Univerza v Mariboru

Fakulteta za organizacijske vede

Magistrsko delo

Organizacija in management informacijskih sistemov

ODPRTOKODNA SPLETNA ARHITEKTURA ZA RAZVOJ VEČIGRALSKIH IGER

Mentor: doc. dr. Uroš Rajkovič

Kandidat: Taj Pelc

Kranj, avgust 2013

ZAHVALA

Zahvaljujem se mentorju doc. dr. Urošu Rajkoviču za pomoč pri načrtovanju in izdelavi magistrske naloge.

Hvala Frenku T. Sedmaku Nahtigalu za pomoč pri razvoju prototipne igre Shell Wars.

Zahvaljujem se tudi drugim članom ekipe Chilly Framework, ki so Frenk Ten Sedmak Nahtigal, Andrés Ledesma, Bidhya Sagar Ghimire in Jari Pirinen.

Hvala moji družini za podporo v času študija.

Hvala Tatjani Ličen za lektoriranje magistrske naloge.

POVZETEK

Čeprav je na voljo veliko orodij za izdelavo spletnih iger HTML5, je razvoj večigralske igre še vedno zapleten. Razvijalci morajo namesto aplikacije za odjemalce razviti tudi strežnik in poskrbeti za komunikacijo med vsemi odjemalci in strežnikom. Da bi skrajšali čas, potreben za izvedbo večigralstva, smo razvili spletno arhitekturo za razvoj večigralskih iger, ki predpisuje pravilo, po katerem lahko delujejo spletne igre. Na podlagi predlagane arhitekture smo razvili celovito platformo za razvoj večigralskih iger Chilly Framework, ki deluje kot spletni strežnik in ponuja funkcije za sinhronizacijo stanja igre med odjemalci in strežnikom. Da smo preverili, ali arhitektura v praksi deluje, smo s pomočjo Chilly Frameworka izdelali prototipno igro Shell Wars.

KLJUČNE BESEDE:

- spletna arhitektura
- razvoj iger
- Chilly Framework
- Shell Wars

ABSTRACT

The development of multiplayer web-based games is still complex, even though a large number of tools for developing HTML5 games exist. In a multiplayer game, developers have to create two separate applications, one for the server and one for the client. They also have to implement two-way communication between the client and the server and keep all the clients and the server in sync. We developed a platform for the development of multiplayer games based on a proposed architecture called Chilly Framework. The platform acts as a web server and exposes a front-end and a back-end library which can be used to communicate between the clients and the server and keep the game state in sync. To test wheter the architecture works in practice we also developed a prototype game with Chilly Framework called Shell Wars.

KEYWORDS:

- web architecture
- game development
- Chilly Framework
- Shell Wars

KAZALO

1. UVOD

1.1. PREDSTAVITEV PROBLEMA

V zadnjih letih smo priča zelo hitremu razvoju spletnih tehnologij in ta trend se nadaljuje. Za to so najzaslužnejši proizvajalci spletnih brskalnikov, ki so s prehodom na hitrejši tempo izdajanja različic močno skrajšali čas med specifikacijo novih funkcij ali standardov in njihovo izvedbo. S politiko nadgrajevanja brskalnika brez uporabnikove interakcije ima čedalje več ljudi nameščeno zadnjo različico svojega brskalnika in s tem dostop do najnovejših tehnologij. Tudi Internet Explorer, ki je z nespoštovanjem standardov dolgo zaviral razvoj spleta (Gwilliam, 2012), je v različici 9 podprl strojno pospeševanje grafike in zadnjo različico JavaScripta, ECMAScript 5 (Bright, 2011).

Napredek pri razvoju brskalnikov pomeni, da lahko razvijalci uporabijo nove tehnologije, ki jih prinaša HTML5 v različnih storitvah in produktih. Adobe Flash, ki je bil še do nedavnega standard za razvoj spletnih iger, se danes vse bolj umika na stranski tir v prid uporabi HTML-ja in JavaScripta(Vaughn-Nichols, 2011). Veliki trije, Google, Apple in Microsoft, to dejavno podpirajo, ni pa omejeno le na osebne računalnike. HTML in JavaScript se seli iz brskalnika tudi na raven operacijskega sistema.

Kljub bliskovitemu razvoju tehnologije je področje spletnih HTML-iger še vedno razmeroma novo. Na njem prevladujejo tehnološki demoti pred igrami, ki so namenjene širokim množicam, čeprav danes razvoj spletne igre za HTML ni več tako zahteven, kot je bil, saj je na voljo več programskih ogrodij JavaScript, namenjenih razvoju iger. To so ImpactJS, LimeJS in CraftyJS. Ti olajšajo delo z novimi elementi HTML-ja (Canvas, Audio ipd.), hkrati pa vsebujejo že pripravljene najpogostejše gradnike iger, kot so fizikalni pogon, izrisovalni sistem in upravljanje z zvokom.

Problem omenjenih programskih knjižnic je, da so namenjene predvsem izdelavi iger, ki se izvajajo zgolj v odjemalcu in so omejene na enega igralca. Razvoj podpore večigralstvu je zahteven, saj je treba spremeniti arhitekturo igre. Razvijalec igre mora izdelati vsaj dve fizično ločeni aplikaciji, ki delujeta na odjemalcu in strežniku. Ko fizično ločimo odjemalca od strežnika, je pri načrtovanju in izvedbi treba upoštevati tudi povezavo med njima. Vsi podatki, ki se morajo deliti med strežnikom in odjemalcem, potujejo po omrežju. Igra lahko dobro deluje le, če smo pri načrtovanju upoštevali omrežno latenco, če preverjamo veljavnost vseh zahtevkov, da preprečimo goljufije, in če smo poskrbeli za sinhronizacijo stanja igre med strežnikom in vsemi odjemalci. Ta dodatna kompleksnost razvoja večigralskih iger pomeni velik vložek tako časovno kot tudi finančno. Tudi po razvoju večigralske igre je zahtevnost vzdrževanja sistema bistveno višja kot pri enoigralskih igrah. Treba je vzdrževati strežnike, reševati težave, ki se pojavijo zaradi prevelikega navala igralcev, in

zakrpati luknje, ki jih lahko nekateri igralci zlorabljajo. Igralcem je treba nuditi tehnično pomoč. Čim bolj je treba zmanjšati število izpadov storitve, saj v nasprotju z enoigralsko igro ob padcu strežnika igre ni mogoče igrati (Ravuya, 2010).

Da bi razvijalcem olajšali delo, smo razvili arhitekturo za razvoj večigralskih iger in na tej podlagi izdelali celovito programsko platformo za razvoj večigralskih HTML5-iger, ki tečejo v najnovejših verzijah modernih brskalnikov, ter jo poimenovali Chilly Framework. V magistrskem delu smo ugotavljali, ali je predlagana arhitektura in na njeni podlagi izdelana platforma Chilly Framework zmožna uspešnega poganjanja spletne večigralske igre. Zato smo izdelali prototip večigralske igre, imenovan Shell Wars.

Chilly Framework je odprtokodna platforma za razvoj večigralskih iger, izdelana na podlagi predlagane arhitekture, ki zajema dva dela. Strežniški del skrbi za stanje igre in omogoča sredstva, ki jih igra za delovanje potrebuje. Vsebuje igralni pogon in skrbi, da je stanje igre časovno usklajeno med odjemalci (igralci). Drugi del, ki deluje v odjemalcu (brskalniku), omogoča povezavo s knjižnicami za razvoj iger, povezavo na strežnik in skrbi za osveževanje stanja igre v brskalniku glede na stanje igre na strežniku.

Osnovne funkcije sistema so:

- dvosmerna komunikacija med strežnikom in odjemalci,
- usklajenost stanja igre med strežnikom in odjemalci,
- združljivost z različnimi knjižicami za izdelavo grafičnega vmesnika na strani uporabnika,
- shranjevanje stanja iger in
- serviranje osnovnih sredstev igre (struktura, grafika, zvok).

V nadaljevanju smo opisali predlagano arhitekturo in odprtokodno platformo za razvoj večigralskih iger Chilly Framework. Opredelili smo arhitekturo igre in tehnične zahteve, ki smo jim želeli zadostiti. Z uporabo Chilly Frameworka smo na podlagi tehničnih zahtev in načrta izvedli prototip igre in preverili njegovo delovanje.

1.2. PREDPOSTAVKE IN OMEJITVE

Predpostavljamo, da sta JavaScript in HTML5 dovolj razvita tako z vidika funkcionalnosti kot tudi razširjenosti, da je razvoj spletne arhitekture za razvoj večigralskih iger v tem okolju smiseln.

Hiter razvoj tehnologije pomeni določeno oviro, saj se nekateri deli arhitekture hitro spreminjajo. To je posledica razvoja in sprememb v programskih knjižicah, ki smo jih uporabili, saj so relativno nove. Predlagana arhitektura je tako le ena izmed mogočih, za katero predpostavljamo, da je v času nastajanja dobra rešitev zastavljenega problema.

Čeprav se podpora brskalnikov izboljšuje, omenjena arhitektura ne bo delovala na vseh brskalnikih in napravah.

1.3. METODE DELA

Za osnovni raziskovalni pristop smo izbrali strategijo načrtovanja in razvoja informacijskega sistema (angl. Design and Creation). Cilj načrtovanja in razvoja je prispevati k znanju o razvitih informacijskih artefaktih, ki so lahko konstrukti, modeli, metode ali primer, oziroma razviti informacijski sistem(Hevner, March, Park, & Ram, 2004).

Uporabili smo pristop Waterfall k razvoju programske opreme. Razvoj poteka v več stopnjah, med katerimi obstaja povratna zanka. Ko dobimo novo znanje, se vrnemo na eno izmed prejšnjih stopenj, kjer lahko uvedemo popravke. Proces teče z ene stopnje na drugo. To so analiza zahtev, načrtovanje, izvedba, testiranje, namestitev in vzdrževanje. (Melonfire, 2007)

V nadaljevanju smo šli skozi naslednje korake:

- analizo tehnologije,
- načrt arhitekture,
- izdelavo odprtokodne prototipne izvedbe arhitekture (Chilly Framework) in
- izdelavo testne igre na podlagi odprtokodne izvedbe (Shell Wars).

Izvedba je v celoti izdelana v programskem jeziku JavaScript tako na odjemalcu kot tudi na strežniku po principih opisanih v knjigi JavaScript: The Good Parts (Crockford, 2008). Strežnik deluje v odprtokodnem okolju Node.js. Za sinhronizacijo stanja igre in dvosmerno komunikacijo s strežnikom smo razvili lastne rešitve na podlagi tehnologije dolgega izpraševanja. Za razvoj grafičnega uporabniškega vmesnika prototipne igre smo uporabili Crafty Game Engine in jQuery. Uporabili smo podatkovno bazo Redis.

Magistrska naloga temelji na delu, ki smo ga začeli v okviru predmeta Project Course na Åbo Akademi University v Turkuju na Finskem, pri katerem so sodelovali (Åbo Akademi, 2012):

- Taj Pelc, tehnični vodja in glavni programer,
- Frenk Ten Sedmak Nahtigal, programer uporabniškega vmesnika,
- Andrés Ledesma, programer strežnika,
- Bidhyja Sagar Ghimire, vodja dokumentacije, in
- Jari Pirinen, vodja projekta.

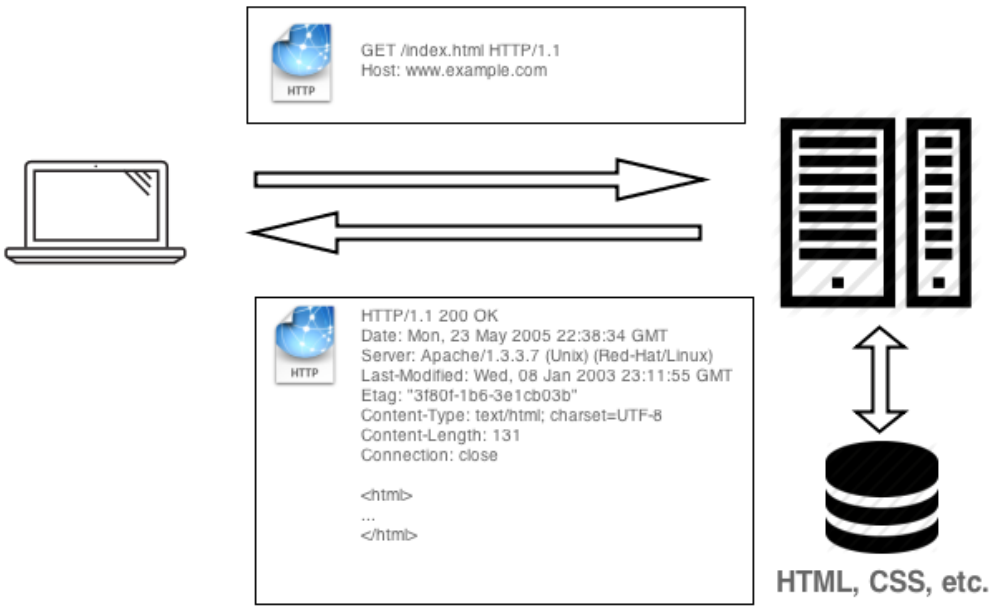
2. SPLETNA ARHITEKTURA

Arhitektura odjemalec – strežnik je osnova spleta, uporablja pa se tudi v večigralskih igrah, ki delujejo kot samostojne aplikacije. Poglejmo najprej, kako Gregory v knjigi Game Engine Architecture definira model odjemalec – strežnik, ki se uporablja pri razvoju večigralskih iger, in smo ga prilagodili našim potrebam.

V modelu odjemalec – strežnik se velika večina logike izvaja na enem samem strežniku. Koda na strežniku je podobna kodi neomrežne igre enoigralski igri. Na strežnik se poveže več odjemalcev, ki sodelujejo v spletni igri. Odjemalec je v tem modelu zgolj "neumen" pogon za upodabljanje (*rendering engine*), ki hkrati zajema uporabnikove akcije na njegovih napravah (tipkovnica, miška) in upravlja igralčeve enote. V večini primerov odjemalec zgolj upodobi in izvede akcije, ki jih dobi od strežnika. V kodi odjemalca se veliko pozornosti nameni temu, da se dejanja uporabnika takoj izvedejo na zaslonu brez zamika. S tem se izognemo za uporabnika izjemno neprijetnemu občutku zapoznelega odziva. Koda, ki skrbi za to, se imenuje predvidevanje odjemalca (client-prediction). Poleg omenjene kode odjemalec ni bistveno več kot pogon za upodabljanje (rendering engine), pogon za zvok (audio engine) in nekaj mrežne kode. Strežnik se lahko izvaja na za to namenjenem računalniku, čemur rečemo način namenskega strežnika (dedicated server). Ni nujno, da sta strežnik in odjemalec na ločenih računalnikih. Kadar se oba izvajata na istem računalniku, govorimo o načinu odjemalca nad strežnikom (client-on-top-of-server). Igralna zanka v načinu odjemalec – strežnik je lahko izvedena na več različnih načinov. Ker sta strežnik in odjemalec konceptualno ločeni entiteti, ju je mogoče izvesti na različne načine. Lahko sta to dva ločena procesa (aplikaciji) ali dve različni niti izvajanja v enem procesu. Oba načina imata to slabost, da dodatna komunikacija med strežnikom in odjemalcem poveča skupno uporabo virov, kar upočasni izvajanje programa. Nekatere večigralske igre zato poganjajo tako strežnik kot odjemalec v eni zanki, pri čemer lahko tečeta ob različnih stopnjah. Na primer v igri Quake teče strežnik pri 20 FPS (sličicah na sekundo), kar se prevede 50 ms na sličico, odjemalec pa teče pri 60 FPS; 16,6 ms na sličico. To je izvedeno tako, da se glavna zanka izvaja pri hitrejšem tempu, koda strežnika pa se požene le enkrat na vsake tri sličice. Sledi se času, ki je pretekel od zadnje posodobitve strežnika, in ko doseže ali preseže 50 ms, se izvede strežniška koda in merilnik časa se ponastavi. (Gregory, 2009)

Strežniški del arhitekture, ki smo jo zasnovali, temelji na tehnologiji, ki v zadnjem času dobiva veliko pozornosti: Node.js. Uporabljajo ga tako velika podjetja, kot so Microsoft, VMWare, Ebay, Yahoo in drugi, pa tudi manjši uporabniki. Glavna prednost tehnologije node.js je ta, da omogoča razvoj aplikacij v realnem času, ki so visoko nadgradljive. Največja težava protokola HTTP, ki se uporablja za komunikacijo med brskalnikom in spletnim strežnikom, je, da ne omogoča dvosmerne komunikacije v realnem času. To povzroča velike težave pri razvoju večigralskih iger.

