Šolski center Novo mesto

Srednja elektro šola in tehniška gimnazija

Šegova ulica 112

8000 Novo mesto

Vesoljski strelec

(Maturitetna seminarska naloga)

Predmet: Računalništvo

Avtor: Gregor Bučar

Razred: T4A

Mentor: Albert Zorko, univ. dipl. inž.

Šentjernej, april 2019

POVZETEK

Seminarska naloga opisuje izdelavo igre v višjem programskem jeziku Java ter v igralnem pogonu Unity. Pri tem so pri izdelavi v Javi predstavljene funkcije igralne figure, funkcije sovražnikov, uporabniški vmesnik, izrisovanje grafike in razredi, s katerimi upravljamo delovanje celotnega algoritma, pri izdelavi v igralnem pogonu pa so predstavljeni grafični urejevalnik, objekt igralne figure, skrbniki igre, uporabniški vmesnik in vizualni ter zvočni efekti.

V prvem delu naloge predstavim zbiranje potrebnih sredstev za izdelavo videoigre v Javi in prav tako v igralnem pogonu, izbiro integriranega razvojnega okolja za obe verziji igre in pa uporabo potrebnih paketov pri izdelavi v Javi ter izbiro programskega jezika pri izdelavi v Unityju.

V drugem delu pa podrobno prikažem delovanje posameznih podenot celotnega algoritma v Javi in nato tudi še okvirno opišem delovanje igre, ustvarjene s pomočjo igralnega pogona ter podrobno predstavim možnosti, ki jih grafični urejevalnik Unity ponuja.

Ključne besede: Java, Unity, zbiranje sredstev, uporabniški vmesnik, funkcije igralne figure, funkcije sovražnikov, izrisovanje grafike, skrbniki igre, zvočni efekti, vizualni efekti.

KAZALO VSEBINE

KAZALO SLIK

1 Uvod

Poglavitni namen seminarske naloge je predstaviti proces izdelovanje videoigre brez uporabe orodij za pomoč pri izdelavi igre ter tudi izdelavo s pomočjo igralnega pogona. Dandanes igranje videoiger predstavljaj velik delež načina sprostitve in oddiha. Z izdelavo videoiger se lahko vsak posameznik, za večje projekte pa je le-ta ni dovolj, zato jih ustvarja več posameznikov skupaj. Moj namen je prikazati izdelavo igre v celoti od ideje do končnega produkta, pri tem pa predstaviti dele, ki jih v večjih projektih opravljajo vodja projekta, programerji, grafični oblikovalci in zvočni oblikovalci. Seminarska naloge je razdeljena na dva glavna dela in sicer na pripravo na izdelavo ter izdelavo igre.

Cilji, katere sem si zastavil pri tem so, da naredim funkcionalni igri v višjem programskem jeziku in nato ustvarim duplikat igre s pomočjo igralnega pogona. Predvsem je cilj tega ugotoviti, kako uporaba orodij za pomoč pri izdelavi videoigre vpliva na kvaliteto končnega produkta. Pri izdelavi projekta sem zbiral veliko podatkov, da sem igro lahko ustvaril, predvsem pa sem igro neprestano testiral, da sem jo lahko optimiziral in odpravil probleme.

2 Priprava na izdelavo

Preden sem se lotil izdelave igre sem moral izbrati programski jezik, v katerem bi nato razvijal igro brez uporabe kakršnihkoli pripomočkov. Po tem sem moral izbrati urejevalnik (integrirano razvojno okolje), v katerem bi algoritme napisal in testiral delovanje ter preučil katere pakete (znotraj razvijalnega okolja) bom moral uporabiti za delovanje igre. Za testiranje algoritmov pa sem potreboval tudi slike, s katerimi bi njihovo delovanje vizualno predstavil.

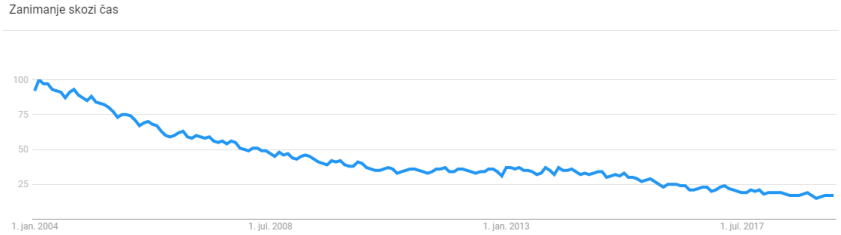
Ko je bila igra v tem programskem jeziku ustvarjena, sem se lotil še razvijanja identične igre (vizualno) s pomočjo igralnega pogona (potrebna izbira igralnega pogona). Preden pa sem s tem začel sem moral prav tako izbrati programski jezik, ki ga igralni pogon podpira, ter tudi razvojno okolje, v katerem bi algoritme zapisoval (torej podobno kot pri izdelavi igre brez eksternih knjižnic). Obenem sem tudi tukaj potreboval slike za grafični prikaz igre in poleg teh še zvočne posnetke, ki bodo v igri služili kot zvočni efekti.

2.1 Programski jezik Java

Za izdelavo igre brez posebnih programskih knjižnic in orodij, ki bi izdelavo olajšale, sem izbral programski jezik Java, saj sem v obdobju šolanja ta jezik najbolj podrobno spoznal. Jezik je bil razvit v podjetju Sun Microsystems (leta 2010 prodano Oraclu) in sicer so ga razvili James Gosling in njegovi sodelavci. »Java je objektno usmerjen, prenosljivi programski jezik, ki deluje na različnih platformah (Windows, Mac, Linux, Raspberry Pi, itd.). « (1)

Programski jezik je dokaj preprost za učenje, enostaven za uporabo, in zastonj. »Lahko se uporablja za izdelavo mobilnih aplikacij (za Androide), namiznih in spletnih aplikacij, predvsem pa se uporablja v aplikacijah kjer je upravljanje s pomnilnikom avtomatizirano.« (2) Z uporabo Jave je mogoče tudi ustvariti igro in delati s podatkovnimi bazami, predvsem pa je pomembno, da ima ogromno podporo skupnosti, saj se več milijonov razvijalcev ukvarja s tem programskim jezikom.

Na sliki 1 je prikazana priljubljenost (iskanja) programskega jezika Java za ves svet zadnjih 15 let. Zanimanje za jezik v poljubnem letu se izračuna glede na najvišjo doseženo zanimanje, ki ga je le-ta dosegel v tem obdobju. Iz slike je razvidno, da programski jezik trenutno ni v trendu, saj se zanimanje za iskanje zanj zmanjšuje, vendar je še vedno v široki uporabi zaradi prednosti pred drugimi programskimi jeziki pri določenih problemih.



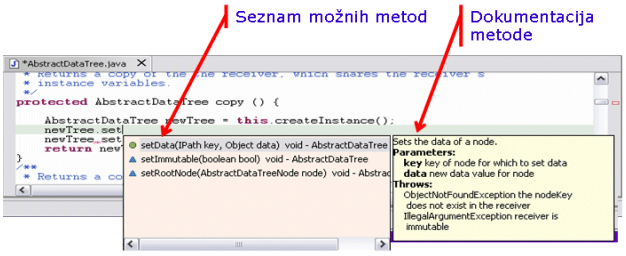
Slika 1: Prikaz zanimanja za iskanje programskega jezika Java v obdobju 2004 – 2019
https://trends.google.com/trends/explore?date=all&q=%2Fm%2F07sbkfb

2.1.1 Izbira integriranega razvojnega okolja

»Integrirano razvojno okolje, tako imenovani IDE (Integrated Development Environment) je programsko orodje, ki programerjem pomaga pri razvoju. Običajno vsebujejo urejevalnik izvorne kode, prevajalnik oziroma tolmač (interpreter), orodje za avtomatizacijo izgradnje programa in (običajno) razhroščevalnik.« (3) Pri izdelavi igre v programskem jeziku Java sem izbral razvojno okolje Eclipse, saj je med prosto dostopnimi okolji eno izmed najbolj priljubljenih, prav tako pa ga lahko uporabljamo za pisanje več različnih programskih jezikov.

»Eclipse je napisan v Javi in C, njegova primarna uporaba pa je namenjena razvijanju aplikacij s programskim jezikom Java. Uporablja se lahko tudi za razvoj aplikacij v drugih programskih jezikih prek vtičnikov« (4) Prvič je bil izdan 7. novembra 2001 in je bil večkrat posodobljen, najnovejša stabilna verzija pa je 4.11 (v času pisanja naloge), ki je bila izdana 20. marca 2019.

To razvojno okolje sem izbral zato, ker omogoča tudi sprotno pomoč, javlja sprotne sintaktične napake, pri pisanju predlaga imena in omogoča avtomatsko dokončevanje stavka, poleg tega pa ima tudi možnost odpiranja dela dokumentacije, vezane na priporočene funkcije ali polja, kot vidimo na sliki 2. Veliko časa pa nam prihrani tudi razhroščevalnik, saj v primeru semantične napake z njegovo pomočjo hitreje ugotovimo mesto napake in jo odpravimo.



