Bachelor’s/Master's thesis

Game technology

NTIETSP12

2017

Joni Anttila

CREATING ROOM DESIGNER PROOF OF CONCEPT WITH THREE JS

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S / MASTER'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Game technology

Completion year of the thesis 2017 | Total number of pages 41

Author(s)

Joni Anttila

Supervisor

Principal Lecturer Mika Luimula**,** Adj.Prof.

CREATING ROOM DESIGNER PROOF OF CONCEPT WITH THREE JS

The objective of this thesis was to research the capabilities of the Three JS library to produce quick results suitable for developing a proof of concept level program in short time. An additional objective was to provide an overview of working with 3D environment and of the ThreeJS library and its features.

ThreeJS is a WebGL based library for presenting 3D graphics in a web browser environment and has been developed since 2010. Three JS was chosen for this project because it is a plain 3D graphics engine instead of being a full game engine, has comprehensive export and import options and it is known within the company already.

The client of this thesis wanted a quick proof of concept level application of a room designer which included only some basic functionalities such as drawing room outlines, generating a room from these outlines, adding movable air terminal devices to the ceiling and a simple air flow simulation. Due to tight schedule, the prototype was developed mostly in two weeks, and was developed using JavaScript, HTML and the ThreeJS library.

The project was very successful and the client was very satisfied with the results. All the feature requirements were fulfilled on time. ThreeJS proved to be suitable for this kind of prototyping as its basics are easy enough to learn quickly and able to produce results quickly. The project left much room for future development and there were also many ideas for that. Further areas for development include an accurate air flow simulation, more different product types and obstacle simulation.

KEYWORDS:

Three JS, WebGL, JavaScript, 3D

OPINNÄYTETYÖ (AMK / YAMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Koulutus

Opinnäytetyön valmistumisajankohta 2017 | Sivumäärä 41

Tekijä(t)

Joni Anttila

Ohjaaja(t)

yliopettaja Mika Luimula, dos.

HUONESUUNNITTELIJA PROTOTYYPIN KEHITYS THREE JS KIRJASTOLLA

Tämän opinnäytetyön tehtävä on tutkia Three JS -kirjaston kykyä tuottaa nopeita tuloksia, jotka sopivat prototyyppitason ohjelman kehitykseen lyhyessä ajassa. Lisäksi työ luo yleisnäkemyksen 3D-ympäristössä työskentelemisee, Three JS -kirjastoon ja sen ominaisuuksiin.

Three JS on WebGL:ään pohjautuva kirjasto 3D-grafiikan näyttämiseen web-selainympäristössä. Sitä on kehitetty vuodesta 2010 asti. Three JS valittiin tähän projektiin, koska se oli puhdas 3D-grafiikkamoottori toisin kuin monet sen kilpailijat, jotka ovat jo suoraan pelimoottoreita. Kirjasto oli myös jo etukäteen jossain määrin tunnettu yrityksen sisällä. Lisäksi Three JS tarjoaa kattavat tuonti- ja vientimahdollisuudet eri tiedostoformaatteihin.

Asiakas halusi nopean prototyyppi tason huonesuunnittelijaohjelman, joka sisälsi joitain perus ominaisuuksia kuten huoneen ääriviivojen piirtäminen, huoneen generointi ääriviivojen perusteella, päätelaitteiden lisääminen kattoon ja yksinkertaisen ilmavirta simulaation. Prototyyppi kehitettiin pääosin kahdessa viikossa käyttäen JavaScriptiä, HTML:ää ja Three JS -kirjastoa.

Projekti oli erittäinen onnistunut ja asiakas oli tyytyväinen saatuun lopputulokseen. Kaikki halutut ominaisuudet saatiin kehitettyä aikarajan puitteissa. Three JS osoittautui olevan sopiva kirjasto tämän tyyppiseen prototyypin kehitykseen, sillä sen peruskäyttö on riittävän helppo omaksua nopeasti ja se saa aikaseksi näkyviä tuloksia nopeasti. Projekti jätti paljon tilaa tulevaisuuden kehitykselle, johon myös oli paljon ideoita. Lisäominaisuudet voisivat olla esimerkiksi tarkka ilmavirtasimulaatio, uusia tuettuja tuoteluokkia tai estesimulaatio.

ASIASANAT:

Three JS, WebGL, JavaScript, 3D

CONTENT

PICTURES

TABLES

LIST OF ABBREVIATIONS (OR) SYMBOLS

Abbreviation	Explanation of abbreviation (Source)
JSON	JavaScript Object Notation. Simple human readable data exchange format.
OBJ	File format used to store object data.
API	Application Programming Interface. With APIs different programs can make requests and exchange data between each other.
GPU	Graphics Processing Unit. Graphics processor in graphics card specialized in rendering images.
CPU	Central Processing Unit. Executes commands given by a computer program.
SVG	Scalable vector graphic. XML-based format for describing 2D vector-based graphics.
VR	Virtual reality.
DOM	Document Object Model.Programming interface to allow modification of web page elements.
UI	User interface.
HTML	Hypertext Markup Language. Markup language for creating web pages and applications.
AOT-compilation	Ahead-Of-Time compilation. Compiling higher level programming language into native machine code.
WebGL	JavaScript API for presenting 3D graphics
Three JS	WebGL based JavaScript library for presenting 3D graphics in web browser.
Silverlight	Application framework for creating web applications.
GLSL	OpenGL Shading Language. A high-level shading language.

Blender	A 3D modeling software.

1 INTRODUCTION

The trend during the last years in software development has been that the applications are increasingly moving from stand-alone applications to web applications. However most of the graphics content is presented in 2D. Traditionally all the graphically demanding applications have been stand-alone, but lately since the rise of WebGL many graphically more impressive applications and games have been seen on web browsers. This kind of shift makes many programs more easily accessible and removes the effort of actually installing the application.

The client for this thesis is Progman Oy. The assignment was to create a proof of concept of a room designer web application. The application could be used to investigate viability of application in room design and the need for this kind of product in the market. On more theoretical side this thesis' goal is to evaluate how well today's top 3D web technologies perform in tasks that might have been done in stand-alone application before. After a comparison between few different 3D presentation framework Three JS was chosen to be used in this proof of concept application. As Three JS was chosen to be used in the work, the thesis will look into the different features of the framework.

The first chapter describes that basics of presenting 3D graphics. The topics reviewed are generally required by most of the game engines or 3D graphic presenters. The chapter begins by explaining behavior of renderers, continuing to scenes and moves all the way to lighting, textures and meshes

The second chapter of the thesis introduces the Three JS 3D library which is one of the most popular libraries of its kind. The chapter will talk about some Three JS's features and showcases some of its more visual feats and applications.

The third chapter outlines the requirements that were given for the project. Then it reviews the actual development of the proof of concept project. The development is described on relatively technical level, discussing what decision had to be made during the development process and what hardships were confronted. Finally the results of the project are evaluated.

The fourth chapter reviews how well the work's preset criteria were met. Next the performance of Three JS for this kind of project is evaluated. This chapter ends with projection of how this project could be developed further.

