HTML5 for 2D Games: Implementation and Comparative Evaluation of a Rendering Framework

Andreas C. Kasch

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2014

© Copyright 2014 Andreas C. Kasch

This work is published under the conditions of the *Creative Commons License Attribution–NonCommercial–NoDerivatives* (CC BY-NC-ND)—see http://creativecommons.org/licenses/by-nc-nd/3.0/.

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, September 29, 2014

Andreas C. Kasch

Contents

Abstract

Creating games is a dream of many teenagers, but there are not many simple tools that allow them to implement their ideas without learning a programming language. The *ZIEGE* project tries to achieve this by providing a game editor that can be used by teenagers without any previous knowledge of programming. It should be possible to run it in the web browser without any plug-ins so it is not necessary to install any software, which could be difficult in schools. To implement this game editor, a rendering framework is required that is capable of drawing permanently changing scenes as well as saving and restoring them at any time. This thesis introduces the H5R framework prototype, which realizes these requirements. The structure and usage of this framework is demonstrated; moreover the implementation is described as well as the decisions and evaluations that lead to this implementation. Finally the framework is evaluated and compared to a number of existing frameworks.

Kurzfassung

Eigene Spiele zu erstellen ist ein Traum vieler Jugendlicher. Allerdings gibt es nur wenige Programme, die es ihnen ermöglichen ihre Ideen zu realisieren, ohne dafür eine Programmiersprache zu erlernen. Das *ZIEGE* Projekt versucht dies zu ermöglichen, indem den Jugendlichen ein Spiele-Editor geboten wird, der ohne vorherige Programmier-Kenntnisse genutzt werden kann. Dieser sollte im Webbrowser ohne PlugIns funktionieren, damit keine Software installiert werden muss – was vor allem an Schulen schwierig sein kann. Um diesen Spiele-Editor umzusetzen wird ein Framework zur Grafik-Ausgabe benötigt, welches dazu in der Lage ist, sich ständig ändernde Szenen zu zeichnen, und diese jederzeit zu speichern und wiederherzustellen. Diese Arbeit stellt einen Prototypen des H5R Frameworks vor, der diese Anforderungen erfüllt. Die Struktur und Benutzung des Frameworks wird erläutert, außerdem werden die Implementierung sowie die Entscheidungen und Beurteilungen, die zu dieser Implementierung führten, erklärt. Schließlich wird das Framework bewertet und mit bestehenden Frameworks verglichen.

Chapter 1

Introduction

The concept of 2D computer games has been around since the second half of the 20th century. There are countless numbers of games and frameworks to create such games. Although the hardware has been capable of rendering complex and sophisticated 3D scenes for years, 2D games are certainly not extinct. Especially the development of smartphones—which did not have such good hardware at first—brought back lots of 2D games. Even with really simple game ideas, such as *Flappy Bird*, some developers have been very successful. Indie Game developers discovered the potential of 2D games as well, mainly because 2D graphics are less time consuming and expensive than 3D graphics, when a certain amount of quality is desired. Indie games such as *World of Goo* and *Terrania* were very successful even though they were not produced by a major publisher. Another platform for 2D games is the Internet. Especially when no download is required, 2D casual games are popular, even though their scope is limited compared to other games. However, even 2D casual games require a lot of work and knowledge. Primarily for younger teenagers, it is their lack of knowledge that prevents them from living out their creativity and ideas for computer games.

This chapter shows the motivation for this thesis, by presenting the requirements of a rendering framework that can be used in a game editor. The research issues and contributions of the proposed open source JavaScript 2D rendering framework, called H5R framework, are briefly discussed. Finally the structure of the thesis is shown.

1.1 Motivation

The idea of letting people—especially teenagers—create their own games without too much knowledge led to the *GOAT* ("Game Online Authoring Tool") project. The aim of the project was to provide a simple game editor that would run on any computer without installing any software—so it could actually be used at schools, where teachers often do not have administrator

Figure 1.1: The first running prototype of the *GOAT* project.

rights on the computers. Thus it was implemented as a web application that does not require any browser plug-ins and that should run in every modern web browser (see fig. 1.1). A 2D rendering framework was required that complies with those requirements. The framework should also be able to handle rapidly changing scenes (so drawing levels and objects in the editor is possible in real time) and offer the possibility to save and load scenes without much effort. After trying several existing 2D rendering frameworks for the Web, it was decided to create a custom framework. This thesis proposes the prototype of an open source 2D rendering framework for HTML5 and JavaScript, called H5R framework.

The H5R framework is used in the *GOAT* project which is a part of the *ZIEGE* ("Zielgruppenorientierter internetbasierter Educational Game Editor") project. The ZIEGE project was funded by the *netidee*[1], an initiative of the *Internet Foundation Austria*.

1.2 Research Questions

The following open questions had to be answered in the course of the implementation of the H5R framework:

- Which rendering technique performs best in the modern web browsers without the use of plug-ins?
- Is the rendering performance of the H5R framework good enough for rendering complex scenes with an appropriate frame rate?

[1] https://www.netidee.at/

- Is it possible to add or remove large amounts of objects to or from the scene in real time?
- How can the scene be saved and restored without much effort?
- What optimizations could improve the rendering performance of the H5R framework?

The aim of this thesis is to answer those questions by comparing different rendering techniques and existing rendering frameworks, describing the architecture of the framework and demonstrating its usage. It is described to what extent the framework differs from existing frameworks and with which optimizations the performance has been improved.

The main contribution of this thesis is to provide a framework complying with those requirements by proposing the open source JavaScript 2D rendering framework H5R, which allows rendering of complex and rapidly changing 2D scenes, which can be saved and restored with little effort.

1.3 Thesis Structure

This thesis is organized into seven chapters, as follows: Chapter 1 illustrates the motivation and open questions that led to the implementation of the H5R framework and this thesis. In chapter 2, the state of the art in web browser gaming is shown. Some plug-ins that can be used for gaming are introduced as well as some techniques that do not require plug-ins, which are described in detail in chapter 3. These techniques are compared according to the requirements to figure out which one would work best for the *ZIEGE* project. In chapter 4, some frameworks are introduced which use some of those techniques; they are compared to each other and their strengths and weaknesses are shown. According to the insights of this chapter and the requirements, the H5R framework is implemented and described in chapter 5. The structure and the usage of the H5R framework is illustrated as well as some optimizations. In chapter 6, the H5R framework is evaluated and compared to the other frameworks introduced in this thesis. Finally a brief outlook of the H5R framework, the *ZIEGE* project, web browsers and games is given in chapter 7.

Chapter 2

State of the Art: Web Browser Gaming

Over the years, different techniques have been used in order to play games in the web browser. Most of them use plug-ins, due to performance reasons. Some formerly popular plug-ins, such as *Shockwave*, have completely disappeared, while new ones are released that offer new possibilities such as 3D graphics. In this chapter, the state of the art in web browser gaming is shown.

2.1 Browser Plug-ins for Gaming

Due to the bad hardware acceleration and performance in JavaScript execution in the past, most browser games that required a real time display of the game—opposed to games such as *OGame*[1] that do not require a game loop (see fig. 2.1)—used browser plug-ins. The most common plug-ins are *Flash*[2], *Java*[3] and *Unity*[4] due to their compatibility with most browsers and their dispersal. While *Java* and *Flash* have existed for a long time and do not aim at gaming in particular, *Unity* is a relatively new competitor (2005) and focuses mainly on gaming, especially 3D games. However, these plug-ins have to be installed before they can be used. Further more, many people have deactivated *Java* and *Flash* in their browsers due to security issues (for more information on security issues, see [21] and [23]). Nevertheless, there are countless games that use *Flash* such as *Pandemic 2*[5] (see fig. 2.2) or *Bloons Tower Defense*[6]. A *Flash* game can be included in multiple websites,

[1] http://en.ogame.gameforge.com/
[2] http://get.adobe.com/flashplayer/
[3] https://www.java.com/en/
[4] http://unity3d.com/
[5] http://www.kongregate.com/games/DarkRealmStudios/pandemic-2
[6] http://www.kongregate.com/games/Ninjakiwi/bloons-tower-defense

Figure 2.1: Some games do not require client-side scripts. *OGame* uses web forms to perform actions in the game. Thus it can be played in any browser and does not require plug-ins. Image source [12].

so there are gaming websites that collect and offer multiple flash games and even extend these games by giving the player a game independent level and achievements for playing those games, such as in the case of *Kongregate*[7]. This website also collects *Java* and *Unity* games. Although most games on the website use *Flash*, the number of *Unity* games has steadily increased over the past years. The number of *Java* games is rather low. Some popular *Java* games that run in the browser are *RuneScape*[8], *Spiral Knights*[9] and some older versions of *Minecraft*[10] (see fig. 2.3). Before *Unity* was popular, 3D games for the browser were often made with *Java* while for 2D games it was easier to use *Flash*. Hence most *Java* games for the browser are 3D. *Unity* has become very popular over the past years; some well known games are *Roadeo*[11] (see fig. 2.4), *Tiny Dice Dungeon*[12] and *Broforce*[13].

[7] http://www.kongregate.com/
[8] http://www.runescape.com/
[9] http://www.spiralknights.com/
[10] https://minecraft.net/
[11] http://www.kongregate.com/games/eventhandler/roadeo
[12] http://www.tinydicedungeon.com/
[13] http://www.broforcegame.com/

Figure 2.2: *Pandemic 2*: A popular *Flash* game that was reimplemented as a smartphone game: Plague Inc. Image source [13].

Figure 2.3: *Minecraft* is implemented in *Java*. Some older versions of it could be played in the browser as a *Java* applet. Image source [14].

2.2 Mobile Devices

On mobile devices, the possibilities for browser-based games were highly limited. *Java* applets are not supported on *Android* and *iOS*, *Flash* games

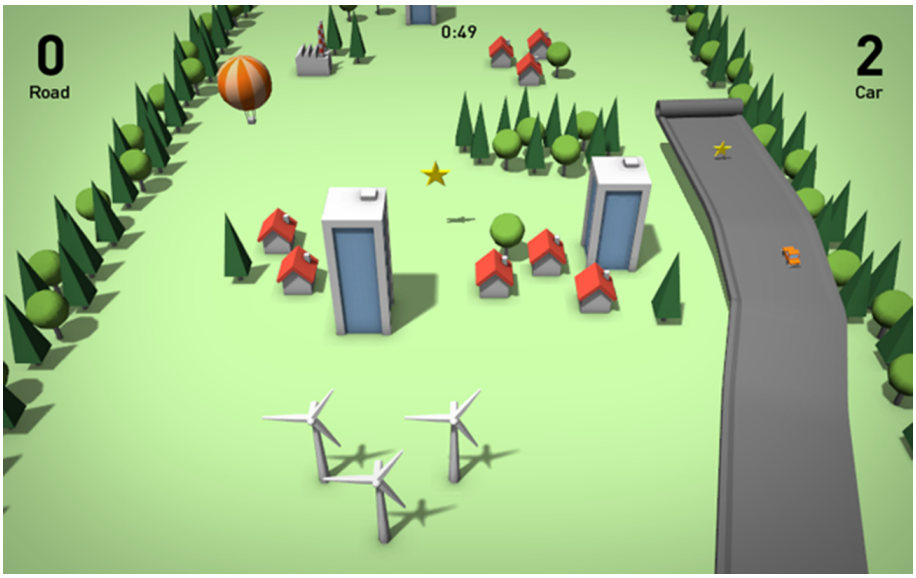

Figure 2.4: *Roadeo* was implemented with *Unity* and can be played in the web browser. Image source [15].

only partially and require additional software. Porting *Flash* or *Java* games for mobile platforms so they could be downloaded as a native application is not possible without a great deal of effort. This is an advantage of *Unity* that has a built in support for native mobile applications for *iOS*, *Android*, *Windows Phone 8* and *BlackBerry 10* as well as some modern gaming consoles[14], which makes it a very popular game engine.

2.3 SVG

Scalable vector graphics (SVG) offer many features for drawing lines, shapes, images and animations. These features can be used to render games and it does not require a browser plug-in. SVG is well supported in all major browsers and works on mobile devices as well. But SVG has one big drawback: the rendering performance is not suitable for most games. In [22], Olivier Cueilliez described it as follows:

> The only serious issue related to SVG is performance. [...] At last, SVG is an excellent technology to design browser-based game as far as timing is not important. This of course means that today real-time browser games cannot use SVG in an intensive way. HTML5 may be a better choice for such games.

[14] A full list of supported platforms in *Unity* is available at http://unity3d.com/unity/multiplatform

This result was verified by the evaluation done in this thesis (see sections 3.3 and 3.6).

2.4 New Possibilities with HTML5

HTML5, which is currently released as "Candidate Recommendation" by *W3C* (see [35], a final release would be called "Recommendation"), introduced the `canvas` element, which allows graphics rendering. With JavaScript, the contents can be changed at real time; a browser plug-in is not required. However, JavaScript could be disabled by the user. In addition to the canvas element, HTML5 also introduces semantic text elements such as `header`, `footer`, `section` and some others. New form elements were introduced to simplify, for example, the selection of a date or time. Elements for video and audio were introduced as well. The canvas can be used with two different contexts by setting different parameters in the `getContext` function:

```
1 var canvas = document.getElementById("myCanvas");
2 var context = canvas.getContext("2d"); // for the 2D context
3 // OR
4 var context = canvas.getContext("webgl"); // for the 3D (WebGL) context
```

The context can be used to draw shapes or images to the canvas. In addition to games, the canvas can be used for interactive videos, charts, animations, drawing tools, image manipulations and much more.

