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 9, 2015

I, **Danny Arnold**, declare that I have read and understood the material exemplifying and explaining cases of plagiarism, and that my paper contains no plagiarized material and is solely my own work. I am aware of the fact that should my paper be found to contain plagiarized material or to have been written in part or whole by someone else, this will entail serious consequences. These include the following: 1. The reason for my failure will be recorded by the department. 2. I will not be given credit for the course. 3. The Honor Board may review my paper and suggest additional sanctions, including expulsion. Furthermore I am aware that submitting the same or a revised paper in two separate courses without the instructors' explicit consent will result in similar consequences.[1]

Freiberg, den August 9, 2015 _____

Danksagung

Danken möchte ich allen voran Prof. Jung und Matthias Lenk für dieses Seminar und all ihre Hilfe. Des Weiteren möchte ich mich bei Strötgen für sein sehr gutes Latex-Template [8] bedanken, welches dieser Arbeit zu Grunde liegt, aber in vielen Bereichen angepasst wurde. Eine weitere sehr große Hilfe war das WikiBook LATEX [7], auch vielen Dank an alle Autoren dieses WikiBooks.

Mein Dank geht auch an die fleißigen Korrekturleser Jana Starke und Felix Kästner .

Thanks!

Froitzheim Jasper Starke, Jana Kästner, Felix

Contents

1 Prelude

The goal of the thesis is to create a 3D editor, using web technologies, which enables its users to postprocess x3d scenes. These scenes are the result of a tool which was implemented as part of the DFG research project [Roundtrip3D]. The scene are automatically derived from software models . . .

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 the 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 the 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 it's own coordinate system. So they need to be translated, rotated and maybe even scaled to result in 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 it's object is simply unknown, if their is no convention for 3D modeling. ~~Usually their is a convention about origin of coord system, scaling and orientation~~ (no there isn't). 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 compose functionalities of X3D scenes

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

The following pictures demonstrate the problem:

Figure 1: Wheel 1

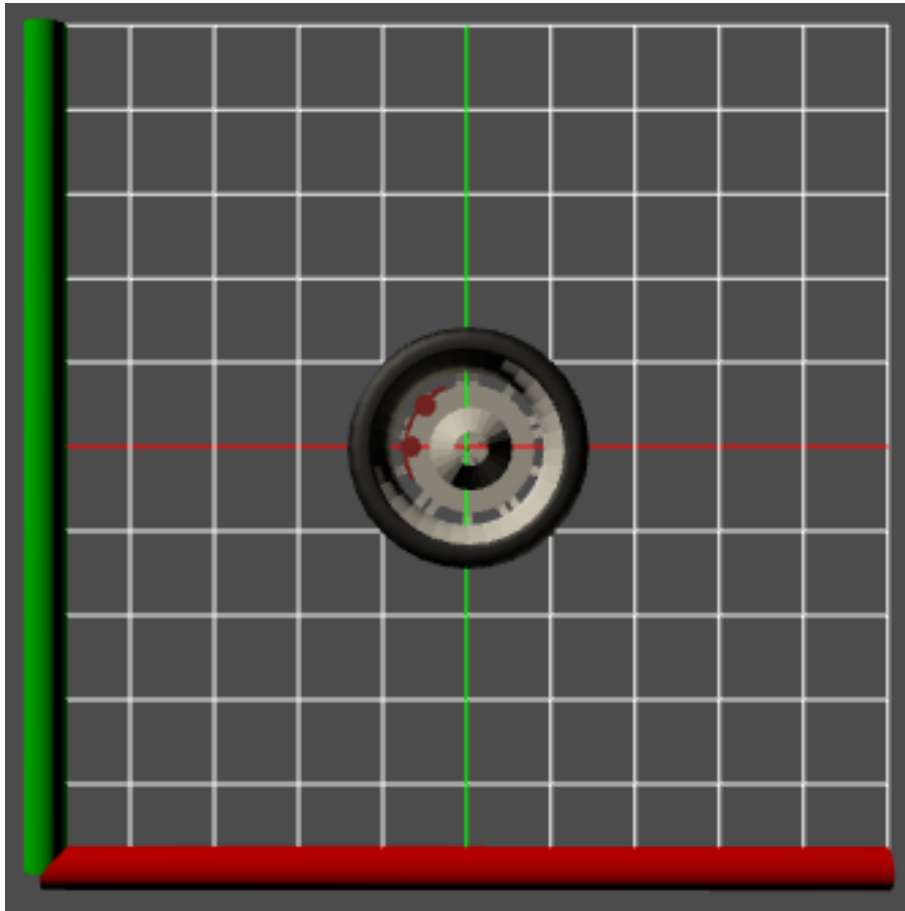These are common orientations, since it's disputable which of these could be considered be the norm.
But if one depends on art from 3rd parties, the orientation and position could be completely arbitrary:

These properties could be added and adjusted via any text editor by opening the generated x3d file, but the resulting workflow isn't user-friendly:
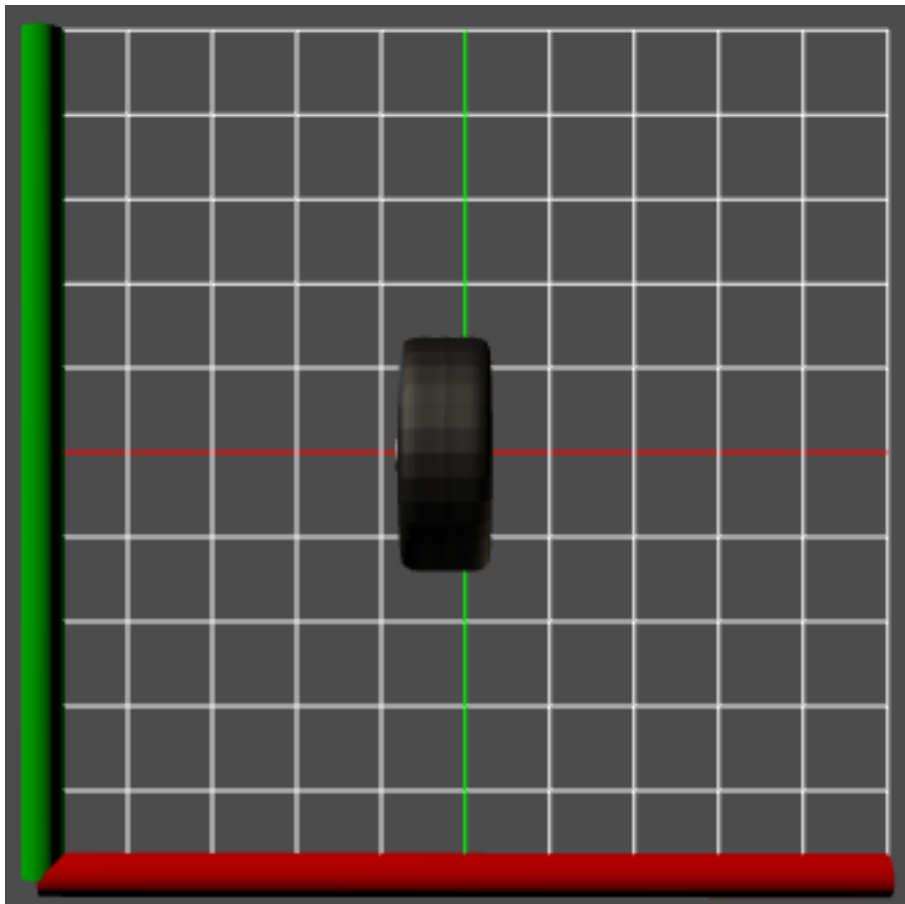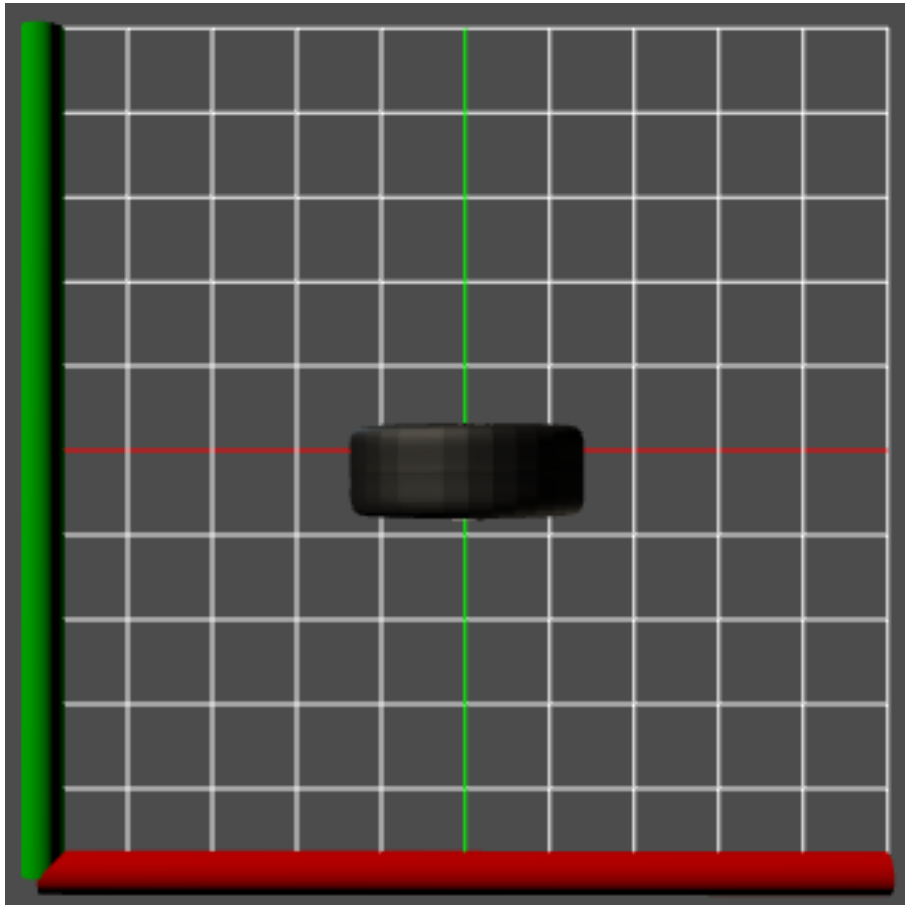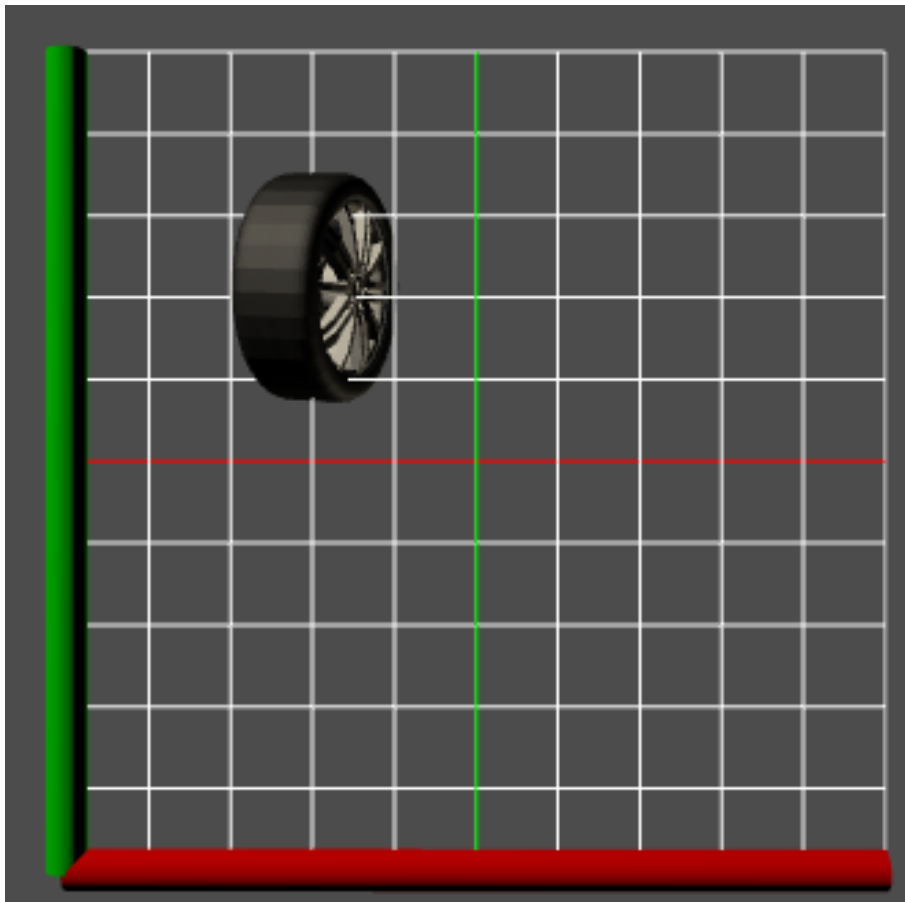
Figure 2: Wheel 2

