Using WebGL to create TV-centric user interfaces

by

Jonathan Karlsson

LIU-IDA/LITH-EX-G--14/058—SE

2014-06-11
Final thesis

Using WebGL to create TV-centric user interfaces

by

Jonathan Karlsson

LIU-IDA/LITH-EX-G--14/058—SE

2014-06-11

Supervisor: Johan Åberg
Examiner: Peter Dalenius
Using WebGL to create TV-centric user interfaces

Jonathan Karlsson
Linköpings universitet
jonathan@elektronvolt.se

ABSTRACT
In recent years the user interfaces of the TV platform have been powered by HTML, but since the platform is starting to support new techniques it might be time to change the focus. HTML is a good choice for interface development because of its high level and platform independence; however, when performance is critical and the requirements are high HTML can impose serious restrictions. WebGL is a technology released in 2011 that brings a low-level graphics API to the web. The API allows for development of advanced 3D graphics and visual effects that were impossible or impractical in the HTML world. The problem is that the hassle of using pure WebGL is in most cases too big to overcome. In this thesis a proof-of-concept was developed to investigate the issues and limitations of WebGL. A conclusion was made that even though the performance was not as good as expected it might still be viable for use in some settings.

Author Keywords
WebGL; TV; SmartTV; Three.js; Set-Top Box; HTML5; jQuery

INTRODUCTION
The market of internet connected home entertainment devices is growing rapidly. The Smart TV is according to most sources destined to the same success as the Smartphone had a couple of years ago [21, 22]. As the hardware of Smart TVs and Set-Top boxes converge with that of gaming consoles the opportunities will become enormous.

With the ever increasing development in the field of TV and home entertainment the demand for fast and advanced graphics and modern user interfaces is greater than ever. The interfaces of today are written using HTML but with WebGL a new degree of freedom and opportunities is introduced to the TV platform. Cheap graphical effects and a new dimension can improve the user experience greatly.

Purpose
The objective of this thesis is to examine the suitability for WebGL as a foundation when developing typical TV centric user interfaces, regarding both performance and ease of use.

Research questions
Two questions have been phrased to guide this project. The focus is divided into two different areas; examination of the platform dependence and analysis of the development issues.

How prepared are today’s TV-platforms for running user interfaces based on WebGL with regards to performance relative to HTML5? - The performance and feel of a user interface is an essential part and it is important to measure and profile WebGL to find out if a gain can be achieved compared to HTML.

Which development problems arise when developing WebGL based user interfaces? - Since WebGL is a low-level API that is not specifically designed for creating user interfaces, there is a possibility that the lack of abstractions will deem it infeasible to use in this manner. It is important to find out what parts of the development process differ significantly from the standard HTML way and what strategies can be employed to simplify the process. A thorough examination of how well WebGL is supported on the different TV platforms is needed to ensure that it can be used without major issues.

Limitations
The performance evaluation performed was purely visual and took into account lag and responsiveness. The JavaScript library Stats.js was used to get a rough indication on frame rates. All measurements were taken with a grain of salt because of the uncertain nature of measuring frame rates.

A library that adds functions to facilitate 3D development will be used; the library chosen was Three.js.

Background
This thesis was conducted in collaboration with Systemagic AB which is a company in the IPTV business stationed in Linköping.

THEORY
This chapter goes through the techniques used in this thesis. It also takes up performance optimisations and defines what the TV platform is.

HTML5
HTML5 is the current version of HTML (HyperText Markup Language) and is the language that together with JavaScript and CSS makes up most of the web pages today. The main part of a web page is the DOM (Document Object Model) where the layout of the page is stored. The DOM is a tree-like structure where the nodes are HTML Elements.

HTML and 3D
In the recent years there has been a step towards isolating parts of web pages and using hardware acceleration on them. With the introduction of CSS3 things like scaling, rotating and translating can be done on the graphics hardware and are hence very fast. A technique called CSS3D has also been
added to most modern browsers which makes it possible to animate pages in 3D and use perspectives[11].

