UNIVERSIDAD
REY JUAN CARLOS

Grado en ingeniería en tecnologías de telecomunicación

Curso Académico 2015/2016

Trabajo Fin de Grado

webGL Based 3D Dashboard for Tracking Software Development

Autor : Adrián Alonso Barriuso

Tutor : Dr. Jesús M. González Barahona

Proyecto Fin de Carrera

FIXME: Título

Autor : Adrián Alonso Barriuso **Tutor :** Dr. Jesús M. González Barahona

La defensa del presente Proyecto Fin de Grado se realizó el día de de 2016, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 20XX

Dedicado a

mi familia / mi abuelo / mi abuela

II

Acknowledgment

Aquí vienen los agradecimientos... Aunque está bien acordarse de la pareja, no hay que olvidarse de dar las gracias a tu madre, que aunque a veces no lo parezca disfrutará tanto de tus logros como tú... Además, la pareja quizás no sea para siempre, pero tu madre sí.

Summary

This project aims to facilitate the visualization of data and statistics from software development in 3D, inside any web browser without the need to install any drivers , plugin or additional software.

To do this,it has been mainly used the JavaScript programming language , and HTML5 as the basis to use WebGL, a library that allow us to render 3D graphics with hardware acceleration natively in almost any browser. However , since webGL is relatively low level and would be very difficult to make applications directly on its API , it has been used a library that provides a higher level of abstraction than WebGL, this library is Three.js.

Resumen

El presente proyecto tiene como objetivo facilitar la visualización de datos y estadísticas de desarrollo de software usando una interfaz y funcionalidad similares dc.js, sierviéndose de las posibilidades que ofrece la creación de gráficos en 3D. Esto será posible dentro de casi cualquier navegador web sin la necesidad de instalar ningún controlador, plugin o software adicional gracias a la tecnología webGL, una librería de código abierto que nos permite renderizar gráficos en 3D con aceleración gráfica de forma nativa.

Para realizarlo se ha utilizado principalmente el lenguaje de programación JavaScript y HTML5 como base para poder servirse de webGL, No obstante, dado que webGL es de relativo bajo nivel y sería muy complicado realizar aplicaciones directamente sobre su API, se ha utilizado una librería que proporciona un mayor nivel de abstracción que webGL, esta librería es Three.js.

Contents

1 Introduction **1**

1.1 Objectives . . . 1

1.1.1 Problem description . . . 1

1.1.2 Main Objective . . . 2

1.1.3 Requeriments . . . 2

2 Used technologies **3**

2.1 HTML5 . . . 3

2.1.1 General description . . . 3

2.1.2 In this project . . . 4

2.2 Javascript . . . 4

2.2.1 Main features . . . 5

2.2.2 In this project . . . 7

2.3 WebGL . . . 7

2.3.1 Support . . . 7

2.3.2 In this project . . . 9

2.4 Three . . . 10

2.4.1 Usage example . . . 12

2.4.2 In this project . . . 14

2.5 Domevents . . . 14

2.6 Crossfilter . . . 15

2.6.1 General description . . . 15

2.6.2 In this project . . . 15

2.7 Orbitcontrols . . . 16

2.8 dat.GUI 16

3 Development **17**

3.1 Scrum methodology 17

3.2 Iteration 0: Investigation and learning 19

3.3 Iteration 1: First demos 22

3.4 Iteration 2: Interactivity and events 26

3.5 Iteration 3: Filtering 29

3.6 Iteration 4: Framework creation 31

3.7 Iteration 5: Panel creation and additional features 33

4 Design and results **37**

4.1 Introduction 37

4.2 The mixin 38

4.3 Bars chart 41

4.4 Line chart 42

4.5 Pie chart 43

4.6 Smooth curve chart 44

4.7 The filters 45

4.8 Panels 46

4.9 Examples 50

5 Conclusions **53**

5.1 Application of lessons learned 55

5.2 Lessons learned 55

5.3 Future work 55

A Appendix **57**

A.1 HTML5 sheet 57

A.2 Three.js and Babylon.js comparison 58

A.3 Three.js's Hello world! 60

A.4 Raycasting example 65

B Bibliography **67**

List of Figures

2.1 Car rendered with Three.js 11
2.2 Basic cube made with Three.js 13

3.1 The scrum framework 19
3.2 Index of Stemkoski tutorials 20
3.3 Three.js 'Hello world' by Stemkoski 21
3.4 Different materials, same light source (the white sphere) 21
3.5 Commits(blue) and authors(red) 24
3.6 Two points of view 24
3.7 Line and pie charts built with line shapes 26
3.8 Color change thanks to ray caster 27
3.9 3D text 28
3.10 dc.js example 29
3.11 Example of filtering, note that the bars chart is ordered from top to bottom for now 30
3.12 Basic mixin structure 32
3.13 Extended mixin structure 34
3.14 Example with panels, labels and grids 35

4.1 Basic structure 38
4.2 From instantiation to rendering 41
4.3 Bars chart example 42
4.4 Line chart example 43
4.5 Pie chart example 44

4.6 Smooth curve chart example 45
4.7 Example of filtering, pie: commits per organization, bars:commits per month. After click on a particular month of the bars chart, the pie chart is rebuilt and we can see what organizations contributed in this particular month. In this case, just two organizations contributed: one with 216 and the other with 33. This proportion is reflected in the pie. 46
4.8 Empty panel 47
4.9 Panel with a chart 48
4.10 Drag's user interface 49
4.11 We can drag any chart out of the panel, if we want to get it back to its place, we can drag it manually or click on the panel and all the charts will return to their place automatically 50
4.12 If we set the opacity property in charts and panels, we will can see through them, so we can put panels and charts behind others and compare them easily . 50
4.13 Although the charts rotation is not implemented in the framework yet, I made a simple demo to show its possibilities, in the future we will be able to put panels in different angles, forming structures, like for example, a house where the walls would be panels. 51

5.1 Two pie charts with the same data. 54

Chapter 1

Introduction

In this chapter we will introduce the project objectives and its context, in order to clarify its basis before to dive into the technical details.

1.1 Objectives

1.1.1 Problem description

On the web we can find lot of frameworks whose objective is to provide abstractions that allow us to create our own dashboards in an easy way with not to many lines of code. A dashboard is an user interface that let us manage a specific software and in our case it allow us to create different kinds of charts that show software developing data and also, they can interact between them. Some popular examples of this frameworks could be Kibana, a tool to visualize and explore data indexed on ElasticSearch, Freeboard, a dashboard orientated to data visualization of internet of things or dc.js a library that allows us to visualize charts with a huge amount of data and filter them in real time. However, all these libraries have something in common, they show us 2D charts, which has its advantages. Due to the fact that they are created in 2D, there is no need of graphic acceleration nor extra plugin, so that, despite the simplicity of 2D rendering(with barely impact in hardware performance) it has been the most common method of data representation. Furthermore, 2D representation also has its disadvantages, we could never visualize charts with more than 2 dimensions for obvious reasons; and collocation and visualization possibilities are limited too.

Those disadvantages are the inspiring reason of this project, be different of the other kinds of libraries and to create one which render 3D charts, what is already possible without any plugin or additional driver thanks to webGL, that allow us to render 3D graphics in almost any web browser.

1.1.2 Main Objective

This project has as main objective to achieve a system which to allows create 3D dashboards to visualize and filter data from software development inside any web browser. However, the data can be extracted from any source, just it must be in the Crossfilter's(2.6) format to be understood by our framework.

1.1.3 Requeriments

The project has the following basic requirements:

1. Achieve an interface and functionality as similar as the Dc.js as possible.
2. Use any webGL renderer, in order to render the dashboard inside the web browser.
3. As fast filter response as we can.
4. Be able to drag, zoom, and displace around the dashboard.
5. Be able to place independent charts at any position
6. Be able to place the charts inside panels, to handle related charts together.
7. Have an object-oriented structure to facilitate future chart types thanks to the inheritance and abstraction.

Chapter 2

Used technologies

In this chapter we will take a look to the most important technologies used to make our library, with each technology, we start with a general description of the technology and then we describe how we use it in the project.

2.1 HTML5

2.1.1 General description

HTML5 is a markup language used for structuring and presenting content on the World Wide Web. It was finalized, and published, on 28 October 2014 by the World Wide Web Consortium (W3C) This is the fifth revision of the HTML standard since the inception of the World Wide Web. The previous version, HTML 4, was standardized in 1997.

Its core aims are to improve the language with support for the latest multimedia while keeping it easily readable by humans and consistently understood by computers and devices (web browsers, parsers, etc.). HTML5 is intended to subsume not only HTML 4, but also XHTML 1 and DOM Level 2 HTML.

In particular, HTML5 adds many new syntactic features. These include the new video, audio and canvas elements, as well as the integration of scalable vector graphics (SVG) content (replacing generic object tags) and MathML for mathematical formulas. These features are designed to make it easy to include and handle multimedia and graphical content on the web without having to resort to proprietary plugins and APIs. Other new page structure elements, such

as main, section, article, header, footer, aside, nav and figure, are designed to enrich the semantic content of documents. New attributes have been introduced, some elements and attributes have been removed and some elements, such as a, cite and menu¿ have been changed, redefined or standardized. The APIs and Document Object Model (DOM) are no longer afterthoughts, but are fundamental parts of the HTML5 specification.HTML5 also defines in some detail the required processing for invalid documents so that syntax errors will be treated uniformly by all conforming browsers and other user agents.

2.1.2 In this project

As we said in the previous subsection, HTML5 adds the new syntactic feature known as Canvas, allowing us to render dynamic graphics and animations on our web pages. Almost all web browsers supports Canvas today. To render 3D charts, we must to tell Three.js the canvas element that will contain our beautiful crafted charts:

```
// attach div element to variable to contain the renderer
container = document.getElementById( 'ThreeJS' );
// attach renderer to the container div
canvas=renderer.domElement
container.appendChild( canvas );
```

Where the container is the div with ThreeJS tag(for example).

2.2 Javascript

JavaScript is a high-level, dynamic, untyped, and interpreted programming language.It has been standardized in the ECMAScript language specification. Alongside HTML and CSS, it is one of the three essential technologies of World Wide Web content production; the majority of websites employ it and it is supported by all modern Web browsers without plug-ins.JavaScript is prototype-based with first-class functions, making it a multi-paradigm language, supporting object-oriented,imperative, and functional programming styles.It has an API for working with text, arrays, dates and regular expressions, but does not include any I/O, such as networking, storage, or graphics facilities, relying for these upon the host environment in which it is embedded.

Despite some naming, syntactic, and standard library similarities, JavaScript and Java are otherwise unrelated and have very different semantics. The syntax of JavaScript is actually derived from C, while the semantics and design are influenced by the Self and Scheme programming languages.

JavaScript is also used in environments that are not Web-based, such as PDF documents, site-specific browsers, and desktop widgets. Newer and faster JavaScript virtual machines (VMs) and platforms built upon them have also increased the popularity of JavaScript for server-side Web applications. On the client side, JavaScript has been traditionally implemented as an interpreted language, but more recent browsers perform just-in-time compilation. It is also used in game development, the creation of desktop and mobile applications, and server-side network programming with runtime environments such as Node.js.

2.2.1 Main features

- Imperative and structured: JavaScript supports much of the structured programming syntax from C (e.g., if statements, while loops, switch statements, do while loops, etc.). One partial exception is scoping: JavaScript originally had only function scoping with var. ECMAScript 2015 adds a let keyword for block scoping, meaning JavaScript now has both function and block scoping. Like C, JavaScript makes a distinction between expressions and statements. One syntactic difference from C is automatic semicolon insertion, which allows the semicolons that would normally terminate statements to be omitted.

- Dynamic: As with most scripting languages, JavaScript is dynamically typed; a type is associated with each value, rather than just with each expression. For example, a variable that is at one time bound to a number may later be re-bound to a string. JavaScript supports various ways to test the type of an object, including duck typing. JavaScript includes an eval function that can execute statements provided as strings at run-time.

- Prototype-based (Object-oriented): JavaScript is almost entirely object-based. In JavaScript, an object is an associative array, augmented with a prototype (see below); each string key provides the name for an object property, and there are two syntactical ways to specify such a name: dot notation (obj.x = 10) and bracket notation (obj['x'] = 10). A property may be added, rebound, or deleted at run-time. Most properties of an object (and any

property that belongs to an object's prototype inheritance chain) can be enumerated using a for...in loop.

JavaScript has a small number of built-in objects, including Function and Date.

- Functional: A function is first-class; a function is considered to be an object. As such, a function may have properties and methods, such as .call() and .bind().A nested function is a function defined within another function. It is created each time the outer function is invoked. In addition, each nested function forms a lexical closure: The lexical scope of the outer function (including any constant, local variable, or argument value) becomes part of the internal state of each inner function object, even after execution of the outer function concludes.JavaScript also supports anonymous functions.

Syntax examples:

```
var x; // defines the variable x, the special value 'undefined' (not to be
   confused with an undefined value) is assigned to it by default
var y = 2; // defines the variable y and assigns the value of 2 to it

//A simple recursive function:
function factorial(n) {
    if (n == 0) {
        return 1;
    }
    return n*factorial(n - 1);
}

//Anonymous function (or lambda) syntax and closure example:
var displayClosure = function() {
    var count = 0;
    return function () {
        return ++count;
    };
}
var inc = displayClosure();
inc(); // returns 1
inc(); // returns 2
inc(); // returns 3
```

2.2.2 In this project

Is have been used JavaScript for all the scripts in the project, and also, all the libraries we have included are written in JavaScript, these libraries are described in detail in the following sections.

2.3 WebGL

WebGL (Web Graphics Library) is a JavaScript API for rendering interactive 3D computer graphics and 2D graphics within any compatible web browser without the use of plug-ins. WebGL is integrated completely into all the web standards of the browser allowing GPU accelerated usage of physics and image processing and effects as part of the web page canvas. WebGL elements can be mixed with other HTML elements and composited with other parts of the page or page background.WebGL programs consist of control code written in JavaScript and shader code that is executed on a computer's Graphics Processing Unit (GPU). WebGL is designed and maintained by the non-profit Khronos Group.

WebGL 1.0 is based on OpenGL ES 2.0 and provides an API for 3D graphics.It uses the HTML5 canvas element and is accessed using Document Object Model interfaces. Automatic memory management is provided as part of the JavaScript language.

Like OpenGL ES 2.0, WebGL does not have the fixed-function APIs introduced in OpenGL 1.0 and deprecated in OpenGL 3.0. This functionality can instead be provided by the user in the JavaScript code space. Shaders in WebGL are expressed directly in GLSL(OpenGL Shading Language,a high-level shading language based on the syntax of the C programming language).

2.3.1 Support

WebGL is widely supported in modern browsers. However its availability is dependent on other factors like the GPU supporting it. The official WebGL website offers a simple test page[1].

More detailed information (like what renderer the browser uses, and what extensions are available) is provided at third-party websites.

Desktop Browsers:

[1] http://get.webgl.org/

- Google Chrome: WebGL has been enabled on all platforms that have a capable graphics card with updated drivers since version 9, released in February 2011.By default on Windows, Chrome uses the ANGLE (Almost Native Graphics Layer Engine) renderer to translate OpenGL ES to Direct X 9.0c or 11.0, which have better driver support.On Linux and Mac OS X the default renderer is OpenGL however.It is also possible to force OpenGL as the renderer on Windows.Since September 2013, Chrome also has a newer Direct3D 11 renderer, which however requires a newer graphics card.

