Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Gregor Šinkovec

Domensko-specifičen jezik za izdelavo preprostih računalniških iger

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: viš. pred. dr. Igor Rožanc

Ljubljana 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:
Domensko-specifičen jezik za izdelavo preprostih računalniških iger

Tematika naloge:

Domensko-specifični jeziki (DSL) so programski jeziki, ki so prilagojeni učinkovitemu reševanju problemov v specifični problemski domeni. Kot taki imajo nekatere pomembne prednosti in tudi omejitve.

V diplomski nalogi najprej preučite in predstavite področje razvoja domensko-specifičnih jezikov. Na podlagi tega v programskem jeziku Ruby zasnujte in izdelajte domensko specifičen jezik za učinkovit razvoj preprostih računalniških iger. Njegovo uporabo prikažite na zgledu razvoja preproste igre. Nalogo zaključite s predstavitvijo vaših izkušenj po opravljenem delu.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Gregor Šinkovec sem avtor diplomskega dela z naslovom:

Domensko-specifičen jezik za izdelavo preprostih računalniških iger

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom viš. pred. dr. Igorja Rožanca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter 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 9. februarja 2016 Podpis avtorja:

Hvala mentorju, viš. pred. dr. Igorju Rožancu, za usmerjanje in potrpežljivost.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
DSL	Domain-Specific Language	domensko-specifičen jezik
GPL	General-Purpose Language	splošno-namenski jezik
HTML	HyperText Markup Language	jezik za označevanje nadbesedila
DOM	Document Object Model	objektni model dokumenta
CSS	Cascading Style Sheets	kaskadne stilske podloge
XML	eXtensible Markup Language	razširljiv označevalni jezik
JS	JavaScript	JavaScript

Povzetek

Cilj diplomskega dela je implementacija domensko-specifičnega jezika, ki bo programerju omogočal hitro generiranje skripte za računalniško igro, pri čemer se programerju ne bo treba obremenjevati s podrobnostmi pogona in zahtevnejših algoritmov. Pri tem se skuša doseči čim večjo neodvisnost od jezika, na katerem končna skripta temelji.

V uvodnem poglavju na kratko pojasnimo, kako izbira programskega jezika vpliva na razvoj iger in kakšna je uporaba domensko-specifičnih jezikov v modernih igralnih pogonih. Povzamemo tudi namen in problematiko diplomskega dela, ki pa sta bol podrobno razdelana v kasnejših poglavjih.

Drugo poglavje na grobo zaobjema domensko-specifične jezike. Izpostavimo splošne skupine, v katere jih delimo, razlike med njimi ter prednosti in slabosti njihove uporabe. Sledi kratka predstavitev jezika Ruby in nekaj besed o motivaciji za njegovo uporabo.

V tretjem poglavju pojasnimo zakaj smo se odločili za izdelavo domensko-specifičnega jezika in utemeljimo našo izbiro končne platforme. Prikažemo tudi glavne dele izdelave preprostega igralnega pogona in implementacijo osnovnih jezikovnih konstruktov, ki jih potrebujemo za jedro našega domensko-specifičnega jezika.

V četrtem poglavju je prikazana uporaba novega jezika na implementaciji preproste igre Breakout. Predstavimo osnovna pravila igre in pokažemo, kako jih opišemo z našim domensko-specifičnim jezikom, nato pa na kratko obrazložimo način prevajanja kode na končno platformo, poganjanje igre v spletnem brskalniku in možnosti testiranja igre.

Na koncu izpostavimo še sklepne ugotovitve in možnosti razvoja drugih iger v narejenem jeziku, kot tudi izboljšav jezika in igralnega pogona.

Ključne besede: domensko-specifični jeziki, Ruby, JavaScript, razvoj računalniških iger.

Abstract

The goal of this thesis is the implementation of a domain-specific language which will allow a programmer to quickly generate the code of a computer game without the need to concern themselves with game engine specifics or complex algorightms. We will endeavour to achieve as much independence from the base programming language as possible.

In the introduction we give a short explanation of how the choice of the programming language affects the development of computer games and how domain-specific languages are used in modern game engines. We also summarize the purpose and the problem domain of the thesis, both of which will be described in more detail in the following chapters.

The second chapter roughly covers domain-specific languages. We highlight the general categories into which they're divided, the differences between them, and the benefits and drawbacks of working with DSLs. This is followed by a short presentation of the Ruby programming language and a few words about our motivation for its use.

In chapter 3, we explain our decision to create a domain-specific language and the rationale behind our choice for the target platform. We also outline the main steps of developing a simple game engine and show how to implement some of the basic language structures required for the core of our domain-specific language.

In chapter 4, we show how to implement a simple Breakout game with the use of our newly-created language. We explain the basic rules of the game and demonstrate how to describe them with our domain-specific language.

This is followed by a short explanation of how to deploy the code to the target platform, how to run the game in a web browser, and a quick summary of the methods that can be used to test our programs.

Finally, we present our conclusions based on the work done and the opportunities for further game development and possible improvements to the language and game engine.

Keywords: domain-specific languages, Ruby, JavaScript, computer game development.

Poglavje 1

Uvod

Ena najpomembnejših odločitev pri izdelavi računalniške igre je izbira programskega jezika, v katerem bo igra narejena. Ta odločitev vpliva na težavnost implementacije igralnega pogona (angl. *game engine*) ne glede na to, ali gre za obstoječi pogon ali popolnoma svežo zasnovo, na programerjevo sposobnost pisanja v tem programskem jeziku, na platforme za katere je igro možno razviti, in podobno.

Večina današnjih pogonov za igre omogoča programerjem izdelavo igre v posebno-namenskem jeziku (npr. UnityScript v pogonu Unity [1] ali Ruby Game Scripting System v orodjih RPG Maker [2]), ki je specifično prirejen izdelavi igre v svojem okolju. Takšni jeziki spadajo v skupino programskih jezikov, ki jim bolj strokovno rečemo domensko-specifični jeziki (angl. *domain-specific languages* ali DSL na kratko), ker se osredotočajo na razvoj programske opreme v neki ožji domeni, v nasprotju s splošno-namenskimi programskimi jeziki (angl. *general-purpose languages* ali GPL), kjer ni tako. Domensko-specifični jeziki seveda niso omejeni na izdelavo računalniških iger, to je zgolj ena izmed mnogih domen, kjer se pogosto uporabljajo.

Cilj tega diplomskega dela je razviti tak domensko-specifičen jezik, ki bo poenostavil implementacijo računalniške igre na podoben način kot zgoraj omenjeni primeri. Pri tem bo razvidno, koliko časa in truda je potrebnega za razvoj in učenje tega jezika ter kakšne so prednosti in slabosti njegove

uporabe. Cilj je pokazati, da je uporaba domensko-specifičnih jezikov na tem področju utemeljena, obenem pa tudi dvigniti splošno osveščenost o domensko-specifičnih jezikih.

V nadaljevanju bomo najprej bolj natančno opredelili domensko-specifične jezike. Pogledali si bomo, kakšne so njihove prednosti in kje pogosto naletimo na težave pri njihovi uporabi. Na grobo bo predstavljen tudi programski jezik Ruby, v katerem bo napisan nov domensko-specifičen jezik.

