An optimal solution for implementing a specific 3D web application

Mathias Nordin
Abstract

An optimal solution for implementing a specific 3D web application

Mathias Nordin

WebGL equips web browsers with the ability to access graphic cards for extra processing power. WebGL uses GLSL ES to communicate with graphics cards, which uses different instructions compared with common web development languages. In order to simplify the development process there are JavaScript libraries handles the communication with WebGL. On the Khronos website there is a listing of 35 different JavaScript libraries that access WebGL. It is time consuming for developers to compare the benefits and disadvantages of all these libraries to find the best WebGL library for their need. This thesis sets up requirements of a specific WebGL application and investigates which libraries that are best for implementing its requirements. The procedure is done in different steps. Firstly is the requirements for the 3D web application defined. Then are all the libraries analyzed and mapped against these requirements. The two libraries that best fulfilled the requirements is Three.js with Physi.js and Babylon.js. The libraries is used in two seperate implementations of the initial game. Three.js with Physi.js is the best libraries for implementig the requirements of the game. A performance test showed that Babylon.js is better then Three.js with Physi.js at rendering an envirionemnt with bouncing spheres.
## Contents

1 Introduction ................................. 5
   1.1 Questions at issue ................................ 5
   1.2 Limitations .................................. 6
   1.3 Procedure .................................. 6

2 Background .................................. 6
   2.1 Software Requirements ......................... 6
      2.1.1 Requirements in the product life cycle ........... 7
      2.1.2 Elicitation ................................ 7
   2.2 Graphics on the web ............................ 7
   2.3 HTML5 ........................................ 8
   2.4 JavaScript .................................... 8
   2.5 WebGL ........................................ 9
   2.6 Code example .................................. 9
   2.7 Rendering pipeline ............................. 15
   2.8 Framebuffer ................................... 16
   2.9 Working with 3D rendering ...................... 16
   2.10 3D rendering libraries ......................... 17
      2.10.1 Three.js .................................. 17
      2.10.2 Babylon.js ................................ 22
      2.10.3 Enchant.js ................................ 23
      2.10.4 CopperLicht ................................ 26
      2.10.5 Turbulenz ................................ 26
      2.10.6 FRAK ...................................... 26
      2.10.7 CubicVR ................................... 27
      2.10.8 SpiderGL ................................... 27
      2.10.9 KickJS ...................................... 27
      2.10.10 PhiloGL ................................... 27
      2.10.11 StormengineC ................................ 27
   2.11 Physics Engine ................................ 27
      2.11.1 Objects .................................... 28
      2.11.2 Contact resolution ......................... 28
      2.11.3 Impulses or Forces .......................... 28
   2.12 JavaScript physics libraries ................. 28
      2.12.1 Ammo.js ................................... 28
      2.12.2 Physi.js .................................... 28
      2.12.3 Bulletjs .................................... 29
      2.12.4 CANNON.js ................................ 29
      2.12.5 JigLibJS ................................... 29

3 Method ...................................... 30
   3.1 Requirements .................................. 30
   3.2 Computer game requirements .................... 30
      3.2.1 Software library requirements .................. 31
      3.2.2 None-functional requirements .................. 32
   3.3 Performance test ................................ 32
   3.4 Development process ............................ 32
   3.5 Correctness ................................... 32
3.6 Selecting libraries ........................................... 32

4 Results ......................................................... 33
  4.1 Libraries which did not qualify .............................. 34
    4.1.1 Enchant.js ........................................... 34
    4.1.2 Copperlicht ........................................... 34
    4.1.3 Turbulenz ............................................ 34
    4.1.4 FRAK .................................................. 34
    4.1.5 CubicVR .............................................. 35
    4.1.6 SpiderGL .............................................. 35
    4.1.7 KickJS .................................................. 35
    4.1.8 PhiloGL ................................................. 35
    4.1.9 StormengineC .......................................... 35
  4.2 Selected libraries ........................................... 35
    4.2.1 Three.js .............................................. 35
    4.2.2 Babylon.js ............................................ 36
  4.3 Development process .......................................... 36
    4.3.1 Three.js with Physi.js ................................. 36
    4.3.2 Babylon.js ............................................ 36
  4.4 Evaluating the games .......................................... 37
    4.4.1 Three.js and Physi.js .................................. 38
    4.4.2 Babylon.js ............................................ 38
  4.5 Performance test ............................................ 38

5 Discussion ..................................................... 41
  5.1 The Performance test ....................................... 42
  5.2 The development process ..................................... 43
    5.2.1 Babylon.js ............................................ 43
    5.2.2 Three.js with Physi.js ................................. 43
  5.3 Criticism .................................................... 43
  5.4 3D web applications in the future .......................... 44

6 Conclusion .................................................... 44

7 Future research ................................................ 45
1 Introduction

The Hyper Text Markup Language (HTML) standard was created by Tim Berners-Lee in 1990 and is similar to ENQUIRE a system he had created to share documents within CERN [1]. HTML is a specification of a document that is sent to other web browsers that translates the tags into a human readable format. A HTML document contains tags that explains its structure [2]. Later he purposed to build a network of these documents that is hosted on servers. He named it the World Wide Web. HTML is still the fundamental document of websites, but new functionalities has appeared that extends the web with new functionalities. Initially the web could only show different kinds of text and links, but with more styling specified in CSS and dynamic behavior enabled through JavaScript, the web become more dynamic and enjoyable. Previously advanced graphics and real time rendering was not suitable for the HTML standard since the web browsers had no access to graphics cards and the performance of JavaScript was not sufficient for advanced graphics. In order to extend web browsers with processing power were they equipped with rendering plug-ins like Flash. [10]. Flash was the first plug-in for graphics released in 1996 and it is still the most used graphics plug-in for the web. Safari released a new HTML element named canvas that enables rendering in web browsers without Flash [3]. In 2008 Google released a new JavaScript engine that improved the performance of the Google Chrome web browser [4]. WebGL was released in 2011 which gave the web browsers the opportunity to access the graphics card, enabling 3D graphics for the web [5]. With the release of Canvas, V8 and WebGL the need of plug-ins has decreased.

There might be problems for web developers to understand these new technologies since WebGL is a shading language that is very different from common web development. Shaders are small programs that are executed in one of the many processors on a graphics card. Programming shaders is very different from web development, resulting in a steep learning curve for web developers. To simplify the development process there has emerged a selection of JavaScript libraries where no knowledge of shaders is needed. The Khronos group is the creators of WebGL and they currently lists 35 different JavaScript libraries on their wiki-website that uses WebGL [6]. This thesis evaluates many of the different libraries and through continuous neglecting the less promising libraries is a final solution presupposed for a specific 3D web application.

1.1 Questions at issue

These are the research questions the thesis will try to solve.

1 Which are the most promising open source WebGL libraries for creating 3D web applications that includes game physics? The question is intentionally a wide unclear question. In order to answer the question a definition of a good library, and the definition of a simple 3D web application has to be defined. The answer of this question will help developers choosing libraries when developing 3D web applications, which will save time and probably increase the quality of the final application.

2 What are the differences between the available WebGL 3D libraries? By comparing the differences between the libraries a general overview of each library will be resolved. It will be interesting to see how diverse the different libraries are; if they have their own fields of application or competitive in the same applications. These overviews will be interesting for developers who wants a library for a specific cause.

3 What expertise are needed to develop 3D web applications? In order to develop a 3D web application you need a computer with a updated web browser, Internet and the knowledge of building 3D applications. Computers with a updated web browser is very common and also the Internet so
the bottleneck for building 3D web application is the knowledge of building 3D web applications. It is interesting to investigate the level of knowledge needed to develop a 3D web application.

1.2 Limitations

3D web applications appears in many different ways, from small visual 3D effects in website and impressive 3D worlds including impressive visual effects and physics environment. Defining software libraries for every different kind of 3D web applications is difficult and time consuming. In order to narrow the scope the thesis defines requirements of a specific 3D web application including game physics that will be a application to implement. It is simpler to find the right libraries to implement the application if the application is clearly defined. The most promising libraries to implement this application might be useless for other applications.

An optimal solution to find the best libraries would be to implement the application with every library. The optimal solution is very time consuming so instead will information be gathered of each library and by comparing the information will the most promising libraries be used to implement the 3D application. The information gathered from the analysis will be taken from the libraries official websites. If a library has more features and better development process then described on the website, that library will not be investigated by a implementation even if it is the best one. But the author assumes that the information on the websites is written in the most positive manner for that specific library describing its advantages and little or none about its disadvantages.

1.3 Procedure

The procedure is divided into different parts. The first part is called background which contains the fundamental knowledge that is needed too fully understand the subject. It contains the early history and development of the web, fundamental knowledge regarding computer graphics and game physics, a small overlook of different WebGL libraries currently available on the web and a explanation how they work. After the information has been gathered requirements are defined for a specific 3D web application that includes physics. The application is a 3D game that is simple but includes common elements in games. When the requirements of the game is defined they will be used to define the requirements for the libraries that will implement the game. Then will the Internet be searched for libraries that can fulfill these requirements. Those libraries that best fits the requirements will be used to implement the game. The selected libraries will also implement a performance test that will measure the performance of the library. Eventually the gathered data will be evaluated and discussed.

2 Background

The background section contains essential information that is needed in order too validate the best way of building 3D web applications without plug-ins. It contains information how to measure and analyze software and explain technical details of 3D applications.

2.1 Software Requirements

Software requirements are specifications of needs that stakeholders requires the software to solve. [7]. A knowledge of software requirements and 3D applications is essential to determine the most promising software libraries for 3D web applications. The fundamental principal of software requirements is to specify the systems output for each of the its input, which might sound like a simple matter, but systems is usually complex with many different input states and representations of data. Software requirements is divided into three different parts. One part is data requirements that handles the data and formats
of the system. Another part is functional requirements that specifies how the data is computed, transmitted, transformed and recorded. The last part is the quality requirements that is further divided into performance, usability and maintenance requirements. All of these aspects of requirements are essential in order to fully determine the quality of a software. As an example can a functional requirement be collision detection between 3D objects, data requirement can be the different colors of objects in a application and quality requirement can be a specific loading time of a application. With requirements like these is it possible to measure and differentiate libraries from each other. A disadvantage using software requirements is that it only measures the requirements already defined, which makes it very important that the requirements have a great coverage to define a good software product.

### 2.1.1 Requirements in the product life cycle

Usually is requirements defined in the initial part of a project, in a process called project inspection where a project team, business goals and project scope are formed [8]. The second phase is called elicitation where business goals and requirements is formed. In the third phase is the requirements documented in a way so developers can implement them easily. The fourth phase is dedicated to feedback where the requirements is validated with customers and stakeholders. The requirements are then used to make a Request For Proposal (RFP), which describes the system that should be developed. Companies can give a proposal to match the RFP and the customer can choose the best proposal of the given RFP. When the customer has chosen a RFP the design and programming can begin. The final part consist of acceptance tests, continuous requirements management and finally the release planning.

### 2.1.2 Elicitation

Elicitation is the process of finding and formulating requirements [9]. It is a crucial part of software development and probably the most challenging. There are many different elicitation techniques that are commonly used like, Stakeholder analysis, interviews, brainstorming, prototyping, risk analysis, goal-domain analysis, similar solutions and cost/benefit diagrams. Each elicitation technique has its own strengths and weaknesses, meaning that none of them are explicit better then any other. The most essential part of elicitation is to have a good knowledge of the stakeholders and the following five questions should be answered. Who are the stakeholders? What goals do they see for the system? Why would they like to contribute? What risks and costs do they see? What kind of solutions and suppliers do they see? It is very important that the stakeholders has good knowledge in the domain and knowledge of a good solution.

