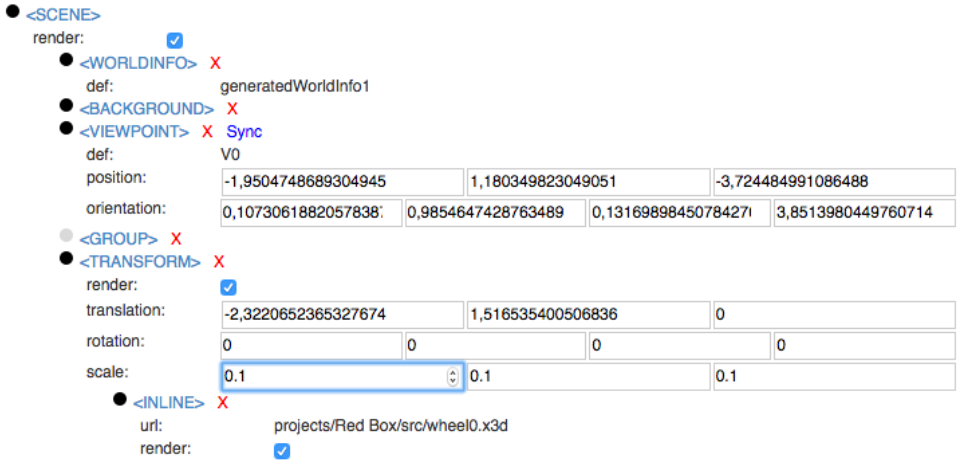


Figure 9: The rendered tree-view.

Figure 10: The X3D scene inside the DOM.

Figure 11: About half of the DOM elements that make up a tree-view of only three tree-view-nodes: a `scene`, `transform` and `inline` node.

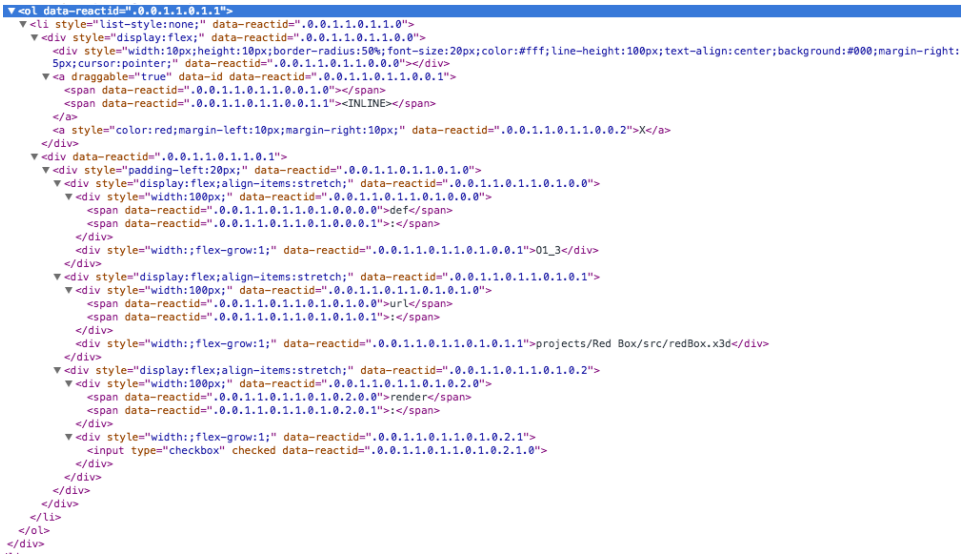


Figure 12: The DOM elements that make up a tree-view-node for an `inline`.

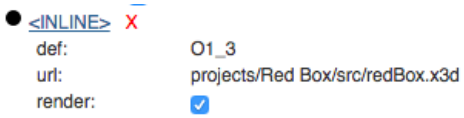


Figure 13: The rendered tree-view-node of an `inline`.

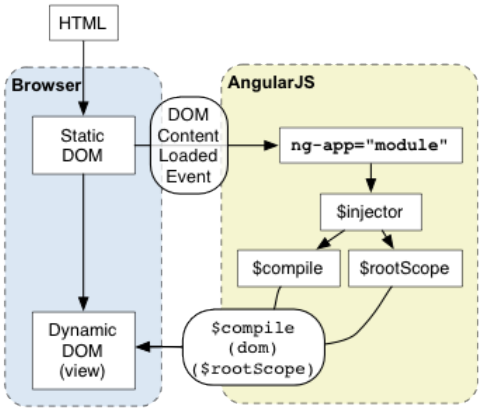


Figure 14: This is angulars bootstrapping process [24].