Osrednji del diplomskega dela je razvoj računalniške igre. Pojasnili bomo, kaj razvoj igralnega pogona pravzaprav zaobjema in kako se lotimo njegove implementacije v splošno-namenskem programskem jeziku JavaScript, nato pa se bomo lotili razvoja domensko-specifičnega jezika, ki na njem temelji.

Za namen testiranja naše hipoteze bo na koncu uporaba novega jezika prikazana na izdelavi računalniške igre Breakout. Dodatno bo demonstrirano tudi poganjanje končnega izdelka in pojasnene možnosti testiranja kode v vseh fazah nastajanja.

Nalogo bomo zaključili z analizo opravljenega dela in možnostmi za nadgradnjo rešitve.

Poglavje 2

Domensko-specifični jeziki

2.1 Domensko-specifični jeziki

Definicija domensko-specifičnih jezikov je relativno preprosta, a razmejitve med povezanimi koncepti (kot so aplikacijski vmesniki, knjižnice in programska orodja) niso vedno tako jasne kot bi si želeli. Prav tako ni vedno očitno, kaj jih ločuje od splošno-namenskih jezikov.

Za izhodiščno točko moramo najprej opredeliti nekaj splošnih izrazov. Programski jeziki so sredstva s katerimi programerji posredujejo ukaze računalnikom [3]. Splošno-namenski programski jeziki so programski jeziki, ki programerjem omogočajo razvoj programske opreme v veliki količini aplikacijskih domen. Vsi splošno-namenski programski jeziki zadoščajo Turingovi popolnosti [4], kar pomeni, da je z njimi možno implementirati vse, kar je izračunljivo s Turingovim strojem. Hkrati to tudi pomeni, da so vsi splošno-namenski jeziki izmenljivi, četudi vsak izmed njih ni nujno primeren za rešitev poljubnega problema.

Domensko-specifični programski jeziki so taki programski jeziki, ki so optimizirani za specifičen nabor problemov, ki jim rečemo aplikacijska domena [4, 5]. Pri razvoju domensko-specifičnih jezikov navadno žrtvujemo fleksibilnost in izraznost splošno-namenskih jezikov v zameno za lažje in bolj konkretno izražanje domenskih konceptov, kar prispeva k produktivnosti programer-

	`Splošno-namenski jeziki`	`Domensko-specifični jeziki`
Domena	velika in kompleksna	manjša in dobro definirana
Obširnost jezika	velik	majhen
Turingova popolnost	vedno	pogosto ne
Uporabniške abstrakcije	sofisticirane	omejene
Izvedba	preko vmesnega GPLja	native
Življenjska doba	leta ali desetletja	meseci ali leta (glede na kontekst)
Razvijalci	guru ali odbor	nekaj inžinirjev in domenskih ekspertov
Skupnost uporabnikov	velika, anonimna in razširjena	majhna, dostopna in lokalna
Razvoj	počasen, ponavadi standardiziran	hiter in dinamičen
Zapadlost kode	praktično nemogoča	možna

Tabela 2.1: Splošne lastnosti programskih jezikov [4].

jev in jedrnatosti kode. Posledično to tudi pomeni, da domensko-specifični jeziki le redkokdaj zadoščajo Turingovi popolnosti. Zaradi manjše obsežnosti domensko-specifičnih jezikov, jim občasno rečemo tudi "mini jeziki" [4].

V tabeli 2.1 izpostavimo še nekaj dodatnih razlik med splošno-namenskimi in domensko-specifičnimi jeziki. Pomembno je poudariti, da v praksi jeziki ne zadoščajo vsem pogojem, ki so navedeni. Večina jezikov (bodisi splošno-namenskih ali domensko-specifičnih) ima pogosto lastnosti iz obeh stolpcev tabele in jih lahko uvrščamo, bodisi na podlagi tega, katerih imajo več, ali pa zgolj glede na kontekst.

Ker linije med domensko-specifičnimi jeziki, splošno-namenskimi jeziki in drugimi sorodnimi koncepti niso jasno začrtane, delitev v večini primerov ni smiselna. Hkrati je točna definicija aplikacijske domene tudi nekaj o čemer se večina ljudi ne strinja, tako da je mogoče bolj naravno razmišljati le o meri domenske-specifičnosti določenega jezika.

Pri domensko-specifičnih programskih jezikih govorimo o treh glavnih sestavinah: *jezikovni zasnovi*, *izvajalnem pogonu* in *ciljni platformi* (slika 2.1).

Jezikovna zasnova je naš domensko-specifičen jezik. Ta sestoji iz konkretne sintakse (notacije, ki jo programer uporablja za pisanje programov), abstraktne sintakse (podatkovni model programov napisanih v jeziku), statične semantike (nabor omejitev in podatkovnih tipov) in izvajalne semantike (dejanski "pomen" programa ob izvajanju) [4].

Slika 2.1: Jezikovna zasnova, izvajalni pogon in ciljna platforma.

Izvajalni pogon (angl. *execution engine*) je v večini primerov izmenljiv in služi kot povezava med jezikom in ciljno platformo. Izvajalne pogone delimo na generatorje, ki kodo domensko-specifičnega jezika preoblikujejo v kodo primerno za izvajanje na ciljni platformi (ponavadi kodo napisano v splošno-namenskem jeziku) in interpreterje, ki kodo naložijo in jo neposredno izvajajo na ciljni platformi [4].

Ciljna platforma je lahko fizična platforma (npr. osebni računalnik ali mobilni telefon) ali pa določeno programsko okolj,e v katerem se bodo končni programi izvajali. Ciljna platforma je v večini primerov fiksna, zato se jezikovna zasnova in pa predvsem izvajalni pogon prilagajata njej.

2.2 Vrste domensko-specifičnih jezikov

Domensko-specifične jezike pogosto delimo na *notranje* in *zunanje*, Fowler pa k tej delitvi dodaja še *jezikovne delavnice* [4, 5]:

- ***Zunanji domensko-specifični jeziki*** so jeziki, ki so v celoti ločeni od glavnega programskega jezika aplikacije v kontekstu katere se izvajajo. Za procesiranje kode zunanjega domensko-specifičnega jezika je ponavadi zadolžen razčlenjevalnik (angl. *parser*) v gostiteljski aplikaciji. Klasičen primer zunanjega domensko specifičnega jezika je SQL (angl. *Structured Query Languge*) za delo s podatkovnimi bazami [6].

- ***Notranji domensko-specifični jeziki*** so jeziki, ki so integrirani neposredno v glavni (splošno-namenski) jezik aplikacije. Koda notranjega domensko-specifičnega jezika je veljavna koda v nekem splošno-namenskem programskem jeziku. Razlike med notranjimi domensko-specifičnimi jeziki in programskimi vmesniki so težko opazne. Fowler meni, da je poglavitvena razlika v načinu uporabe [5].

- ***Jezikovne delavnice*** (angl. *language workbench*) so posebna razvojna okolja, ki so namenjena gradnji in uporabi domensko-specifičnih jezikov ter podpirajo koncept jezikovnega programiranja (angl. *language-oriented programming*).

Voelter dodatno deli domensko-specifične jezike še na *"tehnične"* domensko-specifične jezike, ki so namenjeni za uporabo programerjem, in *"poslovne"* domensko-specifične jezike, ki so prirejeni za ljudi brez programerske podlage (recimo domenskim ekspertom) [4].