Klasični spletni strežniki na protokolu HTTP delujejo po načelu zahtevek/odgovor, kot prikazuje Slika 1, in ne ohranjajo odprte povezave med odjemalcem in strežnikom. Odjemalec pošlje zahtevek za spletno stran na strežnik. Strežnik preveri, ali je dostopen zahtevan vir, in ob uspešni poizvedbi odgovori s statusom 200 OK in vsebino, ki jo je odjemalec zahteval. Povezava med strežnikom in odjemalcem se nato prekine, in šele ko odjemalec ponovno pošlje zahtevek, se povezava ponovno vzpostavi. Strežnik po prekinitvi povezave ne more potisniti podatkov odjemalcu, takoj ko se spremenijo. Za delovanje spletne igre pa je dvosmerna komunikacija bistvenega pomena. (Scott Allen, 2012)



Slika 1: Potek zahtevka HTTP

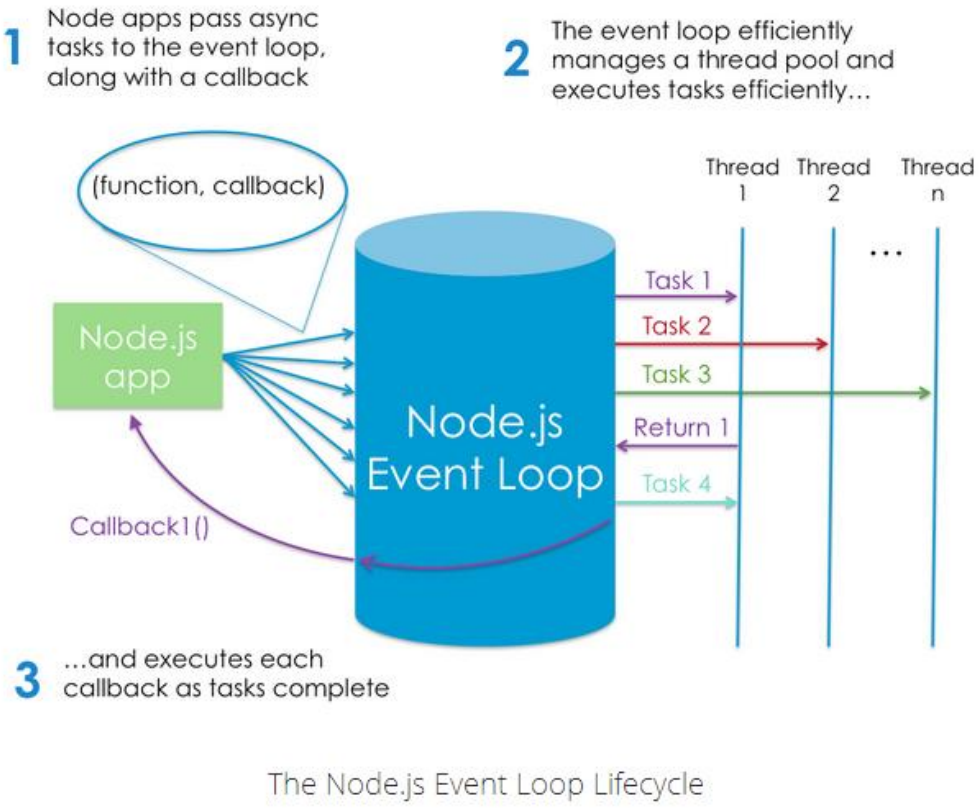
2.1. OKOLJE NODE.JS

Node.js je dogodkovno usmerjeno strežniško okolje v JavaScriptu, ki deluje na podlagi Googlovega pogona V8, vgrajenega tudi v brskalnik Chrome. V8 omogoča, da node.js prevaja JavaScript v strojno kodo. Zato je delovanje pogona, ki tudi podpira vse največje platforme (Windows, OS X in Linux), zelo hitro.

Dodatna prednost za naše potrebe je ta, da se tako na čelnem delu sistema kot tudi v njegovem zalednem delu uporablja samo en programski jezik, kar zmanjša zapletenost razvoja. Za Node.js obstaja tudi veliko knjižnic, ki jih je enostavno namestiti in vključiti v lasten produkt z uporabo upravljalnika paketov NPM. V sistemu NPM je trenutno že 39.187 paketov, dnevno pa se jih prenese 4.222.037. To

kaže na visoko uporabo sistema NPM. Uporabili smo ga tudi pri razvoju našega sistema (nodejs, 2013).

V Node.js se vsi zahtevki, ki potrebujejo dostop do vhodno-izhodnih virov, izvedejo asinhrono. Tradicionalen nitni pristop k asinhronemu programiranju je težaven, potraten s sistemskim spominom in neučinkovit. Node.js zato uporablja dogodkovno zanko, ki skrbi za vse asinhrone operacije. Vsakič, ko mora node izvesti operacijo, ki bi blokirala druge operacije, kot je dostop do trdega diska, doda asinhroni zahtevek na dogodkovno zanko skupaj s povratnim klicem, nato nadaljuje izvajanje programa. Dogodkovna zanka sledi asinhroni operaciji, in ko se ta izvede, vrne rezultat aplikaciji. Tako omogoča izvajanje veliko hkratnih operacij, brez upočasnitve. Slika 2 prikazuje delovanje node.js aplikacije in izvajanje dogodkovne zanke (Constantine, 2013).



Slika 2: Dogodkovna zanka Node.js (vir: udemy.com)

Poleg tega da deluje v klasičnem načinu HTTP, node.js podpira tudi spletne vtičnice (web sockets) s pomočjo knjižnice Socket.io. Spletne vtičnice rešujejo vprašanje klasične spletne arhitekture zahtevkov HTTP in omogočajo dvosmerno komunikacijo med strežnikom in odjemalcem. Spletne vtičnice so le eden izmed načinov za vzpostavitev dvosmerne komunikacije, ki je ne podpirajo vsi brskalniki. Knjižnica

pred razvijalcem skrije tehnologijo, uporabljeno za dvosmerno komunikacijo, in samodejno izbere najprimernejšo glede na podporo brskalnika (JSONP, AJAX, Flash sockets).

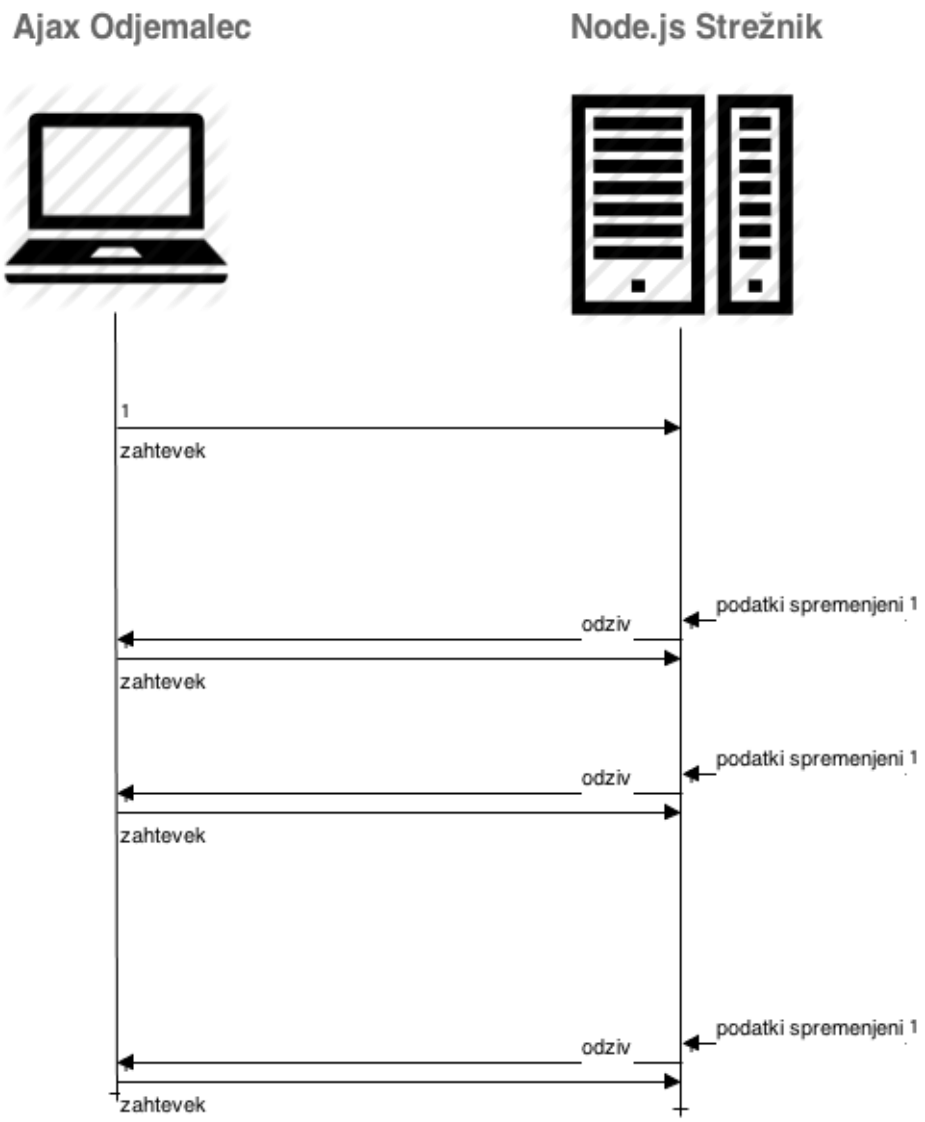
Predlagana arhitektura razdeli strežnik na dve logični enoti. Prva enota je klasični strežnik HTML za statične datoteke, potrebne za delovanje igre (HTML, CSS, JS, zvok, grafika). Izvedli smo ga s knjižnico za Express. Takoj ko uporabnik odpre spletno stran, na kateri je igra, se s strežnika prenesejo datoteke, potrebne za igranje.

Druga logična enota je sistem, ki skrbi za sinhronizacijo igre med uporabniki in prenašanje sporočil. Da bi lahko v spletnem okolju zadostili potrebi po komunikaciji v realnem času od uporabnika proti strežniku in od strežnika do uporabnika, smo razvili knjižnico Chilly Framework. Predlagana arhitektura za komunikacijo od strežnika proti uporabniku uporablja več posodobitvenih kanalov, ki prenašajo podatke s tehnologijo dolgega izpraševanja (*long-polling*), v nasprotni smeri pa s tehnologijo asinhronih zahtevkov HTTP. Poglejmo si, kako se tehnologija dolgega izpraševanja primerja s klasičnimi zahtevki. Pri dolgem izpraševanju se še vedno pošiljajo klasični zahtevki HTTP s pomembno razliko. Dokler strežnik nima svežih podatkov za odjemalca, pusti povezavo odprto. Ko se podatki v sistemu spremenijo, jih lahko strežnik nemudoma potisne po že odprti povezavi. Odjemalec jih prejme, obdela in nemudoma ponovno vzpostavi povezavo. S tem se izognemo omejitvi, da strežnik ne more vzpostaviti povezave na odjemalce.

Iz Slike 3 na naslednji strani je razviden časovni potek dolgega izpraševanja. Ponovimo korake.

1. Odjemalec prvi vzpostavi povezavo (pošlje zahtevek).
2. Strežnik jo zamrzne in pusti odprto, dokler se podatki v sistemu ne spremenijo, potem pa jih pošlje odjemalcu.
3. Odjemalec jih prejme in nemudoma ponovno vzpostavi povezavo, ki se spet zamrzne.

Korake ponavljamo, vse dokler želimo pošiljati podatke s strežnika k uporabniku, ne da bi od njega zahtevali osvežitev strani. S tako komunikacijo med odjemalcem in strežnikom se lahko dovolj približamo realnemu času prenosa podatkov, da igralec ne opazi zamika, dokler so zakasnitve na omrežju v normalnih mejah.

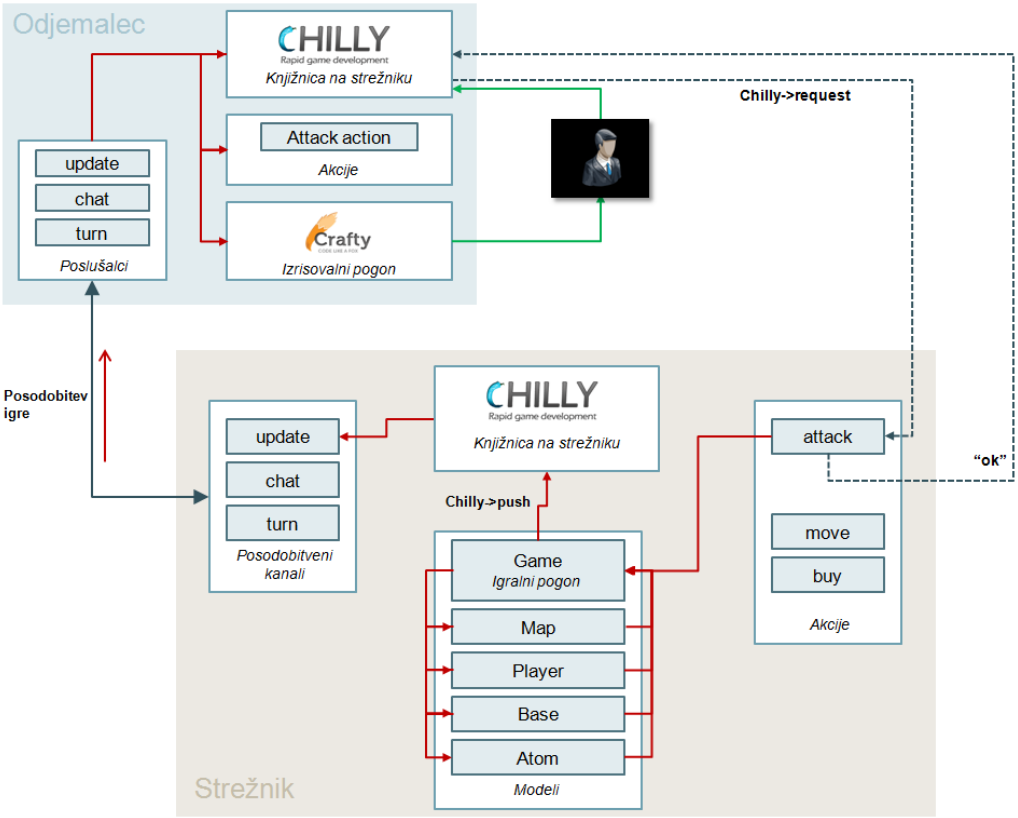


Slika 3: Tehnologija dolgega izpraševanja

2.2. PREDLAGANA ARHITEKTURA

Arhitektura, ki jo predlagamo, je ločena na dva dela, na odjemalca in strežnika. V vsakem delu smo si zamislili knjižnico, ki poenostavi izdelavo večigralske igre. Slika 4**Error! Reference source not found.** prikazuje shemo celotne arhitekture ter povezave posameznih delov in obeh delov. Uporabnik komunicira z odjemalcem, kjer mu poljuben izrisovalni pogon izriše grafični uporabniški vmesnik.

Opišimo komponente s sistema s potekom uporabniške akcije skozi sistem.



Slika 4: Predlagana arhitektura

1. Uporabnik z miško ali na tipkovnici sproži zahtevek na knjižnici Chilly Framework, ki deluje v odjemalcu. Na sliki interakcijo simbolizira zelena puščica proti odjemalcu.
2. Chilly Framework z uporabo asinhronega zahtevka v JavaScriptu, ki je v komponenti *request*, sproži zahtevek na strežniku z akcijo *attack*. Na sliki simbolizira zahtevek modra črtkana puščica proti strežniku.
3. Strežnik se odzove in sporoči, da je bil zahtevek sprejet, kar simbolizira modra črtkana puščica v nasprotni smeri.
4. Akcija *attack* sproži spremembo na igralnem pogonu, ki vzame prejete podatke in spremeni stanje igre. To lahko vključuje komunikacijo z drugimi komponentami, odvisnimi od izvedbe igre, ki teče na strežniku. Interakcijo simbolizirajo rdeče puščice.
5. Ko se v igralnem pogonu spremeni stanje igre, ta izkoristi metodo *push* na Chilly Framework, ki deluje na strežniku, in podatek o vseh prejemnikih posodobitve. Rdeča puščica *push* proti knjižnici Chilly.
6. Chilly Framework pošlje posodobitev po posodobitvenem kanalu update. Rdeča puščica proti posodobitvenim kanalom.

7. Posodobitev se prenese vsem poslušalcem, ki jim je namenjena po posodobitvenem kanalu. Modra puščica v obe smeri med strežnikom in odjemalcem.
8. V odjemalcu poslušalci sprožijo akcijo na Chilly Frameworku, ki so jo opredelili razvijalci igre. Rdeča puščica.
9. Akcija sproži spremembo v izrisovalnem pogonu in uporabniškem vmesniku. Rdeča puščica proti izrisovalnemu pogonu.
10. Uporabniku je vizualno predstavljena sprememba stanja igre.