Slika 2: Primer predlaganja in dokumentacije funkcij
http://colos.fri.uni-lj.si/ERI/RACUNALNISTVO

2.1.2 Izbira in uporaba paketov

Pri izdelavi igre v programskem jeziku Java s pomočjo razvojnega okolja Eclipse sem uporabil osnovne pakete oziroma potrebne razrede, nujne za ustvarjanje okna, izrisovanje grafike na zaslonu, računanje ter za upravljanje z dogodki. Med temi paketi sta *javax* (*javax.swing* in *javax.imageio*) in *java* (*java.util* ter *java.awt*). Na sliki 3 vidimo kateri paketi so uvoženi s ključno besedo *import*, tej pa sledi *Ime_Paketa.Ime_PodPaketa.Razred;*.

```
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.image.BufferedImage;
import java.util.ArrayList;

import javax.imageio.ImageIO;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.Timer;
```

Slika 3: Prikaz uvoženih paketov oziroma razredov

V paketu *java* sem potreboval nekatere stvari znotraj paketa *awt*. »AWT (Abstract Window Toolkit) je vmesnik za programiranje aplikacij za razvoj grafičnih vmesnikov ali okenskih aplikacij v Javi.« (5) Paket *java.awt* vsebuje tudi paket *event*, znotraj le-tega pa imamo razrede, ki se uporabljajo pri delu z dogodki. Razredi, ki sem jih uporabil jaz so *ActionEvent*, *KeyEvent*, *MouseEvent*, *ActionListener*, *KeyListener*, *MouseListener* ter *MouseMotionListener*. Razred *KeyListener* vsebuje funkcije potrebne za pravilno delovanje igre, ko igralec pritisne določeno tipko, Razred *ActionListener* vsebuje funkcijo, katero uporabljam za določanje časovnega intervala slikanja, *MouseListener* in *MouseMotionListener* pa vsebujeta nekatere funkcije, katere potrebujem za pravilno delovanje igre ko igralec povzroča dogodke z miško. Razredi *ActionEvent*, *KeyEvent* in *MouseEvent* pa so tipi podatkov, ki jih dobimo, ko se določen dogodek zgodi. Polet teh razredov so v uporabi še razredi *Color*, *Font* in *Graphics* (znotraj paketa *java.awt*) za risanje grafike ter določanje fonta in barve. Znotraj paketa *util* pa potrebujem razred *ArrayList*, saj z njim določam koordinate vseh objektov.

V paketu *javax* potrebujem nekatere stvari znotraj paketa *swing*. »Swing je del temeljnih razredov Java (JFC), ki se uporabljajo za ustvarjanje okenskih aplikacij. Zgrajen je na vrhu AWT vmesnika za programiranje aplikacij in v celoti napisan v Javi.« (6) V tem paketu potrebujem razrede *JFrame* (ustvarjanje okna), *JPanel* (podlaga, na katero izrisujem grafiko) ter *Timer* (potrebujem za določanje časovnih intervalov risanja grafike in za druge funkcije). Poleg paketa *swing* pa je nujen tudi paket *imageio*, ki vsebuje razred *ImageIO*, le-tega pa uporabljamo za uvoz slik in posledično tudi za njihovo risanje (posredujemo kot argument).

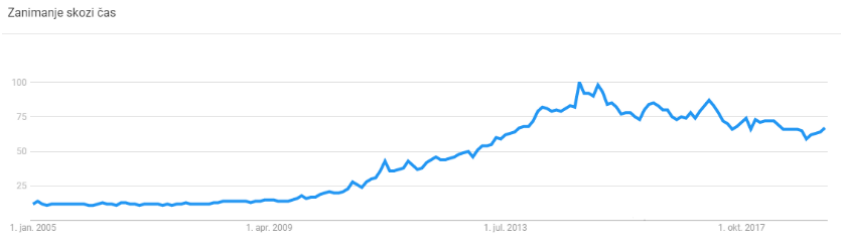
2.2 Igralni pogon Unity

Za igralni pogon za izdelavo igre, katero bom primerjal z igro izdelano v programskem jeziku Java, sem si izbral Unity. »Unity (splošno znano kot Unity3D) je igralni pogon in hkrati tudi integrirano razvojno okolje za ustvarjanje interaktivnih medijev, običajno video iger.« (7) Prvotna verzija Unityja (1.0.0) je bila ustvarjena na Danskem in sicer so bili razvijalci David Helgason, Joachim Ante in Nicholas Francis, ki so sprva ustanovili podjetje Over The Edge. Njihov cilj je bil ustvariti cenovno dostopen igralni pogon, ki bi s profesionalnimi orodji drugim razvijalcem igre omogočil enostavnejše in hitrejše ustvarjanje. Lahko se uporablja za izdelavo 2D ter prav tako 3D iger in tudi za simulacije za številne platforme. V času pisanja je igralni pogon na voljo v štirih možnostih licenciranja, od tega je ena zastonj (Personal), tri pa so na voljo skozi mesečno plačevanje (Plus, Pro in Enterprise). Nekatere funkcije, ki jih igralni pogon nudi so na voljo le v verzijah, kjer je potrebno mesečno plačevanje, vendar je verzija, ki je na voljo zastonj še vedno idealna za učenje.

»Unity Technologies, predhodno znan kot Over The Edge, je podjetje za razvoj video iger, ki je najbolj znano po razvoju Unityja (licenciranega igralnega motorja).« (8) Sprva se je podjetje spopadalo s finančnimi težavami, saj so se morebitni kupci bali, da bi njihov mehanizem propadel kot se je to dogajalo drugim podjetjem za razvoj videoiger. »Mnoga podjetja, ki so se ukvarjala z razvojem videoiger so v tistem času ustvarile igro, ki se nato ni prodajala dobro, zato so posledično odvrgli svoj igralni pogon.« (8) Šele čez 2 leti je podjetje dokazalo, da bo Unity bil ustrezno posodobljen in podprt (verzija 2.0). Skozi leta je bil igralni pogon velikokrat nadgrajen, leta 2008 so zaradi rasti pametnih telefonov začeli razvijati verzijo Unityja, ki bi podpirala objavljanje iger za iPhone.

V času izdaje verzije 3.0 leta 2009 je bilo registriranih že več kot 200,000 razvijalcev. »Unity je postal #1 igralni pogon za izobraževanje in najbolj priljubljena tehnologija na mobilnih platformah« (9) Različica 4.0 je postala dostopna za prenos 13. novembra 2012, nekatere nadgradnje pa so bile: dodan sistem animacije (Mecanim), podpora za Adobe Flash in DirectX 11. Naslednja različica Unityja, 4.3 (12. november 2013), je vključevala podporo za razvoj 2D iger. Verzija Unity 5.0 je med svojimi posodobitvami igralnega pogona vsebovala nadgradnje avdia in osvetljevanja, prav tako tudi pa so bili tudi predstavljeni kinematografski učinki slike (pomagajo pri izgledu slike – manj generično).

»Do leta 2016 naj bi bilo registriran več kot 5.5 milijona uporabnikov.« (10) Po verzija 5 pa je sledila različica Unity 2017, ki je spet vsebovala veliko novega (Timeline, Cinemachine, pogon za upodabljanje grafike v realnem času in več). »Unity 2018 je vključeval virtualno resničnost, razširjeno resničnost in mešano resničnost, prav tako pa tudi orodja za strojno učenje, kot je imitacija učenja, kjer se igre učijo iz resničnih navad igralcev in predloge za nove razvijalce.« (11). Nadgradnje in razvoj so tudi vidne skozi zanimanje za iskanje na sliki 4, saj odražajo, da je bil igralni pogon od datuma svoje prve izdaje pa do danes nadvse razvit in posodobljen. Iz slike 4 je tudi razvidno, da je izraz Unity (mehanizem videoigre) dandanes še zmeraj zelo aktualna iskalna tematika.



Slika 4: Prikaz zanimanja za iskanje izraza Unity od 2005 do 2019
https://trends.google.com/trends/explore?date=2005-01-01%202019-04-13&q=%2Fm%2F0dmyvh

2.2.1 Izbira programskega jezika

V času, ko sem igralno okolje Unity prenesel na svoj računalnik let 2016, sta bila na voljo 2 programska jezika in sicer C# in verzija JavaScripta. »Preden je C# postal primarni programski jezik, je pred tem igralni pogon podpiral Boo (imel sintakso podobno programskemu jeziku Python, uporabljajo ga je zelo malo ljudi), ki je bil odstranjen v izdaji Unity 5, in različico JavaScripta, imenovana UnityScript, ki je bila zastarela avgusta 2017 po izdaji nove verzije Unity 2017.1 v korist C#.« (11)

Torej v času ustvarjana igre sem imel na voljo programski jezik C#, ki je v mojem mnenju programski jezik, ki se ga je nadvse lahko naučiti s predhodnim znanjem Jave. »Jezik naj bi bil preprost, sodoben, splošen namenski, objektno usmerjen programski jezik.« (12) C# je zasnoval Anders Hejlsberg skupaj s svojo ekipo v podjetju Microsoft, najnovejša verzija v času pisanja pa je C# 7.3, ki je bila objavljena skupaj z Visual Studiem 2017 verzije 15.7.2.