Za katero od teh variacij se odločimo (če sploh), je odvisno od problemske domene in strank, za katere jezik razvijamo. V veliko primerih je mogoča implementacija tako zunanjega kot notranjega jezika ali pa tudi več domensko-specifičnih jezikov za različne dele projekta. V ta namen moramo pretehtati vse prednosti in slabosti različnih opcij ter narediti analizo stroškov in koristi [9]. Ne glede na odločitev pa si pri tvorbi novega jezika pogosto pomagamo z zasnovnimi vzorci (angl. *design patterns*), ki so se razvili z leti dela na tem področju [7, 8].

2.3 Prednosti in slabosti

Uporaba domensko-specifičnih jezikov ima naslednje prednosti [4, 5, 8]:

- *Lažje izražanje domenskih konceptov in formalizmov.* Ker so domensko-specifični jeziki posebno prilagojeni ožji domeni, abstrakcije v teh jezikih omogočajo direktno izražanje domenskega znanja. To med drugim

izboljša berljivost kode in komunikacijo med razvojniki (programerji) in domenskimi eksperti.

- *Izboljšana produktivnost.* Zaradi lažjega izražanja konceptov in berljivosti kode se posledično izboljša tudi hitrost pisanja kode, iskanje napak in izvajanje sprememb v kodi.

- *Kvaliteta kode.* Ožja izraznost in enostavnejša sintaksa zagotavljata tudi, da se pri pisanju pojavlja manjša količina napak, ki jih je lažje opaziti. Hkrati je arhitekturna zasnova programa bolj pravilna in koda lažja za vzdrževanje.

- *Zanesljivost programov.* Domensko-specifični jeziki obravnavajo probleme na način, ki se ne obremenjuje z implementacijskimi podrobnostmi, zato so tako napisani programi bolj semantično obogateni. To pomeni, da je na njih tudi lažje izvajati razne analitične postopke in loviti napake pri izvajanju, kar prispeva k uspešni validaciji in verifikaciji.

- *Dolgotrajnost informacije.* Vsaj v primeru zunanjih domensko-specifičnih jezikov so programi neodvisni od izvajalnega pogona in platforme. Ker je informacija izražena na nivoju abstrakcije, ki je smiselna v kontekstu domene, je ponavadi možna in lažja pretvorba na nove tehnologije.

- *Neposredno sodelovanje z domenskimi eksperti.* Domensko-specifični jeziki so navadno razviti v sodelovanju z domenskimi eksperti, kar zagotavlja stalno in učinkovito komunikacijo med njimi in programerji, hkrati pa proces implementacije jezika omogoča boljše razmevanje domene in kode, tako s strani programerjev kot domenskih ekspertov.

- *Učinkovita orodja za zunanje domensko-specifične jezike.* Pri zunanjih domensko-specifičnih jezikih načeloma uporabljamo posebna razvojna orodja, ki se jezika zavedajo in omogočajo produktiven razvoj programov, statično analizo kode, razhroščevanje in ostale bonitete, ki so jih drugače deležni splošno-namenski jeziki.

- *Nižja cena izvajanja.* Pri jezikih, ki generirajo izvorno kodo, je možna implementacija domensko-specifičnih abstrakcij brez rabe dodatnih sistemskih sredstev, saj je generator zmožen avtomatično generirati učinkovito kodo.

- *Neodvisnost od končne platforme.* Domensko-specifični jeziki so ponavadi implementirani z vmesnim izvajalnim pogonom (generator ali interpreter), kar programom pisanim v teh jezikih omogoča popolno neodvisnost od končne platforme in možnost prenosa z menjavo izvajalnega pogona.

Seveda pa implementacija in uporaba domensko-specifičnih jezikov ni brez težav. Spodaj so navedeni pogosti izzivi, ki jih srečamo, ko se odločamo za domensko-specifične jezike [4, 5, 8]:

- *Količina truda potrebnega za implementacijo.* Kreiranje novega jezika ni enostaven problem. Če se ga razvija kot del obstoječega projekta, potem je potrebno oceniti, ali prednosti odtehtajo ceno razvoja.

- *Cena urjenja.* Za urjenje novega jezika je potreben čas, ki ni namenjen dejanskemu delu na projektu. V primeru, da se jezik razvija kot del projekta, je ponavadi učenje jezika že del razvoja in preučevanja domene.

- *Sposobnost kvalitetne izdelave.* Za učinkovito izdelavo domensko-specifičnega jezika so potrebne spretnost in izkušnje. Slabo implementiran jezik je lahko dolgoročno slaba investicija. Za lažjo in bolj kvalitetno implementacijo je na voljo vrsta programskih orodij in vodičev, ki pa ponavadi niso brezplačna.

- *Proces izdelave in vzdrževanja.* Ne glede na to, ali je jezik posvojen ali razvit kot del projekta, je potrebno vzpostaviti učinkovit proces komunikacije med razvijalci, domenskimi eksperti in uporabniki jezika. Hkrati je potrebno ustrezno planirati nadaljni razvoj in vzdrževanje jezika, ki morata biti pri domensko-specifičnih jezikih hitra in učinkovita.

- *Preveliko zanašanje na domensko-specifične jezike.* Ko razvoj domensko-specifičnih jezikov postane relativno enostaven, obstaja možnost, da se razvijalci začnejo zanašati na vedno nove (in ponavadi slabo razvite) domensko-specifične jezike, ki so med seboj sorodni, a vendarle nekompatibilni. Problem se lahko reši z dobrim vodstvom, komunikacijo in uporabo obstoječih kvalitetnih jezikov.

- *Investicija v jezike in orodja.* Uporaba jezikov in njihovih orodij s časom postane bolj učinkovita, zato so velike spremembe lahko navidez neprivlačne predvsem s poslovnega vidika.

- *Kulturna pristranskost.* Kulturna pristranskost je pogost pojav med razvijalci programske opreme. Ljudje s časom razvijejo ustaljene preference in prepričanja, ki preprečujejo napredek v poteku dela in večje spremembe, kot so implementacija domensko-specifičnega jezika.

2.4 Domensko-specifični jeziki in Ruby

Ruby je splošno-namenski programski jezik, ki je striktno objekten in ima močno podporo za meta-programiranje, kar omogoča preprosto in učinkovito implementacijo notranjih domensko-specifičnih jezikov [5, 14]. Razvil ga je japonski programer Yukihiro "Matz" Matsumoto pred približno dvajsetimi leti, v zadnjih desetih letih pa je njegova popularnost narasla na svetovni nivo [14].

Slika 2.2: Logotip Ruby.

Ogrodje za pisanje spletnih aplikacij Ruby on Rails je eden najbol popularnih domensko-specifičnih jezikov razvitih v programskem jeziku Ruby in je močno prispeval k njegovi popularnosti [14, 15]. Na voljo je tudi vrsta drugih vtičnikov, knjižnic, ogrodij in samostojnih programov, ki se jih lahko programerji na preprost način poslužijo z uporabo programskih paketov, ki se jim kolektivno reče `gems` [16].

Ruby se najpogosteje uporablja v njegovi prvotni implementaciji, ki je napisana v programskem jeziku C, obstajajo pa tudi druge izvedbe, recimo JRuby, ki uporablja Java Virtual Machine (JVM) in se poslužuje funkcionalnosti jezika Java [14].