Even though CSS3D allows for - potentially hardware accelerated - 3D graphics in normal HTML, it is not necessarily a good idea. The DOM was not designed with 3D geometries in mind and developing complex applications in 3D is neither intuitive nor easy.

**WebKit and the TV platform**

WebKit is a rendering engine for HTML originally forked from the KHTML engine and developed by Apple but currently open sourced under the LGPL and BSD licences. Many of the devices for the TV platform use web browsers based on WebKit the for their applications and system user interfaces. WebKit is the biggest actor in the browser market and is currently in majority[32] as it is used by big actors such as Apple Safari and in derivative forms in Google Chrome and Opera Next [31, 4, 17].

**WebGL**

WebGL is a web standard maintained by the Khronos Group aimed at bringing low-level graphics to the browser. It is an API closely related to the OpenGL ES 2.0 specification and it exposes the graphics hardware for use on ordinary web pages using the canvas HTML element [26]. WebGL enables so called hardware acceleration for graphics which means that the intensive calculations associated with rendering can be run on the GPU (Graphics Processing Unit) instead of on the CPU (Central Processing Unit). The GPU is heavily optimised for these kinds of calculations which means that the work can be done faster and more efficiently. Because of the parallel nature of many graphics calculations, such as calculating the colour of each pixel on the screen, the performance can theoretically be much better on the GPU which often utilises many processing cores. Even though WebGL allows for 3D graphics it is not a 3D library per se, as the developer needs to supply all the 3D maths. This makes it very hard to do 3D graphics as a beginner or if you have no or little knowledge of vector algebra. When doing user interface development the hassle of having to bother with the maths is a big obstacle that cannot be present.

**Shaders**

A big part of using the WebGL API is using shaders. A shader is a program that is written in a shader language and runs, typically in parallel, on the graphics hardware. The shaders used with WebGL are written in the OpenGL Shading Language, GLSL. GLSL is a C-like language with extra features suitable for certain graphics calculations, things like vector calculations, e.g. dot- and cross product, and colour mixing are very fast and can often be executed in a single cycle.

In WebGL there are two types of shaders: the vertex shader and the fragment shader. The vertex shader acts on a set of vertex attributes and outputs new vertices and attributes [29]. The fragment shader, sometimes called the pixel shader, works on a rasterised image and computes colours and depth values [28]. Both of these shaders are part of the rendering pipeline which is executed every frame. The rendering pipeline consists of a number of steps that are performed to produce the final image. The vertex and fragment shaders are the only programmable parts of the pipeline; other graphic APIs have additional programmable steps for vertex processing such as geometry or tessellation shaders but these are not supported by WebGL. The last step of the rendering pipeline is the frame buffer, this buffer contains the data that is outputted to the screen. An illustration of the WebGL rendering pipeline can be seen in Figure 1.

![Figure 1. The different steps in the rendering process.](image)

**Performance Optimisations**

When the performance of a WebGL application is crucial there are some things that are important to consider. Sometimes a naïve approach can cause serious bottlenecks in the program, especially on embedded devices with less memory than a desktop computer. A series of strategies has to be employed to reduce the issues. One important step is to reduce so called state changes and draw calls. These operations will place a heavy load on the CPU as they often include the copying of data from the memory to the GPU. The data will often be images or vertex data. This copying will lead to a heavy load on the CPU and an idle GPU. The strategies used to minimise the state changes and draw calls are often used to a great extent in game development [2, 1].

**Three.js**

Three.js is a library built on top of WebGL that provides higher level abstractions for 3D development [24]. Three.js is one of the most popular 3D libraries for JavaScript. The library is very easy to get up and running and whilst it provides a higher level of abstraction it still lets the developer use lower level elements of WebGL like shaders. One important feature of Three.js is that it includes a straightforward way of organising the objects in a scene graph, this is much like the way
HTML works using the DOM and can be helpful when developing user interfaces. The scene is a tree-like structure where the nodes are instances of the THREE.Object3D object and can be things like meshes or cameras. The scene itself is also an instance of the THREE.Object3D object.

