Form controls in WebGL

A stepping stone to a WebGL library for developing commercial interactive 3D websites

Anton Tenchev Zhelyazkov

One-year master program Software Engineering

14-08-2015

Thesis supervisor: Tijs van der Storm

Company supervisor: Bas Wouwenaar

Organisation: Ohpen

University of Amsterdam

Amsterdam, The Netherlands

Table of contents

Summary

Web development has been limited by the capabilities of HTML, CSS and JavaScript. Although, web developers and designers have been trying to achieve impressive visual effects using animations, curved corners and images to simulate depth the results have been far from what has been achieved in desktop applications and games. With the introduction of WebGL in 2011, real, hardware accelerated 3D support in the browser was added to the toolbox of the modern front-end developer. The existing libraries which abstract the low level WebGL API, however, do not support all the features needed to build a full 3D website.

In this project we tried to answer the question of how and which of the existing 3D WebGL libraries should we use to create a set of interactive 3D form controls. These controls can later serve as a stepping stone for the creation of a full 3D web development framework.

We used a design research method where a prototype was built in two 3D WebGL libraries, namely Three.js and SceneJS. The number of achieved features, performance and source lines of code were measured for each implementation and later compared and analyzed. Our prototype design was based on the TODOMVC.COM web apllication which is often used as a base for comparing development frameworks.

Our measurements showed that Three.js managed to achieve more features of the prototype design with less code. Qualitative analysis also showed that Three.js is more intuitive to use. On the other hand, the implementation of the prototype with SceneJS achieved more than twice as better frame rate as the Three.js implementation. In conclusion we decided that the overwhelming performance advantage of SceneJS is more important than the few missing modules in its feature set since these missing functionalities can be added more easily than Three.js can be made faster.

Preface

This research served as the fulfilment of the thesis project requirement of the Software Engineering master program in the University of Amsterdam, The Netherlands in 2015. My interest in the latest technologies in front-end software development was the reason for the choice of this topic.

I would like to thank my project supervisor at the University of Amsterdam, Tijs van der Storm and my supervisors at Ohpen, Bas van Oijen and Bas Wouwenaar for their advice and feedback during the project.

Chapter 1 - Introduction

The purpose of this research is to serve as a stepping stone to creating a 3D website development framework using WebGL. The project focused on finding the best way to implement interactive 3D form controls using existing 3D WebGL libraries.

In the second chapter we give the motivation behind the project, then in the next chapters some background and context information is provided and similar research projects are discussed. Next, the structure and execution of the research is described and finally results of the proof of concept implementations are presented and analyzed.

Chapter 2 - Problem statement and motivation

Empirical data has shown that 3D visualizations improve attractiveness and effectiveness of applications in several domains such as entertainment, e-commerce and e-learning [BLV'11][CR'02]. In particular, in the field of e-learning it has been acknowledged that 3D virtual reality can serve as the bridge between theory and practice and help with better absorbing the learning material [ARBAB'14][HO'95]. Similarly, 3D web applications can be easier to understand and learn to use compared to their 2D counterparts. The combination of web and 3D technologies also can significantly reduce the price of developing 3D virtual reality systems which have traditionally been associated with high costs due to special software and hardware requirements [ARBAB'14][CR'07].

Website designs have been limited by the capabilities of HTML, CSS and JavaScript. Although HTML was originally intended for creating static documents, the web community has pushed the boundaries of the existing technologies to create impressive, dynamic and interactive web content. Website designers have been trying to create more realistic web sites using curved corners, gradients, shadows, sliding panels, animations and backgrounds to simulate depth and perspective. These efforts, however, have been far from the 3D visualizations which have been achieved in desktop applications and games. Access to hardware accelerated graphics via the OpenGL and DirectX libraries has given desktop software an edge in interface design. There have been attempts to do the same in the browser with X3D, VRML and Flash being some of the more popular web 3D technologies [BLV'11][Ortiz'10]. All of them, however, failed to catch on because of poor support across browsers, the need to install third party plugins, security considerations, closed proprietary standards and the extreme learning curve for developers.

In 2009, work began on WebGL, a new hardware accelerated graphics JavaScript API for the browser which tried to solve these issues [Anyuru'12]. In 2011, the Khronos Group froze the specifications for WebGL 1.0 and it was soon implemented in Google Chrome, Mozilla Firefox and the development versions of Safari and Opera [Anyuru'12]. With the adoption of WebGL in Internet Explorer 11 and iOS 8, Microsoft and Apple allowed this new technology to become truly cross platform.

Because WebGL is a Javascript API based on OpenGL ES2.0 and built in the browser, it shares the advantages of both web and desktop applications. It benefits from easy distribution, easy maintenance and cross platform support while allowing hardware accelerated rendering in the browser [Anyuru'12]. Moreover, WebGL does not require third party plugins and is an open standard which any browser manufacturer can implement [Anyuru'12].

WebGL is, however, a low level 2D API [JWGLIN] and creating even simple 3D experiments requires a significant knowledge of 3D development concepts and linear algebra to take advantage of its 3D capabilities. There exist abstraction libraries on top of WebGL which simplify 3D content creation [DeLillo'10][WGLCNTR]. These libraries, however, focus on drawing 3D scenes and objects and not on full 3D interactive website implementations. In particular, they lack integrated interactive navigation and form controls.

This project compares some of the most popular WebGL 3D libraries and verifies how and which of them should we use in order to implement the basic set of interactive form controls as part of a larger undertaking of creating a WebGL framework for building commercial interactive 3D websites.

Besides allowing creation of more pleasing website designs, a standardized, cross platform, 3D and interactive web framework would provide a powerful tool to businesses looking to present their products in the most realistic way possible. For example, a furniture selling company can create an immersive 3D web application enabling clients to walk through virtual rooms and inspect the products from all sides. Browser based 3D games [CX'11] can also benefit from integrated form controls.

Often, opponents of 3D web sites argue that 3D designs are distracting, complex and harder to use than their 2D counterparts [VSJP'12]. What is not mentioned is the difference in maturity of the community knowledge on how to properly design each one. 2D websites can just as easily be made distracting and hard to use if not designed properly. There are some standard guidelines in 2D website design which are taken for granted nowadays but actually took many trials and errors to finally agree on. 3D web development, on the other hand, is still in experimentation phase without any guidelines on designing user friendly interfaces.

Missing capabilities in WebGL libraries

Conventional HTML and CSS technologies give web developers a set of building blocks:

- Output to the user
    - presentation of static content such as text and media
    - styling of elements

    - positioning - fixed coordinates, tables, blocks, floating blocks, etc.

- Input from the user

    - page size and content position control - a user may determine the size of the browser window and which part of the content is visible by scrolling

    - interactive, navigational and form controls - selectable text, links, buttons, text boxes, drop downs, check boxes, radio buttons, datepickers (HTML5), range sliders (HTML5), etc

Existing 3D WebGL libraries are already capable of presenting static content (including complicated 3D models), styling and positioning of elements and managing content visibility through first person 3D controls. Three.js and Scene.js are some the more popular WebGL libraries nowadays and they all support similar features [TJSDCS][SJSDCS][X3DCS]:

- Geometries - cube, sphere, cylinder, custom, etc.

- Transformations - translation, rotation, scaling, etc.

- Materials - reflective, lambert, etc

- Textures - image textures, custom shader textures, etc
.
- Cameras - allows different points of view and navigation through the 3D scene

- LIghts - ambient light, directional light, etc.

Although, some of these features need to be improved in order to be used for commercial implementations, they have been utilized in many demos and experiments already. There exists, however, little research on the topic of 3D interactive, navigation and form controls for WebGL which a commercial 3D website would need in order to provide a fully integrated and functional experience. *Anna-Liisa Mattila and Tommi Mikkonen* created [MM'13] a widget library in Three.js which included buttons, checkboxes, event handling and collision detection. This library, however, was implemented only in Three.js and had performance issues with event handling when more than a few interactive elements were added to the scene. A library targeting commercial applications would need to support more diverse set of form controls and to perform better with more elements. To find out what the best way is to achieve this, we carried out a comparison of different technologies. Similarly, *Jonathan Karlsson* found out that using Three.js simplified WebGL but at a cost of performance where the increase of the number of objects resulted in lower frames per second. Another problem discovered in this research was that there was no simple way to render text. A commercial 3D solution needs to address these issues.

Chapter 3 - Background and context

Because of it being a new technology, there is still little academic research on WebGL and even less on interactivity and form controls in WebGL.

In 2013, *Anna-Liisa Mattila* and *Tommi Mikkonen* described [MM'13] the lessons learnt while building a new WebGL widgets library using Three.js. Their library managed the drawing and event handling of the widgets. As a proof of concept the library was used to rewrite Lively 3D - a 3D windowing environment. The HTML 2D main menu in the original Lively 3D, consisting of buttons, checkboxes and text boxes was replaced with a 3D version. The redesigned version of Lively 3D simplified development by reducing the amount of code for the same task.

They also found out [MM'13] that event handling of a 3D object is harder in WebGL than in a traditional HTML. 2D mouse coordinates need to be translated to 3D coordinates and check which 3D objects cover this point in space. This method for event handling caused performance issues [MM'13] in the order of O(n) where n is the number of 3D objects in the scene.

Their research, however, did not cover all typical widgets needed in a commercial web application. It is also not clear if using another library besides Three.js would have yielded different results and what possible practical solutions could be to the encountered performance issues.

In 2014, *Jonathan Karlsson* examined [Karlsson'14] the suitability of WebGL as a framework for developing user interfaces for TVs regarding performance and ease of use. It tried to answer the following two questions:

- "*How prepared are today's TV-platforms for running user interfaces based on WebGL with regards to performance relative to HTML5?*"

- "*Which development problems arise when developing WebGL based user interfaces?*"

He found out that WebGL is a low level library which makes it harder to develop user interfaces compared to HTML. A WebGL developer has to be careful of how many draw calls per frame s/he uses and how much the state changes in order to optimize performance. In his research he used Three.js as the development framework for the proof of concept. He confirmed that Three.js considerably simplified 3D WebGL development while allowing access to the low level capabilities of WebGL such as shader programming.

A proof of concept TV user interface was designed for this research [Karlsson'14]. To get a better understanding of the performance issues of WebGL a low level sample was constructed

[Karlsson'14] which combined all objects in a single buffer. This made it possible to use a single draw call and optimize performance and give directions on what could be a possible reason for the performance problems in the proof of concept implementation. Another big problem was rendering text. One approach was to create a separate canvas with 2D text in it which can then be used as a texture for the 3D objects. This was however a very undesirable approach causing rasterization and performance problems. Another approach was to have the whole alphabet in a single texture which can be shared between the object which need to use it. Both methods however resulted in blurred text edges and did not truly support 3D text.

Jonathan Karlsson also showed [Karlsson'14] that using WebGL libraries such as Three.js came at the cost of performance, especially when having large amount of objects in the scene which all resulted in a separate draw call. He mentioned possible solutions for future research such as using combined geometries in Three.js, using pure WebGL or designing a new 3D library on top of WebGL which focuses more on performance.

In the following chapters we build upon this research by analysing and comparing the Three.js and Scene.js WebGL libraries. The purpose of our analysis is to find solutions to the problems of building a commercial 3D web application. Previous research indicated that 3D text and performance issues caused by the presence of many interactive 3D objects would be some of the central points of our analysis.

Chapter 4 - Research method

In order to find the best way to build 3D form controls and lay the groundwork for a future full 3D website development library, a list of the existing WebGL frameworks was gathered and compared. On the basis of their popularity and feature support, two libraries from this list were then picked for further comparison and analysis. A design research method was chosen where the picked libraries were used to build a proof of concept, measurements were then taken, compared and analyzed.

The chosen libraries were compared and analyzed on the basis of three measurements - implemented features, performance and implementation size where implemented features and performance were considered most important by the Ohpen team and were therefore given higher priority in the final discussion of the results.

It is hypothesized that Three.js would allow implementing the most features from the proof concept requirements since it is the most popular WebGL 3D library with a strong community support and rich feature set.

It is also hypothesized that SceneJS would achieve the best performance of the proof of concept implementation. Our preliminary research showed that SceneJS was designed from the beginning with performance in mind. Its documentation reveals support for several performance optimizations such as color-based object picking, asynchronous object loading, frustum culling, level-of-detail, instancing, vertex sharing and resolution scaling.

Chapter 5 - Domain analysis

There already exist many frameworks which make creating WebGL 3D content easier. The Khronos Group which designed and maintains WebGL currently have 35 frameworks in their list of user contributions [WGLCNTR]. These frameworks differ mainly on popularity, feature set and code size. Some of them target 3D game development, others 3D content creation and drawing and yet others are very lightweight and only provide abstraction of the basic WebGL code structures.

	Three.js	***X3DOM***	***Scene.js***	***Babylon.js***	***PhiloGL***	***GLGE***	***CubicVR***	***SpiderGL***
Popularity	18350	175	80	126	18	56	4	12
Size KB (minified)	410	875	240	637	59	324	339	178
Collision detection	plugins	Yes	plugins	Yes	No	No	No	No
Primitive geometries	20	15	10	8	6	6	6	0
Custom geometries	Custom, Collada, OBJ, glTF, PDB	X3D, X3D-XML, (convert via AOPT)	Custom, OBJ, 3DS, MD2	Custom, .babylon (exporters for 3DSMax, Blender, Cheetah3D)	Custom	Custom, Collada, OBJ, MD2	Custom, Collada	Custom
Primitive materials	14	2	1	2	0	0	0	0
Custom materials	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Image textures	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Movie textures	Yes	Yes	Yes	Yes	No	Yes	Yes	No
3D text	Yes	Yes	Yes	Only in textures	No	Yes	Only in textures	No
Lights	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Cameras	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Shadows	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Custom shaders	Yes	Yes	Yes	Yes	Yes	Vertex	Yes	Yes
Merged geometries	Yes	No	Yes	Yes	No	No	No	No
Transformations	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object selection by ray intersection	Yes	Yes	Yes	Yes	No	No	Yes	No
Object selection by color picking	No	No	Yes	No	Yes	Yes	No	No

Table 1 *- comparison of popular WebGL libraries [TJSDCS][SJSDCS][X3DCS][BJSDCS][PGLDCS][GLGEDCS][CVRDCS][SGLDCS]*

Table 1 provides an overview of the most widely used WebGL frameworks compared by popularity, size and support for important features. Popularity was calculated by the number of search results for each framework on the *stackoverflow.com* platform. *Stackoverflow.com* is, nowadays, the most popular question and answer platform for software developers. What can be found on a certain topic there is a good indication of its popularity in the community.

To ensure abundance of documentation, code samples and information on online forums we chose popular WebGL frameworks. Popular software frameworks tend to enjoy more regular and continuous maintenance too. Furthermore, they had to have a rich feature set which could provide a more solid base to build upon.

Apart from good popularity and support for basic 3D drawing features such as geometries, textures, cameras, lights and transformations, the frameworks which we used for our proof of concept had to support specific features needed in interactive 3D web development. Interactive 3D web applications require efficient rendering of more than a few interactive objects. To achieve this in WebGL, geometries need to be merged in a single big geometry which requires only a single call to the WebGL draw function [Karlsson'14][GGLIOYT'11][GGLIOWS'11]. The technique used for interacting with 3D objects is important as well. Some of the WebGL 3D frameworks support only selection of objects by un-projecting XY screen coordinates to 3D space, casting a ray and testing for intersections. Only some frameworks support the technique called "selection by color picking". Another important feature for 3D web applications is 3D text. Some frameworks support only text in textures while others have no support for 3D text at all. Support for custom shaders is also an important requirement allowing low level control of the rendering on the graphics card and better flexibility for performance optimizations.

Three.js is by far the most popular WebGL framework with over 18000 search results on *stackoverflow.com* followed by X3DOM with 175 hits, Babylon.js with 126 hits, Scene.js with 80 hits, GLGE with 56 hits and the rest with less than 20 hits.

At the time of writing, Scene.js is the only library supporting merged geometries, object selection by color picking, 3D text and custom shaders all together. Three.js follows close in this aspect, only missing support for object selection by color picking. GLGE comes in next with only missing support for merged geometries, ray intersection picking and partial support for custom shaders. X3DOM misses support for merged geometries and object selection by color picking. Babylon.js misses support for 3D text and object selection by color picking. The rest of the frameworks have poorer features sets and are less popular in the community.

Looking at the popularity and feature sets of the different WebGL frameworks we decided to build our proof of concept with Three.js and Scene.js. In the following chapters they are examined and compared in detail as the alternatives for a base framework to use for building an interactive 3D WebGL library.

Chapter 6 - Prototype

For the purpose of this project, a 3D implementation of the TODOMVC.COM [TMVC][KTMVC] web application was built in the chosen WebGL libraries as a proof of concept and a comparison base. TODOMVC.COM is a compact project designed to include core features of data manipulation web applications. It covers user input, data selection, removal and filtering. Although, TODOMVC.COM is intended to be a sample functional design for comparing MVC development frameworks, its focus on the most important data manipulation features made it an excellent target design for our proof of concept as well.

The original 2D design of the TODOMVC.COM application (***Fig. 1***) represents a main input text field at the top where the user can add an item to the underlying “todo” list. The vertical list allows selecting an item by clicking on the checkbox on the left side, editing its text upon double clicking it or deleting it by clicking on the “X” icon on the right side. The footer shows the size of the list, allows filtering (*All, Active or Completed*) and clearing of the completed (selected) items. Furthermore, the main input item at the top contains a global check box allowing the selection of all items at once.

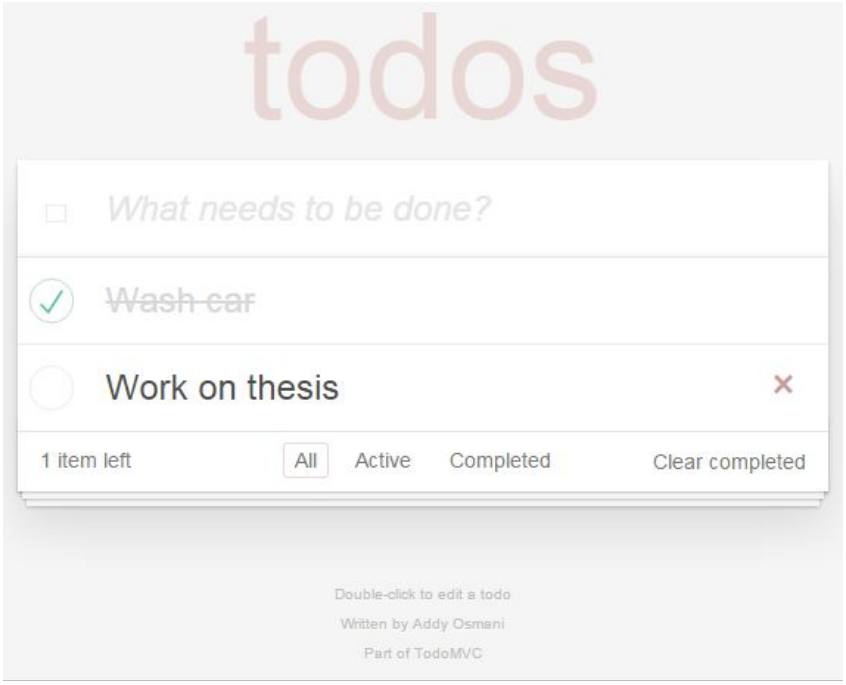

Fig. 1 *- Original 2D design of the TODOMVC.COM web application [TMVC]*

The design of the 3D version (***Fig. 2, Fig. 3, Fig. 4***) was created using the open source 3D editor - Blender [BLNDR]. The 3D version supports the same features as the 2D original while enabling depth of the elements and positioning in the extra dimension as well as first person navigation controls through the scene.