2 3D GRAPHIC PRESENTATION BASICS

2.1 Rendering

To present 3D-graphics with most game engines or libraries, few components are required. First of all, a renderer needs to be present. Renderer is a component which is responsible for drawing the graphics to the screen. This is technically very complex procedure, which means doing the calculations to transform the mathematical data of a scene to form a 2D image. A good analogy would be to compare rendering to taking a photograph with and old camera which requires the photo to be developed and printed before it can be viewed. There are two types of rendering: real-time and pre-rendering. In real-time rendering, as its name suggests, the rendering is done in real time, while in pre-rendering the rendering can be done beforehand. This means that pre-rendering can be applied for purposes that aren't time critical. Common usage for pre-rendering would be rendering a video or a single image. Real-time rendering is used for games and other applications where the next image to be rendered is unknown. This is the reason why for example many game cinematics can include much better graphics than the actual game since the video can be rendered beforehand and it is allowed take more time. (Justin Slick. 2017)

2.2 Scene

Secondly, a scene is required. Scene is a container that holds all the data of the objects that are in the world that is wanted to be renderer. The scene is perceived through an object called camera. The camera could compared to eyes: both can be used to observe the surrounding, the "scene". What is shown through the camera is what is being rendered by the renderer.

2.3 Mesh, texture and material

One of the most common item to be added a to a scene is mesh. Mesh is an object that contains information of the objects shape. A texture and a material can be assigned to a mesh to make it look as wanted. If it was wanted to add a wooden table to a scene, first a mesh shaped of a table would be added. Then it would be assigned a texture, a picture of a surface of a wooden table, to it to make it look like it's made of wood. Still it wouldn't, for instance, reflect light as a wooden table would. To correct this, a material is also assigned to the mesh, which modifies how its surface behaves when it is hit by light. In case of normal wood, it wouldn't reflect very much of the light.

2.4 Light

Finally after having added a mesh to the scene, it still requires light to be seen. Without light, the scene would be only dark and only black screen would be rendered. This can be handled by adding a light source to the scene.

These are the steps required by most game engines or similars to view a simple object on the screen.

2.5 Coordinate system

In order to position the created object in a 3D space, a cooridante system is required. The most common coordinate system used is the Cartesian Space, which introduces x, y and z that describe the object position horizontal placement, vertical placement and depth, respectively. This coordinate system is mutually perpendicular meaning that each of the three axes have a 90 degree angle between them. The object's position can then be described in a vector in relation to the space's origin, for example like this: (2, 1, 3). The object's position presentation is also known as a tuple, which means an orderered list of elements. A tuple with three elements would then be called a 3-tuple. The points

in a tuple defining a position in a 3D coordinate system are scalars that specify their position along a basis vector. The three basis vectors in a 3D space are the following (Kyle Sloka-Frey. 2013):

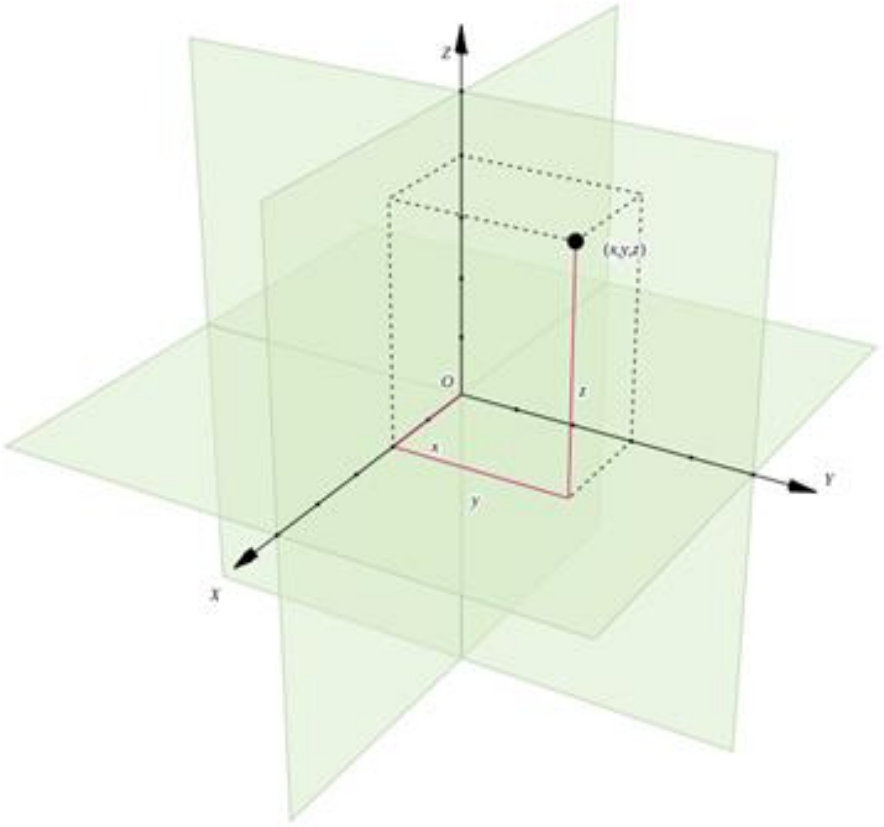
X = (1,0,0)

Y = (0,1,0)

Z = (0,0,1)

The length of each of these vectors needs to be exactly 1, meaning that they are unit vectors. (Kyle Sloka-Frey. 2013)

The point (1,1,1) position in the coordinate system would look like this:



Picture 1. Describing point (1,1,1). (Sloka-Frey, 2013)

Inside the 3D engine moving the object around the space results in adding or substracting vectors.

3 THREE JS

Three JS is a 3D library written in JavaScript. It allows user to create and present 3D graphics and animations in web browser. Three JS relies on WebGL for doing this which gives it the advantage of not needing any browser plugins. Thus it also works on any browser that supports WebGL, which is basically almost any modern day web browser. However programming WebGL itself directly is a demanding task, and requires great amount of knowledge of the details of WebGL. For best results one should also know how to write shader code. This is one of the strengths of Three JS since it allows creation of attractive 3D graphics without the need of learning WebGL. (Nick Pettit. 2013)

Three JS was introduced by Ricardo Cabello (also known as Mr.doob) in April 2010. The birth of Three JS can be tracked to early 2000s when the code was initially developed in ActionScript. In 2009 Cabello decided to port the code to JavaScript. Main deciding factors for choosing JavaScript was its ability to perform on many platforms and removing the need for compiling. Moving the renderer to JavaScript was fairly easy task since the arrival of WebGL and the fact the Three JS renderer was coded as a separate module instead of being part of the core. In addition to Cabello, some of the most important contributors for the project have been Branislav Ulicny, Paul Brunt and Joshua Koo. In total there have been over 650 contributors. (Ricardo Cabello. 2012)

WebGL itself is one of the more visually notable feature of HTML5 standard as it can directly use the processing resources of the GPU. It is developed by Khronos Group and its initial launch in 2011 was backed up by many big companies including Apple, Google, Mozilla and Opera. Like Three JS, it is a JavaScript API, but is far less accessibile. This is due to WebGL functioning on rather low level. WebGL, based on OpenGL ES 2.0, combines JavaScript with shader code which makes it very powerful tool. JavaScipt is used for controlling while shader code is cooperating the with the Graphics Processing Unit. The output is presented through the HTML5 canvas element as Document Object Model interfaces. Current version of WebGL is 1.0, but the version 2.0 is also being developed. (Khronos Group. 2016)

3.1 Why Three JS?

One of the reasons why Three JS was very suitable for this project was simply that it was meant to be used to present 3D-content. Many libraries that have same kind of features are straight up game engines. While Three JS can certainly be used to create games, it still is contains only the essential features for showing 3D content and lacking the game engine elements. This was very desirable for the project since any of the game engine features were not needed, and the library needed to be as lightweight as possible. Although some physics may be needed as the project proceeds, it is not in the way that game engines use it. Three JS is one of the oldest libraries and most popular of its kind and therefore it also has relatively good documentation and is reasonably mature. An important requirement for this project was that there are good import and export features. Three JS features different file type imports for models of which JSON import was important. While there wasn't any finished projects using Three JS within the company, there had been few protype level ones. The developers had been very satisfied with Three JS which encouraged to use it in this project as well.

