MASTERARBEIT

Titel der Masterarbeit

“Game Engine for web-based Serious Games”

Verfasserin

Katharina Meusburger, BSc.

angestrebter akademischer Grad

Diplom-Ingenieurin (Dipl.-Ing.)

Wien, 2014

Studienkennzahl lt. Studienblatt:	A 066 935
Studienrichtung lt. Studienblatt:	Masterstudium Medieninformatik
Betreuer:	Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Contents

1 Introduction

1.1 Motivation

Nowadays the number of children suffering from a psychological illness is increasing [9]. These impairments show first symptoms at early ages and may cause difficulties in communication skills, like emotion recognition or social life. Proper treatment at a young age may significantly improve the children's behavioural disorders. Especially for children alternative therapy strategies are very welcomed, which introduces the field of serious gaming. Research [1] reveals that children with cognitive disabilities are amenable for technology or computer games - they react to game technologies in a positive way. A serious game uses this advantage and is meant to be more than just a simple game. By incorporating a psychological treatment in the game context, the child has fun while playing and is even treated through the therapeutic background. Unfortunately, existing games for emotion recognition are insufficient. Not only because proper game aspects and user needs are not sufficiently addressed, but because there is no game engine that is especially designed for developing serious games. Game engines tend to differ according to the game play scenario, i.e., an engine for a strategy game has a different architecture than an engine for a shooter game [21]. The strong relation between a game an its underlying engine complicates a generalization of the term game engine. Besides common components, which constitute a basic game engine, we further present features for serious gaming. These features enable data collection, either directly through a questionnaire or indirectly through logging. With the web as our target platform, we use new HTML5 and CSS3 concepts like canvas rendering, audio or video elements and keyframe animations, which can even replace Flash animations [45].

In our master thesis project we met the challenge of developing a web serious game, which is used to train the emotion recognition skills of children. While researching and diving into HTML5 and JavaScript concepts, we evaluated several existing web frameworks or game engines that are mostly designed for building basic indie games for the web. However, we could not find any framework that mainly focuses on developing serious games. For the purpose of inventing a serious game for emotion recognition that incorporates a logging tool to store data of the game play and the player's status, we have built our own HTML5 game engine, whose presentation is the aim of this master thesis.

1.2 Outline

In this work we first give a short introduction in the psychological background of serious gaming, emotion recognition and the requirements that arise for proper game development in this field of interest. Afterwards, we describe the main components of which a game engine is generally composed. Moreover, we look at some existing web game engines and evaluate them according to these engine modules. Before we present our game engine *Jumru 5s* in detail, we explain important JavaScript, HTML5 and CSS3 concepts that build the fundamentals of our work. Furthermore, we show how we successfully used the engine based on the example of our serious game *Emo-Jump*. At the end we evaluate the engine by means of the game and provide some performance statistics. Lastly, a conclusion sums this work up.

1.3 Publication

Fortunately, our work is accepted as a full paper by the Programme Committee for the 5th International Conference on SGDA and is published in Springer LNCS 8778:

N. Schweiger, K. Meusburger, H. Hlavacs, M. Sprung,
Jumru 5s - A Game Engine for Serious Games,
The Fifth International Conference on Serious Games Development & Applications (SGDA 2014) 9-10 Oct 2014, Berlin, Germany. (`http://ddsgsa.net/sgda/`)

2 Background

2.1 Psychological Background

ASDs - Autism Spectrum Disorders

As stated earlier in the introduction, the number of children with ASDs has grown during the last couple of years. From 2000 to 2010 the number of children suffering from ASDs has more than doubled and increased from 1 in 150 to 1 in 68 within 10 years [9]. Having a closer look on the term ASDs, CDC[1] provides following definition:

> *"Autism spectrum disorders (ASDs) are a group of developmental disabilities that can cause significant social, communication and behavioural challenges. People with ASDs handle information in their brain differently than other people."*

Speaking of spectrum disorders, people suffer from ASDs at different degrees, though they share some common symptoms that normally complicate social interactions. Symptoms can appear at early ages of childhood and may even start before the age of three. Though there is no cure for ASDs, research shows that an appropriate treatment can alleviate the symptoms. Further research studies demonstrate that improving emotion comprehension skills at an early age may have significant beneficial effects on a person's life and social interactions [10].

People with ASDs struggle in their social life and communication because they have difficulties in interpreting facial or generally non-verbal expressions. A major issue for these people is also to keep eye contact. All these factors lead to difficulties in emotion recognition [1]. Though ASDs may appear in different ways [10], we focus solely on *emotion recognition* and how to train this ability.

[1]Centers for Disease Control and Prevention

Training of Emotion Recognition

For the *Cognitive Atlas* Russ Poldrack [36] defines emotion recognition as *"the process of assigning an emotion to one of the discrete categories of emotion available in a particular culture"*. For our training study we are developing a web game in cooperation with the Department of Clinical Child and Adolescent Psychology of the University of Vienna[2]. The game has two aims: on the one hand it is used to evaluate to what extent children are able to recognize emotions and on the other hand the game shall train the children's emotion recognition skills. To provide a basis for building this online tool we implemented our own web engine, which is especially designed for serious games. The game that is built on this framework is further used to evaluate the engine.

Previous research has also focused on measuring and improving emotion recognition skills and has originated in computer-based intervention tools. For evolving appropriate game technologies, or rather development environments for this field of interest, we have to consider requirements, which a good serious game has to fulfill in order to fit the users needs.

2.2 User Needs and Design Principles

Abirached et al. [1] evaluate user needs and requirements for designing serious games that deal with emotion recognition, because a *"successful deployment of a technology for users with ASDs can only be assured by a clear understanding of end users' needs"*.

To identify requirements for designing serious games, they collect data about children with ASDs through observations and interviews while playing a simple prototype game. Throughout the whole game play process, the children are accompanied by their parents. Analysis of these interviews show that the most important psychological need is to *incorporate the context* behind the emotional statement into the game, because some children even have difficulties in interpreting basic emotions, i.e., sadness. They can only tell someone is sad if the person is crying. Other children can

[2]http://kinder-psy.univie.ac.at/

not recognize complex emotions like embarrassment, because they do not understand the emotional context behind this feeling or why someone acts or reacts in a special way. The goal of this design principle is to embed the emotion in a real-life scenario, which a child can transfer to a real social interaction. This rule also includes learning the cause and effect of an emotion.

Besides this psychological need, they identified following technological design principles that are relevant for game development and could be provided by a game engine, to take the effort off the hands of the game developer:

1. *Allow Customization:* A high level of customization allows the game to fit specific user needs. Customization should be implemented in different areas, like feedback, characters, visual elements, language, audio and video settings, etc. Additionally, the game could also consider the context of the game environment - where is the child playing the game (bus, home, school, etc.).

2. *Enable Adaptability:* The game itself should adapt to the behaviour of a child, i.e., by eye contact, speech recognition or a high level of interactivity. How important adaptability for the user's attention is, is also mentioned in the study of Finkelstein et al. [15]. They also state that the granularity of customizing the interface has a massive impact on the user's concentration. The higher the level of interactivity and adaptability is, the more attentive a user is. If the interface fits the personal interest, the user is more attracted to the game.

3. *Technology Platforms:* A design feature to keep in mind is platform independence. Supporting multiple platforms includes desktop devices as well as mobile devices, since touch screens become more and more popular. Multiple platforms also include movement-based systems such as Nintendo WII or Microsoft Kinect.

Abirached et al. [1] further analysed existing game approaches and complained that these computer-based tools assisting children with ASDs are too static in their game play. To nail these arguments down, they state that *"the fundamental need for a system to assist children with ASDs in recognizing, understanding, and generalizing facial expressions of emotions using an interactive computer-game approach has not been sufficiently addressed".*

3 Related Work

Before we present the main topic of this thesis, our implemented game engine, we have a closer look on the term *game engine* itself and explain which components basically constitute a game engine. Based on this introduction, we evaluate existing web engines that enable game development and eventually show their strengths and qualities for implementing serious games.

3.1 Game Engine

While studying literature for a definition of the term *game engine* we encountered that there is no real or common definition. In his book about general game engine architecture, Jason Gregory [21] outlines that the game engine mostly depends on the use case scenario, for example an engine for a first person shooter game might differ from an engine for a platform runner game or a strategy game. Therefore it is not a simple task to generalize a definition. The strong relation between the game engine and the game itself reveals that the engine is the *heart* of a game, because a game *depends* on its engine. An engine manages the life of a game, encapsulates all things that are *"abstract and applicable to all or most games"*, whereas a game consists of elements that are specific for a single game [41].

Literature covering game engine architecture, especially for the web, is hard to find, because technologies are changing fast. Anderson et al. even say the *"lack of literature and research regarding game engine architecture is perplexing"* [3]. They also outline that existing engines are specialized for a particular game genre, instead of concentrating on the commonality of the engine's components. Nevertheless, research proves that the architecture of an engine itself must consist of some features to enable proper game development. Though there is no standard list of features a game engine must provide, we present basic components that are generally valid. Alan Thorn [41] states that *"to some degree, the range and kind of features a developer chooses to put into an engine reflects their professional experience, design preferences, and business intentions."*.

An architecture is normally divided into manager or subsystems that are responsible for a specific task area, but interact with each other. Among others, Thorn lists following main components [41], which we think an engine should provide at least:

1. **Asset management**: A manager-tool that is responsible for loading game assets, supporting multiple types.

2. **Scene or stage management**: Loaded game data is composed and structured in layers or nodes. Some functionality must enable traversing or manipulating the game's stage.

3. **Physics engine**: The laws of physics like gravity or other forces must be included. If a player jumps or an element is thrown into the air, physical laws bring it automatically back to earth.

4. **Rendering**: After an asset manager has loaded game data, the game engine must also provide functionality to render these assets on screen.

5. **Media management**: A game engine should support multiple media types like image, video and audio. Therefore some management system to correctly play and pause media tracks is necessary.

6. **User input**: This component includes catching and processing user input according to the current game mode.

   In case of developing a game engine for serious games, an important and mostly unadressed feature is the following one:

7. **Logging**: Especially for serious gaming and for analysing the game experience of the player we need an appropriate logging tool that stores data of the game

context at specific events. Sometimes logging in serious games can include further psychophysiological tools like emotives that match the users emotional state against the game experience. By sending event-codes at specific points in the game, the recording software of the player's condition captures these signals and places markers in the recorded data. According to Nacke et al. [29] a logging mechanism has to be able to cope with different types of events. They differentiate between *in-game events*, if the player in the game does something, and *real-world events* that capture the human player's reactions, like *"shouting at the computer in anger"*. To support *real-world events*, of course further methods for measurement would have to be incorporated.

Therefore a game engine is seen as a good engine, if it keeps the game code simple. To what extent existing frameworks or game engines meet these requirements is presented in the following section. Some of these engines are also mentioned as tools on the SEGAN[1] website and often came across our mind during the development.

3.2 Quintus

Quintus[2] is a JavaScript and HTML5 game engine that is open-source, lightweight and easy to use. On the website the authors offer some tutorials on using the engine and explain basic components. In addition to the API and the source codes the developers provide a forum, in case further support is necessary. Someone can use the engine by initially creating the engine object `Quintus()`. Afterwards, access on quintus features is possible. The engine constitutes itself as a modular system, which means modules have to be included into the engine to be able to use their functionality. Just modules that are relevant for a specific game, must be loaded into the project. The modular system design also allows the development of own components by extending the basic `Quintus` object by an own function. The modularity should minimize coupling between the subsystems. The engine controls the game

[1]Serious Games Network `http://seriousgamesnet.eu/tools`
[2]`http://html5quintus.com/`

loop with `requestAnimationFrame()`. Some methods used in the engine are taken from `Undercore.js` [3], which is a library providing 80 utility methods. Quintus incorporates asset loading and forces the user to organize assets in specific a way, i.e., it provides default paths for images or audio files. Otherwise, it recommends using special file formats, for example audio must be provided in .ogg and .mp3. These specifications can be overwritten, but take the necessity off to look for supported file types in the browser. Though the engine has no progress bar visualizing the loading status, a tutorial on the website explains how to build one with CSS. The audio module supports web audio or HTML5 audio according to the browser and contains basic functions for starting, stopping and pausing sounds. Working with video streams is not supported, though the authors explain how to extend the engine in order to enable video streams, which can be found in the documentation of the code [33]. Quintus renders data using the `canvas`-element. Drawable items are sprite-objects and therefore we have to include the sprites module. A sprite contains information like the position, an asset path, a sprite sheet or a point-array that is used for collision detection. This element can also have types like friendly, active or enemy, which are defined by Quintus. A scenes module encapsulates game items like sprites. Multiple scenes can be staged independently on each other, which means sprites of one stage can not interact with sprites of another stage. Inside a stage, collisions are detected by the 2D module that consists of three parts. First, the definition of a `viewport` enables movement in the stage like a scrolling camera. Second, this module contains a `tileLayer` to create tiled backgrounds, i.e., for platform games. A tiled background takes an image and renders it subsequently. The third element is a `2d` object that applies physics (velocity, gravity, etc.) to sprites. An input module handles user inputs, concerning keyboard or mouse input or touch types, like click or drag events. Quintus can even add functionality for a joypad that allows a 4-way movement. By loading the animation module we can set keyframe animations on a sprite sheet, but the definition of tweened animations for a specific duration with an easing-value is

[3]`http://underscorejs.org/`

also possible. An interesting feature of this engine is a selector mechanism that works similar like jQuery. Quintus is a powerful engine, which relies on other libraries like `Undercore.js` or `Tween.js`[4]. It supports desktop devices as well as mobile devices.

3.3 Crafty

The Crafty[5] engine is an open source 2D engine that has some similarities with the Quintus engine. It also has a jQuery-like syntax and is based on different components or modules that can be added on entities. The developers provide a blog [6] containing posts about new releases or upcoming features. The authors also link to a forum for further questions. Generally, the API section of the homepage lists most information about the engine, because the documentation area of the website is not fully prepared. The fundamental elements in Crafty are components (the modules) and entities (the sprites). Components can be built by calling `Crafty.c(name, object)` that takes the name of the newly created component and the component itself as parameters. For example we could define a component called *BoundingBox*, add the properties *top, right, bottom and left* and some functions like `expandBox()`. The function for creating an entity works quite simple: `Crafty.e(componentList)` takes multiple existing components, adds them to the entity and returns the created entity. In contrast to Quintus, the engine offers a choice between canvas or DOM rendering. Crafty supports the loading of image or audio assets by passing the URL of the file. Unlike Quintus, this engine does not provide standard audio formats. An animation module offers tweenings and keyframe (sprite) animations. By applying the 2D module, we can add gravity or collision detection mechanisms (a hitbox) to entities. Crafty is said to support key, mouse and touch input devices and provides storage mechanisms, for example to persist data even if the browser window is closed. A nice feature of Crafty is the debug module, which enables the visualization of game elements, i.e., elements are drawn with a custom alpha value or a visible bounding box.

[4]https://github.com/sole/tween.js/blob/master/src/Tween.js
[5]http://craftyjs.com/
[6]http://craftyjs.tumblr.com

3.4 PandaJS

PandaJS[7] is a modular engine and supports both mobile an desktop devices. It is based on PixiJS[8], a WebGL/Canvas rendering engine. On the project's website the engineers provide an overview of many modules and functionalities of the engine. In PandaJS different scenes consist of sprites, which can use a physics module for collision detection. Sprites can have other sprites as children, i.e., to add an item to a player. Between the scenes the engine enables dynamic asset loading of images and audio files. Further features include animations like tweenings or a particle engine, which can be used for special effects. Besides animations, the engine handles sprite sheets and offers debugging options, like bounding box visualization. It also allows to build own modules to extend the engine by specific functionalities. Technical features enable fullscreen mode or pausing the game, i.e., if another tab is open. The developers also pay attention to performance improvements and provide object pooling, which eases memory management. As further features PandaJS supports local storage possibilities and offline game play mechanisms. The engine also offers many tutorials, screencasts or code examples. In addition to the API and the sources on github, a forum is provided to communicate with other developers.

3.5 EnchantJS

EnchantJS[9] is an object oriented engine and is designed for desktop and mobile devices. It handles code asynchronously, which means execution is shut down if no user interaction takes place and is resumed if user input is captured again. EnchantJS consists of a core library that can be extended through further plug-ins i.e., to provide further interactive input functionality or UI elements. The game is set up by initializing the engine's core object with the screen's width and height. Asset loading is implemented by a `preload()` function, which can take multiple asset paths as parameters.

[7] `http://www.pandajs.net/`
[8] `http://www.pixijs.com/`
[9] `http://enchantjs.com/`

These assets are not loaded until the core-object is started. Besides images, the engine can also preload audio data. Due to the object oriented design, scenes consist of multiple node-elements, which serve as prototypes for entities or groups. Entities and nodes are abstract objects and represent DOM-elements. Groups can contain an arbitrary number of other nodes, which is a necessary function for the scene, why a scene itself inherits from the group. The entity serves as a parent for sprites, for maps (tiled backgrounds) and for labels, i.e., to show scoring or health information. Collisions of entities are detected by intersection or containment (`intersect(), within()`). Though it is allowed to define multiple scenes, just one is visible in the display object tree. Predefined images can be used for characters, tiled maps, etc., which can be drawn either by DOM or canvas rendering. Further plug-ins enable avatars, webGL or box2D or handle user input and interface elements. Simple animations like tweenings are also supported [27]. EnchantJS provides a coding environment called *code.9leap*[10], which shall simplify coding with the engine, even if someone does not have much JavaScript knowledge. After registering, a game coder can also test and share a developed game in this environment. On the website the developers of EnchantJS have listed some publications about the engine like books and hands-on guides, which also describe basic JavaScript concepts. Moreover, the website provides an API section and code samples as an entry point.

3.6 Collie

Not listed under the SEGAN tools, but also a noteworthy framework for creating HTML5 games is Collie [11]. Collie is designed for desktop and mobile devices and therefore supports both DOM and canvas rendering. Based on the detected device, an appropriate rendering method is automatically chosen. Collie describes itself not as a game engine, but as *"the most optimized JS library for mobile"*, why the authors present some performance statistics on the website, showing the strengths for mobile

[10]`http://code.9leap.net/`
[11]`http://jindo.dev.naver.com/collie/`

devices. Additionally, they list some tutorials and demo games as an introduction and provide an API and a link to a forum for further help and information. Collie is also an object oriented library that is based on `Components` and `DisplayObjects`, where the latter are important objects in the framework. A display object is actually derived from the component and represents a drawable element, like images or shapes, i.e., circles. Display objects are just visible on the screen when they are added to a layer component or another display object, why they can contain children of the same type. Elements are drawn by passing the layer to the renderer. Display objects can get animated, for example if they are repeated or put in transition. An `ImageManager` takes care of preloading and handling images, sprite sheets and tiled maps. However, playing further media like audio and video is not possible. Simple collision detection is enabled by a `Sensor` object, but the library box2D is also supported, which can add further physics to the game. For a more accurate collision detection, the hit area of a display object can be set to a specific scope, for example a polygon shape. For debugging options, Collie offers a minified tool script and a FPS console object that can be used for *"checking FPS on screen on demand"*.