Fig. 2 *- Full view of the 3D TODOMVC.COM design*

The frontal view still displays a mutable list of todo items but with added perspective. The text was given depth, squares became cubes, circles converted to spheres and the grid became a 3D shelf (***Fig. 3***).

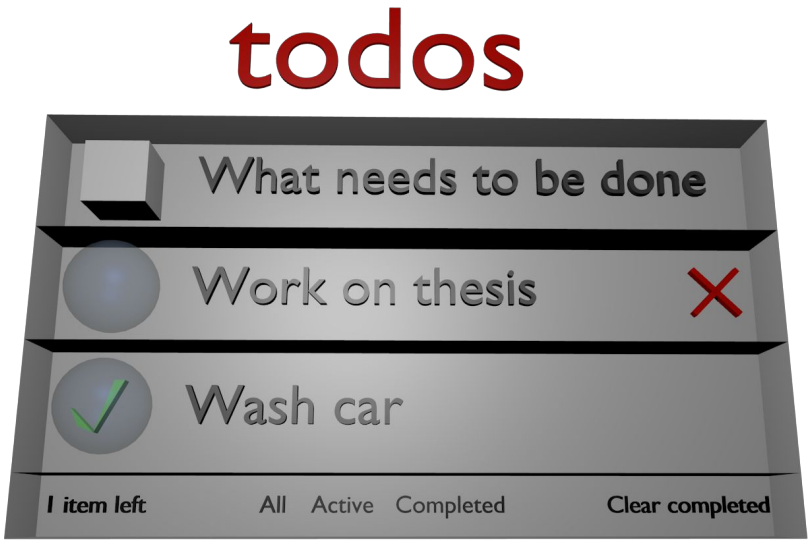

Fig. 3 *- Frontal view of the 3D TODOMVC.COM design*

On the side, perpendicular to the “todo” list, a header containing a logo and a simple menu shows how the extra dimension can be used to position elements (***Fig. 4***).

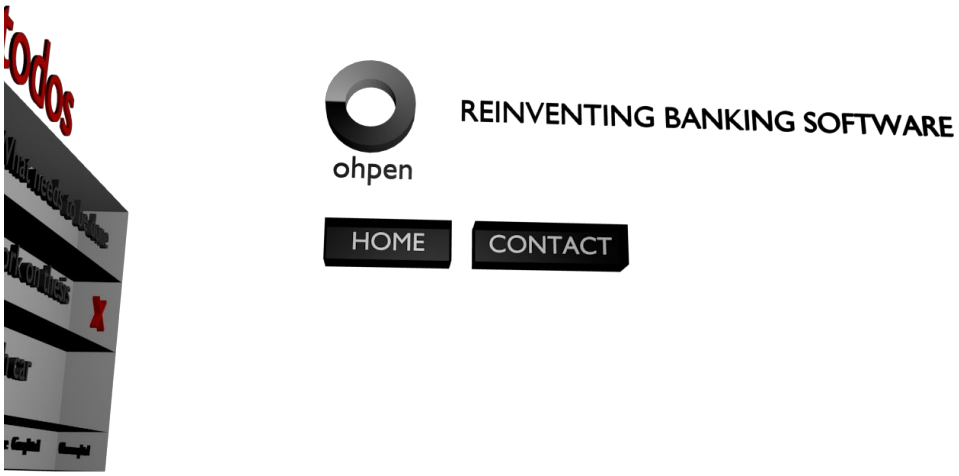

Fig. 4 *- Side view of the 3D TODOMVC.COM design*

All concepts were built with first person walk navigation controls to enable the user to easily explore the 3D scene from all sides.

All implementations of the proof of concept were built as a Microsoft .NET MVC web application in Visual Studio 2013. Once the proof of concept was implemented in the two chosen libraries, measurements could be taken for comparison and analysis.

Chapter 7 - Experimental setup

Prototype implementation setup

The prototype, as described in the previous chapter, was implemented using Three.js and SceneJS. Both implementations were designed to have the same form control interfaces in order to allow keeping the code of the domain logic the same. This ensured that we could efficiently compare the measurements of the two implementations of the same domain logic in the analysis stage of our research.

Measurements

Three metrics were measured and compared between the implementations in the chosen WebGL libraries - implemented features, performance and implementation size.

Achieved features

It was important to see which libraries could accomplish how many of the TODOMVC.COM features. The implementation of the following features was measured and compared:

3D text - the “todos” title, Ohpen slogan, labels of buttons, the text showing the non completed “todo” items, etc.

1. Text geometry
2. Depth
3. Fonts
4. Bold, underline, strikethrough
5. Text selection and copying
6. Changing the value of an already created 3D text

3D button - the “Home” and “Contacts” buttons, the “Remove todo item” button and the “Clear completed” button

7. A button geometry with text
8. “OnClick“ event registration

3D checkbox - the “Select all” checkbox in the header and the “Is completed” checkbox per “todo” item

9. Main geometry with a togglable child “check” geometry
10. “OnChange“ event registration

3D radio button - needed for the filter in the footer of the “todo” items list which controls the visible items

11. A radio button geometry with text
12. “OnChange“ event registration
13. Groups of radio buttons

3D textbox - the textbox where new “todo” items can be entered and the textboxes used to edit each item

14. A textbox geometry
15. Typing of text after focus
16. Text cursor
17. Text selection, copying and pasting
18. Dynamic right alignment of long text
19. Hiding overflown text
20. “OnFocus“ event registration
21. “OnBlur“ event registration
22. “OnKeyDown“ event registration
23. “OnKeyPress“ event registration

3D shelf - the container of each “todo” item

24. A shelf geometry
25. “MouseOver” event registration
26. “MouseOut” event registration
27. “DoubleClick” event registration

Visual effects

28. Transparency
29. Specularity
30. Lights

Performance

We measured frames per second of each implementation in three different browsers - Chrome 44.0.2403.125 m, Firefox 39.0 and Internet Explorer 11.0.9600.17914. Furthermore, the measurements were done in a still state, in an user interaction state and for a different number of 3D objects in the scene - 50, 100, 500, 1000, 5000 and 10000.

To generate the correct number of objects on the scene we varied the number of “todo” items. Each “todo” item in the TODOMVC application contains six 3D objects: the shelf container, the

“is completed” checkbox sphere and check mesh, the “todo” item title text and the delete button container and its text.

Furthermore, there are twenty two fixed objects in the TODOMVC application which are independent of the number of “todo” items: the “todos” title text, the “select all” checkbox and its check mesh, the “add new item” textbox, the “items left” text, the "All" radio button and radio button text, the "Active" radio button and radio button text, the "Completed" radio button and radio button text, the "Clear completed" button container and its text, the Ohpen logo and Ohpen logo text, the slogan text, the “Home” button container and its text and the “Contacts” button and its text.

The total number of 3D objects in the proof of concept could be calculated by the formula: **<number of “todo” items> * 6 + 22.**

It is important to remember that when we talk about objects on the scene in this case we mean the visible geometries on the scene. Color based event handlers create hidden clones of the visible objects as well. Most libraries would translate an object to a WebGL `draw` function call which is why having many objects on the scene results in increased stress on the Javascript engine of the browser.

All tests were executed automatically by a custom-built test runner application which sequentially started the different browsers with the url of each concept implementation and a parameter indicating the number of objects on the scene. We used an adapted version of the Stats.js tool [DBSTTS] to measure the frame rate where the graph was scaled up and level indicators were added to the frames per second axis and the seconds axis. Stats.js [DBSTTS] is a tool developed by the creators of Three.js. It can be used, however, to measure frames per second of any WebGL application. The Stats.js tool was adapted and used to log the frame rate every second for initial duration of 30 seconds in a still state then 30 seconds with simulated user interaction on the page. Finally, a cool down frame rate was logged for 30 more seconds in a still state. The user interaction was simulated by triggering a mousemove event via JavaScript [JSMEV] on top of the list of “todo” items. We then recorded separately the average frame rate for the still state and for the interaction state.

All performance tests were executed on HP Pavilion g7, AMD Athlon II P360 Dual-Core 2.30 GHz, ATI Mobility Radeon HD 4250 512MB, 8GB RAM, 250 GB SSD, Windows 7 Home Premium SP1.

Implementation size

Physical source lines of code were measured to give an indication of the complexity and maintainability of each implementation.

The Ohpen team found “implemented features” and “performance” to be more important than “implementation size”. Therefore, “implemented features” and “performance” weighed heavier in the final discussion of the results.

Chapter 8 - Results

Three.js

Implementation

Framework

Creating a scene in Three.js and adding 3D objects to it is a straightforward task (***Code 1)***. A 3D object is typically represented by a mesh which consists of a geometry and a material. The position, rotation and scale are all properties of the mesh and can be easily manipulated as well. Meshes can be nested in parent child structures where changing, for example, the position of the parent moves the children as well. Different types of cameras and lights are supported and can be added directly to the scene too.

```
var camera = new THREE.PerspectiveCamera(5, window.innerWidth / window.innerHeight, 1, 10);

var scene = new THREE.Scene();

scene.add(camera);

var shape = new THREE.SphereGeometry(2);

var materialShape = new THREE.MeshBasicMaterial({ color: '#FF0000' });

var shapeMesh = new THREE.Mesh(shape, materialShape);

shapeMesh.position.set(-5, 0, 0);

scene.add(shapeMesh);
```

***Code 1** - an example of creating a scene in Three.js and adding a red sphere to it*

Structure

Our implementation of the proof of concept in Three.js contains a separate class for each type of form control - `GlButton, GlCheckBox, GlRadioButton` and `GlTextBox`.

The class `GlScene` serves as a wrapper of Three.js’s `THREE.Scene` class and is the base container to which all other objects are added. `GlShelf` provides the logic for the container of a single “todo“ item. Both `GlScene` and `GlShelf` inherit from `GlContainer` which provides the logic for a parent 3D object with children.

All 3D form controls and containers inherit from the base `GlObject3d` class which contains basic properties such as `id`, `name`, `glClass`, `mainGeometry`, `mainMaterial` and `mainMesh`.

All form controls contain an internal `Init()` function where the object's geometries, materials and meshes are created. ***Code 2*** shows the creation of the geometry, material and mesh of the `GlButton` form control.

```
this.mainGeometry = new THREE.BoxGeometry(this.boxWidth, this.boxHeight, this.boxDepth);

this.mainMaterial = new THREE.MeshPhongMaterial({ color: this.color, opacity: this.opacity,
transparent: true, vertexColors: THREE.FaceColors, side: THREE.DoubleSide, specular:
0x050505, shininess: 100 });

this.mainMesh = new THREE.Mesh(this.mainGeometry, this.mainMaterial);
```

***Code 2** - creating the main mesh of the GlButton form control*

Once the 3D object is created it can easily be added to the scene. **Code 3** shows an example of the creation, positioning and adding to the scene of the "todos" title text:

```
var pageTitleText = new GlText({ value: 'todos', textColor: 0xFF0000, letterSize: 2 });

pageTitleText.SetProperties({ position: { x: -3, y: 7, z: -30} });

glScene.Add(pageTitleText);
```

***Code 3** - adding and positioning of a form control to the scene*

Navigation

Navigation through the scene was implemented using an adapted version of the pointer lock controls as described in the examples within the Three.js library [TJSPL]. The original example uses the pointer lock feature provided by some browsers to prevent the mouse pointer of going out of screen and allowing seamless 360 degrees look around capability. This is not supported in Internet Explorer 11 and some mobile browsers. Therefore, we replaced the pointer lock functionality by one which works in all browsers and where the user has to hold the left mouse button down and then move the mouse to look around. This allows the user to rotate the camera further in the same direction by going back and repeating the dragging motion.

Event handling

In order to find the best performing method for managing user interaction with Three.js, two different types of event handlers were implemented - the "ray based event handler" and the "color based event handler". Each type is encapsulated in a separate class but both can be consumed in exactly the same way by the objects which need to register event listeners. **Code 4** shows an example of how the mouseover event of a "todo" item's container is bound to a function which displays the item's delete icon:

```
EventHandler.AddEventListener('mouseover', shelf,
   function () {
      deleteButton.SetDisplay(true);
   }
);
```

Code 4 *- registering of a "onmouseover" event listener to a shelf object*

Ray based event handler

The ray based event handler is a ray intersection algorithm which registers an event listener on the document per event type. For example, if there are five 3D buttons which require click event handling then one click event is registered on the `document`. The meshes of the five buttons are added to an array which contains the meshes of all clickable objects. When the click event is triggered the array with the clickable object meshes is passed to the ray intersection algorithm to determine which ones came in the path of the ray.

The ray intersection algorithm casts a ray having the same origin as the camera (viewpoint) and direction calculated from the 2D event coordinates. Three.js contains a helper class called "`THREE.Raycaster`" which can be used to easily do the intersection calculations. Here is an example of how the "`THREE.Raycaster`" can be used to find which objects in the array "`testObjects`" were intersected by an event at 2D coordinates "x", and "y":

```
function getPerspectiveIntersects(x, y, context, testObjects) {

    var raycaster = new THREE.Raycaster();

    raycaster.setFromCamera({x: x, y: y}, context.camera);

    var intersects = raycaster.intersectObjects(testObjects);

    return intersects;
}
```

Code 5 *- typical Three.js71 algorithm for finding 3D objects from 2D coordinates [TJSRCD]*

The function from ***Code 5*** does not work when navigation in the scene is enabled using the Three.js pointer lock controls [TJSPL]. The reason for it is that the "`setFromCamera`" function of the "`THREE.Raycaster`" does not take into account the position shift of the camera caused by the pointer lock controls while navigating through the scene. Here is the adapted version of the same function which does take into account the navigation shift of the camera:

```
function getPerspectiveIntersects(x, y, context, testObjects) {

    var raycaster = new THREE.Raycaster();

    raycaster.ray.origin.copy(context.pointerLockControls.getObject().position);

    raycaster.ray.direction
    .set(x, y, 0.5)
    .unproject(context.camera)
    .sub(context.pointerLockControls.getObject().position)
    .normalize();

    var intersects = raycaster.intersectObjects(testObjects);

    return intersects;
}
```

Code 6 *- an adapted Three.js72 algorithm for finding 3D objects from 2D coordinates working together with pointer lock controls [TJSPL]*

The ray intersection algorithm is executed by Javascript and has to go through the list of "`testObjects`" to check for each one if it is positioned in the path of the ray in 3D space. This results in an increasing load on the CPU when more interactive objects are added to the scene. In order to check if there was a better performing method for handling user interaction the "color based event handler" was implemented as well.

Color based event handler

The color based event handler renders all objects registered for event handling second time in a hidden render target using an unique color corresponding to the object's id. Using WebGL's "`readPixels`" function one can then pick the color at the coordinates of the event and convert it back to the id to find which object was interacted with.

When using the "color based event handler" a second scene is created and rendered in a hidden render target. ***Code 7*** and ***Code 8*** show how the hidden render target is created and the second scene rendered on it:

```
pickingRenderTarget = new THREE.WebGLRenderTarget(width, height);
pickingRenderTarget.minFilter = THREE.LinearFilter;
pickingRenderTarget.generateMipmaps = false;
```

Code 7*- creating a hidden render target in Three.js [TJSGPU]*

```
renderer.render(pickingScene, camera, pickingRenderTarget);
```

Code 8 *- rendering a scene on the hidden render target [TJSGPU]*

All objects which are registered for event handling need a clone rendered on the hidden render target in a color which encodes the object's id. ***Code 9*** shows how this copy is created.

```
glObject.pickingMaterial = new THREE.MeshBasicMaterial({ color: glObject.pickingId });

glObject.pickingMesh = new THREE.Mesh(glObject.mainGeometry, glObject.pickingMaterial);

glObject.pickingMesh.position.copy(glObject.mainMesh.position);
glObject.pickingMesh.rotation.copy(glObject.mainMesh.rotation);
glObject.pickingMesh.scale.copy(glObject.mainMesh.scale);
```

Code 9 *- creating a copy of a 3D object, rendered on the hidden render target in a color corresponding to the object's id [TJSGPU]*

Once all interactive objects have a copy of themselves rendered in a separate scene one can read the pixel color from this scene at an event's coordinates. Decoding this color gives us the object's id which can then be looked up in a dictionary containing the objects registered for event handling. ***Code 10*** shows how we read from the hidden scene.

```
renderer.readRenderTargetPixels(pickingRenderTarget, x, pickingRenderTarget.height - y, 1,
1, pixelBuffer);
```

Code 10 *- reading the color from the hidden scene at screen coordinates x and y [TJSGPU]*

Achieved features

	Three.js
3D text	
Text geometry	✔
Depth	✔
Fonts	✔
Bold, underline, strikethrough	✔
Text selection and copying	X
Changing the value of an already created 3D text	✔
3D button	
A button geometry with text	✔
“OnClick“ event registration	✔
3D checkbox	
Main geometry with a togglable child “check” geometry	✔
“OnChange“ event registration	✔
3D radio button	
A radio button geometry with text	✔
“OnChange“ event registration	✔
Groups of radio buttons	✔
3D textbox	
A textbox geometry	✔
Typing of text after focus	✔
Text cursor	X
Text selection, copying and pasting	X
Dynamic right alignment of long text	✔
Hiding overflown text	X
“OnFocus“ event registration	✔
“OnBlur“ event registration	✔
“OnKeyDown“ event registration	✔
“OnKeyPress“ event registration	✔
3D shelf	
A shelf geometry	✔
“MouseOver” event registration	✔
“MouseOut” event registration	✔
“DoubleClick” event registration	✔
Visual effects	
Transparency	✔
Specularity	✔
Lights	✔

***Table 2** - achieved features in the Three.js implementation of the proof of concept*

All features of the original TODOMVC.COM application were successfully implemented except for the following:

- text selection - an achievable feature but required registering event handling to each letter in order to detect where the start and the end of the selection occurred. Another option was to detect at which coordinates of the text container was the selection started or ended and calculate which letter is positioned at this location. In any case, its implementation required considerable amount of time compared to the rest of the features. This functionality was left for future research

- textbox cursor - a possible solution was similar to the solution for the text selection issue since in both cases we needed to check which letter was interacted with. Moreover, reordering of the letters that followed the cursor was needed when the user wanted to insert text in the middle of the textbox. It was an achievable feature but required a sizable time investment compared to the rest of the functionalities and was therefore left for future research

- pasting text in a textbox - this feature required having a textbox cursor in order to know where to add the pasted text

- text overflow in textbox - also left for future research as it required a considerable development effort on its own. It required checking which letters were positioned outside of the boundaries of the textbox in order to make them invisible. To completely reproduce the behaviour of a traditional HTML textbox, we would even have to be able to hide part of a letter since the size of a textbox does not always match the sum of the widths of the visible letters

Eventually, 26 out of the total 30 features were achieved in the implementation of the proof of concept using Three.js (**Table 2**).