Some of the alternatives that were considered are Babylon JS and Unity. Both of these are game engines. Unity has officially featured support for WebGL since the version 5.3. While Unity certainly is easy to use it was too heavy for this project. Babylon JS was originally designed to be a Silverlight game engine which then evolved to be a JavaScript library capable of handling game development as well as generally presenting 3D-graphics and animations on web browsers.

3.2 Features

There are number of things that making them with WebGL would be very difficult that are almost a trivial task for Three JS. Some of these features according to its creators are the following (Three JS documentation, 2016) :

- Renderers: WebGL, <canvas>, <svg>, CSS3D, DOM, Software; effects: anaglyph, crosseyed, stereo and more
- Scenes: add and remove objects at run-time; fog
- Cameras: perspective and orthographic; controllers: trackball, FPS, path and more

- Animation: morph and keyframe
- Lights: ambient, direction, point, spot and hemisphere lights; shadows: cast and receive
- Materials: Lambert, Phong and more - all with textures, smooth-shading and more
- Shaders: access to full WebGL capabilities; lens flare, depth pass and extensive post-processing library
- Objects: meshes, particles, sprites, lines, ribbons, bones and more - all with level of detail
- Geometry: plane, cube, sphere, torus, 3d text and more; modifiers: lathe, extrude and tube
- Loaders: binary, image, JSON and scene
- Utilities: full set of time and 3D math functions including frustum, quaternion, matrix, UVs and more
- Export/Import: utilities to create Three.js-compatible JSON files from within: Blender, CTM, FBX, 3D Max, and OBJ
- Support: API documentation is under construction, public forum and wiki in full operation
- Examples: More than 150 files of coding examples plus fonts, models, textures, sounds and other support files

3.2.1 Renderers

Even though Three JS offers different renderers, the primary option should always be the WebGL renderer. It is the most powerful, fastest and modern of the renderers. The other officially supported renderer is CanvasRenderer. The CanvasRenderer draws the scene not using the WebGL but the Canvas 2D Context API. It is noticeably slower than its WebGL counter-part, but it loses the requirement for WebGL. This might be desirable sometimes for example when an older browser is targeted. For instance Internet Explorer gained WebGL support relatively late. Another option is the SVGRenderer, which is not part of the official library but is provided in the examples. SVG (Scalable Vector Graphics) is a language for describing 2D graphics in XML. It relies on vector graphics which gives it the advantage of resolution independency. Biggest difference to CanvasRenderer is that the SVG remember each drawn shape as and object while CanvasRenderer forgets the graphic once it is drawn. This means that using SVG only the shape needs to be renderer and when using Canvas, the whole scene needs to be redrawn. (W3 Schools, 2017)

3.2.2 Scenes

Scenes are the basis of creating a 3D graphic presentation for browser. It is an environment keeping track of all the objects that are present, including cameras, light and shapes. Without a scene, nothing can be rendered. Setting up a scene in Three JS is very easy, only few lines of code wrapped in some basic html markup:

```
<html>
    <head>
        <title>Three.js example</title>
        <style>
            body { margin: 0; }
            canvas { width: 100%; height: 100% }
        </style>
    </head>
    <body>
        <script src="js/three.js"></script>
        <script>
            var scene = new THREE.Scene();
            var camera = new THREE.PerspectiveCamera( 75, window.innerWidth/window.innerHeight, 0.1, 1000 );

            var renderer = new THREE.WebGLRenderer();
            renderer.setSize( window.innerWidth, window.innerHeight );
            document.body.appendChild( renderer.domElement );

            camera.position.z = 5;

            var render = function () {
                requestAnimationFrame( render );

                renderer.render(scene, camera);
            };

            render();
        </script>
    </body>
</html>
```

This code snippet creates a Three JS scene and adds a camera and a renderer for it and then enters the render loop. The scene is now ready to have some shapes added to it.

3.2.3 Cameras

Three JS features two types of cameras: perspective and orthographic camera. Perspective camera is the more natural one and shows the scene like we perceive the world: the objects further are small and the ones near are biggers. Using orthographic camera all the objects retain their original size regardless of the distance to the camera. Perspective camera is more common because of its resemblance to the real world, but orthographic camera can also be very useful in certain situations. For example it is used

in many older city and world builder games. It can be also useful when measuring or drawing something.

3.2.4 Animations

Simple animation can be achieved by using Tween JS. With it you can change properties of an mesh over time, for example change its position. When animating with external programs user can choose between two types of animations using Three JS: morph and skeletal animations. To do morph animation, a deformed version of the mesh is required. This mesh includes all the vertex position which's position then can be moved to create animation. For skeletal animation, a skeleton needs to be defined in the program. Each bone as a number of vertices and by moving these vertices the bone is moved, and any bone attached will also be moved. However it might prove to be quite challenging to get a good working model export from the animation program which means that in many cases using morph animations might prove to be easier.

3.2.5 Lights

Lights are required to light up the scene. Although some basic materials will be visible even without light, the more advanced and beautiful materials will require light to be seen. Three JS offers 5 different light options: ambient light, directional lights, hemisphere light, point light and spot light. Ambient light is just a general light that illuminates the whole scene, and is applied to all the objects in the scene. Directional lights have a position and a target, meaning that they will be directed towards a target position. It acts if it were very far away and the light will be hitting all the objects from the same angle, kind of like the Sun. Hemisphere light is a light that is directly above the scene. Point light is a punctate light that can be moved around the scene and it will illumante it's surroundings. Spotlight resembles very much a real world spot light. It is set to a position and then it is given a target. The spotlight will then light the target area in a cone. (Three JS, 2017).

3.2.6 Materials

Materials are assigned to a mesh and they have a strong effect on looks of a shape. They define, for instance, how much light the object reflects, how transparent it is or even if it's wireframed. Three JS features many different materials: basic, depth, normal, face, lambert, phong and few special materials. The basic material doesn't really involve anything besides the color. It also doesn't require a light in the scene to be seen. The depth material determines the color of the object depending on it's distance from the camera. The normal material bases the color a face according to its normal. The face material provides possibility to give each face own material. Lambert material can be used to make objects appear non-shiny, turbid. On the contrary phong material gives objects a metallic, shiny look. Both lambert and phong materials take light into account and reflect the light according to the material's properties.