2.4.1 The WebGL Context

WebGL is an application programming interface (API) for advanced 3D graphics on the web. Based on OpenGL ES 2.0, it provides a similar rendering functionality, but in an HTML and JavaScript context. With WebGL, hardware-accelerated 3D graphics inside the browser can be obtained. 3D games or other advanced 3D graphics applications can be created with all the benefits that a web application has. For developers who are familiar with the OpenGL API, it is easy to learn and use WebGL; this made it popular for 3D web applications [3, Chap. 1]. Great examples can be found at *Chrome Experiments*[15] (see fig. 2.5).

Previously, the greatest problem with WebGL was browser compatibility; some major browsers did not support WebGL. However the latest versions of each major browser (*Internet Explorer*, *Chrome*, *Firefox*, *Opera* and *Safari*) support WebGL; merely some mobile browsers do not support it yet (see fig. 2.6) [24].

[15] http://www.chromeexperiments.com/webgl/

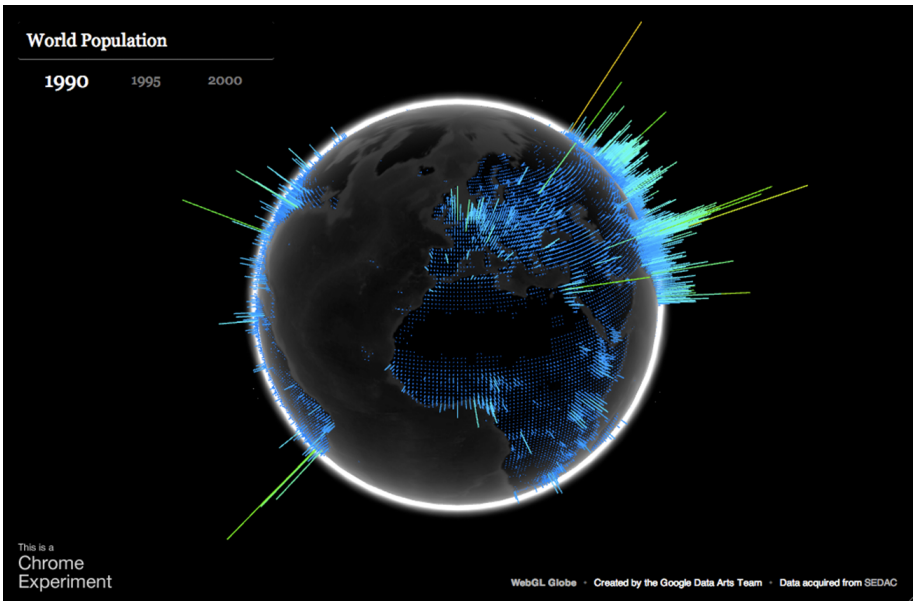

Figure 2.5: A 3D info graphic rendered with WebGL. Image source [16].

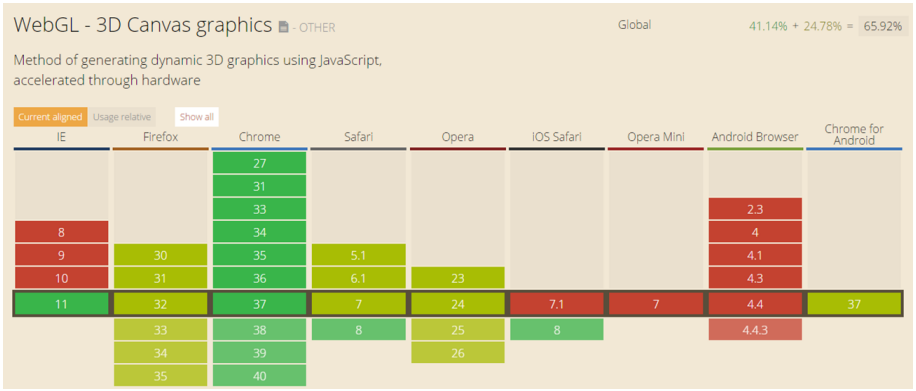

Figure 2.6: As shown, WebGL is now supported by all major non-mobile browsers. Screenshot from [24].

2.4.2 The 2D Context

The 2D context offers basic drawing functionality for lines, curves, shapes, images and text; it is supported by each major browser, and some browsers added hardware acceleration for the 2D context as well [19].

Although the scene can be transformed, rotation and scaling should be used with caution as it might influence the performance due to sub-pixel rendering (for more information see section 5.2.2).

Figure 2.7: A Quake II port running in the browser. Screenshot from [31].

2.5 Summary

HTML5 paves the way for interactive and responsive browser applications, which are more and more important with the increasing number of platforms and devices. It could supersede some browser plug-ins such as *Java* and *Flash* because of its higher platform independence.

A good demonstration of the features HTML5 offers is the *Quake II* GWT Port[16] that uses WebGL, the canvas API, the HTML5 `<audio>` elements, the local storage API and WebSockets (see fig. 2.7) [26].

With the increasing amount of possibilities in HTML, some decisions have to be made, as there are several techniques for specific tasks. A number of techniques (including the 2D and WebGL contexts) that can be used to render graphics in a web browser without the use of a plug-in are introduced and compared in the next chapter.

[16] https://code.google.com/p/quake2-gwt-port/

Chapter 3

2D Rendering Techniques in HTML

There are a variety of techniques available to render graphics in a web browser. Using browser plug-ins can improve the performance as they have better access to hardware acceleration and there are some well-established plug-ins, such as *Adobe Flash*, *Java* Applets and *Unity*, which most users have installed on their computers. Other largely unknown plug-ins, such as *Zigfu*[1] or *3DVIA*[2], have to be installed by most users before being able to see the rendered content. Plug-ins are not considered here, as they are usually limited to certain operating systems or browsers and require additional software. More interesting are techniques that do not require plug-ins as they can be used in any modern browser at any time—even on mobile devices. However JavaScript has to be enabled in the browser to render animated graphics if no plug-ins are used. When rendering 3D graphics, there is only one technique left, the WebGL context on a HTML5 canvas, but when rendering 2D graphics there are more possibilities. This thesis focuses on 2D rendering without using plug-ins. To display a 2D game, the main criteria is the frame-rate that can be achieved when rendering large amounts of sprite animations at the same time. A frame-rate of at least 15 frames per second has to be reached in the major browsers in order to display smooth animations. In this chapter, several techniques are introduced and compared to each other.

[1] http://zigfu.com/
[2] http://www.3dvia.com/studio/downloads/3dvia-player

3.1 DOM-Rendering: Using HTML-Elements for Rendering

The HTML5 DOM (document object model) is the document structure described in HTML; it can be used to display images by using `img` tags or defining `background-image` CSS attributes on `div` tags. To render interactive animated graphics, a static image is not enough, but by using absolute positioning of more than one HTML element (using CSS), an interactive scene can be built (see fig. 3.1):

```
1 #stage{
2   position: relative;
3   /* A "position: relative;" definition sets the point of origin of absolute
       positioned child elements to the upper left corner of this element. */
4 }
5 .entity{
6   position: absolute;
7   /* The "position: absolute;" definition enables the possibility to set an
       explicit position in pixels and allows overlapping of elements. The position
       (top and left attribute), the size (width and height attribute) and the
       drawing order (z-index attribute) are set for each entity separately using
       JavaScript. */
8 }
```

To display a sprite animation, JavaScript is used to replace the displayed images each frame. The `z-index` CSS attribute should also be set to handle overlapping elements if the desired drawing order does not match the order of the elements in the DOM. Accessing a single object is easy by giving it a unique `ID` HTML attribute. This approach makes it simple to handle click events on certain elements in the scene as the browser registers click events on HTML elements and no position calculations have to be done manually. It is also possible to draw some primitive objects such as rectangles and bordered rectangles without creating images for them; even rounded rectangles and circles are possible by using the `border-radius` CSS attribute. Displaying text is also possible. However, drawing a diagonal line or a custom shape is not possible without using images or vector graphics elements.

3.2 Rendering with CSS: Using Background Images

Another way to display a whole scene with HTML and CSS is given by the possibility to add several background images to just one HTML element in CSS (see fig. 3.2). A comma-separated string containing background image URLs is set as a `background-image` attribute of a large HTML element. The position of the images is set by another comma-separated string containing coordinate pairs that is set as a `background-position` attribute:

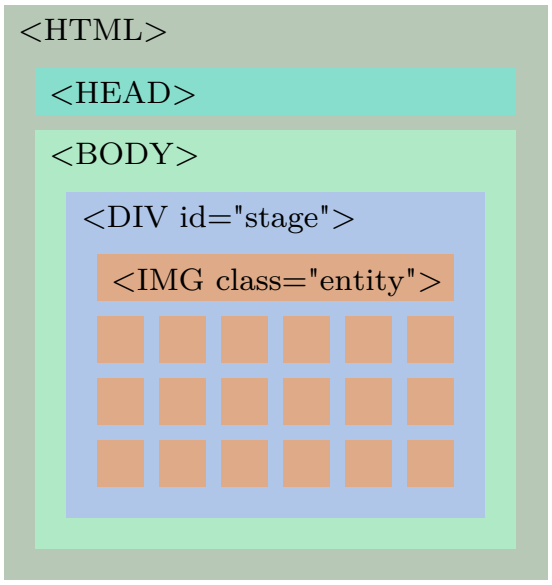

Figure 3.1: Document Object Model when rendering with HTML elements.

```
#stage{
  background-image: url("a.png"), url("b.png"), url("c.png");
  /* The number of images defined in the "background-image" attribute is only
     limited by the performance of the browser. */
  background-position: 0px 0px, 0px 16px, 0px 32px;
  /* The "background-position" attribute is used to define a x and y position
     for each image. */
  background-repeat: no-repeat;
  /* The "background-repeat: no-repeat;" definition prevents the images from
     being repeated. */
  background-size: 16px 16px;
  /* The "background-size" attribute sets the size for all images. If the images
     have different sizes, a comma separated list can be set as value with a
     width and height value pair for each image. */
}
```

The order of the images in these strings has to match the desired drawing order; a z-index cannot be set. Every single change requires all the strings to be changed and reloaded. To create sprite animations, the image URLs in the `background-image` attribute have to be changed each frame. When click events are needed on single objects, the objects at the position of the cursor have to be calculated manually. It is not possible to draw primitive objects or text; only images can be drawn.

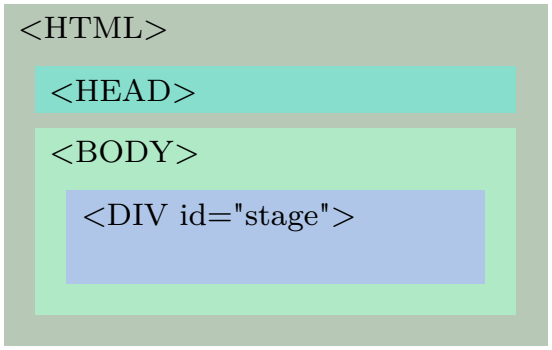

Figure 3.2: Only one element is required when rendering with background images.

3.3 Vector Graphics: SVG

Similar to DOM-Rendering, SVG rendering adds elements to HTML, but within a `svg` tag (see fig. 3.3). "The SVG format is a new XML grammar for defining scalable vector-based 2D graphics for the Web and other applications and usable as an XML Namespace" [8, 11]. SVG (Scalable Vector Graphics) is an advancement of VML (Vector Markup Language)[3] and PGML (Precision Graphics Markup Language)[4] and it has been developed by W3C since 1999 [9]. Thus only SVG is covered in this evaluation and not VML or PGML.

To draw the objects, SVG `image` tags are used, and for sprite animations the image source is exchanged. The drawing order is defined by the order the elements appear in the document, so a z-index cannot be set explicitly. To access the objects, a unique ID can be given as an attribute. When using jQuery to manipulate the document object model, the *jQuery-SVG* plug-in[5] should be used to prevent bugs (otherwise the SVG `image` element would be renamed to `img` automatically). Primitives such as rectangles, circles, and even custom shapes and paths as well as text can be drawn with SVG and the click handling is done automatically and even works for non-rectangular shapes.

3.4 The HTML5 Canvas: 2D Context

HTML5, which is the latest standard for HTML and designed to replace HTML 4 and XHTML [35, Chap. 1], introduced the `canvas` tag , which

[3] VML was submitted to the W3C in 1998 by Microsoft [9].
[4] PGML was also submitted to the W3C in 1998 by Adobe [9].
[5] http://keith-wood.name/svg.html

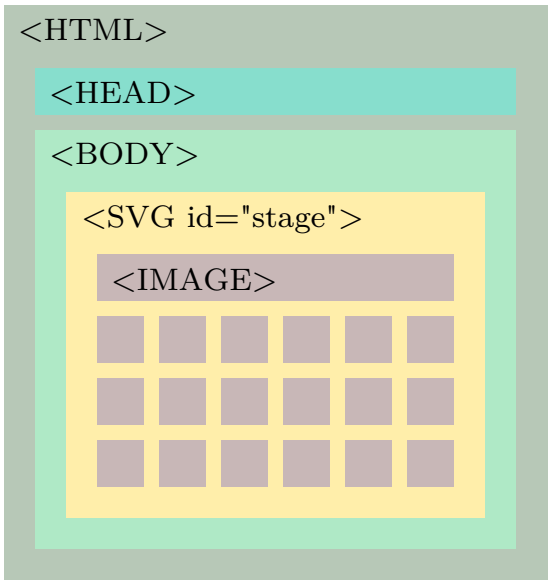

Figure 3.3: SVG rendering is similar to DOM rendering.