- Mozilla Firefox: WebGL has been enabled on all platforms that have a capable graphics card with updated drivers since version 4.0.Since 2013 Firefox also uses DirectX on the Windows platform via ANGLE.

- Safari: Safari 6.0 and newer versions installed on OS X Mountain Lion, Mac OS X Lion and Safari 5.1 on Mac OS X Snow Leopard implemented support for WebGL, which was disabled by default before Safari 8.0.

- Opera: WebGL has been implemented in Opera 11 and 12, although was disabled by default in 2014.

- Internet Explorer:WebGL is partially supported in Internet Explorer 11.It initially failed the majority of official WebGL conformance tests, but Microsoft later released several updates. The latest 0.94 WebGL engine currently passes 97% of Khronos tests. WebGL support can also be manually added to earlier versions of Internet Explorer using third-party plugins such as IEWebGL.

- Microsoft Edge: The initial stable release supports WebGL version 0.95.

Mobile Browsers:

- BlackBerry 10: WebGL is available for BlackBerry devices since OS version 10.00

- BlackBerry 10: WebGL is available via WebWorks and browser in PlayBook OS 2.00

- Android Browser: Basically unsupported, but the Sony Ericsson Xperia range of Android smartphones have had WebGL capabilities following a firmware upgrade.Samsung smartphones also have WebGL enabled (verified on Galaxy SII (4.1.2) and Galaxy Note

8.0 (4.2)). Supported in Google Chrome that replaced Android browser in many phones (but is not a new standard Android Browser).

- Internet Explorer:WebGL is available on Windows Phone 8.1.
- Firefox for mobile WebGL is available for Android and MeeGo devices since Firefox 4.
- Firefox OS
- Google Chrome: WebGL is available for Android devices since Google Chrome 25 and enabled by default since version 30.
- Maemo: In Nokia N900, WebGL is available in the stock microB browser from the PR1.2 firmware update onwards.
- MeeGo:WebGL is unsupported in the stock browser "Web." However, it is available through Firefox.
- Opera Mobile: Opera Mobile 12 supports WebGL (on Android only).
- Sailfish OS: WebGL is supported in the default Sailfish browser.
- Tizen.
- Ubuntu Touch.
- WebOS.
- iOS:WebGL is available for mobile Safari, in iOS 8.

2.3.2 In this project

The WebGL API may be too tedious to use directly without some utility libraries due to its low abstraction level. Loading scene graphs and 3D objects in the popular industry formats is also not directly provided for. JavaScript libraries have been built (or sometimes ported to WebGL) to provide the additional functionality. A non-exhaustive list of libraries that provide many high-level features includes BabylonJS, three.js, O3D, OSG.JS, CopperLicht and GLGE. There also has been a rapid emergence of game engines for WebGL,including Unreal Engine

4 and Unity 5.The Stage3D/Flash-based Away3D high-level library also has a port to WebGL via TypeScript.A more light-weight utility library that provides just the vector and matrix math utilities for shaders is sylvester.js.It is sometimes used in conjunction with a WebGL specific extension called glUtils.js.

To avoid this low abstraction level , we use Three.js in our case, but here[2] we can find other useful webGL frameworks. To see Three.js in detail, we will go to the next section.

2.4 Three

Three.js is a cross-browser JavaScript library/API used to create and display animated 3D computer graphics in a web browser. Three.js uses WebGL. The source code is hosted at a repository on GitHub[3].

Three.js allows the creation of GPU-accelerated 3D animations using the JavaScript language as part of a website without relying on proprietary browser plugins.This is possible thanks to the advent of WebGL.

High-level libraries such as Three.js or GLGE, SceneJS, PhiloGL or a number of other libraries make it possible to author complex 3D computer animations that display in the browser without the effort required for a traditional standalone application or a plugin.

Main features:

1. Effects: Anaglyph, cross-eyed and parallax barrier.
2. Scenes: add and remove objects at run-time; fog.
3. Cameras: perspective and orthographic; controllers: trackball, FPS, path and more.
4. Animation: armatures, forward kinematics, inverse kinematics, morph and keyframe.
5. Lights: ambient, direction, point and spot lights; shadows: cast and receive.
6. Materials: Lambert, Phong, smooth shading, textures and more.
7. Shaders: access to full OpenGL Shading Language (GLSL) capabilities: lens flare, depth pass and extensive post-processing library.

[2] https://en.wikipedia.org/wiki/List_of_WebGL_frameworks/
[3] https://github.com/mrdoob/three.js/

Figure 2.1: Car rendered with Three.js

8. Objects: meshes, particles, sprites, lines, ribbons, bones and more - all with Level of detail.

9. Geometry: plane, cube, sphere, torus, 3D text and more; modifiers: lathe, extrude and tube.

10. Data loaders: binary, image, JSON and scene.

11. Utilities: full set of time and 3D math functions including frustum, matrix, quaternion, UVs and more.

12. Export and import: utilities to create Three.js-compatible JSON files from within: Blender, openCTM, FBX, Max, and OBJ.

13. Support: API documentation is under construction, public forum and wiki in full operation.

14. Examples: Over 150 files of coding examples plus fonts, models, textures, sounds and other support files

15. Debugging: Stats.js, WebGL Inspector, Three.js Inspector.

An example of how complex would be our rendered graphics with Three.js: 3.2.

2.4.1 Usage example

The Three.js library is a single JavaScript file. It can be included within a web page by linking to a local or remote copy:

```
<script src="js/three.js"></script>
```

The following code creates a scene, adds a camera and a cube to the scene, creates a WebGL renderer and adds its viewport in the document.body element. Once loaded, the cube rotates about its X- and Y-axis.

```
var camera, scene, renderer;
var geometry, material, mesh;

init();
animate();

function init() {

    camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.
        innerHeight, 1, 10000);
    camera.position.z = 1000;

    scene = new THREE.Scene();

    var light = new THREE.PointLight(0xffffff,0.8);
    light.position.set(0,0,200);
    scene.add(light);

    geometry = new THREE.BoxGeometry(200, 200, 200);
    material = new THREE.MeshLambertMaterial({
        color: 0xff0000,
        wireframe: false
    });

    mesh = new THREE.Mesh(geometry, material);
    scene.add(mesh);

    renderer = new THREE.WebGLRenderer();
```

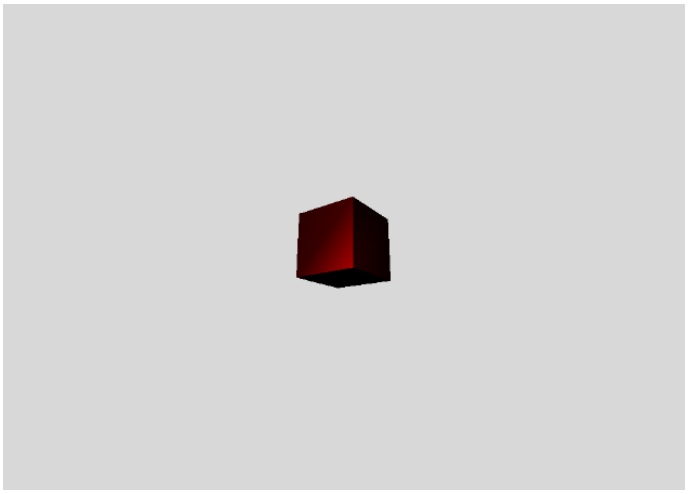


Figure 2.2: Basic cube made with Three.js

```
    renderer.setSize(window.innerWidth, window.innerHeight);
    renderer.setClearColor( 0xd8d8d8 );


    document.body.appendChild(renderer.domElement);

}

function animate() {

    requestAnimationFrame(animate);

    mesh.rotation.x += 0.01;
    mesh.rotation.y += 0.02;

    renderer.render(scene, camera);

}
```

If all went well, we will see something like this: figure 2.2.

Also, you can play with the example following this JSfiddle link [4]

[4] https://jsfiddle.net/no0aeknq/

2.4.2 In this project

We choose Three.js to write our library because is one of the most extended webGL graphics library written on JavaScript and it has a growing community of programmers. However, not all good news, due to it is a relatively new technology, its documentation is a little incomplete and outdated. Therefore, the learning mostly consists of reading code from other developers.

We can find some highly commented use examples in several sites , but they are too old in many cases, using old versions of Three.js that require some changes. One of the used tutorials is a good example of this and we can find it on GitHub [5]. It has a large number of examples of increasing complexity, but we will need to make some changes on its basis if we use the latest version of Three.js.

In addition, we have a very complete Udacity course where we can learn the foundations of interactive 3D graphics using Three.js. It contains videos, examples, exercises and even exams. You can register here [6].

2.5 Domevents

To interact with our charts, we need to manage the classic Javascript events , like 'click','mouseover', 'mouseup'...etc. With Three we must to create an invisible ray which intersects with the scene objects and then do the corresponding thing according to the event and the object. We believe this is a little unintuitive and there is a three.js extension which provide dom events inside our 3d scene allowing us to forget the ray casting. This is THREEx.DomEvent.js[7].

All we have to do is to create a domEvents object and when we need to bind an event with a mesh just apply a method to the domEvents object. Here we have a simple example:

```
//create the domEvents object with Three camera and renderer previously
    created
var domEvents    = new THREEx.DomEvents(camera, renderer.domElement);

//create a simple cube and add it to the scene
 var    geometry = new THREE.BoxGeometry(200, 200, 200);
```

[5] http://stemkoski.github.io/Three.js

[6] https://www.udacity.com/course/interactive-3d-graphics--cs291

[7] https://github.com/jeromeetienne/threex.domevents

```
var    material = new THREE.MeshBasicMaterial({color: 0xff0000});
var  cube = new THREE.Mesh(geometry, material);
scene.add(mesh);

//bind the cube with the event
domEvents.bind(cube, 'click', function(object3d){
        alert('you clicked the cube');
});
```

When we want to unbind an event, just need to call the unbind method like this:

```
domEvents.unbind(cube, 'click');
```

2.6 Crossfilter

2.6.1 General description

Crossfilter is a JavaScript library for exploring large multivariate datasets in the browser. Crossfilter supports extremely fast (30ms) interaction with coordinated views, even with datasets containing a million or more records.

Since most interactions only involve a single dimension, and then only small adjustments are made to the filter values, incremental filtering and reducing is significantly faster than starting from scratch. Crossfilter uses sorted indexes (and a few bit-twiddling hacks) to make this possible, dramatically increasing the performance of live histograms and top-K lists. For more details on how Crossfilter works, see the API reference [8].

2.6.2 In this project

As we will see in its corresponding section, we can filter the dimensions of our data sets, leading to re-render (or rebuild) the charts related to this dimension, so, in order to improve the performance, we need to handle the data sets as fast as we can, and Crossfilter allow us to do it extremely fast.

[8] https://github.com/square/crossfilter/wiki/API-Reference

2.7 Orbitcontrols

This is a Three's extension used to drag, rotate, zoom and move around the scene.

2.8 dat.GUI

A light weight graphical user interface for changing variables in JavaScript. Rather than modify variables manually in your code, use dat.GUI to modify them during execution. Here [9] we have its reference page

[9] https://workshop.chromeexperiments.com/examples/gui/#1--Basic-Usage

Chapter 3

Development

In this chapter we will describe our work methodology and the development process, in the next section we will explain our methodology and in the following sections we will explain the development.

3.1 Scrum methodology

Scrum is an iterative and incremental agile software development framework for managing product development. It defines ”a flexible, holistic product development strategy where a development team works as a unit to reach a common goal”, challenges assumptions of the ”traditional, sequential approach” to product development, and enables teams to self-organize by encouraging physical co-location or close online collaboration of all team members, as well as daily face-to-face communication among all team members and disciplines in the project.

A key principle of scrum is its recognition that during production processes, the customers can change their minds about what they want and need (often called requirements volatility), and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, scrum adopts an empirical approach,accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team’s ability to deliver quickly, to respond to emerging requirements and to adapt to evolving technologies and changes in market conditions.

In scrum there are three main roles defined:

1. **Product owner**: The person responsible for maintaining the product backlog by repre-

senting the interests of the stakeholders, and ensuring the value of the work the development team does.

2. **Scrum master**: The person responsible for the scrum process, making sure it is used correctly and maximizing its benefits.

3. **Development team**: A cross-functional group of people responsible for delivering potentially shippable increments of product at the end of every sprint.

However, in our case the product owner and the scrum master are represented by the project tutor, so the development team is the project author. Apart from that, we follow the scrum methodology faithfully.

A sprint (or iteration) is the basic unit of development in scrum. The sprint is a time-boxed effort; that is, it is restricted to a specific duration. The duration is fixed in advance for each sprint and is normally between one week and one month, with two weeks being the most common.

Each sprint starts with a sprint planning event that aims to define a sprint backlog, identify the work for the sprint, and make an estimated commitment for the sprint goal. Each sprint ends with a sprint review and sprint retrospective, that reviews progress to show to stakeholders and identify lessons and improvements for the next sprints.

Scrum emphasizes working product at the end of the sprint that is really done. In the case of software, this likely includes that the software has been integrated, fully tested and end-user documented.

Sprints or iterations:

1. **Iteration 0**: This step mostly consists of investigation and learning.

2. **Iteration 1**: First simple but functional demos with no interactivity.

3. **Iteration 2**: Add interactivity and handle events.

4. **Iteration 3**: Add Crossfilter functionality.

5. **Iteration 4**: Create the framework architecture and implement it.

6. **Iteration 5**: Add panels to contain related charts and additional features.

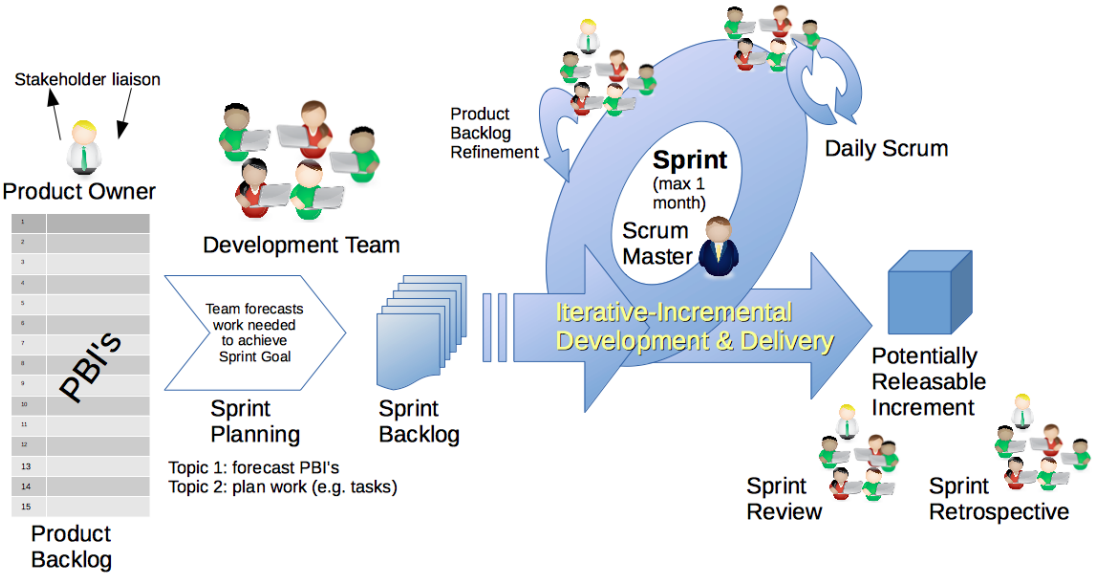


Figure 3.1: The scrum framework

3.2 Iteration 0: Investigation and learning

This is not an iteration itself, but it is a necessary part of the development, so we decided to include it as 'iteration 0' and here we will get the tools we will need to make our software.