3.2.7 Shaders

Shaders can be used to draw special effects to graphics. Shaders are essentially a piece of GLSL code that is ran directly on the GPU without having to burdening the CPU. Both two types of shaders, the pixel and the vertex shader, are supported by Three JS. The vertex shader manipulates the vertices of an object while pixel shader modifies the pixels in the scene. Three JS has some shaders out of the box, but it also supports custom shaders.

3.2.8 Geometry

Geometry is responsible for holding all the data required to represent a 3D model. Three JS comes equipped with many shapes such as sphere, box, torus, plane and many others. Geometries can also be crafted by hand by giving out all the geometry's vertices and faces. To show a geometry in scene, it has to be added to a mesh, and a material needs to be assigned to the mesh. The procedure in the code looks like this:

```
var geometry = new THREE.BoxGeometry( 1, 1, 1 );
var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
var cube = new THREE.Mesh( geometry, material );
scene.add( cube );
```

3.2.9 Loaders and Import & Export

Three JS features many different loaders. Some of the supported formats are JSON, Obj, different kind of image and scenes. These loaders can be used to bring external resources into your Three JS scene. 3D models can be imported using a loader and a JSON or Obj file for example. Importing models from many popular 3D modeling tools such as Blender or 3DS Max is also possible thanks to their abilities to export into JSON files. In Blender's case this requires a community built plugin. This also works the other way around: Three JS can export its geometries, scenes, materials and other objects into different formats like JSON. (Petitcolas. 2015.)

3.3 Applications

As mentioned earlier on, Three JS can be used for many different things like games, marketing materials, visualizations or even interactive music videos. It could be utilized for interactive marketing materials for example.

3.4 Framework comparison,.

This comparison compares three different technologies that can be used to produce WebGL content for browsers: Three JS, Unity and Babylon JS. Unity is one of the most popular game engines in the market that can also build games into WebGL application. Babylon JS is an open-source WebGL based 3D-engine that was originally created by some developers of Microsoft.

Framework	Three JS	Unity	Babylon JS
Language	JavaScript	C#, UnityScript	JavaScript
License	MIT	Propiertary	Apache License 2.0
Mobile support	Yes	No	Yes
Development environment	No restrictions	Unity	No restrictions
Intergrated physics	No	PhysX	Cannon JS, Oimo JS, Energy JS
Import	JSON, OBJ, FBX	OBJ, FBX	OBJ, FBX, Babylon, STL, JSON
Export	JSON, OBJ	No	Formats supported by Blender and 3dsMax
VR support	Yes	Yes	Yes

Table 1: Framwork comparison

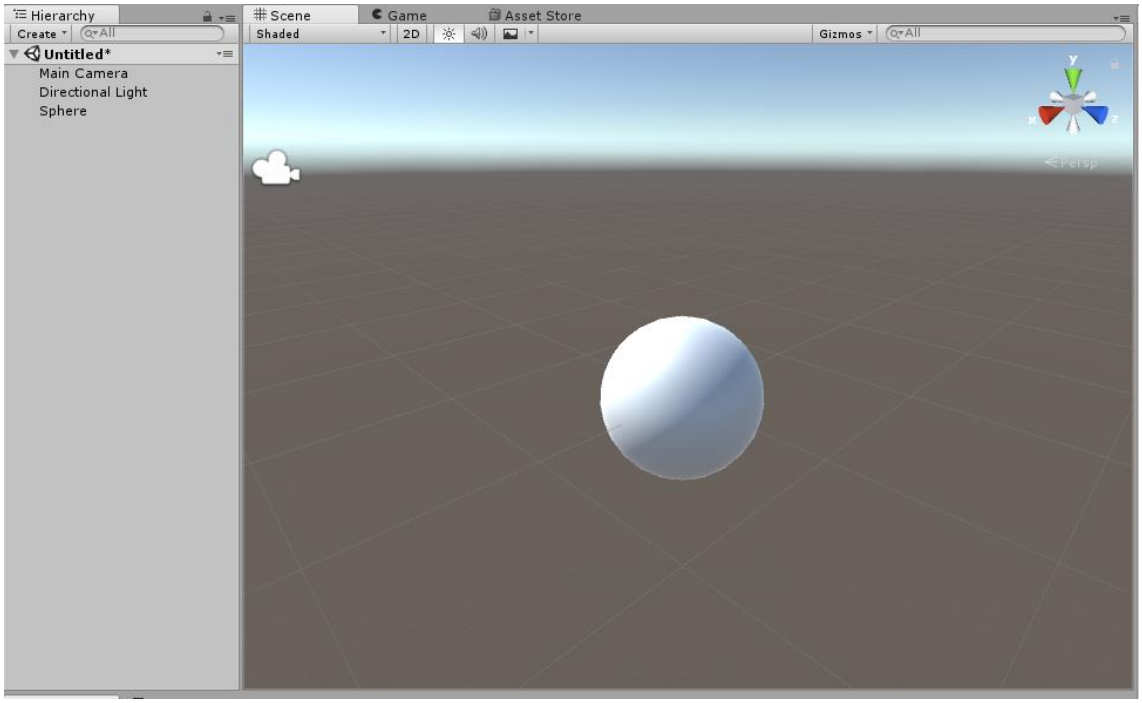
3.4.1 Unity and WebGL

Unity has been officially supporting building into WebGL since the version 5.3. (Matthew Jarvis. 2015). Before this version browser compilation was possible using the Unity Web Player that required a plugin to be installed to the web browser. Nowadays Unity allows the user to convert their C# or UnityScript code into JavaScript code that utilizes HTML5 technologies and the WebGL rendering API. In this conversion process a compilation chain is used as the user created game code is converted into C++ using technology called IL2CPP which then again is complied using emscripten into JavaScript. The Unity runtime code, which is written in C or C++, is also converted into JavaScript for it to work in browser. The output is asm.js, which is a very optimizable subset of JavaScript that allows JavaScript to AOT-compile it into efficient native code. Every major browser is supported by the Unity WebGL to some degree as the level of support and performance varies between browsers. For example Firefox and Edge support the asm.js AOT compilation which may give performance boost since that is also what Unity uses itself. (Unity, 2016)

When developing WebGL content in Unity the user has to use C# or UnityScript instead of JavaScript. This can be very beneficial for developers that are more familiar with the said languages. Though some loss in performance is possible due to the required conversion. Developing with Unity also introduces a new step in application launch as it has to be built separately each time. Which can be relatively lengthy process when compared to the ability of the rival technologies being able to just save the code changes and run the program in browser. One of the biggest shortcoming of Unity WebGL is its incompatibility with mobile devices, which is one of greatest strengths of the original WebGL. Though this will probably be supported by Unity WebGL aswell in the future.

Setting up a basic scene differs a bit from Three JS and Babylon JS, since Unity has its own editor. All the basic things like creating a scene, adding a mesh and a light can be done straight from the editor without coding a single line. This is very basic functionality of Unity and anyone with little experience with the program can do this easily just by adding the necessary objects to the scene from Unity’s menus. Unlike ThreeJS and

BabylonJS, Unity doesn't require a creation of rendering loop which is done by the game engine automatically.