### 2.2 Graphics on the web

Initially, Flash was a plug-in to the Netscape web browser, which was created by a American company named FutureWave. [10] It was originally called FutureSplash and accessed data sent with the html document to render simple graphics. In 1996 Macromedia bought FutureWave Software and Future Splash Animator became Macromedia Flash 1.0. Today Flash is one of the largest gaming platforms in the world including games like Farmville with over 80 million users. Flash has dominated the market as a graphics engine on the web in resent years, but new competitors has occurred which have caused Flash to drop in popularity [11]. Stage3D is the API that makes it possible for Flash to access the graphics card. [12]

The Unity Web Player is another graphics plug-in which has been installed on over 225 million machines around the world. It is created by Unity Technologies which is a company that started as a OS X supported game development tool in 2005, but today the company creates games for many platforms including iOS, Android, Windows Phone 8, BlackBerry 10, Windows, Windows Store Apps, Mac, Linux, Web players, Flash, Playstation 3, Xbox 360 and Nintendo Wii U [13]. Unity technologies provide two
different development licenses, one is named Unity which is free and another is Unity Pro which costs 1500 US $ or 75 US $ per month. The pro version contains more features like 3D textures, LOD support that increase performance by focusing objects near the camera, profiling that analyze the time spent on rendering, physics and animations. They both provide a integrated editor which decreases developing time and simplifies the process of creating games. [14]

2.3 HTML5

HTML5 is a standard that includes many functionalities, and was fully developed in October 2014. The new features makes the web more dynamic and provides faster connections between servers and clients. There are aspects that is required, and simplifies 3D games which is provided by HTML5 like the canvas tag, which is a part of the web page where graphics is rendered. Web sockets is another part that is included in the HTML5 standard and allows full duplex communication between a server and a browser, which makes the web faster. The last two features of HTML5 is the audio tag which makes it possible to play audio on websites and lastly web workers makes it possible to execute JavaScript in parallel, that increases performance of the web. [17]

2.4 JavaScript

JavaScript was released with the NetScape web browser in 1996 which enabled computing at the client side of the web. [21] This was a breakthrough since the web was very slow, and validating data before sending it to servers saved a lot of time. New technologies were introduced like the DOM (Document Object Module) which structured the document and made it reconfigurable by JavaScript. The amount of JavaScript on websites grow steady, but there were little done in the web browsers to handle all this code which resulted in slow web experiences. JavaScript is a interpreted programming language which makes it very flexible, but compiling programming languages has the benefit of optimizing the language before executing it. This issue was handled in 2008 when Google released their Chrome web browser which optimized the JavaScript code and then translated it directly into machine code. The other web browsers followed their lead and soon had all popular web browsers interpretors that optimized JavaScript.

2.5 WebGL

The Khronos group is a consortium currently consisting of over 100 members that are focused on the creation of royalty-free open standards for parallel computing, graphics and dynamic media on a wide variety of platforms and devices [18]. All members have the opportunity of contributing to the standard, and when the development has proceeded the members votes if it is time for a new release. As a member you receive early access to features that are being developed, which thereby pays of as a technical advantage to other companies [19]. OpenGL is the worlds most adopted 2D and 3D graphics API in the industry and is created by the Khronos group. OpenGL—ES is a API platform that contains parts of OpenGL but are made for embedded systems, and is tuned for the safety critical market. WebGL is a royalty-free graphic card API that is built on OpenGL—ES and makes it possible for browsers to render graphics in the HTML5 canvas element. All web major web browsers supports WebGL.

The fundamental part of all OpenGL graphics standards is the shader which is defined in the websites JavaScript section. [20]. WebGL uses a shader language called GLSL ES which is the same language used by OpenGL ES. It contains of two shaders, a vertex shader and a fragment shader. A vertex shader is a program that describes the size and position of a specific vertex, and a fragment shader is a program which handles per-fragment operations like colors. A vertex is a position in the scene that defines a object. Below is a vertex shader written in JavaScript and GLSL ES.
The vertex shader is stored in a JavaScript variable named VSHADER_SOURCE and contains a string and \n tags. The strings are the GLSL ES code and the \n are new line tags in JavaScript. The new line tags do not have any effect on the program but is useful in error handling. The gl_Position specifies the position in the input space where a vertex shall be placed. In this case it is in the middle of the screen position (0.0, 0.0, 0.0). The position is specified by a vec4 vector which contains of four floats where the first three is the x, y and z values. The last float is a homogeneous coordinate which is used to translate the position of the vertex. All the x, y and z positions are divided by the homogeneous float, in this example the positions will be the following x=0.0/1.0, y=0.0/1.0, z=0.0/1.0. GL_PointSize contains the size of the vertex which in this case is 10 pixels wide. Below is an example of a Fragment shader program.

The Fragment shader program except that the code is stored in a variable named FSHADER_SOURCE and it contains a vector named gl_FragColor that holds color values (Red,Green,Blue,Alpha). Alpha is the intensity of the color, so a value of 1.0 in alpha is opaque and 0.0 is transparent.

### 2.6 Code example

This section contains a code example that explains the interaction between WebGL HTML, JavaScript and the user. The example has been taken from WebGL Programming Guide, written by Kouichi Matsuda and Rodger Lea. It writes red squares on the canvas where the user clicks, as can be seen in Figure 1.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Draw a point with a mouse click</title>
</head>
<body onload="main()">
  <script src="../lib/webgl-utils.js"></script>
  <script src="../lib/webgl-debug.js"></script>
  <script src="../lib/cuon-utils.js"></script>
  <script src="ClickedPoints.js"></script>
</body>
</html>
```

This is a simple HTML page. It contains of tags which describes different parts of the page.

```html
<!DOCTYPE html>

The doctype tag seen above, is a tag that should be in all web pages and it tells the browser which html version that will be used.
These are standard elements in html. The html tag covers the html document and everything inside those tags will define the web page. It has a attribute named lang which is set to English. This specify the language that will be used, and is useful since different countries has different characters. The head element contains information that will not be displayed in the web page but could be seen as setting that has to be set before the web page can start displaying its information. The title and other meta data is defined in the Head. The body element contains all the content of the web page.

The meta tag defines which encoding the browsers should encode the characters in. The title tag defines
the title of the web page that will be displayed in the browsers label section.

Please use a browser that supports "canvas"
</canvas>

The canvas element is the part of the html page where the rendering will take place. Its id is set to WebGL so JavaScript can access it in order render the graphics. The canvas element also has a width and height attribute which defines the width and height of the canvas in pixels, in this example the width is 400 and the height is 400.

These statements imports JavaScript and provides basic functionality to access WebGL. It provides functions to send information to the graphics card and functionalities to get feedback if anything goes wrong.

This JavaScript file contains all the code that is specifically written for this website. It will use the imported JavaScript libraries and the canvas element declared in the HTML file to print out squares where the user clicks in the canvas element. The source code of ClickedPoints.js can be found below and everything is explained in detail.

// ClickedPoints.js (c) 2012 matsuda
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;
  void main() {
    gl_Position = a_Position;
    gl_PointSize = 10.0;
  };

// Fragment shader program
var FSHADER_SOURCE =
  'void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
  };

function main() {
  // Retrieve <canvas> element
  var canvas = document.getElementById('webgl');

  // Get the rendering context for WebGL
  var gl = getWebGLContext(canvas);
  if (!gl) {
    console.log('Failed to get the rendering context for WebGL');
    return;
  }
// Initialize shaders
if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
  console.log('Failed to initialize shaders.');
  return;
}

// // Get the storage location of a_Position
var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
if (a_Position < 0) {
  console.log('Failed to get the storage location of a_Position');
  return;
}

// Register function (event handler) to be called on a mouse press
canvas.onmousedown = function(ev){ click(ev, gl, canvas, a_Position); };

// Specify the color for clearing <canvas>
gl.clearColor(0.0, 0.0, 0.0, 1.0);

// Clear <canvas>
gl.clear(gl.COLOR_BUFFER_BIT);

var g_points = []; // The array for the position of a mouse press
function click(ev, gl, canvas, a_Position) {
  var x = ev.clientX; // x coordinate of a mouse pointer
  var y = ev.clientY; // y coordinate of a mouse pointer
  var rect = ev.target.getBoundingClientRect();

  x = ((x - rect.left) - canvas.width/2)/(canvas.width/2);
  y = (canvas.height/2 - (y - rect.top))/(canvas.height/2);
  // Store the coordinates to g_points array
  g_points.push(x); g_points.push(y);

  // Clear <canvas>
gl.clear(gl.COLOR_BUFFER_BIT);

  var len = g_points.length;
  for(var i = 0; i < len; i += 2) {
    // Pass the position of a point to a_Position variable
    gl.vertexAttrib3f(a_Position, g_points[i], g_points[i+1], 0.0);

    // Draw
    gl.drawArrays(gl.POINTS, 0, 1);
  }
}

Below are the fragment shader and vertex shader which is the core parts in WebGL.

// Vertex shader program
var VSHADER_SOURCE =
'attribute vec4 a_Position;
' +
'void main() {
' +
'  gl_Position = a_Position;
' +
'  gl_PointSize = 10.0;
' +
'}
';

// Fragment shader program
var FSHADER_SOURCE =
'void main() {
' +
'  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
' +
'}
';

The shaders are very similar to the shaders described in the previous WebGL chapter. In order to retrieve a good explanation of the shaders refer to the WebGL chapter on page 8. The only difference between the shaders on page 3 and these is that these vertex shaders contains a attribute named a_Position and it has vec4 as a type. By declaring this variable and than set the value to the position of the vertex we only have to give that variable a value instead of creating a whole new shader program when we want to write a new vertex.

<body onload="main()">

The main function is called when the html page has been fully loaded. This is done by the onload parameter that is set in the body element.

function main() {
// Retrieve <canvas> element
var canvas = document.getElementById('webgl');

// Get the rendering context for WebGL
var gl = getWebGLContext(canvas);
if (!gl) {
  console.log('Failed to get the rendering context for WebGL');
  return;
}

The main function starts by retrieving the canvas element and sets it to a WebGL context, which makes it possible for WebGL to render graphics in the canvas element. The context is saved in a variable named gl, which will be called when new graphics is rendered on the canvas. The if statements makes sure it is possible to access both WebGL and canvas, which is not possible for older web browsers.

