Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Marko Novak

Izdelava HTML-igre z mobilnim kontrolerjem

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Rok Rupnik

Ljubljana, 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Razvoj iger za tablične računalnike s kombinirano uporabo različnih tehnologij je velik izziv, ki ga še povečuje smiselna uporaba senzorjev v tabličnih računalnikih.

Zasnujte in razvijte računalniško igro. Pri tem uporabite različne tehnologije, kot so: senzorji v mobilni napravi, HTML5, CSS3, Javascript, Node.js, Socket.io in Express Framework. V okviru razvoja najprej razvijte ogrodje, na katerem se bo igra odvijala v arhitekturi odjemalec/strežnik.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Marko Novak, z vpisno številko **63010100**, sem avtor diplomskega dela z naslovom:

Izdelava HTML-igre z mobilnim kontrolerjem

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Roka Rupnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) in ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 25. junija 2014 Podpis avtorja:

Zahvaljujem se prof. dr. Roku Rupniku za mentorstvo in za vse nasvete ter pomoč pri izdelavi diplomskega dela. Za vzpodbudo in podporo pri študiju se zahvaljujem tudi svojim staršem, ki sta me podpirala v času študija. Posebej se zahvaljujem žani Edini Novak za motivacijo in pomoč pri organizaciji diplomskega dela.

Kazalo

Kazalo slik

Kazalo kode

Seznam uporabljenih kratic

kratica	angleško	slovensko
HMTL	hypertext markup language	jezik za označevanje nadbesedila
CSS	cascading style sheets	predloge, ki določajo izgled spletnih strani
JSON	javascript object notation	oblika za izmenjavo podatkov
GUI	graphical user interface	grafični uporabniški vmesnik
I/O	input/output	vhod/izhod
HTTP	hypertext transfer protocol	protokol za prenos hiperteksta
NPM	node package manager	upravljalec paketov za Node
API	application programming interface	vmesnik za programiranje aplikacij
Object DB	fast object data base for Java	hitra objektna baza za Javo
MathML	mathematical markup language	matematični označevalni jezik
SVG	scalable vector graphics	razširljiva vektorska grafika
W3C	world wide web consortium	konzorcij za svetovni splet
DOM	document object model	objektni model dokumenta
QR	quick response	hitri odziv

Povzetek

V diplomskem delu obravnavamo izdelavo računalniške igre s pomočjo spletnih tehnologij HTML5, CSS3 in Javascript. Spoznamo, kaj so omenjene tehnologije in kako jih lahko uporabimo za izdelavo iger, ki tečejo v spletnih brskalnikih. Prav tako spoznamo spletne tehnologije Node.js, Socket.io in Express Framework, s katerimi postavimo ogrodje aplikacije, na kateri se igra odvija na način strežnik/odjemalec. S pridobljenim znanjem izdelamo igro, ki deluje v spletnih brskalnikih, medtem ko jo uporabnik kontrolira z uporabo mobilne naprave. Za kontroliranje igre uporabljamo senzorje premikanja na mobilnih napravah. Podatke asinhrono pošiljamo na strežnik, kjer jih pretvorimo v ukaze, ki se izvršijo v sami igri. Za igro izdelamo preprost fizični model in detekcijo trkov. Podamo tudi morebitne izboljšave in možnosti uporabe igre.

Ključne besede: html, css, javascript, node, socket.io, express, igra, strežnik, odjemalec,mobilne tehnologije, mobilna naprava.

Abstract

In the thesis we discuss building of computer game with help of web technologies HTML5, CSS3, Javascript. We learn, what are these technologies and how we can use them to make computer games, which run in web browsers. We also learn about Node.js, Socket.io and Express Framework, which we use to build our application framework that runs with server/client protocol. With this knowledge we develop web browser game. Game can be controlled through mobile device. Controls are built using device motion sensors. We're sending controls data asynchronous to the server where we transform them to the command in the game. We develop a basic physics model and collision detection. At the end we discuss possible improvements and different options for game use.

Keywords: hmtl, css, javascript, node, socket.io, express, game, server, client, mobile technologies, mobile device .

Poglavje 1

Uvod

V podjetju, kjer sem zaposlen, izdelujemo produkt, s katerim lahko uporabniki izdelujejo napredne mobilne oglase. Za dobro prepoznavnost se udeležujemo raznih sejmov in predstavitev, na katerih imamo razstavljene različne mobilne naprave, na katerih si morebitni uporabniki/investitorji lahko ogledajo produkt in najzanimivejše oglase, ki se jih lahko s tem produktom ustvari. Iz izkušenj smo ugotovili, da uporabnike na predstavitvah najbolj pritegnejo preproste igre. Ker ima dandanes večina uporabnikov svojo mobilno napravo, so igre, ki omogočajo uporabo le-teh, zelo popularne, saj ni potrebno zagotoviti naprav, saj jih uporabniki že imajo.

Diplomsko delo se prične z izdelavo preproste igre, ki deluje v spletnih brskalnih in jo je mogoče kontrolirati s pomočjo mobilnih naprav. Ključno je, da igra deluje čimbolj gladko in da jo je mogoče igrati na različnih napravah. Igra je narejena s principom strežnik/odjemalec, ki nam omogoča prikaz dogajanja v sami igri na projektorju, medtem ko lahko uporabnik kontrolira objekt v igri s pomočjo lastne mobilne naprave, kar jo naredi zabavnejšo in zanimivejšo za obiskovalce. Prav tako je prednost igre ta, da jo lahko poganja računalnik, ki ima na voljo veliko večjo procesorsko in spominsko moč kot mobilne naprave, sliko dogajanja pa lahko prikažemo na projektorju ali TV sprejemniku. Odjemalčev del igre zajema podatke gibov naprave in jih posreduje strežniku, ki jih vključi v fizikalni model igre.

V 3. poglavju opišemo uporabljene tehnologije, s katerimi naredimo osnovno ogrodje aplikacije in kasneje tudi samo igro, njen grafični vmesnik ter njen fizični model. 4. poglavje se poglobi v pripravo materialov za grafični vmesnik in CSS-kodo, s katero si pomagamo pri izgledu in transformacijah elementov v igri. Na kratko razložimo tudi koncepte kreiranja grafičnih elementov z metodo risanja pikslov. V 5. poglavju pokažemo strukturo kode in razložimo pomen posameznih modulov ter kako delujejo med sabo. V 6. poglavju se poglobimo v samo ogrodje aplikacije in povemo, kaj je potrebno za poganjanje igre. Razložimo kako smo ga razvili in katere tehnologije smo uporabili. 7. poglavje razloži glavne koncepte igre skozi primere kode. Razložimo, zakaj potrebujemo glavno zanko za igro, kako zgradimo preprost fizikalni model za igro in kako naredimo detekcijo trkov. V 8. poglavju omenimo probleme, s katerimi smo se srečevali pri pisanju kode in izbiri tehnologij. V zadnjem poglavju podamo ugotovitve in morebitne izboljšave, ki bi igro še bolj približale uporabnikom ter jo naredile zanimivejšo.

Poglavje 2

Delovanje in koncepti igre

Igra se imenuje SpeedRacer. Že ime igre pove, da bomo opravljali z vozilom. Koncept igre je postavljen na avtocesto z več pasovi, na katerih so različna vozila in ovire, katerim se mora igralec s svojim vozilom izmikati. Pogled na igralno površino je s ptičje perspektive, kar poenostavi izrisovanje grafike za starejše naprave, saj lahko za prikazovanje igre uporabimo mobilno napravo (tablice kot so iPad), računalnik (prenosnik, stacionarni) ali pa tudi mobilno napravo z večjim zaslonom.