Picture 2: Basic scene with a sphere on Unity

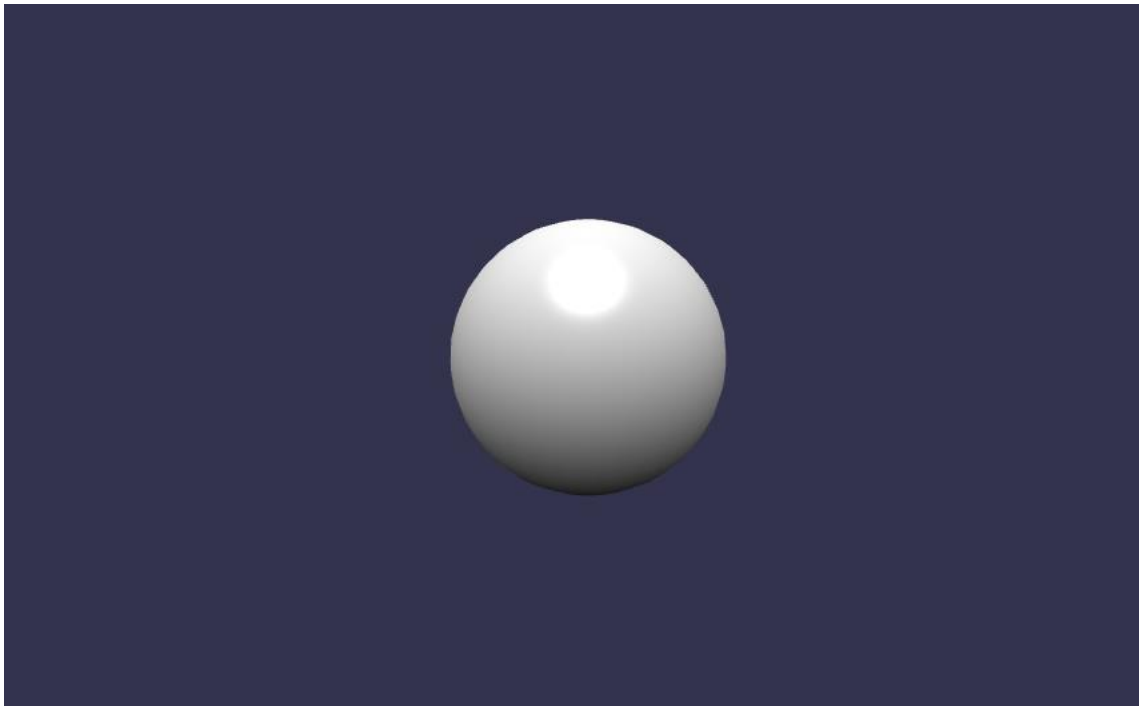
3.4.2 Babylon JS

WebGL development with Babylon JS resembles Three JS a lot more than development with Unity. Main difference is that Babylon JS is meant to be a game engine while Three JS marketed merely as a web 3D content presentation framework. As a game engine Babylon JS features many things that simplify game development such as native supports for collision detection, gravity and game-oriented cameras. Featurewise both of these frameworks offer very similar sets. (Joe Hewitson. 2013). Babylon JS even offers support for VR device such as Oculus Rift. (Raanan. 2015)

Babylon JS is developed using JavaScript as are most of the similar libraries. Like Three JS, only the source code for Babylon JS needs to be downloaded from their website.

Setting up a scene with a single object is very simple with Babylon JS, and is very similar to setting up a scene with Three JS: defining the canvas where the content is drawn, loading the 3D engine, creating a scene and then adding the object in the desired objects to the scene. Then a rendering loop is created in a very similar fashion to Three JS.

```
<script>
    window.addEventListener('DOMContentLoaded', function(){
        // Get the canvas DOM element
        var canvas = document.getElementById('renderCanvas');

        // Load the 3D engine
        var engine = new BABYLON.Engine(canvas, true);

        // Create a basic Babylon JS Scene object
        var scene = new BABYLON.Scene(engine);
        // Create a FreeCamera, and set its position
        var camera = new BABYLON.FreeCamera('camera1', new BABYLON.Vector3(0, 5,-10), scene);
        // Target the camera to scene origin
        camera.setTarget(BABYLON.Vector3.Zero());
        // Attach the camera to the canvas
        camera.attachControl(canvas, false);
        // Create a basic light, headed to the sky
        var light = new BABYLON.HemisphericLight('light1', new BABYLON.Vector3(0,1,0), scene);
        // Create spehere mesh. Its constructor takes 5 params: name, width, depth, subdivisions, scene
        var sphere = BABYLON.Mesh.CreateSphere('sphere1', 16, 2, scene);

        // Render loop
        engine.runRenderLoop(function(){
            scene.render();
        });

        // Handle window resizing
        window.addEventListener('resize', function(){
            engine.resize();
        });
    });
```



Picture 3: The Babylon JS output

4 IMPLEMENTING THE PROOF OF CONCEPT

4.1 Requirement

The task was to create a proof of concept level application that helps in designing building's rooms and their air flow more specifically. The application could be used as a marketing tool and to research such tool's viability. The project started with rather strict schedule, so Three JSs' ability to quickly generate visual content was put to test.

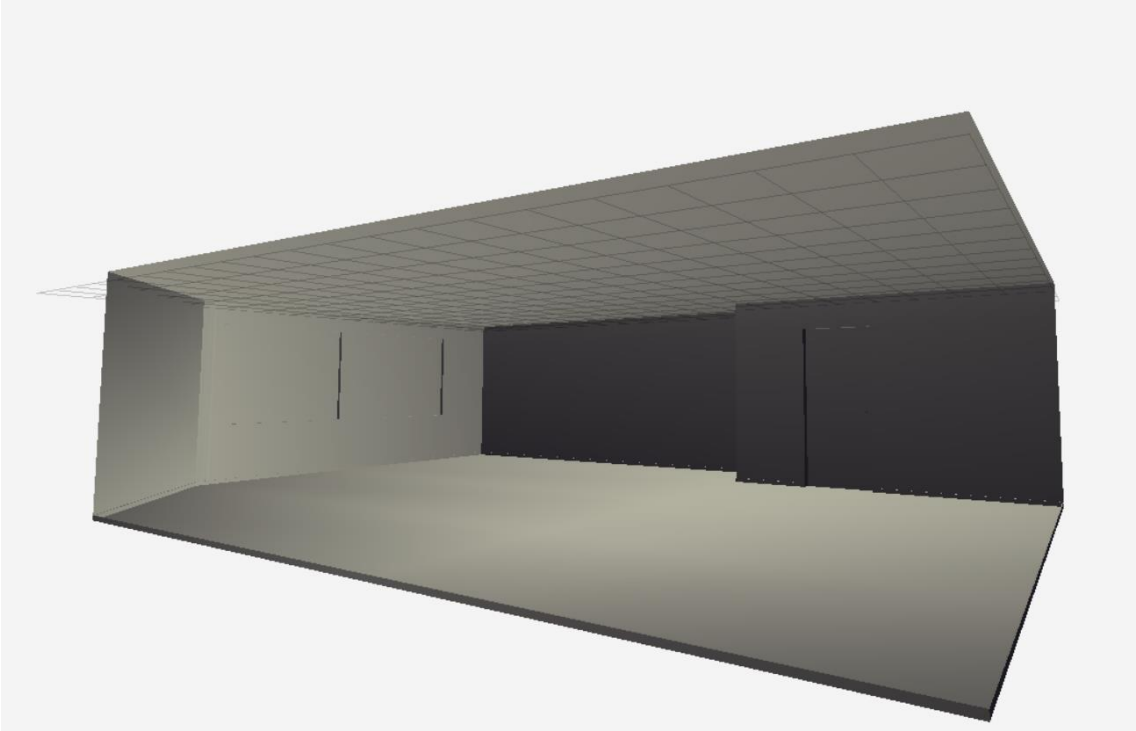
The application was required to allow a user to draw the outlines from point to point for a room. Then the room would be generated accordingly as a 3D model. User needs to be able to add air terminal devices to ceiling, and move them around the room. There also needs to be option to show the throw pattern of the air terminal devices to how the air flows would behave in the room. In addition some kind of UI was requested to give an idea how it might look like. However it wasn't required for the UI to be fully functional.