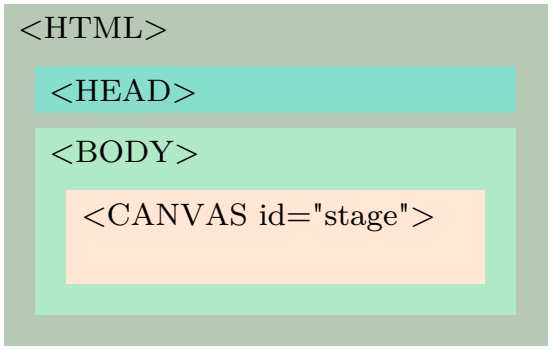

Figure 3.4: 2D and 3D canvas rendering only requires one element on the document object model.

can be used to draw graphics with JavaScript (see fig. 3.4). There are two contexts to choose from; the 2D context and the WebGL context. Using the 2D context enables several functions for drawing images, primitives, custom shapes and text on the canvas. The elements have to be drawn in the right order; a z-index cannot be set. Sprite animations can be done by just drawing a new image each frame. Click handling has to be implemented manually, but some frameworks, such as KineticJS[6] already include it.

[6] http://kineticjs.com/

3.5 WebGL, 3D Graphics in HTML5

Using the WebGL-context on a HTML5 canvas enables hardware acceleration with the graphics card, however it was not supported in every major browser until the latest releases [24]. WebGL is based on OpenGL ES (Open Graphics Library for Embedded Systems) 2.0. Without a framework it is time-consuming and work-intensive to use the WebGL context; the API is similar to OpenGL. Setting up a scene containing a white rectangle on black background requires about 100 lines of code, and additionally a vertex and fragment shader. Using a framework this can be done in about five to ten lines of code. The most popular WebGL framework (apart from *Contruct 2*, which cannot be used in this case due to its automatic code generation, for more information, see section 4.2) according to [20] is *three.js*[7]. To render sprite animated images, plane objects are generated and aligned in the 3D space. The texture of the planes is exchanged each frame to display the sprite animation. Using an orthographic camera, the z-index can be defined by moving the planes along their normal. Click handling can be easily implemented with a ray-caster in *three.js*. Simple shapes such as rectangles and circles can be added as well as text. Other shapes have to be drawn as images or built from a 3D mesh.

3.6 Comparison of these Techniques

The purpose of the comparison is to determine which technique has the best performance on most major browsers and if it is necessary to implement different techniques for different browsers. The test is designed to measure the performance in 2D games; therefore a large number of sprite animations is used. Using sprite animations is a common method to animate characters and other objects in classic 2D games. The game world can also be built from sprite animations or static images. Other games or use cases—such as effects and filters that need a lot of calculations—would require a different test. The five techniques introduced in this chapter have been tested on the same hardware, and the time required to display a fixed amount of frames of the same scene is measured to compare the performance (see fig. 3.6). The times are measured in five different browsers: *Chrome 31.0*, *Firefox 25.0.1*, *Safari 5.1.7*, *Internet Explorer 10.0.92* and *Opera 12.16*. The newest version of each browser—that was available on Windows 8 at the time of the test—was used with the exception of Opera: An older version of Opera was used, as the newer versions use Chromium, which is already covered by Chrome. These browsers were chosen as they are the most commonly used browsers according to browser usage statistics of W3Schools (see [36]) and StatCounter (see [34]) (see fig. 3.5). The scene is also rendered in different

[7] http://threejs.org/

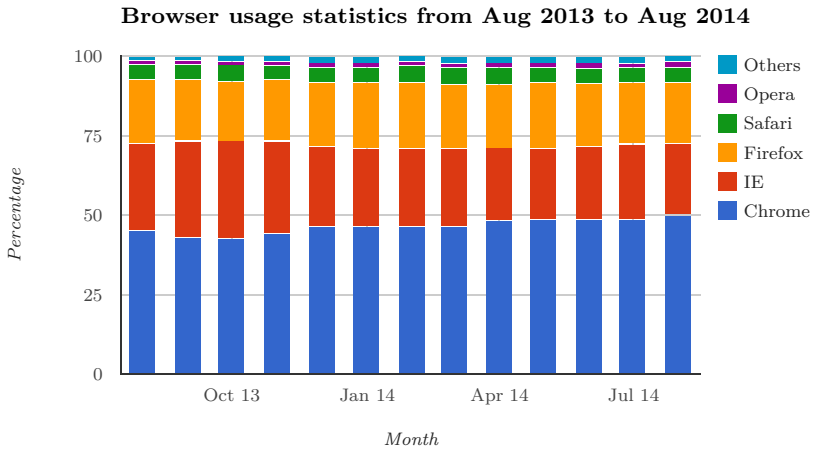

Figure 3.5: Desktop browser usage Statistics. Data source [34].

levels of complexity to see changes in the performance that are caused by the scene complexity. The performance on mobile devices was not part of the test. Using WebGL is tested as well, even though it is not supported by every major browser, to see if it is advantageous to use it on browsers that support it. With a scene with only a single object, it is not possible to identify which technique is the best (see fig. 3.7); it depends on the browser used, but the test shows that *Firefox* is the slowest browser at the time of the test[8], yet still has a frame rate reasonable for playing games with every technique. With an increasing scene complexity (100 objects), it is still unclear which technique is the best (see fig. 3.8), but the results are somewhat clearer. A complex scene with 1250 objects shows that the 2D canvas rendering is the fastest technique for complex scenes and still has an appropriate frame-rate for gaming in every browser (see fig. 3.9). Of particular note is the really good performance in *Internet Explorer* which was unexpected due to its bad reputation among web developers. Using WebGL is not beneficial in any browser, so using the 2D context in every browser is the best solution for this kind of games; implementing a second technique is not required. There are some possibilities to improve the performance, for instance by using HTML5 web workers that enable parallelism (more information can be found in [5–7]) and the `requestAnimationFrame` method that prevents rendering of dropped frames, which is described in section 5.2.1. These and other optimizations should be performed by an appropriate rendering engine.

[8] Firefox has improved its performance since the time of the test and now is one of the fastest browsers at rendering 2D graphics, for more information, see sections 6.1 and 7.2.

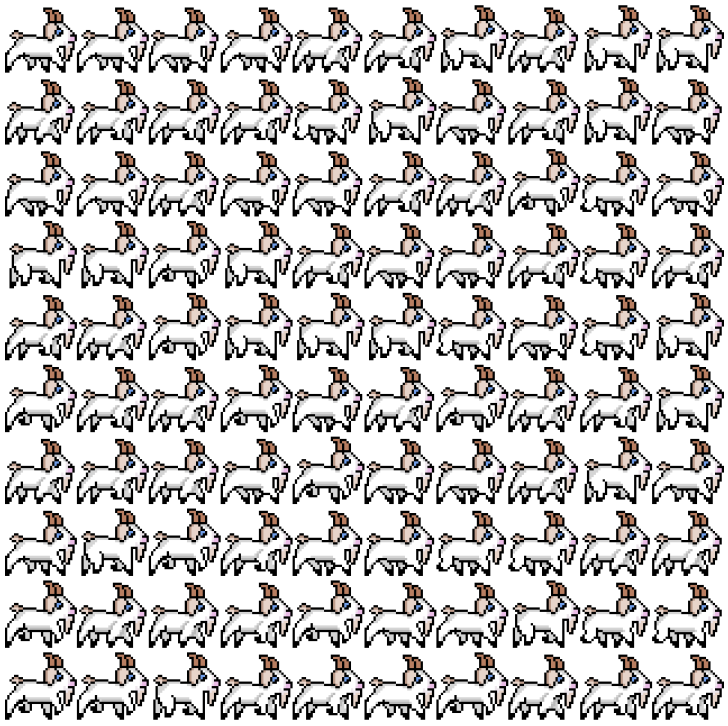

Figure 3.6: Testing the performance of 2D canvas rendering with 100 entities.

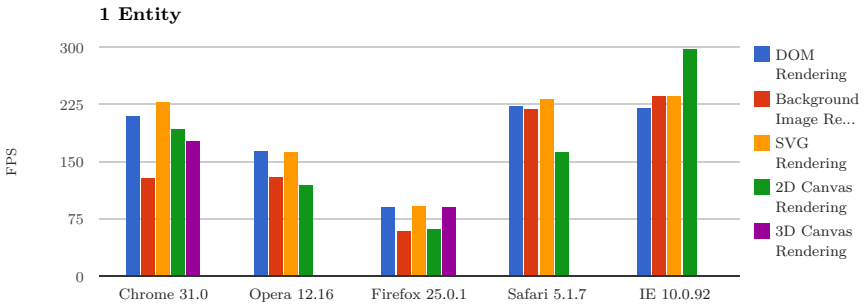

Figure 3.7: Rendering a scene with one entity.

3.7 Summary

As discussed, those rendering techniques offer some advantages and disadvantages. Most interesting is the canvas rendering technique with 2D context as it produces the best results for this use case. Other use cases such as drawing effects or rendering skeletal animations (see fig. 3.10), scenes that require

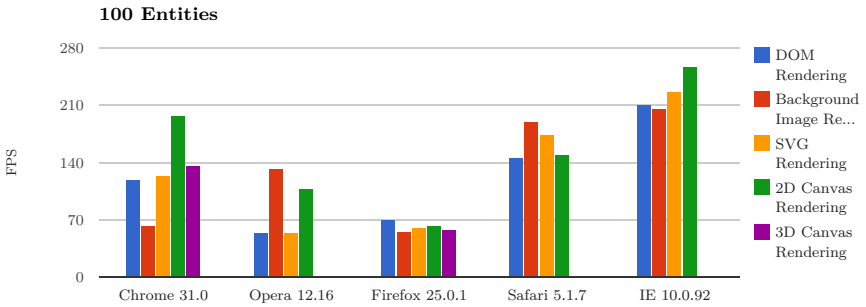

Figure 3.8: Scene with 100 entities.

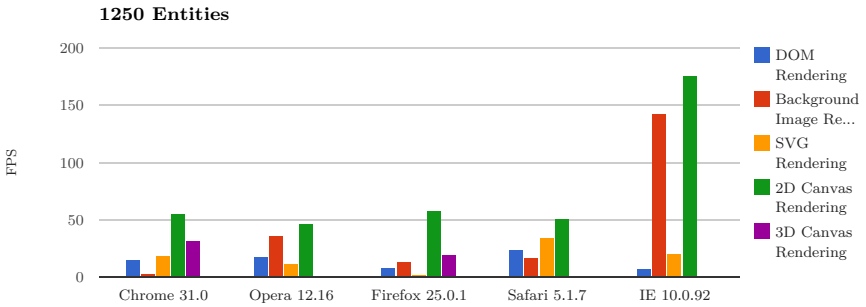

Figure 3.9: Scene with 1250 entities.

floating point positions, rotations and scaling would give different results, and another technique might be more appropriate. The next chapter will introduce a number of frameworks that use these techniques.

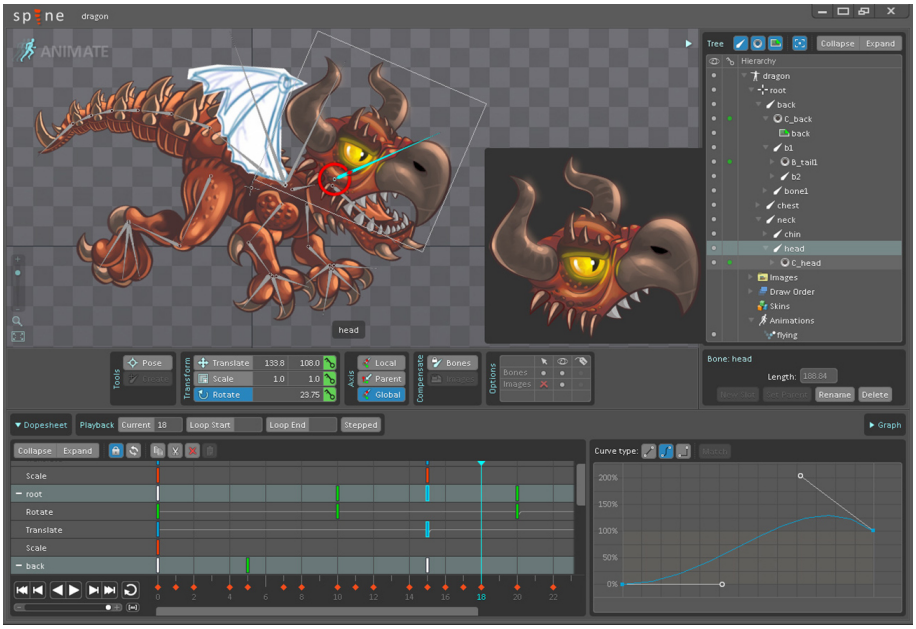

Figure 3.10: A skeletal animation example. The animated object is built from several images that are attached to and rotated against each other. The *pixi.js* framework (see section 4.4) can display such animations (http://www.goodboydigital.com/pixijs/examples/12-2/). Image source [17].

Chapter 4

Rendering Frameworks

For most applications there are rendering frameworks that can help to improve and simplify development. These frameworks have different advantages and weaknesses depending on their major purposes. This chapter introduces a number of frameworks suitable for games. Frameworks for 3D games such as *Three.js*[1], *PlayCanvas*[2] and *Turbulenz*[3] as well as frameworks that are useful for applications other than games, such as *MeteorCharts*[4] for rendering chart graphics, are not covered here. The frameworks introduced here were chosen based on their popularity according to [20] or their unique feature sets. When possible, the source code for a scene with a moving animated sprite image is shown so the frameworks can be compared. The main focus of the comparison is on animated sprites, performance and browser compatibility.