Slika prikazuje komunikacijski krog le enega izmed igralcev, v tem primeru izvajalca akcije. V realnosti je na strežnik povezanih več odjemalcev, kar si lahko predstavljamo tako, da si predstavljamo več z modro označenih odjemalcev, ki imajo vzpostavljene s strežnikom povezave, prikazane na sliki. Strežnik pošilja posodobitve po posodobitvenih kanalih s tehnologijo dolgega izpraševanja, od odjemalca proti strežniku pa poteka pošiljanje sporočil z uporabo klasičnih zahtevkov.

Chilly Framework vsebuje vse potrebno za vzpostavitev komunikacije v realnem času v obe smeri, in z uporabo metod, ki so na voljo, je mogoče ohranjati igro sinhronizirano med uporabniki.

Skupni imenovalec vseh različnih iger, ki delujejo na platformi Chilly, zajema strukturo obeh aplikacij, ki jo bomo predstavili v nadaljevanju, in obe knjižnici, ki se dopolnjujeta. Igralni pogon je v celoti odvisen od razvijalcev igre in se lahko bistveno razlikuje med različnimi igrami. Razvijalci določajo pogon igre, modele, ki jih bodo uporabljali in akcije, ki jih želijo podpreti.

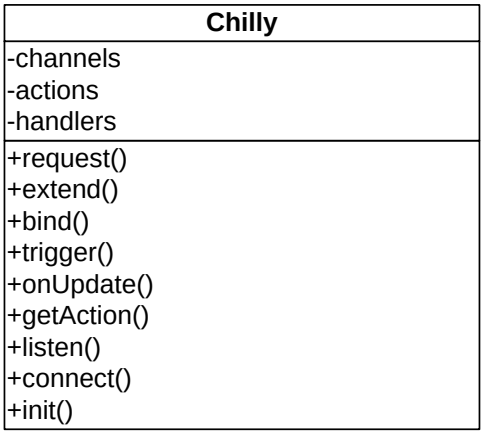
Na pročelju, v odjemalcu, lahko za izvedbo grafičnega vmesnika izberejo poljuben izrisovalni pogon, od obstoječih knjižnic, kot je Crafty Game Engine, do čisto lastne izvedbe. Predvidena arhitektura poskuša čim bolj ločiti abstraktne funkcije sistema, ki so potrebne za dvosmerno komunikacijo, od izvedbe igre, da je arhitektura primerna za čim več primerov uporabe.

V nadaljevanju si podrobneje poglejmo arhitekturo Chilly Frameworka v obeh ločenih enotah arhitekture. Začnimo pri odjemalcu.

2.3. CHILLY FRAMEWORK V ODJEMALCU

Slika 5 prikazuje predvidene atribute in metode objekta Chilly. Z uporabo metode *request* lahko pošljemo zahtevek na strežnik in določimo, kaj naj se zgodi, ko prejmemo odgovor. Chillyjev zahtevek je glavni način komuniciranja s strežnikom od odjemalca proti strežniku. V nasprotni smeri delujejo poslušalci (*angl. listeners*), ki vzpostavijo posodobitveni kanal na strežnik in čakajo na podatke s strežnika. Strežnik pusti odprto povezavo, dokler nima na voljo novih informacij, ki jih želi

posredovati odjemalcu. Privzeto vsa komunikacija poteka po vgrajenem posodobitvenem kanalu z imenom *update*. Z metodo *listen* je mogoče določiti dodatne posodobitvene kanale in povratno funkcijo ali dogodek, ki naj se izvede ob prejemu sporočila po tem kanalu. Za potrebe iger je mogoče določiti tudi dodatne komunikacijske kanale, npr. *chat* in *turnUpdate*. Če igra uporablja tri ločene komunikacijske kanale, se lahko izognemo preveliki preobremenitvi posamičnega kanala. Po dodatnem kanalu *chat* lahko na primer prenašamo sporočila v klepetalnico, *turnUpdate* pa bi prenašal informacije o menjavi krogov. Po vgrajenem posodobitvenem kanalu prenašamo akcije, ki jih igralec izvaja v igri, kot so premiki in kupovanje enot.



Slika 5: Objekt Chilly

Preden Chilly Framework zažene poslušalce, se mora zgoditi proces nalaganja igre, ki smo ga predstavili. Po prenosu statičnih podatkov se požene metoda *init* na Chillyju, ki sproži nalaganje igre.

Potek je povsem odvisen od razvijalca in od tega, kakšne so potrebe igre. Poglejmo si primer integracije Chilly Frameworka z igro:

1. Če igra še ni bila ustvarjena, igra vpraša igralca po uporabniškem imenu in mu poišče partijo.
2. Ko je igra ustvarjena s Chillyjevim zahtevkom, spet zahtevamo stanje igre in z dobljenimi podatki sprožimo dogodek *load*. Na tem mestu se lahko ustvarijo objekt igralca ter scena in ozadje igre, določi se kamera in izriše igralno polje.
3. Po končanem izrisu pokličemo funkcijo, ki na igralno polje doda in izriše vse enote.
4. Na Chilly Frameworku zaženemo vse poslušalce, ki smo jih določili, da na posodobitvenih kanalih poslušajo sporočila o posodobitvah igre.

Vsakič, ko je posodobitev prejeta, Chilly Framework sproži funkcijo, ki jo je razvijalec vezal na ta tip posodobitve in je opredeljena v akcijah. Poslušalec se nato ponovno poveže na strežnik in čaka na nadaljnje posodobitve. Tako poskrbi, da sta odjemalec in strežnik vedno časovno usklajena.

Slika 6 prikazuje predlagano datotečno strukturo aplikacije v odjemalcu.

Slika 6: Datotečna struktura pročelja

Igre, zgrajene z uporabo platforme Chilly Framework, so na pročelju sestavljene iz datoteke HTML, ki določa strukturo uporabniškega vmesnika in datotek css, ki opredeljujejo njegov videz. V mapi *"snd"* so zvočne datoteke.

Programska koda je v mapi JavaScript *"js"*, kjer je razdeljena na več datotek. V mapi *"components"* so shranjene samostojne komponente, ki so del izrisovalnega pogona. To je lahko na primer Crafty Game Engine. V mapi *"config"* je datoteka z nastavitvami.

Tabela 1 prikazuje opis kode v JavaScript datotekah.

Tabela 1: JavaScript datoteke pročelja

Datoteka	
Actions.js	*Določa vse akcije, ki jih uporabnik lahko izvede, kot so "premakni tank", "kupi tank", "izvedi napad".*
Events.js	*Koda, ki se izvede ob določenem dogodku, kot je na primer menjava kroga ali nalaganje igre.*
Listen.js	*Opredeljeni so vsi dodatni komunikacijski kanali, ki jih igra uporablja za komunikacijo s strežnikom.*
Sounds.js	*Razširi Chilly Framework s podporo za zvok z uporabo knjižnice SoundManager.*
Sprites.js	*Definicije vseh sličice (sprites), ki jih igra uporablja.*
Ui.js	*Definicije vseh akcij, ki se odražajo na spremembi uporabniškega vmesnika z uporabo knjižnice jQuery.*

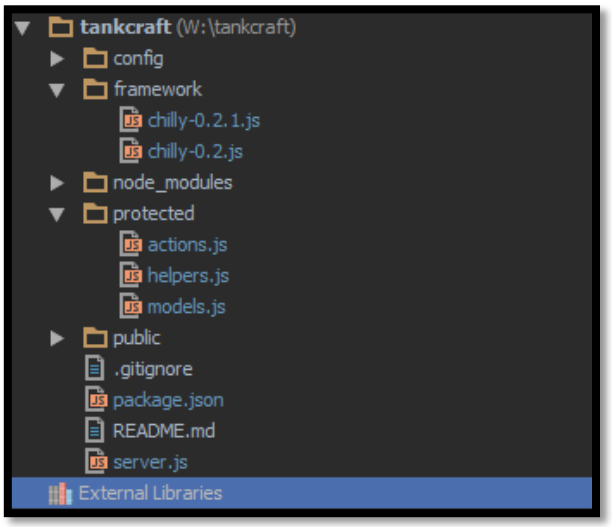
2.4. CHILLY FRAMEWORK NA STREŽNIKU

Osnova za strežnik, ki ga predvideva arhitektura, je node.js, za izvedbo pa smo uporabili knjižnico Express, ki jo je mogoče namestiti z upravljalnikom modulov NPM. Express je strežnik HTTP, zgrajen nad arhitekturo node.js in vmesnim programjem Connect, ki ga je prav tako mogoče namestiti z NPM-jem. Naloga Expressa v naši arhitekturi je, da servira HTML in druge statične datoteke odjemalcem.

Za dvosmerno komunikacijo in sinhronizacijo med uporabniki smo razvili svoj upravljalnik dogodkov, ki je del Chilly Frameworka. V Express ga vključimo po poti *'/action'*. Vsi zahtevki, poslani na to pot, se posredujejo Chilly Frameworku. Chilly Framework podpira dva različna tipa zahtevkov. Prvi prihajajo od komponente Chilly Request v odjemalcu in na katere se nemudoma odzove. Drugi simulirajo komunikacijo od strežnika do odjemalca in jih Chilly Framework dodaja v seznam odprtih povezav, ki so združene po odjemalcih. Tu govorimo o izvedbi posodobitvenih kanalov.

Prijava v sistem, iskalnik partije in zagon nove igre so izvedeni kot akcije, ki jih strežnik lahko izvede po toku, ki smo ga določili v shemi sistema. Vsi aktivni primerki igre *(angl. instance)* se hranijo v spominu v okviru objekta Chilly.

Poglejmo si datotečno strukturo na strežniku, ki jo prikazuje Slika 7, in namen posameznih datotek. V mapi *config* je konfiguracija strežnika, mapa *node_modules* je del NPM-ja, kamor ta shranjuje nameščene module in kjer prebiva Express, v mapi *protected* so datoteke potrebne za izvajanje igre na strežniku, v mapi *public* pa so vsi dokumenti, katerih strukturo smo opisali v poglavju o strukturi na strani odjemalca.



Slika 7: Datotečna struktura zaledja

Poglejmo si strukturo kode na strežniku v različnih datotekah. Tabela 2 prikazuje osnovne datoteke JavaScript, ki so del Chilly Frameworka in njihovo vlogo.

Tabela 2: JavaScript datoteke zaledja

Datoteka	
Chilly-0.2.1.js	*Strežniški del knjižnice Chilly Framework, ki omogoča dvosmerno komunikacijo med odjemalci in strežnikom. Z njim je mogoče definirati tudi odgovore na zahtevke Chilly Request komponente, ustvarjene v odjemalcu.*
Server.js	*Vstopna točka aplikacije. Strežnik se zažene z ukazom "node server.js".*
Actions.js	*Definira vse akcije, ki jih lahko strežnik izvede na zahtevke odjemalca, ki jih v njem ustvari Chilly Request in so definirani v istoimenski datoteki.*
Helpers.js	*Pomožne funkcije.*
Model.js	*Izvedba objektov na Sliki 14 (Game, Player, Base, Map, Atom).*

Express zahteva zgolj minimalno konfiguracijo, da začne servirati datoteke, za naše potrebe pa moramo vklopiti še nekaj dodatnih funkcij. Vsi zahtevki, ki niso zahtevki po statičnih virih, bodo poslani na Chilly Framework z uporabo lastnega upravljalnika zahtevkov. Da lahko ugotovimo, od katerega uporabnika prihaja zahtevek, se morajo podatki ohraniti med posameznimi zahtevki. Zato uporabljamo seje, ki jih shranjujemo v podatkovno bazo Redis. Slika 8 prikazuje kodo vstopne točke v aplikacijo. Iz nje je razvidna tudi konfiguracija Expressa:

- *bodyParser* omogoča, da so v vseh zahtevki na voljo podatki, ki jih je klient poslal na strežnik bodisi s *POST* bodisi z *GET* zahtevkom;
- *methodOverride* omogoča, da simuliramo tudi zahtevke *DELETE* in *PUT;*
- *query* omogoča razčlenitev povpraševalnega niza;
- *cookieParser* omogoča razčlenitev piškotkov;
- *exSession* je podpora bazi podatkov Redis za shranjevanje seje;
- *responseTime* doda v glavo odgovora odzivni čas v milisekundah;
- brskalniku *errorHandler* vrne napako;
- *router* doda podporo za poti.

```
48  /**
49   * Configure express
50   */
51  app.configure(function() {
52      app.use(express.bodyParser());
53      app.use(express.methodOverride());
54      app.use(express.query());
55      app.use(cookieParser);
56      app.use(exSession);
57      app.use('/action', Chilly.requestDispatcher);
58      app.use('/', express.static(__dirname + '/public', { maxAge: CONF.core.httpStaticCache /* 14 days */ }));
59      app.use(express.responseTime());
60      app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
61      app.use(app.router);
62
63      // Catch-all error handler.
64      app.use(function(err, req, res, next){
65          console.error(err.stack);
66          res.send(500, 'An unexpected error occurred! Please check logs.');
67      });
68  });
69
70  /**
71   * Start server
72   */
73  server.listen(CONF.core.port);
```

Slika 8: Server.js – konfiguracija aplikacije Node.js in strežnika

V tej konfiguraciji Expressa vsa komunikacija med strežnikom in odjemalcem, razen statičnih datotek, poteka prek knjižnice Chilly Framework. Poglejmo si, katere metode lahko uporabimo v Chilly Frameworku za izvedbo poljubne igre. Prikazuje jih Tabela 3.

Tabela 3: Osnovne metode knjižnice Chilly Framework na strežniku

Chilly Framework / Backend	
extend()	*Z metodo razširimo Chilly Framework z novo funkcionalnostjo.*
bind()	*Veži dogodek na Chilly Framework.*
trigger()	*Sproži enega izmed določenih dogodkov.*
request()	*Interna metoda za ustvarjanje Chilly Requesta, s katerim lahko upravljamo v Chilly Frameworku. Ima dostop do podatkov iz zahtevka in uporabnikove seje.*
push()	*Z metodo push potisnemo sporočilo več odjemalcem, ki jih podamo kot vhodni podatek metodi. Treba je opredeliti spisek prejemnikov, kanal, po katerem želimo sporočilo poslati in poljubne podatke.*
getMessage()	*Pobere prvo sporočilo iz uporabnikove vrste sporočil in ga odstrani iz vrste.*
sendMessage()	*Potisne sporočilo uporabniku.*
addClient()	*Doda uporabnika na seznam uporabnikov.*
removeClient()	*Odstrani uporabnika s seznama uporabnikov.*
garbageCollector()	*Odstranjuje nedejavne uporabnike in povezave.*
addConnection()	*Zamrzne povezavo in jo doda na določen kanal, kjer čaka na podatke, ki jih želimo posredovati proti uporabniku.*
removeConnection()	*Odstrani zamrznjeno povezavo iz kanala.*
getConnection()	*Vrne povezavo za podanega uporabnika in kanal.*
createChannel()	*Ustvari kanal po katerem lahko potisnemo podatke proti uporabniku. Vsaka povezava na kanal bo zamrznjena, dokler ni na voljo podatkov, ki jih želimo poslati.*
action()	*Opredeljuje akcijo, ki odgovarja na zahtevke, ki jih pošljemo iz pročelja z uporabo metode Chilly.request(). Opredeliti je mogoče različno vedenje glede na to, ali je uporabnik prijavljen ali je anonimen.*
requestDispatcher()	*Povezava s strežnikom Express, ki prestreže zahtevke, namenjene Chilly Framework,u in sproži eno izmed določenih akcij.*
generateGameId()	*Ustvari izviren niz znakov za ID igre.*
generateClientId()	*Ustvari izviren niz znakov za ID igralca.*
generateUniqueId()	*Ustvari izviren niz znakov.*
createGame()	*Ustvari nov primerek igre in ga doda na seznam aktivnih iger.*
getGame()	*Vrne zahtevan primerek igre.*

Chilly Framework ima lahko na strežniku določenih tudi več posodobitvenih kanalov, po katerih z metodo *push* pošiljamo spremembe in sinhroniziramo stanje igre proti uporabniku.

Na platformi Chilly Framework, ki vključuje knjižnico v odjemalcu, knjižnico na strežniku, osnovno konfiguracijo strežnika in predlagano strukturo aplikacije, lahko začnemo izdelavo iger. V nadaljevanju je opisana igra, ki smo jo izdelali za potrebe testiranja.

3. ARHITEKTURA IGRE SHELL WARS

V začetni dobi računalništva so bile prve igre zelo primitivne in v celoti narejene hardversko. Razvoj mikroprocesorjev je hitrost razvoja zelo spremenil in danes se igre igrajo na zmogljivih osebnih računalnikih in namenskih igralnih konzolah. Čeprav je od prvih iger minilo že 50 let, je v primerjavi z drugimi industrija iger še vedno mlada in se hitro spreminja. Po obsegu se že lahko primerja s filmsko, saj je bil trg iger leta 2012 ocenjen na 67 milijard dolarjev, napovedi pa kažejo, da se bo ta številka povečala na 82 milijard dolarjev v letu 2017 (Gaudiosi, 2012).