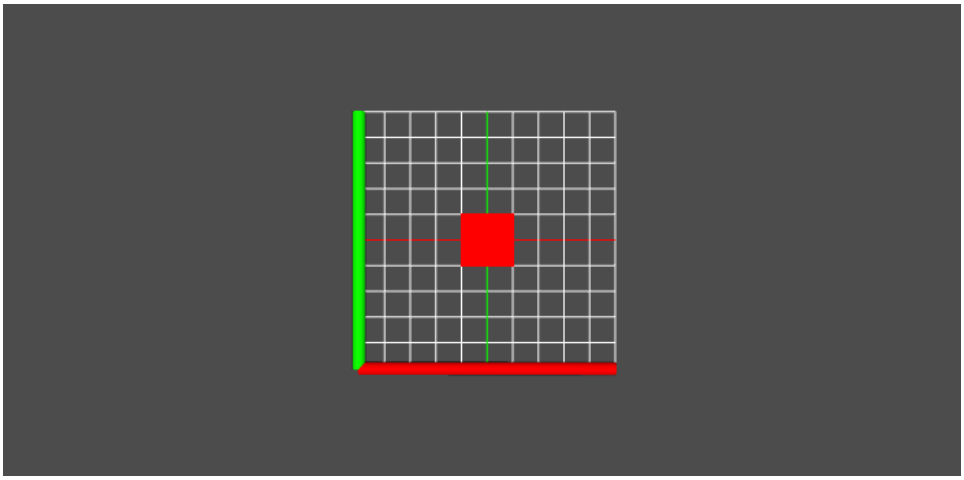


Figure 15: 3 cubes, all centered in the origin of the coordinate system.

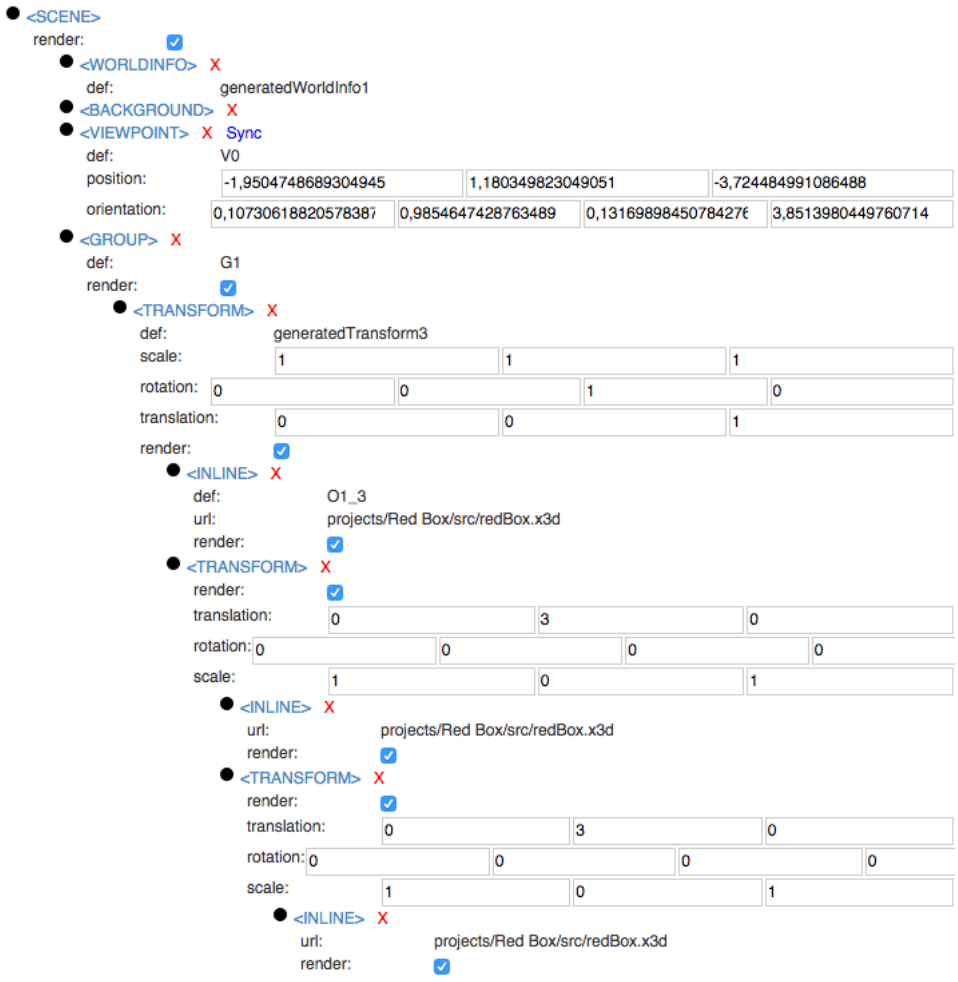


Figure 16: The tree view corresponding to Figure 15.