3.7 LimeJS

We started developing our serious game with LimeJS[12], a lightweight open source framework, which can be found on github. It uses the Google Closure library[13] and Python to build and minify projects, which is explained on the website and serves as a nice entry point. Besides, the website lists some tutorials, demo games and the API to start working with the engine. LimeJS is an object oriented engine and defines a basic `lime.Node`-element, from which every other object in the scope is derived. On top of the DOM-tree a `director` object is placed, which is composed of scenes. Scenes can contain an arbitrary number of layers, sprites or shapes. Drawable items have a `lime.Fill`-element that can either be a single frame, a color-value or a path

[12]`http://www.limejs.com/`
[13]`https://developers.google.com/closure/library/`

to an image object. LimeJS itself does not offer any asset preloading mechanisms directly in the code, though the developers mention how to add a .manifest-file for downloading assets before the game starts. The engine supports DOM and canvas rendering, which can be explicitly set to each node. The developers state that DOM rendering and manipulating CSS properties should be used if there are lots of animations, or if the game is designed to work on mobile devices. Canvas rendering supports polygon shapes and should be used for static elements. Though LimeJS does not contain any debugging mechanisms like other engines, it is possible to define a `stroke`-element, which paints the node's boundaries. The engine provides basic CSS animations, like moving, scaling, rotating, color changes or fading that can be looped or put in a sequence. For media management audio streams are supported, but no video streams. The engine offers a basic button element, which is useful for simple game development. Further event management and input types like keyboard, mouse or touch are enabled with the help of the Google Closure library and is explained on the LimeJS website. The Closure library also provides the bounding box for nodes, which is used for collision detection. As a future task, the developers of LimeJS also want to incorporate the box2D library and even provide some demo tests in the github repository. Besides these general components, the engine offers an automatic adjustment to the screen resolution, which is an interesting feature for mobile devices. It also provides a schedule manager, which automatically controls the game loop using `requestAnimationFrame()`. Though LimeJS is easy to learn and has an active community, we stopped working with the engine when we experienced difficulties, for example when resizing the window. Suddenly the sprite animation of our player running on the ground froze, animations did not stop correctly or we had to hack elements like the `PauseScene`, which is shown while the game is paused. When we started directly coding into the framework files, we decided to implement an own engine for serious web games that would also incorporate a logging module, which all the presented frameworks do not have.

3.8 Summary

To summarize this section, we see that the presented engines generally fall into two categories, each with its strengths and weaknesses. EnchantJS, Collie and LimeJS are object oriented engines, based on a prototypal inheritance model providing functionality in different related hierarchies. The prototypal inheritance is further explained in Section 4.1. The object oriented engine design is a classical approach, which adds more functionality with each layer in the object tree. Due to the prototypal inheritance, the tree-based composition results in "is-a" relationships. Problems arise if the tree and its nodes grow large. Changes in the base node may affect all other nodes in the tree and could cause unexpected consequences [24] [43] [46]. Other engines, like Quintus, CraftyJS and PandaJS are modular or component based which means functionalities are encapsulated in independent and reusable components that can be added to game objects. Each entity in the game is now *"the sum of its parts"*, as Mick West [46] outlines. By using this approach the connection between game objects and components is often referred to a "has-a" relationship. If interfaces of the components are well modeled, the modules are decoupled and complexity is eventually reduced [24] [43] [46]. Modular engines are probably easier to extend by own functionality, by just adding more components and load them when needed. On the other side, the object oriented engine EnchantJS provides a long list of plug-ins that can be loaded into the engine to enable further functionalities. All of the engines are still updated quite regularly, as listed in Table 3.1. The table shows a summary of the engines, when they were last updated (either on github or the website) and provides a short description. All of the projects offer enough tutorials, code samples or documentations to start working with the engine.

Most of the engines provide asset loading mechanisms, though LimeJS does not provide a preloader. Because all engines support mobile devices, all of them implement DOM or canvas rendering, except Quintus, which is just based on the canvas element. PandaJS embeds PixiJS, a rendering engine, which also supports webGL.

Due to the mobile support, functionalities handling user input is provided at different degrees, including keyboard, mouse and touch events like dragging. Quintus for example even supports joypads by loading an appropriate module. According to data types, all engines support image types. Moreover, all engines except Collie support audio streams, but none of them incorporate video streams. For enabling physics some use the library box2D, whereas others implement their own basic physics module or functionalities. Every engine offers at least basic animations and tweenings, including sprites or CSS3 animations. PandaJS even contains a particle engine that is used for special effects. Some of the projects provide extra features for debugging (mostly visualizations or console output) like Crafty, Collie or EnchantJS, which has an own coding environment. Crafty and PandaJS enable offline storage mechanisms to play a game even when the site is down.

Engine	**last updated**	**Summary**
Quintus	28 days ago on github	modular engine supporting various devices, canvas rendering
Crafty	12 days ago on github	modular engine, supporting DOM and canvas rendering
PandaJS	13 days ago on github	modular engine, supports canvas and WebGL rendering, provides various animation tools
EnchantJS	20 days ago on github	object oriented engine, supports canvas, WebGL and DOM rendering, incorporates plug-ins
Collie	Apr 11, 2013 on website	object oriented library, automatically chooses appropriate rendering method according to device
LimeJS	Jun 1, 2014 on github	object oriented engine, supporting DOM and canvas rendering, no asset loading mechanisms

Table 3.1: Overview of game engines, Status: Aug 11, 2014.

4 Technological Aspects

Our engine is built on the web technologies JavaScript, HTML5 and CSS3. A difficult aspect, with which we have to deal when working with these technologies, is cross-browser compatibility. Different browsers have different implementations of JavaScript or CSS standards that can complicate the development progress. Therefore a framework or a game engine should provide some functionalities to avoid cross-browser issues [26]. The next sections deal with the presentation of the core features of these three technologies that are relevant for our framework.

4.1 JavaScript

JavaScript is a dynamic scripting language and consists of simple types like numbers, strings and booleans. Simple types also include `null` and `undefined`. Former describes a value to assign to a variable and indicates *"no value"* , whereas the latter is the value of a variable, which is declared but not yet instantiated [16]. Everything else in JavaScript is an `object`, like arrays or functions. Objects contain properties, which are key-value elements [11]. Whenever values are created in JavaScript, memory is allocated as long as we use these values (read, write). If these values are not needed anymore, memory is freed again, but identifying memory that is not used anymore is an undecidable task [28]. This process of automatic memory management is called garbage collection. Generally, memory is reclaimed by the browser if objects have no more references. Problems arise if elements are referenced in a circular or cyclic way [5].

Namespace

A common advice in JavaScript is to avoid using global variables, because they are *evil* and *"visible in every scope"* and can be altered at any time, as Douglas Crockford [11] mentions. A global variable can either be placed outside of a function, like

```
var globalVariable = value;
```

or it can be appended to a global object:

```
window.variable = value;
```

As we can see, this could cause interferences with other applications or libraries that call the same variable. According to Crockford, global variables *"weaken the resiliency of programs and should be avoided"* [11].

To avoid or to minimize this problem, we implement an imitation of namespacing in JavaScript. Therefore we define one single global object that represents our namespace. Afterwards, all other relevant objects are encapsulated under the namespace object, which reduces the number of global objects to a minimum [20].

```
var jumru = jumru || {};
```

By implementing this strategy we guarantee that if the object `jumru` is already defined, we just extend it by further information and prevent from overriding it [25].

Inheritance

In JavaScript every function is an object and each object is linked to a prototype object. From this prototype it can inherit properties, which means the prototype serves as a parent object. All objects are linked to an `Object.prototype`, which comes standard with JavaScript. The prototype, which serves as a parent object, can be self-defined [11].

By setting an own prototype as a parent object, we can implement **prototypal inheritance**, or prototype chains:

```
jumru.inherits = function(Parent, Child) {
  function F() {}
    F.prototype = Parent.prototype; //set prototype to the parent's prototype
    Child.prototype = new F();
    Child.prototype.constructor = Child; //reset to child's constructor
};
```

The function `jumru.inherits` takes a parent object and a child object as parameters and connects their prototypes. First, a temporary dummy object `F` is created, which is an empty "middle" object. The empty `F` object now inherits all properties from the parent's prototype. Creating the middle object is a necessary step, because otherwise if we would change the child's prototype, this might also affect the parent's prototype. If multiple objects inherit from the same parent, this may lead to changes in other children too. Afterwards, the child's prototype is set to a newly created instance of `F` (which was created by the parent's constructor). At the end we have to reset the constructor-property again, so that if we create an instance of the child, its own constructor and not the constructor of `F` is used. David Shariff [37] provides a visualization of the prototypal inheritance, which is mapped to our example in Figure 4.1:

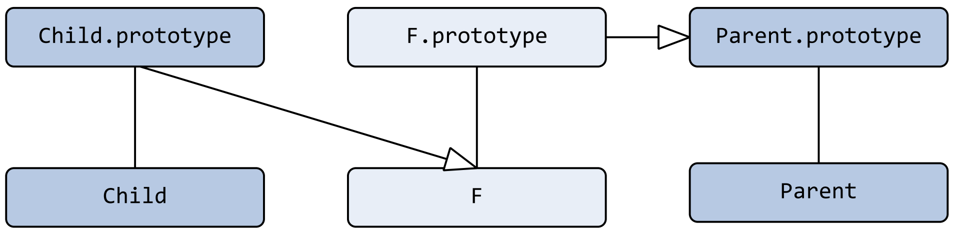


Figure 4.1: Inheritance visualization

RequestAnimationFrame

Normally, there exist two options to loop and schedule an animation in JavaScript: `setInterval(f, ms)` or `setTimeout(f, ms)`, which take the function `f` to loop and the `milliseconds` as arguments. By using this strategy the update rate is fixed and determined by the programmer, which may cause choppy animations or jittering, because frames can get lost. Furthermore, this may downgrade the battery life of the device, because the animation is also executed even if the website is not visible (another tab is open, browser window is minimized, etc.) [30]. To avoid this problem the W3C defined a timing API that lets the user-agent take care of updating the animation at a qualified rate to achieve smooth animations. Therefore they define a `window.requestAnimationFrame`-function, which gets a callback as parameter and can automatically schedule the function. Inside the callback, `requestAnimationFrame(callback)` has to get called again, to keep the animation playing. Even if the window or tab looses focus, this approach can significantly throttle the animation to save CPU power [34]. The browser itself redraws the next frame, when it is ready to do so and not at a fixed update rate. We use the strategy implemented by David Geary [19], whose callback-function gets a `time`-value as parameter, though he states that there is no definition available explaining what exactly this time value represents. He just notes that it means the same for all browsers and helps us to calculate the time passed since the last animation frame. This value is called `now` in our engine code and deals with FPS and game looping.

The `requestAnimationFrame`-approach has been originally invented by Mozilla, was adapted by Webkit and finally standardized by W3C. Different browser vendors lead to different prefixes in the code, which makes it quite difficult to provide the same functionality for all browsers. Therefore the HTML5 community developed a polyfill to get the best out of each browser when using this animation-function. The polyfill even provides a workaround with `setTimeout`, if `requestAnimationFrame` is not supported [19].

Event Handling

During the course of a game, events may occur at different points, i.e., if a button is clicked, if an animation has started or stopped or if a media element is played or paused, etc. An event is always linked to an object (a DOM-element) that implements the `EventTarget` interface, which defines methods for dispatching events or adding and removing event listeners for them. If an event is dispatched, a listener or event handler function is executed and processes code according to the thrown event. Events themselves are objects in JavaScript, which are dispatched either through user interaction, like `onmousemove`, `onclick`, etc., but can also be thrown by the application itself. For such a special purpose `CustomEvents` are created. The constructor of a custom event takes a `type` as parameter, which defines the name of the event [42]. How we use custom events in our engine's code, is further described in Section 5.2.

Minification

In order to achieve better performance and to minimize JavaScript code data, we use Google Closure's minification mechanisms to remove unused or unnecessary lines of code [38]. All JavaScript files of our engine are compiled into one single file. We use this same mechanism for our game code too.

4.2 HTML5

The Hypertext Markup Language is the *"core language of the World Wide Web"*. HTML5, or more precisely the current Version 5.1, which is still work in progress [44], shows some differences to HTML4 according to document structure, new attributes for the input type, a progress element and tags for drawing and managing media content. Concerning the document structure and organization, there exist new tags that define a `section` or `header` and `footer`, etc. The W3C list new and interesting types for the input element, like `tel, email` or `url`. By specifying a pattern for the input element, an own RegExp can be applied to validate the user input. In addition to these few elements, a progress element can visualize the amount of loaded data. Other new tags, like `canvas, audio` and `video` are also used in our game engine. This is just an excerpt of new elements invented by the W3C [31]. We shortly describe how we embed some of these new HTML5 features in our game engine.

Canvas Rendering

The `<canvas>` element is an important component in the HTML5 specification [7] and is responsible for drawing operations within a defined region that has a width and a height. In combination with JavaScript it enables many methods for drawing images or graphics and applying transformations like rotation, translation, scaling, etc. on them. The canvas further provides pixel-based graphic manipulation, for example to grey-scale an image. The drawable entity of the canvas is the `context`, which can be retrieved by calling

```
var context = canvas.getContext("2d");
```

With the canvas we can not only draw images or simple shapes like rectangles, but also lines, arcs, curves and even text [48]. Canvas rendering is hardly relying on the GPU's power, why there exist plenty of hints and tips how to achieve a performance boost in canvas rendering. Optimization strategies that are also part of our engine are listed at the end of Section 5.7.

Audio and Video

Audio and video are both media elements, which provide predefined functions for playing or pausing media streams. An audio element can be created through JavaScript by simply calling its constructor with the `path` of the audio-file as a parameter:

```
var audio = new Audio(path);
```

For video data there is no equivalent to this constructor function. To be able to create a HTML5 video element, we use the old-fashioned way of creating DOM-elements [44]:

```
var video = document.createElement("video");
video.src = path;
```

4.3 CSS3

CSS3 (Cascading Style Sheets) is composed of different modules which provide new functionalities and extend previous CSS specifications. Important modules are media queries enabling responsive designs, or selectors for specifying elements or areas to append styles to [6]. Another new feature is the support of animations and transitions. Transitions are used to translate the style of an element to another style over a specified duration, i.e., if we move over an element. Animations provide more possibilities and can even replace Flash animations [45] [6]. Transitions are triggered *implicitly* if an underlying property of the element changes, whereas animations are *explicitly* called or applied [12]. Animations are widely used in our game play, why we describe their basic structure and functionality.

(Keyframe) Animations

An animation is applied to an element by setting the `animation-name` attribute of the `<style>`-tag. Generally an animation consists of various properties, like the `iterationCount` regulating the number of iterations, a `delay` time (in *s* or *ms*) to wait before playing the animation, or a `fill-mode`, which defines if the animation state is already applied before playing it or kept after playing it. Multiple animations can get executed on a single element but the last added element under the `animation-name` property will override other appended animations [12].

The `fill-mode`-property defines the state of an animation and how it is set to the style of an element. There exist four different fill-modes, whose functioning is visualized in Figure 4.2. The diagram shows the created animation, which has a defined start and end time (specified by setting a `duration`) and a `delay` value. Depending on the chosen fill-mode, the specified transformations by the animation are applied to the element as follows:

1. `forwards`: before the animation is playing, its specified start-styles are already appended to the element.

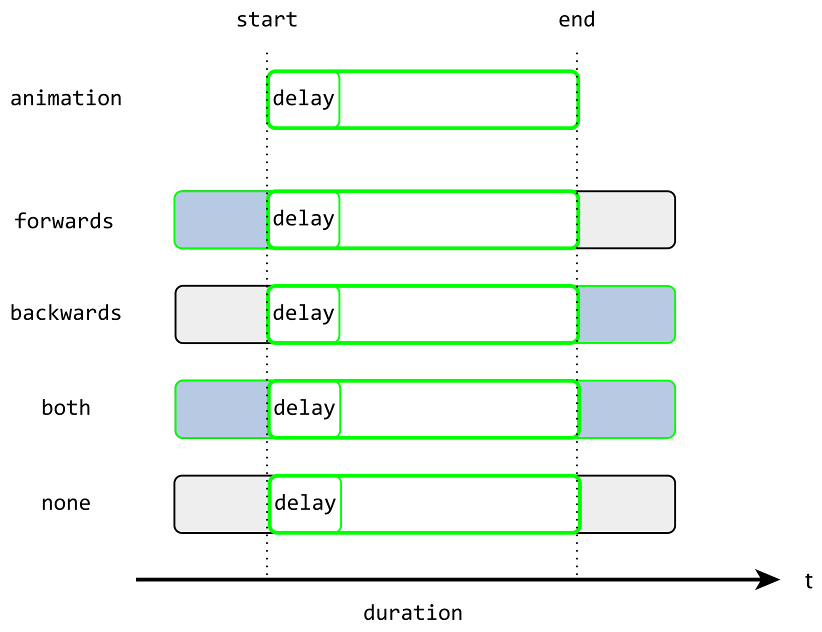


Figure 4.2: CSS3 animation fill-modes

2. `backwards`: same as forwards, but here the style defined at the end of the animation stays on the element.

3. `both`: combination of forwards and backwards.

4. `none`: style is just visible while playing the animation.

How transitions between different keyframes should look like, can be specified by setting a `animation-timing-function`. We call this attribute `easing` in the engine's code. Possible values for the transition type would be `ease-in`, `ease-out`, `linear`, etc.

Most common and simple animations are transformations like scaling, rotation, translation, etc. in a 2D and 3D context, which are applied by setting the objects `style-transform`-property. Transformations are based on matrix multiplications.

By setting the `transform-origin` as an additional property, the center of the transformation can be specified [17].

The following code example below shows how a simple animation can be set up, paying attention to different browser vendors that use different prefixes handling CSS code. The correct prefix for Google Chrome and Safari is `-webkit-`, for Mozilla Firefox it is `-moz-` and for Opera we can either use `-webkit-` or `-o-` [45]. The code creates a yellow box of size 50px x 50px, with a black border on the top. The box moves infinitely from left to right, by translating the x position between 0 and 100. During the translation, the box rolls from left to right, or more specifically from -90 degree to 90 degree.

```
<style>
  @-webkit-keyframes boxroll {
    0%   { -webkit-transform: translateX(0) rotate(-90deg)}
    50%  { -webkit-transform: translateX(100px) rotate(90deg)}
    100% { -webkit-transform: translateX(0px) rotate(-90deg)}
  }
  @-moz-keyframes boxroll {
    0%   { -moz-transform: translateX(0) rotate(-90deg)}
    50%  { -moz-transform: translateX(100px) rotate(90deg)}
    100% { -moz-transform: translateX(0px) rotate(-90deg)}
  }
  div {
    width: 50px;
    height: 50px;
    background: yellow;
    border-top: 5px solid black;

    /* append the animations on the div */
    -webkit-animation: boxroll ease-in-out 2s infinite;
    -moz-animation: boxroll ease-in-out 2s infinite;
  }
</style>
```

4.4 Summary

In this section we presented basic JavaScript concepts and innovations in HTML5 and CSS3. We showed how to use a namespace object in JavaScript to build a scope for the engine's data. We further explained how we implemented prototypal inheritance to be able to put basic properties and functions into an object, which serves as a parent for other engine objects. For game development with these three technologies we introduced the W3C's timing API and outlined differences to old-fashioned ways of looping JavaScript animations. Lastly, we mentioned how to use custom events and how to minify code, to reduce irrelevant data.

HTML5 allows native audio and video support and provides a canvas element for drawing images or for pixel manipulation. In combination with CSS3 it is possible to create flash-like animations and transformations. All these features enable the development of our game engine *Jumru 5s*.

5 Game Engine - Jumru 5s

Based on the knowledge of JavaScript, HTML5, CSS3 concepts and the core features of engine components, we introduce our implemented game engine. We start into this topic with a basic overview of the engine's architecture and its fundamental elements. Afterwards, we list how these fundamentals are implemented and related to each other.

The name of our game engine is built on the application area:

***Jum**p and **Ru**n Framework for HTML**5** **S**erious Games.*

Therefore we define a namespace object called `jumru` that establishes a scope, in which all the framework files are embedded. Figure 5.1 outlines our engines structure, which is further described on the following pages.