Igre so se danes razvile v veliko različnih žanrov, od iger s kartami do množičnih večigralskih iger igranja vlog, kot je World of Warcraft, ki je imel v letu 2010 vrh pri 12 milijonih naročnikov (Harper, 2013). Dandanes niso omejene zgolj na računalnike in konzole, saj jih lahko igrajo praktično na vseh napravah z mikročipom, vse od TV-jev do telefonov in tablic.

Ogromna računska moč danes razvijalcem omogoča razvoj iger z izjemno grafiko, s prostranimi področji, po katerih se lahko igralci prosto gibljejo, in z zapleteno umetno inteligenco. Vendar hkrati s strojnimi zmogljivostmi naraščata tudi zapletenost razvoja iger in obseg. Če je bilo lahko v osemdesetih in na začetku devetdesetih let v ekipah zgolj nekaj ljudi, je danes razvoj večjega naslova za več platform nemogoč s tako majhno ekipo. Kot primer omenimo igro Assasin's Creed 4, ki je trenutno v razvoju za novo generacijo konzol, PlayStation 4 in Xbox One, njena ekipa pa šteje kar tisoč ljudi v sedmih samostojnih studiih (Ta, 2013).

Po drugi strani se je v zadnjih letih začela ob boku z velikimi naslovi na hitro razvijati indie scena. Najbolj prepoznavna predstavnica te vrste iger je Minecraft. Z razvojem je začel Markus Persson z bolj znanim vzdevkom Notch, kljub izjemnemu uspehu z 12 milijoni prodanih izvodov igre pa ekipa ni presegla nekaj ljudi. Tudi igre, kot so Super Meat Boy, Braid, World of Goo in druge, so dokazale, da je mogoče izdelati vrhunsko igro tudi z majhno ekipo (Sliwinski, 2012).

S tako različnimi pristopi k razvoju iger in različnimi žanri, ki se v načinu igranja bistveno razlikujejo, je nemogoče najti eno samo idealno programsko arhitekturo, ki pokriva vse primere uporabe. Igralni pogon je skupek ponovno uporabnih komponent, arhitektura je odsev okolja, v katerem igre razvijamo, potreb konkretne igre in splošnih principov, ki se uporabljajo v vseh okoljih.

Da smo lahko preverili, ali predvidena arhitektura deluje, smo opredelili igro, ki smo jo želeli izdelati. Za cilj smo si zastavili izdelavo večigralske igre po vzoru serije Panzer General, vendar s pomembnimi razlikami. V bistvu gre za izmenično potezno strategijo za več igralcev. Pomembno pa je, da se dogaja v realnem času.

Večigralske igre lahko delujejo v različnih okoljih: kot samostojne aplikacije, mobilne aplikacije, igre *flash* ali igre v brskalniku. Ker je ciljno okolje prototipne

igre brskalnik, jo je mogoče igrati na večini modernih naprav: od računalnika do telefonov in tablic. S tem smo se omejili na konkreten set tehnologij in metod, ki smo jih uporabljali pri izdelavi igre.

Preden smo se lotili izdelave, je bilo treba za razumevanje natančno opredeliti igro, ki smo jo želeli izdelati, in iz tega izpeljati tehnične zahteve (glej prilogo 1) in oblikovanje sistema na vrhu programske platforme Chilly Framework.

3.1. OPIS IGRE SHELL WARS

Igra je potezna strategija, v kateri se lahko pomerijo do štirje igralci, vsak v svoji barvi. Ko se igra začne, se naključno določi igralec, ki bo prvi na potezi. Vsak igralec začne s svojo vojaško bazo v enem izmed kotov igralnega polja. Na voljo ima določeno količino denarja, s katerim kupuje enote, denar pa se v vsakem krogu poveča za fiksni znesek. Cilj igre je podreti vojaška oporišča nasprotnih igralcev in ubraniti svoje pred sovražniki. Igralec, ki edini obdrži svoje oporišče, je zmagovalec.

Igra poteka na igralnem polju, sestavljenem iz 30 x 30 šestkotnikov, po katerem se lahko enote premikajo v vse smeri. Igralno polje je sestavljeno iz prehodnega travnatega terena in ovir v obliki gozdov, skalnatega območja, vode in ruševin.

Da bi lahko dosegli cilj igre, igralci kupujejo tanke, s katerimi napadajo sovražne tanke in oporišča.

Tank ima več lastnosti. Cena določa količino denarja, ki se igralcu odšteje, ko ga kupi. Napad označuje razpon, znotraj katerega se naključno določi obseg škode, ki jo povzroči ob napadu in ki se odšteje od življenskih točk nasprotne tarče. Oklep določa, za koliko se zmanjša vrednost prejetega napada. Doseg premika označuje, koliko polj se lahko tank premakne v eni potezi. Doseg napada označuje največjo razdaljo med tankom in tarčo napada. AP označuje število akcijskih točk, ki so na voljo v vsakem krogu.

Vsak tank ima v enem krogu omejeno število poteznih točk. Vsaka poteza jih zahteva določeno število. Premik zahteva eno potezno točko, napad dve, pri čemer se lahko tank v eni potezi premakne za toliko, kolikor znaša njegov doseg, npr. 10 polj.

Ko igralcu zmanjka poteznih točk ali preteče 60 sekund, je na vrsti naslednji igralec. Na začetku vsakega kroga dobi igralec, ki je na potezi, določeno količino denarja, s katerim lahko kupuje nove tanke.

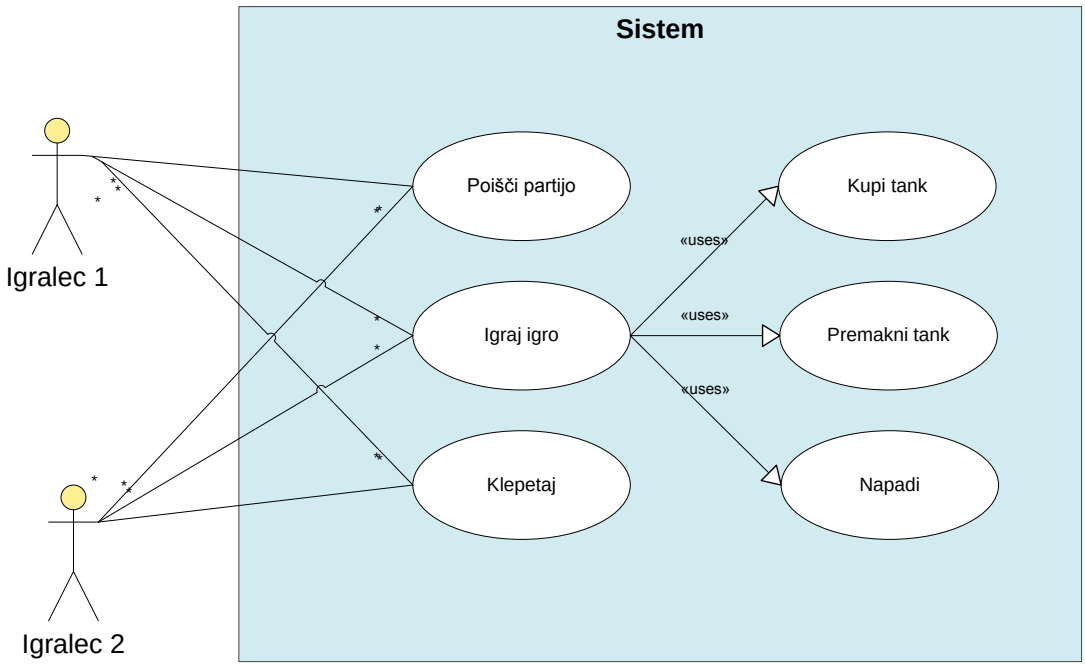
Za uspeh je treba pametno razpolagati s sredstvi, ki so na voljo, in taktično premikati enote. Do pomembnega preobrata v igri lahko pripeljejo tudi zavezništva, ki pa se lahko hitro obrnejo, ko ostane katero izmed oporišč nezavarovano.

Igralci lahko med seboj komunicirajo prek vmesnika v realnem času, med izdajanjem ukaza v vmesniku in njegovo izvedbo pa za igralca ni opaznega zamika.

3.2. INTERAKCIJA UPORABNIKOV S SISTEMOM (IGRO)

Igro smo začeli načrtovati pri primerih uporabe, tako da smo določili procese, ki v njej potekajo. Poglejmo si, kako lahko igralci komunicirajo s sistemom, v našem primeru z igro Shell Wars. Stik s sistemom imajo samo z grafičnim vmesnikom igre, igralni pogon na strežniku pa je skrit pred uporabnikom. Iz opisa igralnosti lahko izločimo naslednje procese in dele sistema, s katerimi komunicirajo uporabniki.

Igra podpira več igralcev. Slika 9 prikazuje dva igralca, ki igrata igro.



Slika 9: UML-primer uporabe sistema v večigralskem načinu

Sistem podpira tri glavne procese. Igralec 1 in 2 poiščeta partijo, kar vključuje združevanje igralcev od dva do štirih igralcev. Sistem jih doda v igro in jo zažene. Igralci potem igrajo igro.

Med igranje so igralcem na voljo trije procesi, s katerimi upravljajo igro in poskušajo doseči pogoje za zmago:

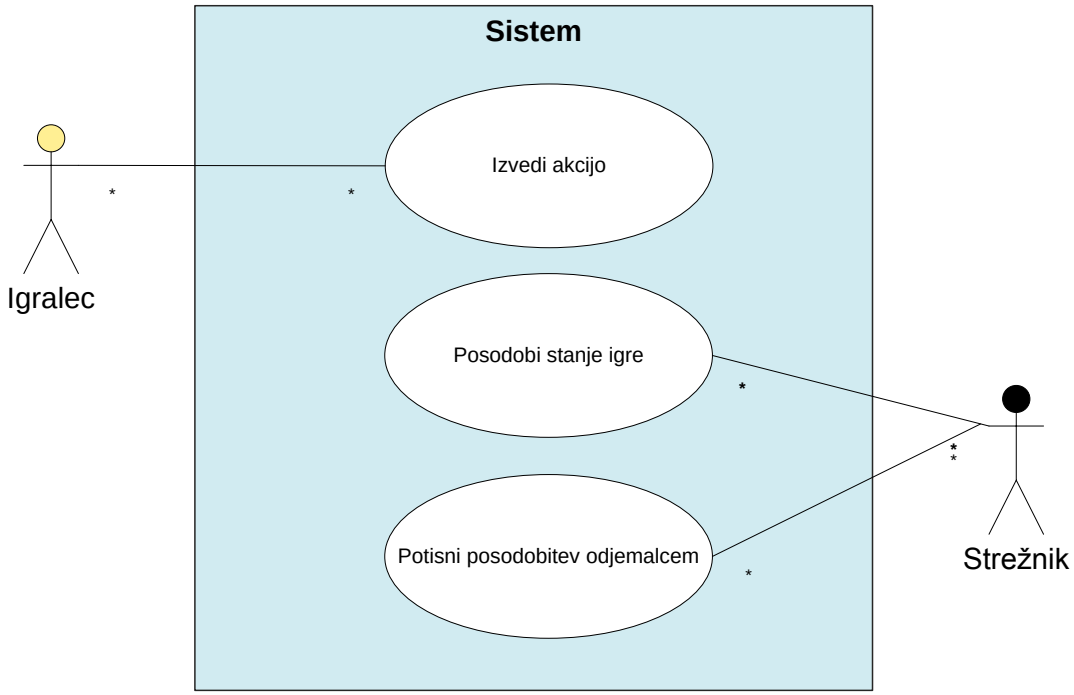
- Pri kupovanju tankov igralci v igri z igralnim denarjem kupijo novo enoto, ki se izriše na zaslonu.

- Tanke lahko igralci premikajo po igralnem polju v okviru omejitev, ki jih sistem določi glede na konfiguracijo igre.
- S tanki lahko igralci napadajo tanke in oporišča nasprotnikov, da bi jih uničili in si tako zagotovili zmago.

Tretji način, s katerim uporabniki komunicirajo s sistemom, je prek vmesnika za klepet. Sporočila lahko pošiljajo v realnem času soigralcem, ki so v isti instanci igre.

Akcija, ki jo igralec med igranjem izvede, se mora, če je veljavna in če je igralec na potezi, prikazati trenutnemu igralcu in posredovati vsem drugim udeležencem igre s čim manjšim zamikom.

Da zadostimo zahtevi po dogajanju v realnem času, je treba akcije časovno uskladiti med odjemalci. Slika 10**Error! Reference source not found.** prikazuje, kako ta interakcija poteka.



Slika 10: UML-primer uporabe posodobitve in sinhronizacije stanja igre

Na Sliki 10 sta dva akterja, igralec in strežnik. Igralec 1 izvede akcijo, kot je premik enote po igralnem polju, in jo pošlje v sistem. Naloga strežnika je, da posodobi stanje igre, če je bila akcija dovoljena, ali pa jo zavrne. Vsakič, ko se stanje igre spremeni, je treba vse spremembe sporočiti odjemalcem. Izogniti se želimo situaciji, ko bi bilo na odjemalcu pri različnih igralcih v isti instanci igre na zaslonu drugačno stanje igre. Strežnik mora imeti zato na voljo možnost potiska posodobitev vsem odjemalcem v trenutni instanci igre. V našem primeru je šlo za spremembo položaja enot na igralnem polju, ki se zdaj pokaže tudi pri drugih igralcih.

Kot smo omenili v uvodu, smo se odločili, da del arhitekture, ki deluje neodvisno od igre, ki se na njej izvaja, ločimo. To je del, ki na odjemalcu skrbi za komunikacijo med strežnikom in sprejema posodobitve, in del, ki na strežniku skrbi za časovno usklajenost med odjemalci.

Za potrebe prototipne igre smo razširili Chilly Framework z dodatnimi zmožnostmi, ki so specifične za igro. Poleg standardnega nabora zmožnosti skrbi za uporabnike in zna ustvarjati nove instance igre. Da lahko bolje razumemo oba dela arhitekture, si poglejmo prikaz poteka igre z vidika pročelja *(angl. front-end)* in *zaledja (angl. back-end)*. Slika 11 prikazuje potek z vidika odjemalca. Ko govorimo o odjemalcu, imamo v mislih logiko, ki se izvaja v brskalniku igralca. Ko govorimo o strežniku, imamo v mislih igralni pogon in spremljevalno logiko, ki se izvaja na strežniku. Slika 12 prikazuje potek igre z vidika strežnika.

Poglejmo si najprej diagram poteka z vidika brskalnika.

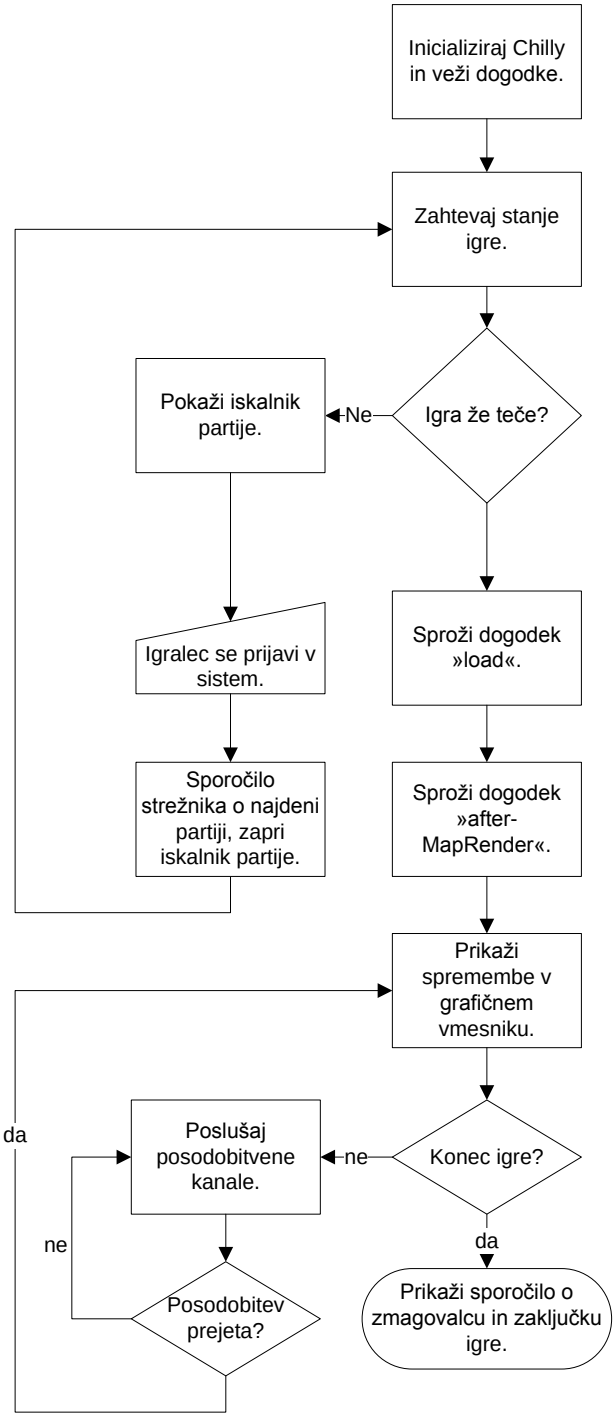
3.3. POTEK IGRE Z VIDIKA ODJEMALCA

Uporabnik najprej obišče URL, na katerem se igra nahaja. V brskalnik se prenesejo vse statične datoteke, kot so HTML, slike in zvok, nato se izvede koda v JavaScriptu, ki inicializira Chilly Framework. Ta poišče in poveže vse definirane funkcije, ki se morajo izvesti ob dogodkih. Nato pošlje na strežnik zahtevek po trenutnem stanju igre. Strežnik sporoči, da uporabnik še ni v nobeni igri. Uporabniški vmesnik izriše okno za iskanje. Uporabnik vpiše v okno svoje uporabniško ime, da lahko strežnik začne z iskanjem partije.