Figure 3: Wheel 3

Figure 4: Wheel 4

1. model 3D application, including 3D scene structure
2. generate 3D scene and application code
3. run 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
5. run the application again and evaluate whether the transformations did the right thing (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 metadata 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.
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 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 children. Each child can be an object or a group, again containing more children. A group can contain associated transformation information, like <u>translation</u>, <u>rotation</u> or <u>scaling</u>. This structure has certain advantages compared to applying all transformations to the raw meshes and sending everything to the [GPU].50 (2015-07-23 15:59)

SSIML scene-graphs differ from the above definition: there are three types of nodes: - transform - geometry - group

2.1.1 Culling

Before using structures like scene-graphs, all polygon would be sent to the GPU and the GPU would need to test what polygons are actually in the view and thus need to be rendered. The problem with that approach was that this information was only known after doing a lot of calculations for every polygon already.

With a scene-graph it's possible to start from the root and traverse the graph, testing the bounding box of each group and only sending it to the GPU it it's completely visible. If it ain't, the whole sub-tree isn't sent to the GPU. If it's partially visible the same process is applied to the sub-tree.

By using a structure that retains more information about what it represents it's 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.
>
> – 50 (2015-07-23 15:59)

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 and 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's possible to reuse their information. If you have a car, it would be enough to have only one node containing the meshes for a tire, all other tires are merely addressing the tire with the geometry information:

- Group
  - Transform
    * Geomatry "Chassis"
  - Transform
    * Geometry "Tire"
  - Translate
    * Use Geometry "Tire"
  - Translate
    * Use Geometry "Tire"
  - Translate
    * Use Geometry "Tire"

That way the memory footprint of an application can be reduced.

2.1.4 X3D

X3D is the [XML] representation of [VRML] which was designed as a universal interactive 3D exchange format, much like 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's called x3dom and it's extensively used by SceGraToo, the tool that arose from this thesis.

x3dom   As said in the previous chapter 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 to entirely different approaches to describe the same thing: * imperative * declarative

The following matrix classifies x3dom together with other common web technologies 40 (2015-07-23 13:00):

	2D	3D
Declarative	[SVG]	X3DOM
Imperative	[Canvas]	[WebGL]

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

2.2   SSIML

Heterogeneous developer groups, groups that are comprised of people from different backgrounds (programmers, 3D designers) have hard time working together. GF15

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.

2.3   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 n)

Roundtrip3D was 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
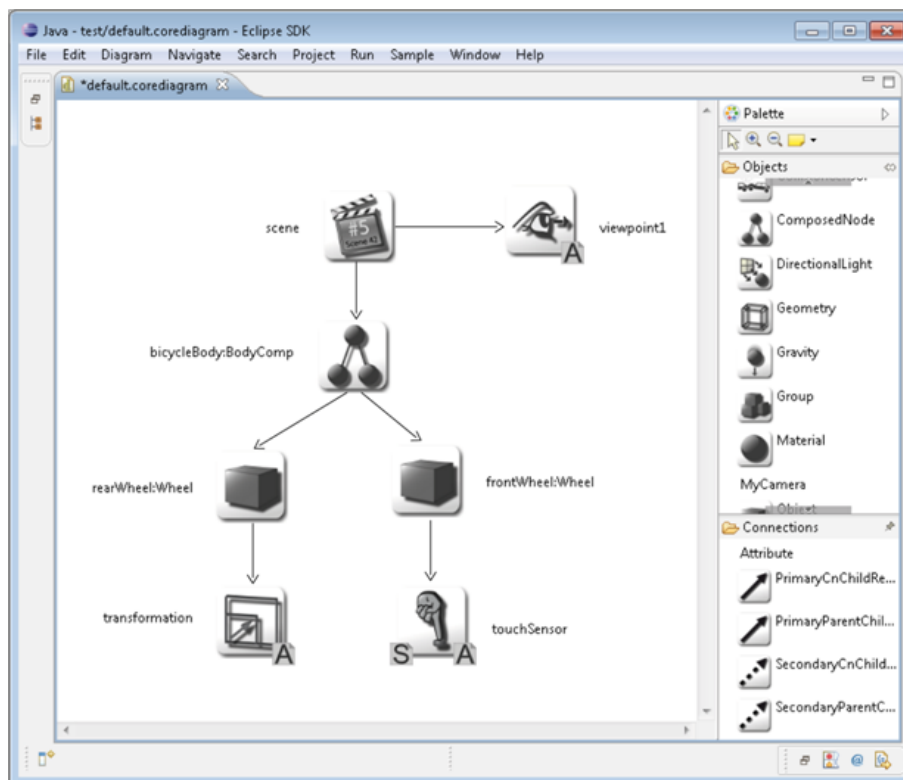
Figure 5: A SSIML Diagram [2]

changes), all working copies are regenerated and delivered to the individual developers. After each roundtrip every developer has a copy of the project that is consistent with everyone else's.