Igralčev avtomobil na začetku igre pospeši na določeno hitrost, ki jo skozi igro obdrži. Igralec s svojo mobilno napravo, ki seveda mora imeti senzor za premike, kontrolira svoj avto z nagibanjem same naprave levo in desno kot, da bi imel v roki volan. Odsotnost zavore oteži igro, saj mora igralec sprejemati hitre odločitve. Vsak trk v drugo vozilo ali oviro zniža igralčeve življenjske točke. Ko igralcu točk zmanjka, se igra konča.

Kontroliranje avtomobila daje občutek analognosti, saj moč zavijanja ustreza nagibu mobilne naprave. Če hoče igralec hitreje zaviti, mora hitreje in bolj nagniti svojo napravo.

Zaradi preprostega fizikalnega modela so odstranjene situacije, kot so prevračanje vozila, zdrs zaradi slabšega oprijema itd. Hitrost vozila se začasno prepolovi, če igralec zapelje iz vozišča na travnato površino na obeh straneh ceste ali če je zaznan trk z oviro ali drugim vozilom. V teh primerih se hitrost

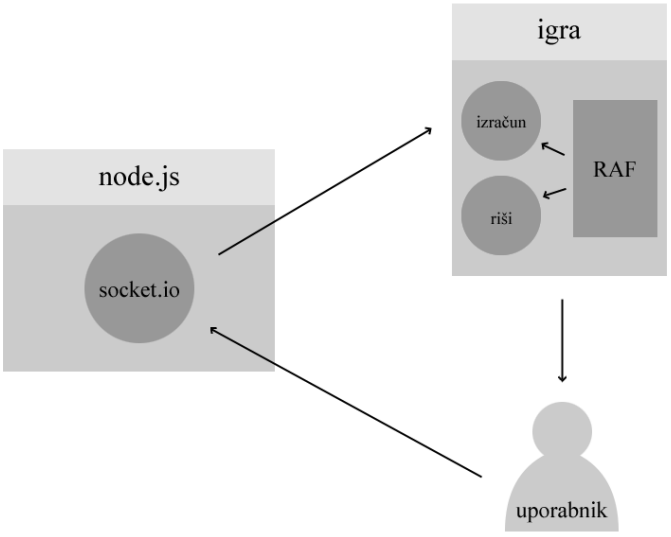
avtomatično poveča na največjo možno hitrost.

Ostala vozila in ovire, ki jih igralec srečuje na cesti, se generirajo naključno. Igra je tako vsakič drugačna, kar preprečuje igralcu, da bi si zapomnil vzorec pojavljanja ovir.

Poglavje 3

Struktura kode

Za lažjo predstavo projekta smo pripravili diagram delovanja igre, ki prikazuje vse potrebne elemente in interakcije med njimi. Strežniški del igre sestavljajo node.js in express ogrodje. Z uporabo socket.io knjižnice poslušamo igralčeve podatke, ki kontrolirajo obnašanje vozila. Te podatke posredujemo v zanko igre, kjer se upoštevajo pri izračunu novih pozicij. Po vseh izračunih glavna zanka igre izriše novo stanje igre in ga prikaže igralcu.



Slika 3.1: Arhitektura odjemalec strežnik igre Speedracer.

Poglavje 4

Uporabljene tehnologije

Za prikaz objektov in njihovega izgleda v brskalniku potrebujemo tehnologiji HTML5 ter CSS3. Objekte, ki jih s HTML-kodo dodamo v DOM, premikamo s spreminjanjem CSS lastnosti. Samo jedro oziroma zanko igre, skodiramo v Javascript jeziku. Ta poskrbi za centralizirano zanko, ki se ponavlja skozi celotno igro in tako skrbi za vse elemente ter relacije med njimi. jQuery je Javascript knjižnica, ki poenostavi določene procese in je uporabljena samo v ta namen. Node je uporabljen za strežnik, na katerem sprejemamo podatke od odjemalca in jih posredujemo naprej v Javascript kodo. Z Expressom si poenostavimo prikaz različnih strani igre. Tako kot jQuery je neobvezen in je namenjen poenostavitvi nekaterih procesov. Socket.io knjižnica je uporabljena za prenos podatkov med strežnikom in odjemalcem v realnem času.

4.1 Node.js

Node.js je programska platforma za strežniške in mrežne aplikacije [4]. Za pisanje aplikacij uporablja Javascript jezik. Node aplikacije so oblikovane tako, da povečajo pretovor in učinkovitost z uporabo ne-blokirajočih I/O ter nesinhronih dogodkov. Aplikacije tečejo kot enonitni procesi, medtem ko sam Node.js uporablja več niti za dogodke med datotekami in omrežjem. Node.js je zaradi svoje asinhrone narave običajno uporabljen za aplikacije, ki tečejo

v realnem času.

Node.js za izvrševanje kode uporablja Googlov V8 Javascript pogon. Velik del osnovnih modulov je napisan v Javascript jeziku. Vsebuje vgrajeno asinhrono I/O knjižnico za datoteke, vtiče in HTTP-komunikacijo. HTTP in podpora vtičem omogoča Node.js, da se obnaša kot spletni strežnik brez dodatne programske opreme za strežnike, kot je Apache.

Upravljalec paketov za Node (NPM) pride prenastavljen z Node.js strežniško platformo. Uporablja se za nameščanje Node programov iz NPM-registra. Paketi, ki jih najdemo v NPM-registru, se lahko raztezajo od enostavnih pomožnih knjižnic, kot je underscore.js, do poganjalcev nalog, kot je grunt.js, ki ni namenjen za uporabo na samo eni aplikaciji.

Priljubljeni NPM-moduli

- **Express** — Express.js je platforma za razvijanje spletnih aplikacij v Node in je dandanes standard za večino Node aplikacij.
- **socket.io in sockjs** — komponenta na strani strežnika dveh najpogostejših spletnih vtičnic.
- **Jade** — eden najpopularnejših pogonov za predloge, ki je že privzet v Express ogrodju.
- **Coffee-script** — prevajalnik, ki omogoča razvijalcem pisanje Node programov z uporabo Coffee sintakse.

Kje naj bi bil Node.js uporabljen?

- klepetalnice,
- API na object DB,
- čakalna vrsta vhodov,
- pretok podatkov,

- namestnik (proxy),
- nadzorna plošča za nadzor aplikacij,
- nadzorna plošča za nadzor sistema.

Kje naj Node.js ne bi bil uporabljen?

- težko strežniško procesiranje izračunov,
- strežniška spletna aplikacija z relacijsko podatkovno bazo.

4.2 Express

Express je majhno in prilagodljivo Node ogrodje za razvoj spletnih aplikacij, ki zagotavlja robusten nabor lastnosti za razvoj eno in več stranskih ter hibridnih spletnih aplikacij [5].

Express nudi tanko plast funkcijskih temeljev za katerekoli spletno aplikacijo, ne da bi se odpovedali funkcijam, ki jih poznamo v Node.

4.3 Socket I/O

Socket.IO je JavaScript knjižnica za spletne aplikacije, ki se izvajajo v realnem času [6]. Ima dva dela: knjižnico na strani odjemalca, ki deluje v brskalniku, in knjižnico na strani strežnika za Node. Obe komponenti imata skoraj identična API-ja. Enako kot Node, temelji na dogodkih.