Ko se v igro prijavi vsaj en igralec, strežnik sporoči odjemalcu, da je iskanje partije končano. Odjemalec ponovno zahteva stanje igre in strežnik odgovori z vsemi potrebnimi podatki za izris uporabniškega vmesnika, igralnega polja in začetek igre. Z dobljenimi podatki se sproži dogodek *load*, ki ustvari objekt trenutnega igralca in začne izrisovati podatke na zaslon. Naložita se igralno polje in uporabniški vmesnik. Po nalaganju igralnega polja se sproži dogodek *afterMapRender*, ki prikaže spremembe v grafičnem vmesniku. Igra vzpostavi v odjemalcu povezavo na strežnik z vsemi opredeljenimi posodobitvenimi kanali. To so kanali od strežnika proti odjemalcu, po katerih prejema odjemalec posodobitve igre in sporočila. Ko odjemalec od strežnika prejme sporočilo o posodobitvi in spremljajoče podatke, lahko v grafičnem vmesniku prikaže spremembe. Proces poteka, dokler niso izpolnjeni pogoji za zmago. Zadnje sporočilo o koncu igre prekine proces poslušanja za posodobitvami s strežnika in igralcu prikaže sporočilo o koncu igre in zmagovalcu.

Poleg opisanega procesa je na strani odjemalca na voljo še funkcija za pošiljanje podatkov na strežnik, ki ni del diagrama poteka v brskalniku, ampak je opisana z vidika strežnika. Z vidika pročelja igre omenimo le, da se ob izvajanju akcij v vmesniku te pošljejo na strežnik, ki akcijo sinhronizira med preostalimi odjemalci, ki so v trenutni igri. Akcije drugih igralcev se prenašajo po posodobitvenih kanalih, kot je razvidno iz diagrama.

Na naslednji strani si poglejmo diagram poteka akcije, ki se dogaja na strežniku v odziv na zahtevke, ki prihajajo od odjemalcev. Diagram predstavlja del Chilly Frameworka, ki skrbi za ustvarjanje novih iger, upravljanje z igralci, iskanje partij in upravlja s igralnimi pogoni v vseh instancah iger.



Slika 11: Diagram poteka v brskalniku

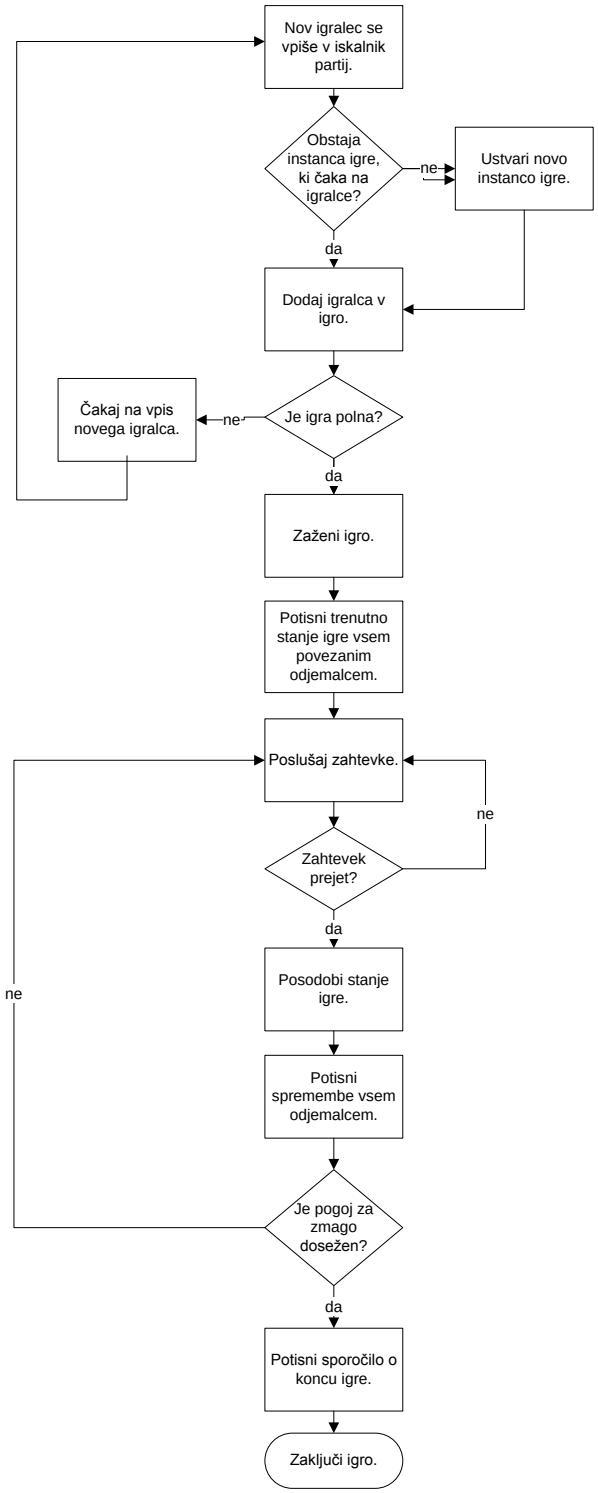
3.4. POTEK IGRE Z VIDIKA STREŽNIKA

Slika 12 na naslednji strani prikazuje potek igre z vidika strežnika.

Strežnik, ki ga ustvarimo s pomočjo Chilly Frameworka, je strežnik HTTP, prek katerega se pred začetkom igre v brskalnik pretočijo vsi potrebni viri. Čeprav odjemalec najprej zahteva stanje igre, lahko ta korak preskočimo in začnemo pri vpisu novega igralca v iskalnik partij.

Ko se na strežnik prijavi nov uporabnik, sistem najprej preveri, ali že obstaja prosta instanca igre, ki čaka na igralce. Če take instance ne najde, ustvari novo in doda igralca v igro. Nato preveri, ali je igra že zapolnjena. Če je, igro zažene, v nasprotnem primeru pa čaka na vpis novega igralca.

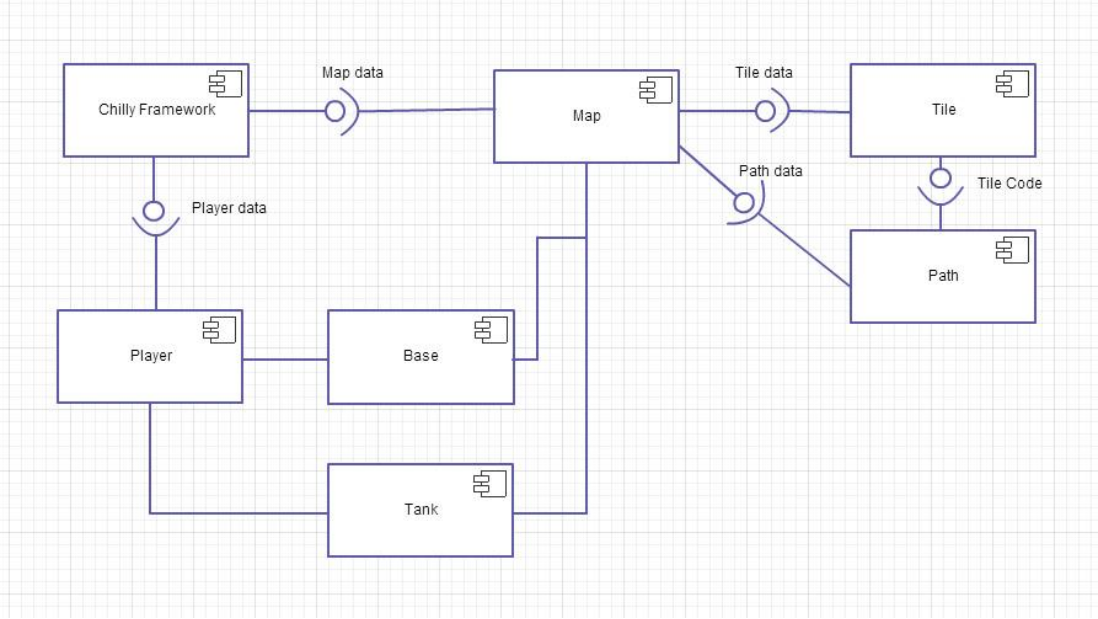
Po zagonu igre potisne stanje igre vsem povezanim odjemalcem in začne poslušati zahtevke, ki prihajajo od odjemalcev. Ko igra prejme zahtevek, na primer za premik enote, posodobi stanje igre. Po vsaki posodobitvi se preverja, ali je pogoj za zmago dosežen. Če je, igra vsem odjemalcem potisne sporočilo o koncu igre in o zmagovalcu ter zapre instanco igre.



Slika 12: Diagram poteka na strežniku

3.5. GRAFIČNI UPORABNIŠKI VMESNIK

Za izdelavo grafičnega uporabniškega vmesnika smo določili komponente, ki ga bodo sestavljale. Glavna komponenta je knjižnica Chilly Framework. Slika 13 prikazuje dodatne komponente, ki izhajajo iz tehničnih zahtev, in povezave med njimi.



Slika 13: Komponentni diagram strukture grafičnega vmesnika igre

Kot smo že omenili, z uporabo Chilly Frameworka poskrbimo za dvosmerno komunikacijo s strežnikom in sinhronizacijo stanja igre. Grafični vmesnik vsebuje komponento *Player*, ki predstavlja igralca, vpisanega v igro. Igralec ima komponenti *Base* in *Tank*, ki se izrisujejo na zaslonu preko komponente *Map*, ki z njimi upravlja.

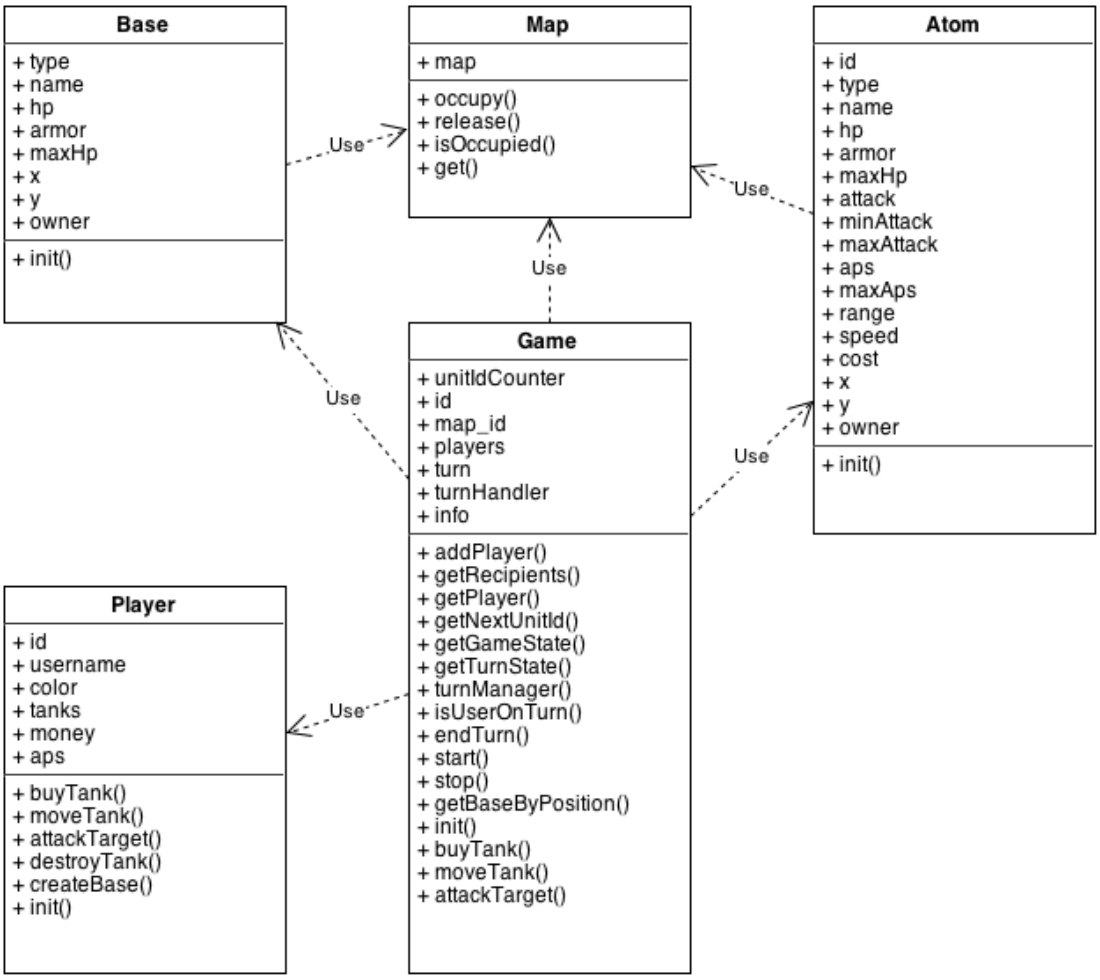
Komponenta *Map* izrisuje igralno polje in upravlja z vsemi grafičnimi elementi, ki so na njem. Sestavljena je iz več entitet, ki podedujejo svoje ravnanje od komponente *Tile*.

Komponenta *Path* je komponenta za izrisovanje poti po šestkotnem igralnem polju. Podatke dobiva od komponente *Map*, ki s pomočjo A* algoritma računa najkrajšo možno pot med poljem A in poljem B.

Vse komponente skupaj predstavljajo grafični pogon igre, ki se izriše uporabniku in predstavlja igro, ki jo igralec vidi in s katero komunicira z uporabo miške in tipkovnice.

3.6. STATIČNA STRUKTURA POGONA IGRE

Igra potrebuje za delovanje pogon, ki se lahko izvaja tako na odjemalcu kot tudi na strežniku. Za potrebe večigralske igre arhitektura predvideva, da se pogon igre izvaja na strežniku. Slika 14 prikazuje objekte, ki so del igralnega pogona.



Slika 14: Specifikacija objektov, potrebnih za delovanje sistema

Opredelili smo pet objektov, ki jih igra za delovanje potrebuje. Glavni objekt je *Game*, ki predstavlja instanco igre. Vsebuje podatke, ki so potrebni za izvedbo posamezne igre, kot so število enot v igri, povezava na karto, vse igralce, kdo je na potezi in upravljalnik potez. Tabela 4 prikazuje metode pogona igre.

Tabela 4: Objekt Game

Game Object	
addPlayer()	*Doda igralca v igro.*
getRecipients()	*Vrne ID-je vseh odjemalcev.*
getPlayer()	*Vrne zahtevanega igralca.*
getNextUnitId()	*Vrne naslednji prost ID enote.*
getGameState()	*Vrne vse potrebne podatke, ki jih odjemalec potrebuje za izris grafičnega vmesnika s trenutnim stanjem igre.*
turnManager()	*Definira in zažene upravljalnik krogov.*
endTurn()	*Predčasno konča krog.*
start()	*Zažene igro.*
stop()	*Ustavi igro.*
getBaseByPosition()	*Vrne vojaško oporišče glede na podano lokacijo.*
init()	*Inicializira igro.*

Igra si shrani podatke o igralcih, ki so določeni v objektu *Player*. Podatki so ID, uporabniško ime, barva, tanki in denar, s katerim razpolagajo. Tabela 5 prikazuje metode objekta Player.

Tabela 5: Objekt Player

Player Object	
buyTank()	*Kupi tank, odšteje igralcu denar in doda tank na igralno polje.*
moveTank()	*Premakne tank na novo lokacijo.*
attackTarget()	*Napade podano tarčo.*
createBase()	*Ustvari vojaško oporišče na začetnem položaju.*
init()	*Inicializira igralca.*

Igra mora shranjevati tudi stanje igralne karte in lokacije vseh enot. To določi objekt *Map*. Tabela 6 prikazuje metode objekta Map.

Tabela 6: Objekt Map

Map Object	
occupy()	*Označi podano polje kot zasedeno.*
release()	*Sprosti podano polje.*
isOccupied()	*Preveri, ali je podano polje zasedeno.*
get()	*Vrne enoto na podanem polju.*

Pogon igre vsebuje še dva objekta, ki imata zgolj *init* metode in hranita podatke, ki jih prikazuje Slika 14.