On technical level, the program was required to be a web application for easy access without the effort of installation. The full product would also have to integrate into some of the company's other products, but this wasn't required in the proof of concept phase.

4.2 Viewing the room

4.2.1 The model

The first thing we wanted to do in this project was to get a 3D model of a room to a browser and to be able to view and rotate it around. The ultimate target was that the user could draw to room himself, but at the beginning we decided just to model a room using Blender and then import the model to the scene. The model was exported from Blender as a JSON file and then imported to Three JS using the JSON Loader.



Picture 4. The Blender model in browser

4.2.2 Rendering

When rendering the room, it is important that only the front side of each plane is shown, so that the inside of the room is revealed. In Blender model's case this mean that all the normals have to point to inside of the room. Three JS makes this is very easy to do since this kind of behavior can be achieved through a single parameter of a material.

4.2.3 Camera

For viewing the room's 3D model we chose to use the perspective camera. This was a rather obvious choice for trying to present the room in most natural manner. As for moving the camera there was more options as Three JS offers many different camera controls out of the box. We ended up using the orbit control scheme which is good for viewing a single object. by holding the left mouse button down the camera can be moved

around a point in the scene which in this case is always in the center of the screen.. The orbit point in the scene can be moved by holding down the right mouse button. Zooming towards the orbit point is done by scrolling the mouse wheel.

4.3 Devices

In the proof of concept application we decided to simply create a button that would add a air terminal device to the scene, no user selection would be necessary at this point. The first and the most important device would be the air terminal devices, which are positioned in the ceiling. The user needed to be able to move the devices in a grid like manner. The actual grid would also be added later on.

4.3.1 Movement

The idea for the movement was simply that the user could drag the device with mouse and move it in the ceiling grid. The first thing to do in achieving this was to determine the position where in the scene the mouse button was pressed down. To do this a raycast was sent from the camera to the mouse position and checked if it hit any of the devices. If it did, the device was moved along with the mouse as along as the mouse button was pressed. The device raycast hit was only used to check if a device was hit. To calculate the correct position we also had to add a plane to the ceiling below the device to correctly offset the position since the device object was a bit above the actual ceiling. To calculate the grid movement, the intersection point of the ray and the ceiling plane had to be stored and passed on to the function that would calculate to which grid cell the device should be moved to.

The grid movement function:

```
var moveObjectOnGridHorizontal = function (object, intersect) {
    object.position.copy(intersect.point).add(intersect.face.normal);
    object.position.divideScalar(0.5).floor().multiplyScalar(0.5).addScalar(0.25);
    object.position.y = webGLViewInstance.roomHeight + 0.05;
    object.position.z -= 1;
}
```

First the function adds the ceiling plane's face's normal to the intersection point vector. Then object position is rounded by dividing the scalar by 0.5, then rounding the result down and then multiplying it again by 0.5. This way we can ensure that the object is always in the center of a grid cell. After the divide and multiply operations a scalar of 0.25 is added to the position to jump the object into the correct cell. There also has to be some offsetting for y and z axis as well. For the y axis it is done to ensure that the device in correct height, and to the z axis to correct some errors from the mouse hit position calculations.

Another important function for the ceiling plane was that using it we could ensure that the device could not be moved out of the ceiling. This could be done by checking that the raycast hit both the device object and the ceiling plane.

4.4 Grid

The grid was required to generated on the ceiling of the room. It could not be hard coded since it would have to match a user drawn room too. There was also an idea that there could be objects placed on walls, so the grid generator needed to be able to produce both horizontal and vertical grids. The grid should be exactly the size of the room's ceiling, so we'd have to figure how to solve that, since the ceiling wouldn't always be rectangular.

4.5 Drawing the grid

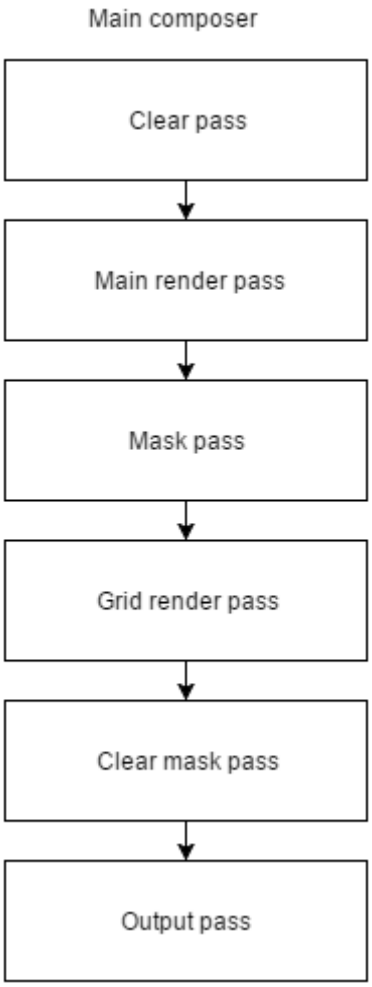
The room would be drawn from point to point, so we'd know all the corners of the room. Easiest way to generate an exact sized grid was to figure out the furthest corners of the room and draw a rectangular grid over the room using those points and the hide the parts that weren't on the ceiling. To generate the rectangular grid I came up with the following code.

```
  var pointGrid = function (startPoint, endPoint, startPointUp, endPointUp, step) {
    var horizNumber = startPoint.distanceTo(startPointUp) / step;
    var currentStep = 0;
    var geometry = new THREE.Geometry();
    var baseVectorHorizontal = new THREE.Vector3(endPoint.x - startPoint.x, endPoint.y - startPoint.y,
endPoint.z - startPoint.z);
    var normalizedHorizontalBaseVector = baseVectorHorizontal.clone().normalize();

    // Horizontal grid lines
    for (var i = 0; i < horizNumber; i++) {
      geometry.vertices.push(new THREE.Vector3(startPoint.x, currentStep, startPoint.z));
      geometry.vertices.push(new THREE.Vector3(endPoint.x, currentStep, endPoint.z));
      currentStep += step;
    }

    // Vertical grid lines
    var vertNumber = startPoint.distanceTo(endPoint) / step;
    currentStep = 0;
    for (var i = 0; i < vertNumber; i++) {
      var additiveVector = normalizedHorizontalBaseVector.clone().multiplyScalar(currentStep);
      geometry.vertices.push(new THREE.Vector3(startPoint.x, startPoint.y,
startPoint.z).add(additiveVector));
      geometry.vertices.push(new THREE.Vector3(startPointUp.x, startPointUp.y,
startPointUp.z).add(additiveVector));
      currentStep += step;
    }

    var material = new THREE.LineBasicMaterial({ color: 0x000000, opacity: 0.2, transparent: true });
    var line = new THREE.LineSegments(geometry, material);
    line.updateMatrix();
    line.position.y = 0;
    return line;
  }
```