At the beginning, there was a lot of things to explore. First of all, we had to choose a webGL framework to avoid its low abstraction level, as we said in the webGL section, there are loads of javascript libraries to do this. Probably, the two most important libraries nowadays are Three.js and Babylon.js, both of them are free software and its code is hosted at Github ,but, although Babylon.js is written by Microsoft workers, Three.js has a larger developer community with more code examples. Here A.2 we have a little code demo to see how to create a similar basic scene in Three.js and Babylon.js:

As we can see in the mentioned example, the two frameworks have strong syntax likenesses, but finally we opted for Three.js because as we said, it has a bigger developer community than Babylon.js.

Once we choose Three.js to write our framework, the next step was to study the foundations of 3D interactive graphics. If you are a passionate video games player (better known as 'gamer') like me, you will be familiar with concepts like rendering, game engine, fps(frames per second), image resolution, pixels, GPU(graphics processing unit)... etc. If you don't, the first lesson of

Figure 3.2: Index of Stemkoski tutorials

the Udacity course mentioned in the Three.js's section 2.4.2, would be helpful.

The next step before studying Three.js was to remember the Javascript language(2.2) and create a Github repository to contain the project. Once that is done, the best way to learn how to use a framework is to read code and play with it (the Udacity course is very good, but it would take much time), so i preferred to follow the Stemkoski's tutorials[1].

Let's have a look to the Stemkoski's 'Hello world' code housed in the appendices: A.3

In this heavily commented code we can see the fundamental aspects of a Three.js scene with no interactivity except for the mouse controls provided by the Three.js extension 'Orbit Controls' which allows us to rotate, zoom and displace the scene, you can see the final result here 3.3 and try the demo into your web browser here[2].

One of the most important features is to choose the mesh material, depending on the chosen material, the light affects in different ways to the meshes. There are four main mesh material types:

1. **Basic**: A material for drawing geometries in a simple shaded (flat or wireframe) way 3.4a.

2. **Lambert**:A material for non-shiny (Lambertian) surfaces, evaluated per vertex 3.4b.

[1] http://stemkoski.github.io/Three.js/
[2] http://stemkoski.github.io/Three.js/HelloWorld.html

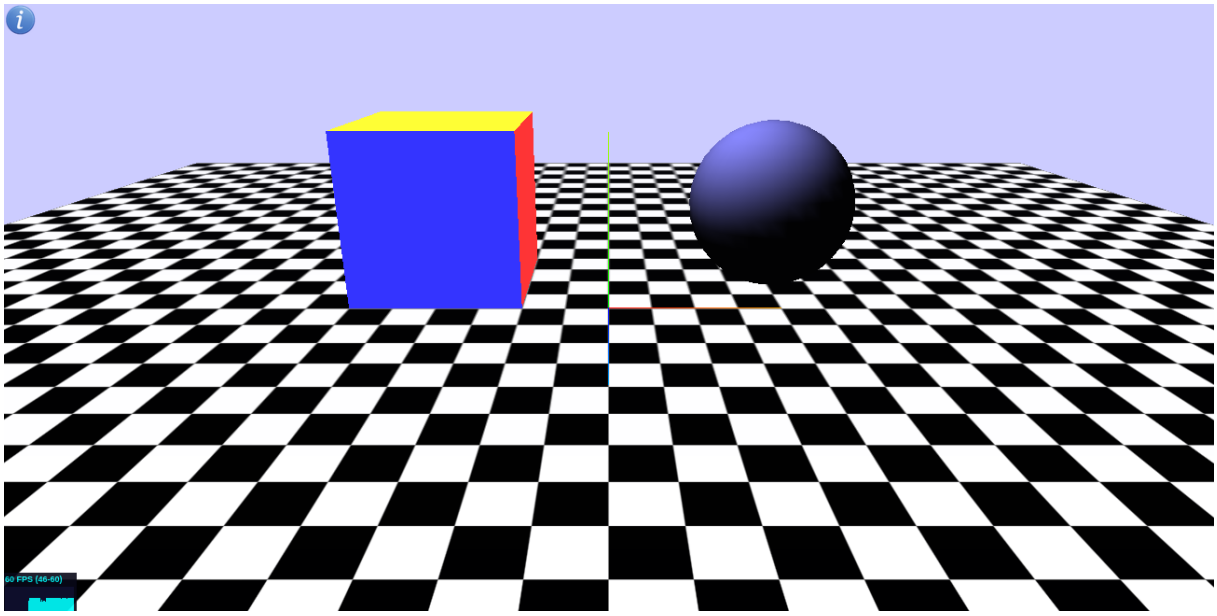
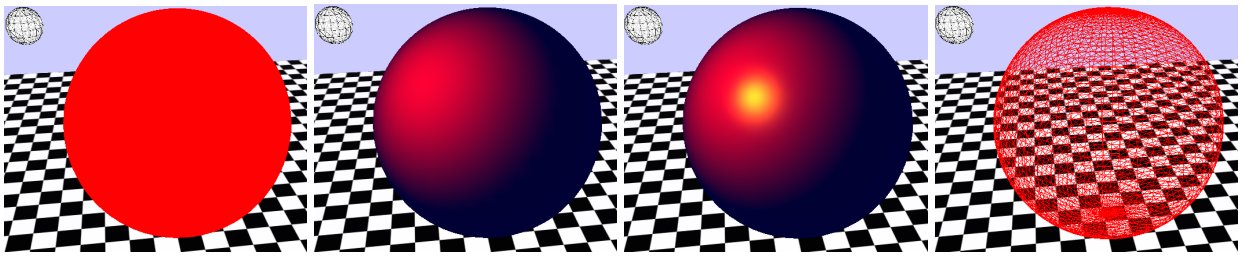


Figure 3.3: Three.js 'Hello world' by Stemkoski



(a) Basic (b) Lambert (c) Phong (d) Wireframe

Figure 3.4: Different materials, same light source (the white sphere)

3. **Phong**:A material for shiny surfaces, evaluated per pixel 3.4c.

4. **Wireframe**: The triangle's edges are displayed instead of surfaces, is not a mesh material itself (it is a property of the other materials) , but it would be useful in many cases 3.4d.

Following the rest of Stemkoski's demos, we can learn how to add shadows, create our own shapes, add textures to the meshes and much more.

Once we had reviewed Javascript and understood the basics of Three.js, we were able to make our firsts dashboard demos, so let's go to the next subsection.

3.3 Iteration 1: First demos

The first iteration objective was to explore how to render different Charts types with Three.js, the main chart types we will create are the bars chart to represent discrete values, line chart for continues values and pie charts. In order to follow an appropriate difficulty curve and for simplicity, the first chart we created was the bars chart, because the 'BoxGeometry(or CubeGeometry)' function provides us everything we needed. Here we have its constructor:

BoxGeometry(width, height, depth, widthSegments, heightSegments, depthSegments):

- width: Width of the sides on the X axis.
- height: Height of the sides on the Y axis.
- depth: Depth of the sides on the Z axis.
- widthSegments: Optional. Number of segmented faces along the width of the sides. Default is 1.
- heightSegments: Optional. Number of segmented faces along the height of the sides. Default is 1.
- depthSegments: Optional. Number of segmented faces along the depth of the sides. Default is 1.

All we have to do is to take some data set and assign each value to the height parameter of the BoxGeometry function, the width value and the depth value are not much important for now (they just have visual impact).

The data set that we used for the firsts demos had an evolutionary time line with statistics from software development, we extracted the data from a JSON and then we built two simple bars charts with two example values: 'commits' facing 'authors' along the time.

Here: A we have the entire HTML5 code to see the extensions we used in this first demo if you want to see it.

Once we created the scene in 'script.js' as we learned, we added the custom bars to it:

```
//we create the COMMITS custom chart
```

```
//coordinates for each bar
var z=1;
var y=0;
var x=1;

for (var i = 0; i < json_data.commits.length; i++) {
        //commit values are normalized for optimal visualization(/10)
        var geometry = new THREE.CubeGeometry( 1, json_data.commits[i]/10,
            10);
        // we divide by two to properly align the bars
        y=json_data.commits[i]/10/2;
        //red color
        var material = new THREE.MeshLambertMaterial( {color: "#0000ff"} );
        var commitBar = new THREE.Mesh(geometry, material);
        commitBar.position.set(x, y, z);
        scene.add(commitBar);
        //the next bar will be on the right
        x+=1;
};
```

Using the same process to build the authors chart we can see the final result: 3.5

And here[3] you can try it in your browser with some interactivity that we will see in the next section.

With this representation we can see an obvious relationship between commits and authors along the time, but thanks to the 3D renderer we can place them in different ways, for example we can form a 90 degrees angle easily just with changing the height dimension by depth dimension: 3.6

In addition, we also explored the possibility of joining both charts by rectangles to see if this could give us some information, however, we decided not to include this representation for the moment. You can see the result here [4]

Now we will focus on the continuous line chart representation, this is a rather more complicated representation than the previous one because we have to draw the line manually. In this case, there is no defined mesh geometry in Three.js that could be useful, so we had to do it with

[3] http://adrianalonsoba.github.io/threeboard/demo2/

[4] http://adrianalonsoba.github.io/threeboard/demo5/

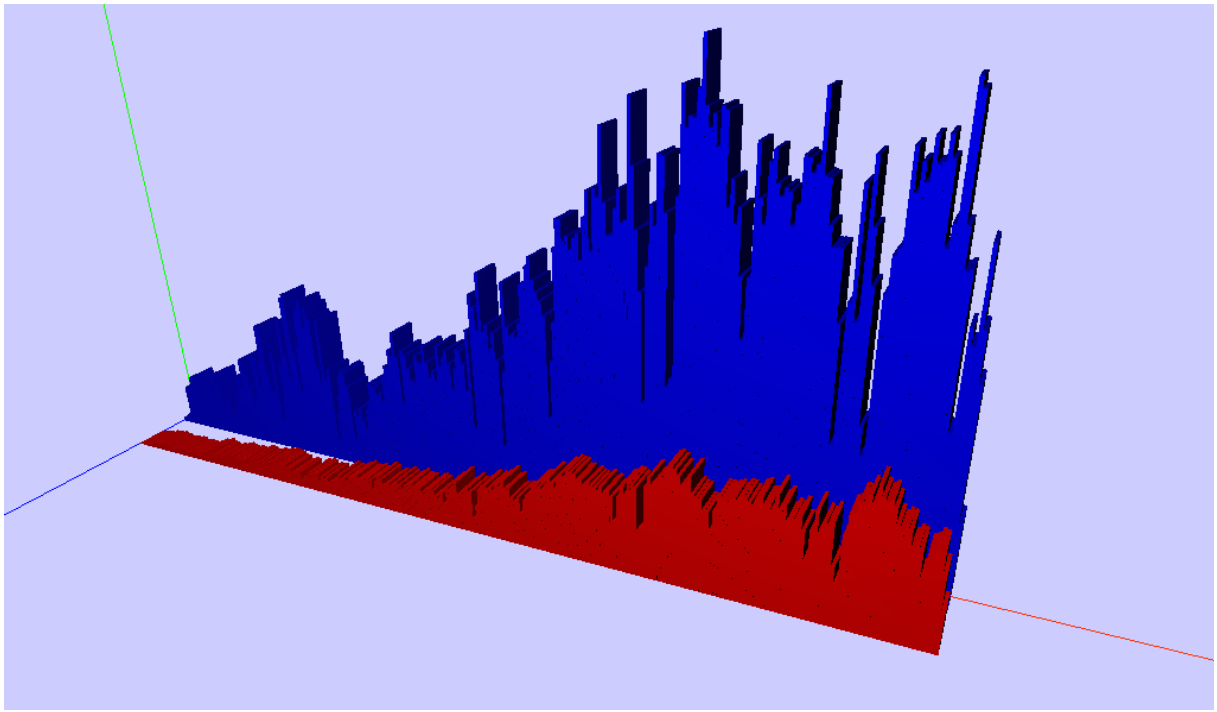
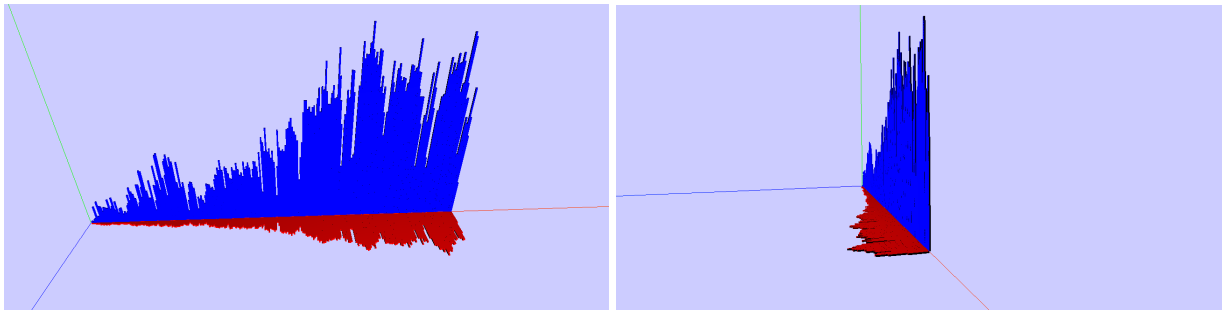


Figure 3.5: Commits(blue) and authors(red)



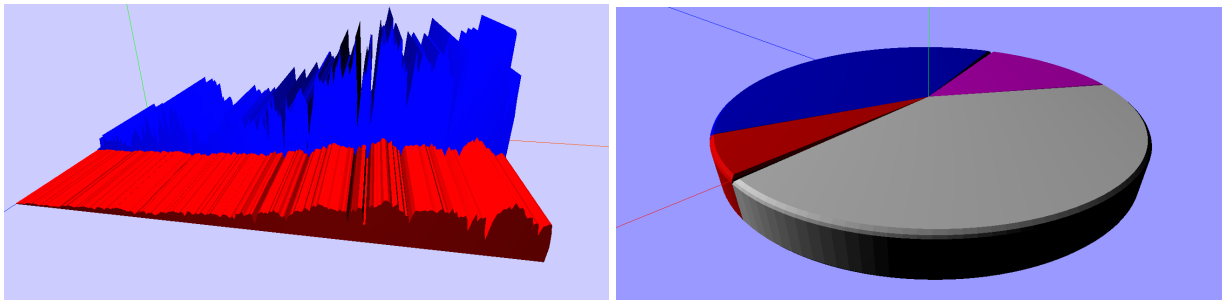
(a) (b)

Figure 3.6: Two points of view

the object 'THREE.Shape'. With that, we can draw any shape by connecting points with different methods, .Once this is donde, we must give depth to the shape with 'THREE.ExtrudeGeometry', a geometry type for the handmade shapes. Here is a code example of how to make a simple line shape made with straight lines:

```
var charShape = new THREE.Shape();
charShape.moveTo( 0,0 );

var x=0;

for (var i = 0; i < json_data.commits.length; i++) {
  charShape.lineTo( x, json_data.commits[i]/10 );
  x+=1.5;
};
//close the shape
charShape.lineTo( x, 0 );
charShape.lineTo( 0, 0 );

var extrusionSettings = {
  size: 30, height: 4, curveSegments: 3,
  bevelThickness: 1, bevelSize: 2, bevelEnabled: false,
  material: 0, extrudeMaterial: 1};

  var charGeometry = new THREE.ExtrudeGeometry( charShape,
      extrusionSettings );

  var materialSide = new THREE.MeshLambertMaterial( { color: 0x0000ff } );

  var extrudeChart = new THREE.Mesh( charGeometry, materialSide );
  extrudeChart.position.set(0,0,0);
  scene.add(extrudeChart);
```

To finish this sprint we could use the THREE.Shape and THREE.ExtrudeGeometry to make the pie charts too, but using other method to join the points. In the previous case we don't need to add curves, so we only had to use 'lineTo', but now we need to add curves for each pie part and this is done with the THREE.Shape method 'arc'. Let's have a look to the result: 3.7



(a) (b)

Figure 3.7: Line and pie charts built with line shapes

3.4 Iteration 2: Interactivity and events

Now that we have the basic charts, the next sprint requirement was to add interactivity to them:

- Show information about the charts by mouse hovering.
- Be able to fix some chart parts by clicking and test some animation.
- Drag the charts.