Three.js contains a substantial collection of common 3D geometries like the cube, the plane or the sphere. It also includes a lot of utilities for vector calculations to help structuring the code in a more efficient way and aid the developer with the sometimes cumbersome maths.

**jQuery**

jQuery is one of the most used JavaScript libraries for manipulating the DOM. It is currently used by over half of the top one million web pages and therefore familiar to most JavaScript developers [12]. jQuery works by wrapping JavaScript objects in so called jQuery objects, it then allows for manipulation by chaining methods on these collections. Most times these objects have HTMLElement in their prototype chain and are therefore HTML elements; however, this is not always the case as jQuery has the ability to wrap any JavaScript type. One main feature of jQuery is its interface to the DOM, the power of this interface lies in the jQuery constructor object often called $ or jQuery. This constructor function lets the developer select elements from the DOM by referencing them by class (.) or id (#). The matching elements are then returned as a jQuery collection ready to be operated upon.

**Extensibility of jQuery**

The jQuery library is built in an extension-friendly manner by letting a developer add functions to either the main object or to the jQuery collection object. Utility functions are added to the jQuery object by using the $.extend() function, methods are added by providing them to the $.fn.extend() function. Because jQuery is pure JavaScript it is very permissive, it is therefore important to follow best practices when creating extensions aimed for production [13].

**The TV platform**

In comparison with the desktop platform, the TV platform has different requirements on its user interfaces. A typical TV interface uses bigger elements, less text, more images and no cursor interaction. The restrictions on these kind of interfaces opens up for different ways to develop them. The TV platform is sometimes called the 10ft (ten feet) environment because it is usually designed to be viewed from a distance of about ten feet. In this thesis this kind of inspiration is derived from the TV development guidelines and definitions from Google [27].

**Set-Top Box**

A Set-Top box, or STB for short, is a device connected to a TV that is used to receive and play media. The current generation of Set-Top boxes are often internet connected and are typically running interfaces that are written in HTML.

**SmartTV**

The SmartTV is a TV that incorporates an internet connected computer able to download and run application. The designs of the user interfaces are often similar to those of the Set-Top boxes.

**Related Work**

The use of a scene graph in user interface framework design can be traced far back. In a study from the year 2000 a framework for 2D graphics was designed using a scene graph like the one used in 3D libraries such as Three.js [3]. The approach is similar to the method used in this thesis. One concern in the study is that a user interface built with a scene graph can be difficult to maintain when the number of nodes increase. A solution is presented in this thesis, by developing a jQuery extension to manage complexity the maintainability can be retained.

Another similar project is the tQuery project [25]. TQuery is an API built on top of Three.js that aims to mimic the behaviour of jQuery. The difference between part of the product in this thesis and tQuery is that the plugin developed here is an extension to jQuery and can enjoy all the benefits of other jQuery extensions and jQuery utilities. Because of the modular design the plugin can easily be extended further at low cost.

WebGL is already in use today for the purpose of user interface development and one notable example is the PlayStation 4 gaming console which has parts of its system user interface using WebGL [18].

Other than interface development, WebGL have been widely used in game development and data visualisation, one notable example is Google Maps which has a version built in WebGL. Many sources recommend the usage of the library Three.js. Some raise the concern that the low level of WebGL and sometimes even Three.js can be hard to grasp for novice 3D developers [15, 9].

**METHOD**

This section outlines the work and is divided into two parts to reflect the research questions. The first part consists of a description of the proof-of-concept that served as a basis for the examination of WebGL, it also includes information about the testing platforms and an additional low-level test that was performed to narrow the analysis of potential bottlenecks in WebGL. The second part describes how development issues were recorded and discussed through the use of a development diary.

**Proof-of-concept**

The main part of the work included the development of a WebGL proof-of-concept user interface. The proof-of-concept was based on an HTML5 example developed by Systemagic that consisted of a typical TV user interface with a grid layout. The example included images, text, transitions, animations and selection. The design of the example is shown in figure 2-3.

The proof-of-concept aimed to look and feel like the example in most aspects but some discrepancies occurred when the divergence between HTML and WebGL was big.