Socket.IO primarno uporablja WebSocket protokol, vendar lahko po potrebi za delovanje uporabi tudi druge metode, kot so Adobe Flash vtičnik, JSONP pozivanje in AJAX dolgo pozivanje, hkrati pa zagotavlja enak vmesnik. Čeprav se lahko uporablja preprosto kot ovoj za WebSocket, zagotavlja veliko več funkcij, vključno z oddajanjem na več vtičnikov, shranjevanje podatkov povezanih z vsakim odjemalcem in asinhronim I/O.

Možna je namestitev z upravljalcem paketov za Node (NPM).

4.4 Javascript

JavaScript je objektni skriptni programski jezik, ki ga je razvil Netscape, da bi spletnim programerjem pomagal pri ustvarjanju interaktivnih spletnih strani [2].

Jezik je bil razvit neodvisno od Jave, vendar si z njo deli številne lastnosti in strukture. JavaScript lahko sodeluje s HTML-kodo in s tem poživi stran z dinamičnim izvajanjem. JavaScript podpirajo velika programska podjetja in kot odprt jezik ga lahko uporablja vsakdo, ne da bi pri tem potreboval licenco. Podpirajo ga vsi novejši spletni brskalniki. Trenutna različica je JavaScript 1.8.5.

Sintaksa jezika JavaScript ohlapno sledi programskemu jeziku C. Prav tako kot C JavaScript nima vgrajenih vhodno izhodnih funkcij, zato je njihova izvedba odvisna od gostitelja.

JavaScript se veliko uporablja za ustvarjanje dinamičnih spletnih strani. Program se vgradi ali pa vključi v HTML, da opravlja naloge, ki niso mogoče samo s statično stranjo. Na primer odpiranje novih oken, preverjanje pravilnost vnesenih podatkov, enostavni izračuni ipd. Na žalost različni spletni brskalniki izpostavijo različne objekte za uporabo. Za podporo vseh brskalnikov je zato treba napisati več različic funkcij.

Zunaj spleta se JavaScript uporablja v različnih orodjih. Adobe Acrobat in Adobe Reader ga podpirata v datotekah PDF. Podpirata ga tudi operacijska sistema Microsoft Windows in Mac OS X.

4.5 jQuery

Programi imajo svoje objektne modele, ki zagotavljajo dostop do gostiteljevega okolja, samo jedro jezika JavaScript pa je v vseh programih večinoma enako.

jQuery je JavaScript knjižnica namenjena poenostavitvi dela s HTML-objekti. Izdal ga je John Resing v BarCampu v New Yorku januarja leta 2006. Razvija ga ekipa, ki je trenutno pod vodstvom Dava Methvina. Uporablja ga

več kot 80 % od 10.000 najobiskanejših spletnih strani. jQuery je trenutno najpribljubljenejša Javascript knjižnica.

jQuery je brezplačna, odprtokodna programska oprema, ki je licencirana pod licenco MIT. jQueryjeva sintaksa je zasnovana za lažjo navigacijo HTML-dokumenta, izbiro DOM-elementov, ustvarjanje animacij, nadzor dogodkov in razvoj Ajax aplikacij. jQuery tudi zagotavlja možnost ustvarjanja vtičnikov čez JavaScript knjižnico. To razvijalcem omogoča ustvarjanje abstrakcije za interakcijo na nizki ravni, napredne učinke in napredne tematske vizualne gradnike. Modularni pristop knjižnice jQuery omogoča ustvarjanje močnih dinamičnih spletnih strani in aplikacij.

Microsoft in Nokia uporabljata jQuery na svojih platformah. Microsoft ga vključuje v Visual Studio za uporabo v okviru Microsoftovega ASP.NET AJAX ogrodja in ASP.NET MVC-ogrodja, medtem ko ga je Nokia integrirala v Web Run-Time platformo za razvoj vizualnih gradnikov. jQuery se uporablja tudi v MediaWiki-ju od verzije 1.16 naprej.

4.6 HTML5

HTML5 je označevalni jezik interneta, ki se uporablja za strukturiranje in predstavljanje vsebine za svetovni splet (WWW). Je peta revizija standarda HTML (ustvarjen leta 1990 in standardiziran kot HTML leta 1997). Njegovi temeljni cilji so izboljšati jezik s podporo za najnovejše multimedije, medtem pa obdržati čitljivost za ljudi in dosledno razumevanje za računalnike in naprave (spletne brskalnike, urejevalnike itd.). HTML5 naj bi vključeval ne le HTML 4.01, temveč tudi XHTML 1 in DOM Level 2 HTML.

HTML5 je poskus opredelitve enotnega označevalnega jezika, ki je lahko napisan bodisi v HTML bodisi v XHTML sintaksi. To vključuje podrobne modele obdelave, z namenom spodbujanja bolj interoperabilnih izvedb; razširja, izboljšuje in racionalizira označevalni jezik na voljo dokumentom, in uvaja označevlni jezik in uporabo programskih vmesnikov (API) za kompleksne spletne aplikacije. Zaradi enakih razlogov je HTML5 tudi potencialni kandi-

dat za mobilne aplikacije, ki tečejo na različnih platformah. Številne funkcije HTML5 so bile zgrajene, da lahko delujejo na nizko-zmogljivih napravah, kot so pametni telefoni in tablice.

HTML5 dodaja mnogo novih sintaktičnih lastnosti. Te vključujejo nove `<video>`, `<audio>` in `<canvas>` elemente, kot tudi integracijo razširljive vektorske grafike (SVG) in MathML za matematične formule. Te lastnosti so zasnovane, da poenostavijo vstavljanje in upravljanje multimedijskih in grafičnih vsebin na spletu, ne da bi potrebovali dodatne vtičnike ali vmesnike za programiranje aplikacij. Drugi elementi, kot so `<section>`, `<article>`, `<header>` in `<nav>` so narejeni, da obogatijo semantično vsebino dokumentov. Zaradi istega razloga so bili predstavljeni novi atributi, medtem ko so bili nekateri elementi ter atributi odstranjeni. Nekateri elementi, kot so `<a>`, `<cite>` in `<menu>` so bili spremenjeni, na novo definirani ali pa standardizirani. API in DOM nista več samoumnevna, ampak sta glavna dela HTML5 specifikacije. HTML5 do neke mere tudi definira potrebno procesiranje neveljavnih dokumentov, da bodo napake s strani različnih brskalnikov in uporabniških agentov obdelane enako.

4.7 CSS3

Predloga, ki določa izgled spletne strani (CSS) je jezik, ki opisuje izgled in format dokumenta. Večinoma je uporabljen za oblikovanje spletnih strani in vmesnikov, napisanih v HTML in XHTML. Uporabi se ga lahko tudi v XML-dokumentih, vključno s preprostim XML, SVG in XUL. CSS je temeljna specifikacija spleta in skoraj vse spletne strani ga uporabljajo za opisovanje izgleda.

CSS je primarno oblikovan, da omogoča ločitev vsebine dokumenta od prezentacije dokumenta vključno z elementi, kot so postavitev, barve in pisave. Ločitev prezentacije in vsebine izboljša dostopnost vsebine, zagotavlja več fleksibilnosti in kontrole v specifikaciji prezentacijskih karakterik, omogoča deljenje istega formata čez več strani in zmanjša kompleksnost in

ponavljanje v strukturi vsebine.

CSS tudi omogoča določeni spletni strani različen prikaz glede na tip prikaza, kot so zaslonski (on-screen), tisk (in-print), govorni (ko ga predvajajo brskalniki, ki omogočajo branje vsebine) in možnost prikaza za naprave, ki podpirajo Braille. Uporablja se ga lahko tudi za različen prikaz spletne strani glede na velikost zaslona naprave, na kateri je stran prikazana. Čeprav avtor dokumenta načeloma naredi povezavo do CSS-datoteke, lahko nekateri bralniki (readers) uporabijo svoje predloge, ki prepišejo avtorjeve. Če avtor ne doda povezave do predloge, bo brskalnik uporabil prednastavljene.