// Initialize shaders
if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
  console.log('Failed to initialize shaders. ');
  return;
}

The shaders are initialized and a error message is printed in the console if anything goes wrong. The initialization sends the shaders to the WebGL system, and specifies the context they will be executed in.

var a_Position = gl.getAttribLocation(gl.program, 'a_Position');

This code retrieves the mouse x and y coordinates by calling gl.getAttribLocation function.
if (a_Position < 0) {
    console.log('Failed to get the storage location of a_Position');
    return;
}

If the coordinates is not retrieved, a error message is printed.

canvas.onmousedown = function(ev) { click(ev, gl, canvas, a_Position );};

A listener is set to detect if the left mouse key is pressed within the canvas and calls a function named click when this happens.

gl.clearColor(0.0, 0.0, 0.0, 1.0);

Before writing to the canvas the screen has to be cleared. The color which will clear the canvas is set by the glClearColor method and passing in a color with none red, none green, none blue values and fully opaque which results in a black color. The gl.clear(gl.COLOR_BUFFER_BIT) method clears the canvas with the color defined in the gl.clearColor method. Lastly the click function is defined, which paints a red square at the clicked position.

var g_points = [];
function click(ev, gl, canvas, a_Position) {
var x = ev.clientX;
var y = ev.clientY;
var rect = ev.target.getBoundingClientRect();
x = ((x - rect.left) - canvas.width/2)/(canvas.width/2);
y = (canvas.height/2 - (y - rect.top))/(canvas.height/2);
// Store the coordinates to g_points array
    g_points.push(x); g_points.push(y);
}

All the places where the user clicks has to be stored, since the canvas will be cleared each rendering cycle. The x and y values from the mouse click are stored in the g_points array. The mouse clicks x and y position is obtained by ev.clientX and ev.clientY. However canvas and WebGL are using different coordinate systems. Canvas positions x=0, y=0 on the top left corner of the canvas but WebGL positions x=0, y=0 in the middle of the canvas. In order to display a square at the position where a user clicks it has to be translated from the canvas coordinate system to the WebGL coordinate system. The x, y values are translated from the canvas positioning system to WebGL positioning system and stored in variables named x and y. Finally the translated x and y positions are stored in the g_points array with the g_points.push method. All the positions where a user clicks are now stored in a array. The final task is to write them on the canvas.

// Clear <canvas>

// Draw

The canvas has to be cleared before any rendering which is done by the `gl.clear(gl.COLOR_BUFFER_BIT)` method. The x and y positions of the squares are sent to WebGL by calling `gl.vertexAttrib3f` for every pair of x and y value in the `a_Position` array. Finally `gl.drawArrays(gl.POINTS, 0, 1)` is called and it render the points on the canvas.

This example clearly shows the integration between JavaScript, HTML and GLSL ES, but the code developed in this thesis will be simpler using other libraries, not requiring as many functions or any shaders.

### 2.7 Rendering pipeline

The rendering pipeline is a very important part of computer graphics, having a huge impact on the rendering performance [22]. The pipeline is divided into different stages which starts when the programmers instructions is entered and finished when the rendered figure is sent to the screen. Like all pipelines, the speed of the pipeline is determined by the slowest stage in the pipeline, thereby it is useful to divide the pipeline in many stages, which are fast instead of doing everything in one place making a big part of the pipeling idle.

**Application stage** The application stage is connected with the program the programmer has written. The application stage is totally handled by the CPU and it usually handles collision detection, animation, user input and other parts of the program which is not connected with rendering the objects. The main purpose of the application stage is setting up the geometry of the objects that later will be fed to the geometry stage. Optimizations in the application stage can effect the execution in the subsequent stages. This is the stage where the programmer has most freedom, and responsibilities.

**Geometry stage** The geometry stage handles what, where and how the different kind of objects should be drawn. These calculations are usually executed at the GPU. It is usually divided in to model, view transforms and vertex shading. Model and view transforms takes the geometries and the scene from the application stage and links them together relatively to the view of the camera. Vertex shading gives the objects their material and specifies how that specific material should be display according to the camera view and the lights.

**Projection** The projection stage handles how the form of the objects should be displayed. This depends on what kind of projection that is used. One kind of projection is a orthographic projection and keeps the objects parallel. Another projection that is commonly used is projection which resizes the objects depending how far or near the camera is to the objects, resulting in objects that will converge to a small dot in the middle or the screen if the camera is too far away from the objects.

**Clipping** Clipping is a optimization technique that only passes the objects or part of objects that is in the field of view of the camera to the rasterizer stage. If only half the circle is in the field of view, only that half will be sent to the next stage, which saves the graphics card from unnecessary calculations.

**Screen Mapping** Screen mapping customize the 3D world so it fits the screen that will display the scene. In other words are the x, y and z coordinates in the scene translated to the screens x and y coordinates. The translation is different if the 3D world is translated to ether OpenGL or DirectX, since the x and y positions is located differently in OpenGL compared to DirectX.
**Rasterizer stage** The rasterizer stage uses the data from application and geometry to render a image and it is completely handled by the GPU. It is divided into two stages named triangle setup and triangle traversal. The triangle setup calculates the differentials of the surface that are covered by the triangles vertices’s. The result will later be interpolated with the shading data from the geometry stage. Triangle traversal gives the pixels inside a triangle its fragment that is calculated by the properties stored in the triangles vertices's.

**Pixel shading** The pixel shading stage calculates the shading for each pixel. The part of the GPUs performing pixel shading is programmable, so there are different techniques for pixel shading.

**Merging** Merging combines the shades from the pixel shading stage and the Red, Green, Blue color values stored in the color buffer. Merging also decides what is visible in the scene, since it is possible for different objects to lay on top of each other hiding objects that is behind. This increase performance and is usually calculated with a algorithm called Z-buffer.

### 2.8 Framebuffer

When the graphics has traversed the whole rendering pipeline it has reached the framebuffer. The framebuffer is a memory that are very near the processing units of the GPU. The framebuffer usually exists of three different sub-buffers.

**Color Buffer** The color buffer is a buffer that consist of a vector of RGB (Red, Green, Blue) or RGBA (Red, Green, Blue, Alpha) values. It is the value that has been set in the WebGL examples. The depth of the framebuffer is the number of bits that represents one pixel on the screen. The depth can be 16, 24 or 32 bits. A common standard is RGB565 where 5 bits represents red, 6 bits represents green and 5 bits represents blue. This standard are used in older cellphones and the total number of colors is \(2^{16} = 65,536\). A 24-bit frame buffer has its colors divided in RGB888 and 32-bit frame buffer has its colors also divided in RGB888 but additional 8 bit for the alpha value. The alpha value describes the transparency of the color.

**Z-Buffer** The Z-Buffer has the same amount of elements as there are pixels in the color buffer, and each element contains the distance from the viewer. The result is a buffer containing all the pixels that are closest to the viewer in the scene.

**Stencil Buffer** The stencil buffer is used to compute where in the color buffer something should be drawn. This is usually used when rendering shadows.

### 2.9 Working with 3D rendering

There are two kinds of rendering, real-time rendering and non real-time rendering. A good example of real-time rendering is computer games where the next shown 3D objects are determined by the input from the user. Real-time rendering has to be very fast, since a user wants instant feedback when input has been entered and waiting for the rendering to be finished is not acceptable. Non real-time rendering refers to rendering which do not has to be fast, the focus is in the quality of the result. Below is the most central functionalities working with 3D libraries, the information has been retrieved through experience working with different 3D libraries.
The scene A scene is the coordinate system that contains all the other different kinds of objects. It consists of a three dimensional area where the different objects is positioned. The area consists of a x value which describes the horizontal position, a y value which describes the vertical position, and a z value which describes the deep position. The scene handles the interaction between the different objects like cameras, lights and 3D objects.

3D objects A 3D object consists of many lines which builds the shape of a object. In the ends of each line there is a vertex. A vertex has a position in a input space defined by a x, y and z value. By combining many lines a grid of an object will be formed. The lines are usually dived into three vertices, which form triangles and the vertices are stored in a buffer called vertex-buffer. To specify how the lines should be connected, another buffer is used called index-buffer. Lets say it is a vertex-buffer with the following inputs \((0.0, 1.0, 0.0,),(1.0,-1.0, 0.0),(1.0,-1.0,0.0,))\), and a index-buffer with the following input \((0, 1,2)\). Then a triangle will be created starting from position 0 in the VertexBuffer \((0.0, 1.0, 0.0)\) and drag a line to position 1 in the vertexBuffer \((-1.0, -1.0, 0.0,\))\), a line from position 1 to position 2 in the vertexBuffer \((1.0,-1.0,0.0,))\) and finally from position 2 back to position 0 in the vertexBuffer. In that way a triangle has been drawn on the screen.

Camera Cameras are essential when building a rendering scene. They determines how the objects should be presented to the user and by moving the cameras the view of the scene changes. This is done by a transformation matrix which changes the values in the vertexBuffer as moves in the scene.

Shaders and reflections Shaders are the methods which handles the lights and shades. A normal shading technique is ray tracing which is similar to the way shades are in the real world. By adding a light source, and direct it to a object you can calculate how shades and reflections should be spread relatively to the camera.

2.10 3D rendering libraries

This section explains the different libraries which is investigated in this thesis. The information is taken directly from the libraries official websites. This means that the libraries are described in a positive manner.

There are many different kinds of JavaScript libraries for game development available on the web. Many of the libraries handles 2D games which do not use WebGL and some of the WebGL libraries are used for other applications like data representation and calculations which is thereby not mentioned in this thesis.

Three popular libraries are Three.js, Babylon.js and Enchant.js. These libraries will contain code examples to show their structure. As can be seen is the code very similar for all three libraries.

2.10.1 Three.js

Three.js is a open source project and can be found on GitHub where it has been forked (Downloaded) more then 3500 times and more then 250 persons has contributed to the project, which is a lot for a WebGL library. \([24]\) It provides functionalities like scenes, lights, animations, shaders, materials etc. In order to get as many people as possible to use the library has it focused on making the library as simple as possible. Three.js does not only support HTML5 Canvas graphics but also Scalable Vector Graphics. Three.js is originally made by Ricardo Cabello, with a alias of Mr.Doob at Github. \([25]\).

Below is the source code of an application that renders a box with random color, angle and position. The application can be seen in figure 2.

Below is the code for the showcase.
<!DOCTYPE html>
<html lang="en">
<head>
    <title>cubeGenerator</title>
    <meta charset="utf-8">
</head>
<body>

<!-- Import Three.js-->
<script src="../Code/js/three.min.js"></script>

<script>
    var container, stats, 
        camera, scene, projector, renderer;

    function init() {
        //creates a div element and appends it to the document. 
        container = document.createElement('div');
        document.body.appendChild(container);

        //Creates a Perspective camera
        camera = new THREE.PerspectiveCamera(70, window.innerWidth / window.innerHeight, 1, 10000);
        camera.position.z = 1000;
        scene = new THREE.Scene();

        // Adds ambient light which makes the shadows not to dark.
        scene.add(new THREE.AmbientLight(0x505050));

        // Initialize a light scource.
        var light = new THREE.SpotLight(0xffffff, 1.5);
        light.position.set(0, 500, 2000);
        light.castShadow = true; // enable shadows

        // Settings set for the camera.
        light.shadowCameraNear = 200;
    }
</script>

Figure 2. A green box is rendered in Three.js
light.shadowCameraFar = camera.far;
light.shadowCameraFov = 50;

light.shadowBias = -0.00022;
light.shadowDarkness = 0.5;

light.shadowMapWidth = 2048;
light.shadowMapHeight = 2048;
// Add the light source to the scene.
scene.add( light );

// Add one cubes to the scene at random positions and rotations
var geometry = new THREE.CubeGeometry( 40, 40, 40 );
// creates a object with a THREE.MeshLambertMaterial
var object = new THREE.Mesh( geometry, new THREE.MeshLambertMaterial( { color: Math.random() } ) );

// Randoms the position, rotation and scale of the Mesh
object.material.ambient = object.material.color;
object.position.x = Math.random() * 1000 - 500;
object.position.y = Math.random() * 600 - 300;
object.position.z = Math.random() * 800 - 400;

object.rotation.x = Math.random() * 2 * Math.PI;
object.rotation.y = Math.random() * 2 * Math.PI;
object.rotation.z = Math.random() * 2 * Math.PI;

object.scale.x = Math.random() * 2 + 1;
object.scale.y = Math.random() * 2 + 1;
object.scale.z = Math.random() * 2 + 1;

scene.add( object );

// Initialize the projector
projector = new THREE.Projector();