**Development Phases**
The proof-of-concept was developed in three iterations with weekly meetings with Systemagic.

The first consisted of getting the development environment up and running with the Set-Top box and the other testing devices. It also included agreeing on what features the example and proof-of-concept should include. Much of the time this first week was spent learning the basics of WebGL, Three.js and 3D programming in general.

During the second iteration some important design decisions regarding the architecture of the code were made. The goal at the time was to create an API that worked similarly to jQuery. Focus was put into implementing a general structure of the objects which allowed for easy manipulation and method chaining. A great deal of time was also spent testing many of the features of WebGL and Three.js on the different platforms.

During the meeting between the second and third iterations a decision was made to investigate jQuery further to see if it was possible to use it to build the proof-of-concept. After some research the API was rewritten as a plugin for jQuery.

Using jQuery

The choice of using jQuery came quite naturally as it is a very convenient and commonly used library for dealing with web based user interfaces.

One notable difference when using jQuery to select Three.js objects instead of working with the DOM is that there is no default data structure that holds all user interface nodes. Normal HTML has the DOM for this and jQuery can safely assume that you want to select from it, this assumption cannot be made when working with Three.js. In Three.js the developer is working with something called scenes, or scene graphs, and it is possible to create multiple scenes. This problem can be solved by creating a function that returns a custom jQuery constructor for a given scene. This allows for multiple scenes in the same project; however, one jQuery constructor must be created for each scene.

Extensions to Three.js

To make user interface development easier, some important extensions to the Three.js library were made. Apart from adding CSS-like selectors to improve the interface to jQuery, some custom 3D objects were also added. All objects can be easily extended with new features using JavaScript’s powerful prototypal inheritance.

THREE.Card

The most important object is the THREE.Card, which is a counterpart to the commonly used div in HTML. The THREE.Card is made up of two planes, one front and one back, and can be used for most rectangular UI elements. Much like an ordinary div the THREE.Card can have a colour or a background image. The THREE.Card is easily extendible and allows for other user interface elements to be created based upon it.

THREE.VideoTexture

This object allows for features that were not present in the example provided by Systemagic but were considered too important to leave unexamined. Because of the focus and importance of video on the TV-platform a special kind of texture object was developed to let the developer use video streams as textures inside a 3D scene. This makes it possible to easily play video on the different user interface elements and essentially works like a HTML video tag that can be used inside the Three.js scene. One interesting by-product of this is that a custom fragment shader can be applied to a video playing on a 3D object to produce graphical effects like blur or colour inversion.

Code examples

Below follows some code examples to show some of the behaviour and features of the jQuery extension.

Create a constructor for a new THREE.Scene.

```javascript
var $s = $.three(new THREE.Scene());
```

This function can then be used to select Three.js objects on the scene. Assuming that we have a set of UI elements arranged in rows we can select them by their classes.

```javascript
var row1 = $s('.row1');
```

Many jQuery features can be used such as promises and animations. All methods belonging to the plug-in will start with the letter t.

```javascript
row1.tAnimate({
  color:'#f00'
}).then(function(){
  row1.tFadeOut();
});
```
One important feature is the use of styles. The tStyle method can be used to change most of the attributes of the Three.js objects.

```javascript
$s('.menu').tStyle({
  image: '/icon.png',
  colorFilter: 'blue'
})
```

```javascript
$s('#textarea').tStyle({
  fontStyle: 'Lobster',
  fontColor: '#111'
})
```

**Low-level WebGL Testing**

To get more information about the behaviour of WebGL the canvas tools of the current version of Firefox Aurora were used. These tools will soon be available in the standard Firefox. The tools include a view over the different WebGL API calls that are executed every frame. In this case the tools were used to inspect the number of GL draw calls and state changes that occurred during rendering.

An additional experiment was performed to further test WebGL and discover where potential performance problems and bottlenecks might occur. The experiment was constructed by diving deeper into WebGL and prototyping a low-level example that used shared textures and a shared geometry for a scene with similar complexity to the proof-of-concept. With this it was possible to get a better understanding if potential problems were due to state changes in the graphics hardware or something else. Instead of using different buffers for the different shapes they were combined into a single buffer that used a texture atlas containing the textures that were previously separated into individual files. With this line of action the entire scene could be rendered by issuing a single GL draw call. The testing example is visualised in Figure 4, the size of the grid is 30 rows by 50 columns yielding a total of 1500 rectangles.