Na sliki 5 je prikazano zanimanje za iskanje izraza C# od leta 2004 do 2019 glede na njegovo najvišje doseženo vrednost v tem obdobju. Razvidno je, da je zanimanje nekoliko padlo (uvedba novih jezikov), saj se C# ne uporablja na vseh področjih, prav tako pa so nekateri jeziki tudi bolj učinkoviti kot le-ta v določenem primeru. Še vedno pa je iskanje tega izraza visoko glede na predhodno sliko zanimanja za Javo (Slika 1), saj je število iskanj še vedno okoli 50 odstotkov od največjega števila (za razliko od Jave – 25%).

Slika 5: Prikaz zanimanja za iskanje programskega jezika C# v obdobju 15 let
https://trends.google.com/trends/explore?date=all&q=%2Fm%2F07657k

2.2.2 Izbira integriranega razvojnega okolja

V času razvoja igre v igralnem pogonu Unity sem sprva uporabljal MonoDevelop (znan tudi pod imenom Xamarin Studio), saj je bila privzeta izbira igralnega okolja (pridobil sem ga skupaj z Unityjem). V verziji Unity 2018.1 pa je privzeto razvojno okolje postal Visual Studio, program pa lahko še vedno napišemo v MonoDevelopu. Osebno sem urejevalnik zamenjal, saj Visual Studio ponuja veliko več kot pa MonoDevelop. Med storitvami, ki jih ponuja je tudi možnost izdelave namiznih aplikacij, spletnih mest, ustvarjanje podatkovne baze ter še veliko več. »Podobno kot vsak drug IDE, Visual Studio vključuje urejevalnik kode, ki podpira poudarjanje sintakse in avtomatsko dokončanje kode z uporabo IntelliSense za spremenljivke, funkcije, metode, zanke in poizvedbe LINQ.« (13)

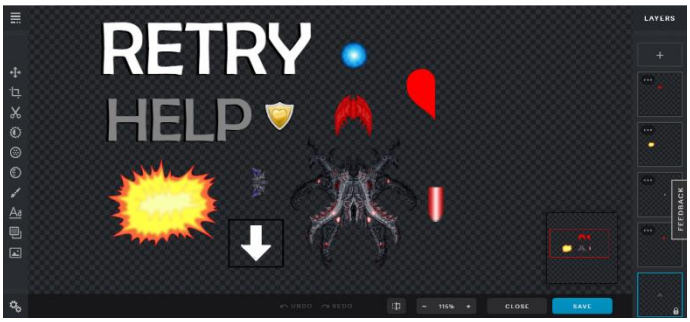
Razvojno okolje kot samo ne podpira nobenega programskega jezika, vendar to omogoča skozi upravitelja paketov NuGet (uporabljamo znotraj urejevalnika) ter namestitveni program Visual Studio Installer, kjer označimo tisto kar potrebujemo, le-ta pa to prenese in namesti na računalnik. Eni izmed najbolj uporabnih stvari, ki sem jih do zdaj izkusil pri uporabi Visual Studia pa sta njegov razhroščevalnik in sistem za upravljanje z različnimi paketov, NuGet. »Visual Studio vključuje razhroščevalnik, ki deluje kot razhroščevalnik na ravni izvora in na ravni stroja. Lahko se uporablja za razhroščevanje aplikacij, napisanih v katerem koli programskem jeziku, ki ga podpira Visual Studio. Poleg tega se lahko pridruži tudi tekočim procesom, nadzira in odpravlja napake v teh procesih.« (14)

2.3 Zbiranje slik, zvočnih posnetkov

Preden sem začel z razvijanjem igre sem potreboval slike, s katerimi bom lahko nato grafično prikazal delovanje igre (prav tako v programskem jeziku Java kot v igralnem pogonu Unity). Nekatere slike sem pridobil na različnih spletnih straneh, vendar sem jih nato moral spremeniti da so postale ustrezne (velikost, oblika, barva ipd.), druge pa sem naredil sam s pomočjo spletnega urejevalnika fotografij.

Za ta namen sem uporabil Pixlr, ki ga je ustvaril Ola Sevandersson leta 2008, zdaj pa je v lasti podjetja 123RF. »Pixlr, ki temelji na oblaku, vsebuje najrazličnejša orodja in pripomočke, vključno s številnimi urejevalniki fotografij, razširitvijo brskalnika za snemanje zaslona in storitvijo za izmenjavo fotografij.« (15) Sprva je bila storitev namenjena amaterjem, sedaj pa je na voljo za preprosto in tudi napredno urejanje slik. Uporablja se lahko na osebnih računalnikih, pa tudi na pametnih telefonih ali tabličnih računalnikih z uporabo mobilne aplikacije.

Na sliki 6 je prikazan posnetek zaslona spletnega urejevalnika PixlrX, katerega sem uporabljal za ustvarjanje slik in urejanje le-teh. Prav tako pa so zraven tudi nekatere slike, ki jih uporabljam za grafično predstavitev igre.



Slika 6: Prikaz spletnega urejevalnika PixlrX in nekaterih slik iz igre

Zbiranje zvočnih posnetkov je potekalo podobno kot pri zbiranju slik, le da pri tem nisem nobenega naredil sam. Večino zvočnih efektov sem pridobil s pomočjo Unityja – Asset Store. Asset Store uporabnikom igralnega pogona in razvijalnega okolja Unity omogoča, da tako hitreje in lažje pridobijo sredstva (predvsem zvočne in vizualni efekte, slike), ki jih potrebujejo za svoj projekt. Sredstva, katere lahko uporabniki pridobijo so brezplačna (večkrat so to sredstva, ki so jih naredili zaposleni v Unity Technologies), večino pa je treba za prenos plačati. To je tudi odličen način zaslužka za tiste, ki se ukvarjajo s 3D modeliranjem, risanjem, z zvočnimi efekti, vizualnimi efekti in drugimi stvarmi, katere je mogoče prodajati preko Unity Asset Store-a.

3 Izdelava igre

Izdelava iger je potekala tako, da sem sprva ustvaril igro v Javi. To je potekalo tako, da sem igro postopoma gradil in ustvarjal algoritme, ki so poskrbeli za določene naloge, katere bom kasneje predstavil. Pri tem sem igro moral neštetokrat testirat tako, da sem glavni razred zagnal ter preveril, ali zaporedja ukazov, katere sem na novo zapisal, pravilno delujejo. Po izdelani igri v programskem jeziku Java, sem se lotil izdelave v igralnem okolju Unity. Tam je izdelave potekala nekoliko drugače, saj grafični urejevalnik vsebuje orodja, ki so mi bila v veliko pomoč. Tudi tukaj sem po vsakem dodanemu elementu igro testiral ter odpravil napake.

3.1 Izdelava v programskem jeziku Java

Igro, napisano v programskem jeziku Java bom predstavil tako, da bom celoten algoritem razdelil na manjše podenote, njihovo pa bom podrobno opisal. Manjši delci programa bodo bili:

- **Uporabniški vmesnik**
- **Funkcije igralca**
- **Funkcije sovražnikov**
- **Izrisovanje grafike**
- **Timerji**

3.1.1 Uporabniški vmesnik

Pod poglavje uporabniški vmesnik spadajo različni meniji, ki omogočajo igralcu ogled pomoči in kontrol za igranje, omogočajo mu tudi izbiro med načinom igranja igre (igralec lahko izbira med neskončnim načinom in zgodbo). Igralec bo lahko v obeh načinih začasno ustavil delovanje igre, kjer se mu bo prikazal meni pavze. Prav tako bo lahko v neskončnem načinu uporabljan meni nadgradenj, kjer bo lahko nadgradil svojo igralno figuro. Ko, oziroma če igralec izgubi, se mu prikaže meni, ki naznanja konec igre, od tam pa lahko ponovno začne igrati, ali pa se vrne v glavni meni.

3.1.1.1 Glavni meni

Pri zagonu igre se prikaže selekcijski glavni meni, ki ga vidimo na sliki 7, kjer se izbere način igre, ki bi jo radi igrali, na tem pa so igralcu na voljo 2 nova stanja: odpre se glavni meni in glede na izbiro ali neskončen način ali zgodba se spremenljivki, ki skrbita za pravilen prikaz igre nastavita na vrednost *true* ali *false*.

Slika 7: Selekcijski glavni meni:

```
}else if(gameModeSelectionMenu) {
    if(mouseMovedX >= (f.getWidth()-640)/2 && mouseMovedX <= (f.ge
        gameModeSelectionMenu = false;
        mainMenu = true;
        campaignMode = true;
    }else if (mouseMovedX >= (f.getWidth()-577)/2 && mouseMovedX <
        gameModeSelectionMenu = false;
        mainMenu = true;
        survivalMode = true;
    }else if (mouseMovedX >= (f.getWidth()-245)/2 && mouseMovedX <
        System.exit(0);
    }
}else if(mainMenu) {
```

Slika 8: Algoritem za selekcijski glavni meni

Na sliki 8 je del programa, kjer ko pritisnemo na miško in jo spustimo se le-ta zažene. Če je uporabnik pritisnil na *Campaign* ali *Survival*, mu prikažemo glavni meni, če pa je pritisni *Exit*, zapremo igro.