There is a specific method to interact with the scene using the mouse: The 'Ray-casting', this method is to create an imaginary ray from the mouse to the scene in the camera direction and it returns an array with all the objects which intersects with the ray. Normally we only take the first intersected object (the first in the array) to do what we need. However, before creating the ray caster object, we need to add some event listeners as we will see.

The first interactivity that we added was to change the color to the object in order to clarify which object is selected by mouse hovering, to do this we need two things, add a 'mouse move' event listener to refresh the mouse coordinates each animation frame (needed by the ray caster) and when the ray returns the array of objects, just change the mesh's color of the first element, you can see an example with code in the appendix A.4

Once the ray-casting method was understood, we realize that it was unintuitive and too much wired. When the ray returns the array of objects, we eventually need to add an 'if' or 'switch' sentence for each mesh type. Having said this, we thought that to use a DOM events similar method would be interesting, being capable of bind any event to any mesh at any moment.

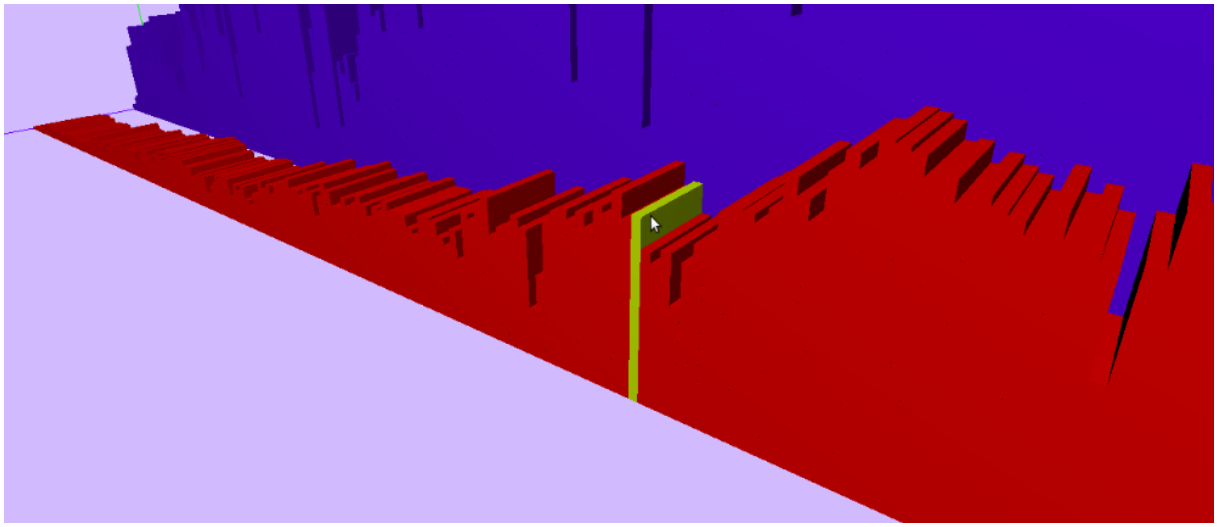


Figure 3.8: Color change thanks to ray caster

Doing a little research over the network we found all we needed: 'threex.domevents'2.5 a little Three.js extension which allowed us to bind any event to any mesh and forget the ray caster.

Adding events is much easier than before, to change the bars color just need two 'binds', one for the mouse over and one for the mouse out to restore its original color. From now on we will use this extension to add all events.

Now that we have all the tools to handle events let's go to add a 3D text label to each mesh showing the number of commits or authors. To do this, we can use 'THREE.TextGeometry', a function which makes simple to add 3D text to the scenes just by passing the text, the font family, the size and other optional parameters.

The second iteration objectives are not included in the final version but they gave us an idea of how fast the renderer can add and remove simple meshes from the scene. In this[5]demo you can fix the bars by clicking and in this[6] you can see an animation made with time intervals.

The third requirement is probably the most important of the iteration. We have to be able to drag the entire charts to put them at any place of the scene. This is not supported by threex.domevents, so that is the only event we must handle with a ray caster. The process starts like before, adding an event listener to update the mouse coordinates and instantiating the ray caster, but , in addition, now we need to create a plane which will intersect with our meshes or charts.

[5] http://adrianalonsoba.github.io/threeboard/demo6/
[6] http://adrianalonsoba.github.io/threeboard/demo8/

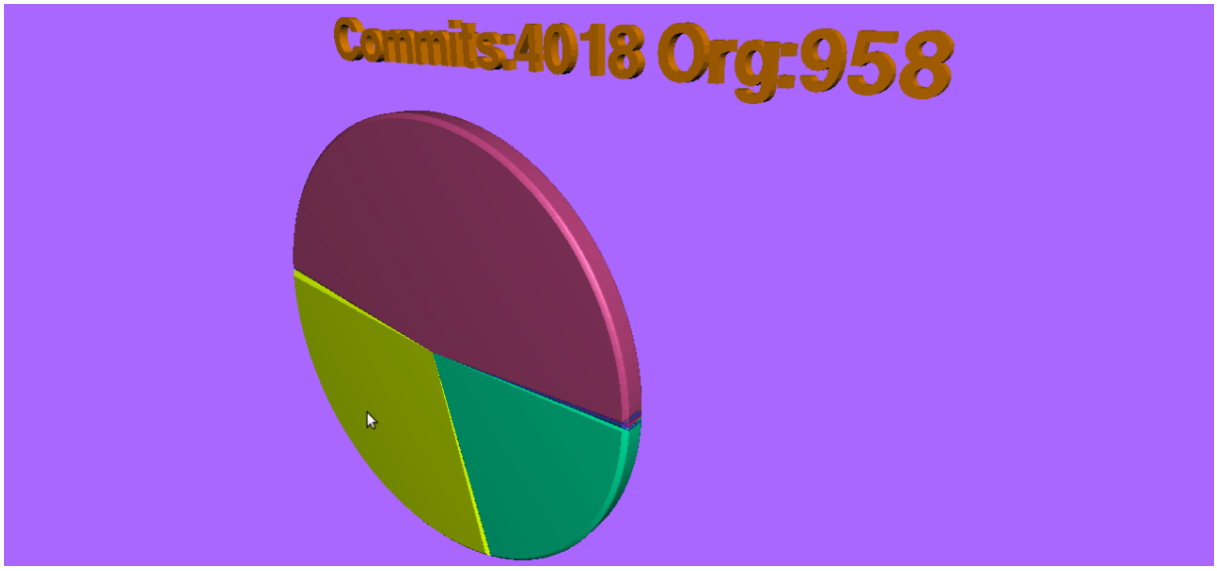


Figure 3.9: 3D text

```
function onMouseMove( event ) {
  mouse.x = ( event.clientX / window.innerWidth ) * 2 - 1;
  mouse.y = - ( event.clientY / window.innerHeight ) * 2 + 1;
  raycaster.setFromCamera( mouse, camera );
  //we select a mesh with a click event using threex.domevents
  if (SELECTED){
    var intersects = raycaster.intersectObject( plane );
    if ( intersects.length > 0 ) {
      //update the mesh position
      SELECTED.position.copy( intersects[ 0 ].point.sub( offset ) );
    }
    return;
  }
}
```

By doing that, we can move any mesh around the plane. Rotating the plane we can move the meshes in any direction, you can check it here and you can rotate the plane by changing the controls [7].

[7] http://adrianalonsoba.github.io/threeboard/demo15/

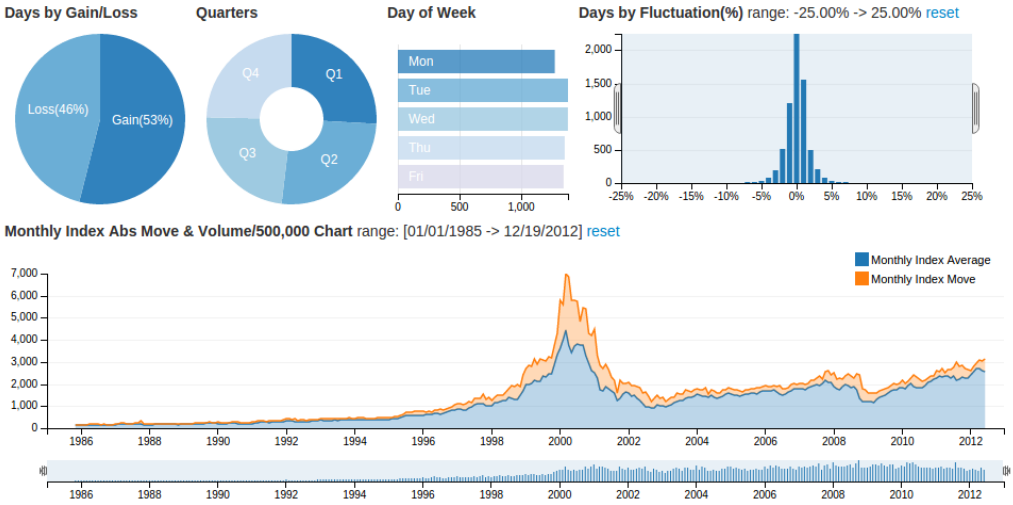


Figure 3.10: dc.js example

3.5 Iteration 3: Filtering

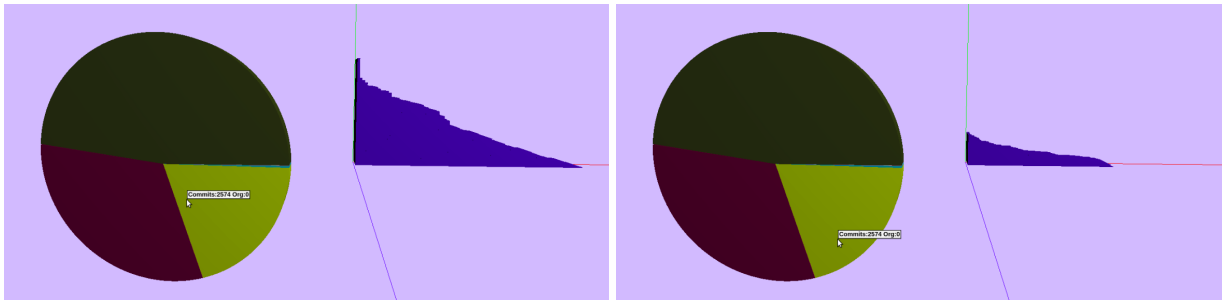
In this sprint, we must integrate crossfilter(2.6) in order to achieve a functionality that allow us to filter data sets in real time by interacting with the charts. There is already a library that do this: dc.js, a charting library with native crossfilter support and allowing highly efficient exploration on large multi-dimensional dataset (inspired by crossfilter's demo). It leverages d3.js engine to render charts in css friendly svg format.

The main objective is to provide similar functionality to that of dc.js in 3D, so first we need to understand how dc.js and crossfilter.js works.

The basic idea is to filter the data set by clicking the charts, if we click on the line chart for example, we can filter the entire data set using the selected value and we see how the charts change in real time. Also, we can add interval filters by selecting the interval with the mouse. Crossfilter.js use the selected values to filter its data set automatically, so when a filter is added, we just need to call the function responsible for drawing the charts.

To create a demo with that, first we need a data set in the format that Crossfilter understands, that is an array of objects with the same number of key-value pairs and the same keys, here we have an example where each object represents a commit:

```
var commits=[
{author: 8, date: "Mon Oct 20 2014 14:42:24 GMT+0200 (CEST)", hour: 12, id:
    11340, month: "Wed Oct 01 2014 00:00:00 GMT+0200 (CEST)",org: 958,repo:
    12,tz: 2},
```



(a) before click (b) after click

Figure 3.11: Example of filtering, note that the bars chart is ordered from top to bottom for now

```
...]
```

The data set contains 12677 objects like this one with different values but the same keys. Now we must create a single crossfilter index, a crossfilter dimension for each chart and a group for each dimension:

```
//crossfilter index
cf=crossfilter(commits);
//create a dimension and a group by month
dimByMonth= cf.dimension(function(p) {return p.month;});
groupByMonth= dimByMonth.group();
//create a dimension and a group by organization
dimByOrg= cf.dimension(function(p) {return p.org;});
groupByOrg= dimByOrg.group();
```

Now we can use the groups to build our charts, one chart will contain the number of commits per month(a bars chart for example) and the other, the number of commits per organization(pie chart). Therefore, if we click on a particular month, crossfilter applies this filter to its index, then we call the function responsible for drawing with **no** changes needed, and we will obtain a pie chart that shows the number of commits per organization in this particular month. What is more, if we click on a particular organization we will see a bars chart with the number of commits per month of this organization.

In this 3.11 figure we can see that not only the bars are lower than before, the chart is shorter too, this means there are months with no commits in the clicked organization.

You can try this simple demo in here [8]

3.6 Iteration 4: Framework creation

Now we have all the needed elements to create our library, so the main requirement for this sprint is to build the main architecture and implement it, after this sprint, we should be able to create the previous demo iteration with a few lines of code.

The first thing we needed is a global object that will contain the data structures, methods and it gives name to the framework, this object is **THREEDC**(three dimension 'dynamic' charting). THREEDC shall have three main methods and one data structure at least:

- **THREEDC.allCharts:[]**: This array will contain each chart instantiation.
- **THREEDC.renderAll()** : This method will use THREEDC.allCharts to call the particular render method of each chart type.
- **THREEDC.removeAll()** : The same as the previous , but removing all charts.
- **THREEDC.baseMixin()** : A mixin object that will be explained next.