Performance

The performance tests were repeated for both types of event handlers: the “ray based event handler” and the “color based event handler”.

Our tests showed that the “ray based event handler” performed better than the “color based event handler”. The difference became more apparent with the increase of the number of objects on the scene.

In Chrome, at 50 and at 100 objects in the still test there was a difference of 4% in the frame rate between the two event handlers. In the interaction test at the same load level the difference was around 8%. At 500 objects the difference between the two event handlers in the still test became close to 10% and 20% in the interaction test. At 1000 objects the difference grew to 17% in the still test and 24% in the interaction test (***Table 3, Chart 1 - Chart 12, Table 4, Chart 13 - Chart 24***).

In the still test, similar frame rate differences between the two event handlers was observed in Firefox and Internet Explorer in relation to the number of objects on the scene. In the interaction test, however, nearly no frame rate drop was detected compared to the still test in these two browsers.

Firefox was the only browser that managed to open the Three.js implementation with 5000 objects but only after accepting three warnings about the page script taking too long and after the browser process had reached 2.8 GB of memory usage. The other browsers became unresponsive at 5000 objects.

Ray based event handler

# objects\browser	Chrome	Firefox	IE
50 objects	still: 50 fps mouseover: 50 fps	still: 59 fps mouseover: 60 fps	still: 9 fps mouseover: 9 fps
100 objects	still: 49 fps mouseover: 49 fps	still: 59 fps mouseover: 59 fps	still: 8 fps mouseover: 8 fps
500 objects	still: 40 fps mouseover: 39 fps	still: 34 fps mouseover: 35 fps	still: 5 fps mouseover: 5 fps
1000 objects	still: 28 fps mouseover: 26 fps	still: 17 fps mouseover: 19 fps	still: 3 fps mouseover: 3 fps
5000 objects	*failed to load*	*failed to load* [1]	*failed to load*
10000 objects	*failed to load*	*failed to load*	*failed to load*

***Table 3** - performance measurements grid of the Three.js proof of concept using the "ray based event handler"*

The average frame rate of the Three.js proof of concept using the "ray based event handler" in the still test was 20.06 FPS and 20.1 FPS in the interaction test. The overall average was 20.08 FPS.

[1] Firefox could be forced to open the page with 5000 objects after accepting a warning that the script on the page is taking too long. The browser process grew to 2.8 GB of memory usage with 5000 objects on the scene and the page was not usable at 0 fps.

Chrome

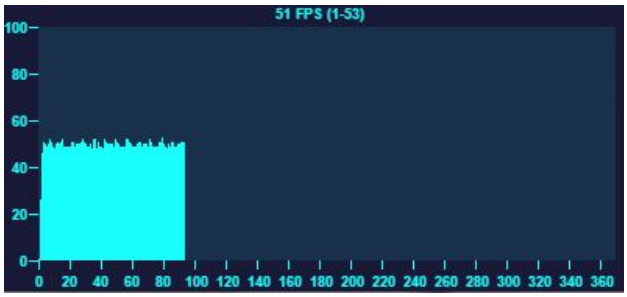

Chart 1 - *Three.js POC, ray based event handling, 50 objects, Chrome*

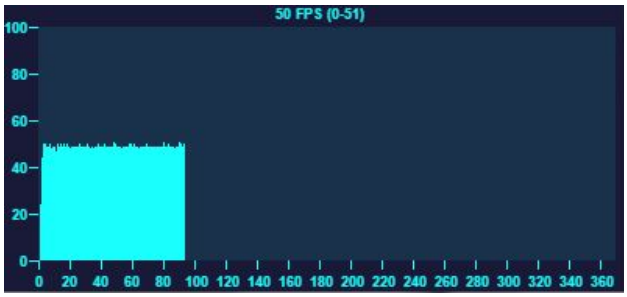

Chart 2 - *Three.js POC, ray based event handling, 100 objects, Chrome*

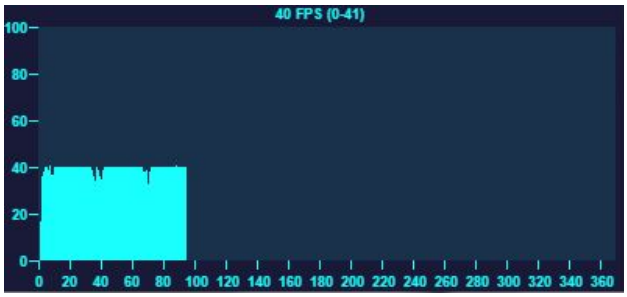

Chart 3 - *Three.js POC, ray based event handling, 500 objects, Chrome*

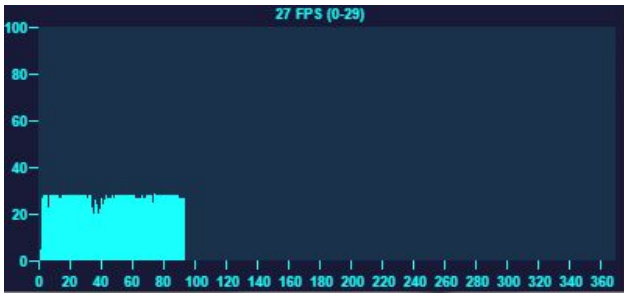

Chart 4 - *Three.js POC, ray based event handling, 1000 objects, Chrome*

Firefox

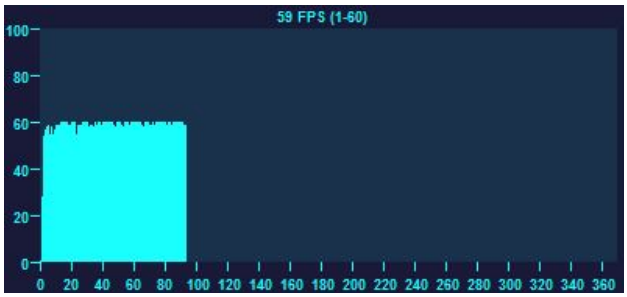

Chart 5 - *Three.js POC, ray based event handling, 50 objects, Firefox*

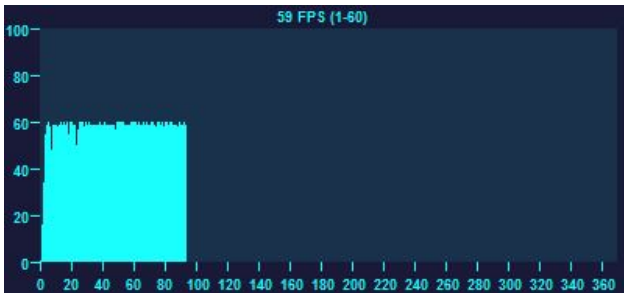

Chart 6 - *Three.js POC, ray based event handling, 100 objects, Firefox*

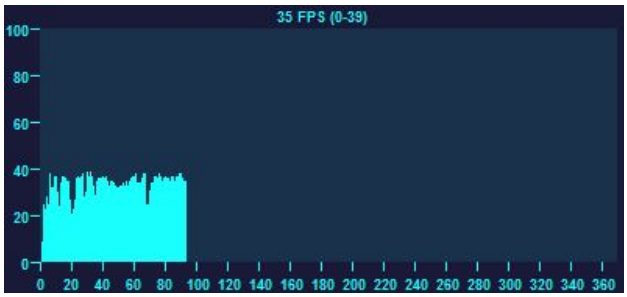

Chart 7 - *Three.js POC, ray based event handling, 500 objects, Firefox*

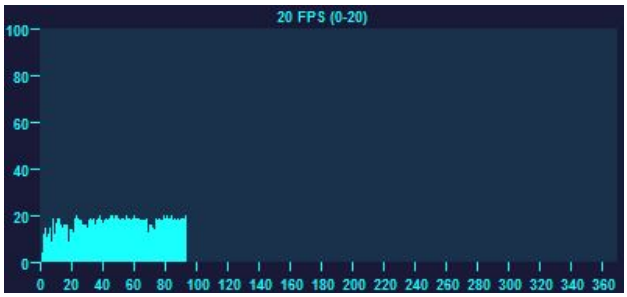

Chart 8 - *Three.js POC, ray based event handling, 1000 objects, Firefox*

Internet Explorer

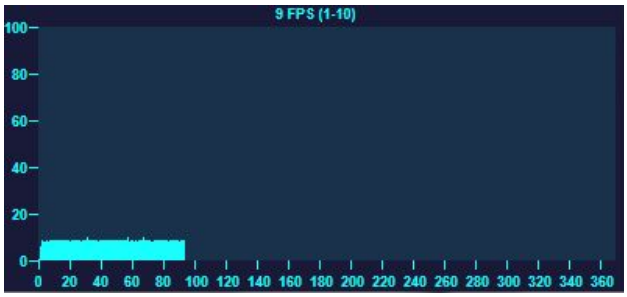

Chart 9 - *Three.js POC, ray based event handling, 50 objects, IE*

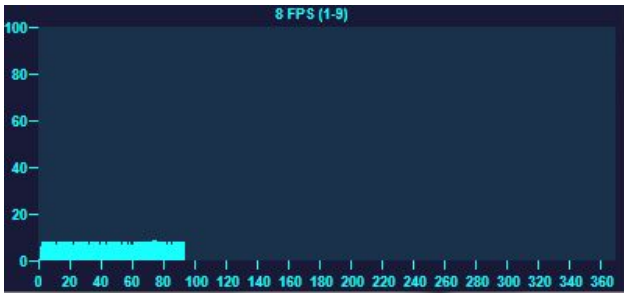

Chart 10 - *Three.js POC, ray based event handling, 100 objects, IE*

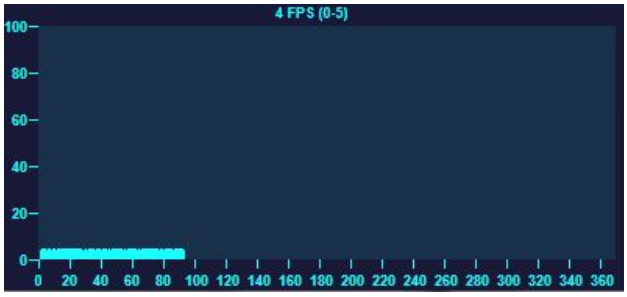

Chart 11 - *Three.js POC, ray based event handling, 500 objects, IE*

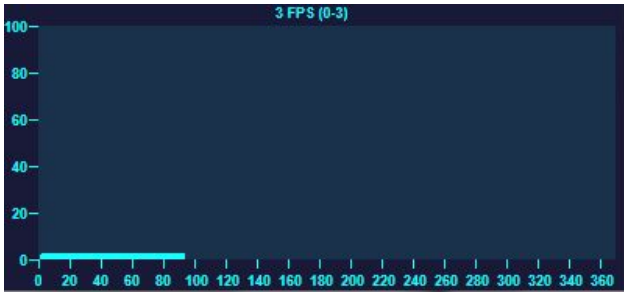

Chart 12 - *Three.js POC, ray based event handling, 1000 objects, IE*

Color based event handler

# objects\browser	Chrome	Firefox	IE
50 objects	still: 48 fps mouseover: 46 fps	still: 57 fps mouseover: 55 fps	still: 8 fps mouseover: 8 fps
100 objects	still: 47 fps mouseover: 45 fps	still: 54 fps mouseover: 53 fps	still: 7 fps mouseover: 7 fps
500 objects	still: 36 fps mouseover: 31 fps	still: 25 fps mouseover: 24 fps	still: 4 fps mouseover: 4 fps
1000 objects	still: 23 fps mouseover: 20 fps	still: 13 fps mouseover: 13 fps	still: 3 fps mouseover: 3 fps
5000 objects	*failed to load*	*failed to load*	*failed to load*
10000 objects	*failed to load*	*failed to load*	*failed to load*

Table 4 *- performance measurements grid of the Three.js proof of concept using the "color based event handler".*

The average frame rate of the Three.js proof of concept using the “color based event handler” in the still test was 18.06 FPS and 17.17 FPS in the interaction test. The overall average was 17.60 FPS.

Chrome

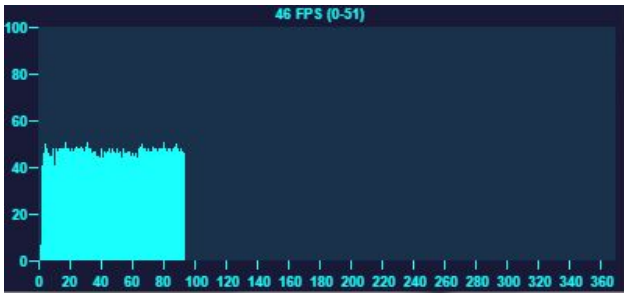

Chart 13 - *Three.js POC, color based event handling, 50 objects, Chrome*

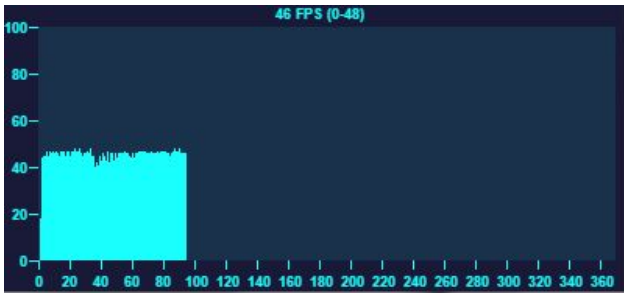

Chart 14 - *Three.js POC, color based event handling, 100 objects, Chrome*

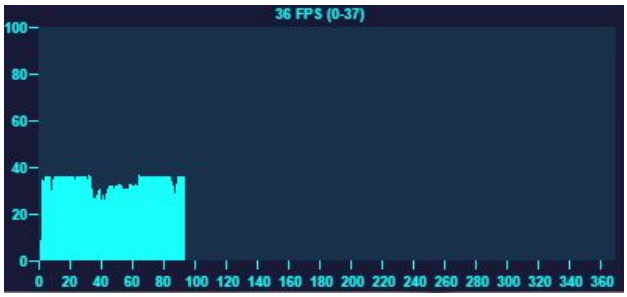

Chart 15 - *Three.js POC, color based event handling, 500 objects, Chrome*

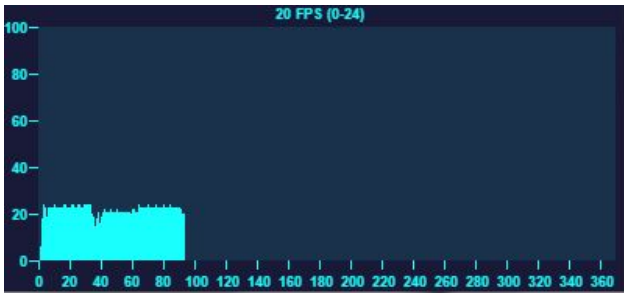

Chart 16 - *Three.js POC, color based event handling, 1000 objects, Chrome*

Firefox

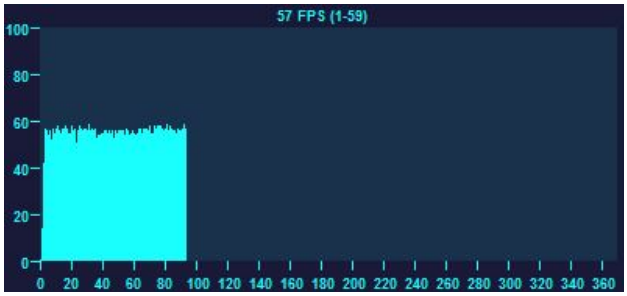

Chart 17 - *Three.js POC, color based event handling, 50 objects, Firefox*

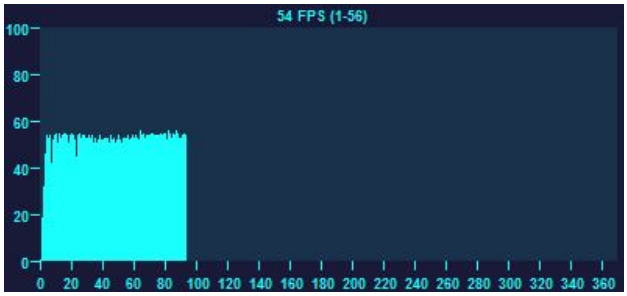

Chart 18 - *Three.js POC, color based event handling, 100 objects, Firefox*

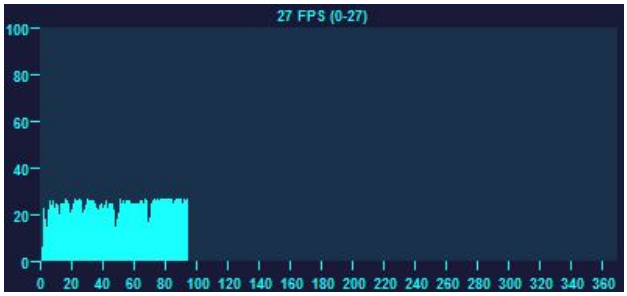

Chart 19 - *Three.js POC, color based event handling, 500 objects, Firefox*

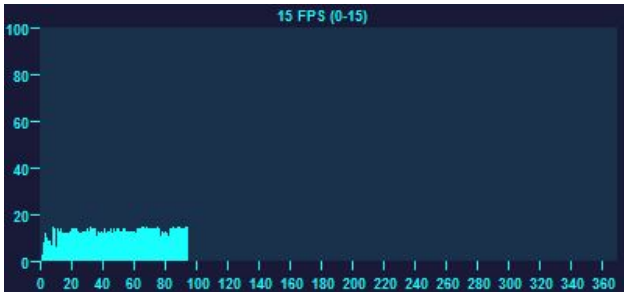

Chart 20 - *Three.js POC, color based event handling, 1000 objects, Firefox*

Internet Explorer

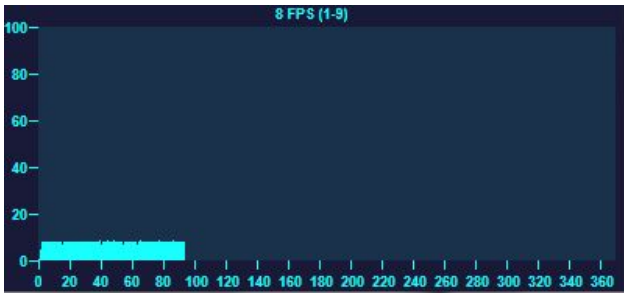

Chart 21 - *Three.js POC, color based event handling, 50 objects, IE*

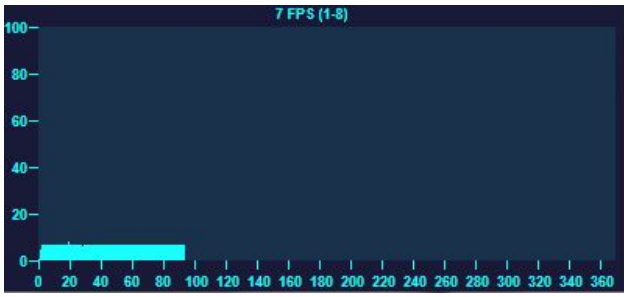

Chart 22 - *Three.js POC, color based event handling, 100 objects, IE*

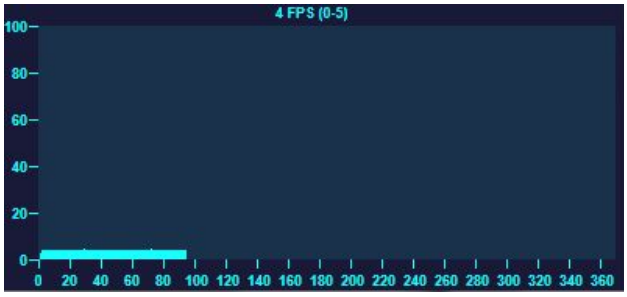

Chart 23 - Three.js POC, color based event handling, 500 objects, IE

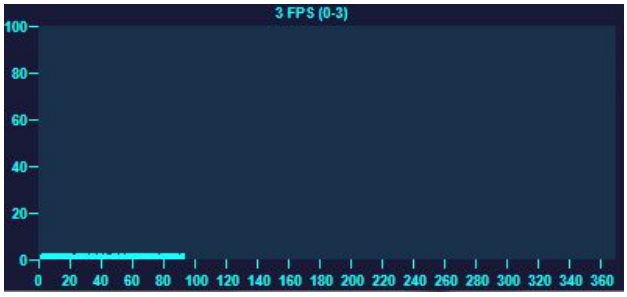

Chart 24 - *Three.js POC, color based event handling, 1000 objects*

Implementation size

The largest and most complicated parts of the Three.js implementation were the event handlers' classes and the textbox form control. The `EventHandlerColorBased` class is larger than the `EventHandlerRayBased` and the rest of the classes since it contains considerable amount of logic which manages the hidden scene and the clones of the objects registered for event handling. The relatively bigger size of the `GlTextBox` class is caused by the logic which controls the adding and removing of a new mesh for each new letter and the positioning of the letter's mesh at the correct offset from the previous letter. The rest of the form controls classes are straightforward implementations only containing logic for creating their respective geometry and for executing their registered events at the right moment. The page where instances of the form controls are created, positioned and added to the scene has the most lines of code compared to the rest of the classes. Its logic, however, is not framework specific but rather specific to the TODOMVC application domain.