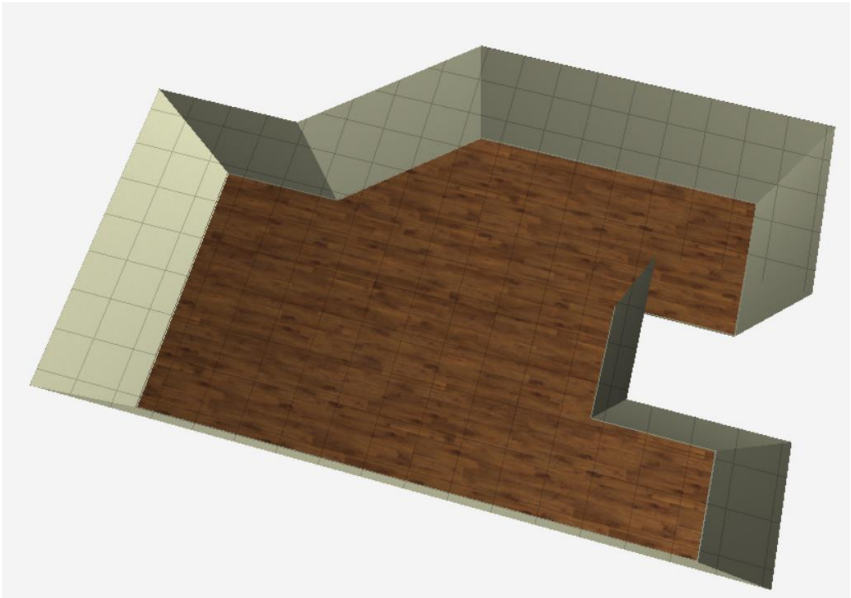
The function takes in the corner points of the grid and the grip step size. It then calculates the how many horizontal grid lines needs to be drawn. Then it stores the normalized vector of the horizontal lines and loops through all the needed horizontal lines adding the needed vertices to the grid geometry. Next the number of vertical lines is calculated and same kind of loop is applied to them. The difference is that their start and end points are unknown. On each loop, an additive vector is drawn between the start point and the end point which then is multiplied by the step scalar. Then the additive vector is added to the actual vertical vector to move it to the correct horizontal position. Now we have all the required vertices and a line segment can be formed of these.

4.5.1 Masking the grid

The problem with the drawn grid is that it is always rectangular. The room can however be also in non-rectangular shapes. To draw the grid into exactly the shape of the room would be relatively difficult so we decided to use postprocessing and mask the overflowing part of the rectangular grid away. To do this, the grid and the actual room had to be placed into different scenes, so there would be a grid scene, and a scene with rest of the objects. Render postprocessing in Three JS is based on EffectComposer class which can be used to add postprocessing passes. On render loop EffectsComposer renders the scenes using the applied passes and then shows the output on the screen.



Picture 5. Pass apply order



Picture 6. Ceiling grid

4.6 Draw tool

One of the most time consuming features of the project was the drawing tool. The tool was required to enable realtime drawing of straight lines between last user entered point and current mouse position. A mouse click would enter a point and drawing of the next line would begin. After the user closed the room outlines, by entering a point to the start point of the drawing, the room would be automatically generated and shown to the user. The drawn line needed to snap into angles of 45 and 90 degrees in relation to the last drawn line. Snapping was also required when the drawn line matched the vertical or horizontal levels of previously drawn lines. Upon of each of the snap events that were done relative to previously drawn lines, a green helper line was required to be drawn between them to help the user to perceive the relation correctly. Pressing right mouse button would erase last drawn line.

4.6.1 Drawing

The drawing was decided to be done using orthographic projection instead of the perspective projection used in the rest of the application. This was chosen so that the ratios of distances between drawn vertice would be preserved. Basically this just meant using orthographic camera instead of the perspective camera used in the rest of the application. Key idea behind the drawing was that each user entered point, more specifically a room vertex, on screen would be stored in an array. After the user had completed the room, a plane would be generated in between each successive vertice.