CSS navede shemo prioritet, ki določa, katero pravilo sloga bo uporabljeno, če je nastavljenih več kot eno za določen element. V tej t. i. kaskadi so izračunane prioritete in uteži, ki so dodeljene pravilom, da so rezultati predvidljivi.

CSS-specifikacije vzdržuje World Wide Web Consortium (W3C).

Poglavje 5

Priprava grafičnih elementov

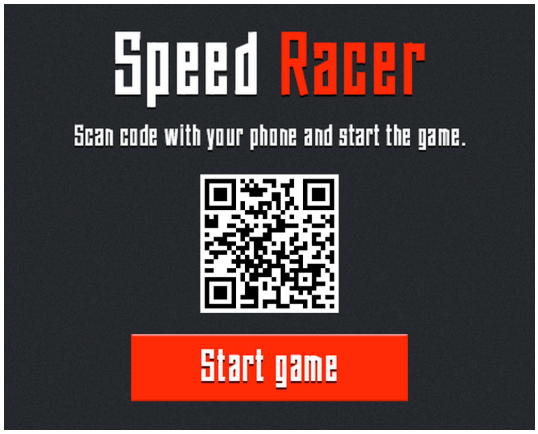
Za pripravo grafičnega vmesnika uporabimo program Adobe Photoshop. Stil igre je zasnovan na starih arkadnih igrah, kjer so bili objekti v igri narisani vsak piksel posebej [7]. Na začetku izdelamo predogled celotnega dizajna igre. Predvsem je potrebno paziti, da uporabljamo neuničujoč postopek pri grajenju grafičnih elementov, da jih lahko po potrebi dodajamo, odstranjujemo in spreminjamo.

Ko je izgled igre končan, iz celotnega vmesnika izrežemo in v posamezne slike shranimo izbrane elemente, ki jih nato uporabimo v HTML in CSS-kodi. Nekateri elementi se v igri ponavljajo, zato lahko s pravilno predpripravo materiala večkrat uporabimo enake slike in s tem zmanjšamo število povpraševanj ter čas nalaganja spletne strani.

5.1 Grafični vmesnik

Igra ima štiri glavne scene. Začetna scena (slika 5.1) je sestavljena iz navodil, naslova, QR-kode in gumba za začetek igre. Uporabnik z aplikacijo na svoji mobilni napravi poskenira QR-kodo, ki ga preusmeri na naslov za kontrole. Vse, kar nato preostane je, da klikne na gumb za začetek igre.

Druga scena (slika 5.2) je prikaz okolja, v katerem se igra dejansko odvija.



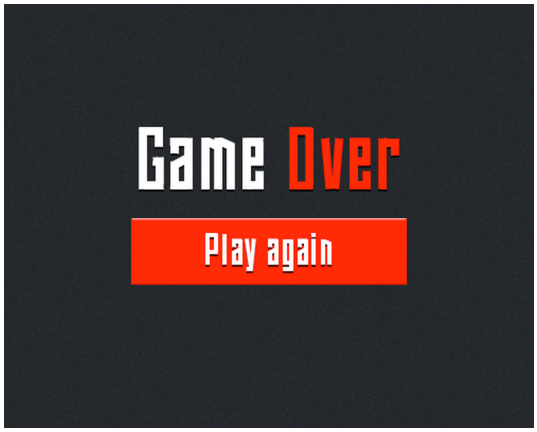
Slika 5.1: Začetni meni

Slika 5.2: Igralno polje

Sestavljena je iz statusne vrstice na vrhu, ki prikazuje naslov igre, igralčevo hitrost in življenjske točke. Preostali del zaslona je grafična predstavitev večpasovne avtoceste, na katero je postavljeno igralčevo vozilo. Oba robova sta travnata in predstavljata območje, kjer se igralčevo vozilo upočasni. Na tem območju ni ovir. Ta območja sta mišljena kot zasilna opcija, ki jo igralec lahko uporabi za izmikanje oviram.

Ovire (slika 5.3) se naključno pojavljajo na naključnih pasovih in imajo različno hitrost. Nekaj ovir se premika z različno hitrostjo, medtem ko so druge statične.

Slika 5.3: Ovire



Slika 5.4: Možnost ponovnega igranja

Tretja scena (slika 5.4) je zaključek igre, ki igralcu ponuja možnost za ponovno igranje igre s klikom na gumb za ponovno igranje.

Zadnja scena so kontrole (slika 5.5), ki se prikažejo na igralčevi mobilni napravi. Kontrole so zelo preproste, saj so na celotni strani prikazana le navodila, kako naj bi igralec upravljal svoje vozilo.

Slika 5.5: Kontrole

5.2 HTML5 in CSS3

Samo HTML-ogrodje igre je zelo preprosto. Z njim predpripravimo in prikažemo osnovne elemente igre.

Z uporabo meta oznake `<meta name="viewport>` nastavimo vidno polje, ki ni povečano ali pomanjšano. Prav tako onemogočimo dodatno spreminjanje velikosti s strani samega uporabnika.

V predlogi, ki določa izgled spletne strani (CSS), najprej ponastavimo osnovne elemente. S tem se znebimo vseh vnaprej določenih robov in mej.

Z nekaj vsebniki elementov poskrbimo, da je vsebina igre postavljena na sredino zaslona. Za element z razredom container uporabimo preprost trik `-webkit-transform: translateZ(0)`, s katerim prisilimo uporabo grafične kartice za izrisovanje vseh elementov znotraj le-tega. To pohitri izrisovanje igre, saj so grafične kartice v računalnikih večinoma primernejše za izrisovanje grafičnih elementov, kot primarni procesor.

Začetno sceno zgradimo z uporabo osnovnih elementov in nekaj CSS-kode. Zgeneriramo tudi sliko kode hitrega odziva, ki uporabnika po skeniranju le-te z mobilno napravo preusmeri na stran s kontrolami. Tu lahko najdemo tudi gumb, ki skrije začetno sceno in požene igro.

Končna scena uporablja skoraj enako strukturo kot začetna.

Statusna vrstica v drugi sceni, kjer prikazujemo dejansko igralno okolje, vsebuje dva elementa. Prvi element je vsebnik slike kazalca števca. Sam vsebnik ima za ozadje nastavljeno sliko števca. Z Javascript kodo posebej premikamo sliko kazalca števca in tako dobimo dinamični element, ki kaže trenutno hitrost vozila.

Drugi element je prikaz življenjskih točk, ki ga prikažemo s pomočjo neurejenega seznama `<ul>`. Vsaka življenjska točka je element tega seznama in je dinamično zgenerirana pred začetkom igre. To nam omogoča, da lahko med igro preprosto dodajamo ali odvzemamo elemente seznama in tako osvežujemo prikaz življenjskih točk igralca.

V samo strukturo HTML-kode so dodani tudi naslednji elementi. Nekateri so na začetku igre skriti in se prikažejo ob določenih situacijah.

5.2.1 Cestišče

Shranjen je samo kratek odsek cestišča, ki ga ponavljamo dokler ne zapolni celega vsebnika cestišča. Vsebnik nato v glavnemu objektu igre premikamo, da dobimo efekt hitrosti.