2.3.1 states

TODO: * explain the DOM * move to

Using something imperative like backbone it would be necessary to create a model (copying the information already in the DOM), a view (rendering that information to the DOM again) and a controller keeping track of the state changing the model where necessary and rerendering the view, and also keeping track of all it's children and removing them when they disappear or creating new ones whenever a new X3D element appears.

2.3.2 Functional Programming

All SceGraToo does is listen to changes of the X3D node and whenever it changes, may it be an attribute that changes or nodes that are added or removed, it calls one function that evaluates the X3D node (basically traversing it) and return the new structure. That structure is diffed against the tree-view that is already in the DOM and react calculates the minimal changes necessary to make the tree-view's old structure coherent with the newly calculated one.

2.4 Koa

TODO: move to implemenation Koa is pretty much boring unless I'd explain node.js's take on asynchronicity, callbacks, promises and then generators. Still, the server is way to simple to be of any interest for this project.

2.5 Related Work

2.5.1 Collaborative Work

2.5.2 3D Meteor

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 apps are comprised of a client side and a server side. The client can subscribe to database collections and get's automatically
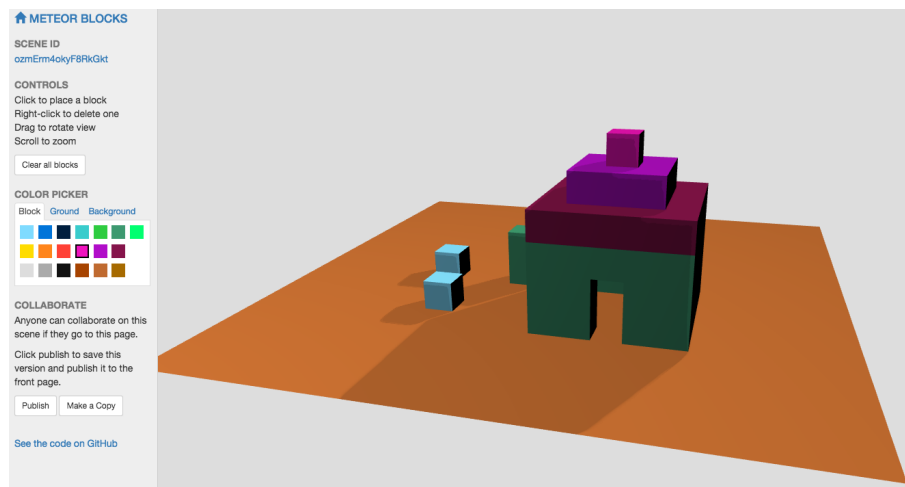
Figure 6: 3D Meteor

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.

2.5.3 Blender Plugin

I don't do anything about collaborative working anymore. So yeah.

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 to wiremeshes and store them on a server. Another artist can download these changes and apply them to his working copy. The diffs are a simple list of vertices and their movement in the x, y and z space.

```
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]
```

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's 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.

2.5.4   3D Widgets

Tilt Brush   Tilt Brush was lauded for it's 3d interface.

2.5.5   Gizmos

Gizmos, also called manipulators, are handles or bounding boxes with handles that manipulate it's containin object in a predefined way when dragged. [?]

In X3D gizmos can be realized with on of X3DDragSensorNode's [3] decedents.

SphereSensor SphereSensor converts pointing device motion into a spherical rotation around the origin of the local coordinate system. xd3om1

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. x3dom2

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. x3dom3

The sensors track drag events on their siblings. In the example in figure 12 (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.

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 adjust.