3.1.1.2 Glavni meni

V glavnem meniju kot je razvidno iz slike 9 ima uporabnik na voljo 3 stanja: lahko začne igrati igro, odpre meni opcij, ali pa se vrne nazaj v selekcijski glavni meni. V primeru da igralec klikne na opcijo *Play*, kot je vidno na sliki 10, odstranimo *MouseListener* in *MouseMotionListener*. »*MouseListener* je obveščen vsakič, ko spremenite stanje miške. Obveščen je proti *MouseEvent* in ima pet metod.« (16) Podobno je tudi *MouseMotionListener* obveščen ob vsaki spremembi stanja miške, le da ima le-ta 3 metode, ki se sprožijo, ko miško premikamo. Ob tem pa še ustavimo *Timer*, ki skrbi za risanje menijev, ter sprožimo druge *Timerje*, ki skrbijo za pravilno delovanje igre. Če uporabnik klikne *Options*, se glavni meni zapre in se odpre meni opcij, ko pa klikne *Back*, se glavni meni zapre in spet odpre glavni selekcijski meni.

Slika 10: Glavni meni

```
}else if(mainMenu) {
    if(mouseMovedX >= (f.getWidth()-310)/2 && mouseMovedX
        f.removeMouseListener(this);
        f.removeMouseMotionListener(this);
        mainMenu = false;
        menuTimer.stop();
        t.start();
        shipTimer.start();
        difficultyTimer.start();
        powerupTimer.setInitialDelay(20000);
        powerupTimer.start();
    }else if (mouseMovedX >= (f.getWidth()-520)/2 && mous
        optionsMenu = true;
    }else if (mouseMovedX >= (f.getWidth()-245)/2 && mous
        mainMenu = false;
        survivalMode = false;
        campaignMode = false;
        gameModeSelectionMenu = true;
    }
}else if(pauseMenu) {
```

Slika 9: Algoritem za glavni meni

3.1.1.3 Meni opcij

V meniju opcij vidnem na sliki 11 ima igralec na voljo 3 stanja: lahko odpre meni za pomoč, meni kontrol ali pa se vrne nazaj v predhodni meni, ki je bil odprt (meni opcij lahko odpremo preko menija pavze ter prav tako glavnega menija). Iz slike 12 je razvidno, da v primeru da kliknemo na gumb *Help* nastavimo spremenljivko, ki določa kdaj odpremo meni za pomoč na *true*, ter enako za meni kontrol, če pa kliknemo *Back* ponovno odpremo prejšnji meni, ki je bil odprt tako, da nastavimo spremenljivko, ki določa ali je meni opcij odprt, na *false*.

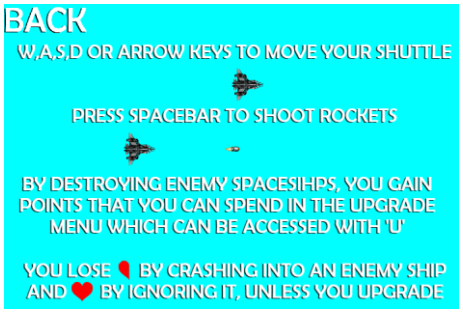
Slika 11: Meni opcij

```
}else if(optionsMenu) {
    if(mouseMovedX >= (f.getWidth()-313)/2 && m
        helpMenu = true;
    }else if (mouseMovedX >= (f.getWidth()-633)
        controlsMenu = true;
    }else if (mouseMovedX >= (f.getWidth()-322)
        optionsMenu = false;
    }
}else if(upgradeMenu) {
```

Slika 12: Algoritem za meni opcij

3.1.1.4 Meni za pomoč

V meniju za pomoč, ki je prikazan na sliki 13, je igralcu napisana pomoč pri igranju igre, pri tem pa sta sliki na kateri se ladja premika in strelja animirani. Prav tako pa ima možnost iti nazaj na prejšnji meni, ki je bil odprt (meni opcij), kot prikazuje slika 14, če uporabnik pritisne gumb *Back*.



Slika 13: Meni za pomoč

```
if(helpMenu) {
    if(mouseMovedX >= 8 && mouseMovedX <=
        helpMenu = false;
    }
```

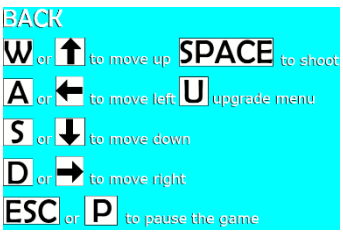
Slika 14: Algoritem za meni za pomoč

3.1.1.5 Meni kontrol

V meniju za pomoč, prikazanemu na sliki 16, so igralcu grafično prikazane kontrole, katere lahko uporablja med igranjem. Če uporabnik pritisne na gumb Back, pa se odpre meni, odprt pred menijem kontrol (slika 15).

```
}else if(controlsMenu) {
    if(mouseMovedX >= 8 && mouseMovedX <= 1
        controlsMenu = false;
    }
}else if(optionsMenu) {
```

Slika 15: Algoritem za meni kontrol



Slika 16: Meni kontrol

3.1.1.6 Meni pavze

Igralec lahko odpre meni pavze natanko takrat, ko je v igri in pritisne na tipko 'P' ali 'ESC' ter ni odprt noben drug meni (razen menija pavze) , kar je razvidno iz slike 17. Če je meni pavze že odprt, se le-ta zapre in se igra nadaljuje s tem, da zaustavimo *Timer*, ki skrbi za menije, in ponovno zaženemo vse *Timerje*, ki skrbijo za ustvarjanje novih sovražnikov, funkcije glavnih sovražnikov, povečanje težavnosti in ustvarjanje začasnih nadgradenj (dodatkov) za igralca. Če pa meni pavze ni odprt in zadošča pogojem, zaustavimo vse *Timerje*, ki skrbijo za delovanje igre in zaženemo *Timer*, ki skrbi za menije, ter dodamo *MouseListener* in *MouseMotionListener*, s katerimi poslušamo za spremembe stanj miške.

```
if((e.getKeyCode() == KeyEvent.VK_ESCAPE || e.getKeyCode() == KeyEvent.VK_P) && !optionsMenu && !upgradeMenu &&
    if(pauseMenu){
        f.removeMouseListener(this); f.removeMouseMotionListener(this);
        pauseMenu = false;
        menuTimer.stop();
        t.start();
        if(timerPauseAmount > 0) {
            shipTimer.setInitialDelay(timerPauseAmount);
        }
        if(difficultyTimerInitialAmount > 0) {
            difficultyTimer.setInitialDelay(difficultyTimerInitialAmount);
        }
        if(bossAbilityTimeInitialAmount> 0) {
            bossAbilityTimer.setInitialDelay(bossAbilityTimeInitialAmount);
        }
        if(powerupTimeInitialAmount > 0) {
            powerupTimer.setInitialDelay(bossAbilityTimeInitialAmount);
        }
        bossAbilityTimer.start();
        powerupTimer.start();
        shipTimer.start();
        difficultyTimer.start();
    }else {
        f.addMouseListener(this);
        f.addMouseMotionListener(this);
        pauseMenu = true;
        t.stop(); shipTimer.stop(); difficultyTimer.stop(); bossAbilityTimer.stop(); powerupTimer.stop();
        menuTimer.start();
    }
```

Slika 17: Pritisk tipke 'P' ali 'ESC'

V meniju pavze, prikazanem na sliki 19, ima igralec na voljo 3 stanja: lahko se vrne nazaj v igro, lahko odpre meni opcij, ali pa se vrne nazaj v glavni meni. Če igralec klikne na gumb *Resume*, se zgodi enako, kot pri sliki 17, ko igralec pritisne tipko 'P' in je meni pavze že odprt. Ko pritisne na gumb *Options* se odpre meni opcij, v primeru klika na *Main Menu* pa se vse spremenljivke resetirajo na njihovo začetno vrednost, vsi *ArrayListi*, ki vsebujejo podatke o lokacijah objektov se izpraznijo, *Timerji* se ponastavijo, prikaže pa se tudi glavni meni (delno vidno na sliki 18).

```
}else if (mouseMovedX >= (f.getWidth()-313)/2
    optionsMenu = true;
}else if (mouseMovedX >= (f.getWidth()-693)/2
    //nared meteod da bo pokiica k bom nucu tu
    //IMPORTANT - ADD ALL THE ORIGINAL VALUES
    compareTime = 0;
    countMissiles = 0;
    countEnemySpaceships = 0;
    countExplosions = 0;
    countEnemyMissiles = 0;
    countEnemyBullets = 0;
```