// Renderer contains the canvas element.
renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setSize( window.innerWidth, window.innerHeight );
// add the canvas to the document
container.appendChild( renderer.domElement );

}

// The animate function is called at 60 frames per second or as fast as possible
function animate() {
    requestAnimationFrame( animate );
    render();
}
function render() {
    // Tells WebGL to render the scene
    renderer.render( scene, camera );
}

// When everything is loaded we can start the website
init();
animate();
</script>
</body>
</html>

This code will not be explained in detail since the main part of JavaScript has already been discussed in the code example at page 9, instead will the example focus on specific Threejs aspects. There is no GLSL ES in the code, everything is made in JavaScript relying on Three.js function calls.

canvas = document.createElement( 'div' );
document.body.appendChild( container );

A specific variable named container is used which contains a div element containing all the elements that will be used in the game. It will be added to the HTML document by the document.body.appendChild function. Having a local variable named container makes it easy to edit objects, changing their position, color etc.

camera = new THREE.PerspectiveCamera( 70, window.innerWidth / window.innerHeight, 1, 10000 );
camera.position.z = 1000;

The camera is of type perspectiveCamera which is a camera that makes the objects appear differently depending on which angle the object is viewf in, its depth position is changed to 1000. The camera is stored in a local variable named camera which makes it easily accessible.

scene = new THREE.Scene();
// Adds ambient light which makes the shadows not to dark.
scene.add( new THREE.AmbientLight( 0x505050 ) );

The scene is the main object containing all the objects that will be used in the application. An ambient light is a light that is similar to the sun lightening all the objects in a scene.

var light = new THREE.SpotLight( 0xffffff, 1.5);
light.position.set( 0, 500, 2000 );
light.castShadow = true; // enable shadows

// Settings set for the camera.
light.shadowCameraNear = 200;
light.shadowCameraFar = camera.far;
light.shadowCameraFov = 50;

light.shadowBias = -0.00022;
light.shadowDarkness = 0.5;

light.shadowMapWidth = 2048;
light.shadowMapHeight = 2048;
// Add the light source to the scene.
scene.add( light );
Above is the code for creating a spotlight which has same characteristics as a normal spotlight. Parameters for the spotlight is changed after it has been added, which change its behavior.

```javascript
var geometry = new THREE.CubeGeometry( 40, 40, 40 );
var object = new THREE.Mesh( geometry, new THREE.MeshLambertMaterial( { color: Math.random() * 0xffffff } ) );
object.material.ambient = object.material.color;
object.position.x = Math.random() * 1000 - 500;
object.position.y = Math.random() * 600 - 300;
object.position.z = Math.random() * 800 - 400;
object.rotation.x = Math.random() * 2 * Math.PI;
object.rotation.y = Math.random() * 2 * Math.PI;
object.rotation.z = Math.random() * 2 * Math.PI;
object.scale.x = Math.random() * 2 + 1;
object.scale.y = Math.random() * 2 + 1;
object.scale.z = Math.random() * 2 + 1;
scene.add( object );
```

This code creates a mesh objects that contains of a geometry and a material. The geometry is the shape of the object which in this example is a cube with sides of length 40, the material is MeshLambertMaterial with a random color. The objects position, rotation and scale are set by JavaScripts Math.random function and then the object is added to the scene.

```javascript
renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setClearColor( 0xf0f0f0 );
renderer.setSize( window.innerWidth, window.innerHeight );
container.appendChild( renderer.domElement );
```

When the scene is finished is a WebGLRenderer created and saved in a parameter named renderer. The renderer is a canvas element that will display the scene in the HTML document. The color of the renderer is set too f0f0f0 and its size is set to the inner height and width of the web browser. Other libraries evaluated in this thesis defines the canvas element in the HTML part and configure it in JavaScript. Finally is the renderer added to the container.

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

The animate function is continuously called by the requestAnimationFrame function which executes the animate function in 60 frames per seconds or as fast as the previous rendering is finished. A frame rate of 60 frames per second has been a standard for moving pictures in a long time since it gives a smoothly impression for the human brain and a higher value has been considered unnecessary. In this case the animate function only renders the box on the scene, without changing any properties, but a animation could be done by changing some of the box’s properties in the animation loop.
2.10.2 Babylon.js

Babylon.js was released in the summer of 2012 and is an open source JavaScript library made by Microsoft employees [26]. Babylon.js includes a physics engine which simplifies the integration with physics. There are many impressive examples available on the official website, showing the library's potential. Below is a description of some of its interesting features.

**Animation engine** There are two ways of doing animations in Babylon.js. They can be created by functions that is called in the render function similar to Three.js or they can use a built in animation function. The built in animation engine controls the animations by key values which the animation will be altered between. An example can be a box moving between two positions and it would be programmed as follow: frame:0,value:20frame:50,value:40frame:100,value:20. The frame value represents the progress of the animation, in this case the animation would start with a x value of 20 alters so it is at position 40 in the middle of the animation and finally reaches position 20 in the end of the animation.

**Video Texture** The video texture is a specific texture that operates like a video screen, that can be added to meshes. The texture provides basic video functionalities like play, pause and stop.

**Particle systems** Particles is as it sounds, small objects that appear on the canvas. They can be used to simulate explosions, sandstorms, magic spells and so on and is very configurable with 19 parameters including color, particleTexture, direction, speed, size, etc.

Like all other JavaScript libraries uses Babylon.js its canvas element to render graphics. Below is the initial code to access the canvas element to start render graphics in it.

```javascript
if (BABYLON.Engine.isSupported()) {
    var canvas = document.getElementById("renderCanvas");
    var engine = new BABYLON.Engine(canvas, true);
}
```

The BABYLON.Engine.isSupported() function returns true if it is possible to render 3D graphics in the current browser and returns false if it is not possible. The canvas element is already created in the html section with a class name of renderCanvas. It is accessed in JavaScript using the DOM element by calling document.getElementById("renderCanvas") and it is stored in a element named canvas. An Babylon engine is created that is the bridge between WebGL and JavaScript. It translates the rotations, objects, lights and everything else defined in JavaScript to GLSL ES functions. When the canvas and the engine is set it is time to start creating a scene and fill it with objects.

```javascript
var camera = new BABYLON.FreeCamera("Camera", new BABYLON.Vector3(0, 0, -10), scene);
var light0 = new BABYLON.PointLight("Omni0", new BABYLON.Vector3(0, 100, 100), scene);
var sphere = BABYLON.Mesh.CreateSphere("Sphere", 16, 3, scene);
```

The way of creating objects in Babylon.js is very similar to Three.js. The cameras lights and objects are stored in variables making them accessible. The first parameter in the functions is a name parameter, that makes the objects accessible when they are added to the scene. The second parameter is a BABYLON.Vector3 which contains the position in the scene defined by x, y and z coordinates. The last parameter is the scene they will be placed in. CreateSphere function creates a mesh in the shape of a sphere. The second parameter is the segments of the sphere and the third parameter is the diameter of the sphere. Below is the render loop of Babylon.js.

```javascript
var renderLoop = function () {
    engine.beginFrame();
    scene.render();
};
```
The render loop consist of four parts:

engine.beginFrame();

This function tells the engine that a new frame is going to be rendered.

scene.render();

Renders all the objects that are in the scene.

engine.endFrame();

Makes sure no more object is rendered and present the result on the canvas.

BABYLON.Tools.QueueNewFrame(renderLoop)

The BABYLON.Tools.QueueNewFrame adds a new frame to a queue containing the frames that will be rendered. The code above will display a sphere in the middle of the screen.

var alpha = 0;
sphere.scaling.x = 0.5;
sphere.scaling.z = 1.5;
scene.beforeRender = function() {
    sphere.rotation.x = alpha;
    sphere.rotation.y = alpha;
    alpha += 0.01;
};

In order to show some animations the sphere has to be changed in the rendering loop. This is done by changing the spheres rotation x and y values in the scenes beforeRender function. Running the code will make the sphere spin.

2.10.3 Enchant.js

Enchant.js is a popular JavaScript game engine library used to create 3D and 2D games [27]. At http://code.9leap.net has it been published over 1000 games created by Enchant.js and most of them is 2D games. It is possible to learn Enchant.js 2D games online at http://code.9leap.net.

A enchant.js game consists of three different kinds of objects, game object, scene objects and node object. The game object monitors the game, the system running it and the games basic functionality. A game object can consist of one or more scene objects. A scene object is a rendering area within a game object and can be a main scene where the user navigates a character, while another scene shows statistics about the character. Each scene contains different kinds of node objects that operates within the scene. There are different kinds of node objects available but examples can be images and text moving in the scene. There is no z-axis to be controlled in Enchant.js, meaning that it can only render 2D graphics, but there are other libraries which are used to extend Enchant.js with the possibilities to create 3D games. The libraries are listed below.

Gl.enchant.js Gl.enchant.js is the main library which extends enchant.js with 3D functionalities and WebGL support.
**glMatrix-1.3.7.min.js**  A open source library which provides mathematics, like matrices and vectors.

**collada.gl.enchant.js**  Allows COLLADA 3D models to be used in the scene.

**primitive.gl.enchant.js**  Provides a set of simple 3D-objects that can be used.

Below is a Enchant.js example game which displays a 3D Android character and the result can be seen in Figure 3.

```javascript
enchant();
var game;
window.onload = function(){
    game = new Core(320, 320);
    game.fps = 60;
    game.preload('images/droid.dae');

    game.onload = function(){
        scene = new Scene3D();
        scene.setDirectionalLight(new DirectionalLight());

        camera = new Camera3D();
        camera.x = -20;
        camera.y = 20;
```
camera.z = -10;
camera.centerZ = -9;
scene.setCamera(camera);

//Create all Droids
var Droid = enchant.gl.Sprite3D();
Droid.set(game.assets["images/droid.dae"]);
Droid.scaleX = 2;
Droid.scaleY = 2;
Droid.scaleZ = 3;
Droid.x = -0.75;
Droid.y = -1.5;
Droid.z = -9.2;
scene.addChild(Droid);

game.start();

As can be seen is the code very small and simple compared with the Babylon.js and Three.js code.

This is the main structure of all applications created in Enchant.js, it does not matter if it is a 3D or 2D game. The enchant call initialize the objects and variables of Enchant.js. When the html page has been loaded a game object is created with a width and height of 320 and the frames per second is set to 60. The game.preload(’images/droid.dae’) function starts loading a .dae file containing a 3D model which later will be used in the game. The onload() function contains the game behaviour and is called by game.start(). All the code below is inside the game.onload() function.

var scene = new Scene3D();
scene.setDirectionalLight(new DirectionalLight());

Initially the game needs to be set with a scene, camera, objects and lights. The scene is defined as a Scene3D object and the light of that scene is set as a DirectionalLight. Gl.enchant.js supports two kinds of lights, DirectionalLight and AmbientLight.

camera = new Camera3D();
camera.x = -20;
camera.y = 20;
camera.z = -10;
camera.centerZ = -9;
scene.setCamera(camera);

The camera is a Camera3D object and its position is set to x=-20, y=20 and z=-10. The centerZ attribute tells the camera where in which direction it will be facing, and scene.setCamera() adds the camera to the scene.

var Droid = enchant.gl.Sprite3D();
Droid.set(game.assets["images/droid.dae"]);
Droid.scaleX = 2;
Droid.scaleY = 2;
Droid.scaleZ = 3;
Droid.x = -0.75;
Droid.y = -1.5;
Droid.z = -9.2;
scene.appendChild(Droid);

A local variable is created containing a Sprite3D object which is a node in the scene. The Droid.set(game.assets["images/droid.dae"]) function assigns the preloaded droid.dae 3D model to the Droid variable. The position of the Sprite3D object is changed by setting its x, y and z values, the scale is set by changing its scaleX, scaleY and scaleZ values.