![Figure 4. The test consisted of an animated grid with multiple tiles.](image)

**Multi-platform Evaluation**

To get a good understanding of the potential performance problems when using WebGL, the proof-of-concept was tested and evaluated on four different platforms. The selection was made to get a broad picture of the potential problems.

1 - MacBook Pro 13”, OS X 10.6.8, Early 2011

This device served as the development machine and was used as a reference where a high frame rate was guaranteed.

2 - Motorola VIP 2952 Set-Top Box, 2012

This device is a typical STB capable of playing high definition video (1080p). The CPU is a BCM7241 and for the graphics there is a Broadcom Videocore IV GPU. The CPU is claimed to perform 3000 DMIPS (Dhrystone MIPS) [6]. The browser in the STB is made by Ekioh which is one of the leading browser vendors for the TV market. In the past much of the focus of Ekioh has been to develop their SVG (Scalable Vector Graphics) browser [8], but support for WebGL was recently introduced.

3 - Samsung Galaxy SII Smartphone, 2011

Even though the Smartphone is not a typical TV-platform it does have similar hardware to the other two TV-platforms. The CPU is a 1.2GHz dual core ARM Cortex-A9 and the graphics chip is a Mali-400 MP GPU which is slightly more powerful than the Videocore IV used in the Motorola STB. Regarding the performance of the processor it is claimed to be 2.5 DMIPS/GHz per core, this is equivalent to 3000 DMIPS per core in this case [5]. This means that the performance is theoretically the same as the BCM7241 used in the Motorola STB. The interesting part of measuring performance on an Android Smartphone and comparing it to the STB is that we get indications on whether performance problems are due to hardware or software. The browser in the Smartphone is backed by Google’s Android team and powered by one of the fastest JavaScript engines available. The WebGL implementation in the Android browser can be assumed to be more mature than in the Ekioh browser used in the STB.

4 - Samsung 32” SmartTV F4515, 2014

To cover the more of the spectrum the proof of concept was also tested on a SmartTV. The hardware specifications for this device are mostly unknown; the processor is claimed to have four cores but no information is available about the graphics hardware. It can be assumed though, that the hardware is somewhat similar to the STB and the Smartphone.

**Development Issues**

To measure potential problems during the implementation a development diary was kept. All issues were recorded and analysed in the diary.

The design of the diary followed some of the guidelines laid out in a study from the *Journal of Personnel Psychology* [16]. The study concludes (p. 85–86) that to promote answering frequency and quality, the time of the questionnaire should not exceed 5–7 minutes and not consist of more that five standardised questions. In the case of this thesis the diary was problem oriented and was divided into two parts: one consisted of questions that were to be answered at the end of
every workday and one more analytical part where the issues were discussed in more depth and solutions were proposed. The second part was not necessarily to be used every day but as far as time allowed, though the objective was to discuss all problems from part one at some time.

The questionnaire
The following five questions were used:
1. What have you been working on today?
2. Have you encountered any compatibility issues?
3. Have you encountered any performance issues?
4. Which were the two biggest problems?
5. Do you have any simple solutions to the problems?

Analysis
The second part of the diary was not discussed in the study mentioned above, but was introduced because of the need to analyse the problems to find out if they would impact the development in a major way. With the addition of this second part the general pattern of the diary closely resembles a commonly practised development method, namely Scrum [20]. The daily questionnaire functions like the daily scrum meetings and the analytical part is designed to resemble the retrospective.

All the issues that surfaced in the diary that were not trivial to solve were included in the results.

RESULT
In this chapter the results from the testing and the development are presented. The first part includes results from the testing of the proof-of-concept and the HTML example. It also consists of the results from the testing of the low-level prototype. The second part presents the development issues discussed in the development diary.