4.1 KineticJS

Most of the frameworks mentioned here focus on games; they come with physics and collision detections, but *KineticJS*[5] does not. The strengths of *KineticJS* are its mouse collision detection on transformed (translated, rotated and scaled) objects with complex boundaries; for example, the alpha channel of an image can be used to define the boundaries of an object. *KineticJS* also supports drag and drop, tweening, sprite animations, layers and grouping. Those features make it a great framework for board and card games, where there are complex clickable regions, smooth animations and layered objects. But for arcade games such as platformers, it cannot be used, as the frame-rate drops below 15 frames per second with an increasing

[1] http://threejs.org/
[2] https://playcanvas.com/
[3] https://github.com/turbulenz/turbulenz_engine
[4] http://meteorcharts.com/
[5] http://kineticjs.com/

number of objects. Due to the possibility of scaling and rotating images, sub-pixel accuracy is required, which influences the rendering speed heavily using the 2D context on the canvas. Another mentionable feature are the image filters that *KineticJS* offers. For example, they can be used to convert the image to grey-scale, invert the colours, or blur the image. The following source code shows an animated sprite image moving from left to right:

```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
    <TITLE>KineticJS Demo</TITLE>
    <SCRIPT type="text/javascript" src="kinetic-v5.1.0.min.js"></SCRIPT>
  </HEAD>
  <BODY>
    <DIV id="stage"></DIV>
    <SCRIPT type="text/javascript">
      var stage = new Kinetic.Stage({
        container: 'stage',
        width: 100,
        height: 100
      });
      var layer = new Kinetic.Layer();

      var actor;
      var startTime;

      var imageObj = new Image();
      imageObj.onload = function() {
        actor = new Kinetic.Sprite({
          x: 0,
          y: 0,
          image: imageObj,
          animation: 'walk',
          animations: {
            walk: [
              // x, y, width, height (9 frames)
              0,0,32,32,
              32,0,32,32,
              64,0,32,32,
              96,0,32,32,
              128,0,32,32,
              160,0,32,32,
              192,0,32,32,
              224,0,32,32,
              256,0,32,32
            ],
          },
          frameRate: 8,
          frameIndex: 0
        });

        layer.add(actor);
        stage.add(layer);
```

```
47
48          actor.start();
49
50          startTime = new Date().getTime();
51          requestAnimationFrame(animate);
52
53        };
54        imageObj.src = 'walk.png';
55
56        var animate = function(){
57          var timePassed = new Date().getTime() - startTime;
58
59          actor.x(timePassed/1000*10);
60
61          requestAnimationFrame(animate);
62        };
63      </SCRIPT>
64    </BODY>
65 </HTML>
```

The frame definition for sprite images is an array of numbers containing the position, width and height. This array becomes long and hard to read when there are a lot of frames in an image. The advantage of this approach is that there are no limitations on how the frames in the sprite image have to be organized. However, it would be easy to create this array automatically for most sprite images. *KineticJS* does not offer a timer or accessible game loop—although there has to be some kind of game loop in *KineticJS* to play the animations—so a custom implementation is required.

4.2 Construct 2

Construct 2 is more than just a rendering framework. It is a development platform to create games for the web (using HTML5) and other platforms with a visual editor. *Construct 2* uses its own rendering and game engine. An analysis of the script *Construct 2* used in an exported game showed that the game is stored in a very large array containing all different data types and other arrays. Thus it is not human readable as there are no labels or variable names; the code is generated by the *Construct 2* editor, which makes it impossible to change the game contents without the use of the editor or even create an entire game without the editor, just using the HTML5 rendering and game engine. Taking a closer look at the rendering and game engine—which has about 20000 lines of code—showed that they re-implemented a lot of methods already available in JavaScript, as for example a method for rounding numbers. Those methods are most likely re-implemented for performance reasons. Some sources state that some mathematics functions can be implemented in a more efficient way than the `Math`-class does. For more information on those performance optimizations see section 5.2.3. A

code example cannot be shown as the code generated by *Construct 2* is not human readable.

4.3 EaselJS

The JavaScript library *EaselJS*[6] is part of *CreateJS*[7]. It focuses on games and on the website it is stated that "[it] provides an API that is familiar to Flash developers" [25]. In addition to drawing sprite animations, shapes, and curves, it offers features such as click handling, masking and image filters. According to statistics, it is one of the most popular JavaScript game engines (see [20]). The following code example shows the animated and moved sprite with *EaselJS*:

```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
    <TITLE>EaselJS Demo</TITLE>
    <SCRIPT type="text/javascript" src="easeljs-0.7.1.min.js"></SCRIPT>
  </HEAD>
  <BODY>
    <canvas id="stage" width="100" height="100"></canvas>
    <SCRIPT type="text/javascript">
      var stage = new createjs.Stage("stage");

      var spriteSheet = new createjs.SpriteSheet({
        "framerate":8,
        "images":["walk.png"],
        "frames": {width:32, height:32},
        "animations":{
          "walk":[0, 8],
        }
      });

      var actor = new createjs.Sprite(spriteSheet, "walk");

      stage.addChild(actor);

      createjs.Ticker.setFPS(60);
      createjs.Ticker.addEventListener("tick", stage);
      createjs.Ticker.addEventListener("tick", function(event){
        actor.x = event.runTime/1000*10;
      });
    </SCRIPT>
  </BODY>
</HTML>
```

The sprite sheet definition can become complex and hard to read, but offers a lot of features. There are no limitations on how the frames in a sprite image

[6] http://www.createjs.com/#!/EaselJS
[7] http://www.createjs.com/

have to be organized. An image file could contain multiple animations, and a single animation could be built from multiple images. The `Ticker` class provides timer and game loop functionalities.

4.4 pixi.js

pixi.js[8] tries to render the contents with the WebGL context when available and offers a fallback to the 2D context. However it focuses on 2D rendering and uses WebGL for better performance, especially when using filters. The *pixi.js* library does not offer all features required for games, such as game physics or defining multiple animations for an object. Just like *EaselJS*, *pixi.js* is one of the most popular HTML5 rendering frameworks according to statistics (see [20]). The following code shows the animated and moved sprite with *pixi.js*:

```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
    <TITLE>pixi.js Demo</TITLE>
    <SCRIPT type="text/javascript" src="pixi.js"></SCRIPT>
  </HEAD>
  <BODY>
    <SCRIPT type="text/javascript">
      var stage = new PIXI.Stage(0xFFFFFF);

      var renderer = PIXI.autoDetectRenderer(100, 100);
      document.body.appendChild(renderer.view);

      var animation = [];
      for(var i = 0; i < 9; i++){
        var rect = new PIXI.Rectangle(i*32, 0, 32, 32);
        var texture = new PIXI.Texture(PIXI.BaseTexture.fromImage("walk.
    png"), rect);
        animation.push(texture);
        console.log(texture);
      }

      var actor = new PIXI.MovieClip(animation);
      actor.animationSpeed = 8 / 60; //running at 60fps, desired animation
    framerate is 8fps
      actor.gotoAndPlay(0);

      stage.addChild(actor);

      var startTime = new Date().getTime();
      requestAnimFrame(animate);

      function animate() {
        requestAnimFrame(animate);
```

[8] http://www.pixijs.com/

```
33
34          var timePassed = new Date().getTime() - startTime;
35          actor.position.x = (timePassed/1000*10 +0.5) | 0; //rounded
        position to prevent animation artefacts
36
37          renderer.render(stage);
38        }
39      </SCRIPT>
40    </BODY>
41 </HTML>
```

Creating a sprite animation with *pixi.js* requires more code than the other frameworks. To define several frames in one image, each frame has to be created as a separate `Texture` loaded from the same file with an additional `Rectangle` as parameter to crop the image. These crop regions can also be defined in a JSON file defining different frames from one image; however, the `Texture` objects for each frame have to be created. When moving the `MovieClip` object that is created from one image with cropped regions, some artefacts from the previous and next frame occur in the image due to the calculations when the object is not aligned with the pixel grid. This can be prevented by rounding the position of the image. The animation speed cannot be set in frames per second or milliseconds between frames, but relative to the rendering frames per second of the scene, which is usually 60 frames per second. Just like *KineticJS*, *pixi.js* does not offer a timer or accessible game loop. However it has a game loop to draw animations. In order to use *pixi.js*, it has to be executed on a web server due to modern browsers "Cross-Origin Resource Sharing policy", as *pixi.js* tries to load images with an AJAX request.

4.5 ImpactJS

The *ImpactJS*[9] framework has to be purchased and is not open source. It includes a browser-based level editor for tile-based levels (see fig. 4.1). While the game engine itself runs without a web server, a PHP web server is required in order to use the level editor. Usually, the JavaScript code of an *ImpactJS* project is in a separate file—*ImpactJS* is the only framework mentioned here that has a predefined project and folder structure—but for demonstration, the JavaScript code is inside the HTML file in the following example:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Impact Demo</title>
5
6   <script type="text/javascript" src="lib/impact/impact.js"></script>
```

[9] http://impactjs.com/

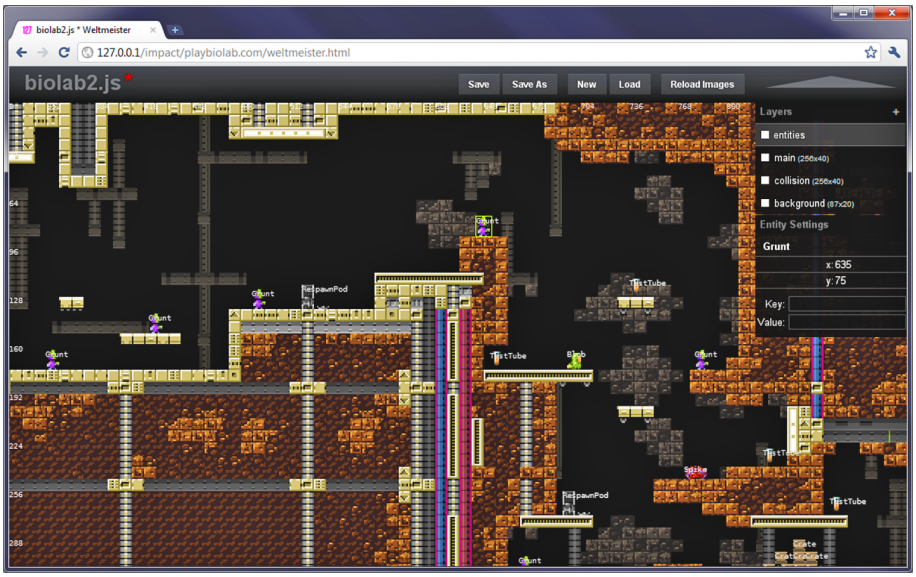

Figure 4.1: The web based level editor of *ImpactJS*: *Weltmeister*. Image source [18].

```
<script type="text/javascript">
  ig.module(
    'game.main'
  )
  .requires(
    'impact.game',
    'impact.font'
  )
  .defines(function(){

    MyGame = ig.Game.extend({

      spriteSheet: new ig.AnimationSheet( 'media/walk.png', 32, 32 ),
      actor: null,
      timer: null,

      init: function() {
        this.timer = new ig.Timer();
        this.actor = new ig.Animation( this.spriteSheet, 1/8,
    [0,1,2,3,4,5,6,7,8] );
      },

      update: function() {
        this.parent();
        this.actor.update();
      },