Pri kreiranju našega domensko-specifičnega jezika smo se odločili za uporabo jezika Ruby, saj nam ta dovoljuje preprost razvoj novega jezika z uporabo programske reflekcije [17] in podajanja blokov kode (angl. *closures*), ima pa tudi močno podporo za procesiranje tekstovnih nizov. Dodatna prednost jezika Ruby je, da za izvajanje uporablja interpreter, kar programerju omogoča hiter razvoj programa brez vmesnih korakov (kot je prevajanje programa v strojno kodo).

Uporaba interpreterja je hkrati glavna hiba jezika Ruby, saj je izvajanje kode (za razliko od programskih jezikov, ki programe najprej prevedejo v strojno kodo) občutno počasnejše. V primeru izdelave našega jezika Ruby služi zgolj kot prevajalnik v končno kodo programa, zato hitrost izvajanja ni tako pomembna.

Poglavje 3

Uporaba DSL v video igrah

Ko je ideja računalniške igre začrtana, je ena prvih odločitev izbira pogona. Ta odločitev omeji vse nadaljne odločitve, vključno z izbiro programskega jezika, v katerem bo igra napisana, in končne platforme.

V današnjih časih je izbira kvalitetnih pogonov za računalniške igre relativno pestra: Unreal Engine [18], CryEngine [19], Unity [20] itd. Velika večina ponuja vsaj en domensko-specifičen jezik, v katerem lahko igro napišemo (zelo popularen je recimo skriptni jezik Lua [21]).

Domensko-specifični jeziki se v računalniških igrah uporabljajo že vsaj dvajset let, kar vsaj na prvi pogled nakazuje, da se njihova uporaba dolgoročno obrestuje. V prejšnem poglavju smo že opisali nekaj splošnih lastnosti domensko-specifičnih jezikov, zanima pa nas, ali je njihova uporaba na tem področju res upravičena. Bolj natančno, ali so prednosti uporabe domensko-specifičnega jezika res nekaj, kar odtehta učenje ali kreiranje novega jezika pri razvoju računalniških iger.

Cilj tega diplomskega dela je razviti domensko-specifičen programski jezik, ki bo omogočal razvoj preproste igre, in primerjati razvoj s tem jezikom z razvojom v splošno-namenskem jeziku. V ta namen bo končna igra razvita dvakrat: prvič izključno v splošno-namenskem jeziku in drugič z novim domensko-specifičnim jezikom.

3.1 Izbira končne platforme

Za razvoj naše igre smo se odločili za programski jezik JavaScript. Grafični in vmesniški elementi so implenetirani v formatu HTML. To nam omogoča skoraj popolno neodvisnost od operacijskega sistema in poskrbi za standarden format, ki je prenosljiv in dolgotrajen.

3.1.1 JavaScript

JavaScript je dinamičen programski jezik, ki se najpogosteje uporablja na Spletu, kjer na uporabniški strani omogoča neposredno interakcijo s spletno stranjo, kontrolo brskalnika, asinhrono komunikacijo in spremembe v predstavitvi spletne strani. JavaScript se te dni uporablja tudi na strežniški strani in kot skriptni jezik v lokalnih aplikacijah, sorodni jeziki iste standardizacije (ECMAScript [10]) pa še v večjem obsegu. S popularizacijo interneta je JavaScript postal eden najpopularnejših programskih jezikov na svetu. Brez uporabe dodatnih vtičnikov ga podpirajo vsi večji brskalniki [11].

3.1.2 HTML5 in CSS3

HTML (HyperText Markup Language) in CSS (Cascading Style Sheets) sta domensko-specifična jezika, ki ju uporabljamo za predstavitev in oblikovanje spletnih strani [12].

HTML ponavadi predstavlja statično vsebino in opiše semantično strukturo spletne strani, hkrati pa podaja iztočnice za grafično predstavitev njenih elementov. Jezik je zapisan v obliki elementov HTML, ki sestojijo iz tako imenovanih *značk* (angl. *tags*) [12].

V svoji peti iteraciji HTML ponuja značko `<canvas>`, ki predstavlja digitalno platno, na katerem lahko izrisujemo grafične elemente.

Za interakcijo s svojimi elementi HTML uporablja specifikacijo DOM. DOM (Document Object Model) je programski vmesnik, ki omogoča dinamično dostopanje do vsebine, strukture in stila dokumentov, ki je neodvisno od platforme in jezika uporabljenega za dostop [13].

CSS je jezik, ki se uporablja za stilsko oblikovanje elementov HTML ali drugih vrst XML dokumentov. Razvit je bil z namenom ločitve vsebine dokumenta od njegove grafične predstavitve, kar zagotavlja vrsto prednosti pri implementaciji [12].

3.2 Zahteve za pogon igre

Za izvedbo pogona računalniške igre je potrebnih nekaj osnovnih gradnikov:

- glavna zanka, ki skrbi za nenehno izvajanje programa,
- uporabniški vmesnik, s katerim uporabnik komunicira z igro,
- fizikalni pogon (fizika gibanja, zaznavanje trkov, itd.),
- funkcije za obravnavanje vhodnih informacij (tipkovnice in miške),
- definicija tipov objektov, ki jih bomo v igri uporabljali.

Pri naprednejših igrah se marsikateri manjši deli razvijejo v samostojne enote. Najbolj očiten primer je grafični pogon, ki nam omogoča učinkovito izrisovanje in animacijo 2D in 3D objektov. V ta namen se pogosto uporabljajo že ustaljene programske knjižnice (na primer DirectX [24] ali OpenGL [25]), ki so ustrezno optimizirane za komuniciranje z grafičnimi karticami.

V primeru nalinijskih iger potrebujemo še del pogona, ki bo upravljal mrežno logiko. Pri igrah z velikim številom igralcov se program velikokrat razdeli na namenski strežnik in odjemalec (dejanska igra), medtem ko je pri kooperativnih igrah z manjšim številom igralcev bolj pogosto, da eden od igralcev služi kot gostitelj in opravlja vlogo strežnika.

Še en pogost element sodobnih pogonov je umetna inteligenca. V zahtevnejših igrah samostojno hranimo algoritme za iskanje poti in logiko agentov (objektov z avtonomnim obnašanjem), na katere se lahko kasneje sklicujemo v kodi.

3.2.1 Game loop

Pri kreiranju računalniške igre najprej potrebujemo glavno zanko (angl. *game loop*), ki se med igro stalno izvaja. Pri tem želimo izvajanje zanke tudi omejeti na stalno hitrost prikazovanja (angl. *frame rate*), kar nam JavaScript dopušča z uporabo funkcije `setInterval()`.

```
1  var Game = { }
2
3  Game.run = function() {
4    Game.update()
5    Game.draw()
6  }
7
8  Game.start = function() {
9    Game._interval = setInterval(Game.run, 1000 / 60)
10 }
11 Game.stop = function() { clearInterval(Game._interval) }
```

Gre seveda za zelo naivno implementacijo pogona, saj kvaliteta dejanske igre tukaj ni ključnega pomena, zato naprednejših funkcionalnosti, kot je recimo preskakovanje okvirjev (kadar izračunavanje logike traja dlje kot to določa hitrost prikazovanja) ne bomo implementirali.