Travnata površina

Na obeh straneh cestišča imamo pas travnate površine. Na njej se vozilo upočasni za polovico največje hitrosti in začne rahlo trzati, kar daje občutek bolj grobe površine. Slika je pripravljena kot majhen odsek trave in je pripravljena na način, ki omogoča ponavljanje slike brez opaznih robov.

Vozilo

Igralčevo vozilo, ki ga postavimo na cestišče. Igralec spreminja pozicijo in naklon vozila s kontrolami na svoji mobilni napravi. Pozicija vozila se spreminja samo po horizontalni osi, glede na ukaze igralca. Ker je vozilo po vertikalni osi statično in je pozicionirano na dnu zaslona, ima igralec dovolj časa, da reagira na prihajajoče ovire.

Animacija eksplozije

Animacija eksplozije je sestavljena iz niza slik eksplozije, ki so shranjene v eno večjo sliko. S CSS-kodo poskrbimo, da se element eksplozije v pravem trenutku prikaže in da se pozicija ozadja spremeni v določenem intervalu. S tem se prikaže efekt animacije. Vsebnik eksplozije cel čas premikamo na lokacijo igralčevega vozila. Ko pride do trka z oviro, prikažemo vsebnik eksplozije in sprožimo animacijo. Po končani animaciji ga skrijemo.

Kontrole

Na igralčevi mobilni napravi prikažemo navodila za uporabo le-te.

Ovire

Med igro se mora igralec umikati različnim oviram. V CSS-datoteki definiramo razred obstacle, ki je skupen vsem oviram. HTML-elementi ovir se naključno generirajo znotraj Javascript kode, kjer se jim določi tudi tip, ki jim določi, kako bodo zgledali in kakšna bo njihova hitrost v igri.

Poglavje 6

Priprava ogrodja

Za strežnik smo izbrali Node.js, ki za kodiranje uporablja jezik Javascript. Ogrodje Express je zelo uporabno, saj je usmerjanje strani z njim zelo enostavno. Za komunikacijo med strežnikom in odjemalcem pa smo uporabili knjižnico Socket.IO, ki nam omogoča komunikacijo v realnem času.

Na spletni strani http://nodejs.org/ si lahko prenesemo najnovejšo različico Node.js. Po namestitvi preverimo, če imamo pravilno različico Node.js in upravljalca paketov za Node (NPM).

Ko preverimo, da sta Node.js in NPM uspešno nameščena, se lahko lotimo nameščanje Express ogrodja in knjižnice Socket.IO.

Po namestitvi vseh potrebnih tehnologij lahko s pomočjo Expressa preprosto zgeneriramo mapo projekta (koda 6.1) z vsemi potrebnimi podmapami.

```
speedracer
  |_ bin
  |_ public
  |_ routes
  |_ views
  app.js
  package.json
```

Koda 6.1: Struktura mape

V package.json datoteki (koda 6.2) najdemo glavne karakteristke projekta, kot so ime, verzija, privatnost, katere skripte naj se poženejo v različnih trenutkih, odvisnost različnih modulov in pogon projekta. Našo datoteko smo malce predelali in iz nje zbrisali nepotrebne pakete. Iz datoteke lahko razberemo, da za zagon aplikacije poženemo ukaz "node app.js" in da imamo poleg Node.js nameščena samo Express in Socket.IO paketa.

```
{
  "name": "speedracer",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.3.4",
    "socket.io": "~0.9.6"
  },
  "engines": {
    "node": "0.10.x"
  }
}
```

Koda 6.2: Datoteka package.json

Zaradi preproste strukture naše aplikacije smo iz mape projekta odstranili podmapi routes in bin. Tako sta nam ostali samo dve podmapi. V podmapi public se nahajajo vse slike, Javascript in CSS-datoteke, ki jih bomo potrebovali v projektu. V podmapi views pa lahko najdemo HTML-datoteke, potrebne za prikaz igre in kontrol.

Vsebino HTML-datotek v podmapi views in datoteke style.css v podmapi stylesheets smo razložili v poglavju 4.2. Glavna datoteka ogrodja je app.js,

ki poskrbi, da se vse konfiguracije pravilno nastavijo. V njej tudi inicializiramo Socket.IO povezavo, potrebno za komunikacijo med strežnikom in odjemalcem.

Najprej definiramo vse potrebne spremenljivke, ki jih bomo uporabljali v datoteki. Strežniku, ki smo ga ustvarili, sporočimo, na katerem naslovu in vhodu naj posluša.

Aplikaciji nastavimo poti do naših predstavitvenih HTML-dokumentov in do naše javne mape, kjer se nahajo slike, Javascript datoteke in CSS-datoteka. Nato še nastavimo poti do posameznih HTML-datotek in jih povežemo s pripadajočimi URL-naslovi. Vse, kar nam še preostane je, da inicializiramo povezavo s Socket.IO, kjer poslušamo na dogodek obračanja volana, ki ga posredujemo iz mobilne naprave in ga posredujemo naprej v datoteko game.js, ki posluša na le-ta dogodek ter s tem spreminja lastnost objekta igre za trenutno pozicijo volana.

Poglavje 7

Razlaga kode

Najpomembnejši del kode se nahaja v podmapi javascripts znotraj mape public. V omenjeni mapi se nahajajo najpomembnejše Javascript datoteke, ki tvorijo pogon igre. Tu lahko najdemo tudi podmapo libs, v kateri se nahajo vse potrebne zunanje knjižnice, v našem primeru knjižnica jQuery.

Javascript koda je razdeljena v tri datoteke. Za večje projekte bi lahko strukturo datotek še povečali in porazdelili v posamezne razrede, vendar je za naš projekt to dovolj.

- `controls.js` — vsebuje objekt kontrol (Controls), potreben za poslušanje senzorjev premikanja mobilne naprave in posredovanje podatkov objektu igre.

- `game.js` — vsebuje glavni objekt igre (Game), ki skrbi za pravilno interpretiranje vhodnih podatkov s strani objekta Controls, inicializacijo zanke igre, izračun fizikalnega modela, izris elementov in grafičnega vmesnika.

- `helpers.js` — vsebuje globalne funkcije, s katerimi si pomagamo pri kodiranju.

7.1 Kontrole

V controls.js datoteki ustvarimo objekt, ki posluša na dogodke orientacije naprave in jih s pomočjo metode drsečega okna posreduje glavni zanki igre. Instanco objekta ustvarimo in jo deklariramo v globalnem območju, ko je dokument pripravljen in se v celoti naloži.

V konstruktorju definiramo osnovne parametre in ustvarimo poslušalca dogodka orientacije naprave. Ustvarimo tudi lastnost objekta socket, ki se poveže na strežnik in vsebuje potrebne metode za oddajanje podatkov.

V metodi **deviceOrientation** dobimo vrednosti rotacije naprave po vseh treh oseh. Za simulacijo volana nas zanima predvsem os X, katere vrednosti shranimo v tabelo podatkov in jih omejimo na 180 stopinj (-90, 90). V tabeli podatkov držimo le toliko zadnjih podatkov, kolikor je vrednost spremenljivke **slidingWindow**. Preden oddamo vrednost rotacije X-osi, izračunamo povprečno vrednost vseh podatkov v tabeli in tako odstranimo vse podatke, ki bi po vrednosti odstopali od povprečja. S tem postopkom zagotovimo gladko zavijanje vozila, saj oddani podatki ne vsebujejo vrednosti, ki bi bile veliko večje ali pa manjše od predhodnih. Metoda drsečega okna je tukaj zelo uporabna, saj nekatere naprave niso sposobne oddajati urejenih podatkov in se med njimi najdejo odstopanja, katera s to metodo zgladimo. Urejene podatke nato oddamo preko dogodka, imenovanega **turning**.