File	Physical SLOC
Page (js code)	328
EventHandlerRayBased.js	111
EventHandlerColorBased.js	239
GlButton.js	64
GlCheckBox.js	122
GlCollada.js	27
GlContainer.js	59
GlForm.js	33
GlObject3d.js	44
GlRadioButton.js	151
GlScene.js	13
GlShelf.js	40
GlText.js	50
GlTextBox.js	184
Utils.js	29
WalkControls.js	117
Total ray based	**1360**
Total color based	**1488**

***Table 5** - lines of code per file of the Three.js POC implementation*

Scene.js

Implementation

Framework

The structure of the 3D objects and transformations in a SceneJS scene is specified in a hierarchical JSON nodes format. For example, to create a blue box and translate it to the left one would have to create a “translate” node with a child blue “material” node and a grandchild “geometry/box” node (***Code 11)***.

```
SceneJS.createScene({
    nodes: [
        {
            id: "tr1", type: "translate", x: -5, y: 0, z: 0,
            nodes: [
                {
                    type: "material", color: { r: 0, g: 0, b: 1 },
                    nodes: [
                        { type: "geometry/box", xSize: 1, ySize: 1, zSize: 1 }
                    ]
                }
            ]
        }
    ]
});
```

***Code 11** - an example of node hierarchy in SceneJS - a blue cube translated to the left*

Retrieving already created nodes is done via the `getNode(id, callback)` function of SceneJS’s `scene` node. Because SceneJS retrieves nodes asynchronously the `getNode` function asks for a callback to execute once the node has been found.

Structure

The code structure of the SceneJS proof of concept is very similar to the Three.js one. Form controls are again divided in separate classes - `GlButton`, `GlCheckBox`, `GlRadioButton`, `GlText`, `GlTextBox` and `GlWavefrontObj`. Container controls such as `GlDiv`, `GlShelf` and `GlScene` are also encapsulated in their own classes and inherit from the base `GlContainer`.

All form controls and containers inherit from the base `GlObject3d` where common properties such as `id`, `name`, `pickName`, `glClass`, `nodeData`, `fullNodeData` are defined.

The `nodeData` property contains a form control’s specific node hierarchy as defined in each concrete implementation of `GlObject3d`. ***Code 12*** shows the assignment of a node hierarchy representing a transparent sphere geometry to the `nodeData` of the `GlCheckBox` form control.

```
jQuery.extend(this.nodeData,
    {
        nodes: [
            {
                type:"flags", flags: { transparent: true },
                nodes:[
                    {
                        type: "material",
                        color: this.color,
                        alpha: this.opacity,
                        nodes: [
                            {
                                type: "geometry/sphere",
                                latudeBands: this.latudeBands,
                                longitudeBands: this.longitudeBands,
                                radius: this.radius
                            }
                        ]
                    }
                ]
            }
        ]
    }
);
```

Code 12 *- assignment of `GlCheckBox` specific node hierarchy to the base `nodeData` property*

The base property `fullNodeData` contains all transformation nodes and the `nodeData` node hierarchy (***Code 13***).

```
this.fullNodeData =
{
    id: this.id + '_full', type: 'name', name: this.pickName,
    nodes: [
        {
            id: this.id + '_enable', type: 'enable', enabled: true,
            nodes:[
                {
                    id: this.id + '_translation', type: "translate", x: 0, y: 0, z: 0,
                    nodes: [
                        {
                            id: this.id + '_rotation', type: "rotate",
                            x: 0, y: -1, z: 0, angle: 0,
                            nodes: [
                                {
                                    id: this.id + '_scale', type: "scale", x: 1, y: 1, z: 1,
                                    nodes: [
                                        this.nodeData
                                    ]
                                }
                            ]
                        }
                    ]
                }
            ]
        }
    ]
};
```

***Code 13** - assignment of the `fullNodeData` property in the base `GlObject3d` class*

The `fullNodeData` enables the necessary transformations of all objects which inherit from the base `GlObject3d` class. Without having manually declared the transformation nodes, SceneJS would not allow translation, scaling and rotation of geometries on their own.

Navigation

SceneJS supports the “orbit” and “trackball” camera types but none of them provided the functionality of the first person walk controls which were needed for our proof of concept. Because little literature and examples could be found online we had to implement a custom first person camera logic. This was achieved via SceneJS’s built-in `lookAt` node and a custom trigonometry algorithm. The built-in `lookAt` node allows setting the camera world position and view direction. To emulate first person controls, however, we needed to be able to rotate the camera by a given angle around the X and Y axises and to move it alongside and perpendicular to the last view direction vector. The conversion from rotation angles to view direction vector was done via a custom algorithm which used a combination of the sine and cosine of the camera’s yaw and pitch angles (***Code 14***).

```
function getDirection (pitchRad, yawRad) {
    var result = {
       x: Math.sin(yawRad) * Math.cos(pitchRad),
       y: Math.sin(pitchRad),
       z: Math.cos(yawRad) * Math.cos(pitchRad)
    };

    return result;
}
```

Code 14 *- algorithm for finding the camera direction vector from its pitch and yaw angles*

Event handling

The default object picking in SceneJS is a color based lgorithm. The `EventHandlerColorBased` class in the SceneJS concept is similar to the `EventHandlerColorBased` class of the Three.js implementation. Again we add each object which requires event listening to a list and once the user triggers the event we pick the name of the 3D object at the event's coordinates and try to find it in this list (***Code 15***). In SceneJS, the picking procedure is much simpler compared to Three.js since SceneJS manages internally the creation of clones of 3D objects and renders them automatically in a hidden target.

```
Pick: function (eventType, x, y) {
    var hit = EventHandler.context.glScene.scene.pick(x, y);

    if (hit) {

        for (var targetIndex in EventHandler[eventType + 'Targets']) {

            if (EventHandler[eventType + 'Targets'][targetIndex].pickName == hit.name) {

                return EventHandler[eventType + 'Targets'][targetIndex];

            }
        }
    }

    return null;
}
```

Code 15 *- the picking function of the SceneJS concept*

SceneJS does not support ray intersection object picking out of the box. The implementation of a ray based event handler in SceneJS was left for future research.

Achieved features

	SceneJS
3D text	
Text geometry	✔
Depth	X
Fonts	X
Bold, underline, strikethrough	X
Text selection and copying	X
Changing the value of an already created 3D text	✔
3D button	
A button geometry with text	✔
“OnClick“ event registration	✔
3D checkbox	
Main geometry with a togglable child “check” geometry	✔
“OnChange“ event registration	✔
3D radio button	
A radio button geometry with text	✔
“OnChange“ event registration	✔
Groups of radio buttons	✔
3D textbox	
A textbox geometry	✔
Typing of text after focus	✔
Text cursor	X
Text selection, copying and pasting	X
Dynamic right alignment of long text	X
Hiding overflown text	X
“OnFocus“ event registration	✔
“OnBlur“ event registration	✔
“OnKeyDown“ event registration	✔
“OnKeyPress“ event registration	✔
3D shelf	
A shelf geometry	✔
“MouseOver” event registration	✔
“MouseOut” event registration	✔
“DoubleClick” event registration	✔
Visual effects	
Transparency	✔
Specularity	✔
Lights	✔

***Table 6** - achieved features in the SceneJS implementation of the proof of concept*

Similar to the Three.js implementation following features were not achieved in SceneJS: text selection, copying and pasting, textbox cursor and hiding overflown text in a textbox.

Another functionality which was not implemented with SceneJS was the right alignment of the text in the `GlTextBox`. It was not possible to move the text of a `GlTextBox` to the left when it reaches the right border of the textbox since parent transformations in SceneJS do not work as expected. Dynamically added nodes to any transform parent behave inconsistently when the parent transform node is being updated in an user event. Sometimes the parent transformation works for all children and sometimes only for the last one.

Furthermore, text could not be given depth, could not be made bold and no fonts could be applied to it. Eventually, 22 out of the total 30 features were achieved in the proof of concept implementation using SceneJS (**Table 6**).

Performance

SceneJS exhibited very stable frame rate in the still tests in all browsers and at all levels of load. All browsers managed close to the maximum framerate of 60 FPS at loads of up to 10000 objects in the still test.

The number of objects had much stronger impact on performance in the interaction tests. Chrome kept the maximum possible frame rate of almost 60 FPS in the interaction tests at 50 and at 100 objects. Above 500 objects the frame rate in Chrome started gradually falling when eventually at 10000 objects it reached close to 0 FPS. Firefox showed similar behaviour with a little steeper fall in frame rate in the interaction tests. Internet Explorer exhibited the biggest difference between the still and interaction tests with keeping a frame rate of 55 FPS at all levels of load in the still test and a much lower frame rate starting at 6 FPS at 50 objects, 4 FPS at 100 objects, 3 FPS at 500 objects, 2 FPS at 1000 objects and 0 FPS for more than a 1000 objects (***Table 5, Chart 25 - Chart 42***).

# objects\browser	Chrome	Firefox	IE
50 objects	still: 60 fps mouseover: 59 fps	still: 59 fps mouseover: 60 fps	still: 55 fps mouseover: 6 fps
100 objects	still: 60 fps mouseover: 59 fps	still: 60 fps mouseover: 59 fps	still: 55 fps mouseover: 4 fps
500 objects	still: 60 fps mouseover: 57 fps	still: 59 fps mouseover: 52 fps	still: 55 fps mouseover: 3 fps
1000 objects	still: 60 fps mouseover: 49 fps	still: 59 fps mouseover: 43 fps	still: 55 fps mouseover: 2 fps
5000 objects	still: 59 fps mouseover: 20 fps	still: 58 fps mouseover: 9 fps	still: 54 fps mouseover: 0 fps
10000 objects	still: 57 fps mouseover: 2 fps	still: 54 fps mouseover: 4 fps	still: 54 fps mouseover: 0 fps

***Table 7** - performance measurements grid of the SceneJS proof of concept using SceneJS's default "color based event handler"*

The average frame rate of the Three.js proof of concept using the "color based event handler" in the still test was 57.39 FPS and 27.11 FPS in the interaction test. The overall average was 42.25 FPS.