I have used two remarkable techniques which are widely used in Javascript, a **mixin** to provide us inheritance between the different chart types avoiding the need of repeating common code(and for abstraction) and the method **chaining**, a technique that can be used to simplify code in scenarios that involve calling multiple functions on the same object consecutively.

The mixin will act like a father class, containing the common features in all charts: Render and build methods, events, and the parameters of the chart:The crossfilter dimensions and groups, chart size and chart color. The parameter methods will return the own mixin in order to make possible the method chaining.

Once we have the mixin structure(3.12), we'll implement the different chart types. All chart types will instantiate a mixin, and then they will add its own building method and parameters. The basic component of a chart will be a '**part**,' a single mesh that will contain data of the crossfilter's groups, that is, keys and values, so the mixin must have another data structure to save this parts. Every chart must build each part separately because the interaction events have

[8] http://adrianalonsoba.github.io/threeboard/demo13/

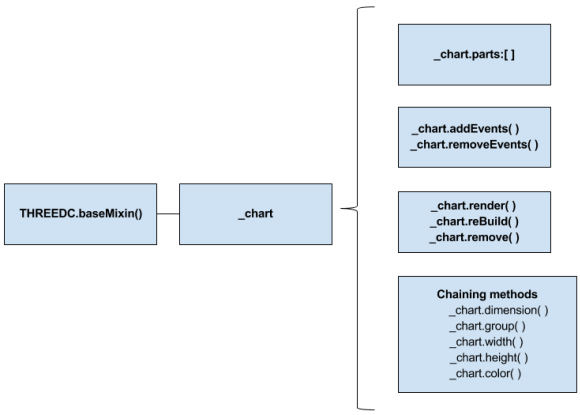


Figure 3.12: Basic mixin structure

to be bound to each part. After call a specific chart type function, it is added to the global data structure (THREEDC.allCharts),then it calls its build function, which creates every chart part, add the basic events using the mixin methods and finally it will be rendered if we call the specific render method of this chart or the global render method (THREEDC.renderAll()).

Here we have a code example which creates the previous demo iteration using the framework:

```
//crossfilter index, dimensions and groups previously created
...
//we pass coordinates to place the chart for now
var bars = THREEDC.barsChart([0,0,0]);
    bars.group(groupOne)
        .dimension(dimemsionOne)
        .width(200)
        .height(200)
        .color(0xff8000);

var pie = THREEDC.pieChart([300,100,0]);
    pie.group(groupTwo)
      .dimension(dimTwo)
      .radius(100)
      .color(0xff0000);
```

```
THREEDC.renderAll();
```

Now we can render a lot of charts with a few lines of code, and also we can set size and color settings as we prefer.

3.7 Iteration 5: Panel creation and additional features

Once we built the basic architecture, in the final iteration we will add some additional features:

- Add a new type of chart, the smooth curve chart.
- Add customizable grids and labels.
- Add panels to contain related charts

We have three basic chart types as we have seen before, all of them rendered in 3D, but it would be interesting have some type of chart rendered in 2D, this is supported by Three.js in many ways, one of them is to build a CatmullRom spline, it is a type of interpolating spline (a curve that goes through its control points) defined by four control points. This results on a smooth curve along the given points. However, this chart does not have interactivity because the construction process does not allow to create single parts, so we can just drag the chart and see how it changes when other filters are applied.

Now that we have various types of charts, we will create a process to add grids and labels to the charts automatically. The grids and the labels are common in all chart types, except for the pie chart, so we need to add more methods to the mixin: 3.13

The creation process of the grids is similar to the chart creation process, using THREE.LineBasicMaterial and THREE.Line, we create single lines which are saved in a new data structure (like the chart parts) and then are placed properly if we call the method gridsOn() at the creation time. The same with the labels, we need another data structure to save them, then they are placed using the coordinates used to place the grids. But with the labels, we have an additional problem, we don't know the label's size "a priori", leading to an overlap possibility. At this point, we had two options,first option,to use some type of size normalization or ,second option , to calculate

Figure 3.13: Extended mixin structure

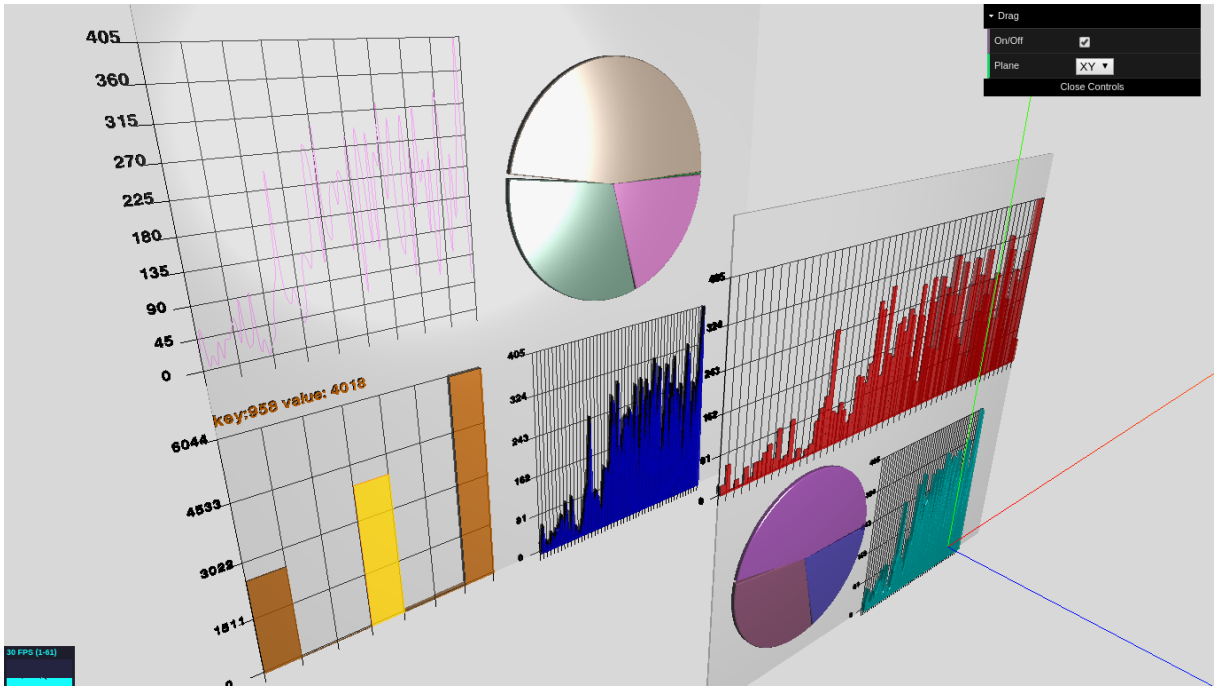


Figure 3.14: Example with panels, labels and grids

each label size and place them properly according to this size. We opted for the second option because the first option would lead to different size labels or cut labels, maybe the choice possibility will be implemented in the future.

Finally we will focus on one of the most important features of the framework, the **panel creation**. Up to now, the charts were dragged individually and we used only coordinates to place them, but we thought that having panels to place the charts as if they were a HTML div allowing us to drag them together(for example), would be useful. To make this, first we need a new THREEDC object, to contain the panels, the **THREEDC.addPanel**. This method will have a ”panel”, a single mesh which will contain the charts we want add to it. The panel need its own render and remove methods besides an array to contain its charts, when a chart is added to the panel, it is saved to the array and placed in the plane using the method ”panel.makeAnchorPoints” which depending on the number of charts we want to add to the panel it creates anchor points to put them, so when a new chart is added, it is automatically placed at the next available anchor point.

Chapter 4

Design and results

4.1 Introduction

In this chapter we will describe the library's final architecture in detail, showing how it works with the basic functions and data structures.

The code follows an object-oriented programming approximation, as we can see in 4.1, we have a global object that contain the data structures and methods, **THREEDC**

which have the next properties:

- **allCharts**: Array with all instantiated charts.
- **allPanels**: Array with all instantiated panels.
- **textLabel**: The single 3D text showed with data of a chart part.
- **chartToDrag**: Current chart to be dragged.
- **intervalFilter**: Filter value or values to be applied

and methods:

- **addPanel(coords,numberOfCharts)**: This method adds a panel which will contain related charts, allowing us to handle them together, it receives coordinates to place the panel, the number of the charts that will contain, and the size.
- **renderAll()**: This method call the render method of each chart contained into "allCharts".

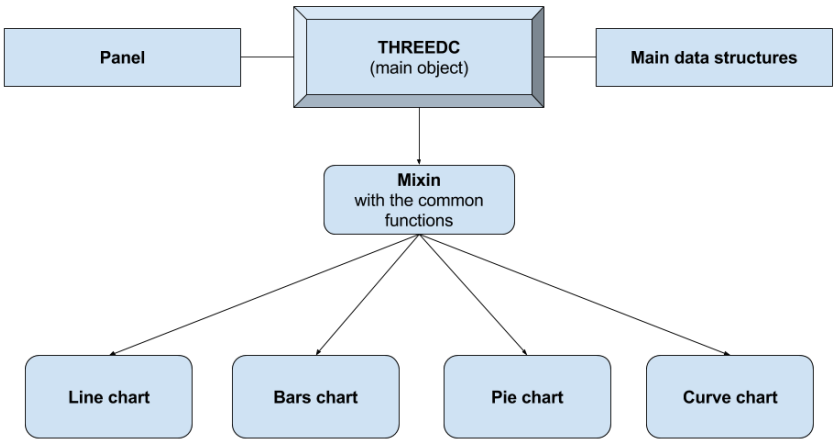


Figure 4.1: Basic structure

- **removeAll()**: It removes all charts from the scene.
- **removeEvents()**: This one removes the events of all charts.
- **baseMixin(_chart)**: This is a mixin with the common properties and methods of all charts, and it will be explained in the next subsection 4.2.

4.2 The mixin

This method must be called by all the chart types to get the common features. It is like a father class, each chart type will inherit its properties and functions and it will add its own ones if it needs them. Each chart must create a variable with the mixin, passing an empty object like this:

```
var _chart = THREEDC.baseMixin({});
```

Then, each chart type will add its necessary properties and methods to the instantiated mixin.

Here we have its properties:

- **parts**: Array with the meshes that making up the chart.
- **xLabels**: Array with the 3D labels in the x axis.

- **yLabels**: Array with the 3D labels in the y axis.
- **xGrids**: Array with the 3D grids in the x axis.
- **yGrids**: Array with the 3D grids in the y axis.
- **_gridsOn**: Boolean that enables the grids, false by default.
- **_numberOfXLabels**: Number of labels rendered in the x axis, 9 by default (10 are showed).
- **_numberOfYLabels**: Number of labels rendered in the y axis, 9 by default (10 are showed).
- **_width**: The chart width in the Three.js's relative units, 100 by default.
- **_height**: The chart height in the Three.js's relative units, 100 by default.

Methods:

- **render**: It calls the specific build method of the chart and adds it to the scene.
- **remove**: This method removes the chart and all its elements before removing it from THREEDC.allCharts.
- **reBuild**: This method rebuilds the chart when a filter is added.
- **addEvents**: This method adds the basic events to every part of the chart.
- **removeEvents**: Remove the events from all parts of the chart.
- **addGrids**: It builds the grids of the chart in function of the number of labels.
- **renderGrids**: Renders the grids.
- **removeGrids**: Removes the grids
- **addLabels**: Builds the labels of the chart, the number of labels is passed by a chaining method.
- **renderLabels**: Renders the labels.

- **removeLabels**: It removes the label.

Chaining methods, this methods are called by the user:

- **group**:Sets the Crossfilters's group of the chart, necessary to build the charts, that is mandatory.
- **dimension**:Sets the Crossfilters's dimension, necessary to build the charts,that is mandatory.
- **width**: Sets the chart's width, 100 by default as we have seen in its corresponding property.
- **height**: Sets the chart's height, 100 by default as we have seen in its corresponding property too.
- **gridsOn**: It enables the grids of the chart, it is disabled by default.
- **gridsOff**: Disables the grids.
- **numberOfXLabels**: Sets the number of labels that we want to be rendered on the x axis,9 by default.
- **numberOfYLabels**: The same with the y axis.
- **color**: Establishes the main color of the chart,(if it isn't a pie) it can be a string with the main colors ("red","blue"...f.e) but it is recommended use the hexadecimal HTML color codes to achieve a properly events functionality.
- **depth**: Sets the depth of the charts, in Three.js's relative units.
- **opacity**: The chart's opacity, 0.8 by default.

In the next sections we will explain how the chart implementations use the mixin and how to use them.

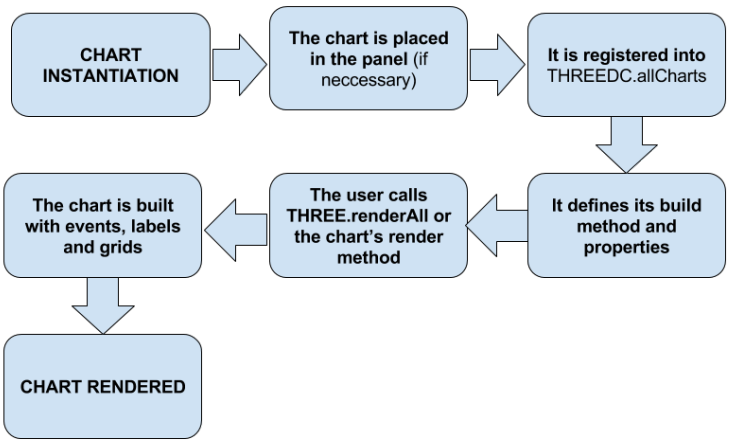


Figure 4.2: From instantiation to rendering

4.3 Bars chart

This is the first chart we implemented and we will use it to explain how each chart implementation use the mixin and its own properties and methods. The most important is the **build** method, it is called by THREEDC.renderAll() method, but every chart type has its own implementation. When a chart is instantiated by the user, first, it creates the mixin, then if it belongs to any panel, it is placed in the panel,then it is registered to THREEDC.allCharts and finally it defines its build method. The build method uses THREE.js, Crossfliter's groups and dimensions and the user parameters passed by the chaining methods to create the chart. Once the chart is built, the events and the labels are added by calling the addEvents and addGrids methods, respectively, the grids are added if we call the gridsOn method at the creation time.