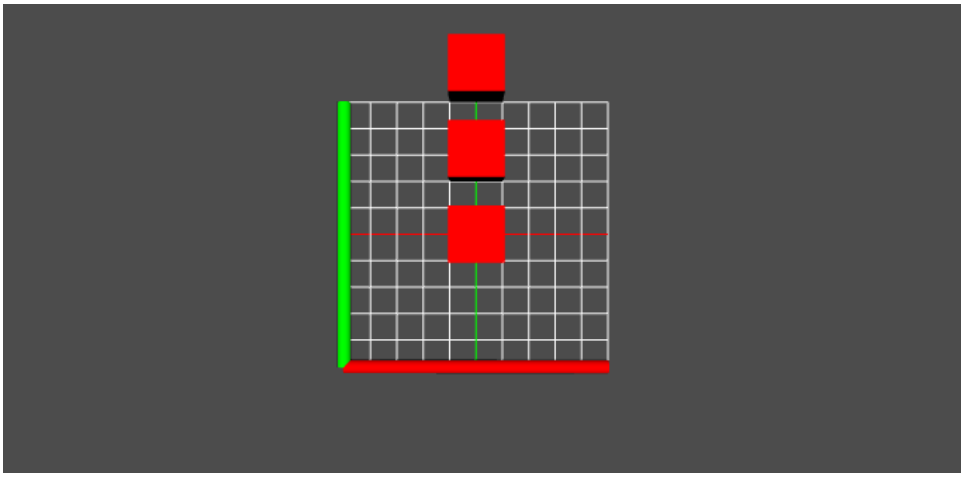


Figure 17: 3 cubes stacked.

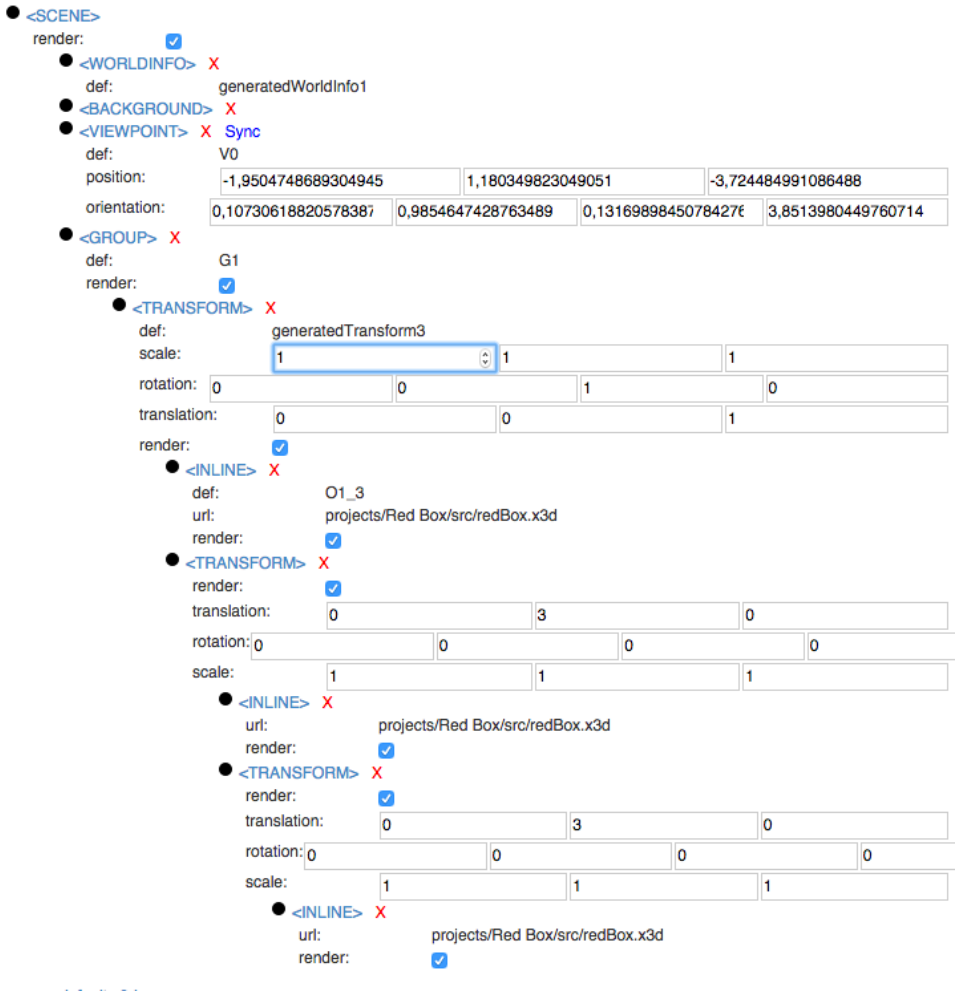


Figure 18: The tree view corresponding to Figure 17.