Chrome

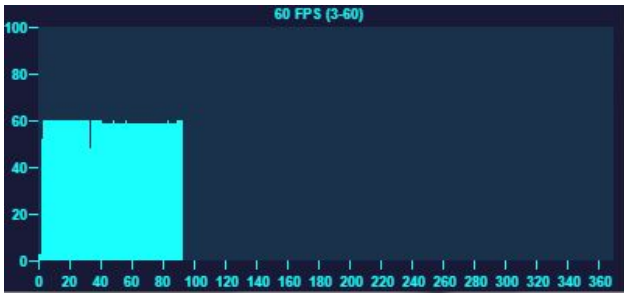

Chart 25 - *SceneJS POC, color based event handling, 50 objects, Chrome*

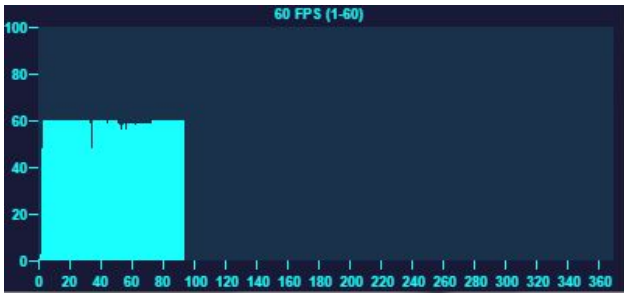

Chart 26 - *SceneJS POC, color based event handling, 100 objects, Chrome*

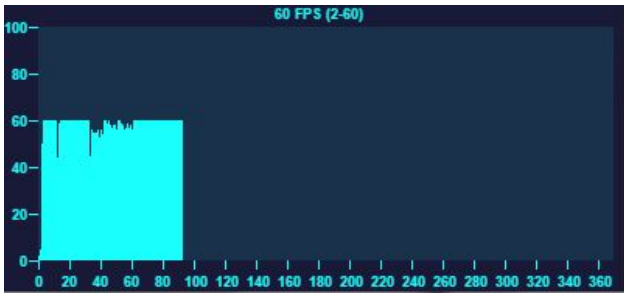

Chart 27 - *SceneJS POC, color based event handling, 500 objects, Chrome*

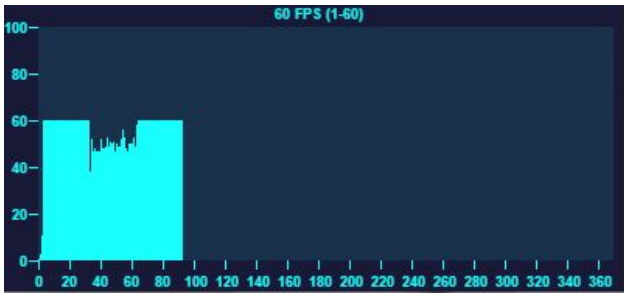

Chart 28 - *SceneJS POC, color based event handling, 1000 objects, Chrome*

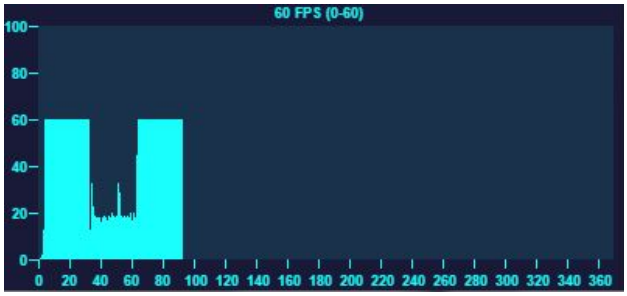

Chart 29 - *SceneJS POC, color based event handling, 5000 objects, Chrome*

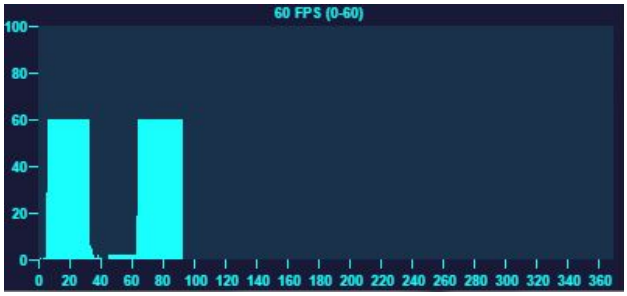

Chart 30 - *SceneJS POC, color based event handling, 10000 objects, Chrome*

Firefox

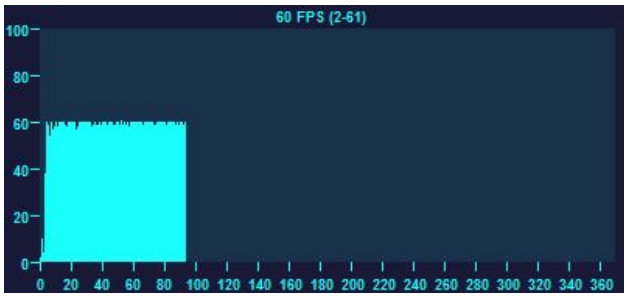

Chart 31 - *SceneJS POC, color based event handling, 50 objects, Firefox*

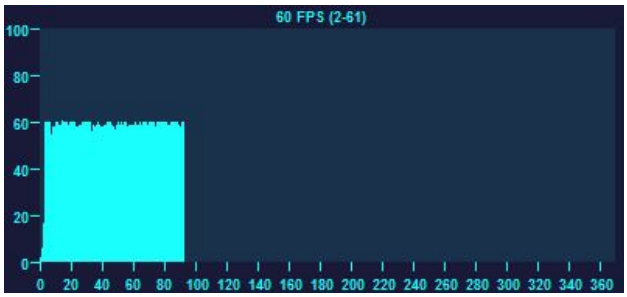

Chart 32 - *SceneJS POC, color based event handling, 100 objects, Firefox*

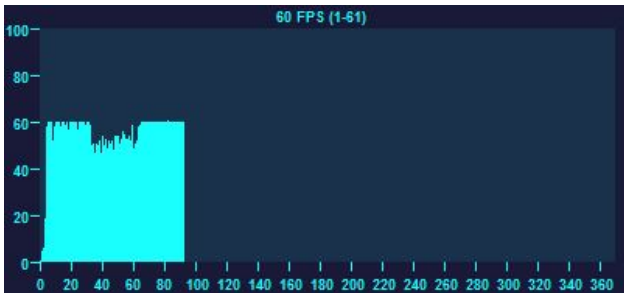

Chart 33 - *SceneJS POC, color based event handling, 500 objects, Firefox*

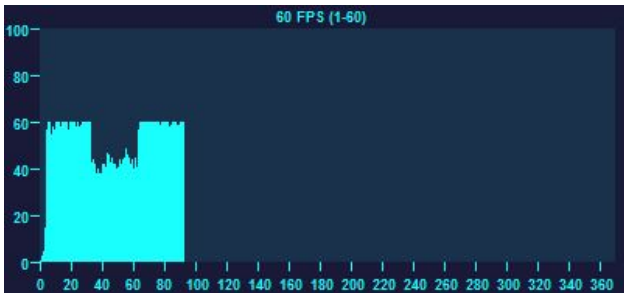

Chart 34 - *SceneJS POC, color based event handling, 1000 objects, Firefox*

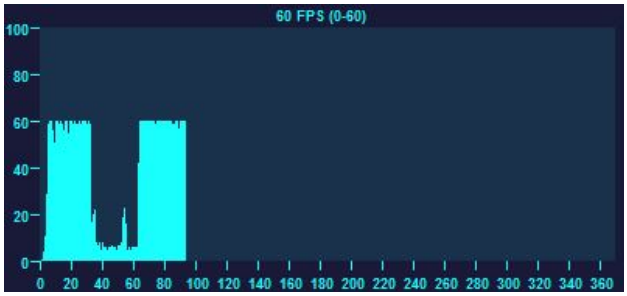

Chart 35 - *SceneJS POC, color based event handling, 5000 objects*

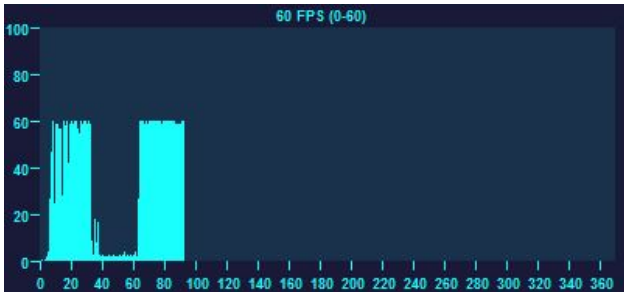

Chart 36 - *SceneJS POC, color based event handling, 10000 objects*

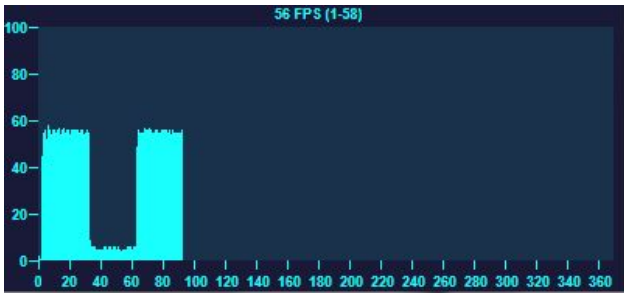

Chart 37 - *SceneJS POC, color based event handling, 50 objects, IE*

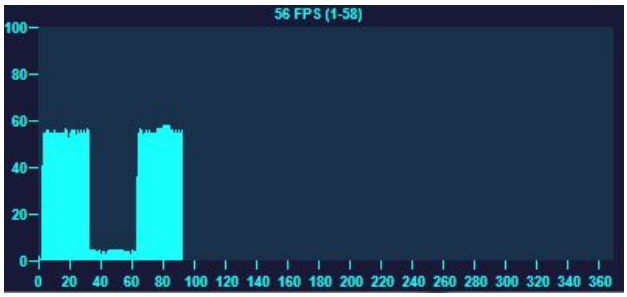

Chart 38 - *SceneJS POC, color based event handling, 100 objects, IE*

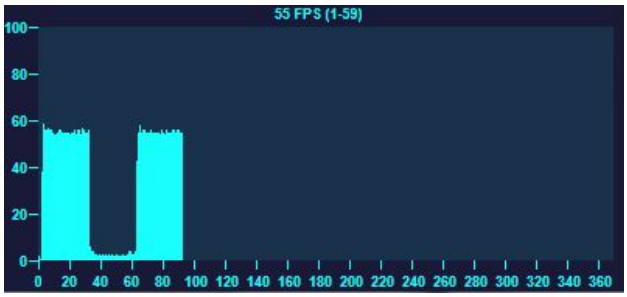

Chart 39 - SceneJS POC, color based event handling, 500 objects, IE

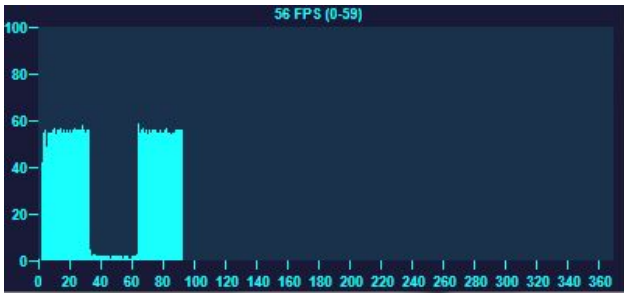

Chart 40 - *SceneJS POC, color based event handling, 1000 objects, IE*

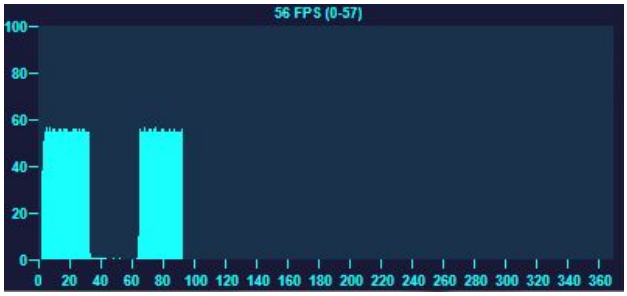

Chart 41 - *SceneJS POC, color based event handling, 5000 objects, IE*

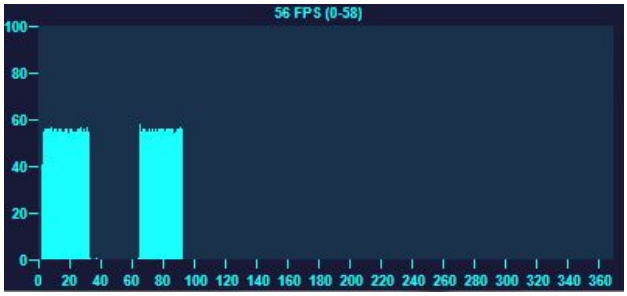

Chart 42 - *SceneJS POC, color based event handling, 10000 objects, IE*

Implementation size

Similarly to the Three.js implementation, the biggest part of code of the SceneJS implementation is located in the page. The page code in the two concepts, however, is almost identical since its logic is not framework specific.
Properties of 3D objects in SceneJS such as translation, rotation, scaling and visibility can only be managed if the 3D objects are manually wrapped in parent nodes of the correct type. Such parent node types are “translate”, “rotate”, “scale” and “enable”. The need to manually enable the transformation capabilities to all 3D objects increases the size of the classes where their geometries are created and managed such as `GlButton`, `GlCheckBox`, `GlObject3d`, `GlRadioButton` and `GlTextBox.`

Another big part of the SceneJS implementation is the navigation controls in the `WalkControls` class. It contains custom logic using trigonometry in order to achieve the first person walk navigation functionality. Its size is caused by the custom navigation algorithm which uses the sine and cosine of the yaw and pitch angles in order to determine the direction vector of the camera.

File	Physical SLOC
Page (js code)	309
EventHandlerColorBased.js	141
GlButton.js	92
GlCheckBox.js	154
GlContainer.js	60
GlContext.js	12
GlDiv.js	15
GlForm.js	33
GlObject3d.js	107
GlRadioButton.js	193
GlScene.js	38
GlShelf.js	47
GlShelfGeometry.js	53
GlText.js	82
GlTextBox.js	234
GlWavefrontObj.js	34
WalkControls.js	183
Total	**1787**

***Table 8** - lines of code per file of the SceneJS POC implementation*

Chapter 9 - Analysis and Conclusions

From the total of 30 features which were defined in the proof of concept design, we managed to implement 26 with Three.js and 22 with Scene.js (**Chart 43**). Text selection, copying and pasting in a textbox, text cursor in a textbox and hiding overflown text in a textbox were not achieved in both Three.js and SceneJS. They required considerable development effort compared to the rest of the features in the concept design and were therefore left for a separate project. The complexity of these functionalities, however, is not framework specific and once they have been implemented in one library, we expect them to be trivial to translate in others as well.

SceneJS failed to achieve 4 more features which Three.js did manage to implement. Support for text depth, text fonts and bold text is available in Three.js but not in SceneJS. SceneJS can be extended with a similar text visual management module but this would be a small project on its own and was left for future research.

Another issue encountered only with SceneJS was the inability to transform child nodes by transforming their parent in a custom event. For example, translating a parent when a click event is triggered, should translate all children as well. Sometimes, this is indeed the case and sometimes only the last child is translated. Since not much information can be found online about issues with SceneJS the only way to debug problems is to investigate the source code of the library.

In conclusion, Three.js provides a richer set of modules and solutions to common issues are easier to find online compared to SceneJS. On the other hand, the missing modules in SceneJS are not many and can be implemented in a reasonable timeframe. SceneJS already has a solid base which abstracts raw WebGL very well and allows for easy extension.

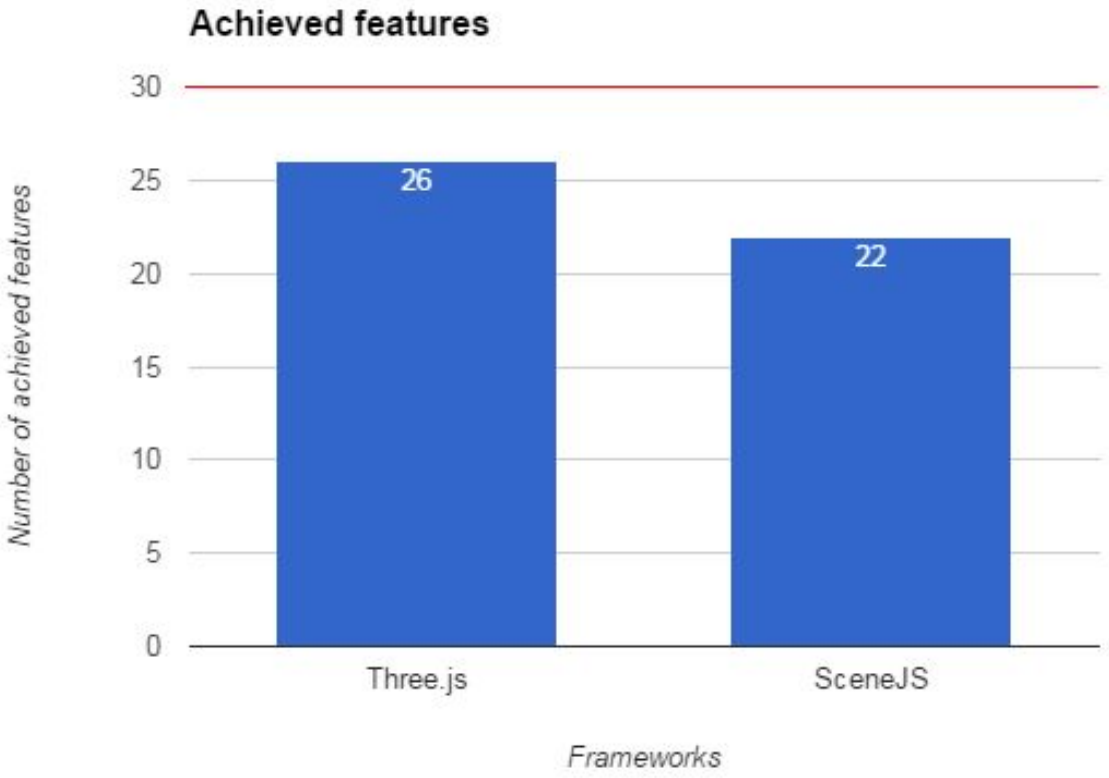

Chart 43 *- number of implemented features per framework out of the total 30 features in the proof of concept*

SceneJS exhibited considerable advantage over Three.js in our performance tests (**Chart 44**). Three.js maintained an average frame rate of around 20 FPS using the ray-based event handler and 17.6 FPS using the color-based event handler while SceneJS managed an average frame rate of around 42 FPS using its default color-based object picking method. It was interesting to see that the ray-based event handler in the Three.js implementation performed better than its color-based alternative while in the same time it underperformed considerably against SceneJS's color-based method. This means that SceneJS has advanced internal optimizations of its color-based picking method which are missing in our color-based method as implemented for the Three.js concept.

The biggest advantage for the SceneJS implementation was observed in the still tests where even at high loads of up to 5000 objects all browsers managed close to perfect frame rates between 55 and 60 FPS. This advantage was confirmed by our observation that while the Three.js implementation was running other applications' performance was significantly impacted. The SceneJS implementation, on the other hand, had very little influence on other running applications. A threat to the validity of the SceneJS performance measurements could be the event at which a frame tick was logged. As described on SceneJS example pages, the frame was logged on the scene's ”ontick” event although further investigation is needed to confirm that this is equivalent to Three.js's frame render event.

Another observation which makes SceneJS's case stronger is that it managed to render the concept application with 5000 objects on the scene in all browsers which was not achieved by the Three.js implementation.

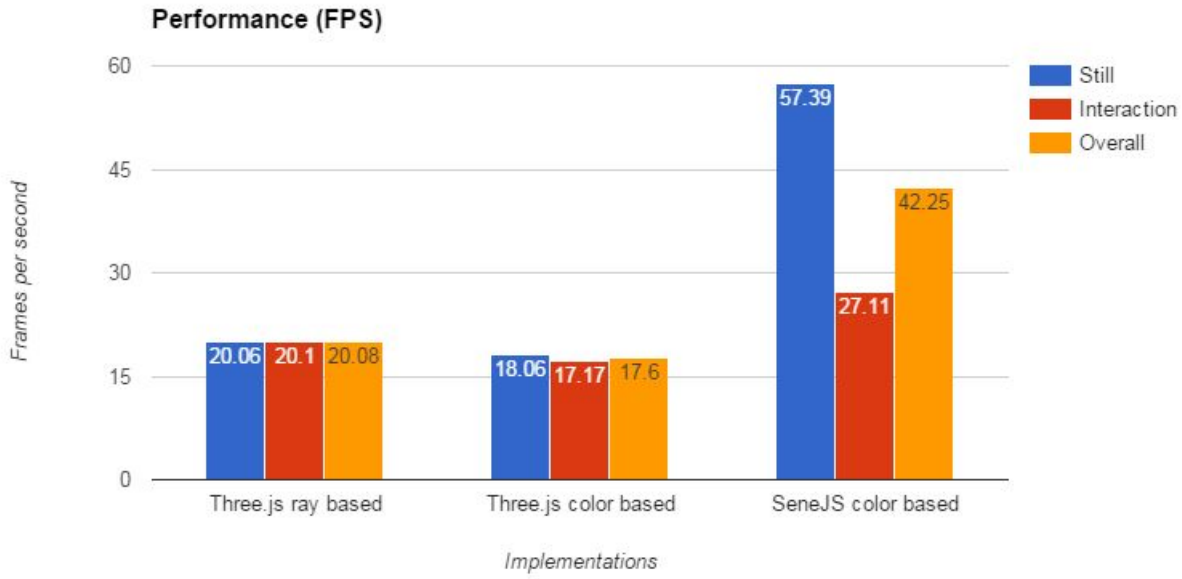

Chart 44 *- average frames per second per framework, event handler type and test type*

Although there were no dramatic differences in the size of our implementations, SceneJS did need more lines of code to achieve the same features. The SceneJS implementation was 1787 physical lines of code, the ray-based implementation with Three.js was 1360 lines of code and the color-based implementation with Three.js was 1488 lines of code (**Chart 45**). The larger size of the SceneJS implementation was caused mainly by the need to manually define the transformation nodes of each 3D object. This can be optimized by encapsulating the transformation nodes in a base class to reduce the size of the implementation.

Compared to other 2D executions of the TODOMVC application design, our implementations were considerably larger. Popular frameworks such as Angularjs, Knockoutjs and Jquery were all in the range of around 200 lines of code while ours were in the range of around 1500 lines of code. This cannot serve as a surprise since the 2D implementations are focusing on solving model-view-controller tasks and our implementations have the added responsibility of drawing the design in 3D space.