Slika 18: Del Algoritma za meni pavze

Slika 19: Meni pavze

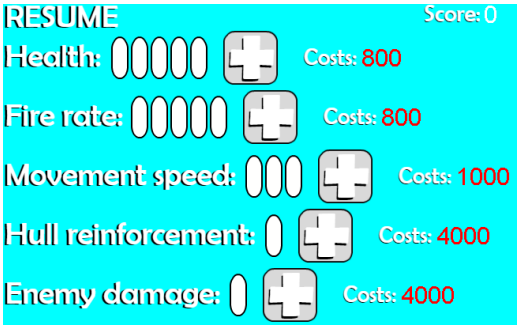
3.1.1.7 Meni nadgradenj

Igralec lahko odpre meni nadgradenj natanko takrat, ko je igra neskončni način igre in pritisne na tipko 'U' ter ni odprt noben drug meni (razen menija nadgradenj) , kar je razvidno iz slike 20. Če je meni nadgradenj že odprt, se le-ta zapre in se igra nadaljuje s tem, da zaustavimo *Timer*, ki skrbi za menije, in ponovno zaženemo vse *Timerje,* ki skrbijo za pravilno delovanje iger. Če uporabnik pritisne tipko 'U' in je meni nadgradenj že odprt, se zgodi enako kot pri meniju pavze na sliki 17, če pa meni nadgradenj še ni odprt, se le-ta prikaže, ostalo pa je identično kot pri meniju pavze.

```
if((e.getKeyCode() == KeyEvent.VK_U && !mainMenu && !pauseMenu && !gameOverMenu &
    if(upgradeMenu){
        f.removeMouseListener(this);
        f.removeMouseMotionListener(this);
        upgradeMenu = false;
        menuTimer.stop();
        t.start();
        if(timerPauseAmount > 0) {
            shipTimer.setInitialDelay(timerPauseAmount);
        }
        if(difficultyTimerInitialAmount > 0) {
            difficultyTimer.setInitialDelay(difficultyTimerInitialAmount);
        }
        if(powerupTimeInitialAmount > 0) {
            powerupTimer.setInitialDelay(bossAbilityTimeInitialAmount);
        }
        powerupTimer.start();
        shipTimer.start();
        difficultyTimer.start();
    }else {
        f.addMouseListener(this);
        f.addMouseMotionListener(this);
        upgradeMenu = true;
        t.stop(); shipTimer.stop(); powerupTimer.stop(); difficultyTimer.stop();
        menuTimer.start();
    }
}
```

Slika 20: Pritisk tipke 'U'

Ko ima igralec pred seboj odprt meni nadgradenj se lahko vrne nazaj v igro s klikom na gumb *Resume* ali s pritiskom na tipko 'U'. Poleg tega ima igralec tudi na voljo 5 različnih stvari, vidne na sliki 21, katere lahko nadgradi. Med temi so število življenj, hitrost streljanja, hitrost premikanja igralne figure, zmanjšanje moči sovražnikov ter ojačitev igralne figure.



Slika 21: Meni nadgradenj

Če igralec klikne na gumb za nadgradnjo, viden na sliki 21 (velik plus znak), in ima trenutno dovolj veliko število točk in število nadgradenj ni večja kot 4, se izvrši algoritem, ki to nadgradnjo opravi. Algoritem, ki to naredi je viden na sliki 22. V primeru nadgradnje življenja igralni figuri dodelimo 3 nova cela življenja, ceno naslednje nadgradnje podražimo in povečamo število nadgradenj za to vrsto nadgradnje. Podobno se zgodi s hitrostjo igralne figure in hitrostjo streljanja, pri ojačitvi igralne figure in zmanjšanju sovražnikove moči pa le nastavimo spremenljivki na *true*.

```
}else if(upgradeMenu) {
    if(mouseMovedX >= 8 && mouseMovedX <= 145+8 && mouseMovedY >= 5+31 && mouseMovedY <= 37+31) {
        f.removeMouseListener(this);
        f.removeMouseMotionListener(this);
        upgradeMenu = false;
        menuTimer.stop();
        t.start();
        if(timerPauseAmount > 0) {
            shipTimer.setInitialDelay(timerPauseAmount);
        }
        if(difficultyTimerInitialAmount > 0) {
            difficultyTimer.setInitialDelay(difficultyTimerInitialAmount);
        }
        difficultyTimer.start();
        shipTimer.start();
    }else if((mouseMovedX >= 280+8 && mouseMovedX <= 280+7+70 && mouseMovedY >= 32+5+28 && mouseM
        halfLives += 6; score -= healthUpgradeCost; healthUpgradeCost += healthUpgradeCost/2;
        healthUpgraded++;
    }else if((mouseMovedX >= 305+8 && mouseMovedX <= 305+7+70 && mouseMovedY >= 117+5+28 && mouse
        fireRate -= 110; score -= fireRateUpgradeCost; fireRateUpgradeCost += fireRateUpgradeCost
        fireRateUpgraded++;
    }else if((mouseMovedX >= 400+8 && mouseMovedX <= 400+7+70 && mouseMovedY >= 182+5+28 && mouse
        movementSpeed++; score -= movementSpeedUpgradeCost; movementSpeedUpgradeCost += movementS
        movementSpeedUpgraded++;
    }else if((mouseMovedX >= 375+8 && mouseMovedX <= 375+7+70 && mouseMovedY >= 257+5+28 && mouse
        hullReinforced = true; score -= hullReinforcementCost;
    }else if((mouseMovedX >= 330+8 && mouseMovedX <= 330+7+70 && mouseMovedY >= 332+5+28 && mouse
        reducedEnemyDamage = true; score -= reduceEnemyDamageCost;
    }
}else if(gameModeSelectionMenu) {
```

Slika 22: Algoritem za meni nadgradenj

3.1.1.8 Končni meni

Končni meni se igralcu prikaže takrat, ko igralna figura nima več življenj. V tem meniju lahko izbira med 3 stanji, prikazanimi na sliki 23: resetira igro in jo ponovno začne igrati, odpre meni opcij ali pa se vrne nazaj v glavni meni. Algoritem za ponovni zagon igre deluje skoraj identično kot algoritem za prikaz glavnega menija, delno viden na sliki 18 (ponastavitev vseh komponent, izbris vseh koordinat iz *ArrayListov*, ponastavitev časa *Timerjev* na osnovne vrednosti itd.) z razliko v tem, da namesto da teče *Timer*, ki skrbi za menije, le-tega zaustavimo in zaženemo *Timerje*, ki skrbijo za pravilno delovanje igre, kot vidimo na sliki 24.

Slika 23: Končni meni

```
powerupTimer.setInitialDelay(20000);
menuTimer.stop();
t.start();
shipTimer.start();
powerupTimer.start();
if(survivalMode) {
    difficultyTimer.start();
}
```

Slika 24: Del algoritma za ponovni zagon igre

3.1.2 Funkcije igralne figure

Igralna figura, katero nadzoruje igralec, ima možnost premikanja in streljanja, ob tem pa v ozadju delujejo tudi algoritmi, ki preverjajo ali je figura trenutno v trku s katerim drugim objektom trenutno narisanim na zaslonu. Ti algoritmi se izvajajo v neskončnem načinu kot tudi v načinu zgodbe.

3.1.2.1 Premikanje igralne figure

Igralec se lahko premika v smeri x in y in sicer z vnaprej določeno hitrostjo, ki pa jo lahko v neskončnem načinu igre zmanjša (z nadgradnjami). Na sliki 25 vidimo, da funkcija z imenom *movePicture* skrbi za to, da se figura premika v tisto smer, ki jo tipka, katero igralec trenutno drži, določa. V primeru da je trenutno na zaslonu narisan tudi boljši sovražnik pa igralni figuri ne pustimo, da gre lahko višje kot pa je trenutno najnižja točka boljšega sovražnika (razvidno iz slike 25).

```
public void movePicture() {
    if(right) {
        if(spaceX <= 627) {spaceX += movementSpeed;}
    }
    if(left) {
        if(spaceX >= 3) {spaceX -= movementSpeed;}
    }
    if(up) {
        if(spaceY >= 3) {
            if(level3) {
                if(!boss || spaceY > bossY + 200) {
                    spaceY -= movementSpeed;
                }
            }else if(level2) {
                if(!boss || spaceY > bossY + 230) {
                    spaceY -= movementSpeed;
                }
            }else{
                if(!boss || spaceY > bossY + 160) {
                    spaceY -= movementSpeed;
                }
            }
        }
    }
    if(down) {
        if(spaceY <= 387) {spaceY += movementSpeed;}
    }
}
```