2.10.4 CopperLicht

CopperLicht is a open source JavaScript library distributed by a company named Ambiera located in Austria [28]. CopperLicht was initially a library used inhouse for CopperCube which is a software for creating games, but was later released open-source separated from the game engine. Developing games using CopperCube makes it possible to create 3D games without any knowledge of programming. There are three games presented on CopperLicht’s official website and of them are currently being produced. CopperLicht uses a small binary file that contains preloaded meshes which reduces the loading time of meshes, which is a bottleneck for all WebGL libraries.

2.10.5 Turbulenz

Turbulenz is a company which focus on creating games [29]. They provide a Game engine which is free of charge and a SDK which is open-source. It is possible to publish games produced on their website, and takes 30% of its revenue. The SDK contains features to create both 3D, 2D games and includes physics. There is currently eight games listed on their website https://ga.me one of them is a 3D game and the rest is a 2D game.

2.10.6 FRAK

FRAK engine was developed by 3D Technologies R&D for 3D Wayfinder, which is a 3D visualization and navigation map for big facilities like train stations and shopping malls [30]. The first public version of FRAK JavaScript 3D engine was released first of august 2013. FRAK has a forum which is filled with spam and there are no discussion regarding FRAK on the forum. The official website provides examples of FRAK rendering simple shapes and text.
2.10.7 CubicVR

CubicVR is an open-source project that is available on Github [31]. It has four main contributors to the project and it has been forked 70 times. The Github page includes many examples which show the potential of the library, including many kinds of meshes, collision detection, and animations. It does not contain any physics provided with the library but the official website provides examples integrated with Bullet physics engine.

2.10.8 SpiderGL

SpiderGL is a low-level rendering platform for 3D rendering, which inherit a structure similar to GLSL ES [32]. One advantage of SpiderGL is its loose structure that enforces many possibilities, which also is a disadvantage since the amount of code needed increases. SpiderGL focuses on enforcing the simplicity and functionalities of WebGL. It does not contain a scene graph and do not prevent access to the underlying WebGL graphics layer.

2.10.9 KickJS

KickJS is another WebGL library which is available through Github [33]. On the official Github page there are tutorials provided. All KickJS applications is made of one scene object, containing different kinds of game objects which contains different kinds of components. Each component contains a transform which can be pointed to other objects, in that way 3D objects can be created, a example can be the wheels of a car which will roll when the car object is moving.

2.10.10 PhiloGL

PhiloGL describes itself as a WebGL framework for Data Visualization, creative coding and game development [34]. There are 16 examples of the library on the website but none of them could be considered a game application. The code structure is not as tightly connected to WebGL as SpiderGL but still more similar to WebGL than Three.js. It says on the website that they try to keep it "to be as close to the gl calls as possible, providing a clear yet tight abstraction to WebGL". Thereby OpenGL knowledge is very helpful creating a game.

2.10.11 StormengineC

StormengineC is an open-source library available on Github [35]. It has only one contributor which continuously makes commits to the repository. It includes physics that is provided by JigLibJS2 which is a physics library exported to JavaScript and originally written in C++ where it is named JigLib. It contains interesting features like SSAO (Screen space ambient occlusion), 3D objects (COLLADA, .obj) and path tracing. None of the examples shows an application that is similar to a game.

2.11 Physics Engine

A physics engine simulates physics in a 3D or 2D world [36]. Before real physic engines existed, simulations were created that simulated physics with no real time interactions. As an example you can create a program that simulate the shot of a arrow hitting a target. In order to shoot different objects like rocks and shuttlecocks the simulation has to be remade from scratch. If a more dynamic game would be created, with many different kinds of objects which can interact with each other, the number of simulations needed would increase by the number objects in the game for every new object that is added to the game. No simulations is needed using a physics engine since each object is affected by the same amount of rules set by the environment. Each object contains parameters that can be customized giving them different
characteristics like weights, friction, and velocities.

A too advanced physics engine relative the software application will increase the loading time and decrease the performance of the web application. Ian Millington has described the balance well by saying "A good physics engine should balance the complexity of the programming with the sophistication of the effects you want to simulate".

2.11.1 Objects

There are two kinds of objects that can be used in a physics engine. One type is Rigid-body engine which treats an object as a whole. The other type is called Mass-aggregate engine which adds masses to parts of objects. By putting these masses on the corners of the body and then combining them with rods a physical body is created. Mass-aggregated physics engines are faster and rigid-body engines gives a more realistic impression.

2.11.2 Contact resolution

Contact resolution is the interaction between objects like collisions and connections. Contact resolution can be calculated using a iterative approach which handles the contacts one by one or by the ”Jacobian-based” approach where firstly all the contacts are calculated and then the forces are applied to all of the objects at the same time. The Jacobian- based approach is more realistic but also more time consuming. There is a final approach which has been proven to be very time consuming and not suitable for games.

2.11.3 Impulses or Forces

A physics engine can either be force driven, impulse driven, or both. The force driven engines is a good simulation of a real world and gives a more realistic impression then a impulse based engine, but the impulse based engine requires less computational power, has simpler mathematics, is easier to implement and more flexible then a force driven engine.

2.12 JavaScript physics libraries

Many JavaScript physics libraries that are available are used for simulations in 2D, but wont be mentioned in this thesis. Many libraries are exported from older libraries created in C or C++.

2.12.1 Ammo.js

Ammo.js is a physics engine and its acronym stands for Avoid Making My Own physics engine. It has been directly translated from Bullet Physics Engine which is a popular physics engine written in C [39]. Ammo.js is a popular physics engine for 3D graphics on the web.

2.12.2 Physi.js

Physi.js is a variety of Ammo.js that is specifically made for Three.js. Physi.js uses one web worker that makes it possible to run the physics calculations on different cores, which improves performance. Like most of the other projects it is available on Github where it has been forked over one hundred times. Thanks due the customization towards Three.js is only five different changes required in order to extend an Three.js application with physics. The five steps are described below.

<script src="physi.js"></script>
Add a script tag to load the library. It has to be loaded before the other four steps are executed.

```javascript
Physijs.scripts.worker = 'physijs_worker.js';
Physijs.scripts.ammo = 'ammo.js';
```

These are two variables that contain the path to two JavaScript files ammo.js and physijs_worker.js. Ammo.js is the physics library which physi.js is built on and physijs_worker.js is another file that contains the web worker of the library. The code in physijs_worker will most likely be executed on another core or processor, which will increase the performance.

```javascript
scene = new Physijs.Scene({ fixedTimeStep: 1 / 120 });
scene.setGravity(new THREE.Vector3( 0, -30, 0 ));
```

The original Three.js scene is replaced by a Physijs.Scene and a fixedTimeStep is set to 1 / 120 meaning that the maximum movement of an object will be 120 units per second. A physijs scene has a gravity parameter that can be set, which specifies the gravity of the scene, which in this case is set to -30 faced down the y axis.

```javascript
shape = new Physijs.BoxMesh( box_geometry, material, 0 );
shape.material.color.setRGB(1,18,255);
shape.castShadow = true;
shape.receiveShadow = true;
shape.rotation.set(0,0,0);
shape.position.set(15,10,20);
scene.add(shape);
objects.push(shape);
```

Physi.js provides nine different standard objects already integrated with the physics engine. In this example is a BoxMesh created with a specific geometry, material and a mass. The other specified parameters is the same as the Three.js object, color, shadow options, rotation and position. Finally the Physi.js object is added to the objects array, which will be added to the scene.

### 2.12.3 Bulletjs

Bulletjs is just as ammo.js another export from the Bullet physics engine written in C++ [41]. It has been forked six times on Github, indicating that ammo.js is more popular as a physics engine.

### 2.12.4 CANNON.js

Cannon.js is a physics library made in JavaScript [42]. It is available on Github were it has been forked almost 100 times. The library is small with a size of only 100Kb, making it fast to load. Cannon is used in other projects like the Babylon.js 3D rendering library. It currently has seven contributors, but the last commit was done 2013-08-11 when checked 2014-05-14, which could mean that the development of the library is not proceeding.

### 2.12.5 JigLibJS

JigLibJS is a library built on JigLib which is a physics engine written in C++ [43]. It has been forked twelve times on Github and has not received any updates since July 2011.
3 Method

This section describes the method used for answering the research questions mentioned in section 1.1. The method should be so well designed that results undoubtedly answers the research questions. The results should be measurable for each library so it is easy to compare the different libraries from the others. A definition of a good software library is essential in order to determine which of the libraries that are the most promising ones.

The method has been distributed into different sections described below. Each section is followed by a text describing its purpose.

Requirements of a 3D web application In order to narrow the scope of the thesis requirements are defined for a computer game that the libraries should implement. This highlights essential requirements of the libraries that can be used fulfilling the game’s requisists.

Requirements for a software library The defined computer game requirements are further divided to specifications, for the software libraries.

Selecting software libraries All libraries brought up in this thesis are compared against each other with the defined requirements as reference. Libraries that do not fit the requirements as good as other get disregarded until only two solutions are available.

Implementing the game The two selected solutions will implement the initial game requirements. Each implementation process will be documented and analyzed. The gathered results will be compared and the best implementation process will be presented.

Performance test Both solutions will be compared with a performance test measuring the FPS (Frames Per Second).

Results and discussion The answers to the research questions will be presented and discussed.

3.1 Requirements

None of the elicitation techniques mentioned in section 2.1.2 were used when gathering requirements in this thesis. The mentioned requirements were made for bigger projects which assumed that there were many stakeholders to gather requirements from. The author had little knowledge regarding software libraries which made it hard to define requirements directly for software libraries. In order to simplify the elicitation process requirements were defined for a computer game. The author had previous knowledge regarding computer games which made it easier to define requirements for a specific computer game. When the games requirements were defined they could be furthered divided into software library requirements.

3.2 Computer game requirements

The requirements for a computer game are listed below.

Game objective and scene The game should take place in a three dimensional empty space, which initially contains no objects. By pressing a specific key a sphere is release in the space. The objective of the game is to move the position of the sphere to a platform in the scene. The sphere is effected by gravity making it falling in the scene. The spheres direction is altered by colliding with platforms that
are added and moved in the scene by the user. The platforms are not affected by that gravity making them still in the scene.

**Free look camera**  The game has a free look camera that is moved and rotated by the input from the keyboard and the mouse. The keyboard inputs moves the camera forward, backwards left and right. The mouse changes the rotation of the camera.

**Game objects**  There are two different game objects. One object is a sphere and the other is a plane. Each object should be able to be configured into different sizes and colors.

**Physics**  The game should include game physics, in the form of gravity and collision detection.

**Graphics**  The game should be in 3D. Ambient and directed lights should be enabled. Shades should be enabled.

**Modes**  Two different modes should be available which changes the view and controls of the game.

  **Drag and drop**  In the first mode the user is provided a cursor to drag and drop the movable objects to new positions, and the rotation of the camera is changed by moving the cursor to the sides of the screen. While a plane is dragged they are selected. When a plane is selected they turns red and their rotation can be changed by pressing the arrow keys on the keyboard. When the plane is dropped the color of the plane returns to its initial color.