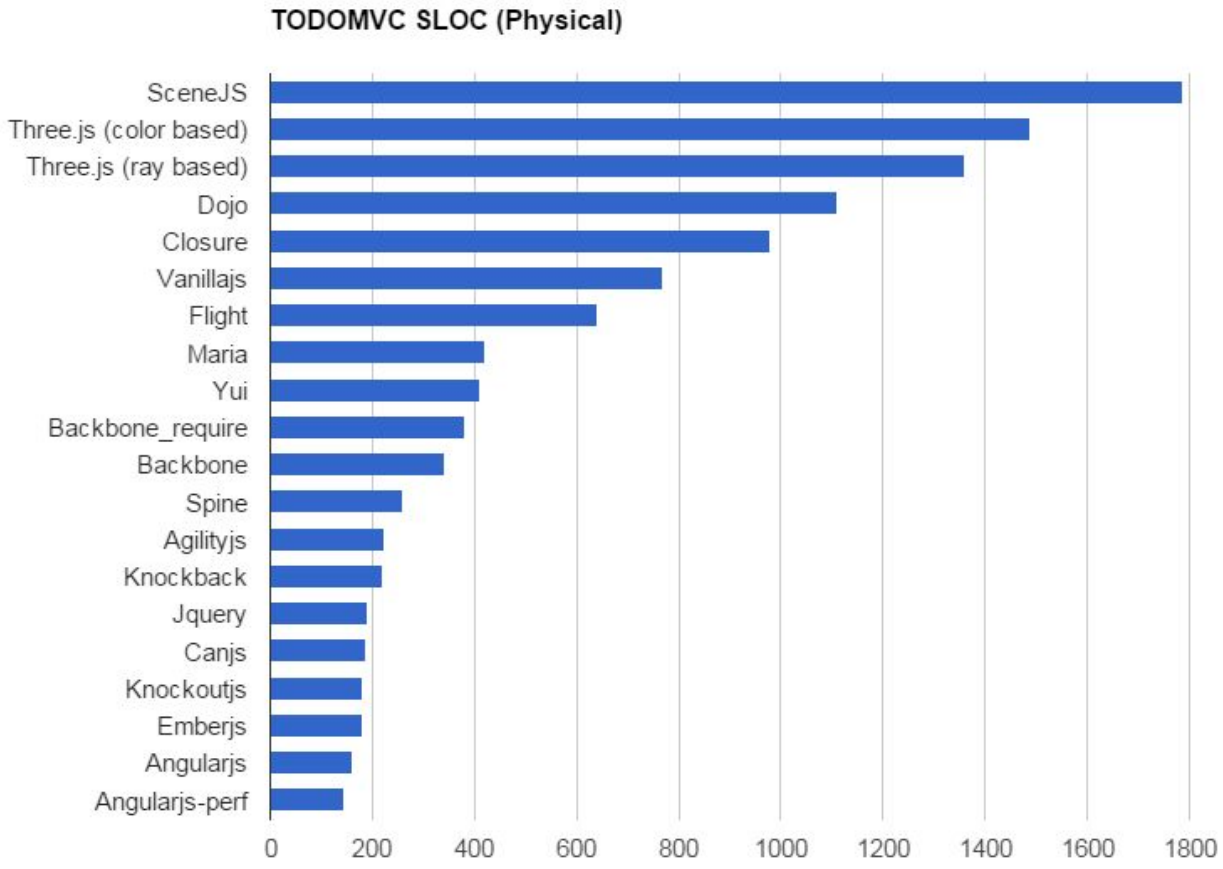

***Chart 45** - source lines of code of the TODOMVC application implementation in the different frameworks [TMVCLC]*

Qualitatively, SceneJS has a longer learning curve due to its non standard way of declaring and managing objects. SceneJS uses a JSON node hierarchy to describe and transform objects in 3D space while Three.js has a very intuitive structures to achieve the same (**Code 16**, **Code**

17). Another disadvantage with SceneJS is that object transformations are not directly available as properties of the object but rather need to be manually declared in the node hierarchy (**Code 17**). The asynchronous nature of SceneJS and the need to always use callback functions increases its complexity as well. Even referencing and managing an already existing object from the scene requires you to use a callback which is executed once the object has been found.

```
var boxGeometry = new THREE.BoxGeometry(1, 1, 1);

var boxMaterial = new THREE.MeshBasicMaterial({ color: '#FF0000' });

var boxMesh = new THREE.Mesh(boxGeometry, boxMaterial);

boxMesh.position.set(-5, 0, 0);

scene.add(boxMesh);
```

Code 16 *- creating and translating a box in Three.js*

```
var scene = SceneJS.createScene({
    nodes: [
        {
            id: "tr1", type: "translate"
            nodes: [
                {
                    type: "material", color: { r: 1, g: 0, b: 0 },
                    nodes: [
                        { type: "geometry/box", xSize: 1, ySize: 1, zSize: 1 }
                    ]
                }
            ]
        }
    ]
});

// finding and manipulating the translation node requires the use of a callback
scene.getNode("tr1", function(trNode) {trNode.setXYZ({x:-5, y: 0, z: 0});});
```

Code 17 *- creating and translating a box in SceneJS*

In conclusion, we found the considerable performance advantage of the SceneJS implementation to be more important than the few extra features which Three.js achieved. SceneJS can be extended with the missing modules more easily than Three.js can be made faster. On the other hand, if Three.js could be made faster it would be the choice of preference because of its popularity, availability of online information and intuitive usage.

Chapter 10 - Future work

Jonathan Karlsson came to the conclusion, in his research of WebGL TV user interfaces [Karlsson'14], that Three.js simplifies 3D web development but at the cost of performance. He proposed an idea on how the performance of Three.js could be improved by merging the separate 3D objects in a single geometry in order to reduce the amount of WebGL "draw" calls [TJSMRG][ATPRT][H5MIL][GGLIOWS'11]. In this section we try to build upon this idea and give details on exactly how this can be implemented.

The geometry of a 3D object is a collection of its vertex coordinates. To merge two geometries we need to combine their vertex coordinate arrays into a single array and then pass it to a single WebGl "draw" call. The WebGL "draw" call would then pass each vertex as an attribute to a vertex shader program [ATSHDR]. The vertex shader is a piece of GLSL [GLSLDCS][SHDRLB] code which runs on the graphics card of the computer for each vertex on each frame and tells the GPU how to translate 3D vertex coordinates to 2D screen coordinates. The `THREE.BufferGeometry` class provides a way to pass attributes to shaders in Three.js. We can, therefore, pass our combined vertex array as an attribute to a `BufferGeometry` (**Code 18**).

```
var sphereGeometry = new THREE.SphereGeometry(1);
var sphereBufferGeometry = new THREE.BufferGeometry().fromGeometry(sphereGeometry);
var spherePositionsArray = sphereBufferGeometry.attributes.position.array;

var boxGeometry = new THREE.BoxGeometry(1, 1, 1);
var boxBufferGeometry = new THREE.BufferGeometry().fromGeometry(boxGeometry);
var boxPositionsArray = boxBufferGeometry.attributes.position.array;

var fullGeometry = new THREE.BufferGeometry();

var positionsArray = concatFloat32Arrays(spherePositionsArray, boxPositionsArray);

fullGeometry.addAttribute('position', new THREE.BufferAttribute(positionsArray, 3));
```

***Code 18** - creating a merged geometry from a sphere and a box geometry and passing it as an attribute to the vertex shader via a `BufferGeometry`*

This allows us to have a single WebGL draw call but creates a problem with managing parts of the combined geometry. For example, if our combined geometry consists of a sphere and a box and we want to rotate only the box we cannot use the standard methods since Three.js does not know anymore of the existence of each part in the combined geometry. We can work around this issue by specifying the part id as an extra vertex shader attribute (**Code 19**). This tells the vertex shader which vertex belongs to which part of the combined geometry. For example, if we want to rotate only the box from our combined geometry we can give the vertex shader the id of the box and the desired angle of rotation (**Code 20**). The shader can then rotate each vertex which has the same id as the box (**Code 22**).

```
var sphereGeometry = new THREE.SphereGeometry(1);
var sphereBufferGeometry = new THREE.BufferGeometry().fromGeometry(sphereGeometry);
var spherePositionsArray = sphereBufferGeometry.attributes.position.array;

var idsArraySphere = new Float32Array(sphereBufferGeometry.attributes.position.length / 3);
for (var i = 0; i < idsArraySphere.length; i++) {
   idsArraySphere[i] = 1;
}


var boxGeometry = new THREE.BoxGeometry(1, 1, 1);
var boxBufferGeometry = new THREE.BufferGeometry().fromGeometry(boxGeometry);
var boxPositionsArray = boxBufferGeometry.attributes.position.array;

var idsArrayBox = new Float32Array(boxBufferGeometry.attributes.position.length / 3);
for (var i = 0; i < idsArrayBox .length; i++) {
   idsArrayBox [i] = 2;
}


var fullGeometry = new THREE.BufferGeometry();

var positionsArray = concatFloat32Arrays(spherePositionsArray, boxPositionsArray);

var idsArray = concatFloat32Arrays(idsArraySphere, idsArrayBox);

fullGeometry.addAttribute('position', new THREE.BufferAttribute(positionsArray, 3));

fullGeometry.addAttribute('id', new THREE.BufferAttribute(idsArray, 1));
```

Code 19 *- adding the id attribute to the* `BufferGeometry`

```
var rawShaderMaterial = new THREE.RawShaderMaterial({

    uniforms: {
        mutatedId: { type: "f", value: 1.0 },
        angle: { type: "f", value: 0.0 }
    },

    attributes: {
        id: { type: "f", value: null }
    },

    vertexShader: document.getElementById('vertexShader').textContent,

    fragmentShader: document.getElementById('fragmentShader').textContent

});
```

Code 20 *- passing the id of the part which we want to rotate to the vertex shader*

```
var fullMesh = new THREE.Mesh(fullGeometry, rawShaderMaterial);
scene.add(fullMesh);
```

***Code 21** - adding the combined geometry mesh to the scene*

```
<script id="vertexShader" type="x-shader/x-vertex">
            precision mediump float;
            precision mediump int;

            uniform mat4 modelViewMatrix;
            uniform mat4 projectionMatrix;
            uniform float mutatedId;
            uniform float angle;

            attribute vec3 position;
            attribute float id;

            varying float vId;

            void main()      {
                vId = id;
                if(id == mutatedId) {
                    mat4 rotMatrix =
                        mat4(
                        cos(angle), 0.0, -sin(angle), 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        sin(angle), 0.0, cos(angle), 0.0,
                        0.0, 0.0, 0.0, 1.0);

                    gl_Position = projectionMatrix *
                                  modelViewMatrix *
                                  rotMatrix *
                                  vec4( position, 1.0 );
                } else {
                    gl_Position = projectionMatrix *
                                  modelViewMatrix *
                                  vec4( position, 1.0 );
                }

            }
        </script>
```

***Code 22** - a vertex shader where vertices belonging to an object with the provided id can be rotated*

Although this method of combining and managing geometries does work, it deviates from the standard methods of object management in Three.js. Furthermore, the need for a custom shader makes the use of many built-in visual effects in Three.js very hard. Further research is necessary to find out if our performance optimization proposal can be integrated in the core of Three.js in a way that allows using the standard object management interfaces and visual effects while supporting single geometry rendering at the same time.

Bibliography

[Anyuru'12] "*Professional WebGL Programming: Developing 3D Graphics for the Web*", Andreas Anyuru, 2012

[ARBAB'14] "*3D graphics on the web: A survey*", Alun, Marco Romeo, Arash Bahrehmand, Javi Agenjo, Josep Blat, 2014

[ATPRT] http://aerotwist.com/tutorials/creating-particles-with-three-js

[ATSHDR] http://aerotwist.com/tutorials/an-introduction-to-shaders-part-1

[BJSDCS] http://doc.babylonjs.com

[BLNDR] https://www.blender.org

[BLV'11] "*Extending Web Applications with 3D Features*", Mario A. Bochicchio, Antonella Longo, Lucia Vaira, Set-Lab, Dept. of Innovation Engineering, University of Salento Lecce, Italy, 2011

[CR'02] "*Dynamic generation of personalized VRML content: a general approach and its application to 3D e-commerce*", L. Chittaro, and R. Ranon, Proc. of the 7th Int. Conf. on 3D Web Technology, p.145-154, February 24-28, 2002, Tempe, Arizona, USA

[CR'07] "*Web3D technologies in learning, education and motivations, issues, opportunities.*", Chittaro L, Ranon R., Comput Educ 2007;49(1):3–18.

[CVRDCS] https://github.com/cjcliffe/CubicVR.js/wiki/CubicVR.js-API-Reference

[CX'11] "*A framework for browser-based Multiplayer Online Games using WebGL and WebSocket*", Bijin Chen, Zhiqi Xu, 2011

[DBSTTS] https://github.com/mrdoob/stats.js

[DeLillo'10] "*WebGLU development library for WebGL*", Benjamin P. DeLillo, 2010

[GGLIOWS'11] http://webglsamples.org/google-io/2011/index.html

[GGLIOYT'11] http://www.youtube.com/watch?v=rfQ8rKGTVlg

[GLGEDCS] http://www.glge.org/api-docs

[GLSLDCS] https://www.opengl.org/documentation/glsl

[H5MIL] http://www.html5rocks.com/en/tutorials/webgl/million_letters

[HO'95] "*Critical characteristics of situated learning: implications for the instructional design of multimedia.*", Herrington J, Oliver R., In: Proceedings of the ASCILITE 1995; 1995. p. 253–62.

[JSMEV] http://marcgrabanski.com/simulating-mouse-click-events-in-javascript

[JWGLIN] http://www.jimmysoftware.net/slides/WebGL/index.html#19

[Karlsson'14] "*Using WebGL to create TV-centric user interfaces*", Jonathan Karlsson, 2014

[KTMVC] http://blog.krawaller.se/posts/a-reflux-todomvc-codebase-walkthrough

[MM'13] "*Designing a 3D Widget Library for WebGL Enabled Browsers*", Anna-Liisa Mattila, Tommi Mikkonen, 2013

[Ortiz'10] "*Is 3D Finally Ready for the Web?*", Ortiz, S., 2010

[PGLDCS] http://www.senchalabs.org/philogl/doc/index.html

[SGLDCS] http://www.spidergl.org/doc/0.2.0/index.html

[SHDRLB] http://codedstructure.net/projects/webgl_shader_lab

[SJSDCS] http://scenejs.org

[TJSDCS] http://threejs.org/docs

[TJSGPU] http://threejs.org/examples/webgl_interactive_cubes_gpu.html

[TJSMRG] http://learningthreejs.com/blog/2011/10/05/performance-merging-geometry

[TJSPL] https://github.com/mrdoob/three.js/blob/master/examples/misc_controls_pointerlock.html

[TJSRCD] http://threejs.org/docs/#Reference/Core/Raycaster

[TMVC] http://todomvc.com/

[TMVCLC] http://blog.coderstats.net/todomvc-complexity *(Author: Ramiro Gómez - ramiro.org)*

[VSJP'12] "*The influence of website dimensionality on customer experiences, perceptions and behavioral intentions: An exploration of 2D vs. 3D web design*", Lucian L. Visinescu, Anna Sidorova, Mary C. Jones, Victor R. Prybutok. 2012

[WGLCNTR] https://www.khronos.org/webgl/wiki/User_Contributions

[X3DCS] http://doc.x3dom.org/author/components.html

Appendix A - Important parts of the Three.js prototype implementation

EventHandlerColorBased.js

```
var EventHandler = {
    lastGlPickableObjectId: 0,
    eventFunctions: {},
    lastFocused: null,
    lastHovered: null,
    pickingRenderTarget: null,
    context: null,
    pickEventTypes: ['click', 'dblclick', 'mousemove'],
    rendererStats: null,

    Init: function (context) {
        EventHandler.context = context;

        EventHandler.rendererStats = new THREEx.RendererStats();
        EventHandler.rendererStats.domElement.style.position = 'absolute';
        EventHandler.rendererStats.domElement.style.left = '100px';
        EventHandler.rendererStats.domElement.style.bottom = '0px';
        document.body.appendChild(EventHandler.rendererStats.domElement);

        if (!EventHandler.pickingRenderTarget) {
            EventHandler.pickingRenderTarget = new THREE.WebGLRenderTarget(window.innerWidth,
window.innerHeight);
            EventHandler.pickingRenderTarget.minFilter = THREE.LinearFilter;
            EventHandler.pickingRenderTarget.generateMipmaps = false;
        }
    },

    AfterAddChild: function(child) {
        var hasChildPickEvents = EventHandler.HasPickEvents(child);
        if (hasChildPickEvents) {
            EventHandler.InitPickingObject3d(child);
        }
    },

    AddEventListener: function (eventType, eventTarget, fn) {

        if (!eventTarget[eventType + 'Events']) eventTarget[eventType + 'Events'] = [];
        if (fn) eventTarget[eventType + 'Events'].push(fn);
        eventTarget[eventType + 'EventsEnabled'] = true;

        var documentEventType = EventHandler.GetDocumentEventType(eventType);

        if (EventHandler.pickEventTypes.indexOf(eventType) > -1)
            EventHandler.InitPickingObject3d(eventTarget);

        if (eventType == 'click' || eventType == 'dblclick') {
            if (!EventHandler['mousemoveTargets']) EventHandler['mousemoveTargets'] = [];

            if (EventHandler['mousemoveTargets'].indexOf(eventTarget) < 0)
```

```
EventHandler['mousemoveTargets'].push(eventTarget);
        }

        if (!EventHandler[documentEventType + 'Targets']) EventHandler[documentEventType +
'Targets'] = [];

        if (EventHandler[documentEventType + 'Targets'].indexOf(eventTarget) < 0)
EventHandler[documentEventType + 'Targets'].push(eventTarget);

        if (!EventHandler.eventFunctions[documentEventType]) {

            EventHandler.eventFunctions[documentEventType] = function (event) {

                if (EventHandler.context.controlsManager.controls.enabled || (event.type == 'click'
&& EventHandler.context.controlsManager.isNavigating)) return;

                var eventTargetFound = null;
                if (EventHandler.pickEventTypes.indexOf(event.type) > -1) {
                    event.preventDefault();

                    eventTargetFound = EventHandler.Pick(event.type, event.clientX, event.clientY);

                    if (event.type == 'click') {

                        if (EventHandler.lastFocused && EventHandler.lastFocused.blurEvents &&
eventTargetFound != EventHandler.lastFocused) {
                            EventHandler.Trigger('blur', EventHandler.lastFocused, event);
                        }
                        EventHandler.Trigger('focus', eventTargetFound, event);
                    } else if (event.type == 'mousemove') {

                        if (eventTargetFound != EventHandler.lastHovered) {
                            if (EventHandler.lastHovered && EventHandler.lastHovered.mouseoutEvents
&& !EventHandler.IsParent(EventHandler.lastHovered, eventTargetFound)) {
                                EventHandler.Trigger('mouseout', EventHandler.lastHovered, event);
                            }

                            EventHandler.Trigger('mouseover', eventTargetFound, event);
                        }
                    }
                } else if (event.type == 'keydown' || event.type == 'keypress') {

                    for (var eventTargetFoundIndex in EventHandler[event.type + 'Targets']) {
                        var possibleEventTargetFound = EventHandler[event.type +
'Targets'][eventTargetFoundIndex];
                        if (possibleEventTargetFound.isFocused) eventTargetFound =
possibleEventTargetFound;
                    }
                }

                EventHandler.Trigger(event.type, eventTargetFound, event);

            };

            document.addEventListener(documentEventType,
EventHandler.eventFunctions[documentEventType], false);
```

```
        }
    },

    RemoveEventListener: function (eventType, eventTarget) {
        eventTarget[eventType + 'EventsEnabled'] = false;
        eventType = EventHandler.GetDocumentEventType(eventType);

        if (!EventHandler[eventType + 'Targets']) return;

        var eventTargetIndex = EventHandler[eventType + 'Targets'].indexOf(eventTarget);

        if (eventTargetIndex >= 0) EventHandler[eventType + 'Targets'].splice(eventTargetIndex, 1);

        EventHandler.RemovePickingObject3d(eventTarget);

        if (EventHandler[eventType + 'Targets'].length < 1) {
            document.removeEventListener(eventType, EventHandler.eventFunctions[eventType], false);
            EventHandler.eventFunctions[eventType] = null;
        }
    },

    Trigger: function (eventType, eventTarget, event) {

        if (!eventTarget || !eventTarget[eventType + 'Events'] || !eventTarget[eventType +
'EventsEnabled']) return;

        if (eventType == 'focus') EventHandler.lastFocused = eventTarget;
        if (eventType == 'mouseover') EventHandler.lastHovered = eventTarget;
        if (eventType == 'mouseout') EventHandler.lastHovered = null;

        for (var customEventIndex in eventTarget[eventType + 'Events']) {
            eventTarget[eventType + 'Events'][customEventIndex](event);
        }
    },

    GetDocumentEventType: function(eventType) {
        var documentEventType = eventType;
        if (documentEventType == 'focus' || documentEventType == 'blur') documentEventType =
'click';
        if (documentEventType == 'mouseover' || documentEventType == 'mouseout') documentEventType =
'mousemove';

        return documentEventType;
    },

    OnRender: function() {
        EventHandler.context.renderer.render(EventHandler.context.glScene.pickingMainObject3d,
EventHandler.context.camera, EventHandler.pickingRenderTarget);

        if (doRenderStats) EventHandler.rendererStats.update(EventHandler.context.renderer);
    },

    Pick: function(eventType, x, y) {

        var pixelBuffer = new Uint8Array( 4 );
```

```
        EventHandler.context.renderer.readRenderTargetPixels(EventHandler.pickingRenderTarget, x,
EventHandler.pickingRenderTarget.height - y, 1, 1, pixelBuffer);

        var id = getIdFromColor(pixelBuffer);
        for (var targetIndex = 0; targetIndex < EventHandler[eventType + 'Targets'].length;
targetIndex++) {
            if (EventHandler[eventType + 'Targets'][targetIndex].pickingId == id) {
                return EventHandler[eventType + 'Targets'][targetIndex];
            }
        }

        return null;
    },

    HasPickEvents: function(glObject) {
        var hasPickEvents = false;
        for (var eventTypeIndex in EventHandler.pickEventTypes) {
            var eventTypeTargets = EventHandler[EventHandler.pickEventTypes[eventTypeIndex] +
'Targets'];
            if (eventTypeTargets && eventTypeTargets.indexOf(glObject) > -1) {
                hasPickEvents = true;
                break;
            }
        }
        return hasPickEvents;
    },

    InitPickingObject3d: function (glObject) {

        if (!glObject.pickingMainObject3d) {

            if (glObject instanceof GlScene) {

                glObject.pickingMainObject3d = new THREE.Scene();

            } else {

                EventHandler.lastGlPickableObjectId++;

                glObject.pickingId = EventHandler.lastGlPickableObjectId;

                glObject.pickingMainMaterial = new THREE.MeshBasicMaterial({ color:
glObject.pickingId, side: THREE.DoubleSide });

                glObject.pickingMainObject3d = new THREE.Mesh(glObject.mainGeometry,
glObject.pickingMainMaterial);
            }

            glObject.pickingMainObject3d.position.copy(glObject.mainObject3d.position);
            glObject.pickingMainObject3d.rotation.copy(glObject.mainObject3d.rotation);
            glObject.pickingMainObject3d.scale.copy(glObject.mainObject3d.scale);
            glObject.pickingMainObject3d.visible = glObject.mainObject3d.visible;
        }

        if (glObject.parent) {
```

```
            EventHandler.InitPickingObject3d(glObject.parent);

            if(glObject.parent.pickingMainObject3d.children.indexOf(glObject.pickingMainObject3d) <
0)
                glObject.parent.pickingMainObject3d.add(glObject.pickingMainObject3d);
        }
    },

    RemovePickingObject3d: function (glObject) {
        if (!glObject) return;
        if (!glObject.pickingMainObject3d) return;
        if (glObject.pickingMainObject3d && glObject.pickingMainObject3d.children.length > 0)
return;

        var hasPickEvents = EventHandler.HasPickEvents(glObject);

        if (!hasPickEvents) {
            if (glObject.parent)
                glObject.parent.pickingMainObject3d.remove(glObject.pickingMainObject3d);

            glObject.pickingMainMaterial.dispose();
            glObject.pickingMainObject3d = null;
            glObject.pickingId = null;

            EventHandler.RemovePickingObject3d(glObject.parent);
        }
    },

    IsParent: function (potentialParent, child) {
        if (!potentialParent || !child) {
            return false;
        }

        if(child.parent === potentialParent) {
            return true;
        }

        return EventHandler.IsParent(potentialParent, child.parent);
    }
}
```