3.2.2 Grafični uporabniški vmesnik

Ker za realizacijo grafičnega dela uporabljamo HTML5, imamo na voljo tudi preprost način izvedbe grafičnega vmesnika in vključitve multimedijskih elementov brez dodatnih knjižnic. Kljub temu, da je podpora za slednje trenutno še vedno odvisna od spletnega brskalnika, je večina sodobnih brskalnikov že v veliki meri skladna s HTML5 standardom [22], zato se s kompatibilnostjo ne bomo obremenjevali.

Za izris grafičnih elementov igre bodo uporabljeni splošni elementi 2D grafičnega konteksta elementa `canvas`. V najnovejših brskalnikih je omogočena tudi uporaba 3D grafike s pomočjo WebGL knjižnice [23], ampak to presega obseg tega dela.

3.2.3 Fizikalni pogon

Teorija zaznavanja trkov v računalniških igrah je zelo obsežna tema. V našem pogonu bomo uporabili samo preproste funkcije za zaznavanje trkov med osnovnimi oblikami, ki so poravnane s koordinatnimi osmi. Napovedi in razreševanje zahtevnejših trkov, ki vključuje simuliranje teže in rotacijo, ne bomo implementirali, saj so te funkcionalnosti v preprostih igrah odveč.

V veliki večini primerov so dovolj čisto enostavne funkcije s katerimi lahko hitro ugotovimo, ali se dva objekta prekrivata. Na spodnjem primeru je prikazana detekcija trka med dvema okroglima objektoma, kjer s Pitagorovim izrekom izračunamo razdaljo med centroma krožnic in jo primerjamo z vsoto njunih polmerov.

```
1 Collider.circleWithCircle = function(b1, b2) {
2   d = Math.sqrt(Math.pow(b2.p[0]-b1.p[0],2) + Math.pow(b2.p[1]-b1.p[1],2))
3   return d <= (b1.r+b2.r)
4 }
```

3.2.4 Vhodne naprave

DOM specifikacija definira vrsto dogodkov uporabniškega vmesnika, ki se prožijo ob pritisku gumbov na vhodnih napravah [26]. Te dogodke lahko z uporabo JavaScripta v brskalniku zaznamo in se na njih ustrezno odzovemo.

Spodaj navedena izseka kode prikazujeta enega osnovnih načinov obravnave gumbov na tipkovnici. V tem primeru ključe gumbov predstavimo kot niz `boolean` vrednosti, ki se spreminjajo glede na to, ali je gumb pritisnjen ali ne. V funkciji `handler` lahko nato vsakega izmed njih preverimo in primerno izvajamo ukaze.

```
1 Input.keymap = [ ]
2 Input.handler = function() {
3   if (Input.keymap[keycode]) {
4     ...
5   }
6 }
```

```
1 document.onkeydown = document.onkeyup = function (e) {
2   Input.keymap[e.keyCode] = e.type == "keydown"
3 }
```

3.2.5 Dedovanje v JavaScriptu

Poseben problem v JavaScriptu je dedovanje. Kljub temu, da je JavaScript objektni programski jezik, ne podpira definicije razredov, kot to omogoča večina sorodnih jezikov. Zato je razredno dedovanje nemogoče, na srečo pa JavaScript omogoča prototipno dedovanje, ki ga lahko simuliramo na spodnji način.

```
1 Function.prototype.inherits = function(parent) {
2   this.prototype = new parent()
3   this.prototype.constructor = this
4   this.prototype.parent = parent.prototype
5   return this
6 }
```

Z razširitvijo prototipa objekta za funkcije lahko dodamo novo funkcijo, ki dinamično doda vse elemente prototipa podanega objekta v objekt, na katerem funkcijo kličemo. Hkrati v prototip dodamo še polje `parent`, če se kasneje želimo sklicevati na starševski objekt.

Mogoče je vredno omeniti, da s spremembo obnašanja jezika in definicijo naravnih funkcij, ki jih lahko verižimo dalje, že definiramo koncepte domensko-specifičnega jezika, čeprav gre v tem primeru zgolj za naključje.

3.2.6 Definicija tipov objektov

Klasičen primer uporabe dedovanja v računalniških igrah je definiranje ponavljajočih tipov objektov in agentov, saj si ti delijo lastnosti, ki so enake za vse sorodne konstrukte.

V spodnjem primeru nakažemo definicijo generičnega okroglega objekta, katerega instanco lahko kreiramo, ko želimo tak objekt postaviti na sceno.

Tip objekta `circle` podeduje vsa polja in funkcije objekta `gameobj`, ki je bil predhodno definiran.

```
function circle(x, y, r) {
  this.p = [x, y]
  this.r = r
  this.type = "circle"
}
circle.inherits(gameobj)
```

Za namen tega diplomskega dela bomo definirali samo objektna tipa `circle` in `rectangle` (slednji je v praksi bolje znan kot "*axis-aligned bounding box*" ali "AABB" na kratko), ki oba podedujeta lastnosti generičnega tipa `gameobj`.

3.3 Kreiranje domensko-specifičnega jezika

Da bo možno v našem jeziku realizirati novo igro, moramo zagotoviti naslednje osnovnr funkcionalnosti:

- definiranje in izvajanje novih funkcij,
- dodajanje in spreminjanje spremenljivk igre,
- kreiranje instanc objektov igre,
- premikanje in izrisevanje objektov,
- definiranje in predvajanje multimedijskih elementov,
- zajemanje elementov HTML na katere se lahko kasneje sklicujemo,
- definiranje funkcij za obravnavanje vhodnih informacij.

Definiranje igralne površine in vmesniških elementov ne bo izvedeno z našim domensko-specifičnim jezikom. HTML in CSS sta že sama po sebi domensko-specifična jezika, ki sta optimizirana za delo z grafičnimi elementi

uporabniških vmesnikov in je z njima najbolje delati neposredno ali z obstoječimi oblikovalskimi orodji, ki so temu namenjena.

Programski jezik Ruby je tudi odlično optimiziran za delo s tekstom, zato bomo v naš namen uporabljali predvsem tekstovne nize, v katerih bomo hranili kodo končnega programa (JS).

3.3.1 Definicija novih funkcij

Tipičen primer funckionalnosti, ki jih Ruby ponuja, je metoda `func`, ki jo uporabljamo za definicijo nove funkcije v igri.

```
1 def func(name, params=[], obj=@context, &block)
2   @func += "#{obj.capitalize}.#{name} = function(#{params.join(", ")}) {\n"
3   if block_given?
4     instance_eval &block
5   else @func += i "// insert code here\n"
6   end
7   @func += "};\n"
8 end
```

To metodo lahko nato uporabimo na tak način:

```
1 func :run do
2   run :update
3   run :draw
4 end
```

Očitno je, da metoda sprejme blok kode kot zadnji parameter. V primeru, da je bil blok podan, ga lahko nato izvedemo na nivoju instance objekta z ukazom `instance_eval`. To nam omogoča, da v bloku izvajamo ukaze, brez da bi se morali eksplicitno sklicevati na objekt.

Še ena posebna lastnost jezika Ruby je sintaksa oblike `#{code}`, ki jo opazimo znotraj nizov. Rečemo ji interpolacija (angl. *interpolation*) in nam dovoljuje ovrednotenje kode znotraj tekstovnega niza, brez potrebe po zaključku tekstovnega niza in njegove združitve z rezultatom zunanje kode.