Low-level WebGL Testing
Using the Aurora canvas tools it was discovered that the Three.js implementation of the user interface used a high number of GL draw calls. It was also established that many of the draws made use of different textures and different geometries. Changing textures and geometries many times for every frame to be drawn is a very inefficient way doing WebGL. Though the approach worked fine on some devices, others might not have the same optimisations regarding texture and geometry changes; this can cause seemingly similar platforms to behave very differently and have vastly different bottlenecks.

Needless to say, there was a huge performance gain when testing the low-level prototype. When running the new prototype, shown in figure 4, the STB was able to display up to around two thousand animated tiles without a significant drop in performance, this compared to the Three.js implementation which introduced a severe performance drop whilst consisting of merely about one hundred equivalent objects. The SmartTV though, did not experience any performance improvement at all. Even a test with a single square and a small texture drawn with a single draw call ran too slow.

Multi-platform Evaluation

MacBook Pro
The Mac produced satisfactory results and could serve as a reference for the other measurements. The frame rate on this device was practically fixed at around 60 frames per second. This is the top speed of most browsers.

Motorola VIP 2952
The testing of the HTML example on the Motorola STB gave satisfactory results; even though there was slight lag in the animations it was perceived as responsive and there were no major performance issues.

The equivalent WebGL implementation suffered from major loss in frame rate, and was perceived as slow. The responsiveness was not too far behind but lag could clearly be noticed. Important to note is that even an empty scene produced a result with a frame rate of about 25 frames per second, this is lower than acceptable and a frame rate over 30 frames per second is preferred.

Samsung Galaxy SII
Both the HTML example and the WebGL implementation produced satisfactory results; there was almost no lag and only a slight drop in frame rate compared to the reference device.

Samsung SmartTV F4515
The WebGL performance of this device was by far the worst of all the testing platforms. No measure could bring the frame rate over 2 frames per second.

Development Issues
The diary succeeded in bringing attention to many of the development issues present when working with WebGL and Three.js.

There are many desirable features that are currently missing or partly missing in some implementations of WebGL. These are things that often exist when working with HTML and will severely affect the appearance of a WebGL user interface. Some of the missing features are specified as WebGL extensions and the implementation is therefore optional for the browser vendors.

Anti-Aliasing
Anti-aliasing is a technique that is commonly used in computer graphics to combat aliasing and smooth out edges.

A common problem with WebGL is that native anti-alias support is quite rare among low-end devices. Some laptops and many embedded devices like STBs and SmartTVs lack support and this severely limits the opportunities to develop nice looking interfaces. The WebGL specification states that the antialias attribute is not a compulsory part of the API [30]. One of the limitations working without anti-aliasing is that all UI elements need to be orthogonal to the camera to avoid aliasing. Elements such as triangles, circles and their 3D counterparts will also become hard to use due to their diagonal or curved lines. The difference can be seen in figures 5-6.
There are ways to partially solve the problem of aliasing with an additional step of post processing. There are approximation algorithms such as FXAA [14], PXAA and TPXAA [19] which can be applied as fragment shaders as a last step in the rendering process. These algorithms essentially works by detecting sharp edges and smoothing them out and are frequently used in computer games because of their light weight. One problem with FXAA that PXAA and TPXAA tries to solve is that it works poorly on 2D textures with text. The author of FXAA advices against rendering HUD (Heads Up Display) and user interface elements of games before applying FXAA [14]. There is an implementation of FXAA in Three.js, but it was considered not usable because it made longer sharp lines wavy and produced texture artefacts on the user interface elements. Compared to native anti-aliasing these approximation algorithms produce an inferior result for the task at hand.

**Texture Problems**

Due to limitations in the graphics hardware of many embedded devices some useful techniques such as anisotropic filtering can be absent. Anisotropic filtering is a technique that aims to reduce texture artefacts on objects viewed from a large angle.

**Video Streams**

One common problem among embedded devices is that even though there is a HTML tag for video, it does not work in the same way as desktop browsers. In the case of the Motorola STB the video tag *punches a hole* in the browser and renders the video with a different program, which is much like the way that Adobe Flash or other browser plugins work. Because of this the video stream is not accessible by JavaScript and cannot be used as a source for textures.