EventHandlerRayBased.js

```
var EventHandler = {

    eventFunctions: {},
    lastFocused: null,
    lastHovered: null,
    context: null,

    Init: function (context) {
        EventHandler.context = context;
    },
```

```
    AfterAddChild: function (child) {

    },

    OnRender: function () {
    },

    AddEventListener: function (eventType, eventTarget, fn) {

        if (!eventTarget[eventType + 'Events']) eventTarget[eventType + 'Events'] = [];
        if (fn) eventTarget[eventType + 'Events'].push(fn);
        eventTarget[eventType + 'EventsEnabled'] = true;

        var documentEventType = EventHandler.GetDocumentEventType(eventType);

        if (!EventHandler[documentEventType + 'Targets']) EventHandler[documentEventType +
'Targets'] = [];
        if (!EventHandler[documentEventType + 'TargetsMeshes']) EventHandler[documentEventType +
'TargetsMeshes'] = [];

        if (EventHandler[documentEventType + 'Targets'].indexOf(eventTarget) < 0)
EventHandler[documentEventType + 'Targets'].push(eventTarget);
        if (EventHandler[documentEventType + 'TargetsMeshes'].indexOf(eventTarget.mainMesh) < 0)
EventHandler[documentEventType + 'TargetsMeshes'].push(eventTarget.mainMesh);

        if (!EventHandler.eventFunctions[documentEventType]) {

            EventHandler.eventFunctions[documentEventType] = function (event) {

                if (EventHandler.context.controlsManager.controls.enabled || (event.type == 'click'
&& EventHandler.context.controlsManager.isNavigating)) return;

                var eventTargetFound = null;
                if (event.type == 'click' || event.type == 'dblclick' || event.type == 'mousemove')
{
                    event.preventDefault();
                    var intersects = getPerspectiveIntersects(event.clientX, event.clientY,
EventHandler.context, EventHandler[event.type + 'TargetsMeshes']);
                    if (intersects.length > 0 && intersects[0].object) {
                        eventTargetFound = EventHandler[event.type +
'Targets'][EventHandler[event.type + 'TargetsMeshes'].indexOf(intersects[0].object)];
                    }

                    if (event.type == 'click') {

                        if (EventHandler.lastFocused && EventHandler.lastFocused.blurEvents &&
eventTargetFound != EventHandler.lastFocused) {
                            EventHandler.Trigger('blur', EventHandler.lastFocused, event);
                        }
                        EventHandler.Trigger('focus', eventTargetFound, event);
                    } else if (event.type == 'mousemove') {

                        if (eventTargetFound != EventHandler.lastHovered) {
                            if (EventHandler.lastHovered && EventHandler.lastHovered.mouseoutEvents)
{
```

```
                            EventHandler.Trigger('mouseout', EventHandler.lastHovered, event);
                        }

                        EventHandler.Trigger('mouseover', eventTargetFound, event);
                    }
                }
            } else if (event.type == 'keydown' || event.type == 'keypress') {

                for (var eventTargetFoundIndex in EventHandler[event.type + 'Targets']) {
                    var possibleEventTargetFound = EventHandler[event.type +
'Targets'][eventTargetFoundIndex];
                    if (possibleEventTargetFound.isFocused) eventTargetFound =
possibleEventTargetFound;
                }
            }

            EventHandler.Trigger(event.type, eventTargetFound, event);

        };

        document.addEventListener(documentEventType,
EventHandler.eventFunctions[documentEventType], false);

    }
},

RemoveEventListener: function (eventType, eventTarget) {
    eventTarget[eventType + 'EventsEnabled'] = false;

    eventType = EventHandler.GetDocumentEventType(eventType);

    if (!EventHandler[eventType + 'Targets']) return;

    var eventTargetIndex = EventHandler[eventType + 'Targets'].indexOf(eventTarget);
    var eventTargetMeshIndex = EventHandler[eventType +
'TargetsMeshes'].indexOf(eventTarget.mainMesh);

    if (eventTargetIndex >= 0) EventHandler[eventType + 'Targets'].splice(eventTargetIndex, 1);
    if (eventTargetMeshIndex >= 0) EventHandler[eventType +
'TargetsMeshes'].splice(eventTargetMeshIndex, 1);

    if (EventHandler[eventType + 'Targets'].length < 1) {
        document.removeEventListener(eventType, EventHandler.eventFunctions[eventType], false);
        EventHandler.eventFunctions[eventType] = null;
    }
},

Trigger: function (eventType, eventTarget, event) {

    if (eventType == 'focus') EventHandler.lastFocused = eventTarget;
    if (eventType == 'mouseover') EventHandler.lastHovered = eventTarget;
    if (eventType == 'mouseout') EventHandler.lastHovered = null;

    if (!eventTarget || !eventTarget[eventType + 'Events'] || !eventTarget[eventType +
'EventsEnabled']) return;
```

```
        for (var customEventIndex in eventTarget[eventType + 'Events']) {
            eventTarget[eventType + 'Events'][customEventIndex](event);
        }
    },

    GetDocumentEventType: function(eventType) {
        var documentEventType = eventType;
        if (documentEventType == 'focus' || documentEventType == 'blur') documentEventType =
'click';
        if (documentEventType == 'mouseover' || documentEventType == 'mouseout') documentEventType =
'mousemove';

        return documentEventType;
    }
}
```

GlTextBox.js

```
GlTextBox = function (args) {
    GlObject3d.call(this);

    this.isFocused = false;
    this.boxWidth = 8;
    this.boxHeight = 2;
    this.boxDepth = 0.2;

    this.textColor = 0x4D4D4D;
    this.textMaterial;
    this.letterSize = this.boxHeight * 0.7;
    this.letterDepth = 0.2;
    this.textFont = 'helvetiker';
    this.textWeight = 'normal';
    this.textStyle = 'normal';
    this.letters = [];

    this.boxColor = 0x999999;
    this.boxOpacity = 1;

    this.visibleTextMesh;
    this.changeEvents = [];
    this.blurEvents = [];

    this.value;

    jQuery.extend(this, args);

    this.Init = function () {
        this.mainGeometry = new THREE.BoxGeometry(this.boxWidth, this.boxHeight, this.boxDepth);
        this.mainGeometry.faces.splice(8, 2);
        this.mainMaterial = new THREE.MeshPhongMaterial({ color: this.boxColor, opacity:
this.boxOpacity, transparent: true, vertexColors: THREE.FaceColors, side: THREE.DoubleSide,
specular: 0x050505, shininess: 100 });
        this.mainMesh = new THREE.Mesh(this.mainGeometry, this.mainMaterial);
```

```
        this.visibleTextMesh = new THREE.Mesh();
        this.mainMesh.add(this.visibleTextMesh);
        //this.mainGeometry.faces.splice(8, 2);

        this.mainObject3d = this.mainMesh;

        this.textMaterial = new THREE.MeshPhongMaterial({ color: this.textColor, overdraw: true,
specular: 0x050505, shininess: 100 });

        if (!this.value) this.value = '';
        this.htmlElement = $('<input>', { type: 'hidden', id: this.id, name: this.name, value:
this.value });
        this.context.form.AddFormControl(this);

        this.AddText(this.value);

        var self = this;
        EventHandler.AddEventListener('keydown', this, function (event) { self.OnKeyDown(event); });
        EventHandler.AddEventListener('keypress', this, function (event) { self.OnKeyPress(event);
});
        EventHandler.AddEventListener('focus', this, function (event) { self.GlFocus(event); });
        EventHandler.AddEventListener('blur', this, function (event) { self.GlBlur(event); });
    }

    this.OnKeyDown = function (event) {
        if (this.context.controlsManager.controls.enabled) return;

        if (this.isFocused) {
            var evt = event || window.event;
            var keycode = 0;
            if (evt) keycode = evt.which || evt.keyCode;

            var prevLetter = this.letters[this.letters.length - 1];

            if (keycode == 8) {
                if (prevLetter) {
                    this.letters.splice(this.letters.indexOf(prevLetter), 1);
                    this.SetValue(this.value.substring(0, this.value.length - 1));
                    this.visibleTextMesh.remove(prevLetter.mesh);

                    this.UpdateTextPosition();
                }
                evt.preventDefault();
            }
        }
    }

    this.OnKeyPress = function (event) {
        if (this.context.controlsManager.controls.enabled) return;

        if (this.isFocused) {

            var evt = event || window.event;
            var keycode = 0;
            if (evt) keycode = evt.which || evt.keyCode;
```

```
            if (keycode >= 9 && keycode <= 46 && keycode != 32) {

            } else {
                var charFromCode = String.fromCharCode(keycode);
                this.AddText(charFromCode);
                this.SetValue(this.value + charFromCode);
            }
        }
    }

    this.AddText = function (text) {
        for (var charIndex in text) {
            var character = text[charIndex];

            var textGeometry = new THREE.TextGeometry(character, { size: this.letterSize, height:
this.letterDepth, curveSegments: 3, font: this.textFont, weight: this.textWeight, style:
this.textStyle });
            var textMesh = new THREE.Mesh(textGeometry, this.textMaterial);

            var letterXpos = -this.boxWidth / 2 + 0.05;
            var prevLetter = this.letters[this.letters.length - 1];
            if (prevLetter) {
                letterXpos = prevLetter.mesh.position.x + this.letterSize + 0.05;
            }

            textMesh.position.x = letterXpos;
            textMesh.position.y = -this.boxHeight / 2 + this.boxHeight * 0.15;
            textMesh.position.z = -this.boxDepth * 0.1;

            this.visibleTextMesh.add(textMesh);
            this.letters.push(new InputLetter(character, textMesh));
        }

        this.UpdateTextPosition();
    }

    this.UpdateTextPosition = function () {
        var totalTextWidth = this.letters.length * (this.letterSize + 0.3);
        var textLeftOverSpaceX = this.boxWidth - totalTextWidth;
        if (textLeftOverSpaceX < 0) {
            this.visibleTextMesh.position.x = textLeftOverSpaceX;
        } else {
            this.visibleTextMesh.position.x = 0;
        }
    }

    this.GlFocus = function () {
        this.isFocused = true;
    }

    this.GlBlur = function () {
        this.isFocused = false;
    }

    this.SetValue = function(value) {
```

```
        this.value = value;
        this.htmlElement.attr('value', value);

        for (var customEventIndex in this.changeEvents) {
            this.changeEvents[customEventIndex]();
        }
    }

    this.Remove = function () {
        EventHandler.RemoveEventListener('keydown', this);
        EventHandler.RemoveEventListener('keypress', this);
        EventHandler.RemoveEventListener('focus', this);
        EventHandler.RemoveEventListener('blur', this);
        this.htmlElement.remove();

        GlObject3d.prototype.Remove.call(this);
    }

    this.SetDisplay = function (display) {
        if (display) {
            EventHandler.AddEventListener('keydown', this, null);
            EventHandler.AddEventListener('keypress', this, null);
            EventHandler.AddEventListener('focus', this);
            EventHandler.AddEventListener('blur', this);
        } else {
            EventHandler.RemoveEventListener('keydown', this);
            EventHandler.RemoveEventListener('keypress', this);
            EventHandler.RemoveEventListener('focus', this);
            EventHandler.RemoveEventListener('blur', this);
        }

        GlObject3d.prototype.SetDisplay.call(this, display);
    }

    this.Init();
}

GlTextBox.prototype = Object.create(GlObject3d.prototype);
GlTextBox.prototype.constructor = GlTextBox;

InputLetter = function (letter, mesh) {
    this.mesh = mesh;
    this.letter = letter;
}
```

Utils.js

```
function cloneMesh(mesh) {
    var result = new THREE.Mesh(mesh.geometry, mesh.material);
    result.position = mesh.position;
    return result;
}

function applyVertexColors(g, c) {
    g.faces.forEach(function (f) {
```

```
        var n = (f instanceof THREE.Face3) ? 3 : 4;
        for (var j = 0; j < n; j++) {
            f.vertexColors[j] = c;
        }
    });
}


function getPerspectiveIntersects(x, y, context, testObjects) {
    var raycaster = new THREE.Raycaster();
    var mouse = new THREE.Vector2();
    mouse.x = (x / context.renderer.domElement.width) * 2 - 1;
    mouse.y = -(y / context.renderer.domElement.height) * 2 + 1;

    raycaster.ray.origin.copy(context.controlsManager.controls.getObject().position);
    raycaster.ray.direction.set(mouse.x, mouse.y,
0.5).unproject(context.camera).sub(context.controlsManager.controls.getObject().position).normalize(
);

    //raycaster.setFromCamera(mouse, camera);

    var intersects = raycaster.intersectObjects(testObjects);

    return intersects;
}
```

Appendix B - Important parts of the SceneJS prototype implementation

EventHandlerColorBased.js

```
var EventHandler = {
    context: null,
    eventFunctions: {},
    lastFocused: null,
    lastHovered: null,
    pickEventTypes: ['click', 'dblclick', 'mousemove'],

    Init: function (context) {
        EventHandler.context = context;
    },

    OnRender: function () {
    },

    AddEventListener: function (eventType, eventTarget, fn) {
        if (!eventTarget[eventType + 'Events']) eventTarget[eventType + 'Events'] = [];
        if (fn) eventTarget[eventType + 'Events'].push(fn);
        eventTarget[eventType + 'EventsEnabled'] = true;

        var documentEventType = EventHandler.GetDocumentEventType(eventType);

        if (eventType == 'click' || eventType == 'dblclick') {
            if (!EventHandler['mousemoveTargets']) EventHandler['mousemoveTargets'] = [];

            if (EventHandler['mousemoveTargets'].indexOf(eventTarget) < 0)
EventHandler['mousemoveTargets'].push(eventTarget);
        }
```

```
        if (!EventHandler[documentEventType + 'Targets']) EventHandler[documentEventType +
'Targets'] = [];

        if (EventHandler[documentEventType + 'Targets'].indexOf(eventTarget) < 0)
EventHandler[documentEventType + 'Targets'].push(eventTarget);

        if (!EventHandler.eventFunctions[documentEventType]) {

            EventHandler.eventFunctions[documentEventType] = function (event) {

                if (EventHandler.context.controlsManager.isEnabled || (event.type == 'click' &&
EventHandler.context.controlsManager.isNavigating)) return;

                var eventTargetFound = null;
                if (EventHandler.pickEventTypes.indexOf(event.type) > -1) {
                    event.preventDefault();

                    eventTargetFound = EventHandler.Pick(event.type, event.clientX, event.clientY);

                    if (event.type == 'click') {

                        if (EventHandler.lastFocused && EventHandler.lastFocused.blurEvents &&
eventTargetFound != EventHandler.lastFocused) {
                            EventHandler.Trigger('blur', EventHandler.lastFocused, event);
                        }
                        EventHandler.Trigger('focus', eventTargetFound, event);
                    } else if (event.type == 'mousemove') {

                        if (eventTargetFound != EventHandler.lastHovered) {
                            if (EventHandler.lastHovered && EventHandler.lastHovered.mouseoutEvents
&& !EventHandler.IsParent(EventHandler.lastHovered, eventTargetFound)) {
                                EventHandler.Trigger('mouseout', EventHandler.lastHovered, event);
                            }

                            EventHandler.Trigger('mouseover', eventTargetFound, event);
                        }
                    }

                } else if (event.type == 'keydown' || event.type == 'keypress') {

                    for (var eventTargetFoundIndex in EventHandler[event.type + 'Targets']) {
                        var possibleEventTargetFound = EventHandler[event.type +
'Targets'][eventTargetFoundIndex];
                        if (possibleEventTargetFound.isFocused) eventTargetFound =
possibleEventTargetFound;
                    }
                }

                EventHandler.Trigger(event.type, eventTargetFound, event);
            };

            document.addEventListener(documentEventType,
EventHandler.eventFunctions[documentEventType], false);

        }

    },

    RemoveEventListener: function (eventType, eventTarget) {
        eventTarget[eventType + 'EventsEnabled'] = false;

        eventType = EventHandler.GetDocumentEventType(eventType);

        if (!EventHandler[eventType + 'Targets']) return;
```

```
        var eventTargetIndex = EventHandler[eventType + 'Targets'].indexOf(eventTarget);

        if (eventTargetIndex >= 0) EventHandler[eventType + 'Targets'].splice(eventTargetIndex, 1);

        if (EventHandler[eventType + 'Targets'].length < 1) {
            document.removeEventListener(eventType, EventHandler.eventFunctions[eventType], false);
            EventHandler.eventFunctions[eventType] = null;
        }
    },

    Trigger: function (eventType, eventTarget, event) {

        if (!eventTarget || !eventTarget[eventType + 'Events'] || !eventTarget[eventType +
'EventsEnabled']) return;

        if (eventType == 'focus') EventHandler.lastFocused = eventTarget;
        if (eventType == 'mouseover') EventHandler.lastHovered = eventTarget;
        if (eventType == 'mouseout') EventHandler.lastHovered = null;

        for (var customEventIndex in eventTarget[eventType + 'Events']) {
            eventTarget[eventType + 'Events'][customEventIndex](event);
        }
    },

    GetDocumentEventType: function(eventType) {
        var documentEventType = eventType;
        if (documentEventType == 'focus' || documentEventType == 'blur') documentEventType =
'click';
        if (documentEventType == 'mouseover' || documentEventType == 'mouseout') documentEventType =
'mousemove';

        return documentEventType;
    },

    Pick: function (eventType, x, y) {
        var hit = EventHandler.context.glScene.scene.pick(x, y);

        if (hit) {
            for (var targetIndex = 0; targetIndex < EventHandler[eventType + 'Targets'].length;
targetIndex++) {
                if (EventHandler[eventType + 'Targets'][targetIndex].pickName == hit.name) {
                    return EventHandler[eventType + 'Targets'][targetIndex];
                }
            }
        }

        return null;
    },

    IsParent: function (potentialParent, child) {
        if (!potentialParent || !child) {
            return false;
        }

        if (child.parent === potentialParent) {
            return true;
        }

        return EventHandler.IsParent(potentialParent, child.parent);
    }
}
```