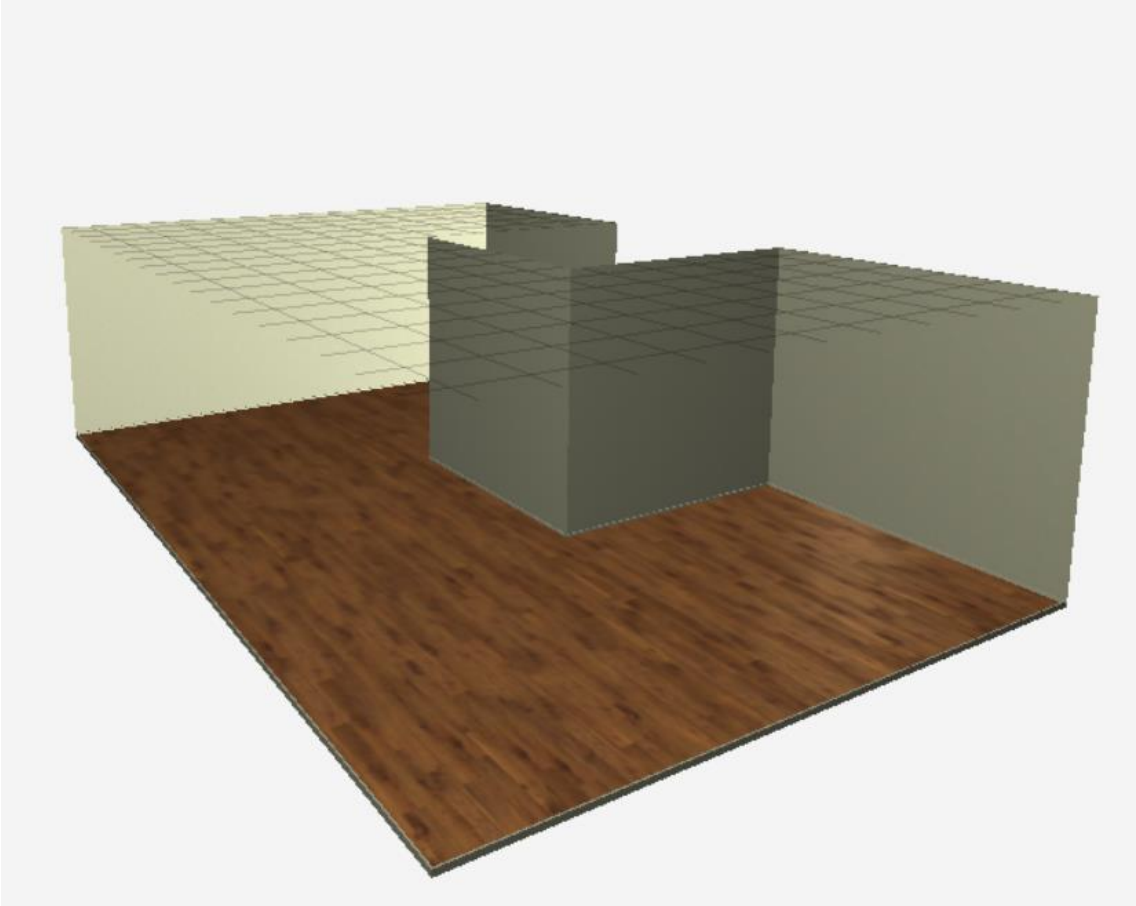
4.6.2 Snap points

The basic idea behind the snap points is to gather and array of possible points to snap to. After getting the possible snap points, the application tests whether any of the points are close enough to snap. If there are multiple snap points, the snapping is done to the closest one. The snap point checking is executed in the draw loop, so that they are updated as the user draws. At first the snap point update function checks few basic conditions for drawing a snap line such as snapping to previous vertice x or z positions and the room starting point. Next the angle based snaps are tested. This starts by calculating the angle to the last drawn line. If the result is 0, 45, 90 or 135,the function

continues to check whether the angle is positive or negative. The snap point is calculated and added to an array of snap points. The snap point array is returned to the drawing function which decides whether any of the snap points is close enough to snap.

4.7 Room generation

Once user has finished finished drawing the room, by entering a point close enough to the starting point, the 3D model of the room is generated automatically. A vertical plane is generated between each point in the vertice array. A plane requires four vertice to be drawn, so the vertice in the array can be thought as the floor level vertice and two ceiling level vertice needed to be calculated based on the positions of the floor vertice. Then a plane is drawn between these four vertice, and this is done to each vertex in the array. These planes form the walls of the room. The floor and the ceiling is generated as a shape between the floor level vertice. A very important thing when generating the room was to only show the inner side of the plane. This means that when observing the outer side of the plane, it would appear invisible. This way user can see inside to the room, and only the opposing walls will be visible.



Picture 7. The generated room with only the opposite walls visible.

5 RESULTS

The main objective of this thesis was to create a prototype of a room designer web application. In addition Three JSs' feasibility for this kind of project is evaluated. To back this, the thesis reviewed the main features of Three JS and introduced some it's competitors and compared Three JS to them.

5.1 Requirements

Requirements for the prototype were the ability to easily draw outlines of a room. The draw feature was required to draw straight lines with snap points at defined angles and relative coordinates. After this the application needed to generate a 3D model of the drawn room. The room was required to be observable and rotable by mouse. The application was expected to able to add air condition devices to the ceiling and the user to be able to move them around. The devices needed to be inserted into a grid, and as such the devices needed to move in a grid-like fashion. In addition a simple simulation of air condition was demanded. To accurately simulate the air movement would have been a very demanding task, so just a simple visual effect to show how it could look like was enough for the proof of concept phase.

5.2 ThreeJS evaluation

Most of the project was executed within only few weeks so Three JSs' ability to produce results quickly were put to test. The framework proved to work really well in this kind of prototyping. While there are a lot less discussion and examples found in the internet, compared to Unity for example, the ThreeJS website provides a good library of examples of most of the features. What makes them a bit less useful is the fact that most of the code in the examples aren't commented at all. Nevertheless with the ability to import external 3D-models, the first observable room was put up very quickly. Features followed

each other in relatively quick succession which made the customer of the project very pleased.

5.3 Project success evaluation

Overall the project can be viewed to be a very successful. The project had a very tight schedule which was met as all the required features were implemented by the deadline. The code itself was not very good in terms of good practices or structuring, but this was a known drawback we had to do match the development time. Nevertheless the application still contains bits and pieces like some algorhitms and functions that could be used for the actual project. The foundation and the architecture of the application however would have to be remade properly and some web frameworks could be introduced to make the application more sophisticated.

5.4 Future development

The project leaves a lot room for future development. At the beginning of the project we had a big list of possible features out of which we chose the most important ones that were required for the minimum viable product. The prototype only supported the insertion of air terminal devices and air flow simulation. Many more product types could be supported to help support room design. For example the next added product type would have probably been the radiators. The user interface was very crude at this point. We had good design plan for it but never had the time to carry it out.

REFERENCES

Unity. 2016 Getting started with WebGL development. https://docs.unity3d.com/Manual/webgl-gettingstarted.html. Read 23.4.2017.

Cabello, R. 2012. ThreeJS history. https://github.com/mrdoob/three.js/issues/1960. Read 16.9.2017.

Hewitson, J. 2013. Three.js and Babylon.js: a Comparison of WebGL Frameworks. https://www.sitepoint.com/three-js-babylon-js-comparison-webgl-frameworks/ . Read 17.4.2017.

Slick, J. 2017. What is Rendering?. https://www.lifewire.com/what-is-rendering-1954. Read 29.3.2017.

W3 Schools. 2016. Differences between SVG and Canvas. http://www.w3schools.com/html/html5_svg.asp. Read 29.10.2016

Cabello, R. 2017. Three.js Features. https://github.com/mrdoob/three.js/wiki/Features. Read 29.10.2016

Habrador. 2015. Improving Unity's physics engine PhysX to achieve higher accuracy. http://blog.habrador.com/2015/09/improving-unitys-physics-engine-physx.html Read 29.10.2016

Petitcolas, J. 2015. Importing a Modeled Mesh From Blender to Three.js. https://www.jonathan-petitcolas.com/2015/07/27/importing-blender-modelized-mesh-in-threejs.html. Read 29.10.2016.

Dirksen, J. 2013. Learning Three.js: The JavaScript 3D Library for WebGL. Birmingham: Packt Publishing.

Vapamedia. 2016. Selaamisen 'uusi' ulottuvuus: WebGL. http://www.vapamedia.fi/artikkeli/selaamisen-uusi-ulottuvuus-webgl/. Read 29.10.2016

Jarvis, M. 2015. Unity 5.3 makes WebGL support official. http://www.develop-online.net/news/unity-5-3-makes-webgl-support-official/0214565. Read. 26.1.2017

BabylonJS. 2016. Physics Engine – Basic Usage. https://doc.babylonjs.com/overviews/using_the_physics_engine. Read 26.1.2017

Weber, R. 2015. Game Development – Babylon.js: Building a Basic Game for the Web. https://msdn.microsoft.com/en-us/magazine/mt595753.aspx. Read 26.1.2017

Sloka-Frey, K. 2013. Let's Build a 3D Graphics Engine: Points, Vectors, and Basic Concepts. https://gamedevelopment.tutsplus.com/tutorials/lets-build-a-3d-graphics-engine-points-vectors-and-basic-concepts--gamedev-8143

Pettit, N. 2013. Beginner's Guide to three.js. http://blog.teamtreehouse.com/the-beginners-guide-to-three-js. Read 10.12.2017

Three JS, 2017. Spotlight. https://threejs.org/docs/#api/lights/SpotLight. Read 10.12.2017