Once the process is understood, we will focus on this particular chart type, the bars chart. It is suitable for discrete values, here we have a use example where we render a bars chart with the number of commits per month from some software development project.

```
//create the crossfilter's index
var cf=crossfilter(data);

//create a dimension and a group per month from the data set
```

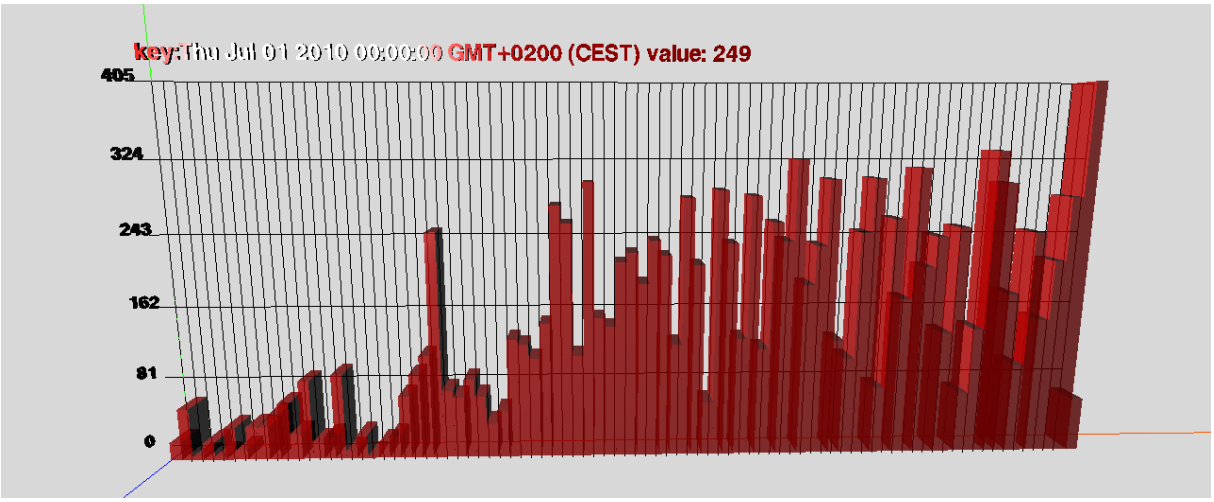


Figure 4.3: Bars chart example

```
dimPerMonth= cf.dimension(function(p) {return p.month;});
groupPerMonth= dimPerMonth.group();

//create the bars chart
var bars = THREEDC.barsChart([0,0,0]);
bars.group(groupByMonth)
    .dimension(dimByMonth)
    .width(500)
    .numberOfXLabels(83)
    .numberOfYLabels(5)
    .gridsOn()
    .height(200)
    .color(0xff0000)
    .depth(20);

THREEDC.renderAll();
```

and here4.3 he have the result

4.4 Line chart

The line chart offers a similar visualization to the bars chart, but with a continuous line, here we have a code example with the same data set as the previous example:

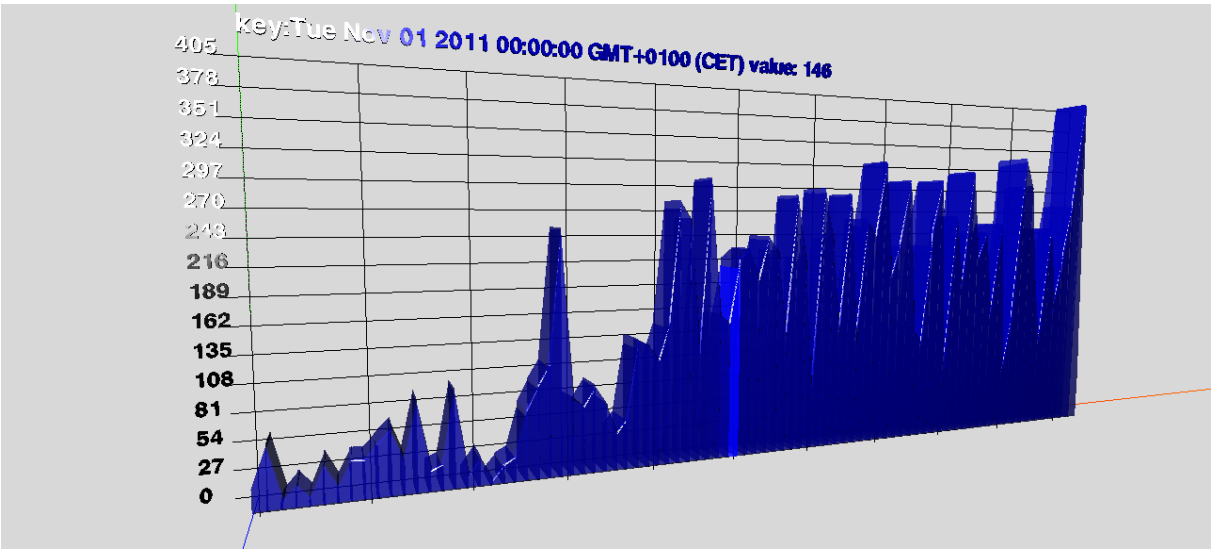


Figure 4.4: Line chart example

```
var line = THREEDC.lineChart([0,0,0]);
line.group(groupByMonth)
    .dimension(dimByMonth)
    .width(500)
    .numberOfXLabels(10)
    .numberOfYLabels(15)
    .gridsOn()
    .height(200)
    .color(0x0000ff)
    .depth(20);

THREEDC.renderAll();
```

4.5 Pie chart

The pie chart provides other type of visualization, it is suitable for reduce data groups(a pie chart with too much groups will result in less visual information) and it is divided into slices to illustrate numerical proportion. It adds an extra property which is not common with the other chart types for obvious reasons, the **radius**.

```
var pie = THREEDC.pieChart([0,0,0]);
pie.group(groupByOrg)
```

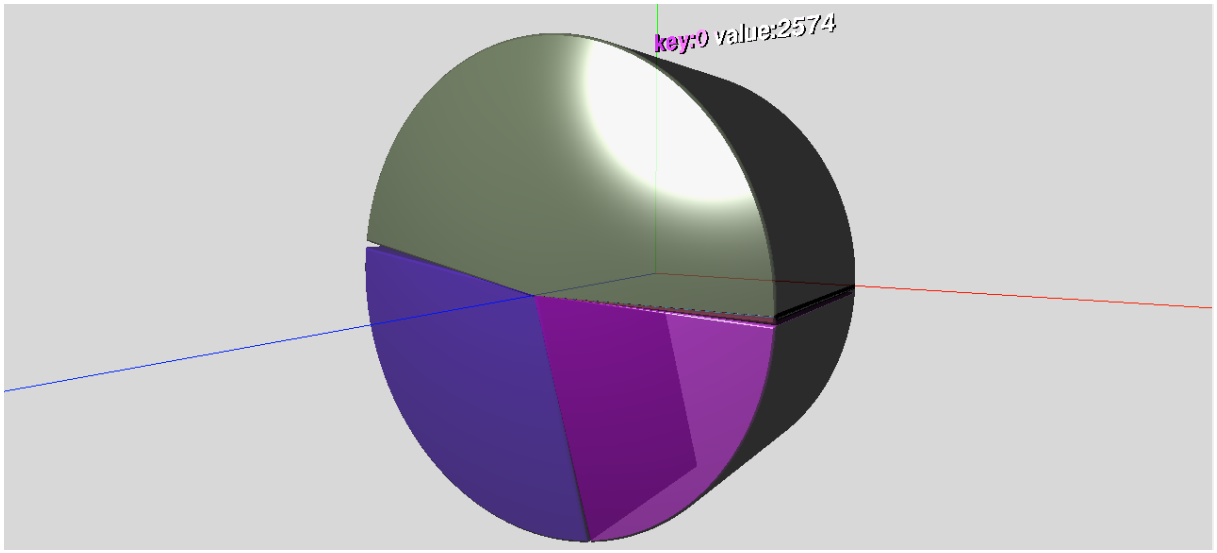


Figure 4.5: Pie chart example

```
    .dimension(dimByOrg)
    .radius(100)
    .depth(100);

THREEDC.renderAll();
```

4.6 Smooth curve chart

This is the simplest chart type, the smooth curve chart uses a CatmullRom spline to draw a curve with a single mesh(it hasn't parts),also, it has no depth and no interactivity (you can't click on it but it can be dragged), however, it changes with the filters like the other chart types. Here we have an example:

```
var curve = THREEDC.smoothCurveChart([0,0,0]);
curve.group(groupByMonth)
    .dimension(dimByMonth)
        .numberOfYLabels(5)
    .numberOfXLabels(0)
    .gridsOn()
    .width(500)
    .height(200)
    .color(0xff0000);
```

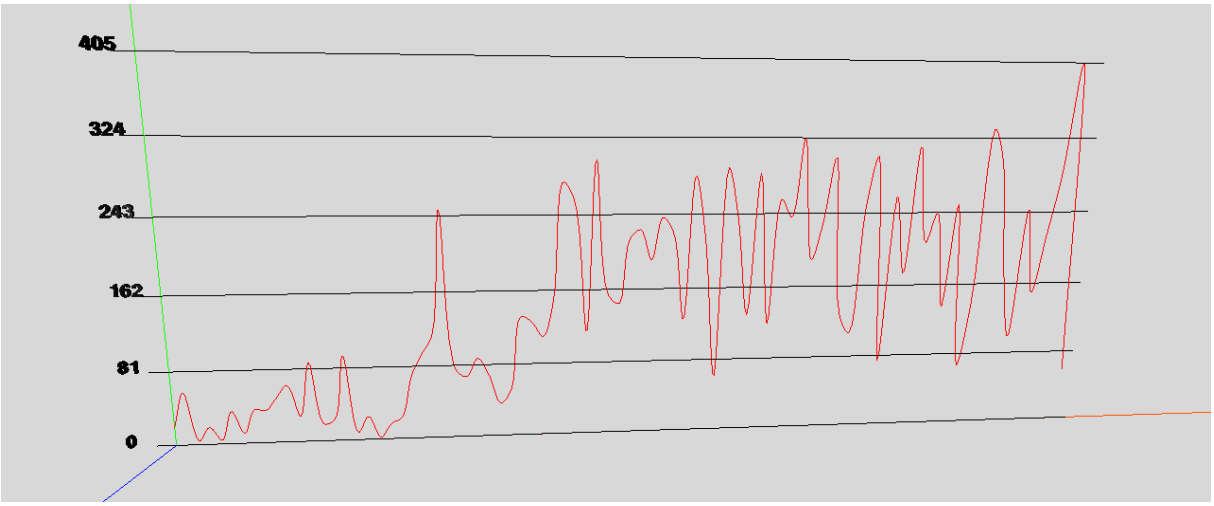


Figure 4.6: Smooth curve chart example

```
THREEDC.renderAll();
```

4.7 The filters

We can filter our data sets by clicking on the charts. We have two filter types:

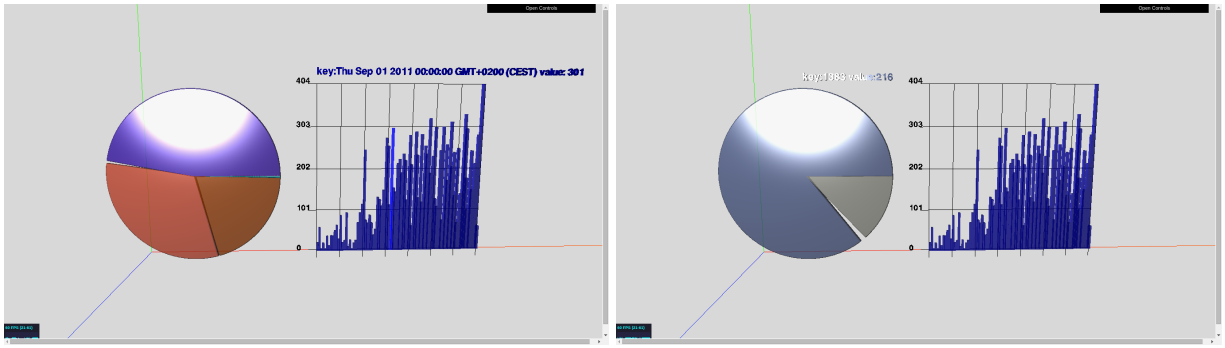
- **Single filter**: To apply it, we must click on some chart's part, and its dimension's value will be added to the Crossfilter's filter, when the filter is applied, the charts are rebuilt with its values filtered.
- **Interval filter**: It works like the single filter, but now we filter a dimension interval by making mouse down in the first interval value and mouse up in the second interval value.

.

More details in 3.5.

Now we are going to create the same example that we have seen in 3.5, but now using the framework:

```
var bars = THREEDC.barsChart([200,0,0]);
bars.group(groupPerMonth)
    .dimension(dimPerMonth)
    .width(200)
    .height(200)
```



(a) before click (b) after click

Figure 4.7: Example of filtering, pie: commits per organization, bars:commits per month. After click on a particular month of the bars chart, the pie chart is rebuilt and we can see what organizations contributed in this particular month. In this case, just two organizations contributed: one with 216 and the other with 33. This proportion is reflected in the pie.

```
        .numberOfXLabels(7)
        .gridsOn()
        .numberOfYLabels(4)
        .color(0x00000ff);

var pie = THREEDC.pieChart([0,0,0]);
pie.group(groupPerOrg)
        .dimension(dimPerOrg)
        .radius(100);

THREEDC.renderAll();
```

4.8 Panels

The panels are simple plane meshes with anchor points where we can place the related charts to handle them together. This is its interface: **THREEDC.addPanel(coords,numberOfCharts,size)**

Where the parameters are:

- **coords**: The panel's geometric center where it will be placed, must be a vector([x,y,z]).
- **numberOfCharts**: The number of charts that the panel will contain.

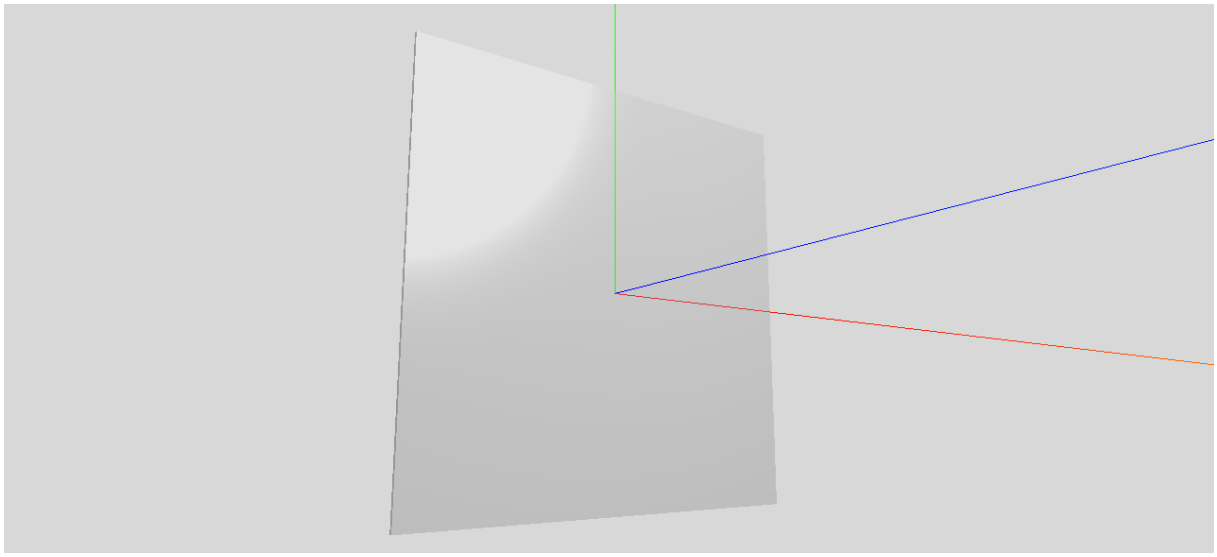


Figure 4.8: Empty panel

- **size**: The size of the panel in the Three.js's relative units, must be a vector ([width, height]).
- **opacity**: The panel's opacity, 0.8 by default.