7.2 Pomožne funkcije

Datoteka helpers.js vsebuje metode, s katerimi si v zanki igre lahko pomagamo pri nekaterih daljših opravilih.

Funkcija **precentToNumber** spremeni procente v številko glede na skupno vrednost in vrne izračunano vrednost.

Funkcija **getRandom** zgenerira in vrne naključno število med spodnjo in zgornjo mejo.

Funkcija **bind** se uporablja za ohranjanje konteksta pri klicu metode **tick**, ki se kliče skozi celotno igro, saj je glavni člen zanke igre.

Funkcija **requestAnimationFrame** ustvari vmesnik za programiranje aplikacij (API), potreben za animiranje znotraj brskalnikov [1]. Prednost te metode je predvsem v hitrejšem in konsistentnejšemu delovanju glavne zanke igre. Alternativna opcija bi bila uporaba intervalov, ki pa so nezanesljivi, saj ne moremo zagotoviti ponovitev v enakih časovnih razmikih, poleg tega pa so tudi procesorsko zahtevnejši. Zaradi procesorske zahtevnosti se na starejših napravah lahko izvajajo dlje, kot smo načrtovali. Zaradi prevelike zasičenosti procesorja se lahko zgodi tudi, da se nekateri cikli preskočijo, kar je opazno v sami animaciji igre, saj v tem primeru konkretno pade število prikazanih slik na sekundo.

Brskalnik lahko optimizira sočasne animacije skupaj v en cikel pretoka in ponovnega risanja, kar omogoča animacije z večjo zvestobo. Prav tako bo brskalnik ustavil animacijsko zanko, če le-ta ne bo potekala v trenutnem odprtem zavihku. S tem prihrani pri uporabi glavnega procesorja, grafičnega procesorja in spomina, kar posledično pripelje do manjše porabe baterije na napravah.

7.3 Zanka igre

Zanka igre je glavni koncept pri ustvarjanju iger, saj skrbi za neskončno ponavljanje kode, ki skrbi za izračun in izris vseh elementov igre [3]. V sami zanki igre shranjujemo čas med dvema klicema, kličemo **update** in **draw** funkciji, ki skrbita za izračun in prikaz elementov igre.

Za hitro in tekoče izvajanje igre je pomembno, da vse vrednosti elementov držimo shranjene v lastnostih objekta in jih ne povprašujemo vsako ponovitev igrine zanke, saj so to zelo zahtevni procesi, ki potrebujejo veliko procesorske moči ter spomina. Pozicije vseh objektov imamo shranjene, saj bi ob povpraševanju trenutne pozicije transformiranih elementov dobili vedno samo začetno pozicijo elementa. To nam omogoča uporabo CSS-transformacij, saj si le-te ne zapomnijo zadnje pozicije elementov, kar jih naredi zelo hitre in nezahtevne.

Zanko igre ustvarimo na podoben način kot kontrole, ko je dokument pripravljen in naložen. Na globalnem objektu ustvarimo novo instanco **game** objekta in nato kličemo init metodo, ki poskrbi za inicializacijo igre.

V konstruktorju objekta **game** najprej nastavimo privatne lastnosti s statičnimi vrednosti, ki predstavljajo faktorje za naš preprost fizikalni model in dimenzije igralne površine:

- največja hitrost vozila (px/s),
- največja hitrost avta na travnati površini (px/s),
- število življenjskih točk igralca,
- faktor za usmerjanje vozila,
- faktor za pospešek vozila,
- število pasov na cesti,
- širina igralne površine,
- višina igralne površine,
- širina travnate površine.

Nato s pomočjo jQuery knjižnice shranimo vse potrebne elemente za interakcijo z igro, kot so gumb za štart, gumb za ponovno igranje, začetna in končna scena, prikaz življenjskih točk ter vsebnik igre.

Ustvarimo Socket.IO objekt, s pomočjo katerega bomo poslušali na dogodke premikanja mobilne naprave.

V lastnosti objekta shranimo tudi vse elemente, s katerimi bomo morali v zanki igre opravljati in jih animirati. Vsaka lastnost je objekt, ki vsebuje vse potrebne podatke za lažje animiranje v zanki igre. Najpomembnejši podatki so širina, višina, pozicija, rotacija in pa jQuery vozel za morebitne dodatne poizvedbe, ki jih jQuery knjižnica omogoča. Ti objekti bi bili lahko posebni razredi, ki bi jih nadgrajevali in bi v sami zanki igre ustavarili samo njihove

instance. V teh razredih bi lahko ustvarili tudi njihove privatne metode za preračunavanje in risanje, vendar smo se zaradi preprostosti projekta odločili za globalni metodi **update** in **draw**.

S klicom funkcije **tick** poženemo zanko igre (koda 7.1). Ker se le-ta kliče rekurzivno poskrbimo, da v njej ohranjamo pravilen doseg objekta igre s pomožno funkcijo bind. Ponoven klic funkcije tick preko funkcije **requestAnimationFrame** namreč spremeni doseg objekta na globalni objekt.

```
this.tick = bind(this, this.tick);
```

Koda 7.1: Zanka igre

7.3.1 Funkcija reset

Ponastavi vse dinamične vrednosti igre na prvotno stanje in odstrani vse morebitne ovire na zaslonu. Prav tako ponastavi pozicijo vozila in življenjske točke.

7.3.2 Funkcija init

Pripne poslušalce dogodkov na gumba za štart in ponovno igranje igre. Funkcijo init kličemo, ko je dokument naložen, saj moramo imeti v objektnem modelu dokumenta (DOM) na voljo oba elementa gumbov. Spremenljivka `self` nam omogoča ohranjanje dosega objekta znotraj povratnih klicev poslušalcev dogodkov.

7.3.3 Funkcije za prikaz scen

Za prikaz različnih scen uporabljamo funkcije, kot so **hideMenu**, **hideEnd** in **showEnd**. Funkcije so zelo kratke, saj imajo samo eno preprosto nalogo.

7.3.4 Funkcija initLifeBar

Napolni pokazatelj življenjskih točk igralca s podano vrednostjo. Kot smo že omenili, je pokazatelj življenjskih točk definiran kot neurejen seznam, v katerega dodamo zgenerirane otroške elemente `<li></li>`. Vsak element predstavlja eno življenjsko točko, ki se v igri po potrebi odstranjujejo.

7.3.5 Funkcija decreaseLife

Zmanjša število življenjskih točk igralca za eno in posodobi prikaz le-teh. Funkcija spremeni tako spremenljivko, ki beleži število življenjskih točk kot tudi odstrani otroški element seznama.

7.3.6 Funkcija start

Začne izvajanje zanke igre s klicem funkcije **tick**. Poslušati začne na dogodke kontrol in postavi vozilo na prvotno mesto (sredina cestišča na spodnjem robu). Funkcijo **start** poženemo s klikom na štartni gumb v prvi sceni.

Funkcija **start** nastavi tudi nekaj spremenljivk, ki so potrebne za pravilno izvajanje zanke igre. Nastavi začetni čas izvajanja, kliče funkcijo **initLifeBar**, ki inicializira prikaz življenjskih točk in nastavi oznake **gameRunning**, ki nam pove, da se igra trenutno odvija in **stopObstacles**, ki nam pove, da lahko sprožimo prikaz ovir. Ovire se sicer v tem trenutku še ne sprožijo, vendar je pravilno nastavljena oznaka pove, da je od tega trenutka to mogoče.