Having the the handles being 3d objects within the scene, that look touchable and intractable, make it easier for users find their way around the application. Instead of haven to learn keyboard shortcuts the user simply uses their intuition about how she would 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>
```
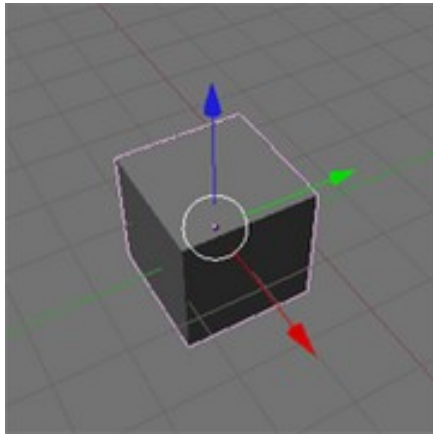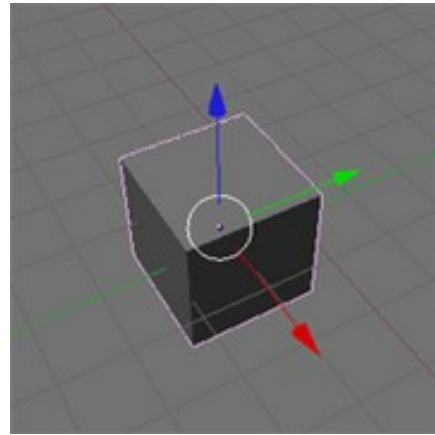
Figure 7: translation gizmo [4]
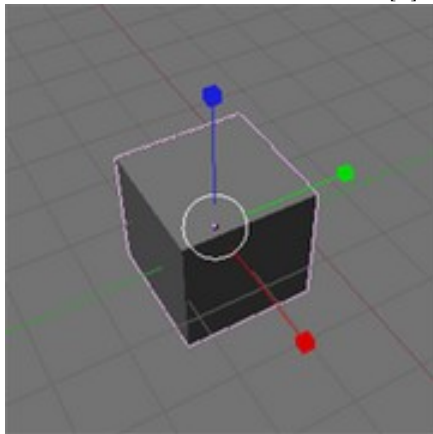

Figure 8: rotation gizmo [4]
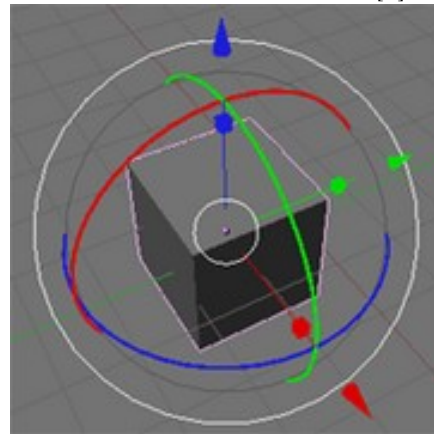

Figure 9: scale gizmo [4]
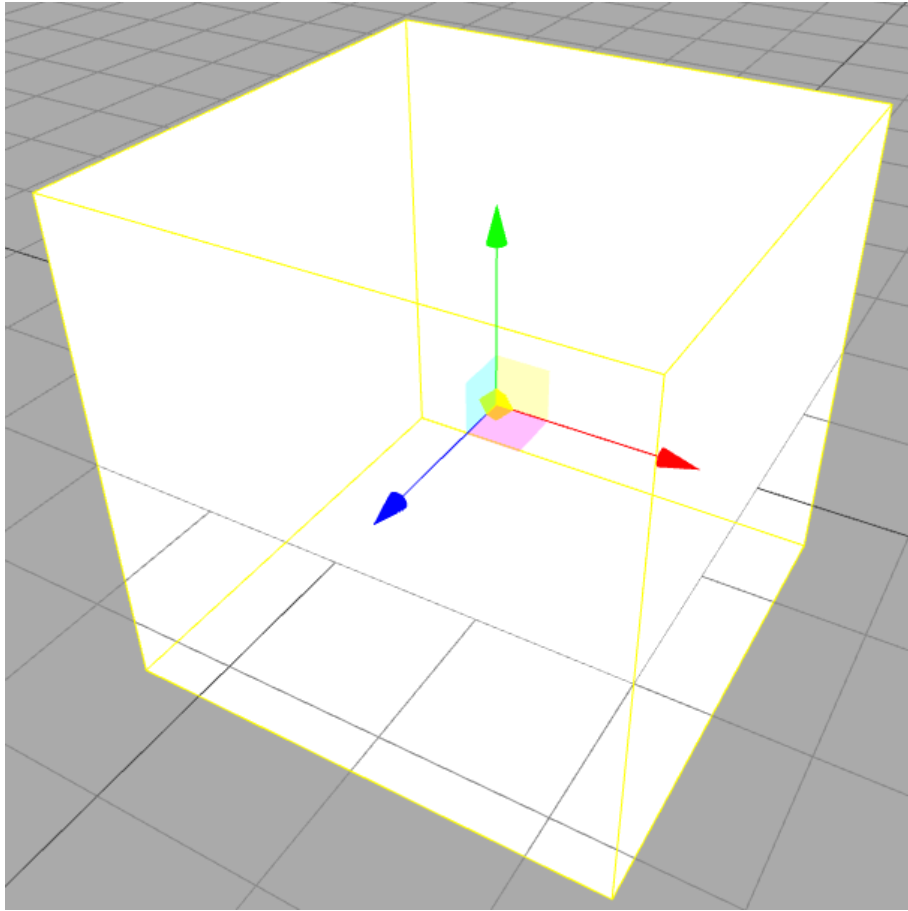

Figure 10: all gizmos together [4]

Figure 11: Shows translate gizmos along the x, y and z axis als well as gizmos that translate the cube along the xy, xz, yz and frustum plane. [5]
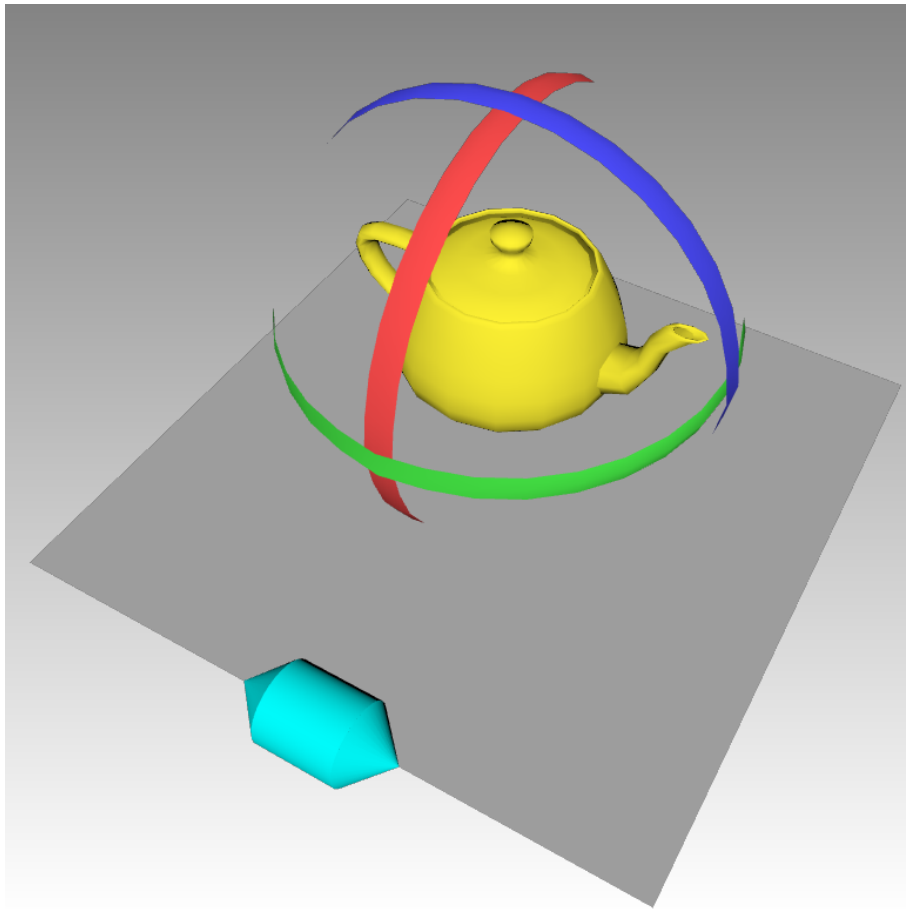
Figure 12:   Shows an official x3dom tutorial for using sensors to create gizmos. [6]

2.5.6   Component Editor

- release 12th of june 2015
- thus after the work on SceGraToo started
- developed over a year
- developed by 3 people working part time
- cannot save or load x3d files
- serealizes the scene into a JSON format