.

Its methods are:

- **makeAnchorPoints**: This method calculates points to place the charts in function of the number of charts that it will contain and de the size of the panel.
- **reBuild**: This method updates the anchor points when the panel changes its place.
- **remove**: It removes the panel and all of its charts.

.

Here we have an example of an empty panel's instantiation:

```
var panel=THREEDC.addPanel([0,0,0],4,[500,500]);
```

Note that the panels are independent to the charts and there is no need to call any render method to see them alone.

Until now, the charts are placed just using coordinates, but now we can place them into the panels, to do this, we can pass the panel which we want to contain the charts instead of the coordinates. After instantiate the previous panel, let's to add a chart to it:

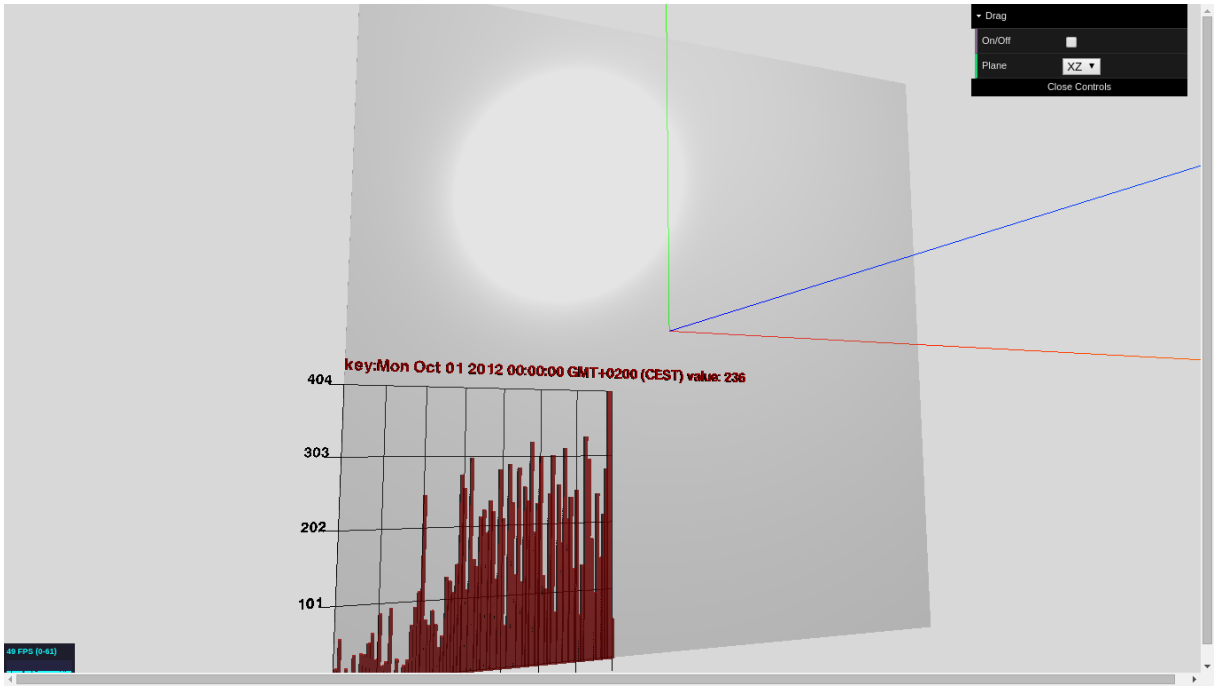


Figure 4.9: Panel with a chart

```
var bars = THREEDC.barsChart(panel);
bars.group(groupByMonth)
    .dimension(dimByMonth)
    .width(200)
    .height(200)
    .numberOfXLabels(7)
    .gridsOn()
    .numberOfYLabels(4)
    .color(0xff00000);

THREEDC.renderAll();
```

As we can see in 4.9, the chart has been placed in the first free anchor point, and if you have a look to the upper right corner of the capture, you will see this 4.13, a simple graphical user's interface where we can set the drag settings. The drag have to be enabled because if we want to drag a single chart, we need to click on it, but maybe we don't want to apply a filter, so, by activating the drag we disable the filter events to make sure we drag the charts easily without applying any filter. Then, when we finish the drag, we must disable it to get back the filter events.

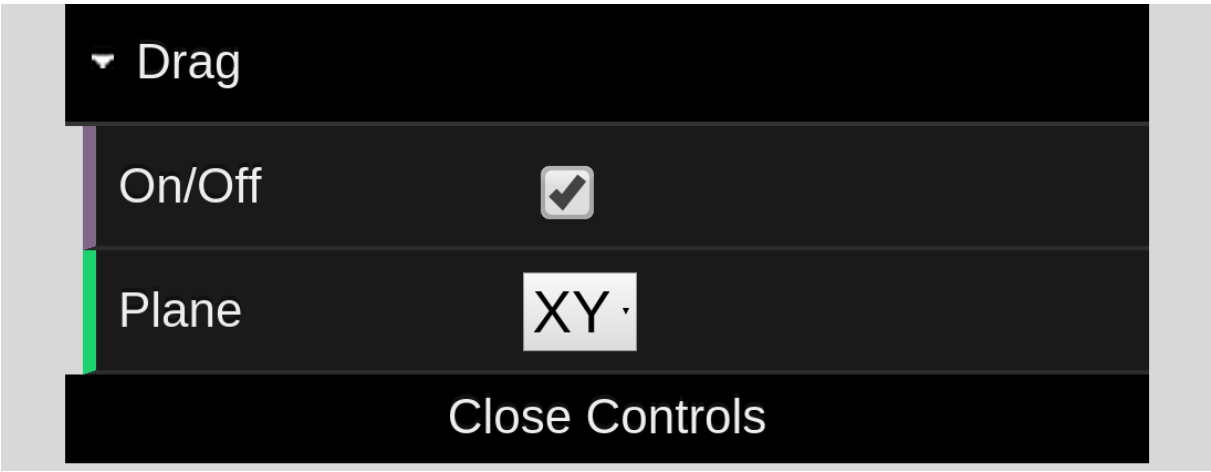
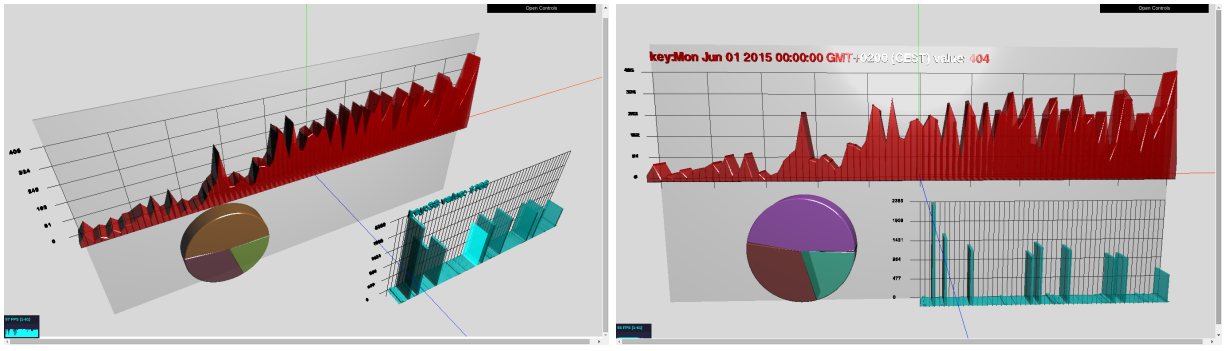


Figure 4.10: Drag's user interface

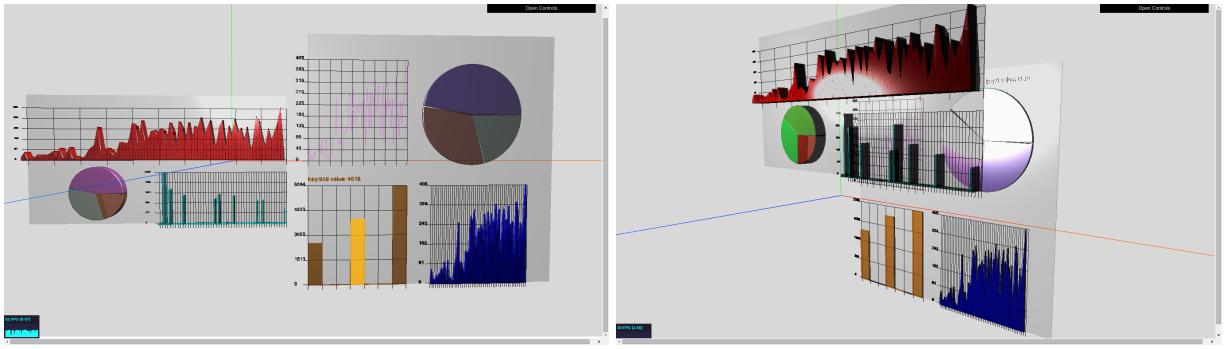
When we click on a chart or a panel with the drag activated and holding the button, we will see a white plane where we can drag around. We have the 'XY' and the 'XZ' planes to switch in the user's interface, they can be switched at any time. Finally, once we choose the position to drop the chart or the panel, just need to release the button.



(a) Chart out of the panel

(b) chart inside the panel

Figure 4.11: We can drag any chart out of the panel, if we want to get it back to its place, we can drag it manually or click on the panel and all the charts will return to their place automatically



(a) Two panels

(b) Translucent panels and charts, we can see how the light goes across the first panel

Figure 4.12: If we set the opacity property in charts and panels, we will can see through them, so we can put panels and charts behind others and compare them easily

4.9 Examples

Now let's have a look to some examples of dashboard configurations, illustrating the 3D dashboard possibilities.

We can have as many panels and charts as you want, the limit is your computer hardware, a great number of charts will lead to a performance decrease, specially if you don't have a dedicated GPU.

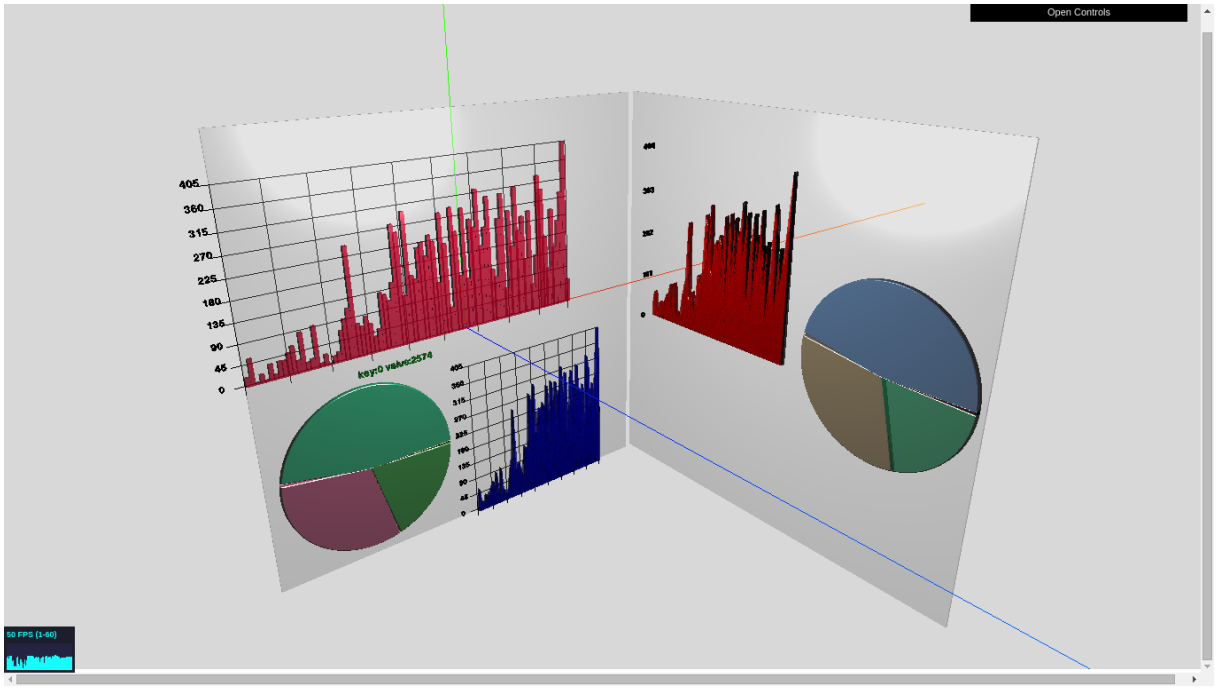


Figure 4.13: Although the charts rotation is not implemented in the framework yet, I made a simple demo to show its possibilities, in the future we will be able to put panels in different angles, forming structures, like for example, a house where the walls would be panels.

Chapter 5

Conclusions

We can say we have fulfilled the main objectives we set at the beginning of the project, that is, achieve a similar functionality to DC.js made in 3D using THREE.js and webGL with free software. That means to build a framework that allow us to create custom 3D dashboards with filtering possibilities inside any web browser, thanks to the webGL's open graphics library. So, if you have any idea about how to use DC.js and THREE.js, you can create your own 3D dashboards easily with few lines of code, if you don't, just take a look to the API [1].

However, there are additional features provided by the 3D rendering which are not possible in 2D, like overlapping charts, we can see through them thanks to the opacity setting, the 3D drag and the camera rotation features. We can also rotate the charts to place them in different angles (not implemented yet, but it will be in the near future). Of course we can place the charts and the panels at any 3D coordinates. You can see examples of this possibilities with real data sets in the previous chapter: 4.9.

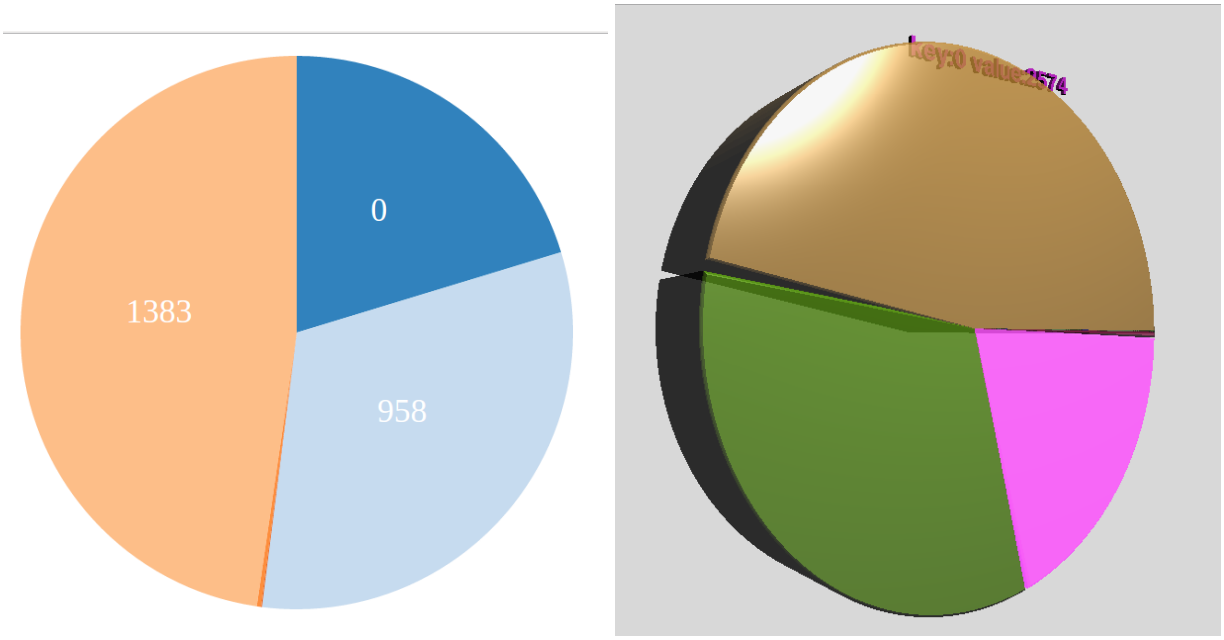
It's important to mention that the framework is 100% compatible with THREE.js, if you have any scene and you want to add a 3D chart to it, you just need to call the framework's code inside your scene.