7.3.7 Funkcija tick

Glavna funkcija zanke igre, ki poskrbi, da se zanka izvaja vsakič, ko je animacijski okvir brskalnika na voljo. Med vsako ponovitvijo beleži pretečen čas, katerega uporabimo pri izračunu pozicij elementov. V funkciji tudi kličemo update in draw funkciji, ki poskrbita za pravilen izračun in izris elementov, glede na podane fizikalne formule ter parametre.

7.3.8 Funkcija update

Skrbi za izračun pozicij vseh elementov v vseh možnih situacijah. Je prva izmed dveh najpomembnejših funkcij v celem objektu igre. Skrbi tudi za klic funkcije za detekcijo trkov med elementi. Kličemo jo s parametrom **elapsed**, ki nam pove, koliko časa je preteklo od zadnjega klica update funkcije, kar nam omogoča izračun novih pozicij elementov, glede na lastnosti objektov.

```
Game.prototype.update = function (elapsed) {
  // hitrost povečujemo kadarkoli je hitrost manjša od največje in
  // vozilo ni na travnati površini
  if (this.speed < this._maxSpeed && !this.carTwitching) {
    this.speed += 1;
  } else {
    // ko vozilo doseže največjo hitrost, sprožimo generiranje ovir
    if (!this.obstaclesTriggered) {
      this.triggerObstacles();
      this.obstaclesTriggered = true;
    }
  }

  // ko vozilo zapelje na travnato površino, mu začnemo zmanjševati
  // hitrost na vnaprej določeno hitrost za travnato površino in
  // nastavimo trzanje, ki simulira grobo površino
  if (this._car.position.x < this._grassWidth -
      Math.round(this._car.size.width / 2) || this._car.position.x
      > this._gameWidth - this._grassWidth -
      Math.round(this._car.size.width / 2)) {
    if (this.speed > this._grassSpeed)
      this.speed -= 2;
    this.carTwitching = true;
  } else {
    this.carTwitching = false;
```

```
}
this.currentSpeed = this.speed;

// v spodnji zanki posodobimo pozicije in hitrosti ovir
// zaradi različnih hitrosti ovir moramo paziti, da se med seboj
// ne prekrivajo
if (this.obstaclesArray.length > 1) {
  // sprehodimo se skozi tabelo vseh ovir
  for (var i = 0; i < this.obstaclesArray.length; i++) {
    // če je ovira premikajočega tipa in pred njo zaznamo drugo
    // oviro, ji moramo nastaviti enako hitrost kot oviri pred njo
    if (this.obstaclesArray[i].motion == 'moving') {
      // še enkrat se sprehodimo čez tabelo ovir
      for (var j = 0; j < this.obstaclesArray.length; j++) {
        // preverimo, če je ovira na enakem pasu cestišča kot
        // trenutna prav tako preverimo pozicijo ovire če je
        // pred trenutno oviro še ena ovira, ji spremenimo hitrost
        // na enako kot oviri pred njo
        if (this.obstaclesArray[j].lane ==
            this.obstaclesArray[i].lane &&
            (this.obstaclesArray[j].position.y + 40) >
            this.obstaclesArray[i].position.y) {
          this.obstaclesArray[i].speed =
              this.obstaclesArray[j].speed;
        }
      }
    }

    // ko je nova hitrost izračunana, lahko trenutni oviri
    // izračunamo novo pozicijo
    this.obstaclesArray[i].position.y +=
        Math.round(((this.currentSpeed -
        this.obstaclesArray[i].speed) * elapsed) / 1000);
```

```
    // poskrbimo še, da se ovira odstrani, ko doseže rob zaslona
    if (this.obstaclesArray[i].position.y > this._gameHeight) {
      this.destroyObstacle(i);
    }
  }
}

// osnovna hitrost vseh elementov
var basicYPosition = Math.round((this.currentSpeed * elapsed) /
    1000);

// posodobimo pozicijo cestišča
this._road.position.y += basicYPosition;
if (this._road.position.y >= 0)
  this._road.position.y = -this._road.size.height;

// posodobimo pozicijo travnate površine
this._grassLeft.position.y += basicYPosition;
if (this._grassLeft.position.y >= 0)
  this._grassLeft.position.y = -this._grassLeft.size.height;

this._grassRight.position.y += basicYPosition;
if (this._grassRight.position.y >= 0)
  this._grassRight.position.y = -this._grassRight.size.height;

// posodobimo pozicijo in rotacijo vozila
this._car.rotation = this.getCarRotation();
this._car.position.x += this.getCarPosition();

// posodobimo pozicijo eksplozije, da je vedno na sredini vozila
this._explosion.position = {
  x: this._car.position.x - this._explosion.size.width / 2 +
```

```
        this._car.size.width / 2,
    y: this._car.position.y,
  }

  // poskrbimo, da igralec ne more zaviti skozi levi in desni rob
  // igre s tem, da vozilo poravnamo in mu nastavimo statično
  // vrednost za horizontalno pozicijo
  if (this._car.position.x <= 0) {
    this._car.position.x = 0;
    this._car.rotation = 0;
  } else if (this._car.position.x >= (this._road.size.width -
      this._car.size.width)) {
    this._car.position.x = this._road.size.width -
        this._car.size.width;
    this._car.rotation = 0;
  }

  // kličemo metodo za preverjanje trkov
  this.checkCollisions();
};
```

Koda 7.2: Funkcija update

7.3.9 Funkcija draw

Poleg funkcije **update** je ena pomembnejših funkcij objekta igre. Skrbi za izris elementov na posodobljenih pozicijah. Prav tako pravilno pozicionira kazalec na števcu za hitrost vozila. Za izris objektov uporablja CSS-transformacije, saj so le-te performačno najbolj primerne za HTML-igre.

```
Game.prototype.draw = function () {
  // zarotiramo kazalec na števcu hitrosti
```

```
  // v CSS datoteki smo poskrbeli, da je točka rotacije pravilno
  // nastavljena
  this.currentSpeed < 20 ? 0 : this.currentSpeed;
  this._speedGaugePointer.node.css({ '-webkit-transform': 'rotate('
      + (this._speedGaugePointer.rotation + this.currentSpeed) +
      'deg)' });

  // nastavimo nove CSS transformacije za vse potrebne elemente
  // (vozilo, cestišče, travna površina, animacija eksplozije)
  this._road.node.css({ '-webkit-transform': 'translate3D(0, ' +
      this._road.position.y + 'px, 0)' });
  this._grassLeft.node.css({ '-webkit-transform': 'translate3D(0, '
      + this._grassLeft.position.y + 'px, 0)' });
  this._grassRight.node.css({ '-webkit-transform': 'translate3D(0,
      ' + this._grassRight.position.y + 'px, 0)' });
  this._car.node.css({ '-webkit-transform': 'translate3D(' +
      this._car.position.x + 'px, 0, 0) rotateZ(' +
      this._car.rotation + 'deg)' });
  this._explosion.node.css({ '-webkit-transform': 'translate3D(' +
      this._explosion.position.x + 'px, ' +
      this._explosion.position.y + 'px, 0)' });

  // nastavimo nove CSS transformacije za vse ovire
  if (this.obstaclesArray.length > 0) {
    for (var i = 0; i < this.obstaclesArray.length; i++) {
      this.obstaclesArray[i].node.css({ '-webkit-transform':
          'translate3D(' + this.obstaclesArray[i].position.x + 'px,
          ' + this.obstaclesArray[i].position.y + 'px, 0)' });
    }
  }
};
```

Koda 7.3: Funkcija draw

7.3.10 Funkcija triggerObstacles

Kliče funkcijo createObstacle, ki kreira novo oviro. To počne rekurzivno z naključnim zamikom med 200 ms in 500 ms. Rekurzivni klic funkcije se prekine, če nastavimo oznako **stopObstacles**.