Slika 25: Funkcija za premikanje igralne figure

3.1.2.2 Funkcija za streljanje

Igralec ima možnost streljanja vsake toliko časa, kot mu je dovoljeno. V neskončnem načinu igranja lahko ta čas zmanjša preko nadgradenj. Na sliki 26 sta vidni funkciji *createMissile*, ki ustvari nov izstrelek, če igralec tišči preslednico in je poteklo dovolj časa, da mu je zdaj dovoljeno streljati. Izstrelek dodamo v dva *ArrayLista* izstrelkov, ki vsebujeta x in y koordinati letega. To uporabljamo nato tudi pri funkciji *moveMissile*, ki preverja ali je izstrelek še v igralnem oknu in če je, mu zmanjša y komponento za določeno vrednost, če ne pa lokacijo izstrelka odstrani iz *ArrayListov*.

```
public void moveMissiles() {
    for(int i = 0; i < countMissiles; i++) {
        int y = missilesY.get(i) - 3;
        missilesY.remove(i);
        if(y > -30) {
            missilesY.add(i, y);
        }else {
            missilesX.remove(i);
            countMissiles--;
        }
    }

}

protected void createMissile() {
    if(compareTime <= 0 && spacebar) {
        missilesX.add(spaceX+25);
        missilesY.add(spaceY-15);
        countMissiles++;
        compareTime = fireRate;
    }
}
```

Slika 26: Prikaz funkcij za ustvarjanje in premikanje izstrelkov

3.1.2.3 Preverjanje kolizij

Preverjanje trkov oziroma kolizij igralne figure z drugimi objekti trenutno aktivnimi na igralnem območju poteka skoraj identično za vse različne trke (objekt, ki je trčil z igralno figuro je izbrisan, figura je prikazana, da je bila zadeta, če je poteklo dovolj časa med dvema kolizijama figure tej odštejemo določeno število življenj ali ščitov ter jo naredimo neranljivo za nekaj časa). Primer na sliki 27 prikazuje funkcijo *checkEnemyMissilesCollision*, ki preverja ali je igralna figura trenutno v trku s sovražnikovimi izstrelki. V primeru, da je le-ta v trku, izbrišemo izstrelek, prikažemo, da je igralna figura bila zadeta in če je bila zadeta kasneje kot pa 0,9 sekunde pred zadnjo izvršitvijo te funkcije, figuri odštejemo določeno število življenj oziroma ščitov, če jih le-ta ima, in nastavimo čas, v katerem je ta figura ranljiva nazaj na 0.9 sekunde. V nasprotnem primeru pa ne naredimo nič, saj je figura trenutno neranljiva.

```
public void checkEnemySpaceshipCollision() {
public void checkEnemyMissilesCollision() {
    for(int i = 0; i < countEnemyMissiles; i++) {
            if((enemyMissilesX.get(i) >= spaceX+10 && enemyMissilesX.get(i)
                yourShipHit = 10;
                enemyMissilesX.remove(i); enemyMissilesY.remove(i);
                countEnemyMissiles--;
                if(invulnerableTime <= 0) {
                    if(shields > 0) {
                        shields--;
                    }else {
                        halfLives--;
                    }
                    invulnerableTime = 900;
                }
                break;
            }
    }
}
public void checkEnemyBulletsCollision() {
public void checkEnemyLasersCollision() {
public void checkEnemy3PlasmaCollision() {
public void checkEnemyBullets2Collision() {
public void checkEnemyBullets2MissileCollision() {
public void checkEnemyPlasmaBulletsCollision() {
public void checkDroneBulletsCollision() {
public void checkPowerupCollision() {
```

Slika 27: Funkcije za trke igralne figure

Edina funkcija, ki se od drugih razlikuje pa je z imenom *checkPowerupCollision*, pri kateri če pride do trka igralni figuri dodelimo začasno nadgradnjo (odvisno od tipa nadgradnje, s katero je igralna figura trčila). V primeru, da je le ta tipa začasne nadgradnje hitrosti, lahko iz slike 28 razberemo, da povečamo hitrost igralne figure za 1, nadgradnja pa bo trajala 10 sekund. V primeru da pa je nadgradnja tipa pridobitev ščita, pa število le-teh povečamo za 1, če je trenutno število manjše kot 5.

```
public void checkPowerupCollision() {
    int y = powerupY;
    int x = powerupX;
    if((x >= spaceX+20 && x <= spaceX+50 && y <= spaceY+70 && y >= spaceY)
        powerupY = 500;
        powerupTime = 10000;
        if(selectedPowerup == speedPowerup) {
            movementSpeed++;
        }else if(selectedPowerup == fireRatePowerup) {
            fireRate = 300;
        }else if(selectedPowerup == healthPowerup) {
            if(halfLives < 8) {
                halfLives += 2;
            }else{
                halfLives = 10;
            }
        }else {
            if(shields < 5) {
                shields++;
            }
        }
    }
}
```

Slika 28: Funkcija za trk začasne nadgradnje z igralno figuro

3.1.3 Funkcije sovražnikov

V igri obstaja več vrst sovražnikov in sicer so razdeljeni na sovražnike tipa 1, 2 in 3, na boljše sovražnike in na tako imenovane »drone« (premikajo se levo in desno ter lahko streljajo). Sovražnik tipa 1 je najšibkejši, tipa 2 je boljši, tipa 3 pa ima tudi možnost streljanja. Boljši sovražniki so le v načinu zgodbe in sicer obstajajo trije. Vsak ima 3 različne napade, katere lahko uporablja in sicer lahko koristi napad 1 ali 2, ki se pri vseh razlikujejo, ali pa ustvarijo sovražnike tipa 1, 2 oziroma 3 (glede na tip boljšega sovražnika).

3.1.3.1 Premikanje sovražnikov

Sovražniki tipa 1, 2 in 3 imajo enak sistem premikanja, viden na sliki 29. Ko sovražnik zapusti okvir igralnega polja, se igralni figuri odšteje določeno število življenjskih točk (odvisno od tipa sovražnika). V primeru, da ima igralna figura vsaj en ščit, se namesto življenja zmanjša število ščitov. Če pa je bila igralna figura v času manj kot 0,9 sekund pred tem dogodkom v trku, se ne zgodi nič. Na koncu odštejemo igralcu še število doseženih točk.

```
public void moveEnemySpaceship() {
        for(int i = 0; i < countEnemySpaceships; i++) {
            int y = enemyShipY.get(i) + 2;
            enemyShipY.remove(i);
            if(y < 485) {
                enemyShipY.add(i, y);
            }else {
                enemyShipX.remove(i);
                if(!reducedEnemyDamage && invulnerableTime <= 0) {
                    if(shields > 0) {
                        shields--;
                    }else {
                        halfLives-=2;
                    }
                    invulnerableTime = 900;
                }else if(invulnerableTime <= 0){
                    if(shields > 0) {
                        shields--;
                    }else {
                        halfLives--;
                    }
                    invulnerableTime = 900;
                }
                if(score != 0) {
                    score -= 10;
                }
                countEnemySpaceships--;
            }
        }
```

Slika 29: Algoritem za premikanje sovražnikov tipa 1, 2 in 3

```
public void moveBoss() {
    if(bossY < 0 && bossExplode == 0) {
        bossY += 0.5;
        if(level3) {
            if(spaceY <= bossY+3+200) {
                spaceY += 1;
            }
        }else if(level2) {
            if(spaceY <= bossY+3+230) {
                spaceY += 1;
            }
        }else {
            if(spaceY <= bossY+3+165) {
                spaceY += 1;
            }
        }
    }else if(bossHealth == 0 && bossExplode != 0) {
        bossY -= 1;
    }
}
```

Slika 30: Algoritem za premikanje boljših sovražnikov

Premikanje boljših sovražnikov pa poteka na nekoliko drugačen način, saj so le-ti v fazi premikanja le ko vstopajo v igro in ko so v fazi uničenja. Ko boljši sovražnik vstopa v igro, ga glede na njegovo velikost v smeri y (tip 1, 2 in 3 boljši sovražniki se razlikujejo glede velikost) premikamo do točke, kjer je zagotovo izrisan na zaslonu celotno, kot vidimo na sliki 30. Ko pa je v fazi uničenja, ga ne glede na njegov tip premikamo z enako hitrostjo izven okvirja igralne površine.

Prav tako je način premikanja dronov različen od načina premikanja drugih tipov sovražnikov, saj se le-ti premikajo le ko igralec igra način igre zgodba in je na igralnem polju boljši sovražnik tipa 3. Droni so način napadanja boljšega sovražnika, imajo pa svoj sistem premikanja in streljanja. Iz slike 31 je razvidno, da glede na to, katera dva drona sta trenutno aktivna, ju premikam do pozicije, kjer lahko nato začneta streljati. To je možno razbrati iz dela algoritma funkcije *moveEnemyDrones*, kjer s pomočjo spremenljivk tipa *boolean* preverimo, katera dva drona sta aktivna in ju premikamo do željene pozicije. Če pa dron trenutno ni aktiven, ga premaknemo nazaj na pozicijo izven vidnega okvirja.

```
public void moveEnemyDrones() {
    if(moveDrone0) {
        if(secondary) {
            if(dronesX.get(0) < 0) {
                int x = dronesX.get(0)+1;
                dronesX.remove(0);
                dronesX.add(0, x);
            }
        }
    }
    if(!moveDrone0 && dronesX.get(0) > - 30) {
        int x = dronesX.get(0) - 1;
        dronesX.remove(0);
        dronesX.add(0, x);
    }
    if(moveDrone1) {
        if(secondary) {
            if(dronesX.get(1) < 0) {
                int x = dronesX.get(1)+1;
                dronesX.remove(1);
                dronesX.add(1, x);
            }
        }
    }
    if(!moveDrone1 && dronesX.get(1) > - 30) {
        int x = dronesX.get(1) - 1;
        dronesX.remove(1);
        dronesX.add(1, x);
    }
```