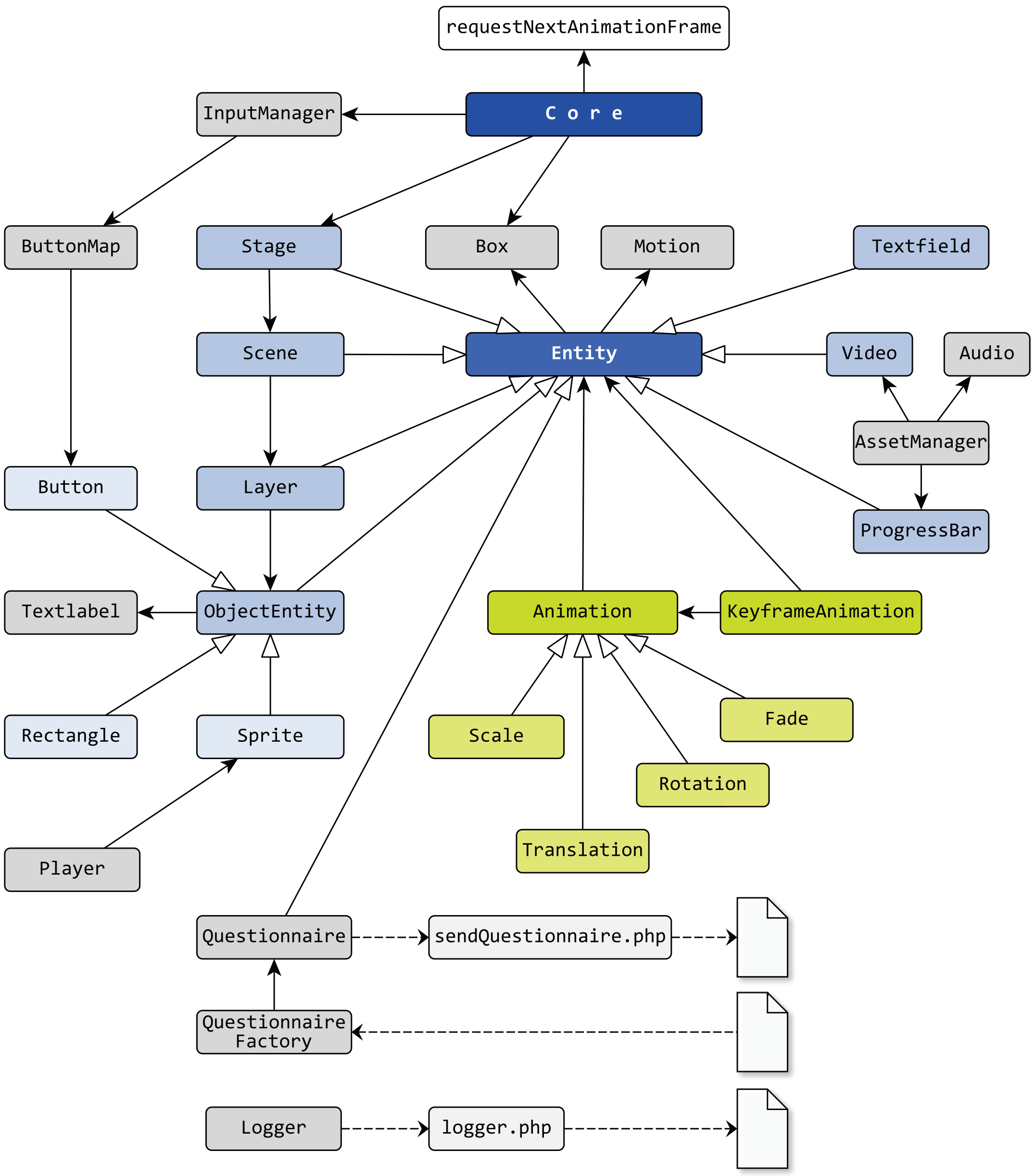


Figure 5.1: *Jumru 5s* - architecture overview.

5.1 Engine Object - jumruCore

The heart of the engine builds the `jumruCore`, which is a global object of type `jumru.Core`. This object is analogous to the game object presented in [20], stores important data of the engine and handles the basic game loop for updating and drawing elements. It can further start or pause the game and is responsible for calculating the frames per second.

The most important elements among others in the `jumruCore` are:

1. *gamemode*: to store the current game mode.

2. *mouseBox*: a 1x1 pixel box representing a bounding box for the mouse. The `mouseBox` is used to identify user clicks on visible elements in the game.

3. *inputmanager*: the input manager is responsible for handling user input (keyboard, mouse).

4. *gamestage*: the `gamestage` is the root node of the DOM-tree and holds all game elements underneath it.

5. *arrays*: the `jumruCore` has some arrays for storing different engine element types like layers, scenes or motions. These arrays ease further game processing like pausing, etc.

6. *gamewidth and gameheight*: constants for the game's size.

7. *fps, averageFps*: values for calculating the passed time since the last frame.

8. *loop and gamecode-loop*: the game coder using the engine has to define an own game loop, which controls the action that happens in the game, i.e., actions that take place according to a specific game mode. The game loop is passed to the engine, which regularly calls it inside an own loop that is responsible for updating and drawing the elements hanging in the stage. The engine's loop is scheduled with David Geary's implementation of `requestNextAnimationFrame` [19].

9. *resizeValue*: this value is calculated if the window dimensions change. In such a case all elements in the game have to get resized, to fit in the screen.

10. *questScene*: this scene is automatically set if a questionnaire is shown.

The `jumruCore`'s `gamestage` consists of displayable objects, which are composed in a tree-based structure. To get an impression on how this tree looks like and what its basic nodes are, we outline the composition of these nodes and the engines internal structures.

5.2 Composition - Stage and Scene

Our engine is generally built on the composite pattern. By using this pattern the game is composed in a tree-based hierarchy of nodes, consisting of composites and leaves. Harmes and Diaz [14] recommend using this pattern when it is necessary to group objects in some sort of hierarchy and if operations should be performed on a subset of these nodes, which means these operations are passed down a subtree. Both points are relevant for our framework: we have some nodes that inherit from each other, are ordered in a tree-based structure and we want to update and draw a subset of them.

Applying the tree-based structure onto our engine, we always have the `jumru.Stage`, called `gamestage` in the code, as the tree's root node. The `gamestage` encapsulates all other nodes in the tree. Multiple `jumru.Scenes` ly directly under the root on the next level, but only one scene can be active at any point in time. Therefore we can switch between defined scenes and always set a different `activeScene`. The scene, which is active is shown, updated and drawn. All other scenes are invisible and therefore not considered at this time. A scene on the other hand consists of an arbitrary number of `jumru.Layers`. Layers encapsulate elements of type `jumru.ObjectEntity` that represent drawable nodes. Object entities are leaf nodes and can not contain any children. Stage, scene and layer are `<div>`-containers, whereas object entities are `<canvas>`-elements that draw images, shapes or buttons.

Figure 5.2 visualizes the tree under the `gamestage`, with one scene marked as active that is updated and drawn in the game loop cycle.

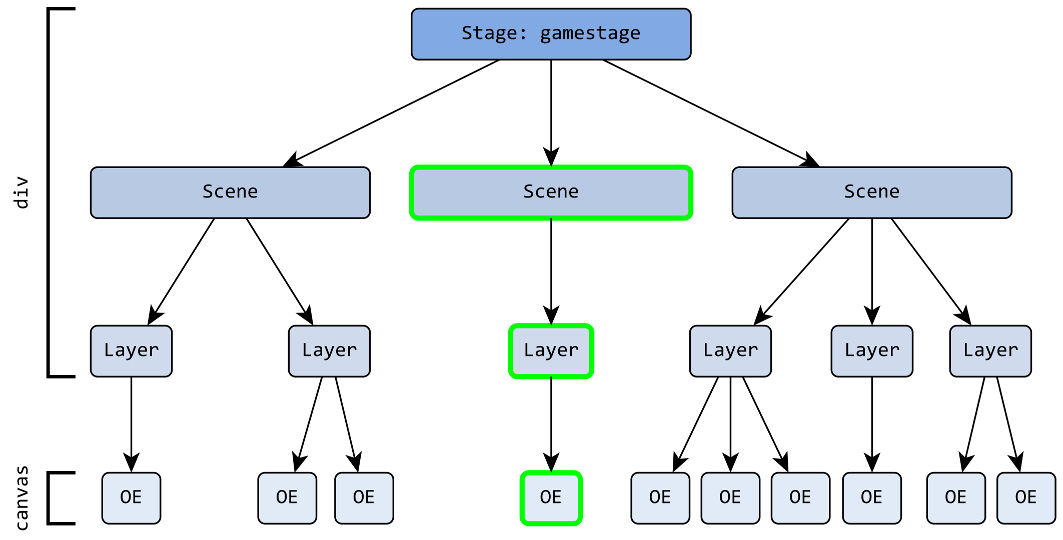


Figure 5.2: Stage tree with an `activeScene`.

Entity

The main node, from which almost every other node in the framework is derived, is the `jumru.Entity`. Entities are abstract elements, which means they should not be instantiated, though it is still possible to create entity objects[1]. They just serve as a main node providing basic functionality. When speaking of basic functionality we think of features like positioning, size and resizing, traversing mechanisms, event handling, collision detection or visibility and activity.

[1]Which we even do in our case study *EmoJump*. Barriers for example have no image, why we just created empty entities for them.

Positioning and Resizing

When positioning an element we have to differentiate between DOM-positioning and relative positioning within the DOM-element. Each entity contains a *document.element* called `domEl`, which represents the actual DOM-node. When instantiating an entity, the `domEl` can have following types that are defined in the `jumru.Entity.DomElement`-object: `canvas`, `div`, `input` or `progress`, for visualizing a progress bar.

DOM-elements have a width and height, which are set to the *document.element.style*-attribute. Here we have to pay attention to `<canvas>`-elements, because a canvas itself has a `width` and `height` attribute that define the real size of the canvas, whereas the size attributes of the `style`-tag can additionally scale the canvas dimensions:

```
//set width w and height h to the element, according to its type
if(this.domEl.tagName.toLowerCase()=="canvas"){
  this.domEl.width = w;
  this.domEl.height = h;
}
//div, input, progress
else {
  this.domEl.style.width=w+"px";
  this.domEl.style.height=h+"px";
}
```

Stage, scene, layer and object entities are set to the game's width and height in the constructor, whereas the progress bar and input field have other fixed standard values. DOM-positioning also includes a `style.position`-attribute, which is set to `absolute` in the objects constructor, so each `<div>`- and `<canvas>`-element is placed on top of each other. This enables staging of different scenes, layers and object entities.

The `gamestage` and all its children are automatically positioned horizontally in the center of the screen, which is maintained even when the window gets stretched by the user and elements have to get resized. Therefore we implemented a method in the `jumruCore` that calculates a `resizeValue` according to the current aspect ratio, which is used to resize each entity, so the game always fits the screen size:

```
/**
 * calculate aspect ratio.
 */
jumru.Core.prototype.calculateResize = function() {
  var windowWidth = window.innerWidth;
  var windowHeight = window.innerHeight;

  //scale width to fit game size
  var scaleToFitX = windowWidth / jumruCore.width();
  //scale height to fit game size
  var scaleToFitY = windowHeight / jumruCore.height();
  //take the smaller value
  this.resizeValue = Math.min(scaleToFitX, scaleToFitY, 1);
};
```

Due to the calculated value and the composite pattern, we can now successively resize the entities by multiplying their DOM-size with the `resizeValue`:

```
this.domEl.style.width = ((this.domWidth * jumruCore.resizeValue)>>0)+"px";
this.domEl.style.height = ((this.domHeight * jumruCore.resizeValue)>>0)+"px";
```

Traversing Mechanisms

As stated earlier, see Section 5.2, our engine is based on the composite pattern. Therefore the root node (the stage) contains children in a tree-based structure, built on the DOM-elements. An entity has a `children`-array, which contains all children appended to this entity, i.e., a stage contains scenes, a scene contains layers and a layer contains the leave nodes of type `jumru.ObjectEntity`. If a node is appended to another node, we set a `parent`-attribute of the childnode, draw the childnode and append it directly to the DOM-tree. We also provide a function to remove a single child, respectively to remove all children of a node. To avoid *circular references* we do not only delete the node out of the children's array and set its parent to null, but we also remove the node out of the DOM-tree. The method for removing all children just invokes the function for removing a single child for each entity stored in the current children-array.

Event Handling

Entities contain a *document.element* called `domEl`, which can catch and dispatch specific events, i.e., if a button is clicked. Therefore we define `CustomEvents`, as described in Section 4.1 that ease event handling in the engine code. For example we have 5 different events for processing button states (`click, mouseup, mousedown, focus, unfocus`) that are created in a similar way:

```
var ButtonClickEvent = new CustomEvent("ButtonClick", {
  detail:{}, bubbles: true, cancelable: true
});
```

The entity or more precisely its *document.element* serves as the event target and implements methods for adding or removing event listeners and is also able to dispatch events. The methods for controlling events on the *document.element* are:

```
/**
 * functions handling eventlisteners
 * @param {CustomEvent} event - own defined event
 * @param {Object} callback - callback function to remove
 * @param {boolean} bool - specifies the useCapture
 */
jumru.Entity.prototype.listen = function(event, callback, bool) {
  this.domEl.addEventListener(event, callback, bool);
};
jumru.Entity.prototype.ignore = function(event, callback, bool) {
  this.domEl.removeEventListener(event, callback, bool);
};
jumru.Entity.prototype.fire = function(event) {
  this.domEl.dispatchEvent(event);
};
```

Collision Detection

To be able to recognize collisions in the game, i.e., if an item is collected by a player, we need to have some collision detection methods. Each entity has a `jumru.Box` called `boundingBox` surrounding the item depending on its size and position. A box contains the values `left, right, top` and `bottom` which define the entity's boundaries. Collisions are resolved by checking if the boxes do not overlap and building the negation of it. In this case we can concatenate the if-statements through disjunction, which needs less time than checking 4 conjunctions.

```
/**
 * check if this box collides with another box
 * @param {jumru.Box} otherBox - other box which might collide
 * @returns {boolean} true, if this box collides with otherBox, else false
 */
jumru.Box.prototype.collide = function(otherBox){
  return !(
    (this.right < otherBox.left) || (this.left > otherBox.right) ||
    (this.bottom < otherBox.top) || (this.top > otherBox.bottom)
  );
};
```

We further provide a function to expand the `boundingBox`, to adjust the collectable area. The bounding box is not updated in each game loop step, because this could be an expensive and useless task, i.e., if the entity is moving through the screen. The entity provides a getter for the bounding box that automatically updates the box before returning its dimensions to the programmer.

Visibility and Activity

Each entity has a `visible` and an `active` property. Visibility determines whether the entity is displayed or not. If an element is set invisible, it is automatically set inactive as well. Just active elements are updated, drawn and used for collision detection, but just setting an element inactive does not make it invisible at the same time. In order to support different browser vendors, the `jumru.Entity.prototype.setVisible()`-function is automatically set during the pageload, because we experienced difficulties in combination with CSS3 animations in Webkit browsers when manipulating the `domEl.style.display`-property. Further explanations how we circumvented this problem, is described in Section 5.9

Object Entity

A `jumru.ObjectEntity` represents a drawable object (image or shape) and contains a `<canvas>`-element for drawing operations. If the object holds an image, further variables can influence the image's opacity or size, i.e., if the image is scaled. By manipulating the image data (each pixel), we can grey out an image by changing the RGB-values. Each object entity is marked as empty at its creation time through the boolean `isEmpty`, which is set to false, if an image is passed or if a shape is instantiated. Just objects that are not empty are drawn to the canvas. Optionally an object entity can have a text label, which can be a single string or a `jumru.Textlabel` that is drawn in the object entity's canvas. A text label is not an entity itself, but contains properties like x and y position, size, text, font- or fill-style, etc.

Offscreen Canvas

An important property of an object entity is its **offscreen canvas**, which is a separate canvas element that is not appended to the DOM-tree. After an image is passed to the object entity, the size of the offscreen canvas is set to the size of the image and the image is drawn. During the game play, the offscreen canvas is drawn onto the real canvas, which is appended in the DOM:

```
//use offscreenCanvas instead of image
this.context.drawImage(this.offscreenCanvas, this.tempX, this.tempY,
        this.scaledWidth, this.scaledHeight);
```

This significantly reduces expensive drawing operations[2] [39].

[2]http://jsperf.com/canvas-vs-imageperformance

Rounded Pixel

Generally, rendering in canvas can strongly influence the performance of the game, why we always kept an eye on improving rendering strategies. Besides the offscreen canvas and checking if an element is active, another strategy to consider when using canvas drawing operations are **rounded pixels**. If we draw on subpixels due to computations during the game play, the images start to blur because the canvas automatically applies anti-aliasing to soften the edges. Therefore we take care of rendering to **rounded pixels** [39]. As David Catuhe [8] shows, the way of rounding pixels has also an impact on performance: calling `Math.floor`, `Math.round` or `parseInt` can be very intense. Therefore we round each pixel by

```
var roundedValue = value >> 0;
```

In addition to these canvas optimization strategies, we describe further improvements in Section 5.7.

Other objects in the engine inherit functionality from the object entity. These objects include sprites, buttons and rectangles.

Sprite

A `jumru.Sprite` is an object entity that consists of a sprite sheet. The frames of a sprite sheet can be arranged horizontally or vertically and are played one after another (either forwards or backwards). Therefore the x and y positions of the frames are stored in arrays, keeping an index of the current frame to be displayed and drawn. A sprite is not redrawn in each game loop step, because this would update the sprite too often and the sequence of the frames would not be visible. For this reason a sprite has its own update rate that is defined by a variable `msPerFrame`, which is initially set to 100. This update rate guarantees that a sprite is just redrawn if 100ms have passed by since the last frame. To avoid updating the sprite in each game loop step, we have overwritten its update function, which has a counter for the passed time since the last drawn frame. The passed time is accumulated until the passed time is higher than the threshold of `msPerFrame`. In this case, the index of the current frame is increased and the sprite gets redrawn. Additional properties define if the sprite animation is played and drawn once, or started over again if the last frame is reached.

Rectangle

The `jumru.Rectangle` is a simple shape that has parameters like a `fillStyle` or a `gradient` and is used to draw i.e., a simple background in a specific color.

Button

Another element that is derived from the object entity is the `jumru.Button`, which is basically a clickable rectangle. It contains some styling parameters, i.e., to provide rounded corners or a coloured border, which is drawn if the element is focused. Furthermore, it includes two callback functions for focusing and releasing the button, i.e., for menu toggling. Focusing a button is just supported if the button is `enabled` - otherwise it is transparent and not clickable. A button can optionally contain a text string, but it also supports images as its content.

Canvas elements of object entities are set to the game's width and height, therefore clicking buttons inside the canvas does not easily work. In HTML user clicks are just caught on *document.elements* and because in the engine multiple canvas lie on top of each other, we can not decide immediately which canvas is the target of the click.

To be able to process user clicks, we deal with this problem by catching clicks on the current `activeScene`. After clicking on the scene, it checks its active children and determines which child is directly lying underneath the mouse-position. For this check the `jumruCore`'s `mouseBox` is used. As described in Section 5.1, the `mouseBox` is a bounding box that is updated with the current cursor position and is used for collision detection with the button's bounding box, to decide whether a button is touched or not. The same touch mechanism generally works for object entities. If the button is clicked via keyboard, just the click event is directly thrown at the selected button.

5.3 Player

Our engine supports jump and run or endless runner games and therefore incorporates a standard `jumru.Player`. The player is also an entity and consists of multiple `jumru.Sprites` that are stored in a sprite-array and are appended to a layer. Predefined enums control the player's type or its direction, whether he's aligned left or right. Player types, which are already provided, are `RUN` or `JUMP`, though new types can be added, i.e., if the game contains a flying version of a player. Sprites are put in the array under a specific type and an optional direction. The engine always sets one sprite as an `activeSprite` that is displayed and constantly updated. The bounding box of the active sprite is also used for collision detection between the player and other entities. The player can contain a `jumru.Motion`, to enable the movement of the player, i.e., if the user presses buttons for running left or right.