```json
{
  "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"
    }
  }
}
```

- this would loose meta information like the IDs in comments.

2.5.7   Real-Time Collaborative Scientific WebGL Visualization with Web Sockets acm

Using web sockets instead of AJAX is an interesting approach. Especially the cut down on latency. It is over all very similar to the approach I was thinking about. Except that they propagate only some information, like the camera position and view angle. In SceGraToo's case the whole scene needs to be synchronized. So to have specttors like they have we either have to send the whoe model to the server after eacht change or use some kind of tree diff algorithm. They visualize a specific dataset in a specific format (which?). We only need to visualize X3D data, so there is no reason to use some generic Visualization tool kit, x3dom does a pretty good job doint this. on the other

hand we not only have to visualize the data, but also manipulate it and this is way easiear in a descriptive scene graph instead of manimpulatiing webgl scenes or data formats like vtk.

2.5.8 ParaViewWeb pvweb

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 sadly 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

skizze wie dom rendering funktioniert oder auch nicht, da es ziemlich uninteressant ist

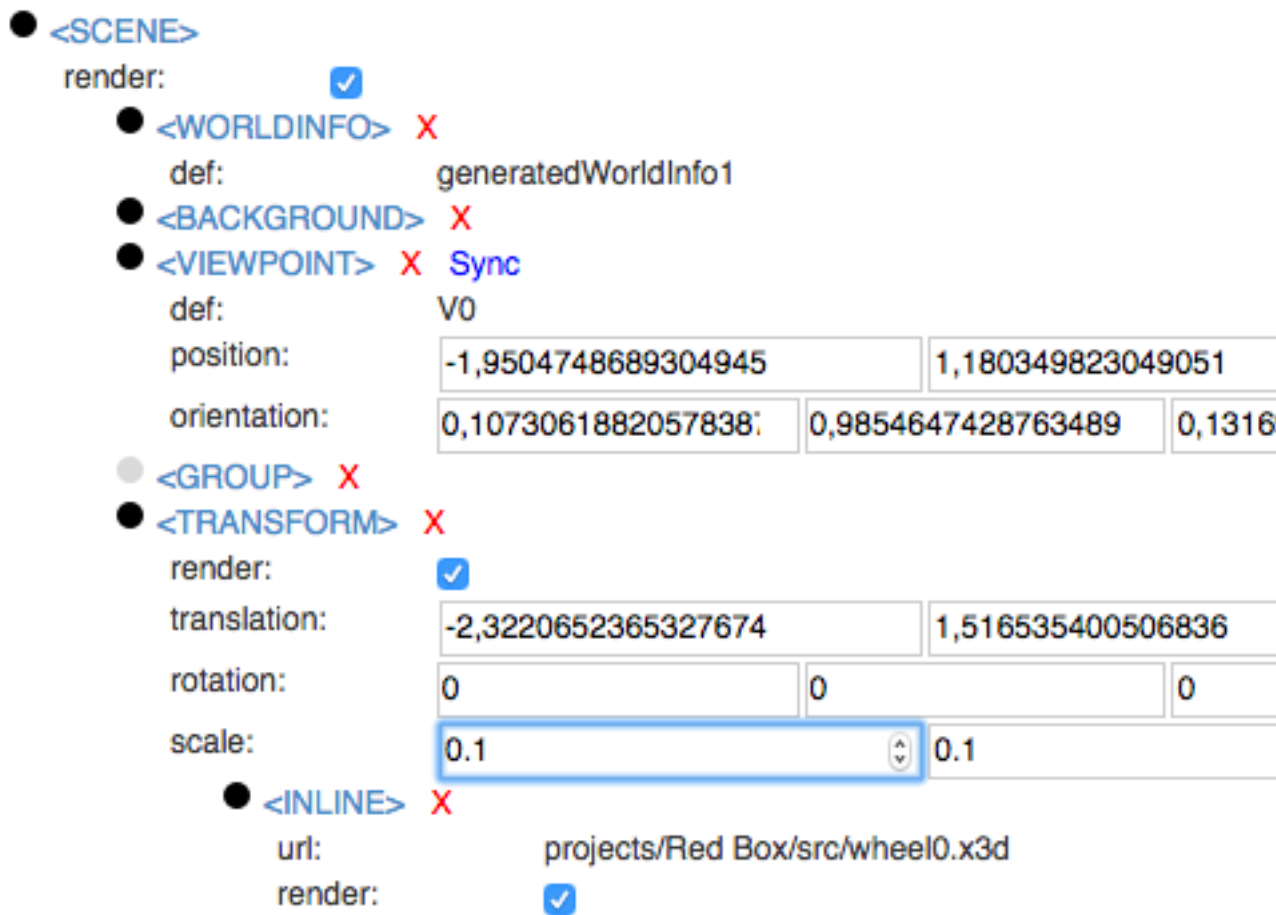## 3.1 How

3.1.1 Server

3.1.2 Client

Figure 13: treeview

Synchronization Process

"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

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

The scene-graph 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 theses requirements:

1. custom html elements as part of tree nodes (multiple checkboxes or multiple inputs)
2. ability to observe the tree node's state changing
3. binding to an arbitrary model, that can recover inconsistent state

with mixed results:

Partially not met requirements:

- custom elements as part of tree nodes
- ability to listen to changes to the tree node

Requirements none of the tested tools met:

- binding to an arbitrary model, that can recover inconsistent views

None of the off-the-shelf solutions could satisfy all expectations.

The most complicated part is keeping the tree-view-controller in sync with the scene-graph while the scene-graph is being modified and vice versa.

Rerendering and Diffing   Terminology
scene-graph:
The HTML/XML/X3D representation of the scene:

```
<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 def="generatedWorldInfo1" title="Orgel"
↪    info="">
    </worldinfo>

    <background skycolor="0.3 0.3 0.3" groundcolor=""
↪    groundangle="" skyangle="" backurl="" bottomurl=""
↪    fronturl="" lefturl="" righturl=""
↪    topurl=""></background>

    <viewpoint def="V0" fieldofview="0.7"
↪    position="-1.9504748689304945 1.180349823049051
↪    -3.724484991086488" orientation="0.10730618820578387
↪    0.9854647428763489 0.13169898450784276
↪    3.8513980449760714" centerofrotation="0 0 0"
↪    znear="-1" zfar="-1">
    </viewpoint>

    <!-- id=8d3f0a8a-b6d7-4acc-922b-ea59364443fa -->
    <group def="G1" render="true" bboxcenter="0,0,0"
↪    bboxsize="-1,-1,-1">
      <transform def="generatedTransform3" scale="1 1 1"
↪      rotation="0 0 1 0" translation="0 0 0"
↪      render="true" bboxcenter="0,0,0"
↪      bboxsize="-1,-1,-1" center="0,0,0"
↪      scaleorientation="0,0,0,0">
        <!-- id=8459b736-5a9d-4688-b624-e519857a92fd -->
        <inline def="O1_3" namespacename="O1_3"
↪        url="projects/Red Box/src/redBox.x3d"
↪        render="true" bboxcenter="0,0,0"
↪        bboxsize="-1,-1,-1" load="true">
          <Shape render="true" bboxCenter="0,0,0"
↪          bboxSize="-1,-1,-1" isPickable="true">
            <Appearance sortType="auto"
↪            alphaClipThreshold="0.1">
              <Material diffuseColor="1 0 0"
↪              ambientIntensity="0.2"
↪              emissiveColor="0,0,0" shininess="0.2"
```

tree-view-controller:
Loosely defined term that comprises all objects, methods, functions, event-listeners and callbacks related to parsing the scene-graph and creating the initial tree-view-view.

tree-view-node-controller: Loosely defined term that comprises all objects, methods, functions, event-listeners end callbacks related to keeping a subtree of the scene-graph up to date.

tree-view-view:
The HTML representation of the tree-view-controller (html output shortened and simplified):

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

The aim is to keep the tree-view-view a consistent representation of the scene-graph. The tree-view-view only shows specific nodes, I can't list them here because they will change over the

where specific scene-graph nodes, ones that , and their properties are presented in an up to date and editable form.

todo: explain what nodes and what properties and why

First design approach is to let the tree-view-controller create an initial tree-view-view by traversing it and creating tree-view-node-controllers ad hoc. These tree-view-node-controllers create the corresponding tree-view-node-views while traversing the scene-graph. For each scene-graph node create a corresponding tree-view-view node. Each tree-view-node-controller observes its corresponding scene-graph node for attribute mutations and added or removed child nodes and change it's view accordingly.

Depending on how the scene-graph is mutated 3 main cases can be differentiated.

a scene-graph node is added
  a new tree-view-node-controller is instantiated and renders a new tree-view-node-view
a scene-graph node is deleted
  the corresponding tree-view-node-controller is destroyed
a scene-graph node is mutated
  the corresponding tree-view-node-view is altered

Tree-view-node-views can also be used to edit a scene-graph node's properties.
When a tree-view-node-views is edited it's tree-view-node-controller is notified and applies the new properties to the corresponding scene-graph node. It's assumed that the updates will always lead to consistent state, where the scene-graph and the tree node converge.

The synchronization process has no ability to detect if updates lead to a consistent state. It also has no ability to recover from an inconsistent state, though without the ability to detect inconsistencies this does not really matter.

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 prop-

erties 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 4 things have to implemented:

1. code for parsing the the scene-graph

2. code to generate the tree-view-node-view

3. code to synchronize changes from the scene-graph to the tree-view-node-view

4. code to synchronize changes from the tree-view-node-view to the scene-graph

This design bears more complexity than reparsing and regenerating the complete tree-view-view on every change.

Rerendering solves Problem 1 and reduces problem 2 to two steps:

1. code for parsing the scene-graph and generating the tree-view-node-view

2. code to synchronize changes from the tree-view-node-view to the scene-graph

Rerendering everything on every change is usually inefficient. React is used to circumvent this problem. React calculates a lightweight representation of what is going to be rendered to the DOM and compares that to what is already rendered. It calculates a set of patches and only applies these to the DOM.

The complete rerender of the view happens only im memory and is never sent to the DOM and thus never

The code below is for explanation purposes and does not resemble react's implementation in any way.

Old Virtiual DOM:

```html
<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>
```

New Virtual DOM:

```html
<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>
```

Patch:

```javascript
var li = document.createElement('li')
li.innerText = 'group'

↪  document.querySelector('[data-reactid=".0.0.0"]').appendChild(li)
```

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

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 I'm just concerning angular directives, 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 `treenode` renders the node itself and a `nodelist`, that renders a list of tree-nodes for each child node. Mind the recursion.