  **First person**  In the second mode is the cursor locked in the middle of the screen and the camera rotation is instantly changed when the user is moving the mouse. In the middle of the screen is an aim that has the form of a black dot. By pointing the aim over a box and pressing the left mouse key the plane gets selected and moves in front of the camera. Just as in the drag and drop mode changes the planes color to red and the rotation of the plane can be changed. The plane gets deselected by pressing the left mouse button again and then the position of the plane is fixed.

**3.2.1 Software library requirements**

**Predefined meshes**  The library should provide resizable sphere and plane meshes that can be used in the game.

**Availability of multiplied objects and functionalities**  There should be no limitation in amount of objects that can be used, set by the library. The limitation should appear naturally when the performance reduces by calculations needed.

**Basic functionalities**  The library should provide basic objects like a rendering scene, cameras, lights and different kinds of meshes.

**Physics**  The library should have physics integrated or have documentation describing how to integrate with a physics library. The physics environment should be configurable with gravity specified affecting the scene. It should be possible to create objects which is not affected by the gravity but still affects other objects with collisions.
3.2.2 None-functional requirements

**Open source** The library should be open-source.

**Performance** The library should be able to render 100 bouncing meshes simultaneously without any visual lags.

**Tutorials** There should be some kind of tutorials that simplifies the development process.

**Simplicity and usability** There should be a clear structure of the library and it should be well documented. It is a big plus if there is a community where questions can be asked and answered.

3.3 Performance test

The two selected libraries will be evaluated in a performance test that includes a scene with a physics engine, a big plane and 100 boxes that will be added to the scene above the plane, simultaneously. The boxes will fall and bouncing against the plane and each other which is a challenging task for the libraries. Performance will be measured using FPS (frame per second) which will be recorded six times on the same time intervals for each library. The higher value of FPS, the better.

3.4 Development process

The development process is an important part of the evaluation but is hard to measure. The development process can be summarized as the simplicity, usability and enjoyment working with the libraries.

3.5 Correctness

A grasp of correctness will be evaluated in the thesis since their is a need for realism in the application. It is very hard to measure correctness in real time 3D applications where speed is more valued then perfect correctness. Thereby is correctness determined by visual factors where the author determines if the library delivers a natural behavior.

3.6 Selecting libraries

When all requirements have been elicited they are ranked regarding their importance. Below is each category listed wit a rank and motivation, the lower number a category has the better rank.

1. **Functionalities** The functional requirements is regarded as the most important requirement. If a library do not have enough functionalities to implement the functional requirements they will be disregarded.

2. **Simplicity and usability** Developers have different knowledge regarding graphics, web development, and 3D real-time applications. Thereby it is good to make a library as simple as possible so that many people can use it. If the library is simple the code will also be simple and simple code with good structure results in better programs. A simple library attracts more developers which results in more contributors and stakeholders.

3. **Performance** Performance in this aspect is the speed and correctness of the library. Performance is ranked as an important factor since end users usually has high demands regarding performance. The performance is measured in a performance test.
Table 1. Library categories

<table>
<thead>
<tr>
<th>Plugins</th>
<th>Not Plugins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physics Included</td>
<td>No physics included</td>
</tr>
<tr>
<td>Company</td>
<td>Company</td>
</tr>
<tr>
<td>Silverlight</td>
<td>SpiderGL</td>
</tr>
<tr>
<td>Flash 3D</td>
<td>KickJS</td>
</tr>
<tr>
<td>Unity</td>
<td>PhiloGL</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: A categorization of the WebGL libraries brought up in this thesis.

Table 2. Library specifications

<table>
<thead>
<tr>
<th>Library</th>
<th>Community</th>
<th>2D compatibilities</th>
<th>Programming level</th>
<th>Last updated</th>
<th>Google hits in thousands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three.js</td>
<td>Strong</td>
<td>NO</td>
<td>High Level</td>
<td>2014-04-24</td>
<td>525</td>
</tr>
<tr>
<td>Babylon.js</td>
<td>Strong</td>
<td>NO</td>
<td>High Level</td>
<td>2014-04-28</td>
<td>41.5</td>
</tr>
<tr>
<td>Enchant.js</td>
<td>Weak</td>
<td>YES</td>
<td>High Level</td>
<td>2014-03-28</td>
<td>203</td>
</tr>
<tr>
<td>CubicVR</td>
<td>Weak</td>
<td>NO</td>
<td>Low level</td>
<td>2014-01-13</td>
<td>14.3</td>
</tr>
<tr>
<td>SpiderGL</td>
<td>Weak</td>
<td>NO</td>
<td>Low level</td>
<td>2013-06-02</td>
<td>3.76</td>
</tr>
<tr>
<td>KickJS</td>
<td>Weak</td>
<td>NO</td>
<td>High level</td>
<td>2014-04-09</td>
<td>1.48</td>
</tr>
<tr>
<td>PhiloGL</td>
<td>Weak</td>
<td>NO</td>
<td>High level</td>
<td>2014-05-14</td>
<td>6.81</td>
</tr>
<tr>
<td>CopperLicht</td>
<td>Middle</td>
<td>YES</td>
<td>High level</td>
<td>2014-04-06</td>
<td>16.2</td>
</tr>
<tr>
<td>Turbulenz</td>
<td>Middle</td>
<td>YES</td>
<td>High level</td>
<td>2014-04-06</td>
<td>306</td>
</tr>
<tr>
<td>StormengineC</td>
<td>Weak</td>
<td>NO</td>
<td>High level</td>
<td>2014-04-12</td>
<td>0.959</td>
</tr>
<tr>
<td>FRAK</td>
<td>Weak</td>
<td>NO</td>
<td>High level</td>
<td>2013-08-01</td>
<td>0.447</td>
</tr>
</tbody>
</table>

Table 2: A further categorization of the libraries.

4. Community and tutorials Having a dedicated community where people help each other makes a library more attractive and easier to work with.

5. The amount of users The amount of people using the library might be interesting since the library possibly is more stable since it has been tested before.

4 Results

This section describes the results that were produced from the method described at section 3. In Table 1 shows an overview of the different WebGL libraries that were compared in this thesis. The libraries have been categorized in different sections, describing if they are plugins, made by a company and if they include physics.

All libraries has been divided into two different programming levels, High and Low. High level libraries only require JavaScript as prerequisites while low level libraries require knowledge regarding GLSL ES
and shaders.

The community of a library is either graded as Weak, Middle or Strong. Communities graded as Weak are communities which do not provide any communication platform such as forums or IRC channel. Communities graded as Middle have a communication platform, but there are low or none communication on the platform, question asked might be answered after a week or later. Communities graded as Strong has a communication platform and there is a high amount of communication on the platform, question asked will be answered the same or next day.

4.1 Libraries which did not qualify

Since this thesis has a limitation in time a big part of the libraries were disregarded simply by reading about them at their home-pages. This means that libraries can be better then this thesis has investigated, but disregarded since they have bad marketing or not updated websites. Generally the libraries where disregarded since they did not show any real game implementations. Many of the libraries did not mention how to include physics in a simple way, and the community was small or none existent. Below are all the libraries which did not qualified followed by a description of why it did not get selected.

4.1.1 Enchant.js

Most of the games listed at Enchant.js official website are 2D games executed in small canvases. The published 3D games were very simple which made it hard to determine if a library has enough potential for advanced 3D games. The rendering quality is lower in Enchant.js resulting in animations with visible pixels. The development process is very easy using the supplied book but the section describing 3D possibilities is very limited. The development of the code seems to have stopped and there is no activity on the forum.

4.1.2 Copperlicht

CopperLicht is one of the more interesting alternatives when evaluating the JavaScript libraries. It has promising demo applications and includes many tutorials explaining the library. The forum provided by Copperlicht contains little activity with a few questions to the libraries author. This is a big difference from the selected libraries were its communities have discussions and showcases making the development process more social. There is always a risk that libraries provided by companies will be limited or new features starts costing money when the library becomes popular.

4.1.3 Turbulenz

Turbulenz is a impressive library with different games that all were fully functional. The most impressive game was a multilayer first person running almost smoothly, unfortunately was the source code was not provided. The tutorials of the selected libraries is more comprehensive and the communities of the selected libraries have a higher grade of activity. Just as Copperlicht is Turbulenz owned by a company, which could result in further versions starts costing money.

4.1.4 FRAK

None of the examples provided for FRAK contained any physics simulations. The forum is filled with spam and no activity regarding the library, making it useless. There was long time since the last contribution to the library codebase.
4.1.5 CubicVR

CubicVR is one of the earliest libraries, used as a showcase for FierFox to show the new functionalities of WebGL. There are very few examples on its website. None of the examples contains physics or controllable objects. There has hardly been any contributions to the library since 2013, the forum holds 35 topics with the latest post in 2013.

4.1.6 SpiderGL

SpiderGL did not qualify since since the library is a low level library which requires developers to have good knowledge regarding GLSL ES. The library has only two topics so the discussion can be summarized to zero. The examples on the contains no movable objects and no physics.

4.1.7 KickJS

The latest available version of KickJs is version 0.5.5 which gives the impression of an unfinished version. The last contribution to the library was in 2014-04-09. The most impressive tutorial on the website is a simple chess game, not including any physics. There are no provided communication channel for its community which makes it hard to retrieve inspiration and help.

4.1.8 PhiloGL

PhiloGL has many examples of 3D rendering on its website, but none of them includes physics. The discussion forum includes many topics but the last topic was posted 2013-07-15. The best way of learning PhiloGL is to use one of the many WebGL books listed on its website. PhiloGL is a low level library which means that a knowledge of GLSL ES and 3D rendering is essential to develop applications for the library.

4.1.9 StormengineC

The demo’s posted on the website did not work and were printing error messages. The community only has one member which is the only contributor to the library. The last commit to the library were done in late April 2014.

4.2 Selected libraries

Factors that made Three.js and Babylon.js the selected libraries were their many examples that were presented on the websites and their big communities. Both of the libraries are open-scource and have many contributors to the libraries. Babylon.js and Three.js both had impressive game examples on their websites. Some games included almost all requirements that were set in this thesis. Their communities are very developed, most problems and discussions can be found by a simple search on Google. Both showed examples how to easily include physics to the developed games.

4.2.1 Three.js

Three.js is probably the most used 3D library for WebGL. It is the library with the most activity on its Github page compared with the other libraries and it steadily receives new updates. There are 75 impressive examples on the website where many are games including physics and collision detection. The library has its own google circle page, an own IRC channel and many questions are asked and answered at Stackoverfl ow, which is a question and answer forum. Three.js does not include any physics in the library but with physi.js which is special made for Three.js they together become a impressive game engine. Three.js is qualified for all requirements.
4.2.2 Babylon.js

Babylon.js has 60 different examples on its main website, where some of them includes game physics. Babylon.js is the only library that includes physics which simplifies the development process. It has a very active forum for discussion, help and inspiration. A question asked on the forum is usually answered within one day. Some examples available on the website fulfils most of the this thesis requirements.

4.3 Development process

This section describes the process of implementing the requirements defined at section 3.1. The development process is a very critical part in order to determine the best library. A library that is easy to work with will attract more developers, more games, more contributors and a stronger community.

Both libraries is regarded as High Level libraries meaning that no knowledge is needed regarding 3D rendering. A person who has been working with JavaScript before can easily develop applications using Babylon.js and Three.js with Physi.js.

4.3.1 Three.js with Physi.js

The development using Three.js and Physi.js was done using an example downloaded from Physi.js official website. The example was a game of the common blocks game jenga. The game had already solved the physics requirement, the drag and drop requirement, the free look camera requirement and the graphics requirement. The advantage developing applications from examples is the fun that directly comes from the development process. Instead of starting with adding a ball, working some hours to make it affected by physics and even more hours to make it possible to drag and drop it, a developer can directly start experimenting with the provided environment. This is also a disadvantage since it might be a hassle to understand code that other persons has been writing. Developers gets no real knowledge from working with finished examples unless they carefully investigate and elaborate with the code.