Because forces affecting physics are actually just used for the player when he is jumping, relevant variables are stored here, i.e., the `apex` of the jump, a `gravity` value, a jumping `velocity` or boolean variables checking if the player is ascending or standing on the ground. The player also incorporates a standard jump function, which is based on the formula for a trajectory parabola and describes an object that is just vertically thrown into the air. Because we are calculating in a pixel coordinate system, we have to work with opposite signs. Jumping in the game must not only take care of a different game speed, but also a variable FPS, which was not an effortless task for us.

Before the player jumps we calculate the apex of the jump, which is defined as the maximum height he can achieve:

$$apex = y - \frac{velocity_0^2}{2 * gravity}$$

The basic formula for a trajectory that is adapted in the engine in order to fit our jump function, is calculated in the following way:

$$y = velocity_0 * time - \frac{gravity}{2} time^2$$

In each game loop step we call the jump function that accumulates the `time` passed since the start of the jump, updates the `velocity` according to the `gravity` and the passed time and it updates the y `position` of the player. The time value is not initially set to $t0 = 0$, but to a high value if we have a high level speed and low FPS, because the higher the initial time value is, the higher the player jumps immediately after the start. The player has to start jumping from a lowest position, if the speed is low and the FPS is high. Setting the time value not to 0 gives the jump some kind of boost at the beginning, which is visualized in Figure 5.3 and further explained below. Afterwards, the time value, the velocity and the position of the player are accumulated by

```
//accumulate time
this.time += (this.speedValue/jumruCore.avgFps);
//update velocity and the y position of the player
this.velocity += this.gravity * this.time;
this.positionY += this.velocity * this.time;
```

which does not only include the time passed since the last frame ($1/jumruCore.avgFps$), but it also takes the level speed into account.

Figure 5.3 visualizes the jump values for two levels with different speeds and different FPS. We played the slowest level (L11) and the fastest level (L14) on a computer with a high FPS of 60 and another computer with a low FPS of 30. On the x axis we see the time for each game loop cycle and on the y axis the y position of the player, which is positioned on $y = 480px$ in a pixel coordinate system. The graph shows that the player jumps faster in the air and achieves higher jump values in the fast level (see the purple line), where he has generally a steeper ascent. The apex is the same for each level, because it is calculated solely with the initial velocity and the gravity. If the apex is reached, we smooth down the time value, so the player stays a little longer in the air, until he falls normally to the ground again. The reset of the time value results in left skewed graphs. Because of the different speeds, the player has to fall faster to the ground in the quick level, because the ground is moving faster too. If in addition

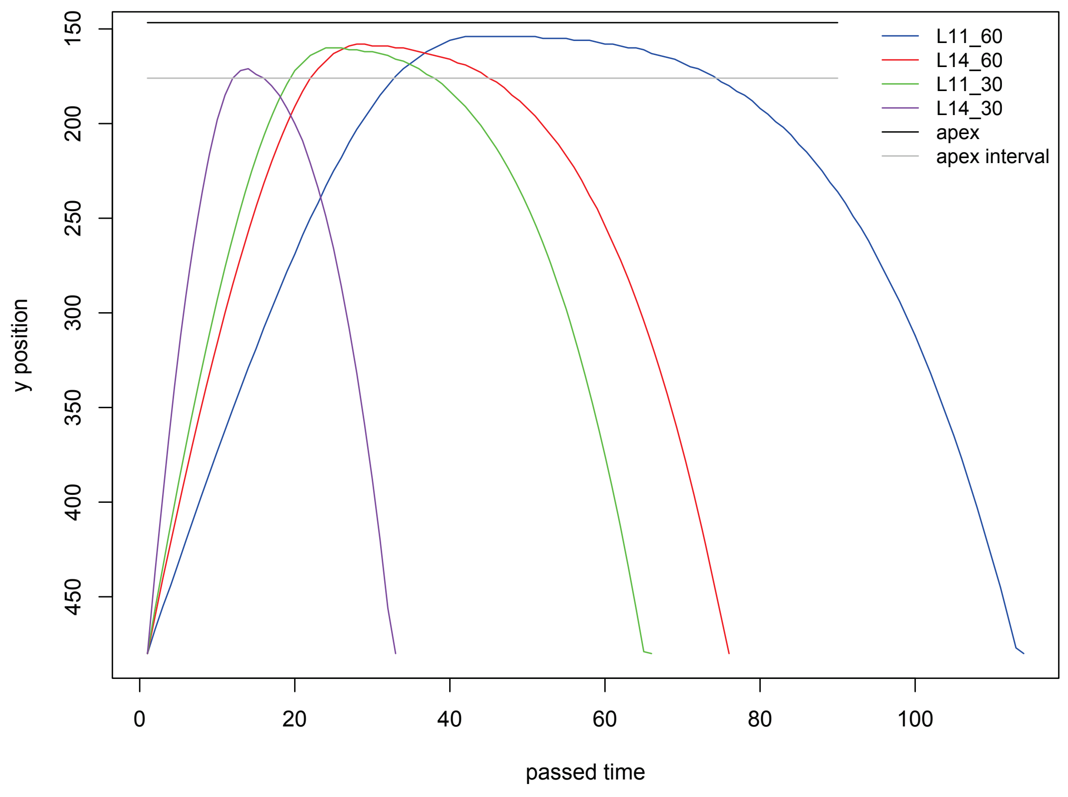


Figure 5.3: Graph of player jump in a slow and a fast level with a FPS of 60 and 30.

to the high level speed the FPS is low, the faster the animation takes place in the game, because elements are moving by a higher offset value. Therefore the jump curve is smallest for the highest level speed with low FPS (purple) and takes the longest time and tiniest jump steps for the slow level with high FPS (blue).

5.4 Media Management

Jumru 5s does not only support HTML5 audio streams, but also video elements. For both types we embed functions for playing, pausing or stopping media data. Moreover, we define a `jumru.MediaEvent.END` that is optionally dispatched if the media stream has ended.

Audio

A `jumru.Audio` object contains a HTML5 audio file object, which is the actual audio element appended to the DOM. Audio elements are automatically attached to the document during the asset loading. If an audio file is not appended to the DOM, it cannot be played. An audio element further stores some parameters to control the volume or to check if the file has to get looped.

Video

In Section 3 we saw that hardly any other engine supported the HTML5 video type. Our `jumru.Video` object inherits from the `jumru.Entity`, because it can also be positioned or sized within a layer. In opposition to the audio element, a video element is not initially appended to the DOM-tree, but has to get attached to a layer to ensure that the video can be played.

5.5 Asset Manager and Progress Bar

For further processing it is important to preload game assets before the execution takes place. Therefore we implement a `jumru.AssetManager` that solves this task and just proceeds with the code after game assets are preloaded and stored in the engine. An asset is added by passing its path to the manager, which supports video, audio and image types. By calling `loadResources(callback){...}` the added data is preloaded and stored in an asset-array. If all files are preloaded, the passed callback function is called. To be able to correctly address stored game data, the manager demands a basic URL of the website, because data is stored under its absolute path. After defining the basic URL of the site, access on assets works like the following example, because now the manager itself knows the basic URL and returns the asset stored under its full path:

```
//game code
assetManager.getAsset('images/myimage.png');
//engine code
return this.assets[this.basicURL+'images/myimage.png'];
```

For visualizing the loading progress we provide a simple `jumru.ProgressBar`. An instance of the progress bar is stored in the asset manager and is successively updated while assets are loaded.

5.6 Input Manager

The `jumru.InputManager` is an object defined in the `jumruCore` and allows the definition of menu structures that consist of different button maps. Buttons that are defined i.e., for a specific game mode, like a gender selection between girl and boy, are stored in a `jumru.ButtonMap`. A button map can store various buttons in form of a matrix and keeps the vertical and horizontal index of the current button, which enables toggling in the map with the arrow keys. For a menu structure a game developer can define a button map for a game mode. In this example two buttons for the gender selection are stored in a single row:

```
var btnGirl = new jumru.Button("BtnGirl");
var btnBoy = new jumru.Button("BtnBoy");
var genderMap = new jumru.ButtonMap().addRow(new Array(btnGirl, btnBoy));
//store the created buttonMap under its gamemode
jumruCore.inputmanager.menuMap[GENDER_SELECTION_MODE] = genderMap;
```

If a new row in the button map is added, the function `addRow()` is called again. After storing the map under its mode in the input manager, the user can toggle between the two buttons with the arrow keys, if the specific game mode is active. By pressing space, the button at the selected position is clicked. Further handler for input types are specified by calling the `addHandler()`-function. The method takes one of the predefined key types (`jumru.InputManager.Type.DOWN` or `.UP`), a mode as a string and a callback function as parameters. The input manager holds arrays for both key types.

According to the passed type, the callback function is stored under its mode identifier. Therefore different callbacks can get stored under a single mode, i.e., to catch a jump press and jump release event.

For example we could define two keyboard actions, to let the player jump after hitting the space bar or to pause the game if the user presses escape. According to the `jumruCore.gamemode`, the input manager always knows which mode is active and can proceed with the correct handler for the mode.

```
jumruCore.inputmanager.addHandler(jumru.InputManager.Type.DOWN,
  emojump.GAME_PLAY, function(evt){
    // Space jump
    if(evt.keyCode == 32) {   /* player jump */   }
    // Escape Key - Pause Mode
    if(evt.keyCode == 27) {   /* pause game */   }
  });
```

5.7 Rendering - Game loop

The `jumruCore`'s game loop is the heart of the framework and contains functions for calculating the frames per second (FPS), looping the game's game loop and updating and drawing the `gamestage`, or more precisely the gamestage's `activeScene`.

```
jumru.Core.prototype.gameloop = function(now) {
  if(jumruCore.isPlaying) {
    jumruCore.fps = jumruCore.calculateFPS(now);
    jumruCore.loop();
    jumruCore.gamestage.draw();
    jumruCore.requestId = requestNextAnimationFrame(jumruCore.gameloop);
  }
};
```

calculateFPS()

In each game loop step, we calculate the amount of time that has passed since the last animation frame. The passed time is stored in `jumruCore.loopTime`. At a normal

FPS of 60 showing one frame takes 16.6ms. By dividing 1000/`jumruCore.loopTime` we get the amount of frames that are visible in one second. The FPS should always be between 30-60, to get a smooth visible game play, otherwise the game starts jittering [19]. Therefore we catch extreme outliers and limit the FPS values.

```
jumru.Core.prototype.calculateFPS = function(now) {
  var fps = 1000 / (jumruCore.loopTime=now-jumruCore.lastAnimationFrameTime);
  jumruCore.lastAnimationFrameTime = now;
  fps = fps > 70 ? 60 : (fps > 10 ? fps : 30);
  this.calculateAvgFPS(fps);
  return fps;
};
```

By calling `this.calculateAvgFPS(fps)` we successively calculate the average of all FPS values. The resulting average frame rate for the whole game could be used for logging or for displaying it to the player [47]. After setting the frames per second, we call the game loop defined by the user by calling `jumruCore.loop()`. Afterwards, the actual updating and drawing of the framework elements used in the game takes place. By calling `jumruCore.gamestage.draw()` we force the `gamestage` to draw the `activeScene` and all its children. The basic draw method is implemented in the `jumru.Entity`:

```
jumru.Entity.prototype.draw = function() {
  if(this.active) {
    this.update();
    var length = this.children.length;
    for(var i = 0; i < length; i++) {
      if(this.children[i].active) {
        this.children[i].draw();
      }
    }
    this.setDirty(false);
  }
  return this;
};
```

Each element updates itself before the drawing happens. Scene and layer use the entity's draw function, whereas object entities and other drawable elements have their own implementation of the draw method. Because the draw operation itself is very costly, we mark every element as `active`. If we explicitly set an element inactive, for instance after drawing it once like a never changing background image, it is not updated and drawn again. The same applies to object entities and layers. In case a motion is appended to an entity, elements are redrawn in each cycle as long as the motion is active. After elements are drawn, they are marked as *non-dirty* until their properties change again. If the properties of a layer change, all its children are marked as dirty too. If an element is not dirty, it will not get redrawn in the next game loop cycle, which reduces the number of canvas operations.

Time-based Motion

Our engine supports a `jumru.Motion`-object that allows the movement of object entities or even layers with all its children. This motion can be used for a scrolling background or for applying a parallax effect to the game. Parallaxing enables the movement of elements lying on different levels at different speeds, which simulates a 3D effect. All created motions during the game's life cycle are automatically stored in an array in the `jumruCore`. As an example the array can be used if the game pauses and all motions need to get stopped at once.

Currently *Jumru 5s* just supports horizontal movement, in left or right direction. Besides defining the direction, a motion consists of a `speed` value controlling the velocity of the movement, a `left` and `right bound` that limit the movable area and a `target` representing the target entity. If a motion is set to an entity, it is automatically called in each game loop step, otherwise it has to get paused explicitly. According to David Geary's recommendations [19] the animation should be time-based, which means the speed of the animation should be independent of the games frame rate. The speed of an animation is defined in pixels per second, whereas the FPS is defined as frames per second. According to these two values an element is consequently moved by an `offset` value that is calculated according to the motions speed and the current FPS:

```
this.newOffset = this.offset + this.speed/jumruCore.fps;
this.tempx = this.target.parent.tempX+this.target.x+
            (this.newOffset*this.direction);
```

By dividing the speed (pixels per second) by the frames per second, we achieve the number of pixels to move the element in each frame. The resulting offset values are accumulated and the target entity's position is updated in horizontal direction. Afterwards, we can check if the element has moved out of the defined motion area by checking its left and right boundaries. If the limit is exceeded, a `MotionStopEvent` is thrown and the offset is set back to 0 in order to start the animation again:

```
//out of bounds, set offset back
if(this.tempx < this.leftBound || this.tempx > this.rightBound){
  this.target.fire(MotionStopEvent);
  this.offset = 0;
}
//if new position is in between boundaries - set new offset
else {
  this.offset = this.newOffset;
}
return (this.offset*this.direction)>>0;
```

Canvas Optimization Strategies

Adding motion to the game means the canvas of the animated elements have to get cleared and redrawn again in each loop cycle. This continuous process of refreshing the canvas is an exhausting task, which may cause performance deficits. Besides the improvements for object entities, which are described in Section 5.2, we have implemented several optimization strategies to get the best out of using HTML5 canvas for painting. All approaches are summarized in Table 5.1.

Dirty Mechanism

In each game loop call the canvas has to get cleared and painted again, if a motion is applied to an entity. By calling

```
this.context.clearRect(x, y, width, height);
```

the canvas is cleared from position (x, y) by `width` and `height`. Instead of calling this on the whole canvas dimensions, we implemented a dirty-mechanism that just marks the area dirty, which has been painted. For example if we draw an element with a size of $60px$ x $60px$ at position $(10, 40)$ on a canvas with a width of $600px$ and a height of $400px$, just the painted area is marked as dirty and not the whole canvas. The dirty area for this element would be $(10, 40, 60, 60)$ instead of $(0, 0, 600, 400)$.

This **dirty mechanism** is one of some major performance improvements we have implemented. Not only `clearRect()` is an time-consuming function, but also `drawImage()`, where we can significantly improve code to reduce performance lacks. Further canvas optimization strategies are described below [39].

In-canvas Mechanism

As mentioned in Section 5.2, just active elements are updated and drawn. If an element is explicitly set inactive after drawing it once, it will not get updated or drawn again. Furthermore, we also implemented a function for object entities called `isInCanvas()` that checks if an object entity is positioned inside the canvas, to avoid costly rendering operations, if the element is not even visible in the canvas.

Hardware Acceleration

Canvas rendering methods hardly rely on the potential of the graphical processing unit, the GPU. To be able to achieve best results, we can use it in combination with the central processing unit, the CPU. This is called hardware acceleration and uses the GPU over the CPU for complex rendering operations, whenever it is possible [35].

Each browser provides mechanisms to find out, if hardware acceleration is currently available and activated, see Section 7.1.

Strategy	**Description**	**see jumru-*`Class`* and *`function`***
active elements, 5.2	just draw elements that are marked as active	*`Entity, draw`*
offscreen canvas, 5.2	pre-render images in an offscreen canvas and use this offscreen canvas instead of the image for consecutive drawing operations	*`ObjectEntity, draw`*
rounded pixel, 5.2	avoid aliasing by drawing on rounded pixels	*`ObjectEntity, draw`*
dirty mechanism, 5.7	just clear and redraw dirty areas in the canvas	*`ObjectEntity, draw`*
In-canvas mechanism, 5.7	avoid drawing elements that are positioned outside the canvas	*`ObjectEntity, isInCanvas`*
hardware acceleration, 5.7	use hardware acceleration for canvas drawing	*- browser options*

Table 5.1: Overview of performance optimization strategies

5.8 (Keyframe) Animations

As mentioned in Section 4.3 CSS3 implements animation and transition effects that can be appended on *document.elements*. To be able to set and modify CSS3 animations via JavaScript, we implemented some standard animation types in our engine.

All of the listed animation types below share a common `jumru.Animation` constructor property. The common animation object defines properties like the animation-type, whether we set the `transform`-property or if we just change style-attributes, for example for fading effects. It further defines the animation's `value` that determines the value to be used for scaling or rotating an element and it stores an `anchorPoint`, which just represents the transform-origin of the animation. An animation can be appended on an entity by calling the `play()`-function that takes the target entity as a parameter and changes its style-attribute:

```
jumru.Animation.prototype.play = function(entity) {
  entity.domEl.style[jumru.browserSpec.JS+"TransformOrigin"]=this.anchorPoint;
  entity.domEl.style[jumru.browserSpec.JS+"Transform"]=this.value;
};
```

Note: The `jumru.browserSpec.JS` object is used to handle cross-browser compatibility and is further explained in Section 5.9.

Currently *Jumru 5s* supports following animation types:

- *Scale*: scales the *document.element* of an entity in x and y direction, appended via transform property.
- *Rotation*: rotates the *document.element* of an entity by some degree, appended via transform property.
- *Translation*: translates the *document.element* of an entity by some pixels in x and y direction, appended via transform property.
- *Fade*: changes the opacity of the *document.element* of an entity. In this case, adapting the transform property is not necessary.

For example we can create three scaling and rotating animations:

```
var scale1 = new jumru.Scale().setPercent(0).setScale(1.0);
var scale2 = new jumru.Scale().setPercent(50).setScale(0.9);
var scale3 = new jumru.Scale().setPercent(100).setScale(1.0);

var rotate1 = new jumru.Rotation().setPercent(0).setDegree(-3);
var rotate2 = new jumru.Rotation().setPercent(50).setDegree(3);
var rotate3 = new jumru.Rotation().setPercent(100).setDegree(-3);
```

Without specifying the percent property, each animation could be played independently on an entity by calling the `jumru.Animation.prototype.play()`-function. Instead of applying each single animation, we combine them in a `jumru.Keyframe-Animation`. Therefore we call the `setPercent()`-function, which sets a percentage that defines the point in time when the specified animation and its value should occur in the keyframe animation.

A keyframe animation has a specified `name`, with which the animation is stored in the style sheet. This property is passed to the constructor. Besides the properties that are mentioned in Section 4.3, a keyframe animation also contains a `dirty` boolean, which is set true, if a value or a sub-animation of the keyframe animation has changed. If the animation is marked dirty, it is deleted out of the style sheet, updated and written into it again. Just if the animation is stored in the style sheet, it can be appended to a *document.element*.

By adding multiple animations to a keyframe animation, combinations of different animation types are possible. According to the example above, we could combine each scale animation with a rotation animation. Therefore the animations are stored in a key-value map, which uses the percentage of the keyframe as the key and a `jumru.AnimationStore` as the value. An animation-store is a list that consists of different animations:

```
var itemAnim = new jumru.KeyframeAnimation("itemAnimPulse");
itemAnim.addAnimation(scale1).addAnimation(scale2).addAnimation(scale3);
itemAnim.addAnimation(rotate1).addAnimation(rotate2).addAnimation(rotate3);
itemAnim.setDuration(2).setIterationCount('infinite').setEasing('linear');
itemAnim.play(itemEntity);
```

For each animation to add, we check if an animation-store for this percentage key already exists and add the animation under its percentage key. An animation-store differentiates between animations affecting the transform property (scale, rotate, translate) and the fading animation, which does not affect this property. Building on the class properties for the current keyframe animation and the animation-stores in it, we construct a proper CSS3 keyframe animation string. This string is written to the document's style sheet, by reading the `document.styleSheets` and the `cssRules`-array in it. Afterwards, the keyframe animation can be played analogous to the simple animation, just here we set further properties like the fill-mode or the duration, etc.