7.3.11 Funkcija createObstacle

Zgenerira novo oviro na cestišču. V funkciji naključno zgeneriramo tip ovire in pas cestišča, na katerem se bo ovira pojavila. Zgeneriramo lahko 6 različnih tipov ovir. Polovica ovir je statičnih, druga polovica pa so vozila, ki imajo nastavljeno svojo hitrost premikanja, ki je vedno manjša od hitrosti igralčevega vozila, saj lahko le tako poksrbimo, da se bo igralec moral oviram izmikati.

Ovire so HTML-elementi, ki jim nastavimo pravilne CSS-razrede, ki jim določijo izgled in pozicijo na cestišču. Funkcija tudi definira javascript objekt, ki vsebuje vse potrebne podatke, ki jih uporabimo v funkciji **update** (tip, pas cestišča, pozicija, velikost, hitrost). Objekt nato shranimo v tabelo ovir in ga odstranimo, ko ovira doseže rob cestišča.

7.3.12 Funkcija destroyObstacle

Odstrani oviro iz objektnega modela dokumenta (DOM) in iz tabele ovir. Funkcija se kliče, ko ovira doseže rob cestišča in jo moramo odstraniti.

7.3.13 Funkcija getCarRotation

Izračuna in vrne trenutno rotacijo igralčevega vozila. Funkcija se kliče na začetku **update** funkcije. Rotacija se izračuna glede na kot nagiba igralčeve naprave. Če je igralčevo vozilo trenutno na travnati površini, pa se rotaciji doda še naključna vrednost, kar se v igri opazi kot trzanje vozila in simulira grobo površino.

7.3.14 Funkcija getCarPosition

Izračuna in vrne trenutno pozicijo vozila. Za pozicioniranje vozila uporabljamo faktor za usmerjanje vozila, ki poskrbi, da se vozilo ne premika po horizontalni osi, če miruje.

7.3.15 Funkcija checkCollisions

Skrbi za zaznavanje trkov med igralčevim vozilom in ovirami.

```
Game.prototype.checkCollisions = function () {
  // sprehodimo se čez vse ovire v tabeli ovir
  for (var i = 0; i < this.obstaclesArray.length; i++) {
    // izračunamo zadnjo sredinsko točko ovire
    var obstacleHitpoint = {
      x: Math.round(this.obstaclesArray[i].position.x +
          (this.obstaclesArray[i].size.width / 2)),
      y: this.obstaclesArray[i].position.y +
          this.obstaclesArray[i].size.height
    };
    // izračunamo sprednjo sredinsko točko vozila
    var carHitpoint = {
      x: Math.round(this._car.position.x + (this._car.size.width /
          2)),
      y: this._car.position.y
    };
    // trk sprožimo, če sta obe točki narazen manj kot širina ovire
    // in če je y-koordinata točke vozila med y-koordinato ovire in
    // višino ovire
    if (Math.abs(obstacleHitpoint.x - carHitpoint.x) <
        this.obstaclesArray[i].size.width && obstacleHitpoint.y -
        carHitpoint.y > 0 && obstacleHitpoint.y - carHitpoint.y <
        this.obstaclesArray[i].size.height) {
```

```
      if (!this.collisionDetected) {
        // nastavimo oznako za trk, da se ta ne ponovi večkrat
        this.collisionDetected = true;

        // zapomnimo si s katero oviro je bil zaznan trk
        this.obstacleHit = this.obstaclesArray[i];
        // sprožimo trk
        this.triggerCollision(this.obstaclesArray[i]);
      }
    // ko vozilo in ovira nista več v poziciji za trk, odstranimo
    // oznako in s tem omogočimo nove trke
    } else if (this.obstacleHit === this.obstaclesArray[i]) {
      this.collisionDetected = false;
    }
  }
};
```

Koda 7.4: Funkcija checkCollisions

7.3.16 Funkcija triggerCollision

Skrbi za ustrezno zmanjšanje življenjskih točk in hitrosti, prikaz animacije eksplozije in morebiten zaključek igre, če igralec ostane brez življenjskih točk.

7.3.17 Funkcija playCollisionAnimation

Elementu eksplozije doda razred animated, ki HTML-element prikaže in ga animira. Sproži jo funkcija **triggerCollision**.

7.3.18 Funkcija stopGame

Ustavi igro, kliče funkcijo za ponastavitev vseh parametrov in prikaže končno sceno.

Poglavje 8

Sklepne ugotovitve

Z izdelavo igre smo bolje spoznali, kako delujejo mobilne naprave, kako v realnem času pošiljati podatke iz naprave na strežnik, zakaj je potrebno kodo za HTML-igre optimizirati in seveda, kako pripraviti grafične elemente.

Pri kodiranju smo naleteli kar na nekaj problemov. Za gostovanje spletne strani smo se najprej odločili uporabiti Heroku platformo, ki pa se je v zastonjski različici pokazala za prepočasno, saj je imela preveč zakasnitev za igro, v kateri so potrebne hitre akcije. Za predstavitvene namene smo se odločili igro zagnati lokalno, saj le tako poskrbimo za najboljše pogoje, prav tako pa je igro še vedno mogoče predstavljati na sejmih. Problem se je pokazal tudi v zanki igre, ki je bila na začetku skodirana s pomočjo intervalov, ki so procesorsko prezahtevni in nezanesljivi. Zaradi počasnosti nekaterih naprav se je izrisovanje igre močno upočasnilo, kar je pripeljalo do slabšega zaznavanja trkov in posledično slabše igralne izkušnje. Problem smo rešili z uporabo **requestAnimationFrame** metode in CSS-tranzicij, kar je delovanje igre zelo pospešilo.

Igro bi lahko precej izboljšali s posodobljeno kodo, ki bi manj smetila pomnilnik in bi tako povečala intervale čiščenja pomnilnika, ki povzročajo rahlo zatikanje med igro. Posodobili bi lahko tudi samo logiko igre, dodali bi lahko več različnih ovir z dodatnimi lastnostmi. Izboljšali bi lahko fizikalni pogon igre, saj sedaj uporabljamo zelo osnovnega. Igro bi lahko naredili še

tekmovalnejšo, če bi dodali štetje točk, glede na pretečen čas in shrambo le-teh v bazi. S tem bi lahko dodali na končno sceno še prikaz najboljših igralcev.

Igri smo dodali tudi različico za uporabo na Blackberry mobilnih napravah, ji odstranili strežniški in odjemalski del ter kontrole prestavili v samo kodo igre. Ta različica za demonstracijo ne potrebuje dveh naprav, igro se lahko igra z nagibanjem naprave, na kateri igralec igra.

Literatura

[1] P. Irish (Februar 2011) dostopno na: http://www.paulirish.com/2011/requestanimationframe-for-smart-animating

[2] D. Crockford "Javascript: The Good Parts", May 2008

[3] Koonsolo "deWiTTERS Game Loop", Julij 2009 dostopno na: http://www.koonsolo.com/news/dewitters-gameloop/

[4] Joyent, inc "Node.js Documentation" dostopno na: http://nodejs.org/documentation/

[5] Express Framework dostopno na: http://expressjs.com/guide.html

[6] Socket.IO dostopno na: http://socket.io/docs/

[7] A. Feldman "Designing Arcade Computer Game Graphics", 2000