```
(car (cons 1 '(2)))
```

Listing 1: Example of a listing.

Listing 1 contains an example of a listing.

The data:

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

Listing 2: Example of a listing.

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

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

```
<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>
```

```
<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>
```

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

The scene-graph is traversed and for each eligible child node a new `treenode` is created. The double curly 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.

3.1.3 Interaction

4   Implementation

4.1   Used Tech

angular

- dependency management
- resource management
- less dynamic templates

react

- highly dynamic views (like the tree view)

4.2   Problems

synchronization Process   This design leads to brittle code that is hard to maintain and hard to adapt to new use cases.

The first culrpit in this design is that the tree-view-controller mutates itself.

Let's externalize the mutation observation and update capabilities into another actor: a mutation observer.

The mutation observer can map each graph node to its corresponding tree node and thus change the tree node to represent the matching graph node again.

The tree-view-controller does not mutate itself anymore. All the mutation logic that synchronizes the tree-view-controller with the scene-graph is in the mutation observer. But the scene-graph and the tree-view-controller can still diverge, since moving the mutation logic from the tree-view-controller to the mutation observer does not make it easier to reason about and thus less error prone.

Assuming that the first operation (traversing the scene-graph and creating the tree-view-controller) is correctly implemented and resilient, the easiest way to make sure the tree-view-controller and the scene-graph are in sync would be to recreate the tree-view-controller whenever the scene-graph changes.

At first sight that seems to be awfully slow, that needn't to be true. Instead of recreating the the whole tree using [DOM] elements, a model could be created using simple javascript objects. This virtual tree-view-controller

can then be compared to the tree-view-controller already in the [DOM] and only the changes need to be applied.

That might sound like it's no easier than doing the synchronization manually like in the first diagram, but the diff algorithm doesn't actually need to know what it's diffing. Meaning it could be developed and tested once and be used by all kinds of projects. That's one thing react provides. The rendering pipeline looks finally like this:

React calls the virtual representation of what will be rendered virtual DOM. The other important feature of react is it's way to build the virtual DOM. A is a factory that returns A virtual DOM node is the return value of component's render function, the render function can nest other virtual [DOM] nodes in its return value.

A quick examples:

```
// TreeNodeAttributeList :: [Attribute] -> div
var TreeNodeAttributeList = React.createClass({
  render: function () {
    // the scene-graph node that was passed 'createElement'
    ↪ by the caller
    var node = this.props.node;

    var whitelist = ['def', 'diffusecolor', 'orientation',
    ↪ 'position', 'render', 'rotation', 'scale',
    ↪ 'translation', 'url'];

    var attributesToRender = node.attributes.filter(function
    ↪ (attribute) {
      return
      ↪ propsToRender.includes(attribute.name.toLowerCase());
    })

    return (
      <div>
        {
          attributesToRender.map(function (attribute) {
            return <TreeNodeAttribute attribute={a}
            ↪ owner={node}/>
          })
        }
      </div>
    );
  }
});
```

The usage of HTML tags is just syntactic sugar, it's transpiled into:

```javascript
'use strict';

// TreeNodeAttributeList :: [Attribute] -> div
var TreeNodeAttributeList = React.createClass({
  render: function render() {
    // the scene-graph node that was passed 'createElement'
    ↪  by the caller
    var node = this.props.node;

    var whitelist = ['def', 'diffusecolor', 'orientation',
    ↪  'position', 'render', 'rotation', 'scale',
    ↪  'translation', 'url'];

    var attributesToRender = node.attributes.filter(function
    ↪  (attribute) {
      return
      ↪  propsToRender.includes(attribute.name.toLowerCase());
    });

    return React.createElement(
      'div',
      null,
      attributesToRender.map(function (attribute) {
        return React.createElement(TreeNodeAttribute, {
        ↪  attribute: a, owner: node });
      })
    );
  }
});
```

This is code is the a simplified version of the code that renders a graph's nodes attributes into the tree-view-controller. This component simply decides what attributes should be rendered into the tree. `TreeNodeAttribute` is another component that will renders different elements depending on what attribute is passed in.

Because the outputted html is only a function of its input it easy to parse the scene-graph: 1. choose a graph node as the root 2. call the node

component with that graph node 3. if the graph node has child nodes call the node component again with each child node and return their return values wrapped in an element 4. if the graph node has no children return an empty element

4.2.1 Why the chosen approach will work

After trying different solutions it turned out the a functional solution would also suit SceGraToo the best. First Chaplin (a backbone successor) was used to implement SceGraToo, but soon the first version of it suffocated under it's own complexity (probably also due to the incompetence of its user - me). It turned out that MVC had serious flaws when trying to use it to describe an ever changing declarative scene-graph. The naive solution would be to observe the X3D node and rerender the whole tree structure whenever it changed, that would also mean to rerender every time an attribute is changed. To make that clear, that means rerendering every time an object is moved with the mouse, since the translation is an attribute. This thesis will not contain any benchmarks proving that manipulating the DOM from javascript is slow, rather than that it is left as an exercise to the reader to research it if she wants. React solves this issue quite elegantly by creating the dom structure in javascript and diffing it with the [DOM], only applying the minimum of changes to the DOM to realize the corresponding result. That made it possible to use the X3D node of the DOM as the only source of truth and minimize the state that needs to be kept to make the tree-view-controller work.

A solution that handles state manually can work, but requires more discipline and a more complicated mental model. The programmer has to keep all side effects and cascading effects in mind when changing parts of the program.

5 Results

5.1 Why Scegratoo is the best since sliced Bread

List of Tables

List of Figures

Bibliography

[1] URL `http://www.as.uni-heidelberg.de/studium/downloads.php`.

[2] URL `http://vr.tu-freiberg.de/roundtrip3d/`.

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

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

[5] URL `http://threejs.org/editor/`.

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

[7] Latex, 04 2012. URL `http://en.wikibooks.org/wiki/LaTeX`.

[8] Strötgen, Jannik. Latextemplate für die seminararbeit, 05 2010. URL `http://dbs.ifi.uni-heidelberg.de/fileadmin/docs/`.

Index