Playing an animation means that the style-attribute of the *document.element* is overwritten. To be able to replay an animation after a specific amount of time, we explicitly have to remove the animation again. The animation is removed by calling the `removeAnimationOfEntity()`-function of the keyframe- or the animation-object. Existing values for the animation are cleared and the animation can get applied again.

5.9 Cross Browser Compatibility

While implementing CSS3 animations and effects we encountered troubles dealing with different browser vendors that use different prefixes. To avoid this problem we define a `jumru.browserSpec` object that stores individual prefixes, which are used to handle specific CSS or JavaScript code. Paying attention to different vendors in the code is important, i.e., we differentiate between Mozilla or Webkit browsers for keyframe animations.

Dealing with cross browser compatibility is also important if we listen to an animation's start or stop event. While Firefox is throwing an *animationstart* and *animationend* event, Webkit browser dispatch a *webkitAnimationStart* and *webkitAnimationEnd* event. Besides these issues, we experienced difficulties when setting elements invisible, which we mentioned in Section 5.2. In our game we are playing some CSS3 animations while the user can pause the game. If the game pauses, we set the current scene invisible by setting its `style.display` to `none`. After getting back to the game play by setting the scene visible again, the animations are forced to get replayed in Webkit browsers, though not in Firefox. To cope with this issue, we explicitly set a negative z-index to the element to place it in the background, if it is set invisible. Therefore the `setVisible()`-function of the entity is automatically set on the pageload, depending on the browser vendor. For Firefox it is sufficient to manipulate the `style.display` property, whereas Webkit browsers set a negative z-index.

5.10 Logging

Logger

An unnoticed but important component for serious gaming, as mentioned above in Section 3.1, is logging. *Jumru 5s* provides a basic `jumru.Logger` that stores multiple `jumru.Log`s in a `logs`-array. A `jumru.Log` generally describes a row that is stacked in the `logs`-array. The array is sent to a `logger.php` which is lying on the server. The first index of the array is reserved for the filename. The PHP file reads out the chosen filename and walks through the rest of the array, evaluates the entries and creates a text string in CSV file format. The file is stored on the server and the PHP reports if everything went fine or if an error occurred. When a `jumru.Log` is flushed to the logger's `logs`-array, a timestamp is created and stored at the beginning of the log-row. A visualization of the logger's structure is shown in Figure 5.4. The logs can be seen stored as stacked row entries, where each log row starting from index 1 contains a timestamp. The first index is reserved for the filename and does not include a timestamp.

Questionnaire

Besides logging that happens unnoticed in the background, we also provide a mechanism to collect data directly from the user. A `jumru.Questionnaire` is a div-container and holds a HTML5 form element. Inside the form element a table is placed, which contains the data to be shown to the user. The table is styled with a CSS file, though the developer using the engine can override its contents. We use PHP to store the form as a TXT file on the server. As a generic approach and to enable non-developers the possibility to create the form's content easily, we use XML as the basic language. The XML must be consistent with the following structure:

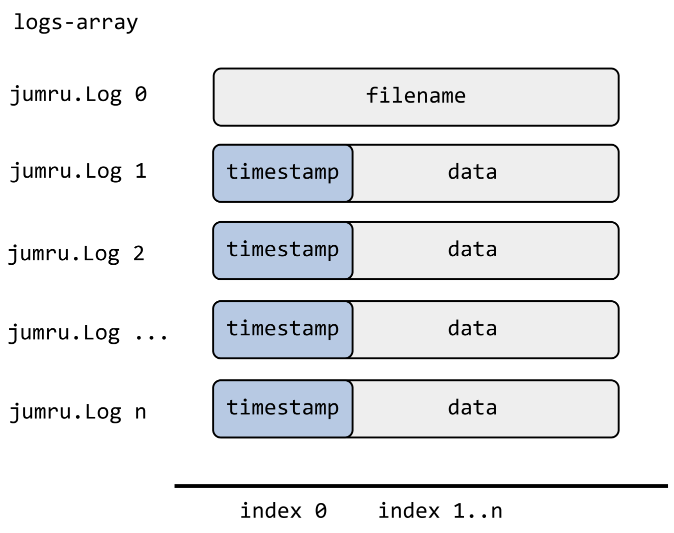


Figure 5.4: Basic structure of the `jumru.Logger`.

```
<questionnaires>, root node
- <quest> (*), with an id and a title
--- <entry> (*)
------ <label> (1), text
------ <answer> (1)
--------- <input> (*), with an id, type, name. supported input types:
                       text, number, range, radio, checkbox
--- <button> (1), for submitting
```

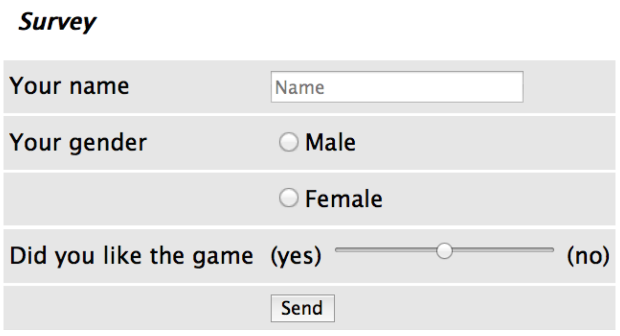
The root node `questionnaires` can contain multiple forms, named `quest`. Each questionnaire must contain an unique `id` and a `title`. A questionnaire consists of various `entries` that have a `label` and an `answer` as children. A label is a text element, which is placed on the left in the form. The corresponding answer is placed right next to it and can hold multiple `input` elements that are ordered vertically. Each input element must contain a name attribute, which is necessary for storing the form's content. At the moment the engine supports these input types:

- *text*: for text input
- *number*: i.e., for providing age input
- *range*: i.e., to state how the user liked the game. This input type needs more attributes like `from` and `to`, which specify the range
- *radio*: i.e., for gender selection
- *checkbox*: for providing multiple solutions

In addition to these input types, a questionnaire must always contain a `button`, which is used for submitting the form to the server. The `jumru.QuestionnaireFactory` takes a XML file as a parameter, parses it and creates appropriate HTML5 table entries. After creating the questionnaire, the engine stores it under its `id`. The `jumruCore` holds a `questScene` that is set as an `activeScene`, if the questionnaire is loaded with the `jumru.Core.prototype.showFormular()`-function. If the form is shown to the user, we store the current time as an invisible table entry. Furthermore, if the user submits the form, the end time is captured again and may be used to evaluate the duration. After the user has filled in all entries and has clicked the submit button, the PHP returns a prompt to an `iframe` within the div-container, to prevent the browser from opening a new tab or window. To be able to remove the `questScene` after submitting the form, the engine appends a close button for each questionnaire. Therefore the `showFormular()`-function optionally takes a background image and a string defining the button text as parameters. By default no background image is set and the button's content is *"close"*. In the code excerpt below, we initialize the factory, create a questionnaire with a `xmlfile` and call the appropriate function for showing the form with the id `id01`. Furthermore, we pass an image to be shown after submitting the form. Figure 5.5 shows the form, based on the XML file.

```
jumruCore.questFactory = new jumru.QuestionnaireFactory();
jumruCore.questFactory.create(xmlfile);
jumruCore.showFormular('id01', img_thankyou);
```

```
<questionnaires>
  <quest id="id01" title="Survey">
    <entry>
      <label>Your name</label>
      <answer>
        <input id="Name" type="text" name="Name">Name</input>
      </answer>
    </entry>
    <entry>
      <label>Your gender</label>
      <answer>
        <input type="radio" name="Gender">Male</input>
        <input type="radio" name="Gender">Female</input>
      </answer>
    </entry>
    <entry>
      <label>Did you like the game</label>
      <answer>
        <input type="range" name="Value" from="0" to="10"
               fromtext="(yes)" totext="(no)" step="1">5</input>
      </answer>
    </entry>
    <button>Send</button>
  </quest>
</questionnaires>
```


Survey

Your name	Name		
Your gender	○ Male		
	○ Female		
Did you like the game	(yes)		(no)
	Send		

Figure 5.5: Questionnaire, based on the XML file.

6 Case Study - EmoJump

On the basis of *Jumru 5s* we developed a serious game that is used to train the emotion recognition skills of children aged between 5 and 12. We invented the game in a close collaboration with the Department of Clinical Child and Adolescent Psychology of the University of Vienna, which use the game in a training study. During the game play, the logging tool delivers information in the background. The data is stored as a CSV file on the server and is further analysed to evaluate the effectiveness of the training.

EmoJump is an endless runner game, in which the player has to collect emotional faces that are displayed as coins. While collecting coins the player has to avoid crashing in barriers on the ground. The game consists of three levels representing different stages of emotion recognition. The three levels slightly differ from each other, though they share some common game play mechanisms. In each level some kind of scenario in form of a thought bubble or a comic strip is presented to the children. In these scenarios we see different stakeholders, of which one is always a child in a red shirt. Now the children playing the game have to put themselves into the scene and have to understand the feelings of the child in the red shirt. Afterwards, the player starts running and coins in form of emotions are moving through the screen. Four basic emotions can appear in the game, namely happy, sad, angry and scared. An illustration of level 1, which is also used for further evaluations, is shown in Figure 6.1. Before the player starts running, he is standing still and looks at the presented thought bubble. In the picture we see the thought bubble showing a child in a red shirt, standing in front of a house with a broken window, because it was playing soccer. How does the child feel in this moment? By pressing space the player starts running, which means the ground is moving towards the player, who stays at the same position. Afterwards, several coins are randomly set on the ground. The correct emotion type (in this case *scared*) is always part of the random set. If the player collects an emotion, the backpack in the left top is bubbling. Additionally, if the user does not understand what the thought bubble is presenting, he can click on a joker button, which acoustically explains the situation. If the player crashes into a barrier like the coppice on the ground, his lives are decremented.

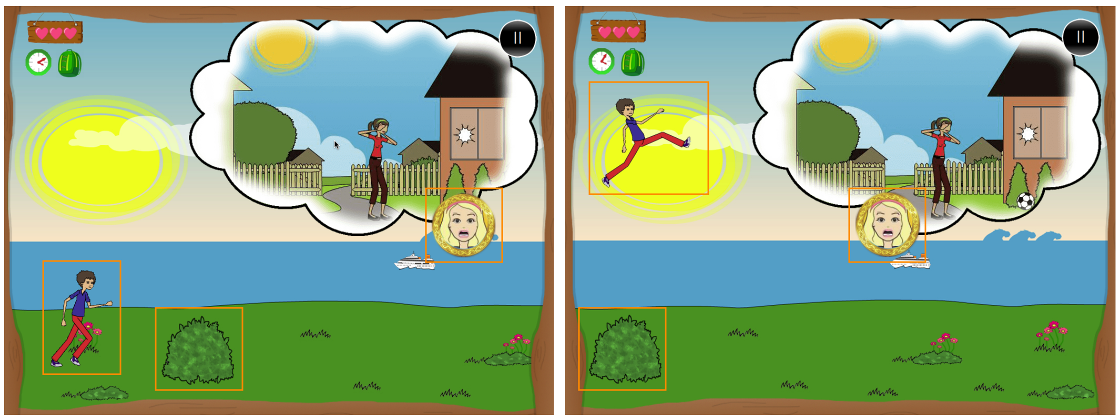


Figure 6.1: Example of *EmoJump* - Level 1.

6.1 Game setting

In this section we mainly describe how we build the game with the help of *Jumru 5s*, which elements we use and how they interact with each other. When starting implementing the game, we can instantly use the `jumruCore` to set first game objects. For initializing the game it is important to set the `jumruCore.gamestage`'s properties, like the game's width or height. *EmoJump* is generally divided into several scenes, at which just one scene is active for some time. Each scene consists of various layers and object entities. In the game we have following instances of `jumru.Scene`:

1. *Menu*: shown during menu selection.
2. *Preloading*: shown while game data is loaded in the background.
3. *Game*: this scene is active during the whole game play.

4. *Pause*: shown if the game is paused.
5. *Crash*: shown after the player crashes into a barrier.
6. *Gameover*: shown if the player has no more lives left after multiple crashes.
7. *Win*: shown at the end of the level and provides further buttons to jump back to the menu scene or the game scene, if the user decides to play the next level.

After starting the game, the user sees the menu scene. Optionally he can specify some settings by clicking on various `jumru.Buttons`. Furthermore, he can choose a character (object entities with images of boy and girl) and enter a name for it in a `jumru.Textfield`. Finally, he selects a level and a sublevel, whereupon a `jumru.Video` introduces the user to the game play. Afterwards, the menu scene switches to the preloading scene, while the `jumru.AssetManager` loads game assets in the background. When finished, the scene switches to the game scene, which is present most of the time.

Figure 6.1 further shows the game scene and its main components. At the very back the background image is painted. Multiple layers and object entities are placed above. In the left upper corner a layer contains several control elements, which are positioned next to an object entity showing the thought bubble. The player is standing in the left lower corner. To create the endless runner motion two object entities for the ground are set next to each other, each having its own motion object. A visualization of the motion of the two grounds is shown in Figure 6.2.

The motion object translates the ground through the canvas from right to left and dispatches a stop event if the position of the object entity has reached the motion's boundaries. If the motion has stopped, a new ground is randomly chosen and the motion is reset. During the movement the player is always staying at the same position. The ground images are repainted in each game loop step in order to create a seamless motion, but the sprites of the player have their own update rate. Just if the player is jumping, he gets repainted in each loop to achieve a smooth jumping curve. Motions

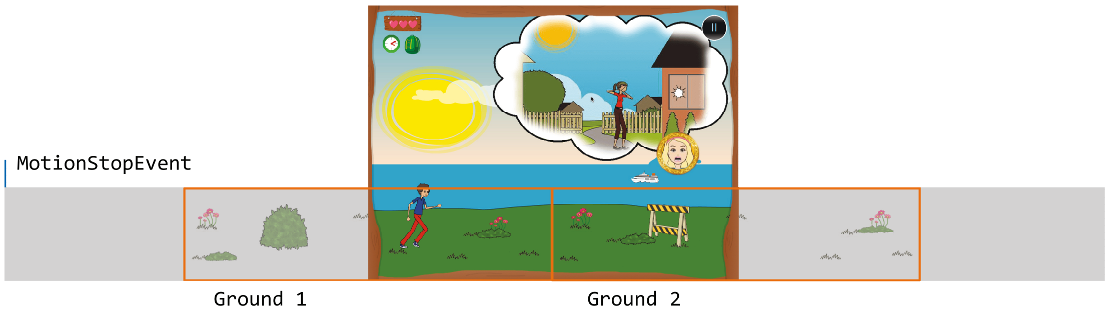


Figure 6.2: Motion of grounds.

are also used to create a parallax effect, i.e., in Figure 6.1 we also see a boat in the background, which is moving slower than the floor in the foreground.

The logger in *EmoJump* is derived from the framework logger and logs information about the game. The data includes general information about the level, like day and time, duration of the game play, level speed or the average FPS. Furthermore, we store the player's name, each thought bubble and the collected items, lost lives, etc.

In addition to the logger, we implemented some simple questionnaires to illustrate their functionalities, though they are not part of the official final *EmoJump* game.

7 Performance Evaluation

On the basis of our game *EmoJump* we evaluate the performance of our engine *Jumru 5s*. For the evaluation we tested the game on different platforms in different browsers on a MacBook Pro 9.2, 2.9 GHz Intel Core i7 with 8GB RAM and a Intel HD Graphics 4000 GPU (1GB VRAM). Therefore we played level 1, sublevel 1 for three minutes and logged the frame rate and the corresponding time value in each game loop step. Afterwards, we plotted and analysed the frame rate values with the software R[1].

7.1 Test Setting

The game, or rather the engine is evaluated on following platforms and browsers, of which some are virtualized with the VMWare Player [2]:

1. **OS X 10.9.3**
   Mozilla Firefox 31.0, Google Chrome 36.0.1985.143, Safari 7.0.4
2. **Windows 7 Professional 64bit**
   Mozilla Firefox 31.0, Google Chrome 36.0.1985.143
3. **XUbuntu 14.04 64bit as a VM in VMWare Player on Windows 7**
   Mozilla Firefox 31.0, Google Chrome 36.0.1985.143
4. **Linux Mint 17 Cinnamon 64bit as a VM in VMWare Player on Windows 7**
   Mozilla Firefox 31.0, Google Chrome 36.0.1985.143

We could not test the game in Safari on Windows 7, because Apple stopped upgrading their browser for Windows users. The latest update on their website is Sa-

[1] `http://www.r-project.org/`

[2] `https://my.vmware.com/de/web/vmware/free#desktop_end_user_computing/vmware_player/6_0`

fari 5.1.7 and was released on May 9, 2012. This version can not cope with the current HTML5 version, i.e., video or audio playback is not supported [40]. Furthermore, we are not testing Safari on Linux, because no Linux version of this browser is available [23].

To set up different operating systems, we installed Windows 7 on a MacBook Pro with the BootCamp Assistant. We virtualized two Linux distributions (XUbuntu and Mint) in VMWare Player on Windows 7, because the player is just available for Windows or Linux platforms. At the beginning we achieved low FPS values between 15 and 20 for XUbuntu and Mint in Firefox and Chrome, because Hardware Acceleration, see Section 5.7, was not enabled by default. After setting the acceleration, we could improve results for Google Chrome. We activated it manually by following the steps below:

Firefox [2,13]

1. We looked up the `about:support` site and checked the *Graphics* information. No *GPU-accelerated Windows* were set: `(0/1 Basic)`.

2. We manually adapted parameters in `about:config` and set *layers.acceleration.force-enabled* and *layers.offmainthread.composition.enabled* to *true*.

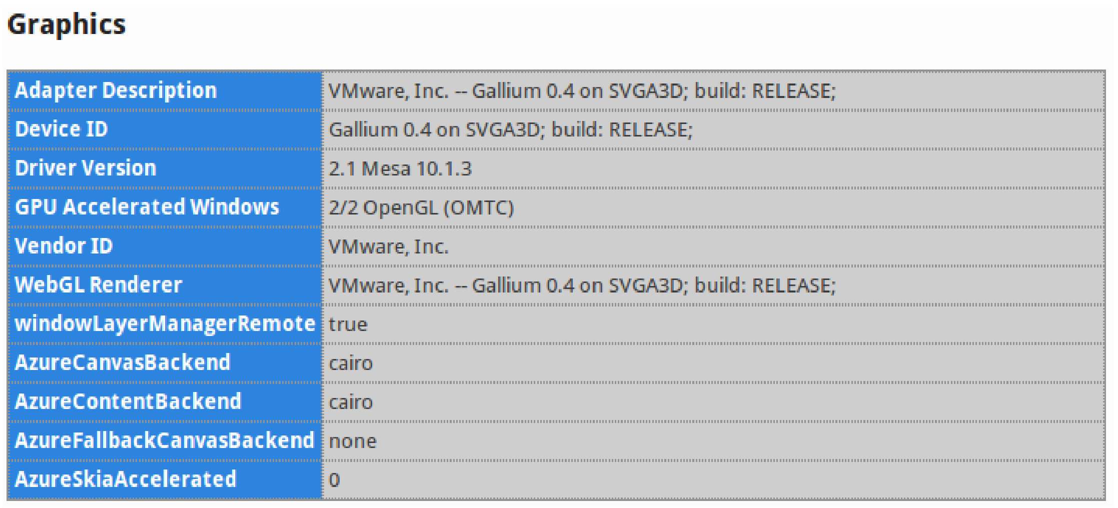
3. Afterwards, we opened Firefox via terminal with `env MOZ_USE_OMTC=1 firefox` and hardware acceleration was now enabled under `about:support`: `(2/2 OpenGL (OMTC))`, see Figure 7.1.