Chilly Framework upravlja z objektom Game, ki je vstopna točka do instance igre. Na druge objekte je mogoče vplivati posredno z metodami, ki jih odkriva objekt Game.

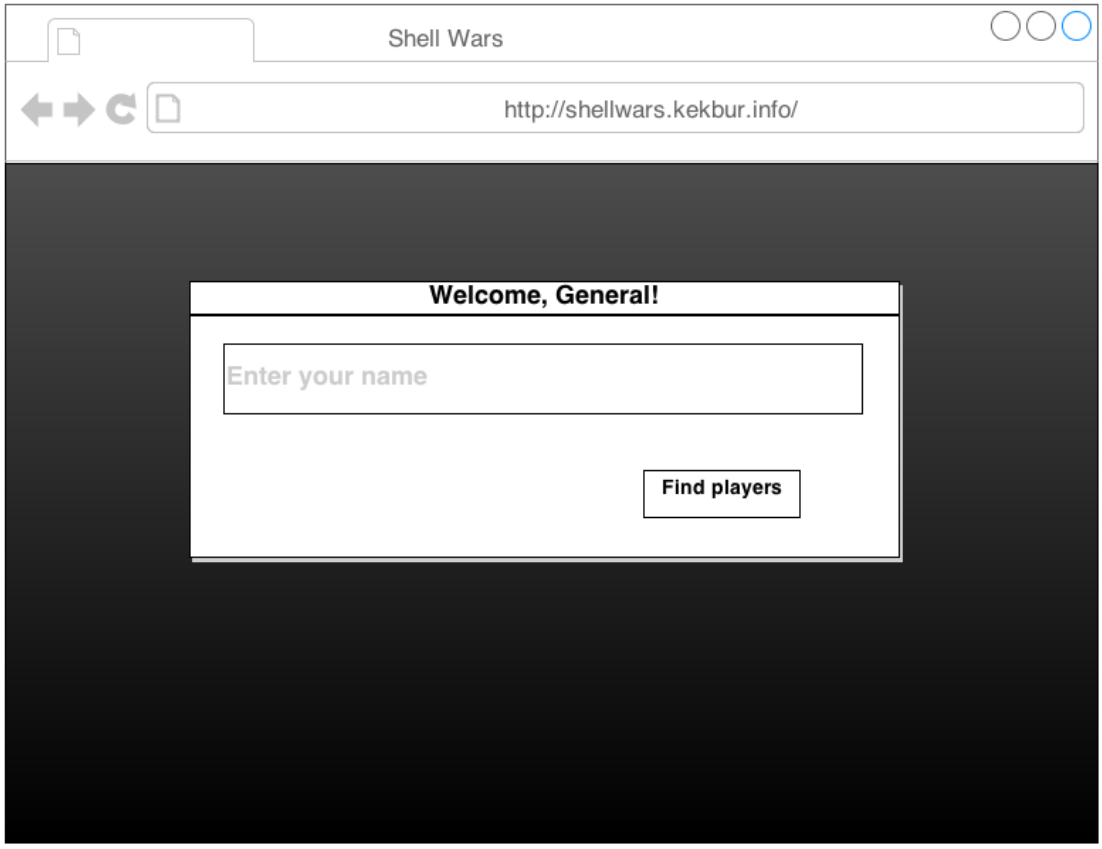
3.7. NAČRT UPORABNIŠKEGA VMESNIKA

Načini oblikovanja uporabniških vmesnikov zagotavljajo, da je najpomembnejša naloga vsakega vmesnika ta, da je jasen za uporabo. Da ga lahko uporabniki učinkovito uporabljajo, morajo prepoznati njegov namen, prepoznati, zakaj bi ga uporabili, in predvideti, kaj se bo zgodilo, ko ga uporabljajo. Prostora za zmedo ni, saj jasnost vmesnika v ljudeh vzbuja samozavest in vodi k nadaljnji uporabi sistema (Porter, 2013).

Pri načrtovanju našega vmesnika smo se osredotočili na preprostost uporabe. Vmesnik zato vsebuje zgolj nujno potrebne elemente, da izpolni tehnične zahteve, ki smo jih določili. Treba je upoštevati, da bo prikazan na različnih napravah, ki imajo različno velike zaslone.

Igra vsebuje samo dva različna zaslona. Prvi je iskalnik partije, drugi je glavni zaslon igre in se prikaže, ko igra poteka. Vse nadaljnje dogajanje po uspešne iskanju partije poteka na glavnem zaslonu.

3.7.1. ISKALNIK PARTIJE

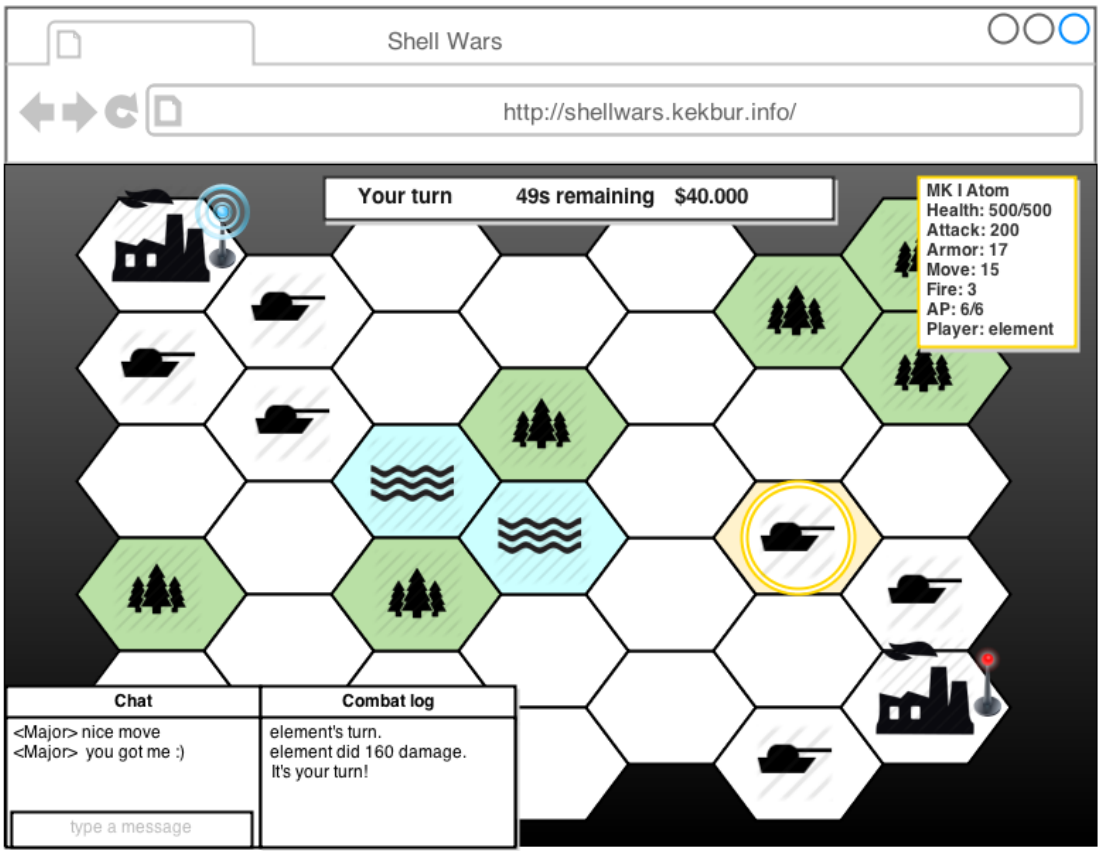


Slika 15: Vmesnik za iskanje partije

Preden se igra lahko začne, mora igralec poiskati svoje nasprotnike. Slika 15 prikazuje podobo vmesnika za iskanje partije, ki pokriva tehnično zahtevo ***REQ-1.12***. Vmesnik igralcu izriše okno za vpis uporabniškega imena, s katerim se bo predstavljal v igri. Igralec klikne na gumb *Find players*. Prikaže se sporočilo, da igra išče soigralce.

Ko pride igralec številka dve in izvede prijavo v iskalnik partije, igra ustvari novo instanco igre. Igralca doda v igro in naključno določi tistega, ki bo prvi na potezi. Nato zažene igro. Vmesnik za iskanje partije se zapre in začne se izris igralnega polja in HUD-a.

3.7.2. GLAVNI ZASLON IGRE



Slika 16: Grafični uporabniški vmesnik igre

Na Sliki 16 je prikazan grafični uporabniški vmesnik igre, ki izhaja iz tehničnih zahtev ***REQ-1.1, REQ-1.2, REQ-1.3, REQ-1.9***. Na njem so izrisane tri ločene plasti. Ozadje, igralno polje z vojaškimi enotami in HUD (*heads-up display*), ki predstavlja vse elemente, postavljene nad igralnim poljem. Prikazana je igra v konfiguraciji za dva igralca.

Vsak izmed igralcev začne igro v svojem kotu igralnega polja. V spodnjem desnem kotu je z rdečim oddajnikom označeno vojaško oporišče trenutnega igralca. Nasprotnikovo oporišče je označeno z modrim oddajnikom v zgornjem levem kotu.

Na konceptni sliki igra že poteka in vsak izmed igralcev ima pred vojaškim oporiščem tri tanke, ki se lahko premikajo po belih poljih. Z zeleno barvo so označena gozdna polja, po katerih gibanje ni mogoče. Prav tako gibanje ni mogoče po modrih poljih, ki označujejo vodo.

Ko igralec klikne na element, ki ga lahko na igralnem polju označi, se okrog elementa izriše rumen krog. V uporabniškem vmesniku se v zgornjem desnem kotu odpre novo okno, na katerem so izpisane vse pomembne informacije o trenutno izbranem elementu na igralnem polju. Na sliki je izbran tank MK I Atom in njegovi podatki so napisani v oknu z informacijami.

V zgornjem delu HUD-a je vrstica z informacijami o trenutnem krogu, času do konca kroga in količina denarja, ki ga ima igralec na voljo.

V spodnjem levem kotu HUD-a je vmesnik za klepetalnico, v kateri se lahko igralci pogovarjajo. V vnosno polje napišejo sporočilo in ga pošljejo s pritiskom na tipko *enter*. Na desni strani klepetalnice je bojni dnevnik. V njem se hrani zgodovina vseh izvedenih akcij v igri. To vključuje tako premike enot in napade kot tudi informacije o tem, kdo je na vrsti. Iz bojnega dnevnika je razviden celoten potek igre.

Če je igralno polje večje od enega zaslona, ga lahko uporabnik premika, če nad njim drži desno miškino tipko in ga potegne v eno izmed smeri.

4. PROTOTIP IGRE

Osnova za izdelavo prototipa je platforma Chilly Framework. Na podlagi tehničnih zahtev in načrta, predstavljenega v prejšnjem poglavju, smo izdelali dve ločeni aplikaciji. Prva deluje v odjemalcu, druga pa na strežniku. Obe ustrezata arhitekturi, ki smo jo opredelili v prvem poglavju.

Na čelnem delu sistema je brskalnik, ki skrbi za izrisovanje grafike in uporabniškega vmesnika (HUD). To je del, prek katerega bodo igralci upravljali z igro. Za izdelavo grafičnega vmesnika in izpolnitev tehnične zahteve ***REQ-1.1*** smo uporabili knjižnico Crafty Game Engine. Čeprav je mogoče izdelati igro brez namenskih knjižnic, uporaba že narejenega igralnega pogona bistveno skrajša čas razvoja in ga poenostavi.

Za izdelavo igre smo izbrali naslednje knjižnice za JavaScript, ki so prikazane v tabeli 7.

Tabela 7: Uporabljene knjižnice

Knjižnica	*Različica*	*Opis*
Chilly Framework	*0.2.1*	*Interno razvita knjižnica za komunikacijo s strežnikom.*
CraftyJS	*0.5.3*	*Igralni pogon HTML5.*
SoundManager	*2.97a*	*Knjižnica za predvajanje zvoka v brskalniku.*
UnderscoreJS	*1.4.2*	*Pomožna knjižnica za JavaScript s podporo funkcijskemu programiranju.*
jQuery	*1.7.0*	*Knjižnica za manipulacijo dokumentov HTML in upravljanje z dogodki.*

Crafty Game Engine je odprtokodni igralni pogon, ki je sestavljen iz zbirke knjižic za JavaScript, te pa so ogrodje za razvoj iger v brskalniku na podlagi standardnih spletnih tehnologij.

Izbrali smo ga, ker je relativno majhen (le 100kB), enostaven za uporabo in visoko prilagodljiv. V Craftyju so najpomembnejše strukturne komponente entitete in komponente. Entiteta je objekt, ki obstaja v svetu igre in ima poljubno vedenje, četudi je to le, da zaseda vizualni prostor. Vsak element na zaslonu je v terminologiji Crafty entiteta na zaslonu. Komponenta določa zbirko podatkov ali vedenj, ki se lahko nanašajo na eno ali več komponent (Torpey, 2013).

Povezava med komponentami in entitetami začne obstajati, ko entiteti določimo, da vsebuje eno ali več komponent. Tako lahko enostavno dodajamo nova vedenja na entitete. Če kot primer vzamemo komponento "premikanje", ki vsebuje kodo za zajem smernih tipk na tipkovnici in premikanje po X in Y osi glede na vhodne

podatke s tipkovnice, lahko na enostaven način določimo novo entiteto "avto", ki izvede "premikanje". Isto komponento lahko nato uporabimo še za entiteto "avtobus", ki si deli isto vedenje. Komponento-entitetni sistem smo uporabili za razvoj grafičnih entitet na zaslonu, igralnega polja in ozadja. Craftyjev pogon podpira tudi animiranje entitet, z njegovo pomočjo smo animirali tanke in eksplozije.

Za izdelavo uporabniškega vmesnika, HUD-a, smo uporabili HTML in knjižnico jQuery, ki se uporablja za enostavno manipulacijo dokumenta HTML in upravljanje z dogodki (Earle & Sharkie, 2010).

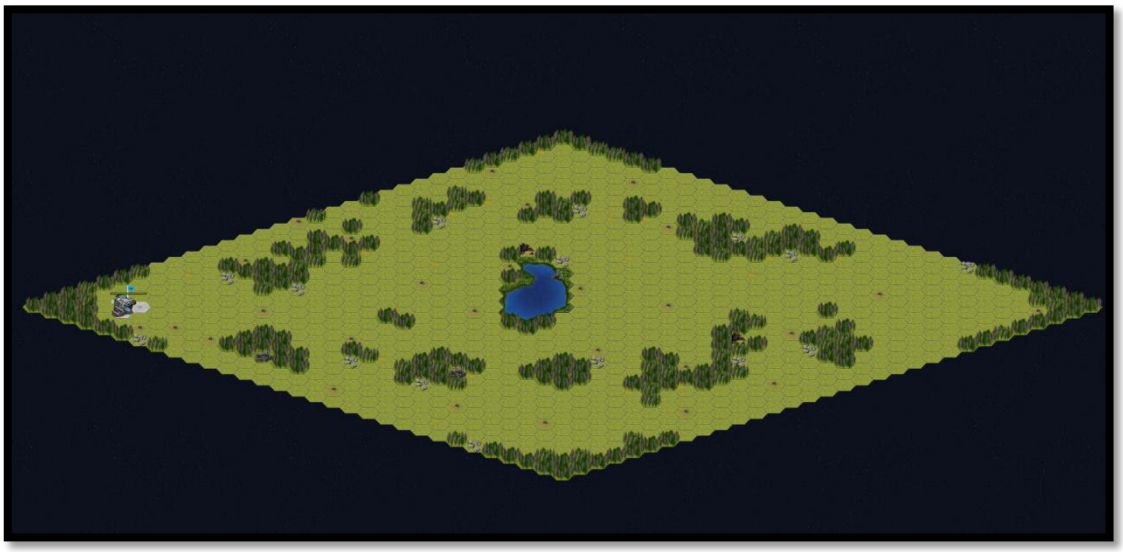
Podporne vlogo ima knjižnica underscore, ki razširi JavaScript in ponudi napredne jezikovne konstrukte, ki jih JavaScript nima ali pa v trenutnem brskalniku niso podprte.

Ker brskalniki še vedno niso idealno okolje za delo z zvokom, smo se odločili, da uporabimo knjižnico SoundManager, ki to nalogo olajša. Specifikacija HTML5 sicer podpira zvočno komponento, a je še vedno v razvoju. Podpora med brskalniki je zato slaba. Razvijalcu knjižnica odkrije enotni vmesnik, ne glede na tehnologijo, ki jo v ozadju uporablja za predvajanje zvoka in je odvisna od podpore brskalnika.

4.1. KOMPONENTE CRAFTY

Igralno polje in druge grafične elemente v igri smo izdelali kot komponente Crafty. Poglejmo si opis komponent in njihovo vlogo pri izvedbi igre.