**Sizes and Perspective**

One thing that is important to remember when working in 3D is that sizes are not always what one might expect. Even though sizes are set using pixels, the objects need not necessarily be displayed with that size when viewed on the screen. The perspective is clearly illustrated in figure 7. This can cause some problems when working with images and text.

**Rendering Text**

One huge problem when it comes to user interfaces is that there is no obvious way to render text in a consistent and platform-independent way. The only way to display 2D text in a 3D scene is to include it as a texture. Using any arbitrary image with some text on it is not a problem; however, when you want to include UI elements with dynamic text things will become more difficult.

To combat the problem, an extension to the card object was created called a TextCard, which enables UI elements with dynamic text. The method used was to draw the text on a separate canvas using a 2D context instead of a WebGL context. This canvas was then used to create a new texture for the THREE.TextCard.

This is a method that works fairly well but there are some problems and limitations. The first problem is that since the text is rasterised beforehand it will become blurry once viewed from a different distance or angle than intended due to texture filtering. In some WebGL implementations that use superior texture filtering algorithms, the text could be rasterised at twice the size and then scaled down; this would allow for sharper text at the cost of memory and performance. The second problem is that the canvas 2D API for text is very
limited and all the logic for things like word wrapping and line endings needs to be supplied manually.

The method mentioned above might be quite straightforward, but it is also naïve. The approach will require new textures to be created every time a new text is added or changed and there is no way to share textures between objects with the same text. This will impact the performance in a negative way because texture changes is a fairly expensive operation on the GPU. A better approach would be to render the entire alphabet to a larger shared texture and then use UV mapping to render the letters to smaller rectangles. This would let us share a single texture between all the text areas in the application given that they use the same font.

There are some ways to combat the blurriness of text and vector illustrations, most notably a method developed by the gaming company Valve and currently used in their game engine Source. The method uses a distance field instead of a bitmap to represent the font. Each piece of the field contains the distance from the current texel to the edge of the character. This lets the fragment shader calculate the sharp edges and crisp text can be achieved with a smaller resolution of the texture, even at large magnification. The method also allows for essentially free effects to be applied to the text such as glow or outline, which can be very desirable in a user interface.

DISCUSSION

In this chapter the results of the performance tests are firstly discussed and analysed. After that there is a part about the development issues experienced and some discussion about the proposed solutions. The third part is a discussion about the method used in this thesis. The final part is about the ethical implications of a transition from HTML to WebGL.

Performance

The clear division in performance between the three devices was surprising given that they utilise very similar hardware. It is hard to pinpoint the exact location of the bottleneck(s) though one possible hypothesis is that the maturity of the browsers is a big factor.

Motorola VIP 2952

Ekioh does not explicitly advertise WebGL support on their website and that can serve as an indication that the released implementation is not entirely ready for production [7]. Another possible explanation is that the the Set-Top box is not running code ready production but is of a preview build. It is claimed in the boot log that OpenGL ES performance could be significantly impacted.

Samsung Galaxy SII

The results of the SmartPhone tests indicated that the bottleneck(s) of the other devices probably reside in the system or browser software layer.

Samsung SmartTV F4515

This device produced the most surprising results. Even though the TV potentially uses superior hardware than both of the other two devices it produced the least satisfactory results. The WebGL support was supposedly introduced in the 2013 generation of the Samsung SmartTVs and still the result was terrible. This strongly indicates that the problem lies in the browser software. The system user interface of the TV ran in a smooth fashion and was probably 3D accelerated with OpenGL ES, but the support for WebGL was apparently not mature enough.

Low-level Test

The low-level experiment suggests that there may, in some cases, be a considerable overhead of using libraries such as Three.js. Typical user interfaces are generally composed of many smaller objects such as buttons, text areas and images and if all these objects cause, a draw call of their own problems of performance can occur. Possible solutions include using texture atlases and shared geometries, while a more fundamental solution is to try replacing Three.js with something more low-level or disregarding libraries completely. This would severely affect the development speed and ease.