V primeru uporabe izpostavimo še uporabo simbolov. Simboli so v programskem jeziku Ruby posebna podatkovna struktura, ki se med celotnim izvajanjem programa ne spreminja. Kreiramo jih z uporabo dvopičja (npr. `:run`), vsakemu simbolu pa je poleg podane tekstovne vrednosti določena še numerična vrednost, ki je prav tako konstantna. Ruby hrani posebno tabelo vseh uporabljenih simbolov. Kazalec na simbol vedno kaže na isto mesto v tej tabeli, ne glede na kontekst sklicevanja na simbol. Vrednosti simbola je nemogoče sprementi; simbol `:s` bo imel vedno enako tekstovno in numerično predstavitev.

3.3.2 Dodajanje in spreminjanje spremenljivk

Za delo s spremenljivkami igre smo izvedli dve metodi: `vars` in `set`. Metoda `vars` nam omogoča hitro tvorbo serije spremenljivk na kontekstnem objektu z uporabo zbirke `hashmap`. Z metodo `set` lahko vrednosti spremenljivk spreminjamo na poljuben način.

```
1 vars(
2   x: 3,
3   y: 14
4 )
5 func :bawk do
6   set :x, "chicken"
7 end
```

3.3.3 Funkcije za kontrolo teka in testiranje trkov

Za kreiranje novih funkcij smo dodali metodo `func`, ki smo jo že izpostavili v prejšnem delu, za bolj podrobno delo z ukazi znotraj nove funkcije pa smo dodali še nekaj pomožnih metod (za pogojne stavke, zanke in vračanje rezultatov: `sif`, `else_if`, `loop` in `returns`) ter metodo za testiranje trkov med objekti (`collide`). Za izvajanje funkcij smo dodali metodo `run`.

```
func :beep do
  loop "i", 0, 3 do
    play :beep
  end
end
```

```
func :boop do
  sif (collide o(:ball) & o(:brick)) do
    run :beep
    returns true
  else_if
    returns false
  end
end
```

3.3.4 Funkcije za delo z objekti

Metodi `create` in `spawn` skrbita za definiranje novih objektov igre in njihovo postavitev na sceni. Metoda `create` uporablja programsko reflekcijo s katero poskrbi, da je nov objekt ustreznega tipa. To nam omogoča dodajanje novih tipov objektov brez dodatnih sprememb v kodi.

```
create :ball, :circle, [10, 50, 5]
func :setup do
  spawn :ball
end
```

Za premikanje in izrisovanje obstoječih objektov igre smo dodali preprosti metodi `move` in `draw`, ki na objektu kličeta funkciji `move()` ali `draw()`.

```
func :thwack do
  move :ball
  draw :ball
end
```

3.3.5 Funkcije za delo s HTML elementi

Ker bomo od multimedijskih vsebin uporabljali zgolj zvočne datoteke, smo v naš jezik dodali samo metodi `audio`, ki zajame seznam audio elementov v dokumentu HTML, in `play`, ki zajete elemente lahko predvaja.

```
audio("smack","pow","crack")
func :punch do
  play :smack
end
```

Za zajemanje drugih elementov HTML smo kreirali metodo `elements`.

```
elements("buffer","gui","info")
```

3.3.6 Zajemanje vhodnih informacij

Podatke iz vhodnih naprav lahko obravnavamo z metodo `input`, ki sprejme numerično kodo pritisnjene tipke in blok kode, ki ga želimo ob pritisku izvesti.

```
input 27 do
  run :reset
end
```

3.3.7 Pomožne metode

Nenazadnje smo dodali še nekaj pomožnih metod, s katerimi si lahko pomagamo pri implementaciji iger. Te so:

- `context:` izrecno določi objekt za sklic.
- `x:` poda tekstovni niz kot izraz kode (*"expression"*).
- `o:` poišče objekt s podanim ključem v zbirki (*"object"*).
- `i:` doda ustrezen zamik v kodi (*"indent"*).

- `code:` doda tekstovni niz neposredno v kodo.
- `write:` zapiše celoten program v datoteko.

3.4 Od izbire jezika do izdelave igre

Metode in strukture, ki smo jih implementirali, predstavljajo jedro našega jezika. Jezik je seveda moč še poljubno razširiti (tako kot pogon), a za potrebe igre, ki jo bomo razvili, je to odveč.

Z razvitim pogonom in jezikom se lahko lotimo izdelave igre. Podrobnosti o izdelavi igre bomo obdelali v naslednjem poglavju. Za začetek shranimo izdelan jezik v posebno datoteko, ki bo kasneje vključnena kot knjižnica v kodi naše igre z ukazom `require_relative` (ali `require`, če se na datoteko ne želimo sklicevati iz konteksta našega programa) kot je prikazano spodaj.

```
1 require_relative "GameDSL"
```

Uporaba končnice datoteke (`.rb`) je v tem primeru nepotrebna in se jo navadno izogibamo, saj jo Ruby tekom izvajanja razbere iz vsebine mape.

Shranjena datoteka (v našem primeru `GameDSL.rb`) skupaj z jezikom Ruby služi kot naša jezikovna zasnova, prav tako pa tudi kot del izvajalnega pogona v kombinaciji z razčlenjevalnikom jezika Ruby.

Poglavje 4

Implementacija igre

Uporabo našega domensko-specifičnega bomo prikazali pri implementaciji igre Breakout. Breakout je klasična igra, nastala izpod rok industrijskih pionirjev (Bushnell, Bristow in Wozniak) zaposlenih pri podjetju Atari, ki je izšla na trg daljnega leta 1976. Originalna igra je bila ralizirana z uporabo logičnih vezij v času, ko mikroprocesorji še niso prišli v splošno rabo [27].

4.1 Pravila igre

Osnovna pravila igre Breakout so preprosta:

1. Igralec nadzoruje *plošček* na dnu površine, s katerim odbija *žogico*.
2. Zgornji del ekrana prekrivajo uničljive *opeke*.
3. Opeke se razbijejo, ko se žogica od njih odbije.
4. Če žogica pade na tla, igralec izgubi življenje.
5. Cilj je razbiti vse opeke.
6. Igra se konča predčasno, če igralcu zmanjka življenj.

Ena od posebnih lastnosti večine iger stila Breakout je ta, da se žogica od ploščka ne odbija v skladu z zakoni fizike, ampak sta njen odbojni kot in

hitrost določena glede na lokacijo trka s ploščkom. S tem lahko natančno nadziramo pot žogice in se izognemo večini primerov, ko bi se žogica odbijala vedno pod istim kotom.