4.1.1. KOMPONENTA MAP



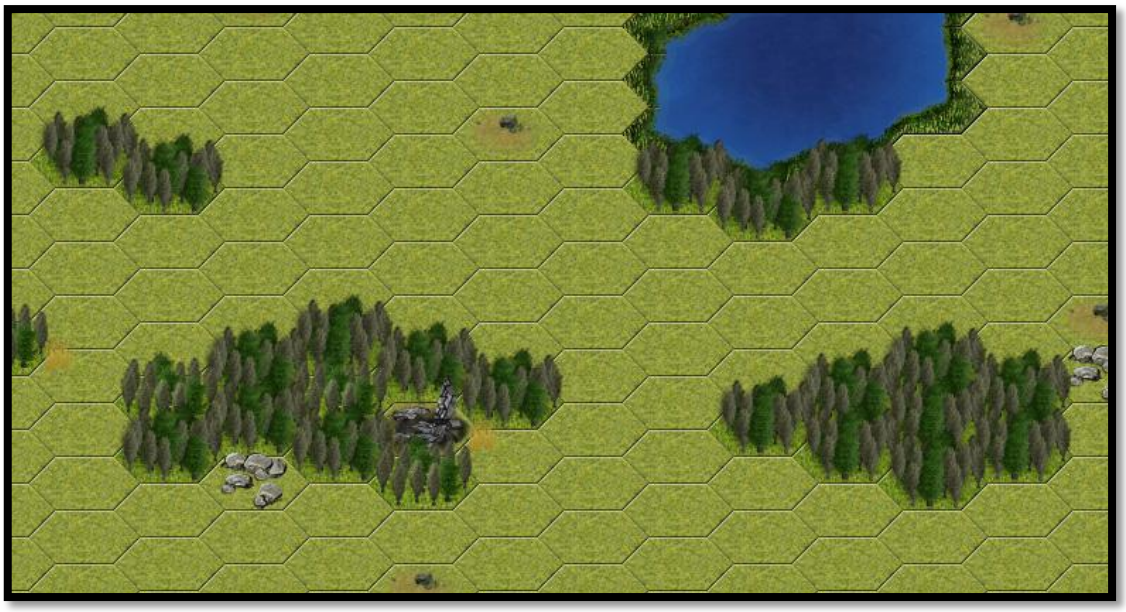
Slika 17: Igralno polje 30 x 30

Največja in za izvedbo najzahtevnejša komponenta je igralno polje. Slika 17 prikazuje končano igralno polje, sestavljeno iz šestkotnikov, v velikosti 30 x 30 enot, ki se nanaša na tehnično zahtevo ***REQ-1.2***. Podatki o tipih polj se naložijo iz JavaScript datoteke. Igralno polje je lahko poljubno veliko, saj smo razvili dinamično nalaganje polj, glede na velikost igralčevega vidnega polja. Tako tudi ob zelo velikem igralnem polju igra deluje odzivno.

Komponenta "HexMap" vsebuje vso logiko za izris igralnega polja in manipulacijo z njim. Glede na podatke o velikosti mreže in tipa vsakega šestkotnika, komponenta sestavi mapo in prevede koordinate X, Y v koordinate na zaslonu glede na velikost šestkotnika, ki smo jo nastavili. Hrani tudi vse podatke o tem, na katerih poljih so enote in katera so prosta, in podatke o trenutno označenem polju.

Dinamično nalaganje deluje v kombinaciji s Craftyjevim predstavitvenim poljem, pogleda njegov zamik in njegovo velikost ter poišče samo šestkotnike, ki so znotraj teh koordinat. S tem se prepreči izris šestkotnikov, ki niso v predstavitvenem polju in predstavljajo nepotrebno porabo sistemskih virov.

Potem ko se požene metoda za izris šestkotnikov dobimo končno podobo igralnega polja, kot je prikazano na Sliki 18.

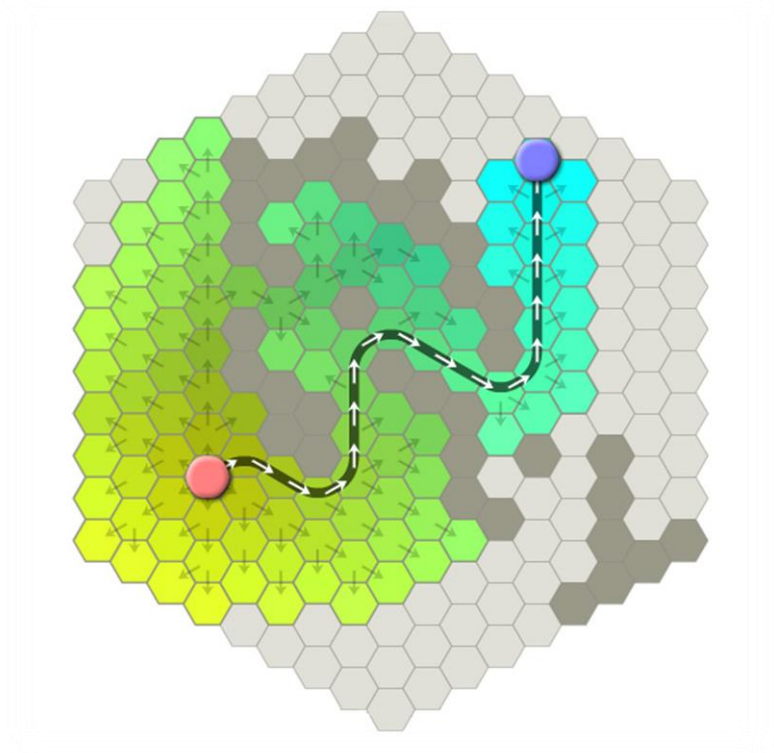


Slika 18: Mreža šestkotnikov

4.1.2. KOMPONENTA TILE

Slika 19: Različni tipi šestkotnikov, iz katerih je sestavljeno igralno polje

Komponenta Tile vsebuje kodo, ki jo izriše na zaslonu na položaju, ki se izračuna glede na koordinate in velikost in jo določi komponenta "Map" ob izrisu igralnega polja. Slika 19 prikazuje različne vrste šestkotnikov. Na voljo so različni tipi podlage, ki so osnovni gradniki polja. Med njimi so prehodni v zgornji vrstici, drugi in zadnji, v spodnji vrstici pa drugi in tretji (od leve proti desni). Drugi tipi so tipi neprehodnega terena. Slika 20 prikazuje, kako algoritem A* išče pot od točke rdeče do modre točke.



Slika 20: A iskanje najkrajše poti na šestkotni mreži*

V komponento "HexMap" smo uvedli metodo A* za iskanje najkrajše poti med dvema šestkotnikoma, ki se uporablja pri premikanju enot in izpolnjuje tehnično zahtevo ***REQ-1.10***. Računalniški algoritmi v matematičnem smislu delujejo na podlagi skupine točk v grafu. Igralno polje, sestavljeno iz plošč, lahko poenostavimo na skupek vozlišč. Iskanje z A* algoritmom je hitro in natančno. A* uporablja metodo iskanja najprej najboljših točk, da najde pot z najnižjo ceno od začetnega do končnega vozlišča. Sledi poti najmanjše pričakovane skupne cene oz. razdalje in hrani v pomnilniku prioritetno vrsto alternativnih segmentov poti po poti. Primer iskanja poti A* je razviden iz Slike 20.

4.1.3. KOMPONENTA BASE

Komponenta "base" vsebuje kodo za izris oporišča na igralnem polju. Slika 21 prikazuje *sprite* oporišča. Sprite je večja slika, ki vsebuje dele, ki se prikažejo na zaslonu. Crafty lahko izreže in na zaslonu prikaže samo določen del slike z enim izmed oporišč v začetnem ali porušenem stanju. Z zamikom po višini in širini ter širino in dolžino izreza smo določili, kateri del slike naj bo prikazan in v kakšnih pogojih se bo prikazal.

Slika 21: Vojaško oporišče v štirih različnih barvah

Ob napadu na oporišče smo želeli uporabniku vidno sporočiti, da je bil napad uspešen, zato se na zaslonu nad oporiščem prikaže eksplozija. Animirali smo jo z uporabo komponente SpriteAnimation, ki je sestavni del knjižnice Crafty. Treba je bilo izdelati vse sličice animacije, ki se prikažejo v hitrem zaporedju za kratek čas. To tvori tekočo animacijo, ki jo vidi igralec. Slika 22 prikazuje *sprite* za animacijo eksplozije na vojaškem oporišču.

Slika 22: Sličice animacije za eksplozijo na oporišču

4.1.4. KOMPONENTA TANK

Ta komponenta vsebuje podatke, ki so potrebni za izris tanka na igralnem polju. Slika 23 prikazuje sličice animacije, ki sestavljajo izstrelitev tankovske granate. Tank se lahko premika po igralnem polju, zato ima komponenta vgrajeno tudi animacijo premikanja po mapi glede na pot, ki jo izračuna A* algoritem na komponenti Map. Vsebuje tudi metodo za napad na sovražni tank ali oporišče. Obe akciji, premik in napad, ob izvedbi pošljeta podatke na strežnik, da jo ta pošlje tudi preostalim odjemalcem v igri. Tam lokalna komponenta "Tank" dobi podatke in izvede enak enako akcijo.

Slika 23: Sprite za animacijo tanka

4.2. STREŽNIK

Za izdelavo strežnika smo izvedli pogon igre (***REQ-1.4***), kot je opisan v poglavju 3.6. in kot izhaja iz tehničnih zahtev. Naša izvedba podpira večigralstvo do štirih oseb, kot izhaja iz zahteve ***REQ-1.5***. Izvedba objekta Game vsebuje upravitelja potez, kot izhaja iz zahteve ***REQ-1.6***.

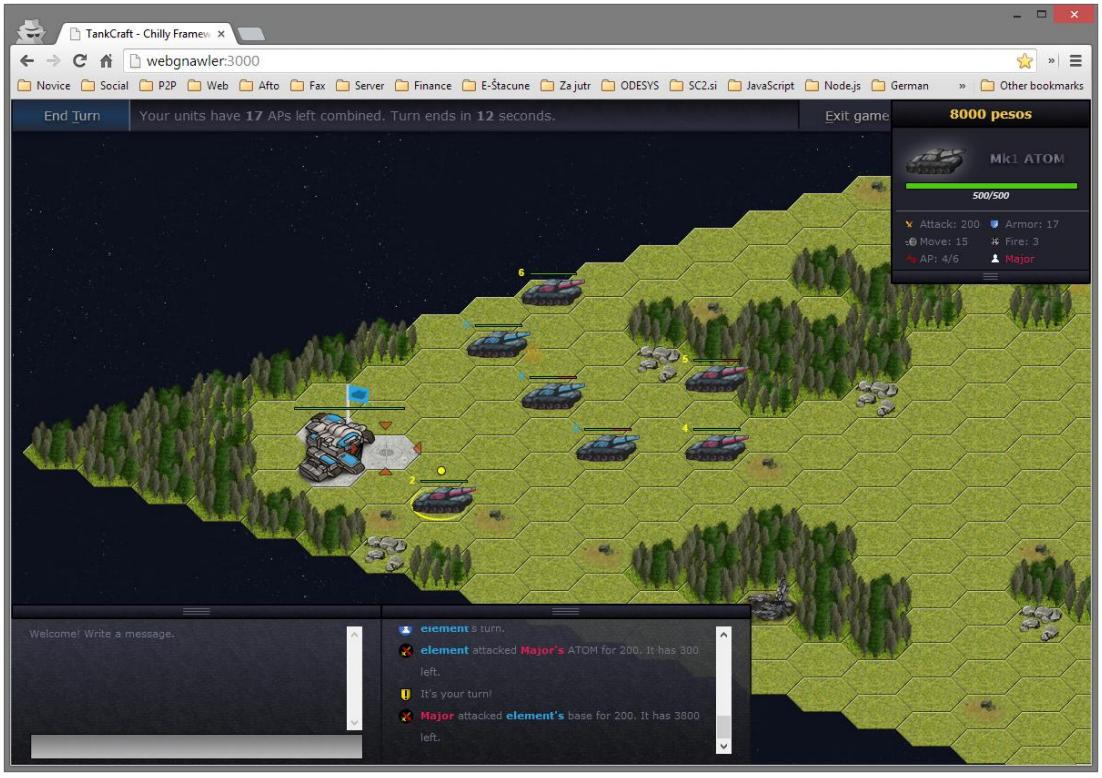
Sistem se odziva na naslednje zahtevke, ki prihajajo od odjemalcev. Uporabnik se prijavi v sistem s klicem na *login*. Chilly Framework ga doda v spisek uporabnikov in zažene iskalnik partije, ki je opredeljen v tehnični zahtevi ***REQ-1.12***. Ko partijo najde, odjemalcu potisne sporočilo o začetku igre. Na zahtevek *state* vrne trenutno stanje igre, ki je v seji igralca. Z zahtevkom na *sendChat* lahko uporabnik pošlje sporočilo drugim igralcem v igri, kar izpolnjuje tehnično zahtevo ***REQ-1.9***. Zahtevek na akcijo *over* predčasno konča potezo igralca in jo preda naslednjemu igralcu. Zahtevki na akcije *buy*, *move* in *attack* se prenesejo v trenutno delujočo igro in izvedejo v okviru igre. Igralec tako upravlja z enotami na igralnem polju.

Ko iskalnik partije zažene igro, se zažene tudi upravljalnik krogov, ki posodablja stanje igre vsakih 60 sekund, ko se konča krog. Vmesne posodobitve so odvisne od zahtevkov, ki prihajajo od odjemalcev.

Da lahko igra deluje, potrebuje objekte, ki smo jih določili v drugem poglavju. Uvedli smo jih v datoteko models.js po shemi, ki smo o opisali.

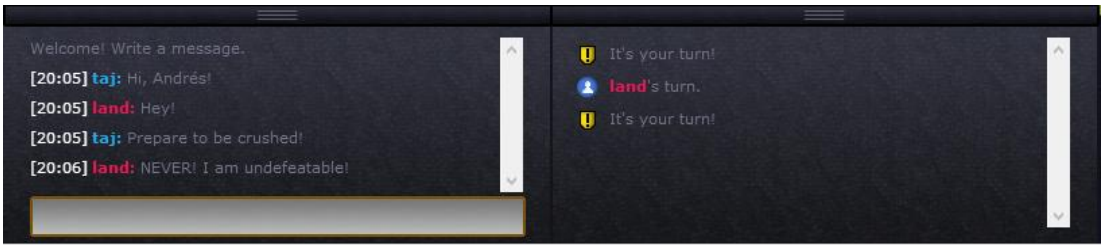
4.3. PRIKAZ DELOVANJA IGRE

Slika 24 prikazuje delovanje igre v brskalniku Google Chrome. Na sliki lahko vidimo uporabniški vmesnik HUD, ki izhaja iz tehnične zahteve ***REQ-1.3***. Igra teče v konfiguraciji dveh igralcev, uspešno pa smo jo testirali v konfiguraciji za največ štiri igralce. Vsak igralec začne v svojem kotu. Rdečemu igralcu se je uspelo prebiti do oporišča modrega igralca. Na igralnem polju je označil enega izmed svojih tankov in z njim napada vojaško oporišče rdečega igralca.



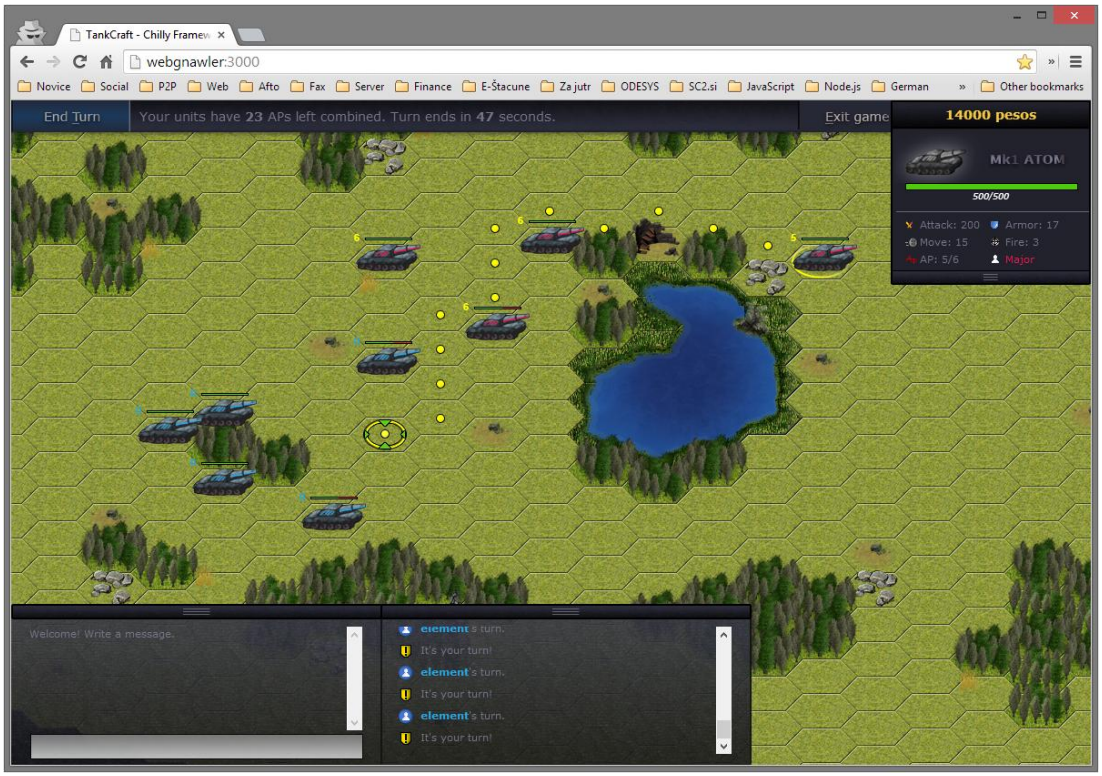
Slika 24: Rdeči igralec napada oporišče modrega igralca

Slika 25 prikazuje potek igre, ki je razviden iz vojaškega dnevnika, na levi pa je okno klepetalnice, v kateri sta se igralca pogovarjala.