Method

Proof-of-concept

Because of WebGL’s low-level nature it would probably not be feasible to use it directly for user interface development; however, with a jQuery extension for 3D development present it could be possible to get rid of the underlying layer of abstraction. Even though this would require a new layer in place instead of Three.js it could still be beneficial.

The experiences gained in this thesis suggests that a framework that is built around the concept of shared buffers could be developed to improve the performance of these kinds of user interfaces. This framework would probably have to be more integrated with the jQuery extension. It would require a great deal of research and experience in graphics programming to be able to do this in an efficient and developer friendly way.

Testing

Because performance measurements are very hard to evaluate in a reliable and confident way when there are multiple platforms involved. Because of this, the measurements were not done in absolute numbers for this thesis. The expected result was binary and only aimed to answer if there was a performance problem or not. To draw better conclusions about the performance issues further testing is necessary. A more elaborate method of profiling and benchmarking the devices is needed to identify and rule out possible causes for the degradations in performance.

Diary

The development diary is a very useful tool for determining and quantifying problems; however, there are some things that are worth considering before conducting a problem oriented diary study. The experiences gained in this thesis and through the literature [16, 20] can be formalised in three recommendations.

The first important thing to consider is the sample size. In this thesis the sample size was very small, one person, and this can reduce the value of the conclusions drawn. In the study on diary studies mentioned earlier [16], the sample size
is discussed and the general idea is that with a greater sample size it is much easier to draw general conclusions. The sample size is indeed important, but not all types of problems are affected by the sample size. Objective issues are often possible to detect even if there is only a single person participating in the study. The hard part is the subjective issues, the issues related to personal experience and more dependant on the person asked. In a case where the issues are of a subjective nature it would probably not be wise to rely on such a small sample.

The second thing worth noting is that a diary study designed to resemble the Scrum method can be better at discovering issues than a simple diary. With a second analytical part where solutions are discussed the utility of the study can be greatly improved. In the case of multiple participants the analysis should be done together and the questionnaire individually.

The third recommendation is to take measures to maximise answering frequency. This is discussed in the study and the experience gained in this thesis suggests that this is very important. The most important thing to note is that it is crucial not to confuse the two diary parts, if the questionnaire becomes to heavy the answering frequency will go down.

Ethical implications
The transition from an HTML based to a WebGL based user interface framework may raise some environmental concerns. The practice of rendering using WebGL is different from the way HTML works and could lead to an increased power consumption for TVs and Set-Top boxes. The TV-platform currently houses a large market and the increase in power consumption could be very significant.

A measurement of power consumption was made to compare the proof-of-concept to the HTML example. A power meter (Esic PM498) was used together with a laptop to get a rough indication whether there was an increase. It was found that the HTML example had an idle power consumption between 15-20 W and the WebGL proof-of-concept stabilised at 25-30 W, this in an increase of about 60 percent. The increase can be assumed to be somewhat less significant on embedded devices because they often utilise passive cooling in contrast with the laptop; however, the measurement still clearly suggests that power consumption is a significant factor to consider when transitioning to WebGL.

To put the numbers in perspective, one of the biggest actors on the Swedish market had close to 1.5 million IPTV customers in northern Europe 2012 [23]. With a market at this size a transition towards WebGL could have negative consequences for the environment.

CONCLUSIONS
Even though there were some serious issues regarding compatibility and performance, WebGL might still be a possible way forward to broaden the possibilities for the future of TV. With a simple API, such as the jQuery extension developed as a part of this thesis, the issues related to layout and design are quite easy to overcome. The developer will still have to know about the structure of Three.js and to do advanced graphical effects shaders have to be learnt. Some types of applications will demand a lower level of the development to be able to fully take advantage of the performance but in general it should be sufficient with Three.js plus jQuery.

The compatibility and performance issues will probably decrease as the browser software matures further and at the point when the software catches up the hardware will be even faster.

REFERENCES


19. PXAA and TPXAA.


22. Strategy Analytics on SmartTV Market.

23. Telia TV statistics.


25. tQuery Website.


27. Google TV Development Guidelines.


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Jonathan Karlsson