4.2 Tipi objektov

Za našo igro torej potrebujemo tri tipe objektov: žogico, plošček in opeko. Opeke bomo razdelili v pet vrstic po dvajset opek z različnimi barvami, da jih lahko med seboj ločimo. Ploček bomo postavili iz začetka na sredino tal, žogico pa nad njim pod kotom.

```
create :paddle, :rectangle, [340,585,120,15]
create :ball, :circle, [320,400,8]
# koordinate v obliki izraza omogočajo dinamično postavitev
create :bricks, :rectangle, ["j*40", "50+i*30", 40, 30]
```

4.3 Pogoji za zmago in izgubo

Če želimo preveriti, kdaj se igra konča, moramo slediti tako številu igralčevih življenj kot opek na sceni. Implementirati moramo funkciji, ki preverita pogoje za zmago in izgubo.

```
context :player
vars(lives: 5)

context :game
vars(
  bnum: 100,
  colors: ["red","orange","yellow","green","blue"]
)

func :victory do
  returns "(Game.bnum <= 0)"
end
func :loss do
  returns "(Player.lives <= 0)"
end
```

4.4 Posodabljanje in izrisevanje stanja igre

Za poganjanje igre potrebujemo osnovno zanko in implementacijo funkcij `update()` in `draw()`, ki sta klicani v njej. V funkciji `update()` moramo preveriti informacije iz vhodnih naprav, premakniti objekte in preveriti, ali se objekti trkajo med seboj ali z zidovi. Vse trke moramo nato tudi razrešiti.

```
1  func :run
2    run :update
3    run :draw
4  end
5
6  func :update do
7    run :handler, [], :input
8    move :paddle
9    move :ball
10   sif (collide o(:ball) & o(:paddle)) ...
11 end
12
13 func :draw do
14   run :clear
15   draw :paddle
16   draw :ball
17   ...
18 end
```

4.5 Postavitev objektov

Najprej moramo vse objekte postaviti na sceno, kar med drugim storimo v funkciji `setup()`.

```
1  func :setup do
2    spawn :paddle
3    spawn :ball
4    loop "i", 0, 5 do
5      loop "j", 0, 20 do
6        spawn :bricks, "bricks[i][j]"
7      end
8    end
9  end
```

4.6 Nadzor igre

Za nadzor igre moramo dodati funkciji `start()` in `stop()`.

```
1  func :start do
2    sif "!Game.running" do
3      set :_interval, x("setInterval(Game.run, 1000 / Game.FPS)")
4    end
5  end
6
7  func :stop do
8    sif "Game.running" do
9      run :clearInterval, [x("Game._interval")], nil
10   end
11 end
```

4.7 Nadzor ploščka

Igre ni mogoče igrati brez nadzora ploščka zato dodamo še funkciji za spremljanje vnosa s tipkovnice.

```
1  input 37 do # levo
2    set "paddle.p[0]", -5, "+="
3  end
4  input 39 do # desno
5    set "paddle.p[0]", 5, "+="
6  end
```

V tem delu kode poskrbimo, da se plošček ustrezno premika, ko pritisnemo gumb ← (koda 37) ali → (koda 39).

4.8 Izvajanje in testiranje

Naš program lahko prevedemo v končno kodo preko konzole s preprostim ukazom na kateremkoli sistemu, kjer je nameščen Ruby.

```
1  ruby breakout.rb
```

4.8.1 Izvajanje programa

Da lahko zaženemo našo igro, moramo pridobljeno JavaScript datoteko vstaviti v naš HTML dokument, ki je za namen osnovnega testiranja lahko implementiran na zelo preprost način.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Breakout</title>
</head>
<body>
  <h1>Breakout</h1>
  <div id="viewport">
    <div id="gui_info"></div>
    <canvas id="game" width=800 height=600></canvas>
  </div>
  <button id="start" type="button" onclick="Game.start()">Start</button>
  <button id="stop" type="button" onclick="Game.stop()">Stop</button>