The integration of Physi.js with Three.js is very simple due the examples provided by the Physi.js and its instructions retrieved at the github page. Obstacles that occurred were solved by simple Google search and the solutions were usually found at stack overflow.

If objects do not have a mass they will not be affected by the gravity or the collision detection, but in our requirements we want objects with collision detection but not affected by gravity. Thanks due the flexibility of JavaScript it was simple to add a parameter to all objects that wants to bypass physics calculation, and in the gravitation calculation look for this parameter and exclude them from the physics calculation.

4.3.2 Babylon.js

Babylon.js also included games when the latest version where downloaded from github, but the codes where encrypted so they could not be used as examples. Instead the development process was in Babylon.js focused on tutorials which clearly explains the structure of the library. On the forum html5gamedevs.com there are steadily discussions where developers can ask questions and discuss anything regarding the library. The forum has a very dedicated community and questions are usually answered within one day. Since the game was created from scratch and not from examples it results in more functionalities that have to be developed. This ensures that the developer has a fully understanding of the application. Integrating physics in Babylon.js is a lot simpler then integration physics with Three.js and Physi.js, thanks due that physics is already contained in the library. Enabling physics is simply done by turning on a parameter, since all the objects with standard values already are contained in the scene. Even though the integration is very simple in Babylon.js there is a lack of flexibility and functionalities,
resulting in difficulties in the development process. The freeCamera in Babylon.js is a very good example to show the simplicity of Babylon.js and its lack of flexibility. The freeCamera is a camera which is rotated and moved by the user. Below is the code listed to attach a new camera to the scene.

```javascript
var camera = new BABYLON.FreeCamera("camera", new BABYLON.Vector3(0,0,-15), scene);
scene.activeCamera = camera;
camera.attachControl(renderer, true);
```

By these lines the camera is positioned in the middle of the scene in position \((x, y, z) = (0, 0, 0)\). The camera can be moved with the arrow keys and the mouse. The arrow keys handles the position of the camera, and by keeping the left mouse button pressed while moving the mouse, the rotation of the camera is changed. It is also possible to change the keys so instead of using the standard arrow keys on the keyboard to change the position of the camera, they can be changed to any other keys the user defines. Below is an example how the Keys are changed to the W, A, S, D keys, to fit the requirements.

```javascript
camera.keysLeft = [65]; // 65 = A key
camera.keysRight = [68]; // 68 = D key
camera.keysUp = [87]; // 87 = W key
camera.keysDown = [83]; // 83 = S key
camera.angularSensibility = 6000;
```

These few lines of code is everything that is needed to implement a movable camera in Babylon.js. In order to implement a camera in Three.js there are more lines of code that has to be used to achieve the same results.

One tricky part in Three.js was the drag and drop functionalities since a mesh should be moved in a three dimensional world, but the mouse can only be moved in two dimensions. This was solved by setting up a plane that restricted the mesh of moving in one dimension and then move the mesh in only two dimensions. This solution was directly taken from a example that were retrieved while downloading the library. The examples does not contain any comments, which can bot be considered good since the developer has to read the code carefully to understand it, while it can be bad since it takes more time to realize the code behaviour.

But this is not the functionalities of the camera desired in the game. The left mouse key should not be pressed in order to change the rotation of the camera. There was no parameter to set to change the way the camera should rotate, so the solution was to implement a free moving camera from scratch. In three.js there were so many applications developed which made it possible to copy the solution from a different application. In Babylon.js the solution had to be found by searching how a free moving camera is implemented in a other language to be implemented in Babylon.js, a solution which needs a knowledge of matrices, vectors and mathematics.

This is a good example describing the differences of the libraries. Babylon.js consists of small and powerful functions which construct applications with less code. But these function calls also limit the developer, resulting in workarounds if the function does not completely fulfil the requirements. In Three.js the application consisted of more code but it was also more flexible then Babylon.js. Three.js had more functionalities compared with Babylon.js.

### 4.4 Evaluating the games

The developed games where similar in the first appearance, but the versions are very different. All requirements were filled in the Three.js game while there were some of the requirements that never
The final result of the Three.js game were fulfilled in Babylon.js. Since Three.js has more functionalities that are highly configurable they can be combined to build an impressive game. In the subsections below there is a description of each implementation.

### 4.4.1 Three.js and Physi.js

The final version of the Three.js and Physi.js game can be seen in Figure 4. All the requirements were fulfilled and the game works as intended. The game had problems when it comes to performance since there exist lags that occurs frequently. This has a detrimental effect on the game and reduces its overall impression.

### 4.4.2 Babylon.js

The final version of the Babylon.js game can be seen in Figure 5. Many problems occurred during the development process. One big problem was the flexibility of the library where modifications and customization to the functions were not possible. An example of this is the free lock camera which were available but its controls could not be customized. In order to fulfill the requirements of the game is a own implementation of the camera needed. These modifications takes long time to implement, which results in less features implemented. The features which were implemented was the physics engine, the user input and a free look camera. The problems could probably be implemented if there were more time given to the development of the game. The game runs more smoothly with no lags and physics were more realistic in Babylon.js compared with Three.js and Physi.js.

### 4.5 Performance test

The performance test was created on the same computer with the following specification described at Table 3.

In order to validate the performance of the libraries two identical scenes was constructed one for Babylon.js and one for Three.js with Physi.js. The Babylon.js scene can be seen in Figure 5 and the Three.js scene can be seen in Figure 6.
Figure 5. The Final result of the Babylon.js game

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>AMD Turion II P520 Dual-Core Processor X 2</td>
</tr>
<tr>
<td>Graphics</td>
<td>Gallium 0.4 on AMD REDWOOD</td>
</tr>
<tr>
<td>OS</td>
<td>Linux Ubuntu 13.10 64 bit</td>
</tr>
<tr>
<td>RAM</td>
<td>4GB</td>
</tr>
</tbody>
</table>

Table 3 the hardware specification of the test machine.
The tests consisted of 100 boxes that were randomly placed above a big plane. The frames per second (FPS) was measured in two different ways. The first measurement was done through a FPS implementation in the code and is called FPS1, which was a piece of code that were placed in the rendering loop which calculates the number of calls per second. The second measure was done by a FPS measure implemented by the library and is called FPS2. Since the physics is calculated at a different core in the Three.js with Physi.js solution there is another measurement parameter needed to show the FPS of the physics. Table 4 shows the results of the performance test done in Babylon.js and Table 5 shows the performance test done with Three.js and Physi.js.

The boxes are not moving the first three seconds of the performance tests due to initialization and
Table 4 shows the results of the Babylon.js performance tests.

<table>
<thead>
<tr>
<th>Seconds</th>
<th>3</th>
<th>6</th>
<th>9</th>
<th>12</th>
<th>15</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS1</td>
<td>60</td>
<td>48</td>
<td>30</td>
<td>24</td>
<td>21</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>FPS2</td>
<td>37-45</td>
<td>30-32</td>
<td>18-23</td>
<td>15-20</td>
<td>17-21</td>
<td>18-21</td>
<td>20-27</td>
</tr>
</tbody>
</table>

Table 5 shows the results of the Three.js performance tests.

<table>
<thead>
<tr>
<th>Seconds</th>
<th>3</th>
<th>6</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS1</td>
<td>40-60</td>
<td>52-60</td>
<td>55-60</td>
<td>55-60</td>
<td>55-60</td>
<td>60</td>
</tr>
<tr>
<td>FPS2</td>
<td>10-15</td>
<td>15-18</td>
<td>18-25</td>
<td>13-40</td>
<td>16-26</td>
<td>17-28</td>
</tr>
<tr>
<td>FPS2P</td>
<td>46-47</td>
<td>50-53</td>
<td>49-53</td>
<td>49-57</td>
<td>56-57</td>
<td>57-58</td>
</tr>
</tbody>
</table>

As can be seen in Table 3, the FPSP of Three.js is low in the beginning but increases towards the end, which is clearly seen in the execution of the performance test. This is different from Babylon.js which has a high grade of FPS in the beginning of the performance test but decreases after time. Visualizing the performance tests shows that the performance of Babylon.js is performing better than the performance of Three.js which continuously has small lags that destroys the visualization. This is very interesting since the FPS of Three.js is always higher than the FPS of Babylon.js. If more time was given, the root cause of these lags would have been investigated.

5 Discussion

Early in the analysis it was clear that the libraries could be categorized into different groups. Table 1 at section 4 shows an overview of the different categories. Each library is categorized by weather is is a plug-in, or if it contains physics or if it is developed by a company or not. The table shows that all the plug-in based libraries are provided by companies and has physics included. This is not surprising since the companies holds an economic interest in their libraries. If a company would provide a library which has less functionalities or quality compared to its competitors, it would loose customers and revenue. The libraries provided by companies are made for the game development business which explains why all of them also includes physics.

In order to narrow the scope of the thesis the plug-in based libraries has been excluded from the thesis. The plug-in based libraries have development tools which develops games using drag and drop. The process of selecting good libraries is complicated if these aspects are included since the aspect of money and knowledge of the developers becomes far to crucial in the process of selecting good libraries. Thereby it is a requirement that the library should be free of charge and that the developer has knowledge of JavaScript.

Turbulence, Enchant.js and CopperLicht are all libraries provided by a companies and they all contains physics. It is beneficial for these big projects to have a company that can pay for the development process and make sure its proceeding, while a open-source project requires motivated individuals to finish the project without any pay. There were no WebGl libraries developed by companies that did not contain physics, the reason might that the open-source WebGL frameworks is sufficient, or that there are frameworks that are kept in-house. Even though the libraries like Turbulenz and CopperLicht provided
impressive examples they are not as popular as Three.js. This could have to do with the impression of having a library that is dependent on a company. It is a possibility that further versions of the library might start costing money when the library becomes popular.

Table 2 is a further categorisation of the different libraries without the plug-ins. Many of the libraries were abandoned projects which had been created by enthusiasts but not attracted any developers. This can be seen by the number of hits on Google and the latest commits to the the GitHub repositories. The more active libraries like Three.js and Babylon.js has a higher amount of hits and contributors to their libraries. Google hits is a very good measurement for libraries with names that ends with .js, since that shows a distinct number of hits for that specific library. It is important to remember that the number of hits regarding Enchant.js also includes the 2D part of the library, and the library is mostly used for 2D game development. Turbulenz is the German name for turbulence, meaning that the Google search gives hits on German texts mentioning turbulence. The Library with most Google hits is Three.js which is one of the selected libraries.

It is common that developers gather around a few libraries since they benefits from it. Having many developers using the same library results in more developers answering questions and more developers contributing to the library. The best way of attracting developers is to have good marketing and a simple development process. The two selected libraries had very good marketing. Three.js was the library that had the most examples on their website, which were developed by its community. Babylon.js has the same concept of showcase on their main website and also a popular forum where help and information is available. The forum contains many different topics that is correlated to Babylon.js in one way or another. This forum is great for posting questions regarding a specific task or questions about the library. It is a atmosphere of creative and smart people giving advices, showing their applications and the creator of the library puts a lot of effort in the forum creating a social community. Being part of a community makes it very fun to develop games. Questions posted on the forum is answered within 24 hours.