      draw: function() {
```

```
34            this.parent();
35            var x = this.timer.delta()*10;
36            this.actor.draw(x, 0);
37          }
38        });
39
40        ig.main( '#stage', MyGame, 60, 100, 100, 1 );
41
42      });
43    </script>
44 </head>
45 <body>
46   <canvas id="stage"></canvas>
47 </body>
48 </html>
```

ImpactJS has a clear structure, it offers an initialization, update and draw function in the `Game` object. Unlike the other frameworks, the sprite images may not contain paddings between the single frames, because only the frame width and height can be specified. It is not possible to create an `Animation` that automatically uses all frames from the `AnimationSheet`, the Frames have to be defined in an array containing the frame indices. These arrays could be generated automatically. The `update` and `draw` functions are part of a built in game loop, the `Timer` class can be used to get time differences. As opposed to the other frameworks and common custom timer implementations, the `Timer` class of *ImpactJS* works with seconds instead of milliseconds.

4.6 Framework Comparison

The frameworks were compared by drawing a scene of 5000 animated sprite images for a certain amount of frames and measuring the time (see fig. 4.2). Because most frameworks limit the frame rate to 60 frames per second, a high number of sprites was used, so the frame rate would drop beyond the limit. Usually a scene does not contain more than about 1000 entities. All frameworks worked in every browser, but the performance differed. *KineticJS* has a poor performance compared to the other frameworks, which underlines that it is not intended to be used for arcade games. *EaselJS* performs better but still drops below 30 frames per second. Yet the performance is admissible as common scenes contain less entities. *ImpactJS* and *pixi.js* perform best in this comparison. Since *ImpactJS* is commercial and not open source, it is not unusual that it is highly optimized, even though it does not use multi threading or WebGL. The great performance of *pixi.js* could be a result of it being a pure rendering framework and not a game engine. It uses WebGL when possible but that does not improve rendering of sprite animations, as shown in section 3.5. For the implementation of a game and especially the game editor *ZIEGE* that lead to this evaluation,

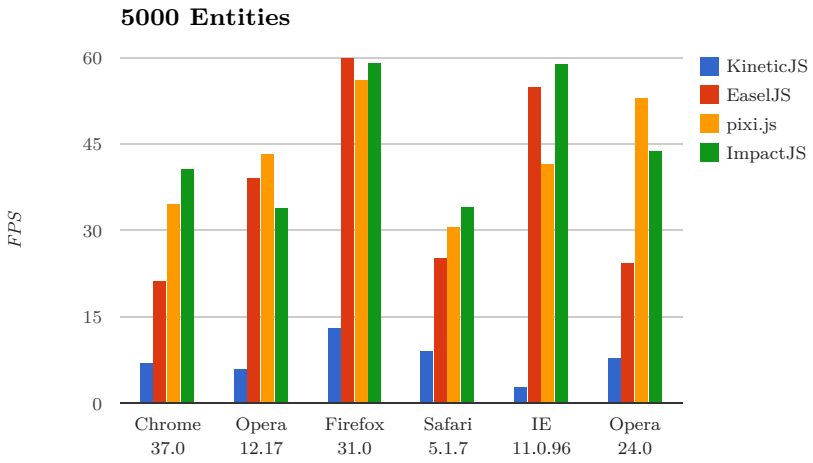

Figure 4.2: Rendering a scene with 5000 entities using different frameworks and browsers.

pixi.js would be hard to use as it does not have a timer or game loop: The animation of each entity would have to be started and stopped separately. Changing the animation of an entity is not simple as well. *ImpactJS* could not be used as it is commercial—the funding of the project was bound to the condition that the project is open source. As the performance of *KineticJS* is too poor and the code of *Construct 2* is not human readable and thus an editor could not be created, only *EaselJS* could be used. The rather complex sprite sheet definitions and the inflexible scenegraph (the child objects of a `Stage` or `Container` object can only be referenced by a consecutive numeric ID, so the scenegraph is made of arrays only) of *EaselJS* makes it hard to create or remove large amounts of entities or to optimize it for displaying the entities aligned in a grid. Thus it was decided to create a custom rendering framework for the *ZIEGE* project.

Chapter 5

H5R Framework Prototype Implementation

The aim of this rendering framework—which will be referred to as the H5R framework—is to be able to create rendered 2D scenes with little effort, to create easily readable code and to be able to save and load scenes without parsing or converting any data by hand while still having a reasonable performance when displaying large amounts of sprite animations. These requirements are defined by the *ZIEGE* project, which allows the user to create 2D games in the web browser. To be able to use the project on any computer without special permissions to install any software, it should run in every major web browser without the installation of any plug-ins. While the ability to create easily readable code cannot be measured as it is highly subjective, the other requirements can. The creation of scenes with little effort can be measured by counting the lines of code required to achieve certain tasks and comparing them to the code required with other frameworks. The ability to save and load scenes without parsing or converting any data by hand is fulfilled if the entire scene is stored in and read from a JSON object. Whether the performance is reasonable when displaying large scenes can be measured by defining a number of elements that have to be drawn at the same time and measuring the frame-rate—which has to be higher than 15 frames per second, but ideally even higher than 30 frames per second. In [4], Mark Claypool draws the conclusion:

> In general, user performance shows a marked drop in performance below 15 frames per second with a modest increase from 15 to 30 frames per second.

This chapter shows the implementation of the framework and describes how these requirements are handled. The implementation is a prototype, thus it is not complete but shows the approach and it can be used in certain projects. Some features are not implemented on purpose (such as entity rotation or scaling) as those features heavily affect performance.

5.1 Structure and Usage

The framework works with two classes: The class `H5R` (see section 5.1.4) controls the game loop, observes the document object model for changes, manages stages, pre-loads the tileset images and updates the scenegraph. The class `H5R.Stage` (see section 5.1.5) renders the contents on a canvas; each `H5R.Stage` has its own canvas to draw on and can render the entire scenegraph or a part of it. A camera can be defined to draw the scene from another point of view. The tileset, scenegraph and camera are handled as JSON Objects (see fig. 5.1), as it is human readable, has smaller amounts of data than XML, and can be handled well with JavaScript and server technologies like *nodejs*[1] and *mongoDB*[2]. The framework is designed to draw sprite animations and images, but custom graphics such as shapes or text can be drawn as well, as the scenegraph offers the possibility to access the rendering context at any position. To add game logic or physics, an update function can also be added at any position in the scenegraph.

5.1.1 Defining the Tileset

The format of the tileset object is JSON, thus the structure can be defined entirely by the user; it could be a deep object structure, a flat array structure, or a combination of both, but it may not contain circular references. There are two object name keywords that can be used in the tileset object to define an image resource. The keyword `src` can be used to define an image URL, but it has to be of type `string` in order to be read as image URL by the parser. Using this keyword, only a single image can be defined and it cannot be a sprite animation. Using the keyword `img`, either an image URL, an object or an array of objects and strings can be defined. Defining an image URL with type of `string` equals the functionality of the `src` keyword. When the keyword `img` references an object, the object has to contain a `src` keyword defining an image url with the type of `string`, and the object may contain the keywords `sprites` defining the number of sprites in the image with the type of `number` and `fps` with the type of `number` defining the number of frames per second. When the keyword `img` references an array, the array has to contain either elements with type of `string` containing an image URL or objects as described before, or a combination of strings and objects. Each image parsed from the tileset object is given a human readable ID defined by its position in the JSON object. When parsing the tileset object, the object keys and array indexes are concatenated separated by dots (excluding the `img` and `src` keys). These IDs have to be used with the `_img` keyword in the scenegraph to reference a certain image from the tileset. If the `img` keyword references an array, the image to be drawn can be

[1] http://nodejs.org/
[2] http://www.mongodb.org/

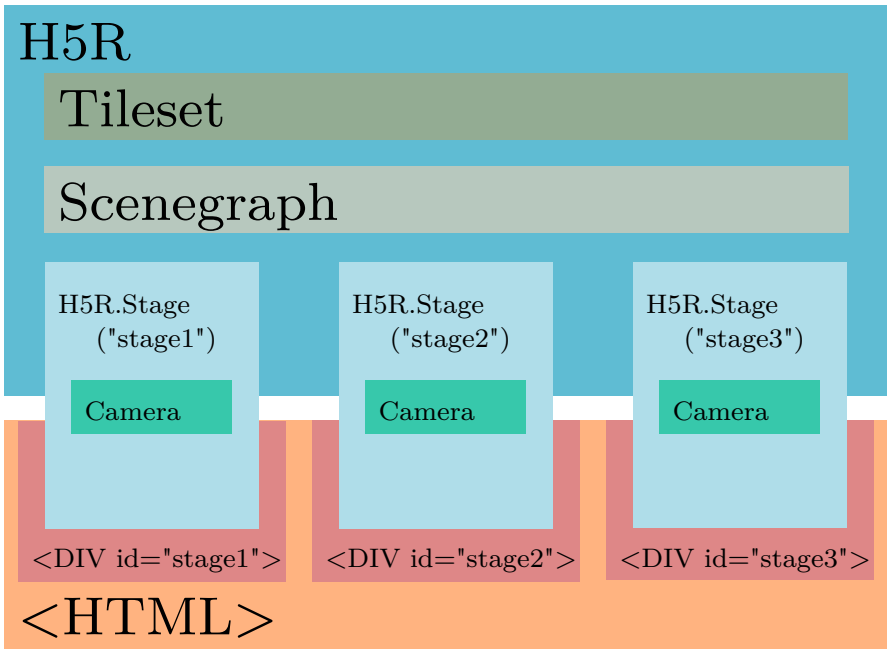

Figure 5.1: The structure of the H5R framework when using three different canvases. The multiple canvases share one game loop, tileset and scenegraph, but can display different positions set by the cameras and render different parts of the scenegraph, provided that scenegraph parts are specified for the H5R.Stage objects. This makes it possible to draw different layers to different canvases that are stacked on top of each other, or to display different positions of the scene, which could be useful in a multi-player game that runs on a single device (split-screen).

selected with the `_animation` keyword in the scenegraph (see section 5.1.2). The code below shows an example tileset object:

```
1  var tileset = {
2    "wall": {
3      "brick": {
4        img: "brick.png", // _img: "wall.brick"
5      },
6      "wood": {
7        src: "wood.png", // _img: "wall.wood"
8      },
9    },
10   "decoration": {
11     "torch": {
12       img: {
13         src: "torch.png", // _img: "decoration.torch"
14         sprites: 8,
15         fps: 16,
16       },
```

```
17      },
18    },
19    "enemies": [
20      {
21        "spike": {
22          img: "spike.png", // _img: "enemies.0.spike"
23        },
24        "lava": {
25          img: "lava.png", // _img: "enemies.0.lava"
26        },
27      },
28      {
29        img: [
30          "idle.png", // _img: "enemies.1", _animation: 0
31          {
32            src: "walk.jpg", // _img: "enemies.1", _animation: 1
33            sprites: 8,
34            fps: 12,
35          },
36        ],
37      },
38    ],
39  };
```

5.1.2 Defining the Scenegraph

The scenegraph itself is also a JSON object and its structure can be defined by the user, and as in the case for tilesets, it may not contain circular references. There are some keywords that control how the scenegraph is drawn to the canvas. When the key of an object starts with an underscore, it will be ignored by the parser unless it is a keyword. Some of the keywords can be defined by the user, while others are set by the rendering framework. The keywords that can be defined by the user are as follows:

`_x` ⟨**number**⟩
: is used to define the position in x direction. The point of origin is the upper left corner, the x-axis points right. All child nodes of the scenegraph part will have their origin translated by this value.

`_y` ⟨**number**⟩
: is used to define the position in y direction. The point of origin is the upper left corner, the y-axis points down. All child nodes of the scenegraph part will have their origin translated by this value.

`_img` ⟨**string**⟩
: is used to reference an image ID generated from the tileset. The referenced image will be drawn at the defined position. If the image ID is not found, no image will be drawn.

`_animation` ⟨**number**⟩
: is used to define the animation to be drawn if the image id references

an array of images. The default value is 0. If the value is invalid (less than zero or greater or equal than the array length), no image will be drawn.

`_freeze` ⟨**boolean**⟩
: if set to true, the sprite animation will be stopped.

`_startFrme` ⟨**number**⟩
: if set, the animation will start at the specified frame.

`_flipX` ⟨**boolean**⟩
: if set to true, the image is mirrored in x-direction.

`_flipY` ⟨**boolean**⟩
: if set to true, the image is mirrored in y-direction.

`_repeatX` ⟨**number**⟩
: is used to define how often the image should be repeated in x-direction.

`_repeatY` ⟨**number**⟩
: is used to define how often the image should be repeated in y-direction.

`_hide` ⟨**boolean**⟩
: if set to true, the scenegraph part and its children will not be drawn. However they will be parsed and updated.

`_hideChildren` ⟨**boolean**⟩
: if set to true, the child elements of the scenegraph part will not be drawn. However they will be parsed and updated.

`_update` ⟨**function**⟩
: is used to define a custom update function which is called from the scenegraph parser. It can be used for game logic and physics.

`_render` ⟨**function**⟩
: is used to define a custom render function which is called from the renderer. It can be used to draw custom shapes or text.

There are some keywords that are set by the rendering framework. The user can access them from the custom update and render function, but they might not be set (it takes up to two rendering cycles to calculate them):

`_frame` ⟨**number**⟩
: is set to the currently displayed frame.

`_previousFrame` ⟨**number**⟩
: is set to the previously displayed frame.

`_imgWidth` ⟨**number**⟩
: is set to the width of the currently displayed image.

`_imgHeight` ⟨**number**⟩
: is set to the height of the currently displayed image.

`_width` ⟨**number**⟩
: is set to the width of the currently displayed image multiplied by the `_repeatX` parameter or 1 if it is not set.

`_height` ⟨**number**⟩
: is set to the height of the currently displayed image multiplied by the `_repeatY` parameter or 1 if it is not set.

`_fps` ⟨**number**⟩
: is set to the speed in frames per second, the referenced image is drawn at.

The Custom Update Function

As mentioned before, a custom update function can be defined at any position in the scenegraph using the `_update` keyword. It will be called from the scenegraph parser once each update cycle. The function takes four parameters: The first parameter is the part of the scenegraph containing the `_update` keyword, so other keywords defined on the same object in the scenegraph, as well as child objects, can be accessed easily. The second parameter is the time passed since the previous update cycle in milliseconds. The third parameter is the total time passed since the first update cycle in milliseconds. The fourth parameter is an object containing the `getImage` function, which can be used to access objects from the scenegraph. It is the same function that is used by the renderer to get the images (see section 5.1.4). The update function can be used to change the position and appearance of an object, game logic and physics can be defined here. The code below shows an example scenegraph with a custom update function that moves an object to the right by ten pixels each second:

```
1 var scenegraph = {
2   _x: 0,
3   _y: 0,
4   _img: "enemies.1",
5   _animation: 1,
6   _update: function(self, timeDiff, timePassed, renderer){
7     self._x += timeDiff/1000*10;
8   },
9 };
```

The Custom Render Function

The custom render function can be defined with the `_render` keyword. It will be called from the renderer once each render cycle from each `H5R.Stage` that renders this part of the scenegraph. The function takes three parameters: The first parameter is the same as in the custom update function; it is the part of the scenegraph containing the `_render` keyword. The second parameter is the 2D rendering context of the canvas. The third parameter is the same as the fourth parameter of the custom update function, an object containing the `getImage` function. The render function can be used to draw

custom shapes or text to the canvas. The code below shows an example scenegraph with a custom render function that draws a bordered rectangle:

```
1 var scenegraph = {
2   _render: function(self, ctx, renderer){
3     ctx.beginPath();
4     ctx.rect(0, 0, 20, 20);
5     ctx.fillStyle = "rgba(255,255,255,0.5)";
6     ctx.fill();
7     ctx.lineWidth = 2;
8     ctx.strokeStyle = "rgba(0,0,255,0.5)";
9     ctx.stroke();
10  },
11 };
```

5.1.3 Defining the Camera

The camera can be defined using a JSON object containing an `x` and `y` value, both of type `number`. If no camera is set for a `H5R.Stage`, the camera is at position (0,0). The width and height of the area drawn is defined by the width and height of the canvas. The code below shows an example camera object:

```
1 var camera = {
2   x: 200,
3   y: 450,
4 };
```

5.1.4 The H5R Class

The `H5R` class is the core of the framework. It contains the game loop, manages the rendering stages, observes the document object model for changes, pre-loads images and updates the scenegraph. The constructor of the class does not take any parameters. The `H5R` class has the following public functions:

`H5R.Stage addStage(H5R.Stage stage)`
: adds the specified `H5R.Stage` to the renderer and returns the added `H5R.Stage` object.

`H5R.Stage getStage(string ID)`
: returns the `H5R.Stage` with the specified `ID` or `null` if the stage was not added to the renderer before.

`void removeStage(string ID)`
: removes the `H5R.Stage` with the specified `ID` from the renderer.

`void updateStages()`
: checks for each `H5R.Stage` that was added to the `H5R` class, if the HTML-element with the `ID` of stage the exists. If the element exists and the canvas inside of it is not yet created, it creates it. Usually

the method is called from the document object model observer of the renderer, but the observer does not work in every browser, so it is recommended to call the method when the website is fully loaded and whenever the HTML-element of a stage changes.

`void restart()`
: resets the timer, sets the paused state to `false` and starts the game loop.

`void start()`
: is an alias for `restart()`.

`void stop()`
: stops the game loop, resets the timer, sets the paused state to `false`.

`void setPaused(boolean paused)`
: changes the paused state to the specified value. When paused, the scenegraph is not updated or rendered.

`boolean isPaused()`
: returns the paused state.

`void drawFrame()`
: renders the stages with the time difference to the last frame set to 0 and the current total time, no matter if the `H5R` is stopped or paused. It does not update the scenegraph.

`void setTileset(object tileset, function onLoad)`
: removes all loaded image data, parses the specified tileset object and pre-loads the images defined in it. If specified, the `onLoad` callback function is called as soon as all images are loaded.

`void updateTileset(function onLoad)`
: clears all loaded image data, parses the previously specified tileset object again and pre-loads the images defined in it. If specified, the `onLoad` callback function is called as soon as all images are loaded. When the tileset object changes, this method has to be called.

`void setScenegraph(object scenegraph)`
: sets the scenegraph to the specified object. It is not necessary to call this method again or any update method whenever the scenegraph object changes, as it is parsed each game loop cycle.

`object getImage(string ID, number animation)`
: returns an image descriptor object of the image with the specified `ID` and `animation` (see section 5.1.1). The returned object contains the following keywords:

  `width` ⟨**number**⟩
  : is set to the image width divided by the defined amount of sprites.

  `height` ⟨**number**⟩
  : is set to the image height.

`sprites` ⟨**number**⟩
: is set to the amount of sprites in the image.

`fps` ⟨**number**⟩
: is set to the number of frames per second defined for the image.

`src` ⟨**string**⟩
: is set to source URL of the image.

`img` ⟨**object**⟩
: is set to preloaded image object.

`raw` ⟨**object**⟩
: is set to the part of the tileset object that contained the `src` or `img` keyword.

5.1.5 The H5R.Stage Class

Each `H5R.Stage` class represents a stage and renders the entire scenegraph or a part of it to the canvas. The constructor of the class requires an `ID` of type `string` to be set as parameter; this `ID` defines the HTML-element that should contain the canvas. The `H5R.Stage` has the following public functions:

`H5R.Stage setScenegraphPart(string sgPart)`
: sets the part of the scenegraph to be rendered on this stage; the parameter is a concatenation of the object keys and array indexes separated by dots. If the parameter is an empty string, the entire scenegraph will be rendered. The function returns the `H5R.Stage` object itself.

`H5R.Stage setCamera(object camera)`
: sets the camera object to be used on this stage. The function returns the `H5R.Stage` object itself.

`string getId()`
: returns the `ID` of the stage.

`boolean isValid()`
: returns `true` if an `ID` was set as parameter in the constructor.

`void checkDOMElement()`
: checks if the HTML-element with the `ID` of stage the exists. If the element exists and the canvas inside of it is non-existent, it creates it. This function is called from the `updateStages()` function of the `H5R` class.

`object getCanvas()`
: returns an object representing the canvas element of the stage. It can be used to copy the contents of the canvas, for example for screenshots.

5.1.6 Using the H5R-Framework

Rendering a scene with the H5R-Framework requires a minimum of a tileset object, a scenegraph object, an instance of the `H5R` class, an HTML-element with a specified `ID` and an instance of the `H5R.Stage` class initialized with the `ID` of the HTML-element. The code below shows an example on how to use the H5R-Framework:

```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
    <TITLE>H5R Framework Demo</TITLE>
    <SCRIPT type="text/javascript" src="H5R.js"></SCRIPT>
    <SCRIPT type="text/javascript">
      renderer = new H5R();
      renderer.addStage(new H5R.Stage("stage"));

      var tileset = {
        "walk": {
          img: {
            src: "walk.png",
            sprites: 9,
            fps: 8,
          }
        }
      };

      var scenegraph = {
        actor: {
          _x: 0,
          _img: "walk",
          _update: function(self, timediff, timepassed){
            self._x = timepassed/1000*10;
          },
        }
      };

      renderer.setScenegraph(scenegraph);
      renderer.setTileset(tileset);
    </SCRIPT>
  </HEAD>
  <BODY>
    <DIV id="stage" style="width: 100px; height: 100px;"></DIV>
  </BODY>
</HTML>
```

In appendix A, the code for a simple *Flappy Bird* clone implemented with the H5R framework can be found. This should illustrate the readability of the code as well as the required amount of code when using the H5R framework. It includes features like saving and restoring the scene (when the player dies), image pre-loading, animated sprites, custom update and render functions and custom values in the scenegraph.

5.2 Optimizations

Optimizations in JavaScript are only possible to a certain extent, as the low-level code cannot be changed; only high-level optimizations are possible. This section introduces some possible optimizations and evaluates most of them by directly comparing the results with and without the optimization.

5.2.1 Synchronizing the Framework with the Browser

The game loop could be executed consecutively using the `setTimeout` or `setInterval` functions of JavaScript, which offer the possibility to set a delay in milliseconds so the frame rate could be set. The `setTimeout` function executes the callback function only once while the `setInterval` function executes the callback function consecutively until it gets stopped by the `clearInterval` function. The following example shows how a game loop could be implemented using either the `setTimeout` or the `setInterval` function:

```
1 var animateWithTimeOut = function(){
2   window.setTimeout(animateWithTimeOut, 1000/60);
3   console.log("tick with timeout");
4 };
5 window.setTimeout(animateWithTimeOut, 1000/60);
6
7 var animateWithInterval = function(){
8   console.log("tick with interval");
9 };
10 window.setInterval(animateWithInterval, 1000/60);
```

This implementation would work; however, it has several disadvantages. First of all "[...] the millisecond delay is not an indication of when the code will be executed, only an indication of when the job will be queued" [37], thus the real delay between the function executions might be different. The range of this variance differs among the browsers. *Interent Explorer 8* and earlier have a resolution of about 16 milliseconds which equals the difference between 30 and 60 frames per second; *Internet Explorer 9* and above, as well as *Chrome* have a resolution of 4 milliseconds and *Firefox* and *Safari* have a resolution of about 10 milliseconds [30, 37]. Another problem with this approach is, that the game loop is not paused when the page is hidden (when the browser is minimized or the tab is hidden), which affects performance even though the animation is not visible. "Chrome does throttle setInterval and setTimeout to 1fps in hidden tabs, but this [is not] to be relied upon for all browsers" [30]. Another drawback is that the `setTimeout` and `setInterval` functions usually require the screen to redraw. The browser usually updates the screen itself at a fixed frame rate. If the frame rates are not synchronized, either the screen is redrawn in between the regular redraws, or the frames are dropped (not drawn and thus calculated unnec-

essarily); either way it takes more processing power and drains the batteries of mobile devices [30, 37].

These problems are solved using the `requestAnimationFrame` function that was proposed by Mozilla and "adopted and improved by the WebKit team" [30] which synchronizes the frame rate of the rendering cycle to the frame rate of the browser:

```
1 var animate = function(){
2   window.requestAnimationFrame(animate);
3   console.log("tick with requestAnimationFrame");
4 };
5 window.requestAnimationFrame(animate);
```

Usually the browser executes this function at a frame rate of 60 frames per second which equals a typical display that refreshes at 60Hz. The browsers are able to optimize performance based on page visibility and battery status by throttling or stopping the execution. Another advantage of this function is that the screen is only redrawn once each frame even though there are multiple rendering loops [30, 37].

To ensure that the function works with every browser (some browsers have a different name for the function and older browsers do not have the function at all), the following solution with fall-back functions is used (taken from [27]):

```
1 window.requestAnimFrame = (function(){
2   return  window.requestAnimationFrame       ||
3           window.webkitRequestAnimationFrame ||
4           window.mozRequestAnimationFrame    ||
5           function( callback ){
6             window.setTimeout(callback, 1000 / 60);
7           };
8 })();
9
10 var anim = function(){
11   window.requestAnimFrame(anim);
12   console.log("tick with requestAnimFrame");
13 };
14 window.requestAnimFrame(anim);
```

5.2.2 Sub-Pixel Rendering

Evaluations show that some browsers are slower when rendering images that are not aligned to the pixel grid (e.g. having floating point positions), scaled or rotated, as the pixel values have to be interpolated instead of copied from the image (see fig. 5.4). Some sources including [28] and [33] confirm this observation. Hence the H5R Framework does not support rotation and scaling, and rounds the objects positions before drawing them.

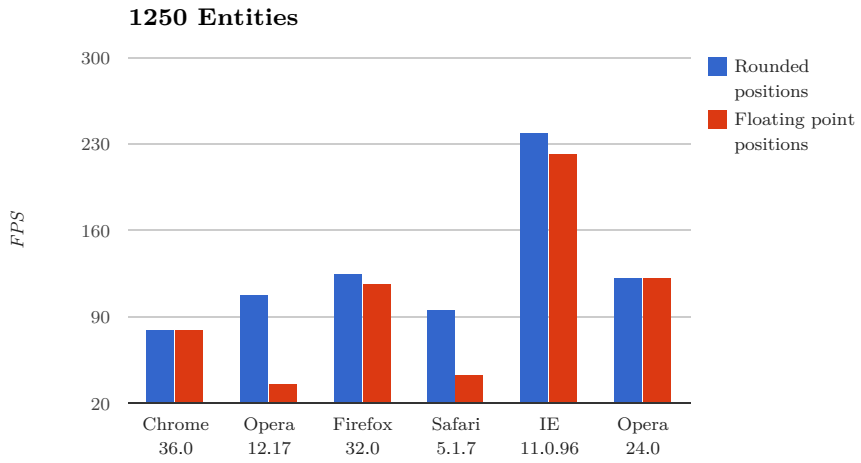

Figure 5.2: The frame rates when drawing 1250 entities with rounded or floating point positions.

5.2.3 Math-Functions

According to a number of sources including [28], [29] and [32], some `Math` functions of JavaScript can be improved with custom implementations, especially the functions for rounding numbers. There are different approaches on reimplementing those methods, the following example shows possible implementations for rounding numbers down:

Math class implementation:

The standard implementation of the `Math` class in JavaScript:

```
1  var floor = Math.floor(number);
```

Bit shift by 0:

When shifting the bits by zero, the decimal places are cut off. Negative numbers have to be treated differently:

```
1  var floor = (number < 0 && number % 1 != 0)? (number << 0) -1 :
     number << 0;
```

Double bitwise NOT operation:

A bitwise NOT operation executed twice returns the same number without the decimal places:

```
1  var floor = (number < 0 && number % 1 != 0)? (~~number) -1 : ~~
     number;
```

Bitwise OR operation with 0:

A bitwise OR operation with zero also returns the same number without the decimal places:

```
var floor = (number < 0 && number % 1 != 0)? (number | 0) -1 :
  number | 0;
```

This implementation can also be found in *Construct 2*:

```
cr.floor = function (x)
{
  if (x >= 0)
    return x | 0;
  else
    return (x | 0) - 1;
};
```

The implementation of *Construct 2* is similar to the one shown above, but it does not check if a negative number has any decimal places; in this case Construct 2 would still subtract one, and thus return a wrong value.

Comparing those implementations (see fig. 5.3) shows that the custom implementations are at most half a second faster at 100 million calculations. Using Google Chrome, the standard implementation is one second faster than the custom implementations. The differences in speed between the implementations are insignificant. Usually there are not more than several thousand rounding calculations in each game loop cycle. With 100000 calculations the difference would be, at most, one millisecond, so this optimization approach can be neglected. More significant are the differences between the browsers.

5.2.4 Off-Screen Canvas Rendering

Another optimization technique proposed by [33] is rendering the scene to a canvas that is not added to the scenegraph (and thus not visible) and copying the contents to the visible canvas. This approach should prevent the browser from rendering in-between states of the scene. The scene is only rendered to the display when completely drawn. The evaluation shows that this optimization only works well in *Google Chrome* (and other Browsers using *Chromium*); in all other browsers it is slightly slower (see fig. 5.4).

5.2.5 Other Optimizations

Rendering different parts of the scene to different layers can be done by rendering them to different canvases (different `H5R.Stage`-Objects with the H5R framework) that are stacked on top of each other (which can be done with absolute positioning in CSS). Using those layers will only update and render layers where contents have changed, thus the performance is improved. Another possibility for performance optimizations would be to render only parts of the scenes where content has changed, but to do this the engine

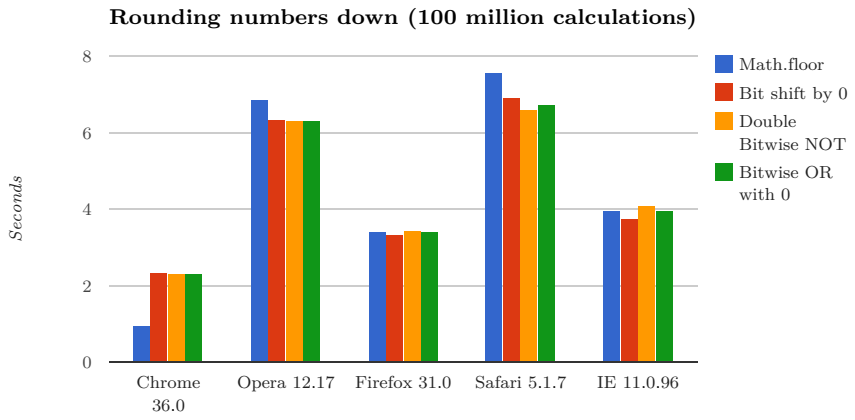

Figure 5.3: The time required for rounding down 100 million numbers with different implementations.

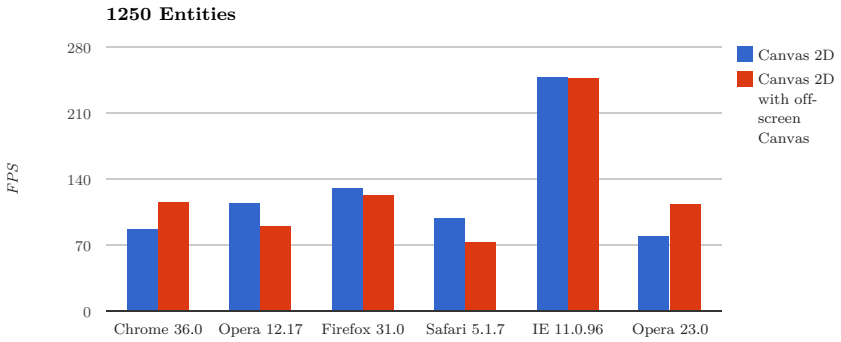

Figure 5.4: Rendering a scene with 1250 entities with and without an off-screen canvas.

would have to know what parts have changed—this is not implemented in the H5R framework. Not rendering elements that are outside the scenegraph could improve performance as well, but a relatively fast function to check if an element is inside or not would be required.

5.3 Summary

The H5R framework tries to combine simplicity and performance by providing a manageable API and performing some optimizations. However, some

functions such as rotating images or sub-pixel rendering are not supported in favour of the performance. On the other hand it provides features for adding several stages using the same scenegraph and renderer, for saving and restoring scenes easily and for making large changes to the scenegraph in real time, which is required for the `ZIEGE` project. Being a prototype, some features are missing such as sound and collision detection and handling, which would be required for a full-featured game engine.

Chapter 6

Evaluation of the H5R Framework

The use of the H5R framework in the *ZIEGE* project shows that it meets its requirements; the frame rates are acceptable, even in complex scenes. However, the H5R framework has been compared to the other frameworks introduced earlier in this thesis. This chapter shows how the remaining research questions from the introduction are addressed. It is difficult to measure how good a framework is; the simplicity and usability is highly subjective. Less code required for the same scene does not mean it is better or easier. Indeed it could even be more difficult. The number of methods a framework or API offers could be measured as well, but a high number does not mean it is more complex or poorly structured; it could just have more features. What can be measured is the frame rate of the same scene built with different frameworks. However, a framework could be fast at rendering one scene, but slow at rendering another, depending on the contents of the scene. Thus a use case for the framework has to be defined; in this case it is the rendering of a scene with a large number of sprite animations. Most scenes can be rendered at 60 frames per second with most of the frameworks; the frame rate is not higher as it is limited to 60 frames per second with most frameworks. Thus the performance cannot be compared in a common game scene. Therefore a stress test is used in which a much larger number of sprite animations is drawn so that the frame rate would drop below the 60 frames per second. This makes it possible to compare the frame rates but it is not representative for a common game scene. The *ZIEGE* project has several layers which are rendered to a number of canvases. Rendering to multiple canvases can improve performance when the content of one or more canvases stays the same from frame to frame, while others are moving, so they do not have to be repainted. If the content on all canvases change, the performance stays the same. So this test ignores that one layer might not be repainted, as it renders only one layer. This means that the performance in the *ZIEGE* project

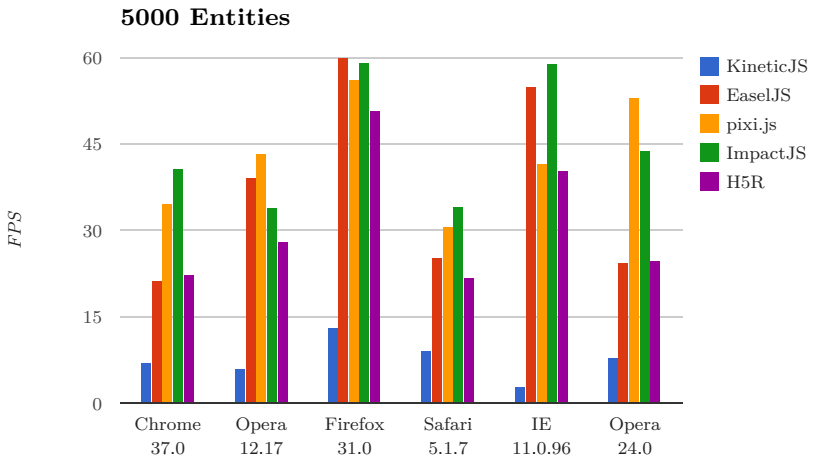

Figure 6.1: Rendering a scene with 5000 entities using the H5R framework, the other frameworks and different browsers.

might be better, but not worse than in the test with the same amount of entities. Drawing to different canvases would be more work using one of the other frameworks.

6.1 Rendering Performance Evaluation

The display rate in frames per second is measured in order to find out if the H5R framework meets the requirements of a game rendering framework. According to [1] and [10], it is "the common metric to measure the responsiveness of interactive applications including computer games". The rendering speed of the H5R framework was measured the same way as it was measured for the other frameworks, so it could be compared directly (see fig. 6.1). The performance of the H5R framework is about the same as the performance of *EaselJS* on most browsers, but still not as good as the other frameworks. However, the performance is good enough, as the 60 frames per second are easily reached when rendering a scene with 1250 entities, which is still more entities than most scenes contain. Even with lesser hardware, a frame rate between 50 and 60 frames per second is reached; it can draw about 2800 entities at a frame rate of 30 frames per second, which is still appropriate for gaming.

6.2 Changing, Saving and Restoring the Scenegraph

Adding or removing 5000 entities to or from the scenegraph takes about one to five milliseconds; it would only marginally affect the frame rate, so it can be considered real-time. The *ZIEGE* project has a playing field of 40 by 30 grid units, which can be extended in any direction, but only 40 by 30 is displayed at any time. Even though there are three layers for level elements, it is not possible to draw on more than one at once. Thus the *ZIEGE* project allows 1200 (40×30) elements at most to be added or removed at once, which would take not more than two milliseconds.

The scenegraph or parts of it can be saved by using the `JSON.stringify` function and storing the text object locally or on a server. The only precondition is that there are no circular references in it. However there are some issues about what can be stored: If there are several fields referencing the same object, they will reference different objects with the same initial values when restoring the JSON object from the text using `JSON.parse`. If a field is referencing a function, it will not be saved using the `JSON.stringify` function. On the one hand, it is a drawback, as the `_update` and `_render` functions cannot be saved this way, but on the other hand it would be an enormous security vulnerability, as program code would be evaluated from text that is probably stored on a server, and could be manipulated by a user. Restoring the saved scenegraph from text can be done using the `JSON.parse` function.

Chapter 7

Conclusion and Outlook

7.1 H5R Framework and the ZIEGE Project

Even though the H5R framework works well in the *ZIEGE* project, it is still a prototype, so some features are missing and some optimizations could be done. In future, sound support will be added as well as some game physics. An alternative to the `JSON.stringify` and `JSON.parse` functions will be added, so references to the same object as well as circular references and functions could be preserved and stored. The *ZIEGE* project will also be continued and extended. Some features such as a behaviour editor for the entities is already planned and only the UI implementation is missing (see fig. 7.1).

7.2 Web Technologies

Web technologies are getting better and better and the support of HTML5 features among the browsers as well as the performance is increasing steadily. Even during the writing of this thesis, the HTML5 support and performance in some browsers increased substantially. *Firefox* had almost 10 new version releases and the improvements in performance can be seen by comparing the charts from chapters 3 and 4. It shows that *Firefox* was one of the slowest browsers and became one of the fastest again. It is a steady head-to-head race among the browsers. With the increasing performance and improving technologies, the web is transforming from a document-oriented to a application-oriented platform. A good summary can be found in [2]:

> The ability to run non-trivial applications (as opposed to simple scripts) inside the web browser has increased tremendously over the last years. Thanks to high performance JavaScript engines, raw JavaScript execution speed has increased by two orders of magnitude compared to the situation only three years ago. In this area, competition between browser manufacturers

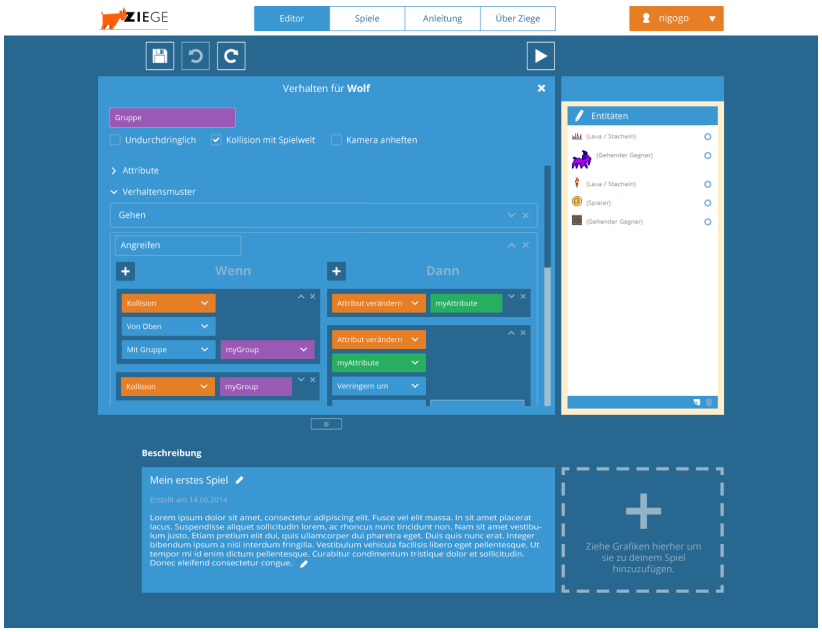

Figure 7.1: The planned UI design of the behaviour editor in the *ZIEGE* project.

> has clearly benefited the customer and led to a situation in which web applications and environments that were barely runnable in 2007 [...] can now run even in mobile devices without major problems. In general, limitations associated with raw JavaScript execution performance have mostly disappeared.

7.3 Games

The number of platforms for gaming is increasing and so are the requirements for games. Consoles, mobile devices, browsers, *Steam OS* are just some examples of gaming platforms. To reach as many people as possible, cross-platform development is becoming more and more important. Game engines like *Unity* support a number of platforms that the games could be released on, but nevertheless some platform-specific code is almost always required. Web browser games that do not use any plug-ins are a great alternative, as most platforms and operating systems can run them without many changes to the code. But especially with mobile devices, the performance is still not adequate for browser games. Yet this will most likely change in the future. In addition to the desktop browsers, also mobile browsers are improving steadily, but it will take some time until most mobile devices have an adequate browser for gaming installed.

Appendix A

Simple Flappy Bird Clone with the H5R Framework

The code below shows a simple *Flappy Bird* clone implemented using the H5R framework (see fig. A.1). There are three graphics used: A background image with a size of 640 by 480 pixels, which equals the scene size. The bird graphic is a sprite animation with three frames with a size of 64 by 48 pixels each. The obstacles are displayed using a brick graphic with a size of 16 by 16 pixels. The graphic is repeated to match the size of the obstacles.