To finish this section, we will create two pie charts in order to illustrate the similarities between DC.js and THREEDC.js syntax.

Code in DC.js:

```
//HTML5 div to attach the pie
var pieDC = dc.pieChart("#test");
```

[1] https://github.com/adrianalonsoba/threeboard/blob/master/docs/api.md



(a) Pie rendered with dc.js

(b) Pie rendered with THREEDC.js

Figure 5.1: Two pie charts with the same data.

```
pieDC.radius(100)
            .dimension(dimPerOrg)
            .group(groupPerOrg);

dc.renderAll();
```

Code in THREEDC.js:

```
//Coordinates to attach the pie, a panel could be used too.
var pieTHREEDC =  THREEDC.pieChart([0,0,0]);

pieTHREEDC.radius(100)
            .dimension(dimPerOrg)
            .group(groupPerOrg);

THREEDC.renderAll();
```

And here we have the result: 5.1

5.1 Application of lessons learned

There is a lot of subjects related to this project in my degree, at least one-third of the subjects has some type of programming, a big number if we consider that this is not a computer engineering degree, so i have good programming basis to face this project, at least i reckon so. The most related subjects could be the ones based on Javascript, like ”Ingeniería de Sistemas de Información”(Information Systems Engineering) or ”Desarrollo de Aplicaciones Telemáticas”(Telematic Application Development) where we learned web development mainly using Javacript. Other related examples would be ”Ingeniería de Sistemas Telemáticos”(Telematic Systems Engineering) and ”Servicios y Aplicaciones Telemáticas”(Telematic Applications and Services) where we learned object-oriented programming using Java and Python respectively.

5.2 Lessons learned

There is also a lot of things that i learned through the project development, here there are some examples:

- Deeper knowledge of Javascript programming language and HTML5.
- webGL use and implications, like 3D rendering techniques and concepts.
- Use of some free dashboards and dynamic charting frameworks, like Freeboard, DC and Kibana.
- Use of LaTeXby developing the project memory, an interesting document preparation system.
- English level improved, this is my first text written in English and although it has been difficult , I have learned a lot.

5.3 Future work

”A software project never ends” said several times by several teachers along the degree, there is always something to add, something to improve or something to fix, and here we have some examples:

1. Graphics improvements: add shadows, textures, reflections,... Without neglecting the performance.
2. Add more chart types and charts with more dimensions.
3. Add some type of animations.
4. Add customization settings to the graphical user's interface to change the chart's colors, sizes,...etc in real time.
5. Get the data from Elasticsearch and achieve integration with Kibana.
6. Drag improvements through making drag groups, that applies the same space transformation to all the meshes contained in the groups.
7. To implement charts rotation.
8. General optimization in order to improve the performance.

Appendix A

Appendix

A.1 HTML5 sheet

The HTML5 sheet used in the first development demo.

```
<!DOCTYPE html>
<html>
        <head>
                <title>Threeboard</title>
                <meta charset="utf-8">
                <meta name="viewport" content="width=device-width, user-
                    scalable=no, minimum-scale=1.0, maximum-scale=1.0">
                <link rel=stylesheet href="../css/base.css"/>
        </head>
        <body>
                <div id="ThreeJS" style="z-index: 1; position: absolute;
                    left:0px; top:0px"></div>
                <!--only to get the JSON -->
                <script src="https://ajax.googleapis.com/ajax/libs/jquery
                    /1.11.2/jquery.min.js"></script>
                <script src="../js/Three.js"></script>
                <script src="../js/OrbitControls.js"></script>
                <!-- to handle window size changes -->
                <script src="../js/THREEx.WindowResize.js"></script>
                <!-- full screen by pressing 'm' -->
                <script src="../js/THREEx.FullScreen.js"></script>
```

```
        <!--here is our custom code with the scene -->
        <script src="script.js"></script>
    </body>
</html>
```

A.2 Three.js and Babylon.js comparison

- Initialize a blank canvas:

Three:

```
//<div id="ThreeJS"></div>
var container = document.getElementById('ThreeJS');
```

Babylon:

```
//<div style="height:250px; width: 250px;" id="babylon">
//<canvas id="babylonCanvas"></canvas></div>
var container = document.getElementById('babylonCanvas');
```

On one hand, by using Three.js we simply create an empty div as our container for the animation. Babylon.js, on the other hand, makes use of an explicitly defined HTML5 canvas to hold its 3D graphics.

-After that, we load the renderer (engine in the case of Babylon) which will draw on the canvas:

Three:

```
var renderer = new THREE.WebGLRenderer();
renderer.setSize(width, height);
container.appendChild(renderer.domElement);
```

Babylon:

```
var engine = new BABYLON.Engine(container, true);
```

-Add scene and cameras:

Three:

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(100, width / height, 1, 1000);
```

```
camera.position.set(0,150,400);
camera.lookAt(scene.position);
```

Babylon:

```
var scene = new BABYLON.Scene(engine);
var camera = new BABYLON.ArcRotateCamera
("camera", 1, 0.8, 10, new BABYLON.Vector3(0, 0, 0), scene);
scene.activeCamera.attachControl(canvas);
```

-Create a simple cube and add it to the scene:

Three:

```
var geometry = new THREE.CubeGeometry(100, 100, 100);
var texture = THREE.ImageUtils.loadTexture('texture.gif');
texture.anisotropy = renderer.getMaxAnisotropy();
var material = new THREE.MeshBasicMaterial({ map: texture });
var cube = new THREE.Mesh(cube, material);
scene.add(mesh);
```

Babylon:

```
var box = BABYLON.Mesh.CreateBox("box", 3.0, sceneB);
var material = new BABYLON.StandardMaterial("texture", sceneB);
box.material = material;
material.diffuseTexture = new BABYLON.Texture("texture.gif", sceneB);
```

-Finally, add a simple rotate animation and call the animate loop: Three:

```
function animate() {
    requestAnimationFrame(animate);
    cube.rotation.x += 0.005;
    cube.rotation.y += 0.01;
    renderer.render(scene, camera);
}
```

Babylon:

```
engine.runRenderLoop(function () {
    box.rotation.x += 0.005;
    box.rotation.y += 0.01;
    scene.render();
```

```
});
```

A.3 Three.js's Hello world!

A commented example of basic a Three.js scene.

```
        //////////
        // SCENE //
        //////////
        scene = new THREE.Scene();

        ///////////
        // CAMERA //
        ///////////

        // set the view size in pixels (custom or according to window size)
        // var SCREEN_WIDTH = 400, SCREEN_HEIGHT = 300;
        var SCREEN_WIDTH = window.innerWidth, SCREEN_HEIGHT = window.
            innerHeight;
        // camera attributes
        var VIEW_ANGLE = 45, ASPECT = SCREEN_WIDTH / SCREEN_HEIGHT, NEAR =
            0.1, FAR = 20000;
        // set up camera
        camera = new THREE.PerspectiveCamera( VIEW_ANGLE, ASPECT, NEAR, FAR
            );
        // add the camera to the scene
        scene.add(camera);
        // the camera defaults to position (0,0,0)
        //       so pull it back (z = 400) and up (y = 100) and set the
            angle towards the scene origin
        camera.position.set(0,150,400);
        camera.lookAt(scene.position);

        ////////////
        // RENDERER //
        ////////////
```

```
// create and start the renderer; choose antialias setting.
if ( Detector.webgl )
        renderer = new THREE.WebGLRenderer( {antialias:true} );
else
        renderer = new THREE.CanvasRenderer();

renderer.setSize(SCREEN_WIDTH, SCREEN_HEIGHT);

// attach div element to variable to contain the renderer
container = document.getElementById( 'ThreeJS' );
// alternatively: to create the div at runtime, use:
//   container = document.createElement( 'div' );
//    document.body.appendChild( container );

// attach renderer to the container div
container.appendChild( renderer.domElement );

//////////////
// CONTROLS //
//////////////

// move mouse and: left   click to rotate,
//                 middle click to zoom,
//                 right  click to pan
controls = new THREE.OrbitControls( camera, renderer.domElement );

///////////
// STATS //
///////////

// displays current and past frames per second attained by scene
stats = new Stats();
stats.domElement.style.position = 'absolute';
stats.domElement.style.bottom = '0px';
stats.domElement.style.zIndex = 100;
container.appendChild( stats.domElement );
```

```
//////////
// LIGHT //
//////////

// create a light
var light = new THREE.PointLight(0xff0000);
light.position.set(0,250,0);
scene.add(light);
var ambientLight = new THREE.AmbientLight(0x111111);
 //scene.add(ambientLight);

/////////////
// GEOMETRY //
/////////////

// most objects displayed are a "mesh":
//  a collection of points ("geometry") and
//  a set of surface parameters ("material")

// Sphere parameters: radius, segments along width, segments along
    height
var sphereGeometry = new THREE.SphereGeometry( 50, 32, 16 );
// use a "lambert" material rather than "basic" for realistic
    lighting.
//   (don't forget to add (at least one) light)
var sphereMaterial = new THREE.MeshLambertMaterial( {color: 0
    x8888ff} );
var sphere = new THREE.Mesh(sphereGeometry, sphereMaterial);
sphere.position.set(100, 50, -50);
scene.add(sphere);

// Create an array of materials to be used in a cube, one for each
    side
var cubeMaterialArray = [];
// order to add materials: x+,x-,y+,y-,z+,z-
cubeMaterialArray.push( new THREE.MeshBasicMaterial( { color: 0
    xff3333 } ) );
```

```
cubeMaterialArray.push( new THREE.MeshBasicMaterial( { color: 0
    xff8800 } ) );
cubeMaterialArray.push( new THREE.MeshBasicMaterial( { color: 0
    xffff33 } ) );
cubeMaterialArray.push( new THREE.MeshBasicMaterial( { color: 0
    x33ff33 } ) );
cubeMaterialArray.push( new THREE.MeshBasicMaterial( { color: 0
    x3333ff } ) );
cubeMaterialArray.push( new THREE.MeshBasicMaterial( { color: 0
    x8833ff } ) );
var cubeMaterials = new THREE.MeshFaceMaterial( cubeMaterialArray )
    ;
// Cube parameters: width (x), height (y), depth (z),
//        (optional) segments along x, segments along y, segments
    along z
var cubeGeometry = new THREE.CubeGeometry( 100, 100, 100, 1, 1, 1 )
    ;
// using THREE.MeshFaceMaterial() in the constructor below
//   causes the mesh to use the materials stored in the geometry
cube = new THREE.Mesh( cubeGeometry, cubeMaterials );
cube.position.set(-100, 50, -50);
scene.add( cube );

// create a set of coordinate axes to help orient user
//    specify length in pixels in each direction
var axes = new THREE.AxisHelper(100);
scene.add( axes );

//////////
// FLOOR //
//////////

// note: 4x4 checkboard pattern scaled so that each square is 25 by
     25 pixels.
var floorTexture = new THREE.ImageUtils.loadTexture( 'images/
    checkerboard.jpg' );
floorTexture.wrapS = floorTexture.wrapT = THREE.RepeatWrapping;
```

```
        floorTexture.repeat.set( 10, 10 );
        // DoubleSide: render texture on both sides of mesh
        var floorMaterial = new THREE.MeshBasicMaterial( { map:
            floorTexture, side: THREE.DoubleSide } );
        var floorGeometry = new THREE.PlaneGeometry(1000, 1000, 1, 1);
        var floor = new THREE.Mesh(floorGeometry, floorMaterial);
        floor.position.y = -0.5;
        floor.rotation.x = Math.PI / 2;
        scene.add(floor);

        ////////
        // SKY //
        ////////

        // recommend either a skybox or fog effect (can't use both at the
            same time)
        // without one of these, the scene's background color is determined
             by webpage background

        // make sure the camera's "far" value is large enough so that it
            will render the skyBox!
        var skyBoxGeometry = new THREE.CubeGeometry( 10000, 10000, 10000 );
        // BackSide: render faces from inside of the cube, instead of from
            outside (default).
        var skyBoxMaterial = new THREE.MeshBasicMaterial( { color: 0x9999ff
            , side: THREE.BackSide } );
        var skyBox = new THREE.Mesh( skyBoxGeometry, skyBoxMaterial );
        // scene.add(skyBox);

        // fog must be added to scene before first render
        scene.fog = new THREE.FogExp2( 0x9999ff, 0.00025 );
}

function animate()
{
    requestAnimationFrame( animate );
        render();
```

```
        update();
}

function update()
{
        // delta = change in time since last call (in seconds)
        var delta = clock.getDelta();
        controls.update();
        stats.update();
}

function render()
{
        renderer.render( scene, camera );
}
});
```

A.4 Raycasting example

This example change the color of the meshes.

```
var raycaster = new THREE.Raycaster();
var mouse = new THREE.Vector2();

function onMouseMove( event ) {
        // calculate mouse position in normalized device coordinates
        // (-1 to +1) for both components

        mouse.x = ( event.clientX / window.innerWidth ) * 2 - 1;
        mouse.y = - ( event.clientY / window.innerHeight ) * 2 + 1;
}

function render() {
        // update the picking ray with the camera and mouse position
        raycaster.setFromCamera( mouse, camera );

        // calculate objects intersecting the picking ray
```

```
        var intersects = raycaster.intersectObjects( scene.children );

        for ( var i = 0; i < intersects.length; i++ ) {
                //change color of all intersected meshes
                intersects[ i ].object.material.color.set( 0xff0000 );
        }
        renderer.render( scene, camera );
}
window.addEventListener( 'mousemove', onMouseMove, false );
window.requestAnimationFrame(render);
```

Appendix B

Bibliography

1. Javascript:

   Marijn Haverbeke, *Eloquent JavaScript*. 2014

   http://www.w3schools.com/js/

2. Web theoretical reference:

   https://en.wikipedia.org/wiki/Main_Page

3. Three.js:

   http://threejs.org/docs/

   https://www.udacity.com/course/interactive-3d-graphics--cs291

   http://stemkoski.github.io/Three.js/

4. Crossfilter:

   https://github.com/square/crossfilter/wiki/API-Reference

5. webGL:

   https://www.khronos.org/webgl/

6. HTML5:

   http://www.w3schools.com/html/html5_intro.asp

7. Domevents:

   https://github.com/jeromeetienne/threex.domevents