  <script type="text/javascript" src="js/breakout.js"></script>
</body>
</html>
```

Kot je razvidno iz kode, smo kreirali element `canvas`, na katerem se bo igra izrisovala, dodaten element `div`, v katerega lahko zapisujemo informacije o igri in dva gumba, s katerima bomo klicali naši funkciji `start` in `stop`. Skripta programa je dodana na konec, da se lahko sklicujemo na predhodnje elemente dokumenta.

4.8.2 Testiranje

Kljub temu, da je določene manjše enote v računalniških igrah možno testirati z avtomatskimi testi (in je to pogosto tudi priporočeno [28]), se večina testiranja tradicionalno izvaja z neposrednim testiranjem programa.

Ena od prednosti implementacije igre v spletnem brskalniku so vgrajene funkcionalnosti (recimo JavasScript konzola in razhroščevalnik), ki jih lahko najdemo v večini modernih brskalnikov. Na voljo so tudi dodatni vtičniki,

kot je recimo vtičnik Firebug za brskalnike tipa Firefox [29], ki še nadalje razširijo sposobnosti razhroščevanja kode.

Za namen testiranja enot jezik Ruby ponuja knjižnico `Test::Unit`. Za JavaScript v ta namen obstaja vrsta neuradnih knjižnic, kot je naprimer QUnit [30].

4.9 Rezultati dela

Zgornji izseki kode vsekakor ne predstavljajo celoten program, temveč le okvirno sliko potrebnih korakov za izvedbo igre. Celotno izvedbo si je moč ogledati v testnem primeru, ki je zapakiran skupaj s kodo domensko-specifičnega jezika [31].

Za primerjavo izvedbe v programskem jeziku JavaScript in izvedbe v narejenem domensko-specifičnem jeziku si lahko ogledamo tabelo 4.1.

	JavaScript	**DSL**
dolžina kode (št. znakov)	7405	4759
čas za razvoj jezika (≈)	0 h	60 h
čas za razvoj igre (≈)	40 h	8 h

Tabela 4.1: Lastnosti izvedb.

Iz tabele je razvidno, da se je dolžina programa v primeru domensko-specifičnega jezika opazno zmanjšala. To lahko predvsem pripišemo enostavnejši sintaksi in povečani izraznosti kode.

Kljub temu, da se je količina časa za razvoj igre v domensko-specifičnem jeziku očitno zmanjšala na zgolj petino časa porabljenega za prvotni razvoj igre, je potrebno vzeti v zakup znanje pridobljeno med samim razvojem. Na porabljen čas nedvomno vplivata dolžina in zahtevnost kode, a za bolj natančno statistiko zmanjšnanja časa, bi bilo poskus potrebno ponoviti večkrat s pomočjo več razvojnikov.

Končna koda programa je v obeh primerih identična zato med njima ni performančnih razlik. Ker gre za relativno preprosto igro, je količina časa potrebnega za prevajanje kode prav tako zanemarljiva.

Končan program izgleda približno tako kot je prikazano na sliki 4.1, kjer je igralec že uspešno razbil približno tretjino opek.

Slika 4.1: Primer poteka igre.

Poglavje 5

Sklepne ugotovitve

5.1 Analiza rešitve

Z uporabo našega domensko-specifičnega jezika, se je koda igre skrajšala za več kot tretjino. Zasledili smo tudi opazno zmanjšanje števila tako sintaktičnih kot semantičnih napak, deloma po zaslugi lažje berljive kode in zmanjšanja količine (potrebnih) ločil.

Z vidika uporabnika je bilo delo v domensko-specifičnem jeziku hitrejše in bolj tekoče. Seveda, če vzamemo v zakup čas, ki je bil namenjen razvoju novega jezika, je bila celotna količina vloženega časa in sredstev v našem primeru dejansko daljša kot pri "klasični" implementaciji. Za boljši izkoristek domensko-specifičnega jezika bi se morali lotiti večjega projekta ali pa več manjših projektov.

Razvijalec domensko-specifičnega jezika je med razvojom prisiljen v bolj podrobno razmevanje domene problema (v našem primeru razvoja iger), kar pozitivno vpliva na njegove sposobnosti razvoja na tem področju. Prav tako implementacija učinkovitega izvajalnega pogona od razvijalca zahteva dobro optimizacijo kode in prispeva k uporabi primernih metod in prijemov.

To vsaj površinsko nakazuje na prednosti, ki smo jih navedli v drugem poglavju. Ena od že omenjenih glavnih slabosti, je bila količina časa za razvoj novega jezika za razvijalca, ki na tem področju ni imel predhodnih izkušenj.

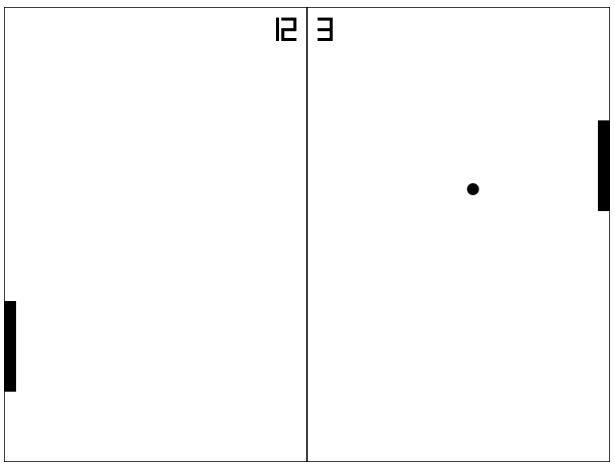
Ker smo se odločili za izdelavo notranjega domensko-specifičnega jezika, je bilo potrebnih tudi nekaj kompromisov, da bi bila implementacija v jeziku Ruby sploh mogoča, predvsem zaradi striktne strukture, ki ji je potrebno slediti.

Mnenje avtorja je, da je uporaba domensko-specifičnih programskih jezikov pri razvoju računalniških iger upravičena in priporočena, kadar je to le mogoče, pod pogojem, da gre za delo na večjih projektih (kot je navadno pri izdelavi modernih iger) in je izvajalni pogon ustrezno optimiziran na tak način, da končni program ne vpliva bistveno na performanse igre.

5.2 Nadaljni razvoj in možne izboljšave

5.2.1 Razvoj drugih iger

Kljub relativni preprostosti implementiranega jezika so osnovni gradniki na zadovoljivem nivoju, zato je z njim mogoče razviti tudi druge igre primerljive zahtevnosti. En tak primer je igra Pong [32], katere implementacija z našim domensko-specifičnim jezikom je prikazana na sliki 5.1.



Slika 5.1: Klon igre Pong.

5.2.2 Avtonomnost jezika

Ena od glavnih izboljšav jezika je njegova avtonomnost. Če bi se želeli v celoti ločiti od končnega jezikat, bi v idealnem primeru lahko dinamično kreirali zbirko vseh konstruktov, ki se v programu pojavljajo. Na njih bi se neposredno sklicevali v kodi, generiranje programa pa bi bilo izvedeno v končni fazi z uporabo reflekcije in pametnih transformacij.

5.2.3 Zahtevnost pogona

V zadnjih štiridesetih letih odkar je bila igra Breakout prvič izdana na tržišče, je zahtevnost računalniških iger (in njihovih pogonov) skupaj z zahtevnostjo strojne opreme naraščala skoraj eksponentno [33, 34].

Predstavljeni pogon je tako zelo primitivna realizacija, neprimerljiva s pristopi sodobnih pogonov. Njegove komponente bi lahko poljubno razširili in izboljšali brez večjih sprememb v osnovnem domensko-specifičnem jeziku. Seveda pa bi morali jezik ustrezno razviti naprej s pogonom, da bi nam ta omogočal uporabo naprednejših funkcionalnosti.

Končni izdelek tega dela je primeren zgolj za izhodiščno točko, iz katere bi lahko začeli z razvojem sodobnega pogona in jezika, ki bi na njem temeljil.

Literatura

[1] Unity: Creating and Using Scripts [Online]. Dosegljivo: http://docs.unity3d.com/Manual/CreatingAndUsingScripts.html.

[2] RPG Maker VX Ace [Online]. Dosegljivo: http://www.rpgmakerweb.com/products/programs/rpg-maker-vx-ace.

[3] Programming language članek na Wikipediji. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Programming_language.

[4] M. Voelter, "DSL Engineering: Designing, Implementing and Using Domain-Specific Languages", 2013.

[5] M. Fowler, "Domain-Specific Languages", 2010.

[6] SQL članek na Wikipediji. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/SQL.

[7] T. Parr, "Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages", 2010.

[8] D. Spinellis, "Notable design patterns for domain specific languages", v zborniku: *Journal of Systems and Software*, 56(1):91–99, februar 2001.

[9] M. Mernik, J. Heering, A. M. Sloane, "When and how to develop domain-specific languages", v zborniku: *ACM Computing Surveys*, 37(4):316–344, 2005.

[10] ECMAScript. [Online]. Dosegljivo: http://www.ecmascript.org/.

[11] JavaScript članek na Wikipediji. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/JavaScript.

[12] W3 Standards for Web Design. [Online]. Dosegljivo: http://www.w3.org/standards/webdesign/.

[13] W3C Document Object Model. [Online]. Dosegljivo: http://www.w3.org/DOM/.

[14] About Ruby. [Online]. Dosegljivo: https://www.ruby-lang.org/en/about/.

[15] Ruby on Rails. [Online]. Dosegljivo: http://rubyonrails.org/.

[16] RubyGems.org. [Online]. Dosegljivo: https://rubygems.org/.

[17] Reflection, ObjectSpace, and Distributed Ruby. [Online]. Dosegljivo: http://phrogz.net/programmingruby/ospace.html

[18] Unreal Engine. [Online]. Dosegljivo: https://www.unrealengine.com/.

[19] CryEngine. [Online]. Dosegljivo: http://cryengine.com/.

[20] Unity - Game Engine. [Online]. Dosegljivo: https://unity3d.com/.

[21] Lua: about. [Online]. Dosegljivo: http://www.lua.org/about.html.

[22] HTML5 Test. [Online]. Dosegljivo: https://html5test.com/.

[23] WebGL - OpenGL ES 20.0 for the Web. [Online]. Dosegljivo: https://www.khronos.org/webgl/

[24] Developing games - DirectX. [Online]. Dosegljivo: http://msdn.microsoft.com/directx.

[25] OpenGL - The Industry Standard for High Performance Graphics. [Online]. Dosegljivo: http://www.opengl.org/.

[26] UI Events. [Online]. Dosegljivo: http://www.w3.org/TR/DOM-Level-3-Events/.

[27] Breakout članek na Wikipediji. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Breakout_(video_game)

[28] N. Llopis, Stepping Through the Looking Glass: Test-Driven Game Development (Part 1). [Online]. Dosegljivo: http://gamesfromwithin.com/stepping-through-the-looking-glass-test-driven-game-development-part-1.

[29] Firebug. [Online]. Dosegljivo: http://getfirebug.com/.

[30] QUnit. [Online]. Dosegljivo: https://qunitjs.com/.

[31] Arhiv z rezultati dela in testnim primerom. [Online]. Dosegljivo: https://www.dropbox.com/s/mvtgm04wcxvrtv0/GameDSL.zip?dl=1.

[32] Pong-Story: Welcome. [Online]. Dosegljivo: http://www.pong-story.com/.

[33] R. Stanton, "A Brief History of Video Games", 2015.

[34] Moore's law članek na Wikipediji. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Moore's_law.