```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
    <TITLE>H5R framework demo: Flappy Bird clone</TITLE>
    <SCRIPT type="text/javascript" src="H5R.js"></SCRIPT>
  </HEAD>
  <BODY>
    <DIV id="stage" style="width: 640px; height: 480px;"></DIV>
    <SCRIPT type="text/javascript">
      var isCollision = function(a, b){
        return !(a._x >= b._x + b._width ||
            a._y >= b._y + b._height ||
            b._x >= a._x + a._width ||
            b._y >= a._y + a._height);
      };

      var TS = {
        "bird": { //size: 64x48
          img: {
            src: "bird.png",
            sprites: 3,
            fps: 8,
          },
        },
        "background": {
          img: "background.png",
        },
```

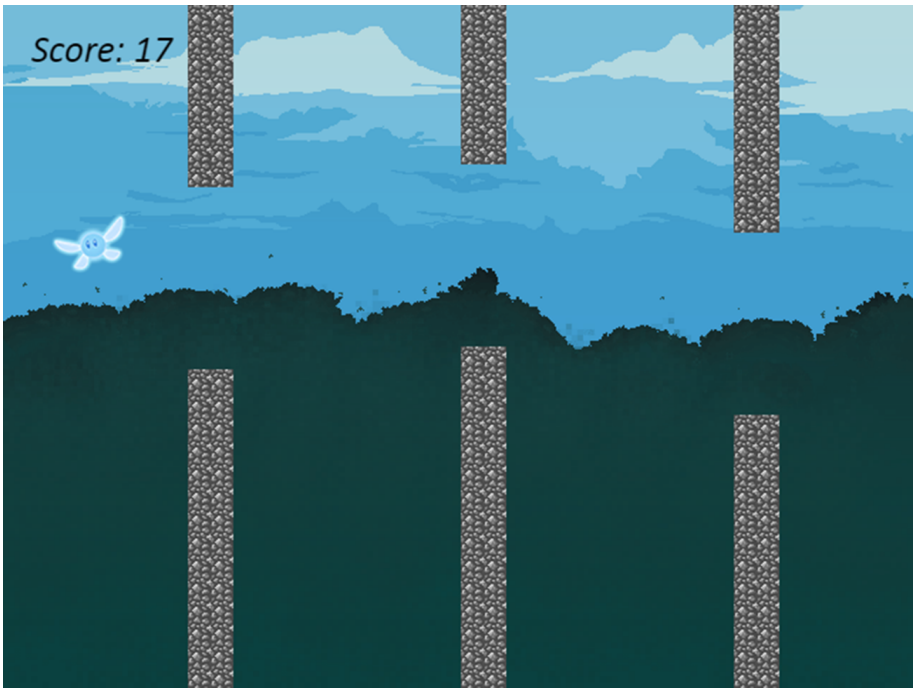

Figure A.1: A *Flappy Bird* clone implemented using the H5R framework.

```
28          "brick": { // size: 16x16
29            img: "brick.png",
30          },
31        };
32
33        var SG = {
34          bg: {
35            _img: "background",
36          },
37          bricks: [],
38          bird: {
39            _x: 32,
40            _y: 64,
41            _img: "bird",
42            _width: 64,
43            _height: 64,
44            vY: 0,
45          },
46          score: {
47            value: 0,
48          }
49        };
50
51        var storedSG = JSON.stringify(SG);
52
53        var sceneBounds = {
```

```
54          _x: 0,
55          _y: 0,
56          _width: 680, //added more space so spawned gates would not disappear
        immediately
57          _height: 480
58        };
59
60        var R = new H5R();
61        R.addStage(new H5R.Stage("stage"));
62        R.setScenegraph(SG);
63
64        var die = function(){
65          //show score and restart when player dies
66          alert("You died! Final score: " + SG.score.value);
67          R.stop();
68          SG = JSON.parse(storedSG);
69          R.setScenegraph(SG);
70          play();
71        };
72
73        var play = function(){
74          R.start();
75          SG.score._render = function(self, ctx){
76            //display score on screen
77            ctx.font = 'italic 20pt Calibri';
78            ctx.fillText('Score: ' + self.value, 20, 40);
79          };
80          SG.bird._update = function(self, dT, time){
81            //bird gravity
82            self.vY += 20 * dT * 0.005;
83            var dY = self.vY * dT * 0.005;
84            self._y += dY;
85          };
86          SG._update = function(self, dT, time){
87            //add new gate when necessary
88            if(SG.bricks.length < 1 || SG.bricks[SG.bricks.length-1].top.
        _x <= 432){
89              var x = 664;
90              if(SG.bricks.length > 0){
91                x = SG.bricks[SG.bricks.length-1].top._x + 192;
92              }
93              var gatePosition = (Math.random()*8)|0;
94              SG.bricks.push({
95                top:{
96                  _img: "brick",
97                  _x: x,
98                  _y: 0,
99                  _repeatX: 2,
100                 _repeatY: 4+gatePosition
101               },
102               bottom:{
103                 _img: "brick",
104                 _x: x,
105                 _y: (12+gatePosition)*16,
```

```
106                 _repeatX: 2,
107                 _repeatY: 18-gatePosition
108               },
109               scored: false,
110               _update: function(self, dT, time){
111                 //move gate
112                 var newX = self.top._x - dT*0.1;
113                 self.top._x = newX;
114                 self.bottom._x = newX;
115                 //score when gate is passed
116                 if(newX <= 48 && !self.scored){
117                   SG.score.value++;
118                   self.scored = true;
119                 }
120               },
121             });
122           }
123           //check collisions
124           if(SG.bricks.length > 0){
125             //remove gate when out of screen
126             if(!isCollision(SG.bricks[0].top, sceneBounds)) SG.bricks.
      shift();
127           }
128           if(!isCollision(SG.bird, sceneBounds)){
129             //player vs. scene bounds
130             die();
131           }
132           for(var i = 0; i < SG.bricks.length; i++){
133             //player vs. gates
134             if(isCollision(SG.bricks[i].top, SG.bird)) die();
135             if(isCollision(SG.bricks[i].bottom, SG.bird)) die();
136           }
137         };
138
139         window.onclick = function(){
140           SG.bird.vY = -50;
141         };
142       };
143
144       R.setTileset(TS, function(){
145         play();
146       });
147     </SCRIPT>
148   </BODY>
149 </HTML>
```

References

Literature

[1] Navid Ahmadi, Mehdi Jazayeri, and Alexander Repenning. "Performance Evaluation of User-created Open-web Games". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC '12. Trento, Italy: ACM, 2012, pp. 730–732. URL: http://doi.acm.org/10.1145/2245276.2245414 (cit. on p. 47).

[2] Matti Anttonen et al. "Transforming the Web into a Real Application Platform: New Technologies, Emerging Trends and Missing Pieces". In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC '11. TaiChung, Taiwan: ACM, 2011, pp. 800–807. URL: http://doi.acm.org/10.1145/1982185.1982357 (cit. on p. 49).

[3] Andreas Anyuru. *Professional WebGL Programming: Developing 3D Graphics for the Web*. 2nd. Birmingham, UK, UK: Wrox Press Ltd., 2012 (cit. on p. 8).

[4] Mark Claypool and Kajal Claypool. "Perspectives, Frame Rates and Resolutions: It's All in the Game". In: *Proceedings of the 4th International Conference on Foundations of Digital Games*. FDG '09. Orlando, Florida: ACM, 2009, pp. 42–49. URL: http://doi.acm.org/10.1145/1536513.1536530 (cit. on p. 30).

[5] Aiman Erbad, Norman C. Hutchinson, and Charles Krasic. "DOHA: Scalable Real-time Web Applications Through Adaptive Concurrent Execution". In: *Proceedings of the 21st International Conference on World Wide Web*. WWW '12. Lyon, France: ACM, 2012, pp. 161–170. URL: http://doi.acm.org/10.1145/2187836.2187859 (cit. on p. 17).

[6] Stephan Herhut et al. "River Trail: A Path to Parallelism in JavaScript". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 729–744. URL: http://doi.acm.org/10.1145/2509136.2509516 (cit. on p. 17).

[7] Shusuke Okamoto and Masaki Kohana. “Load Distribution by Using Web Workers for a Real-time Web Application”. In: *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*. iiWAS ’10. Paris, France: ACM, 2010, pp. 592–597. URL: http://doi.acm.org/10.1145/1967486.1967577 (cit. on p. 17).

[8] Chengyuan Peng. “SCALABLE VECTOR GRAPHICS (SVG)”. In: *Tik-111.590 Research Seminar on Interactive Digital Media*. Helsinki, Finland: Helsinki University of Technology, 2000. URL: http://www.tml.tkk.fi/Opinnot/Tik-111.590/2000/Papers/svg.pdf (cit. on p. 14).

[9] Steve Probets et al. “Vector Graphics: From PostScript and Flash to SVG”. In: *Proceedings of the 2001 ACM Symposium on Document Engineering*. DocEng ’01. Atlanta, Georgia, USA: ACM, 2001, pp. 135–143. URL: http://doi.acm.org/10.1145/502187.502207 (cit. on p. 14).

[10] Ben Shneiderman. “Response Time and Display Rate in Human Performance with Computers”. In: *ACM Comput. Surv.* 16.3 (Sept. 1984), pp. 265–285. URL: http://doi.acm.org/10.1145/2514.2517 (cit. on p. 47).

[11] Gojko Vladić. *Evaluating Web browser graphics rendering system performance by using dynamically generated SVG*. Ed. by Darko Avramović. 2012, Online–Ressource (cit. on p. 14).

Online sources

[12] URL: http://img.fettspielen.de/img/prepages/000000082/ogame%5C_screenshot342%5C_n.jpg (cit. on p. 5).

[13] URL: http://kosmix.co/static/2455a2306c62fe3f607d932e21185b01.jpg (cit. on p. 6).

[14] URL: http://www.stagetwo.eu/Attachment/4689-minecraft-5-jpeg/ (cit. on p. 6).

[15] URL: http://www.indiegames.com/images/timw/roadeo2a.png (cit. on p. 7).

[16] URL: http://3.bp.blogspot.com/-8qcBaKla9oE/TcIgCBW6hWI/AAAAAAAAAVM/A-NQJ0tU0G4/s1600/globescreenshot.png (cit. on p. 9).

[17] URL: http://esotericsoftware.com/files/dragon-screenshot.jpg (cit. on p. 20).

[18] URL: http://impactjs.com/files/weltmeister-new.png (cit. on p. 27).

[19] John Bauman, Brian Salomon, and Pixel Engineers. *Chromium Blog: GPU accelerating 2D Canvas and enabling 3D content for older GPUs*. 2012. URL: http://blog.chromium.org/2012/02/gpu-accelerating-2d-canvas-and-enabling.html (cit. on p. 9).

[20] Clay.io. *HTML5 Game Engines - Find Which is Right For You*. 2014. URL: http://html5gameengine.com/ (cit. on pp. 16, 21, 24, 25).

[21] Lucian Constantin. *Researchers: Java's security problems unlikely to be resolved soon*. 2013. URL: http://www.pcworld.com/article/2030778/researchers-javas-security-problems-unlikely-to-be-resolved-soon.html (cit. on p. 4).

[22] Oliver Cueilliez. *An Original Approach to Web Game Development Using SVG*. 2011. URL: http://www.svgopen.org/2011/papers/14-An%5C_Original%5C_Approach%5C_to%5C_Web%5C_Game%5C_Development%5C_Using%5C_SVG/ (cit. on p. 7).

[23] CVE. *Adobe Flash Player : CVE security vulnerabilities, versions and detailed reports*. 2014. URL: http://www.cvedetails.com/product/6761/Adobe-Flash-Player.html (cit. on p. 4).

[24] Alexis Deveria. *Can I use... Support tables for HTML5, CSS3, etc.* 2014. URL: http://caniuse.com/%5C#feat=webgl (cit. on pp. 8, 9, 16).

[25] EaselJS. *A Javascript library that makes working with the HTML5 Canvas element easy*. 2014. URL: http://www.createjs.com/%5C#!/EaselJS (cit. on p. 24).

[26] Stefan Haustein, Joel Webber, and Ray Cromwell. *Quake II GWT Port*. 2010. URL: https://code.google.com/p/quake2-gwt-port/ (cit. on p. 10).

[27] Paul Irish. *requestAnimationFrame for smart animating*. 2011. URL: http://www.paulirish.com/2011/requestanimationframe-for-smart-animating/ (cit. on p. 41).

[28] Seb Lee-Delisle. *HTML5 canvas sprite optimisation*. 2011. URL: http://seb.ly/2011/02/html5-canvas-sprite-optimisation/ (cit. on pp. 41, 42).

[29] Afshin Mehrabani. *~~ is faster than Math.floor()*. 2012. URL: https://coderwall.com/p/9b6ksa (cit. on p. 42).

[30] Paul Neave. *requestAnimationFrame*. 2013. URL: http://creativejs.com/resources/requestanimationframe/ (cit. on pp. 40, 41).

[31] *PlayN Quake II Demo Port*. URL: http://quake2playn.appspot.com/ (cit. on p. 10).

[32] Rix Library. *JavaScript Math.floor Optimization*. 2014. URL: http://rix.li/javascript-math-floor-optimization/ (cit. on p. 42).

[33] Boris Smus. *Improving HTML5 Canvas Performance - HTML5 Rocks*. 2013. URL: http://www.html5rocks.com/en/tutorials/canvas/performance/ (cit. on pp. 41, 43).

[34] StatCounter. *Top 5 Desktop Browsers from Aug 2013 to Aug 2014*. 2014. URL: http://gs.statcounter.com/%5C#desktop-browser-ww-monthly-201308-201408 (cit. on pp. 16, 17).

[35] W3C et al. *HTML5 - Candidate Recommendation*. 2014. URL: http://www.w3.org/TR/html5/ (cit. on pp. 8, 14).

[36] W3Schools. *Browser Statistics*. 2014. URL: http://www.w3schools.com/browsers/browsers%5C_stats.asp (cit. on p. 16).

[37] Nicholas C. Zakas. *Better JavaScript animations with requestAnimationFrame*. 2011. URL: http://www.nczonline.net/blog/2011/05/03/better-javascript-animations-with-requestanimationframe/ (cit. on pp. 40, 41).