Chrome [4]

1. We looked up the `chrome://gpu` settings and encountered under *Graphics Feature Status* that *Canvas* support was just set to *software only*.

2. In `chrome://flags` we manually enabled *Override software rendering list* and restarted the browser.

3. Afterwards, canvas support for Chrome was now *Hardware accelerated*.


Graphics

Adapter Description	VMware, Inc. -- Gallium 0.4 on SVGA3D; build: RELEASE;
Device ID	Gallium 0.4 on SVGA3D; build: RELEASE;
Driver Version	2.1 Mesa 10.1.3
GPU Accelerated Windows	2/2 OpenGL (OMTC)
Vendor ID	VMware, Inc.
WebGL Renderer	VMware, Inc. -- Gallium 0.4 on SVGA3D; build: RELEASE;
windowLayerManagerRemote	true
AzureCanvasBackend	cairo
AzureContentBackend	cairo
AzureFallbackCanvasBackend	none
AzureSkiaAccelerated	0

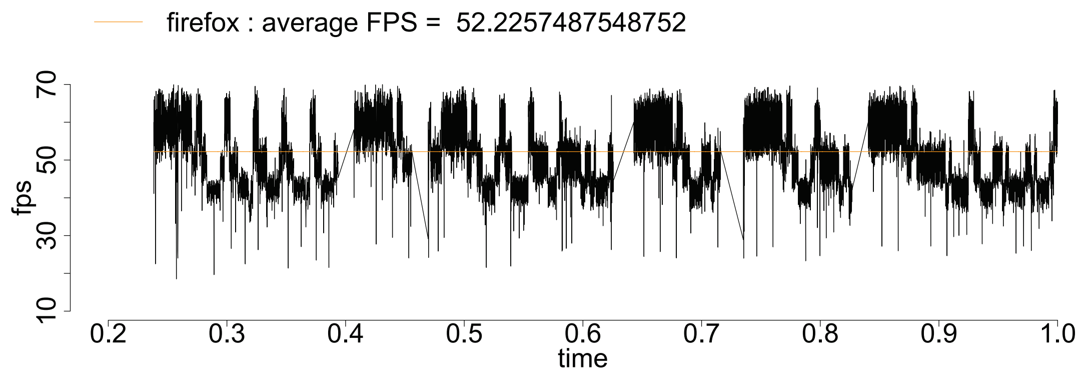
Figure 7.1: Firefox - `about:support`

7.2 Results

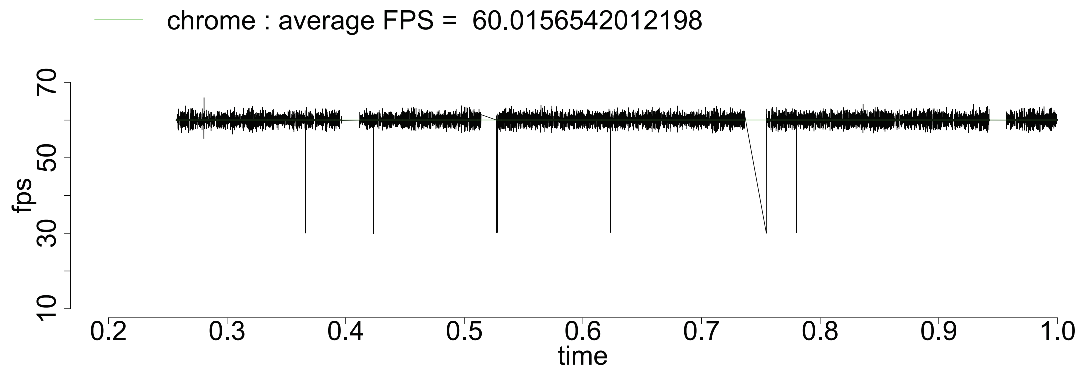
The following sections show detailed information about test results on different operating systems and in different browsers, whereas for each platform Firefox is marked orange, Chrome is marked green and Safari is marked blue. For visualizing performance outcomes we provide two kind of R plots: on the one hand we show graphs of the logged frame rate, which is captured in each game loop step. On the other hand we provide an overview of the average FPS in combination with error bars, indicating the standard deviation of the frame rate. The bar plot for all test cases and a tabular comparison of the results can be found in the summary in Section 7.3.

OS X 10.9.3

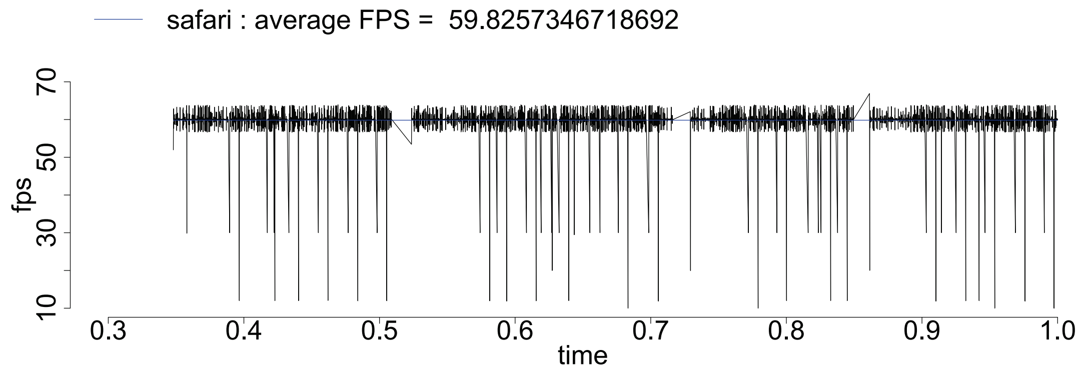
For OS X Mavericks 10.9.3 we tested the game in Firefox, Chrome and Safari and achieved an average FPS of 60 for Webkit browsers (Safari and Chrome). In Firefox FPS was a bit lower, resulting in an average FPS of 52. Figure 7.2 shows the frame rate logged during the game play. On the x axis we plot the time values with the corresponding frame rate on the y axis.



(a) OS X, Firefox



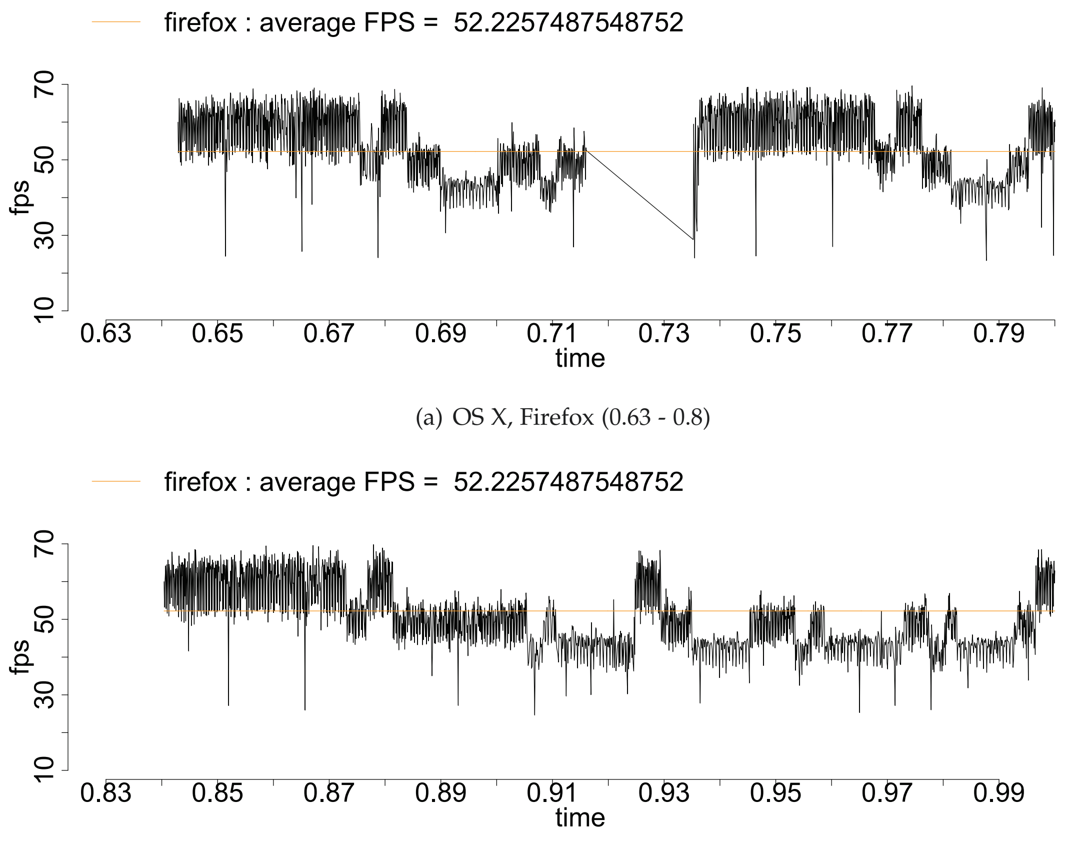
(b) OS X, Chrome



(c) OS X, Safari

Figure 7.2: FPS in OS X 10.9.3 for Firefox, Chrome and Safari.

Chrome 7.2(b) shows the smallest number of fluctuations and provides a steady frame rate. As the other Webkit browser, Safari 7.2(c) reveals similarities to the Chrome plot, but has more low FPS values. Interestingly, the FPS in Firefox 7.2(a) signalizes hops and regular patterns, which occur because of specific actions taking place in the game code, i.e., collecting items or collisions with barriers. The diagonal lines represent time breaks, i.e., if the game loop pauses due to a crash or a new presented thought bubble.



(a) OS X, Firefox (0.63 - 0.8)

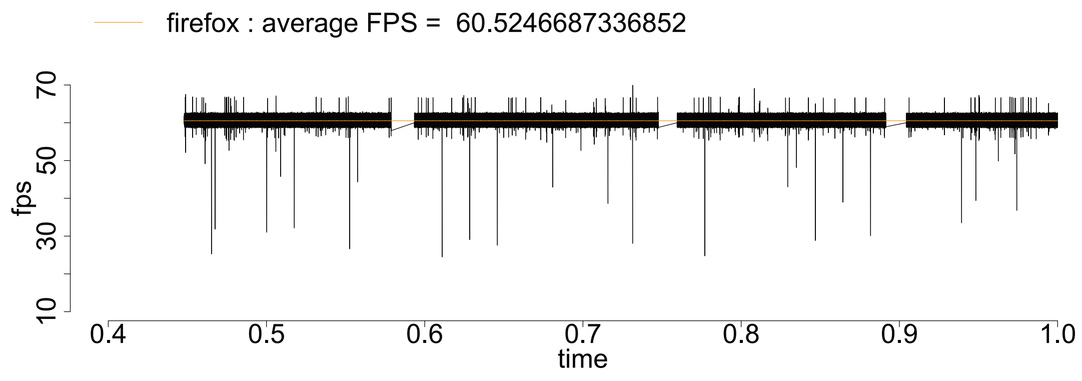
(b) OS X, Firefox (0.83 - 1.0)

Figure 7.3: Zoom-In in Firefox for OS X.

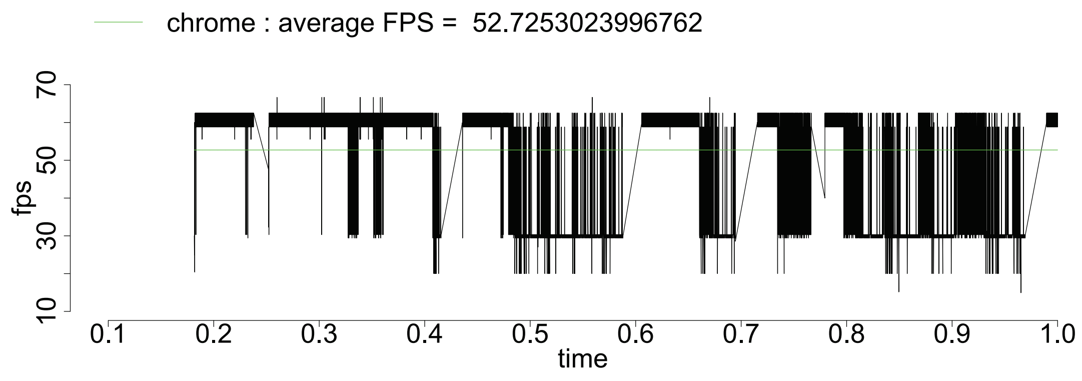
If we zoom into the Firefox plot, see Figure 7.3, we have a closer look on these patterns. After each pause, indicated by diagonal lines, the frame rate remains stable around a value of 60. We can see this robustness in the first subplot 7.3(a) from time 0.64 to 0.675 and from 0.74 to 0.77. In the second subplot 7.3(b) this is also observable from time 0.84 to 0.87. During this time range of 0.03, which represents about 7 seconds in the game play for an average FPS of 52, only the player is running until the first barrier or the first emotion enters the screen. Afterwards, we successively check for collisions with barriers or emotions until the current thought bubble ends or the player crashes and the loop pauses again. These checks probably result in the recurring steps in the graph.

Windows 7 Professional

On Windows 7 we obtained best results in Firefox achieving an average FPS of 61, see Figure 7.4(a), whereas in Chrome the result was best at a frame rate of 53, which can be seen in Figure 7.4(b).



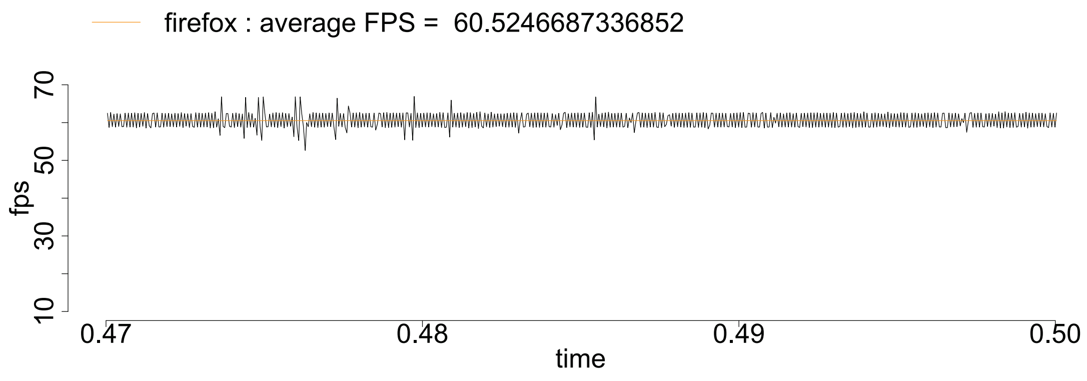
(a) Windows, Firefox



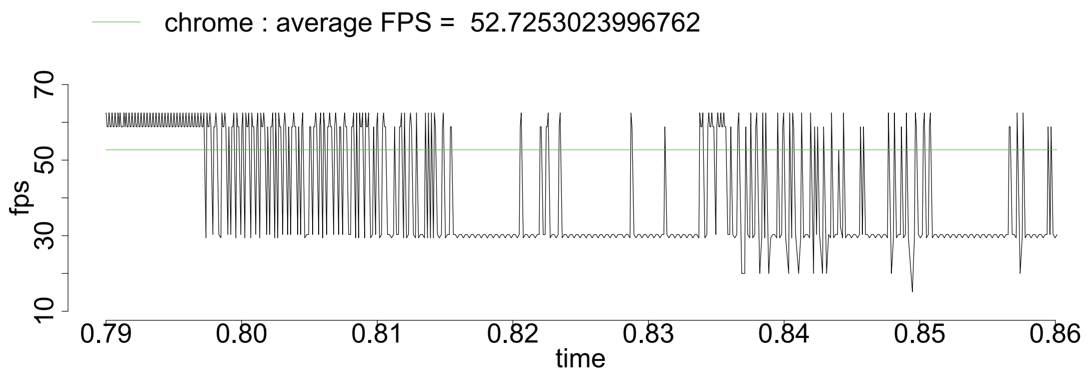
(b) Windows, Chrome

Figure 7.4: FPS in Windows 7 for Firefox and Chrome.

The plot shows some outliers for Firefox, but the frame rate keeps mostly stable among the logging process. In Chrome we experienced a higher fan speed, which is visualized in the middle of the graph and especially at the end of the plot. At halftime, the plot indicates a staircase-shaped pattern as a result of sudden changes in the frame rate. This jittering between low and high values impaired the gaming experience, which was primarily noticeable when jumping over barriers, resulting in a choppy jump curve.



(a) Windows, Firefox (0.47 - 0.5)



(b) Windows, Chrome (0.79 - 0.86)

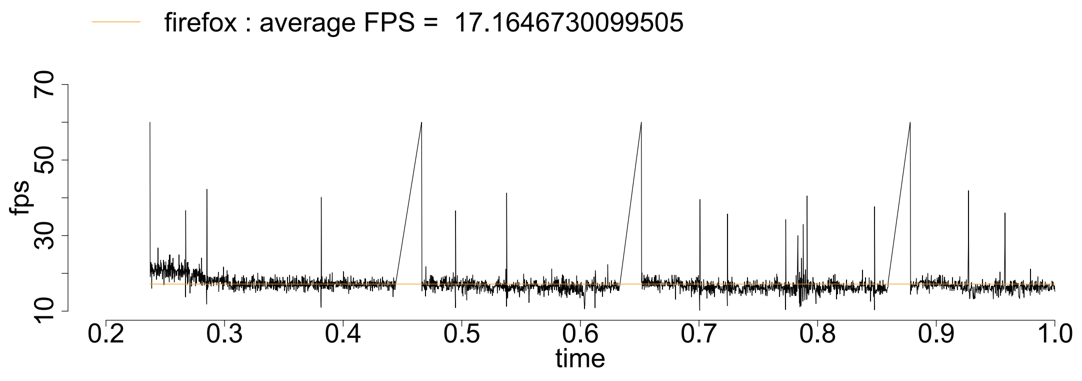
Figure 7.5: Zoom-In in Firefox and Chrome for Windows 7.

Figure 7.5 shows a zoom in Firefox and Chrome. In the first zoom for Firefox 7.5(a) we see how steady the frame rate is, with a few outliers at the beginning around time 0.475. Otherwise, the frame rate keeps stable and does not offer any noticeable patterns. Instead the plot underneath visualizes how jittery the frame rate is for Chrome 7.5(b). Like for Firefox in OS X after restarting the game after a pause, the frame rate is constant for a specific amount of time (0.79 - 0.798), until elements appear in the screen again. When the objects entered the screen, we also jumped to collect these objects or to avoid crashing into them. These calculations are visible in the dropping frame rate.

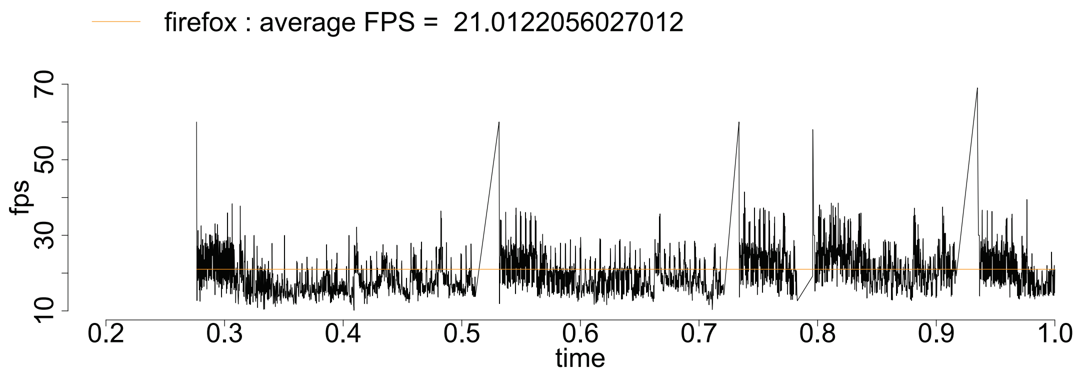
XUbuntu 14.04

The first Linux distribution, which we virtualized in Windows 7 is XUbuntu 14.04. On this platform we tested the game in Firefox and Chrome. Analogous to OS X 10.9.3 we achieved best results in a Webkit browser, or more precisely Google Chrome, resulting in an average FPS of 60, after enabling hardware acceleration, see Figure 7.6(c). The frame rate during the game play stays quite robust and does not exhibit many outliers. Furthermore, the graph for Chrome shows a slight periodic pattern.