Slika 31: Del algoritma za premikanje sovražnikov tipa dron

3.1.3.2 Streljanje sovražnikov

Izmed sovražnikov tipa 1, 2 in 3 imajo le tisti, ki so tipa 3, možnost streljanja. Na sliki 32 je viden algoritem, ki skrbi za ustvarjanje novega izstrelka in sicer uporabljamo *ArrayList*, ki vsebuje hitrost streljanja, s tem pa preverimo ali lahko posamezen sovražnik tipa 3 strelja. V primeru, da lahko, dodamo nov izstrelek v *ArrayListe*, ki držijo x in y koordinate le-tega, obenem pa resetiramo čas, potreben da poteče pred ponovnim streljanjem.

```
public void enemyShips3Shoot() {
    for(int i = 0; i < countEnemySpaceships3; i++) {
        if(enemyShip3AttackTime.get(i) <= 0 && enemyShip3Y.get(i) >= -60) {
            enemyShip3AttackTime.remove(i);
            enemyShip3AttackTime.add(i, 1500);
            enemyPlasmaX.add(enemyShip3X.get(i)+25);
            enemyPlasmaY.add(enemyShip3Y.get(i)+30);
            countEnemyPlasma++;
        }
    }
}
```

Slika 32: Algoritem za streljanje sovražnikov tipa 3

Za razliko od navadnih sovražnikov tipa 1, 2 in 3 imajo vsi boljši sovražniki možnost streljanja. Pravzaprav imajo 3 različne napade: imajo možnost ustvarjanja navadnih sovražnikov, možnost uporabe primarnega napada ter možnost uporabe sekundarnega napada. Slednja sta pri sovražnikih različna, to pa pomeni da v ozadju deluje drugačen algoritem odvisno od tipa boljšega sovražnika, ki je trenutno v igri.

Na sliki 33 so napisani vsi algoritmi, katere uporabljam za streljanje boljših sovražnikov. Vsi delujejo na enak princip, vendar z rahlimi variacijami, kot delujeta algoritma, ki je na sliki 33 viden v celoti in del funkcije, vidne na sliki 34. Funkcija *dronesShoot* se izvede natanko takrat, ko je v igri boljši sovražnik tipa 3 in je le-ta uporabil svoj sekundarni napad. Njegov primarni napad pa je znotraj funkcije *firePrimaryWeapon*, kjer ustvarimo dva nova izstrelka, jima določimo hitrost letenja in nato še spremembo lokacije (delta pot) , katero uporabljamo pri premiku izstrelka. Velikosti sta izračunani tako, da izberem X in Y lokacijo (odvisno kam se igralna figura premika) in glede razliko med začetno lokacijo in to lokacijo nastavim veličino hitrosti.

```
public void createMinions() {
public void firePrimaryWeapon() {
public void fireSecondaryWeapon() {
public void createEnemyBullet() {
public void createEnemyLasers() {
public void dronesShoot() {
    if(dronesAttackTime <= 0) {
        if(moveDrone0) {
            droneBulletsX.add(dronesX.get(0) + 20.0);
            droneBulletsY.add(dronesY.get(0) + 15.0);
            countEnemyDroneBullets++;
        }
        if(moveDrone1) {
            droneBulletsX.add(dronesX.get(1) + 20.0);
            droneBulletsY.add(dronesY.get(1) + 15.0);
            countEnemyDroneBullets++;
        }
        if(moveDrone2) {
            droneBullets2X.add(dronesX.get(2) + 5.0);
            droneBullets2Y.add(dronesY.get(2) + 15.0);
            countEnemyDroneBullets2++;
        }
        if(moveDrone3) {
            droneBullets2X.add(dronesX.get(3) + 5.0);
            droneBullets2Y.add(dronesY.get(3) + 15.0);
            countEnemyDroneBullets2++;
        }
        dronesAttackTime = 1800;
    }
}
```

Slika 33: Algoritem sekundarnega napada boljšega sovražnika tipa 3

```
public void firePrimaryWeapon() {
    System.out.println("primary");
    if(level3) {
        enemyPlasmaBulletsX.add((this.getWidth()-220)/2-8 + 75.0);
        enemyPlasmaBulletsY.add(150.0);
        double x = enemyPlasmaBulletsX.get(countEnemyPlasmaBullets);//začetka X lokacija
        double y = spaceY - 120;//začetna Y lokacija
        if(left) {//če se igralna figura premika levo
            x = spaceX-75 - x;
            if(up) {//če se igralna figura premika gor
                y -= 40;
            }
        }else if(right) {//če se igralna figura premika desno
            x = spaceX+35 - x;
        }else {x = spaceX-10 - x;}
        double a = Math.max(Math.abs(x), Math.abs(y))/Math.min(Math.abs(x), Math.abs(y));
        if(x == 0) {a = Math.abs(y);}
        double v = Math.sqrt(16.0/(Math.pow(a, 2) + 1));//izračun skupne hitrosti
        double vx = v; double vy = v;
        if(x < 0) {
            vx *= -1;
        }
        double sx, sy = 0;
        if(Math.abs(x) > y) {
            sx = a*vx; sy = vy;//določitev "poti", ki jo izstrelek opravi vsak repaint v smeri X
        }else {
            sy = a*vy; sx = vx;//določitev "poti", ki jo izstrelek opravi vsak repaint v smeri Y
        }
        deltaMovePlasmaBulletsX.add(sx); deltaMovePlasmaBulletsY.add(sy);//shranim razliko poti v obeh smereh
        countEnemyPlasmaBullets++;
```

Slika 34: Del algoritma primarnega napada boljšega sovražnika tipa 3

Za premikanje izstrelkov pa prav tako uporavljam X in Y koordinate objektov, ki so zapisane v *ArrayListih*. To je tudi vidno v primeru algoritma (*moveEnemyPlasmaBullets*) za premikanje izstrelkov boljšega sovražnika tipa 3 na sliki 35, kjer v primeru, da je objekt še v igralnem polju le-tega algoritem premakne za delta X in Y, drugače pa lokacije in posledično objekt izbrišemo.

```
public void moveEnemyMissiles() {
public void moveEnemyBullets() {
public void moveEnemyBullets2() {
public void moveEnemyLasers() {
public void moveDroneBullets() {
public void moveEnemy3Plasma() {
public void moveEnemyPlasmaBullets() {
    for(int i = 0; i < countEnemyPlasmaBullets; i++) {
        double y = enemyPlasmaBulletsY.get(i) + deltaMovePlasmaBulletsY.get(i);
        double x = enemyPlasmaBulletsX.get(i) + deltaMovePlasmaBulletsX.get(i);
        enemyPlasmaBulletsY.remove(i);
        enemyPlasmaBulletsX.remove(i);
        if(y < 495 && x > -20 && x < 720) {
            enemyPlasmaBulletsY.add(i, y);
            enemyPlasmaBulletsX.add(i, x);
        }else {
            countEnemyPlasmaBullets--;
            deltaMovePlasmaBulletsX.remove(i);
            deltaMovePlasmaBulletsY.remove(i);
        }
    }
}
```

Slika 35: Algoritmi za premikanje izstrelkov sovražnikov

3.1.3.3 Kolizije sovražnikov

Ker obstaja več vrst sovražnikov in imajo le-ti možnost streljanja, vsebuje ta del celotnega algoritma veliko funkcij za preverjanje trkov med temi objekti in igralno figuro oziroma izstrelkom igralne figure. Na sliki 36 je prikazan del algoritma za preverjanje trkov med boljšim sovražnikom in izstrelki igralne figure. Le-ta v primeru, da je igra trenutno v načinu zgodbe in je tip boljšega sovražnika 3, izvrši ukaze, ki mu zmanjšajo število življenjskih točk, prikažejo, da je bil objekt zadet, koordinate izstrelka se odstranijo in število izstrelkov se zmanjša za 1. Ko se število življenjskih točk zmanjša na število 0, boljši sovražnik preneha z uporabo svojih napadov, se začne premikati izven okvirja igralnega območja, pri tem pa se na sovražniku začnejo izrisovati eksplozije (določeno, koliko časa bodo bile aktivne – dokler sovražnik ne zapusti okvirja).

```
public void checkBossHit() {
    for(int j = 0; j < countMissiles; j++) {
        if(level3) {
            //w+200 (220-20)=max
            //w-20 = min
            int w = (this.getWidth()-220)/2-8;
            if((missilesX.get(j) <= w+30 && missilesX.get(j) >= w-10/*or -8, not sur
                missilesX.remove(j);
                missilesY.remove(j);
                countMissiles--;
                if(bossHealth != 0) {
                    bossHit = 10;//indikator, da je bil boljši sovražnik zadet
                    bossHealth--;//zmanjšamo število življenskih točk
                    if(bossHealth == 0) {//v primeru, da nima več življenskih točk
                        bossExplode = 25;
                        secondary = false;
                        primary = false;
                    }
                }
                break;
            }
        }else if(level2) {
            int w = (this.getWidth()-230)/2-8;
            //w+210 - (230 - 20) = max
            //w-20 = min
            if((missilesX.get(j) <= w+26 && missilesX.get(j) >= w-7 && missilesY.get
                missilesX.remove(j);
                missilesY.remove(j);
                countMissiles--;
```