It is not surprising that most libraries would be small and not fully developed since WebGL is still a young technology. Starting a project is easy but fulfilling it is a different story. It takes a lot of time to develop a competitive WebGL library, even more time to create a competitive game engine. Initially there is a hype around new technologies attracting enthusiasts to investigate the new technologies like WebGL. When they have investigated the technology for some time their needs might be fulfilled, even though there are some requirements that should be implemented, leaving the project unfinished. It is important mentioning that some of the libraries is not made for game development. Many are 3D rendering libraries which makes WebGL available in a solution that is used by JavaScript. These libraries are more suitable for applications where only rendering is needed. There is also low level libraries which are more similar too GLSL ES making them perfect for users that needs a lightweight library that is highly mutable.

5.1 The Performance test

The performance test at table 4 and 5 showed interesting results. The hypothesis assumed that a high FPS value will result in a smoother and natural impression of the physics. Both libraries had good measure of FPS but lags occurred in the Three.js with Physi.js solution even though it has a overall higher FPS measurement. An explanation could be that there were mistakes in the code with the integration of Three.js and Physi.js. The frames per second could in fact be higher then Babylon.js but still containing lags if the frames are not equally spread with the same interval between the frames. As can be seen in the FPS values at section 4.5 the physics frame per second is lower then the rendering FPS. This means that the object’s position will be the bottleneck of the rendering and it does not matter how fast the frames are updated if the pictures that are rendered are identical.
5.2 The development process

Three.js with Physi.js and Babylon.js were selected for further investigation in a implementation of the game requirements. Both libraries has the same simple structure with a rendering cycle were the scene gets updated. No pre knowledge of real-time rendering and graphics is needed before developing games using Three.js with Physi.js or Babylon.js. Knowledge of web development or programming simplifies the development process. Building 3D applications is a very fun way of programming were you get instant feedback while programming and a few lines can make a big visual impact. Below is a summary of the development process for both the libraries.

5.2.1 Babylon.js

The development process using Babylon.js is very simple with few functions that handles big parts of the application. It is simpler to develop game using Babylon.js than Three.js if the available functions provided by Babylon.js is enough to fulfil the requirements. The functions provided were not sufficient to fulfil the requirements of the game, thereby was the implementation difficult and troublesome. A good example of this is the first person camera requirement. Babylon.js provides a finished solution regarding moving cameras but its controls differs from the requirement. The camera is not instantly rotated by moving the mouse, instead it is rotated by clicking and rotating. There is no way of customizing the rotation by setting other controls, so the rotation had to be implemented by hacks bypassing Babylon.js functions. Since the library still receives updates is it likely that the library will provide functions to implement all requirements in the future. The development process is entertaining since the forum contains many interesting and creative persons, which helps each other to reach their goals. The main contributor and creator has very good knowledge about Wl rendering since he has been working with Wl rendering before at Microsoft, developing its Silverlight plug-in.

5.2.2 Three.js with Physi.js

The solution was built from an initial jenga example game, which had solved most of the requirements like drag and drop. Requirements that were not implemented were solved by quick search on Google. An example of this is the first person camera. Since Three.js with Physi.js does not provide as general functions as Babylon.js, there is more code needed to build the same solution as Babylon.js. This gives more freedom to the developer, freedom that made it easy to implement the requirements of the game.

5.3 Criticism

The information regarding the different libraries is obtained from the libraries websites. The information is not written scientifically, and probably written in a positive manner. Information regarding libraries that is not mentioned on the websites has not been used in this thesis, resulting in miscalculation of the libraries potential fulfilling the requirements. As an example has a lot of information been gathered by reviewing the demo’s and examples provided by the libraries, if there is no explanation or example showing an integration with physics has the library been judged as a library with no potential of physics. An optimal solution to cover this hazard would be to implement the same game for each library, but that optimal solution would be too time consuming.

There is no elicitation technique used to retrieve the libraries requirements. Since the requirements is determined only by the authors judgement of important aspects of 3D software libraries there is a chance that important requirements has been lost. If the requirement do not cover important aspects regarding 3D web applications the results of this thesis is misleading. If more time were given a better solution would be to use different elicitation techniques in order to gather the requirements.

The performance test showed small lags for Physi.js. This was unexpected since the FPS showed that Physi.js had a better average score compared with Babylon.js. FPS should be a good measurement
regarding rendering cycle where a high number results in a smoother result. A reason could be that the integration of the physics might have been wrongly implemented. Another reason can be the hardware on the computer that is not sufficient. Physi.js has been constructed completely from scratch while Babylon.js has been translated from a stable library written in C. If more time were given regarding the thesis a answer to the lags would be an interesting aspect to investigate.

5.4 3D web applications in the future

Since the release of WebGL it has emerged 3D frameworks on the web. The web still receives new functionalities and performance upgrades, resulting in more applications and decreasing the need of installed programs on computers. Publishing a 3D web application on the web makes it instantly accessible by millions of people, on many different platforms, all over the world. This is a huge advantage compared to downloading software and then installing it for every computer. The disadvantage of 3D web applications is the bandwidth that is needed in order to execute the programs. 3D meshes is relatively big and takes a long time to download resulting in long waiting times for the users. But the speed of internet has always improved and probably will continue doing so.

Right now it has been a trend for web browsers and big companies to build interesting new applications that impress users and customers with the new functionalities of WebGL. These applications has been more as a showcase instead of fulfilling any practical use. In the future will 3D web applications probably be more useful and used in normal 3D applications like games, design tools and video making. WebGL might also be used in websites as luxury elements including animations and 3D environments. JavaScript has previously been categorized as a sloppy computer language with bad structure and poor performance. But thanks due the stronger JavaScript engines has the language been popular and is today seen as a language for the future.

Another possible future would be that all computation is done in the cloud and no computation is done on computer devices. Then there is no need for good GPU and CPUs in computers, phones and tablets, which will be the end of 3D web applications using JavaScript and WebGL. This future is extremely dependent on fast network.

6 Conclusion

The goal of the thesis is to find an optimal solution for creating 3D web applications on the Web using WebGL without any plug-ins. This was done by defining requirements of a real 3D web application including physics, splitting these requirements to fit JavaScript libraries that will implement them. When the requirements were defined for the JavaScript libraries, they were matched against eleven different JavaScript libraries. The two libraries that best matched the requirements were selected to be used implementing the initial requirements of the 3D web application. The two selected libraries were also used in a performance test with challenging 3D rendering including physics. Finally Three.js handling 3D graphics and Physi.js handling physics was chosen as the best libraries to be used implementing the requirements of the initial game. Below is the answers to the initial questions.

Which are the most promising open source WebGL libraries for creating 3D games that includes game physics?  Three.js for 3D rendering combined with Physi.js handling the game physics is the best solution for fulfilling the requirement of the defined 3D web application. Another promising library is Babylon.js that has better rendering performance then Three.js with Physi.js and functions that is simpler to use. Unfortunately the functions provided by Babylon.js is not sufficient to fulfill the requirements which leads to time consuming modifications. If the functions provided by Babylon.js would be sufficient to implement the requirements of the 3D web application, Babylon.js would be regarded as the best library. What makes Babylon.js and Three.js with Physi.js the best of the 11 libraries is their
simple integration with 3D physics, many functionalities, strong community and very simple development process.

**What are the differences between the available WebGL 3D libraries?** There are many differences between the available 3D libraries. Most of the libraries are used for 3D rendering while some are used for parallel computing. The graphics libraries can be furthered be categorized as 3D rendering libraries, 2D rendering libraries and 2D & 3D rendering libraries. There are libraries that can be developed using tools where no knowledge of programming is needed. The 3D libraries can be furthered diverged into libraries that are similar to the shading standard GLSL ES, were knowledge of shaders are needed and libraries were only knowledge of JavaScript is needed. Most of the 3D rendering libraries are abandoned projects with limited functionalities and help. A categorization of the different libraries can be seen at section 4.

**What expertise is needed to develop 3D games with game physics?** Hardly any knowledge is needed to develop 3D web application with game physics when using a developing application, but it will take some time getting used to the program. These programs are limited and can be extended by a purchase. In order to develop a 3D web application through programming a basic knowledge of JavaScript and object oriented programming is useful. The Three.js with Physi.js and Babylon.js are good examples of libraries with simple development processes which provides examples and forums that are useful and helps during the development process. 3D web development is a entertaining way of programming since small pieces of code have big impact on the application.

7 Future research

Delays occurred for Three.js and Physi.js during the performance test. These delays were clearly visible even though the FPS measurement showed that it had an better score then Babylon.js. It would be very interesting to investigate sources of these delays. Was it hardware specifications that made the delays occur or was it a wrong implementation of the physics? It would be an advantage for Babylon.js if these delays are general for more applications that uses Three.js and Physi.js, since that would mean that Babylon.js is the best library for advanced 3D applications with many objects that should be rendered.

The requirements of the application were very simple. Many popular games are actually very simple but in a scientific point of view would it be interesting to scale up the requirements to very challenging applications. 3D web applications faces the challenge of bandwidth when loading highly detailed 3D objects. Further work could investigate and try to find a solution for loading 3D objects on the fly while the user is playing the game. This is a very interesting topic and a good solution would be a breakthrough in 3D web applications.

It would be interesting to make a demanding 3D rendering test. What is the amount of detail that Babylon.js and Three.js can handle without introducing delays in the rendering? At which level of rendering would a user consider that the 3D application is unmanageable?

References

1. CERN: A summary of the Tim Berners-Lee and the worlds first website.  
   http://home.web.cern.ch/topics/birth-web 2014-02-08


7. Soren Lauesen, Software Requirements Styles and Techniques Pearson Education Limited, Edinburgh Gate, 2002 pages 12-15,


14. Unity Website: Overview of their editor environment http://unity3d.com/unity/workflow/integrated-editor 2014-02-08

15. A presentation made by Google about the features of HTML5 http://slides.html5rocks.com/#landing-slide, 2014-02-07


17. A presentation made by Google about the features of HTML5 http://slides.html5rocks.com/#landing-slide 2014-02-07


20. WebGL programming guide: Interactive 3D graphics programming with WebGL by Kouichi Matsuda & Rodger Lea Edwards Published June 2013 in United States by Brothers Malloyin Ann Arbor, Michigan page 59-61,
21. High Performance JavaScript: by Nicholas C. Zakas Published by OReilly Media, printed March 2010 XI - XIII


24. Game development with Three.js: by Isaac Sukin Packt Publishing Ltd. Livery Place 35 Livery Street Birmingham B3 2PB, UK. published October 2013 page 5-8


30. FRAK official web page http://www.3dtech-rd.net/frak/main/, 2014-05-14

31. CubicVr a open-source 3D JavaScript library https://github.com/cjcliffe/CubicVR.js/

32. SpiderGL a open-source 3D JavaScript library http://spideryl.org/

33. Kick.js a open-source 3D JavaScript library https://github.com/mortennobel/KickJS/

34. PhiloGL a open-source 3D JavaScript library A open-source 3D JavaScript library

35. StormengineC the official github page https://github.com/3DRoberto/stormenginec 2014-05-14

36. Game Physics engine development by Ian Millington published march 7 2007 by CRC n, page 1-5

37. Game Physics engine development by Ian Millington published march 7 2007 by CRC n, page 5-6

38. Game Physics engine development by Ian Millington published march 7 2007 by CRC n, page 6-7


40. The official github page of Physi.js https://github.com/chandlerprall/Physijs, 2014-02-15


42. CANNON.js official github repository https://github.com/scheppe/cannon.js, 2014-05-14

43. JigLibJS the official github webpage https://github.com/supereggbert/JigLibJS, 2014-05-14