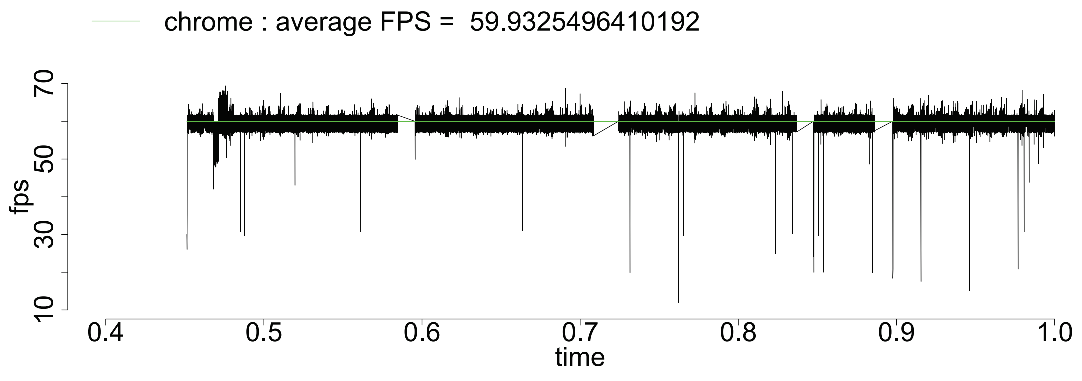
Enabling hardware acceleration on Firefox did not significantly improve performance statistics. Before activating the hardware support we achieved a FPS of 17, whereas after enabling it the frame rate increased to 21, though without hardware acceleration the frame rate seems to keep steadier among the logging process. In Figure 7.6 we plotted values for Firefox without and with hardware acceleration and the values we got for Chrome. In the first graph 7.6(a) for Firefox, which visualizes the game play without acceleration, we see smaller differences between the minimal and maximal turning points than for the second graph 7.6(b), showing the frame rate after enabling hardware acceleration. Otherwise plots for Firefox in UNIX based systems (OS X, Mint) generally tend to have a greater variance from maximal and minimal turning points, which is also shown in the barplot in Section 7.3.



(a) XUbuntu, Firefox without hardware acceleration



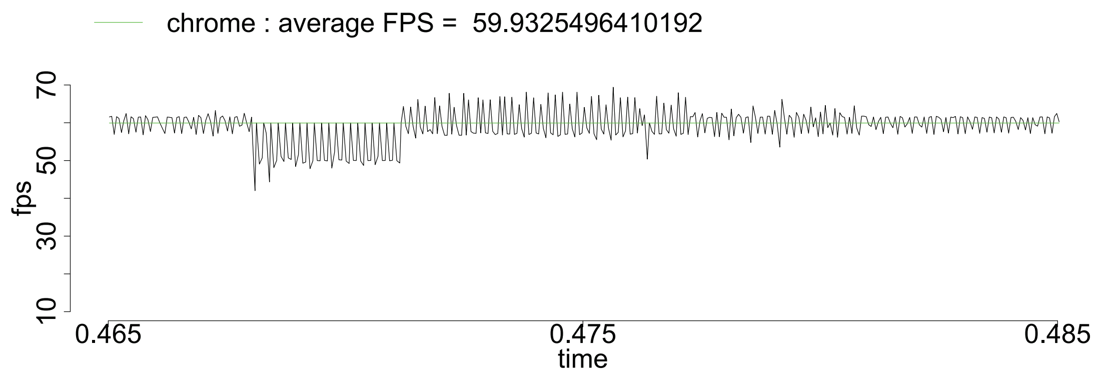
(b) XUbuntu, Firefox with hardware acceleration



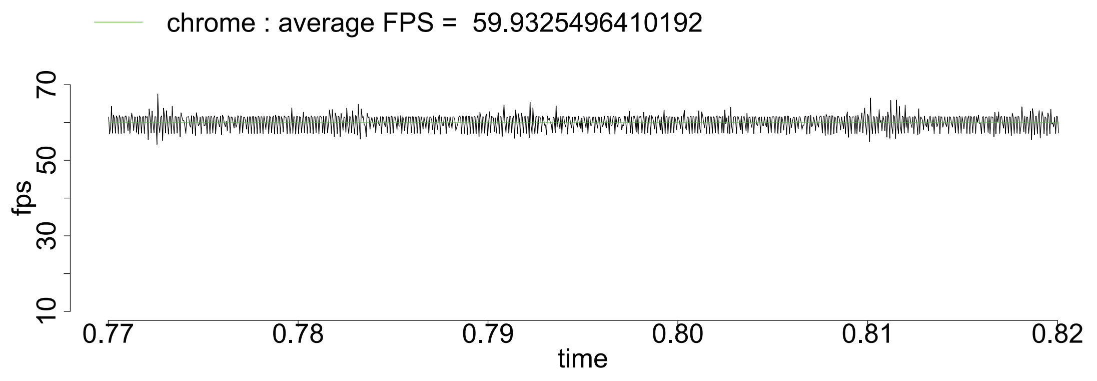
(c) XUbuntu, Chrome

Figure 7.6: FPS in XUbuntu for Firefox and Chrome.

When zooming in the plot of Chrome from time 0.465 to 0.485, we first see a negative, and shortly thereafter a positive amplitude around the time of 0.47, see the first subplot in Figure 7.7(a). At this point of time the parallax in the back moved out of screen and we jumped over a barrier. This results in extra calculations for swapping the parallax, performing the jump in each game loop step and checking if the player crashes into the barrier, which probably caused the peaks at the beginning. As mentioned earlier, the graph for Chrome illustrates some periodicities. Underneath the first subplot, see Figure 7.7(b), we zoomed in from time 0.77 to 0.82 to have a closer look on the regularity of the pattern. A timestep of 0.01 represents 3 seconds in realtime for a FPS of about 60. Every 0.01 steps on the time axis we see some outliers. This may be caused by functions checking for barrier collision or emotion collision that mostly happen every 3 seconds.



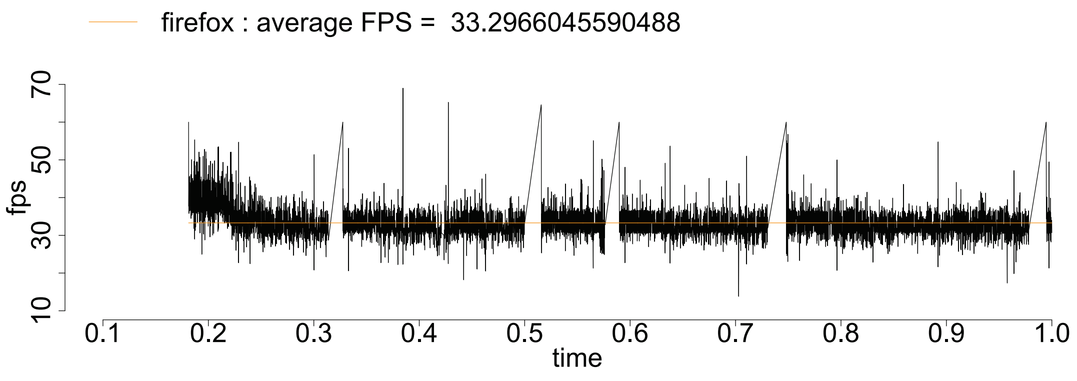
(a) XUbuntu, Chrome (0.465 - 0.485)



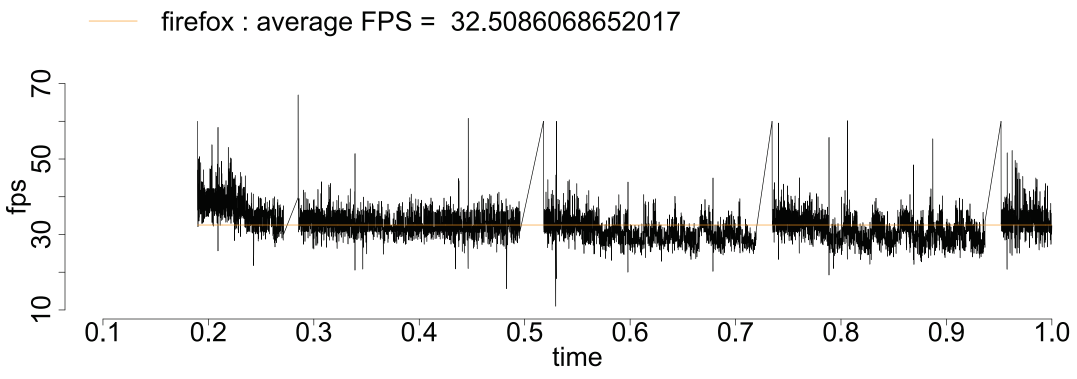
(b) XUbuntu, Chrome (0.77 - 0.82)

Figure 7.7: Zoom-In in Chrome for XUbuntu.

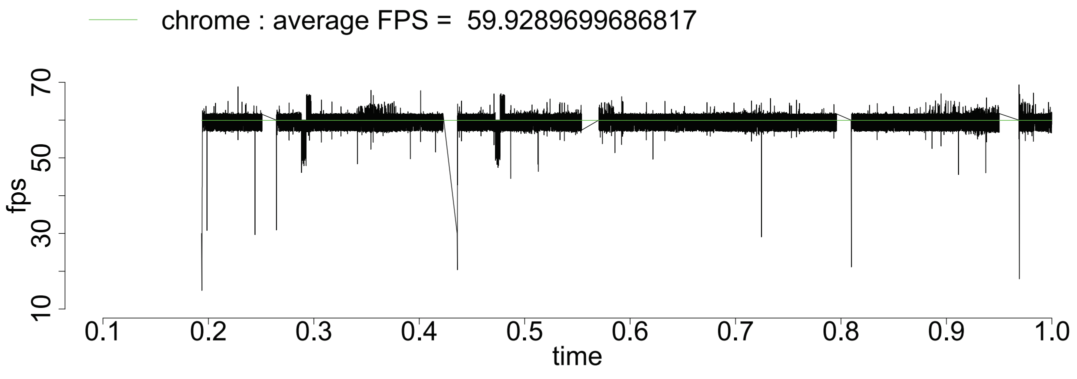
Linux Mint 17



(a) Linux Mint, Firefox without hardware acceleration



(b) Linux Mint, Firefox with hardware acceleration



(c) Linux Mint, Chrome

Figure 7.8: FPS in Linux Mint for Firefox and Chrome.

We wanted to find out if the game would perform better in other Linux distributions. Therefore we virtualized Linux Mint 17 in Windows 7, which already has a pre-installed Firefox browser. We updated it manually from Version 28.0 to Version 31.0 and tested it with and without hardware acceleration. Interestingly, playing without acceleration resulted in an average frame rate of 33, see Figure 7.8(a). However, enabling the hardware acceleration did not improve the performance of the game. Instead the FPS was almost the same with a rounded value of 33, which is illustrated in the second plot 7.8(b) below. Once more for a UNIX based system the best performance is obtained in Chrome 7.8(c), resulting in an average FPS of 60. The graph for Chrome reveals a regularity of a descent followed by an ascent, taking place at time 0.3 and 0.48. Like for XUbuntu in Chrome, at this point in time the parallax moved out of the canvas and a new barrier entered the screen.

7.3 Summary

Table 7.1 shows the platforms and the tested browsers with the average FPS we achieved for each test case. Additionally, the bar plot in Figure 7.9 provides an overview of all test cases and the average frame rate with their standard deviations. Once more, for OS X 10.9.3 Webkit browsers (Safari, Chrome) obtain best results with an average FPS of 60, but Safari shows a little more fluctuations among the game play resulting in a greater standard deviation. XUbuntu and Linux Mint are significantly better in Chrome, whereas Windows achieves better results in Firefox.

	Firefox 31.0	**Chrome 36.0.1985.143**	**Safari 7.0.4**
OS X 10.9.3	52	60	60
Windows 7	61	53	/
XUbuntu 14.04 (VM on Windows 7)	21 (with h.acc.)	60	/
Linux Mint 17 (VM on Windows 7)	33 (without h.acc.)	60	/

Table 7.1: Overview of performance tests with the resulting average FPS (rounded).

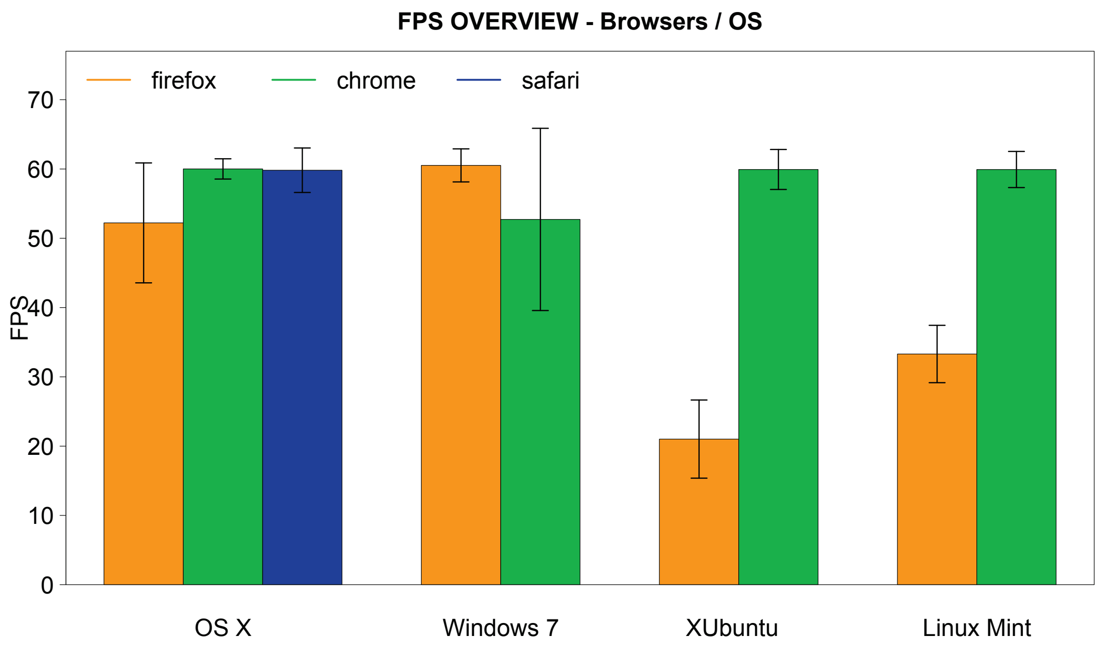
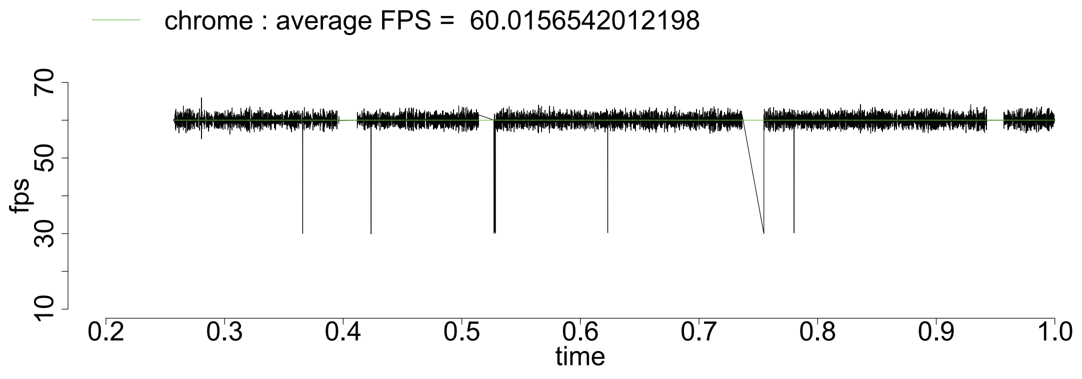


Figure 7.9: Bar plot of all test cases with their standard deviations.

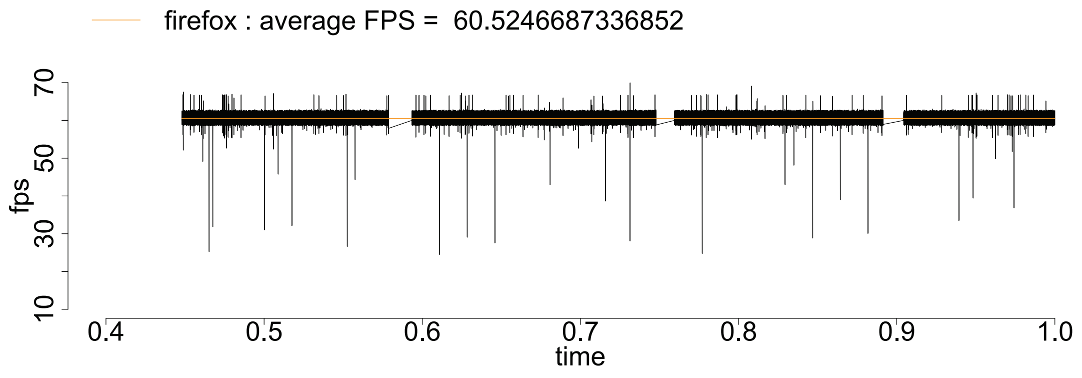
```
      browser   os        mean     sd      min      max      length
 [1,] "firefox" "osx"     "52.223" "8.653" "18.519" "69.977" "8260"
 [2,] "chrome"  "osx"     "60.012" "1.462" "29.893" "65.967" "9703"
 [3,] "safari"  "osx"     "59.822" "3.211" "10.003" "66.900" "9544"
 [4,] "firefox" "win7"    "60.521" "2.384" "24.483" "69.952" "10101"
 [5,] "chrome"  "win7"    "52.722" "13.15" "14.925" "66.666" "7755"
 [6,] "firefox" "xubuntu" "21.015" "5.639" "10.136" "69.021" "3104"
 [7,] "chrome"  "xubuntu" "59.929" "2.885" "11.957" "69.405" "9818"
 [8,] "firefox" "mint"    "33.296" "4.141" "13.808" "68.994" "5732"
 [9,] "chrome"  "mint"    "59.925" "2.607" "14.899" "69.367" "9765"
```

We created the bar plot on the basis of the statistics listed in the code excerpt above, though we present rounded values here. The highest standard deviation of 13.15 was received for Windows 7 and Chrome, due to the cascaded plot, see Figure 7.4(b). Otherwise Chrome generally exhibits the lowest standard deviations with values between 1.462 for OS X and 2.885 for XUbuntu. For UNIX based systems (OS X, XUbuntu, Mint) Firefox generally has a higher standard deviation than Webkit browsers. The number of values (called *length* in the code excerpt above), which are logged during 3 minutes of game play, is determined by the refresh rate of the browser. The faster the update rate and the higher the FPS, the more values are extracted. On the opposite, the lower the FPS, the less values are logged.