Slika 25: Klepetalnica in vojaški dnevnik

Slika 26 prikazuje dva igralca, ki sta na sredini igralnega polja. Rdeči igralec se poskuša prebiti proti oporišču modrega igralca, ki je v levem spodnjem kotu igralnega polja. Označen je eden izmed tankov in v HUD-u so prikazani vsi pomembni podatki o njem. Rumena pot prikazuje delovanje A* algoritma in izračunano najkrajšo pot do ciljnega polja.



Slika 26: Rdeči igralec premika tank na drugo polje

5. ZAKLJUČEK

Na podlagi arhitekture, ki smo si jo zamislili in predstavili v prvem poglavju, smo izdelali platformo za razvoj večigralskih iger Chilly Framework. Prototip igre, ki smo jo izdelali na tej podlagi, je izpolnil vse postavljene zahteve. Po uspešno izvedeni prototipni rešitvi ugotavljamo, da je predlagana arhitektura primerna za razvoj večigralskih iger.

Ugotovili smo, da je mogoče arhitekturo uporabljati za razvoj tudi drugih tipov aplikacij. Pri izvedbi igre smo razvili klepetalnico, ki je bila zgolj ena izmed sestavin igre. Namesto igre bi bila lahko glavna usmeritev prototipa izdelava spletne klepetalnice z več sobami in dodatno funkcionalnostjo. Še vedno bi lahko uporabili isto arhitekturo, le oblikovanje odjemalca in izvedba na strani strežnika bi se razlikovala od igre. Ker je mogoče tako na strani odjemalca kot na strani strežnika izdelati poljubno kodo in zgolj uporabiti zmožnosti obeh knjižnic Chilly Frameworka za komunikacijo med strežnikom in odjemalcem, je mogoče graditi zelo raznolike aplikacije.

Ugotavljamo, da uporaba programske platforme Chilly Framework skrajša čas do izvedbe igre, saj je enostavno uvesti vse potrebno za dvosmerno komunikacijo in sinhronizacijo uporabnikov. Priložnosti za razvoj na Chilly Frameworku zato vidimo povsod, kjer narava aplikacije zahteva sinhronizacijo med uporabniki samimi in med uporabniki in strežnikom. Seveda arhitektura ni brez omejitev. Tehnologija dolgega izpraševanja ima pomanjkljivosti. Če je prednost njena dobra podpora med različnimi tipi odjemalcev, je po drugi strani prepustnost in hitrost prenosa podatkov premajhna za nekatere primere. Igre, na primer prvoosebne streljanke, morajo sporočiti premik igralca večdesetkrat na sekundo. V takih primerih odzivnost predlagane arhitekture pade pod uporabno mejo. Arhitektura tudi ne upošteva zamika pri prenosu podatkov, kar je v določenih tipih iger spet problem.

Priložnost za nadaljnji razvoj vidimo prav v odpravi omenjenih pomanjkljivosti. Namesto tehnologije dolgega izpraševanja bi lahko uporabili vse pogostejšo in hitrejšo različico dvosmerne komunikacije med odjemalcem in strežnikom imenovano spletne vtičnice. Razlika med spletnimi vtičnicami in tehnologijo, ki smo jo uporabili v sistemu, je ta, da prve vzpostavijo trajno povezavo med odjemalcem in strežnikom, po kateri lahko komunikacija poteka v obe smeri brez nepotrebnega balasta, ki ga prinese protokol HTTP, ki pri veliki količini poslanih sporočil ni zanemarljiv. Spletne vtičnice je mogoče izvesti na obstoječi arhitekturi node.js in testno smo jih že preverili, vendar še nismo v celoti zamenjali obstoječe rešitve dolgega izpraševanja (Ubl & Kitamura, 2010).

Da bo arhitektura lahko podprla tudi razvoj bolj zapletenih iger, ki potrebujejo bistveno večjo hitrost komuniciranja med odjemalci in strežnikom, bi bilo treba uvesti nekatere spremembe, ki hitro dvignejo zapletenost. Izvedba kompenzacije

omrežnih zakasnitev bi omogočala razvoj takih iger. Če igra vsebuje kompenzacijo zakasnitev, to omogoča igralcu, da ne opazi vplivov omrežne zakasnitve. Da je to mogoče, vsak igralec izvaja celoten pogon igre v svojem odjemalcu. Poleg tega se pogon igre izvaja tudi na strežniku, kjer se mora voditi celotna zgodovina igre in zamika do odjemalcev. Vsaka akcija od odjemalca se izvede v stanju iz preteklosti. Iz napisanega lahko vidimo, da je tema obsežna in zapletena in bi lahko bila obravnavana v ločenem magistrskem delu.

Chilly Framework smo objavili pod odprtokodnima licencama MIT in GPL na spletni strani chillyframework.com, kjer je prosto dostopen za prenos. Na spletni strani sta tudi forum za pomoč uporabnikom in dokumentacija v angleščini. Do kode je mogoče dostopati tudi na spletni strani GitHub[1], kjer jo je mogoče enostavno kopirati v svoj repozitorij Git in začeti gradnjo igre. Prav tako ga je mogoče dobiti z uporabo upravljalnika node.js paketov NPM.

Na podlagi uspešno razvite prototipne različice igre Shell Wars smo se odločili, da razvoj nadaljujemo. Slika 27 prikazuje grafično oblikovanje produkcijske različice.

Slika 27: Produkcijska različica igre Shell Wars

1 https://github.com/TajPelc/Chilly-Framework

LITERATURA IN VIRI

Åbo Akademi. (2012). *Åbo Akademi Department of Information Technologies*. Prevzeto 16. 7 2013 iz ICT Showroom is an annual event for student projects: https://research.it.abo.fi/news-and-events/stories/ict-showroom-is-an-annual-event-for-student-projects

Bright, P. (15. 3 2011). *ArsTechnica*. Prevzeto 4. 6 2012 iz The most modern browser there is: Internet Explorer 9 reviewed: http://arstechnica.com/information-technology/2011/03/the-most-modern-browser-there-is-internet-explorer-9-reviewed/

Constantine, A. C. (9. 6 2013). *Udemy*. Prevzeto 15. 8 2013 iz Why You Should Learn Node.js Today: https://www.udemy.com/blog/learn-node-js/

Crockford, D. (2008). *Javascript: The Good Parts.* Sebastopol: O'Reilly Media.

Earle, C., & Sharkie, C. (2010). *jQuery: Novice to Ninja.* Collingwood: Sitepoint Pty. Ltd.

Gaudiosi, J. (18. 7 2012). *Forbes*. Prevzeto 16. 8 2013 iz New Reports Forecast Global Video Game Industry Will Reach $82 Billion By 2017: http://www.forbes.com/sites/johngaudiosi/2012/07/18/new-reports-forecasts-global-video-game-industry-will-reach-82-billion-by-2017/

Gregory, J. (2009). *Game Engine Architecture.* Wellesly, Massachusetts: A K Peters, Ltd.

Gwilliam, A. (2012). *Quora*. Prevzeto 6. 4 2012 iz Web Browsers: Why is Internet Explorer so bad?

Harper, E. (1. 8 2013). *Jostiq*. Prevzeto 15. 8 2013 iz Activision earnings call offers more insight on WoW subscriber losses: http://wow.joystiq.com/2013/08/01/activision-earnings-call-offers-more-insight-on-wow-subscriber-l/

Herron, D. (2011). *Node Web Development.* Birmingham: PACKT Publishing.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). *Design Science in Information Systems Research.* MIS Quarterly.

Melonfire, C. (8. 6 2007). *Tech Republic*. Prevzeto 31. 8 2013 iz Understanding the pros and cons of the Waterfall Model of software development: http://www.techrepublic.com/article/understanding-the-pros-and-cons-of-the-waterfall-model-of-software-development/

nodejs. (2013). *nodejs.org*. Prevzeto 23. 8 2013 iz Node Packaged Modules: https://npmjs.org/

Novak, J. (2012). *Game Development Essentials: An Introduction.* Hampshire: Cengage Learning.

Porter, J. (16. 8 2013). *Bokardo*. Pridobljeno iz Principles of user interface design: http://bokardo.com/principles-of-user-interface-design/

Ravuya, G. (14. 6 2010). *GameDev StackExchange*. Prevzeto 15. 6 2012 iz Why is it so hard to develop a MMO?: http://gamedev.stackexchange.com/questions/90/why-is-it-so-hard-to-develop-a-mmo

Scott Allen, K. (2012). *What Every Web Developer Should Know About HTTP*. OdeToCode.com.

Sliwinski, A. (3. 1 2012). *Super Meat Boy surpasses 1 million sales*. Prevzeto 20. 8 2013 iz http://www.joystiq.com/2012/01/03/super-meat-boy-surpasses-1-million-sales/: http://www.joystiq.com/2012/01/03/super-meat-boy-surpasses-1-million-sales/

Ta, S. (14. 3 2013). *Examiner*. Prevzeto 17. 8 2013 iz 'Assassin's Creed 4' PS4 and Xbox 720 development team size revealed: http://www.examiner.com/article/assassin-s-creed-4-ps4-and-xbox-720-development-team-size-revealed

Torpey, D. (1. 2 2013). *Buildnewgames*. Prevzeto 20. 8 2013 iz An Introduction to the Crafty Game Engine: http://buildnewgames.com/introduction-to-crafty/

Ubl, M., & Kitamura, E. (20. 9 2010). *HTML5 ROCKS*. Prevzeto 28. 8 2013 iz Introducing WebSockets: Bringing Sockets to the Web: http://www.html5rocks.com/en/tutorials/websockets/basics/

Vaughn-Nichols, S. J. (9. 11 2011). *ZDnet*. Pridobljeno iz Flas is dead. Long live HTML5.: http://www.zdnet.com/blog/networking/flash-is-dead-long-live-html5/1633

KAZALO SLIK

KAZALO TABEL

PRILOGA 1: SPECIFIKACIJA TEHNIČNIH ZAHTEV

ID	***REQ-1.1***
Opis	*Igra mora podpirati grafični uporabniški vmesnik v standardu HTML5, ki je združljiv z vsemi novejšimi brskalniki IE10+, Mozilla, Chrome, Opera.*
Prioriteta	*1 – visoka.*
Odvisnosti	
Vhod	*Igralci odprejo URL-naslov, na katerem se nahaja igra.*
Izhod	*Igralci komunicirajo z igro prek grafičnega vmesnika s pomočjo miške.*
Akterji	*Vsi igralci.*

ID	***REQ-1.2***
Opis	*Igra mora prikazovati igralno polje v velikosti 30 x 30 enot, sestavljeno iz šestkotnikov, na podlagi v datoteki shranjenega modela.*
Prioriteta	*1 – visoka-*
Odvisnosti	*REQ-1.1*
Vhod	*Igralci odprejo URL-naslov, na katerem se nahaja igra .*
Izhod	*Igralci komunicirajo z igro prek grafičnega vmesnika s pomočjo miške.*
Akterji	*Vsi igralci.*

ID	***REQ-1.3***
Opis	*Igra mora imeti HUD (head-up display), ki prikazuje vse potrebne informacije o stanju igre. Podatki obsegajo število akcijskih točk, ki so na voljo v tem krogu, čas do konca kroga, gumb za predčasno končanje kroga, informacijo o količini denarja, ki ga ima igralec. Na desni strani mora biti informacija o trenutno izbrani enoti na igralnem polju.*
Prioriteta	*1 – visoka.*
Odvisnosti	*REQ-1.1*
Vhod	*Igralci vstopijo v igro .*
Izhod	*Igra predstavi HUD-vmesnik s pomembnimi informacijami.*
Akterji	*Vsi igralci.*

ID	***REQ-1.4***
Opis	*Igra mora podpirati igralni pogon na strežniku, ki vodi evidenco o stanju igre in preverja veljavnost zahtev, ki jih pošiljajo igralci.*
Prioriteta	*1 – visoka.*
Odvisnosti	
Vhod	*Igralec premakne tank na novo polje.*
Izhod	*Tank se premakne pri vseh igralcih, če je akcija dovoljena (so pogoji izpolnjeni).*
Akterji	*Vsi igralci.*

ID	***REQ-1.5***
Opis	*Igra mora podpirati večigralstvo od dveh do štirih igralcev v eni partiji in večje število hkratnih partij (omejeno samo s sistemskimi viri).*
Prioriteta	*1 – visoka.*
Odvisnosti	
Vhod	*Igralci se povežejo na strežnik.*
Izhod	*Igralci lahko igrajo proti drugim igralcem.*
Akterji	*Vsi igralci.*

ID	***REQ-1.6***
Opis	*Igra mora vsebovati upravitelja potez, ki skrbi za časovno omejitev na potezo in kateri igralec je na vrsti. Pognati mora zahtevane ukaze vsakič, ko se zamenja igralec, kot je na primer povečati količino denarja, ki jo ima trenutno aktivni igralec.*
Prioriteta	*1 – visoka.*
Odvisnosti	*REQ-1.4*
Vhod	*Igralec klikne na gumb "končaj potezo".*
Izhod	*Upravitelj potez konča potezo trenutnega igralca in omogoči interakcijo z enotami na igralnem polju naslednjemu igralcu.*
Akterji	*Vsi igralci.*

ID	***REQ-1.7***
Opis	*Igra mora podpirati sinhronizacijo stanja igre med uporabniki v realnem času.*
Prioriteta	*1 – visoka.*
Odvisnosti	*REQ-1.4*
Vhod	*Igralec premakne tank na novo polje.*
Izhod	*Vsi drugi igralci nemudoma vidijo premik tanka.*
Akterji	*Vsi igralci.*

ID	***REQ-1.8***
Opis	*Igra mora v primeru prekinitvi povezave in njene ponovne vzpostavitve posredovati podatke o stanju igre in ponovno izrisati igralno polje z najnovejši podatki. Igra se potem nadaljuje.*
Prioriteta	*2 – poprečna.*
Odvisnosti	*REQ-1.1, REQ-1.2, REQ-1.3, REQ-1.4*
Vhod	*Igralec pritisne gumb "osveži" v brskalniku.*
Izhod	*Igra se ponovno naloži v trenutnem stanju.*
Akterji	*Vsi igralci.*

ID	***REQ-1.9***
Opis	*Igra mora podpirati pomenkovalnico v realnem času.*
Prioriteta	*2 – poprečna.*
Odvisnosti	*REQ-1.1, REQ-1.2, REQ-1.3, REQ-1.4*
Vhod	*Igralec pošlje sporočilo.*
Izhod	*Igra se spet naloži v trenutnem stanju.*
Akterji	*Vsi igralci.*

ID	***REQ-1.10***
Opis	*Igre mora podpirati A* algoritem za iskanje najkrajše poti od točke A do B na mreži šestkotnikov.*
Prioriteta	*2 – poprečna.*
Odvisnosti	*REQ-1.1, REQ-1.2*
Vhod	*Igralec izbere tank in prestavi miško nad prazno polje.*
Izhod	*Igra poišče najkrajšo prosto pot med tankom in praznim poljem.*
Akterji	*Vsi igralci.*

ID	***REQ-1.11***
Opis	*Igra mora podpirati grafični urejevalnik igralnega polja, s katerim je mogoče na enostaven način izdelovati dodatne ravni.*
Prioriteta	*3 – nizka.*
Odvisnosti	*REQ-1.1, REQ-1.2*
Vhod	*Razvijalec v grafičnem okolju spreminja tip igralnih polj.*
Izhod	*Igra shrani novo različico igralnega nivoja.*
Akterji	*Razvijalci.*

ID	***REQ-1.12***
Opis	*Igra mora podpirati iskalnik partij, s katerim lahko igralci najdejo soigralce.*
Prioriteta	*2 – poprečna.*
Odvisnosti	*REQ-1.1, REQ-1.2*
Vhod	*Razvijalec v grafičnem okolju spreminja tip igralnih polj.*
Izhod	*Igra shrani novo različico igralnega nivoja.*
Akterji	*Vsi igralci.*