GlObject3d.js

```
GlObject3d = function (args) {
    this.parent;
    this.context;

    this.id;
    this.name;
    this.pickName;
    this.glClass;
    this.nodeData;
    this.fullNodeData;

    jQuery.extend(this, args);

    this.SetProperties = function (properties) {

        if (properties.translation) this.GetTranslationNode(function (translationNode) {
translationNode.set(properties.translation); });
        if (properties.rotation) this.GetRotationNode(function (rotationNode) {
rotationNode.set(properties.rotation); });
        if (properties.scale) this.GetScaleNode(function (scaleNode) {
scaleNode.set(properties.scale); });

    }

    this.GetNode = function (fn) {
        var scene = this.scene ? this.scene : this.context.glScene.scene;
        var result = scene.getNode(this.id, fn);
        return result;
    }

    this.GetFullNode = function (fn) {
        var scene = this.scene ? this.scene : this.context.glScene.scene;
        var result = scene.getNode(this.id + '_full', fn);
        return result;
    }

    this.GetEnableNode = function (fn) {
        var scene = this.scene ? this.scene : this.context.glScene.scene;
        var result = scene.getNode(this.id + '_enable', fn);
        return result;
    }

    this.GetTranslationNode = function (fn) {
        var result = this.context.glScene.scene.getNode(this.id + '_translation', fn);
        return result;
    }

    this.GetRotationNode = function (fn) {
        var result = this.context.glScene.scene.getNode(this.id + '_rotation', fn);
        return result;
    }

    this.GetScaleNode = function (fn) {
        var result = this.context.glScene.scene.getNode(this.id + '_scale', fn);
        return result;
    }
}

GlObject3d.prototype.Init = function () {

    this.context.controlIdCounter++;
```

```
    if (!this.id) this.id = 'control' + this.context.controlIdCounter;
    if (!this.pickName) this.pickName = this.id + '_pickName';

    this.nodeData = { id: this.id };

    this.fullNodeData =
    {
        id: this.id + '_full', type: 'name', name: this.pickName,
        nodes: [
            {
                id: this.id + '_enable', type: 'enable', enabled: true,
                nodes:[
                    {
                        id: this.id + '_translation', type: "translate", x: 0, y: 0, z: 0,
                        nodes: [
                            {
                                id: this.id + '_rotation', type: "rotate", x: 0, y: -1, z: 0, angle:
0,
                                nodes: [
                                    {
                                        id: this.id + '_scale', type: "scale", x: 1, y: 1, z: 1,
                                        nodes: [
                                            this.nodeData
                                        ]
                                    }
                                ]
                            }
                        ]
                    }
                ]
            }
        ]
    };
}

GlObject3d.prototype.SetDisplay = function (display) {

    this.GetEnableNode(function (enableNode) { enableNode.setEnabled(display); });

}

GlObject3d.prototype.Remove = function () {
    this.GetFullNode(function (fullNode) { fullNode.destroy(); });

    if (this.parent) {
        this.parent.children.splice(this.parent.children.indexOf(this), 1);
        this.parent = null;
    }
}
```

GlTextBox.js

```
GlTextBox = function (args) {
    GlObject3d.call(this);

    this.isFocused = false;
    this.boxWidth = 4;
    this.boxHeight = 2;
    this.boxDepth = 0.2;

    this.textColor = '#000000';
    this.letterSize = this.boxHeight * 0.7;
```

```
    this.letterDepth = 0.2;

    this.letters = [];

    this.boxColor = '#999999';
    this.boxOpacity = 1;

    this.changeEvents = [];
    this.blurEvents = [];

    this.value;
    this.letterCounter = 0;

    jQuery.extend(this, args);

    this.Init = function () {
        GlObject3d.prototype.Init.call(this);

        jQuery.extend(this.nodeData,
            {
                nodes: [
                    {
                        type: "flags", flags: { transparent: true },
                        nodes: [
                            {
                                type: "material", color: hexToOneBaseRgb(this.boxColor), alpha:
this.boxOpacity,
                                nodes: [new GlShelfGeometry({ width: this.boxWidth, height:
this.boxHeight, depth: this.boxDepth })]
                            }
                        ]
                    }
                ]
            }
        );

        this.nodeData.nodes.push({ id: this.GetVisibleTextNodeId(), type: "translate", x: 0, y: 0,
z: 0 });

        if (!this.value) this.value = '';
        this.htmlElement = $('<input>', { type: 'hidden', id: this.id, name: this.name, value:
this.value });
        this.context.form.AddFormControl(this);

        this.AddText(this.value);

        var self = this;
        EventHandler.AddEventListener('keydown', this, function (event) { self.OnKeyDown(event); });
        EventHandler.AddEventListener('keypress', this, function (event) { self.OnKeyPress(event);
});
        EventHandler.AddEventListener('focus', this, function (event) { self.GlFocus(event); });
        EventHandler.AddEventListener('blur', this, function (event) { self.GlBlur(event); });
    }

    this.GetVisibleTextNodeId = function () {
        var result = this.id + '_visibleTextNode';
        return result;
    }

    this.GetVisibleTextNode = function (fn) {
        var result = this.context.glScene.scene.getNode(this.GetVisibleTextNodeId(), fn);
        return result;
    }

    this.OnKeyDown = function (event) {
```

```
        var thisPublic = this;

        if (this.context.controlsManager.isEnabled) return;

        if (this.isFocused) {
            var evt = event || window.event;
            var keycode = 0;
            if (evt) keycode = evt.which || evt.keyCode;

            var prevLetter = this.letters[this.letters.length - 1];

            if (keycode == 8) {
                if (prevLetter) {
                    this.letters.splice(this.letters.indexOf(prevLetter), 1);
                    this.SetValue(this.value.substring(0, this.value.length - 1));

                    this.context.glScene.scene.getNode(prevLetter.letterNodeId, function(letterNode)
{
                        letterNode.destroy();
                        thisPublic.UpdateTextPosition();
                    });
                }

                evt.preventDefault();
            }
        }
    }

    this.OnKeyPress = function (event) {
        if (this.context.controlsManager.isEnabled) return;

        if (this.isFocused) {

            var evt = event || window.event;
            var keycode = 0;
            if (evt) keycode = evt.which || evt.keyCode;

            if (keycode >= 9 && keycode <= 46 && keycode != 32) {

            } else {
                var charFromCode = String.fromCharCode(keycode);
                this.AddText(charFromCode);
                this.SetValue(this.value + charFromCode);
            }
        }
    }

    this.AddText = function (text) {
        var self = this;

        this.GetVisibleTextNode(function (visibleTextNode) {

            for (var charIndex in text) {
                var character = text[charIndex];

                self.letterCounter++;
                var letterNodeId = self.id + '_letter' + self.letterCounter;
                var inputLetter = new InputLetter(character, letterNodeId);
                self.letters.push(inputLetter);

                var letterXpos = -self.boxWidth / 2 + 0.05;
                var prevLetter = self.letters[self.letters.length - 2];
                if (prevLetter) {
                    var letterNode = self.context.glScene.scene.getNode(prevLetter.letterNodeId);
                    letterXpos = letterNode.getX() + self.letterSize + 0.05;
```

```
                }

                var letterFullNodeData = {
                    id: inputLetter.letterNodeId, type: "translate", x: letterXpos, y:
-self.boxHeight / 2 + self.boxHeight * 0.15, z: -self.boxDepth * 0.1,
                    nodes: [
                        {
                            type: "scale", x: self.letterSize, y: self.letterSize, z:
self.letterDepth,
                            nodes: [
                                {
                                    type: "material", color: hexToOneBaseRgb(self.textColor),
                                    nodes: [{ type: "geometry/vectorText", text: character }]
                                }
                            ]
                        }
                    ]
                };

                visibleTextNode.addNode(letterFullNodeData);
            }

            if (text && text.length > 0) self.UpdateTextPosition();
        });
    }

    this.UpdateTextPosition = function () {
        return; //adjusting the translation of nodes which have been dynamically added breaks the
position of all children of the parent

        var thisPublic = this;
        var totalTextWidth = this.letters.length * (this.letterSize + 0.05);
        var textLeftOverSpaceX = this.boxWidth - totalTextWidth;

        this.GetVisibleTextNode(function (visibleTextNode) {

            if (textLeftOverSpaceX < 0) {
                visibleTextNode.setX(textLeftOverSpaceX);
            } else {
                visibleTextNode.setX(thisPublic.GetTranslationNode().getX() - thisPublic.boxWidth /
2 + 0.1);
            }
        });
    }

    this.GlFocus = function () {
        this.isFocused = true;
    }

    this.GlBlur = function () {
        this.isFocused = false;
    }

    this.SetValue = function(value) {
        this.value = value;
        this.htmlElement.attr('value', value);

        for (var customEventIndex in this.changeEvents) {
            this.changeEvents[customEventIndex]();
        }
    }

    this.Remove = function () {
        EventHandler.RemoveEventListener('keydown', this);
        EventHandler.RemoveEventListener('keypress', this);
```

```
        EventHandler.RemoveEventListener('focus', this);
        EventHandler.RemoveEventListener('blur', this);
        EventHandler.RemoveEventListener('click', this);
        EventHandler.RemoveEventListener('dblclick', this);
        this.htmlElement.remove();

        GlObject3d.prototype.Remove.call(this);
    }

    this.SetDisplay = function (display) {
        if (display) {
            EventHandler.AddEventListener('keydown', this, null);
            EventHandler.AddEventListener('keypress', this, null);
            EventHandler.AddEventListener('focus', this, null);
            EventHandler.AddEventListener('blur', this, null);
            EventHandler.AddEventListener('click', this, null);
            EventHandler.AddEventListener('dblclick', this, null);
        } else {
            EventHandler.RemoveEventListener('keydown', this);
            EventHandler.RemoveEventListener('keypress', this);
            EventHandler.RemoveEventListener('focus', this);
            EventHandler.RemoveEventListener('blur', this);
            EventHandler.RemoveEventListener('click', this);
            EventHandler.RemoveEventListener('dblclick', this);
        }

        GlObject3d.prototype.SetDisplay.call(this, display);
    }

    this.Init();
}

GlTextBox.prototype = Object.create(GlObject3d.prototype);
GlTextBox.prototype.constructor = GlTextBox;

InputLetter = function (letter, letterNodeId) {
    this.letterNodeId = letterNodeId;
    this.letter = letter;
}
```

WalkControls.js

```
WalkControls = function (args) {

    this.isEnabled = false;
    this.isNavigating = false;
    this.glScene;

    this.prevTime = performance.now();
    this.velocity = { x: 0, y: 0, z: 0 };
    this.moveForward = false;
    this.moveBackward = false;
    this.moveLeft = false;
    this.moveRight = false;
    this.lookAround = { lastMouseDownX: null, lastMouseDownY: null, lastMouseUpX: null,
lastMouseUpY: null };

    this.lastMouseMoveX = null;
    this.lastMouseMoveY = null;
    this.pitch = 0;
    this.yaw = Math.PI;
```

```
    this.lookRotateSensitivity = 0.002;

    jQuery.extend(this, args);

    this.moveEye = function () {
        var self = this;
        this.glScene.GetCameraNode(
            function (cameraNode) {

                if (self.velocity.z != 0) {
                    var moveVector = getDirection(self.pitch, self.yaw);
                    cameraNode.incEyeX(-moveVector.x * self.velocity.z);
                    cameraNode.incLookX(-moveVector.x * self.velocity.z);
                    cameraNode.incEyeZ(-moveVector.z * self.velocity.z);
                    cameraNode.incLookZ(-moveVector.z * self.velocity.z);
                }

                if (self.velocity.x != 0) {
                    var moveVectorLateral = getDirection(self.pitch, self.yaw + Math.PI / 2);
                    cameraNode.incEyeX(-moveVectorLateral.x * self.velocity.x);
                    cameraNode.incLookX(-moveVectorLateral.x * self.velocity.x);
                    cameraNode.incEyeZ(-moveVectorLateral.z * self.velocity.x);
                    cameraNode.incLookZ(-moveVectorLateral.z * self.velocity.x);
                }
            }
        );
    }

    this.Reset = function () {
        this.prevTime = performance.now();
        this.moveForward = false;
        this.moveBackward = false;
        this.moveLeft = false;
        this.moveRight = false;
        this.velocity = { x: 0, y: 0, z: 0 };
    }

    this.moveCameraOnYBy = function (yDelta) {
        var self = this;
        self.glScene.GetCameraNode(
            function (cameraNode) {
                if (yDelta != 0) {
                    cameraNode.incEyeY(yDelta);
                    cameraNode.incLookY(yDelta);
                }
            }
        );
    }

    this.Init = function () {
        var self = this;

        $(document).mousemove(
            function (event) {
                if (self.isEnabled && !self.isNavigating) {
                    if (Math.abs(self.lookAround.lastMouseDownX - event.clientX) > 15 ||
Math.abs(self.lookAround.lastMouseDownY - event.clientY) > 15) {
                        self.isNavigating = true;
                    }
                }

                if (!self.isEnabled) return;

                self.glScene.GetCameraNode(
                    function (cameraNode) {
```

```
                        var movementX = 0;
                        var movementY = 0;

                        if (self.lastMouseMoveY && self.lastMouseMoveX) {
                            movementY = event.clientY - self.lastMouseMoveY;
                            movementX = event.clientX - self.lastMouseMoveX;
                        }

                        self.pitch -= movementY * self.lookRotateSensitivity;
                        self.yaw -= movementX * self.lookRotateSensitivity;

                        var eye = cameraNode.getEye();
                        var newLookDirection = getDirection(self.pitch, self.yaw);

                        cameraNode.setLookX(newLookDirection.x + eye.x);
                        cameraNode.setLookY(newLookDirection.y + eye.y);
                        cameraNode.setLookZ(newLookDirection.z + eye.z);

                        self.lastMouseMoveY = event.clientY;
                        self.lastMouseMoveX = event.clientX;
                    }
                );
            }
        );

        $(document).mousedown(
            function (event) {
                self.isNavigating = false;
                self.Reset();
                self.lookAround.lastMouseDownX = event.clientX;
                self.lookAround.lastMouseDownY = event.clientY;
                self.lastMouseMoveX = null;
                self.lastMouseMoveY = null;
                self.isEnabled = true;
            }
        );

        $(document).mouseup(
            function (event) {
                self.lookAround.lastMouseUpX = event.clientX;
                self.lookAround.lastMouseUpY = event.clientY;
                self.isEnabled = false;
                self.Reset();
            }
        );

        $(document).keydown(
            function (event) {
                if (self.isEnabled) {
                    self.moveForward = event.keyCode == 38 || event.keyCode == 87 ? true :
self.moveForward;
                    self.moveBackward = event.keyCode == 40 || event.keyCode == 83 ? true :
self.moveBackward;
                    self.moveLeft = event.keyCode == 37 || event.keyCode == 65 ? true :
self.moveLeft;
                    self.moveRight = event.keyCode == 39 || event.keyCode == 68 ? true :
self.moveRight;
                }

                if (self.moveForward || self.moveBackward || self.moveLeft || self.moveRight)
                    self.isNavigating = true;
            }
        );
```

```
        $(document).keyup(
            function (event) {
                if (!self.isEnabled) return;

                self.moveForward = event.keyCode == 38 || event.keyCode == 87 ? false :
self.moveForward;
                self.moveBackward = event.keyCode == 40 || event.keyCode == 83 ? false :
self.moveBackward;
                self.moveLeft = event.keyCode == 37 || event.keyCode == 65 ? false : self.moveLeft;
                self.moveRight = event.keyCode == 39 || event.keyCode == 68 ? false :
self.moveRight;
            }
        );

        $(document).bind('mousewheel DOMMouseScroll',
            function (e) {
                var sign = e.originalEvent.wheelDelta ?
                    e.originalEvent.wheelDelta / Math.abs(e.originalEvent.wheelDelta) :
                    e.originalEvent.detail / Math.abs(e.originalEvent.detail) * -1;
                self.moveCameraOnYBy(sign * 120 * 0.02);
            }
        );

        this.glScene.scene.on('tick', function () {
            if (self.isEnabled) {
                var time = performance.now();
                var delta = (time - self.prevTime) / 1000;
                self.velocity.x -= self.velocity.x * 10.0 * delta;
                self.velocity.z -= self.velocity.z * 10.0 * delta;
                if (self.moveForward) self.velocity.z -= 20.0 * delta;
                if (self.moveBackward) self.velocity.z += 20.0 * delta;
                if (self.moveLeft) self.velocity.x -= 20.0 * delta;
                if (self.moveRight) self.velocity.x += 20.0 * delta;

                self.prevTime = time;

                self.moveEye();
            }
        });
    }

    this.Init();
}
```