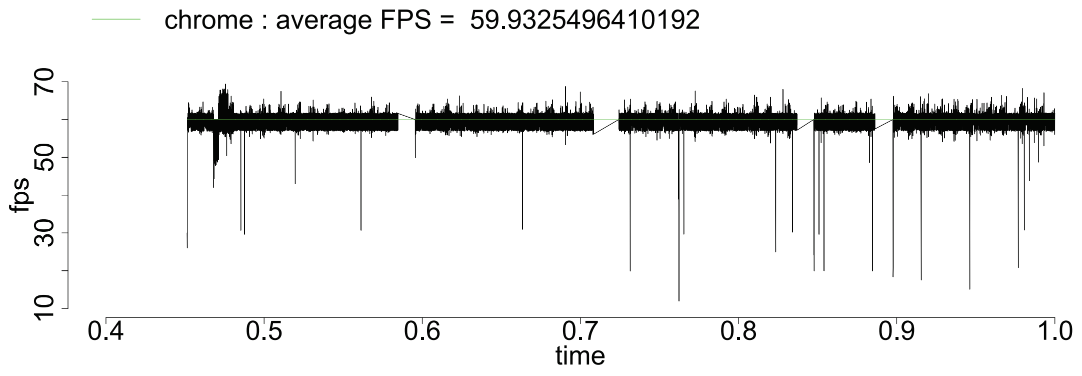
Figure 7.10 summarizes the performance evaluation and shows a comparison of all winning browsers of each tested platform. The final test cases show similarities, like a steady frame rate and a small standard deviation.



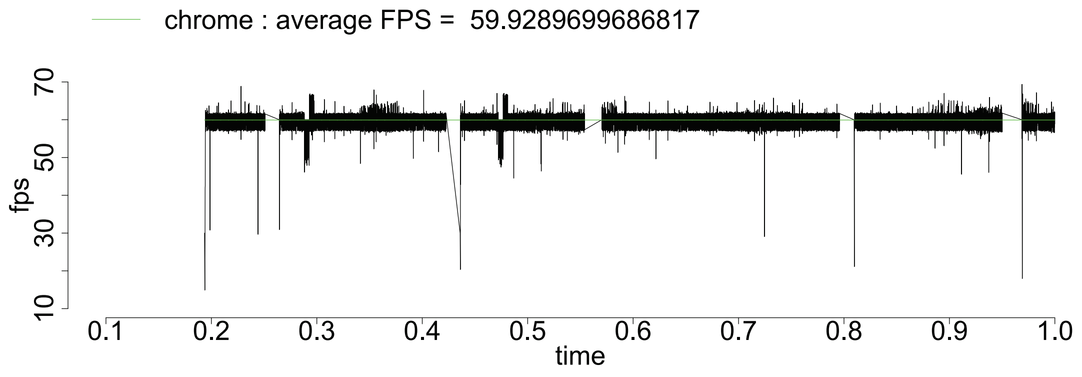
(a) OS X, Chrome



(b) Windows 7, Firefox



(c) XUbuntu, Chrome



(d) Linux Mint, Chrome

Figure 7.10: Frame rate of all winning browsers for each platform.

8 Conclusion and Future Work

This master thesis describes a web game engine for serious games, which supports jump and run and endless runner games. On the basis of our engine we implemented a serious game, which served as a case study to evaluate the engine's potentials. We started by giving a general introduction in the background (2) of serious games that are especially designed for teaching emotion recognition skills. The purpose of our game is to train the emotion recognition skills of children, why we have taken a close look at existing web game engines (3). After getting familiar with JavaScript, HTML5 and CSS3 technologies (4) and evaluating several existing engines, we decided to implement an own object oriented engine, which is especially designed for serious games and incorporates modules for data collection. Therefore we provided a general definition of the term game engine. Based on the knowledge gained during research, we developed our engine *Jumru 5s*, whose presentation is the aim of this work and builds the main part (5). Building on *Jumru 5s* we introduced our serious game *EmoJump* (6), which is used for the performance evaluation of our engine (7). The performance tests incorporated several platforms and browsers, which were all tested on the same device. Results show that the current status of the engine works well for most platforms, at least for one browser, but performance improvements are still possible, especially for rendering. To be able to support multiple devices, related game engines not only use canvas for rendering, but implement further strategies like DOM-sprites.

Back in 2008, after the first draft version of HTML5 was released by the W3C, mobile support of HTML5 was very low [32]. Since then enhancements in mobile technologies for example better hardware enabled new possibilities and wider support for HTML5, especially for audio and video, CSS3 and canvas rendering. In his book David Geary [18] notes that during the time of writing his book about HTML5 canvas and game development *"Canvas performance on mobile devices was dismal"*. Geary, who published his book in 2012 further mentions that the performance on mobile devices significantly improved when iOS5 was released, which supported hardware acceleration. He states that *"hardware acceleration dramatically improved performance, making it possible to implement smooth animations and video games"*. The software iOS5 has been released in 2011 [22]. Supporting multiple devices is still a complex issue, because different devices provide different features like processing power, storage capacity, hardware acceleration, screen resolution or input types. Interestingly, Paul Rettig further outlines that the hardware of the device is important, but *"the browser that runs on your target device is the most important arbiter of the features available"* [32].

Apart from rendering optimizations and mobile support, further improvements concerning our engine can be made by extending the logging package. Besides the logger and the questionnaire, the engine could provide mechanisms to enable recording software or other psychophysiological tools like emotives. Future work could also incorporate visualizations of bounding boxes to provide some debugging instruments. In addition to these ideas, the field of animation and motion is still capable of development. Therefore further implementations of CSS3 animations or vertical motions are possible.

As we can see, game engine development shows much space for enhancements and new ideas. Implementing these features is not an easy task, since games should work on various devices, each with different hardware specifications, operating systems and browsers. Furthermore, the engine's components have to work hand in hand in order to simplify game coding, rather than complicating game development. Based on these findings, I would like to conclude with a quote from Aristotle, who summarizes what I've learned about game engine design:

"The whole is greater than the sum of its parts".
Aristotle.

Bibliography

[1] Bretagne Abirached, Yan Zhang, and Ji Hyun Park. Understanding user needs for serious games for teaching children with autism spectrum disorders emotions. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2012*, pages 1054–1063, Denver, Colorado, USA, June 2012. AACE.

[2] _abraxas. [SOLVED] Firefox 28 stopped using hardware acceleration. `https://bbs.archlinux.org/viewtopic.php?pid=1394765`, accessed: 2014-08-18, 03 2014.

[3] Eike Falk Anderson, Steffen Engel, Peter Comninos, and Leigh McLoughlin. The case for research in game engine architecture. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, Future Play '08, pages 228–231, New York, NY, USA, 2008. ACM.

[4] Andrew (Alin Andrei). Enable Hardware Acceleration In Chrome/Chromium Browser [Quick Tip]. `http://www.webupd8.org/2014/01/enable-hardware-acceleration-in-chrome.html`, accessed: 2014-08-18, 01 2014.

[5] Abhijeet Bhattacharya. Memory leak patterns in JavaScript. `http://www.ibm.com/developerworks/web/library/wa-memleak/`, accessed: 2014-04-06, 04 2007.

[6] Bert Bos. W3C CSS Specifications. `http://www.w3.org/Style/CSS/current-work`, accessed: 2014-04-02, 03 2014.

[7] Rik Cabanier, Jatinder Mann, Jay Munro, Tom Wiltzius, and Ian Hickson. HTML Canvas 2D Context. `http://www.w3.org/html/wg/drafts/2dcontext/html5_canvas_CR/`, accessed: 2014-04-02, 04 2014.

[8] David Catuhe. Unleash the power of HTML 5 Canvas for gaming. http://blogs.msdn.com/b/eternalcoding/archive/2012/03/22/unleash-the-power-of-html-5-canvas-for-gaming-part-1.aspx?Redirected=true, accessed: 2014-03-16, 03 2012.

[9] Centers for Disease Control CDC and Prevention. Autism Spectrum Disorders (ASDs) - Data & Statistics. http://www.cdc.gov/ncbddd/autism/data.html, accessed: 2014-03-09, 12 2013.

[10] Centers for Disease Control CDC and Prevention. Autism Spectrum Disorders (ASDs) - Facts About ASDs. www.cdc.gov/ncbddd/autism/facts.html, accessed: 2014-03-10, 12 2013.

[11] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[12] Dean Jackson and David Hyatt and Chris Marrin and Sylvain Galineau and L. David Baron. Css animations. http://www.w3.org/TR/css3-animations/, accessed: 2014-04-02, 02 2013.

[13] d.free. Force Enable Hardware Acceleration in Firefox. http://askubuntu.com/questions/491750/force-enable-hardware-acceleration-in-firefox, accessed: 2014-08-18, 07 2014.

[14] Dustin Diaz and Ross Harmes. *Pro JavaScript Design Patterns*. The expert's voice in web development. Apress, 2008.

[15] Samanta Finkelstein, Andrea Nickel, Lane Harrison, Evan A. Suma, and Tiffany Barnes. cMotion: A New Game Design to Teach Emotion Recognition and Programming Logic to Children using Virtual Humans. In *IEEE Virtual Reality*, pages 249–250, 2009.

[16] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 6th edition, 03 2011.

[17] Simon Fraser, Dean Jackson, Edward O'Connor, and Dirk Schulze. CSS Transforms Module Level 1. `http://www.w3.org/TR/css-transforms-1/`, accessed: 2014-04-08, 11 2013.

[18] David Geary. *Core HTML5 Canvas. Graphics, Animation and Game Development*. Prentice Hall, Crawfordsville, Indiana, USA, 2012.

[19] David Geary. HTML5 2D game development: Graphics and animation. Drawing into the canvas and putting things in motion. `http://www.ibm.com/developerworks/java/library/j-html5-game2/index.html`, accessed: 2014-03-19, 08 2012.

[20] David Geary. HTML5 2D game development: Setting the stage. Implementing the game object, pausing, freezing, thawing, and keyboard input. `http://www.ibm.com/developerworks/java/library/j-html5-game3/index.html`, accessed: 2014-03-14, 08 2012.

[21] Jason Gregory. *Game Engine Architecture*. Taylor & Francis Group, 2009.

[22] Apple Inc. Apple to Unveil Next Generation Software at Keynote Address on Monday, June 6. `http://www.apple.com/pr/library/2011/05/31Apple-to-Unveil-Next-Generation-Software-at-Keynote-Address-on-Monday-June-6.html`, accessed: 2014-09-24, 05 2011.

[23] Apple Inc. Downloads - Safari. `http://support.apple.com/downloads/#safari`, accessed: 2014-08-21, 09 2013.

[24] Entropy Interactive. Game Engine Design – Component Based Entities. `http://entropyinteractive.com/2011/02/game-engine-design-component-based-entities/`, accessed: 2014-09-22, 02 2011.

[25] Brigitte Jellinek. Web Development. Ein Lehrbuch für das Informatik oder Medien-Informatik Studium. `http://web-development.github.io/javascript/module/`, accessed: 2014-03-12, 2012.

[26] Joe Lennon. Compare JavaScript frameworks. An overview of the frameworks that greatly enhance JavaScript development. http://www.ibm.com/developerworks/library/wa-jsframeworks/, accessed: 2014-03-21, 02 2010.

[27] Brandon McInnis, Ryo Shimizu, Hidekazu Furukawa, Ryohei Fushimi, Ryo Tanaka, and Kevin Kratzer. *HTML5 Game Programming with Enchant.Js*. Apress, Berkely, CA, USA, 1st edition, 2013.

[28] velvel53 Mozilla Developer Network. Memory Management. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management, accessed: 2014-04-06, 09 2013.

[29] Lennart Nacke, Craig Lindley, and Sophie Stellmach. Log who's playing: Psychophysiological game analysis made easy through event logging. In *Fun and Games*, volume 5294 of *Lecture Notes in Computer Science*, pages 150–157. Springer, 2008.

[30] Microsoft Developer Network. Timing control for script-based animations ("requestAnimationFrame"). http://msdn.microsoft.com/en-us/library/hh920765%28v=vs.85%29.aspx, accessed: 2014-05-08.

[31] Simon Pieters. HTML5 - Differences from HTML4. http://www.w3.org/TR/html5-diff/, accessed: 2014-03-23, 05 2013.

[32] Pascal Rettig. *Professional HTML5 Mobile Game Development*. John Wiley & Sons, Inc., Indianapolis, Indiana, USA, 2012.

[33] Pascal Rettig. Quintus Documentation, lib/quintus.js. http://html5quintus.com/api/files/lib_quintus.js.html, accessed: 2014-08-11, 2012.

[34] James Robinson and Cameron McCormack. Timing control for script-based animations. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/RequestAnimationFrame/Overview.html, accessed: 2014-03-19, 10 2013.

[35] Paul Rouget. What is hardware acceleration. https://hacks.mozilla.org/2010/09/hardware-acceleration/, accessed: 2014-04-22, 2010.

[36] UCLA Russ Poldrack and Cognitive Atlas. Emotion Recognition. http://www.cognitiveatlas.org/concept/id/trm_4a3fd79d0b665, accessed: 2014-03-11, 2012.

[37] David Shariff. JavaScript Inheritance Patterns. http://davidshariff.com/blog/javascript-inheritance-patterns/, accessed: 2014-03-12, 02 2012.

[38] Google Developers Site. Getting Started with the Closure Compiler Application. https://developers.google.com/closure/compiler/docs/gettingstarted_app?hl=de, accessed: 2014-05-08, 12 2013.

[39] Boris Smus. Improving HTML5 Canvas Performance. http://www.html5rocks.com/en/tutorials/canvas/performance/, accessed: 2014-03-16, 10 2013.

[40] tested by Niels Leenheer. HTML5Test - How well does your browser support HTML5? http://html5test.com/s/0bcfe81dfcdcd93b.html, accessed: 2014-08-18.

[41] Alan Thorn. *Game Engine Design and Implementation*. Foundations of game development. Jones & Bartlett Learning, 2011.

[42] Anne van Kesteren, Aryeh Gregor, Ms2ger, Alex Russell, and Robin Berjon. W3C DOM4 - Events. http://www.w3.org/TR/dom/#events, accessed: 2014-08-04, 07 2014.

[43] venomation. Composition VS Inheritance. `http://www.codeproject.com/Articles/80045/Composition-VS-Inheritance`, accessed: 2014-09-22, 05 2010.

[44] W3C. HTML 5.1 Nightly. A vocabulary and associated APIs for HTML and XHTML. `http://www.w3.org/html/wg/drafts/html/master/Overview.html`, accessed: 2014-03-20, 03 2014.

[45] w3school. CSS3 Animations. `http://www.w3schools.com/css/css3_animations.asp`, accessed: 2014-05-23.

[46] Mick West. Evolve your Hierarchy. `http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/`, accessed: 2014-09-22, 01 2007.

[47] Wikipedia. Cumulative moving average. `http://en.wikipedia.org/wiki/Moving_average#Cumulative_moving_average`, accessed: 2014-03-19, 09 2013.

[48] James L. Williams. *Learning HTML5 Game Programming: A Hands-on Guide to Building Online Games Using Canvas, SVG, and WebGL*. Learning. Pearson Education, 2011.

Abstract

The area of serious gaming introduces a new research field and enables the therapy of people with psychological disorders in a playful way. Especially children are very amenable for computer games, which offers new and innovative treatment options. In cooperation with the Department of the Clinical Child and Adolescent Psychology (Applied Psychology) of the University of Vienna we developed a game that trains the emotion recognition skills of children. Enhancements in emotion recognition prevent from further psychological health issues and improve the children's social life. For developing the game in HTML5, JavaScript and CSS3 we dealt closely with these technologies and analysed existing web game engines. The field of game engine design discloses a large application area, because the term game engine itself is not explicitly defined. During our researches and first implementations on the basis of existing engines we encountered that none of these engines are especially designed for serious games. Particularly for psychological treatment some kind of data collection, like gaining information about the player is desirable. Information can be gathered either directly through user questions or indirectly through logging mechanisms. None of these engines supported some kind of logging tools, which led us to the development of an own web game engine that focuses on serious games. Our engine *Jumru 5s* should simplify the implementation of serious games and provide mechanisms for data collection. By means of our engine we developed a HTML5 Jump and Run game to train the emotion recognition skills of children aged between 5 and 12, which is used as a case study to evaluate the engine. We analysed the engine according to performance aspects on different platforms and in different browsers and illustrated our outcomes.

Kurzfassung

Der Bereich des Serious Gaming beschreibt einen neuen Forschungszweig, der es ermöglicht, auf spielerische Art und Weise, Menschen mit psychischen Gesundheitsproblemen zu therapieren. Speziell Kinder reagieren positiv auf Computerspiele, wodurch sich neue und innovative Behandlungsmöglichkeiten ergeben. In Kooperation mit dem Arbeitsbereich für klinische Kinder- und Jugendpsychologie der Universität Wien haben wir ein Spiel entwickelt, welches das Emotionsverständnis von Kindern trainieren soll. Ein besseres Emotionsverständnis beugt psychischen Gesundheitsproblemen vor und kann das Sozialleben der Kinder verbessern. Zur Entwicklung des Spieles in HTML5, JavaScript und CSS3 haben wir uns intensiv mit diesen Technologien beschäftigt und existierende Web Game Engines analysiert. Der Bereich der Game Engines umfasst ein großes Forschungsfeld, da der Begriff Game Engine selbst nicht eindeutig definiert ist. Im Verlauf unserer Recherchen und Implementierungen auf Basis vorhandener Engines hat sich herausgestellt, dass keine Engine explizit für Serious Games entwickelt wurde. Gerade zu Therapiezwecken wäre es wünschenswert, dass Informationen über Spieler/innen, sei es direkt durch Benutzerabfragen oder indirekt durch Logging, gesammelt werden können. Keine Engine hat solch eine Funktionalität bereitgestellt, weswegen wir zur Erstellung unseres Spieles eine eigene Web Game Engine, mit dem Fokus auf Serious Games, entwickelt haben. Unsere Engine *Jumru 5s* soll die Implementierung von Serious Games vereinfachen und wichtige Aspekte für diesen Forschungsbereich, wie Datenerhebung, bereitstellen. Mithilfe der Engine haben wir ein HTML5 Jump and Run-Spiel zum Training des Emotionsverständnisses von Kindern im Volksschulalter entwickelt, welches als Fallstudie zur Evaluierung der Engine dient. Wir haben die Engine bezüglich Performanceaspekte auf verschiedenen Betriebssystemen und Browsern analysiert und Ergebnisse festgehalten.

Acknowledgements

First, I want to thank all the people who supported and encouraged me during the completion of this master thesis. I want to take this opportunity to thank Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs for initiating this project and for enabling me to be part of it. He always helped our team immediately if questions or problems have occurred. When speaking of our team I want to thank my colleague Natascha Schweiger at this point, for the great teamwork and collaboration on this project. Furthermore, my gratitude goes out to all my friends, who were helping me with words and deeds whenever I needed them. Lastly, I gratefully acknowledge my parents for supporting me and backing me up throughout my studies.

Katharina Meusburger, BSc.

Curriculum Vitae

Geburtsdatum: 28.07.1989, Bregenz, Österreich
Kontakt: katharina.meusburger@gmx.at

Education

10/2011 - heute	**Universität Wien**: Masterstudium Medieninformatik
04/2011 - 10/2011	**Universität Wien**: Bachelorstudium Theater- Film- und Medienwissenschaften
10/2007 - 03/2011	**Technische Universität Wien**: Bachelorstudium Medieninformatik Abschluss: Bachelor of Science, am 24.03.2011 Bachelorarbeit: "Überblick über freie DVD-Authoring Software"
09/1999 - 06/2007	**Bundesgymnasium Bregenz Gallusstraße**

Publications

2014	N. Schweiger, K. Meusburger, H. Hlavacs and M. Sprung, *Jumru 5s - A Game Engine for Serious Games*, The Fifth International Conference on Serious Games Development & Applications (SGDA 2014) 9-10 Oct 2014, Berlin, Germany.

Experience

08/2012 - 09/2012	VTG - Vorarlberger Informatik- und Telekommunikationsdienstleistungsgesellschaft mbh: Webdesign, Projektrecherche
08/2011 - 09/2011 **08/2010 - 09/2010** **08/2009 - 09/2009**	Dr. Thomas Meusburger: Administration
07/2008	Firma Julius Blum GmbH: Ferialarbeit