Slika 36: Trki izstrelkov igralne figure z boljšim sovražnikom

Precej podobno delujejo tudi funkcije, vidne na sliki 37, z razliko v tem, da tukaj preverjam ali je bila igralna figura zadeta s katerokoli vrsto sovražnika ali sovražnikovega izstrelka. Na sliki 37 je prikazan algoritem za preverjanje trka med igralno figuro in izstrelkom boljšega sovražnika tipa 3 (primarni napad), pri katerem ob trku odstranim X in Y koordinato izstrelka ter prav tako izstrelek sam. Ob tem tudi prikažem, da je bila figura zadeta in ji odstranim določeno število ščitov oziroma življenjskih točk, če trenutno ni neranljiva.

```
public void checkEnemySpaceshipCollision() {
public void checkEnemyBulletsCollision() {
public void checkEnemyLasersCollision() {
public void checkEnemy3PlasmaCollision() {
    for(int i = 0; i < countEnemyPlasma; i++) {
        if((enemyPlasmaX.get(i) >= spaceX+5 && enemyPlasmaX.get(i) < spaceX+30
            yourShipHit = 10;//indikator, da je bila igralna figura zadeta
            enemyPlasmaX.remove(i); //odstranimo X koordinato izstrelka
            enemyPlasmaY.remove(i); //odstranimo Y koordinato izstrelka
            countEnemyPlasma--; //izstrelek odstranimo
            if(invulnerableTime <= 0) {//če figura trenutno ni ranljiva
                if(shields > 0) {
                    shields--;
                }else {
                    halfLives--;
                }
                invulnerableTime = 900;//igralna figura je neranljiva 0.9s
            }
            break;
        }
    }
}
public void checkEnemyBullets2Collision() {
public void checkEnemyBullets2MissileCollision() {
public void checkEnemyPlasmaBulletsCollision() {
```

Slika 37: Algoritmi za preverjanje kolizij sovražnikov in izstrelkov

3.1.4 Izrisovanje grafike

Prikaz vseh slik, ki jih želim v igri v določenem primeru, se zgodi tako, da se sproži funkcija, imenovana *paintComponent*, ki je vidna na sliki 38. Če na kratko opišem delovanje te funkcije, kot vhodni podatek zahteva objekt tipa *Graphics* in se sproži takrat, ko se na zaslonu karkoli spremeni (npr. premik miške, pritisk na tipko, klik miške) ali pa jo sprožimo avtomatsko s klicem funkcije *repaint*. »Razred *Graphics* je abstraktni osnovni razred za vse grafične kontekste, ki aplikaciji omogoča risanje na komponente, ki so realizirane na različnih napravah, kot tudi na slike izven zaslona.« (17) Znotraj *paintComponent* glede na stanje igre (lahko je odprt eden izmed menijev ali pa igra trenutno teče) pokličemo funkcije, ki so odgovorne za izrisovanje določenih menijev oziroma za izrisovanje vseh objektov trenutno v igri. Funkcija ki skrbi za izrisovanje celotne igre, *draw* pa prav tako vzame parameter tipa *Graphics*, kot je vidno na sliki 39.

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    if(helpMenu) {
        showHelpMenu(g);
    }else if(controlsMenu) {
        showControlsMenu(g);
    }else if((optionsMenu && mainMenu) || (optionsMenu && gameOverMenu)) {
        showOptionsMenu(g);
    }else if(optionsMenu && pauseMenu) {
        draw(g);
        showOptionsMenu(g);
    }else if(mainMenu) {
        showMainMenu(g);
    }else if(pauseMenu) {
        draw(g);
        showPauseMenu(g);
    }else if(upgradeMenu) {
        showUpgradeMenu(g);
    }else if (gameOverMenu) {
        showGameOverMenu(g);
    }else if (winMenu) {
        showWinMenu(g);
    }else if (gameModeSelectionMenu) {
        showGameModeSelectionMenu(g);
    }else {
        if(mouseMovedX != 0 && mouseMovedY != 0) {
            mouseMovedX = 0; mouseMovedY = 0;
        }
        draw(g);
    }
```

Slika 38: Prikaz funkcije paintComponent za izrisovanje grafike

Izrisovanje poteka na podoben način za vse objekte, prisotne v igri, in sicer na sliki 39 je prikazano izrisovanje ozadja igre, ki se ponavlja, ter različnih izstrelkov sovražniki. Proces poteka tako, da gremo s pomočjo zanke skozi celotna ArrayLista, ki držita X in Y komponento določenega objekta, le-tega pa izrišemo tako, da uporabimo funkcijo *drawImage*. Kot argumente ji podamo X in Y koordinato ter objekt tipa *BufferedImage*, ki je slika, katero bomo izrisali.

```
public void draw(Graphics g) {
    //Risanje in premikanje slike vesolja
    int eyc = (int)ey;
    while(eyc > -725) {
        g.drawImage(space, ex, eyc, null);
        eyc -= 720;
    }
    //risanje laserja (BOSS LVL2 primary attack)
    for(int i = 0; i < countEnemyLasers; i++) {
        g.drawImage(enemyLaser, enemyLasersX.get(i), enemyLasersY.get(i), null);
    }
    //risanje raket(metki) igralca
    for(int i = 0; i < countMissiles; i++) {
        g.drawImage(missile, missilesX.get(i), missilesY.get(i), 40, 40, null);
    }
    //risanje metkov dronov, ki se premikajo v ENI smeri -> enemy type 3
    for(int i = 0; i < countEnemyDroneBullets; i++) {
        double x = droneBulletsX.get(i); int xd = (int)x;
        double y = droneBulletsY.get(i); int yd = (int)y;
        g.drawImage(droneBullet1, xd, yd, null);
    }
    //risanje metkov dronov, ki se premikajo v DRUGI smeri -> enemy type 3
    for(int i = 0; i < countEnemyDroneBullets2; i++) {
        double x = droneBullets2X.get(i); int xd = (int)x;
        double y = droneBullets2Y.get(i); int yd = (int)y;
        g.drawImage(droneBullet2, xd, yd, null);
    }
    //če smo v levelu 3 -> narišemo 4 drone 2 na levi in 2 na desni
```

Slika 39: Prikaz dela algoritma za izrisovanje objektov v igri:

Edina stvar, ki pa se razlikuje od opisanih izrisovanj pa so eksplozije. Na sliki 40 je del algoritma, ki izrisuje eksplozije na sovražnikih tipa 1, 2 in 3. Ko je bil sovražnik uničen, sem nastavil število ponovitev risanja eksplozije, tega pa v tem delu uporabljamo. Če je število ponovitev risanj eksplozije večja ali enaka 4, rišemo eksplozijo, ki je normalne velikosti, ko pa se število ponovitev risanj zmanjša (vsako ponovitev se spremeni za 1), zmanjšamo tudi velikost eksplozije. Ko pa se število ponovitev zmanjša na število 0, eksplozijo odstranimo iz zaslona. S tem simuliram animirano eksplozijo.

```
//risanje eksplozij --> v bistvu simularimo animacijo (približno)
for(int i = 0; i < countExplosions; i++) {
        //če je število risanj te eksplozije, ki jih želimo opraviti > 0
        if(howManyRepaints.get(i) != 0) {
            //če je število risanj te eksplozije > 4 rišemo le-to večjo
            if(howManyRepaints.get(i) > 4) {
                    //rišemo sovražnika, ki je bil uničen s pomočjo enemyType-a ->
                    //nam pove kateri je bil
                    if(enemyType.get(i) == 1) {
                        g.drawImage(enemyShip, explosionsX.get(i),explosionsY.get(i), null);
                    }else if(enemyType.get(i) == 2) {
                        g.drawImage(enemyShip2, explosionsX.get(i),explosionsY.get(i), null);
                    }else if(enemyType.get(i) == 3) {
                        g.drawImage(enemyShip3, explosionsX.get(i),explosionsY.get(i), null);
                    }
                    if(enemyType.get(i) == 3) {
                        g.drawImage(explosion, explosionsX.get(i)+15,explosionsY.get(i)+5+5, 40, 40, null);
                    }else {
                        g.drawImage(explosion, explosionsX.get(i)+5,explosionsY.get(i)+5, 40, 40, null);
                    }
            //če je število risanj te eksplozije > 2 rišemo le-to nekoliko manjšo
            }else if(howManyRepaints.get(i) > 2) {

                if(enemyType.get(i) == 3) {
                    g.drawImage(explosion, explosionsX.get(i)+19,explosionsY.get(i)+9+5, 33, 33, null);
                }else {
                    g.drawImage(explosion, explosionsX.get(i)+9,explosionsY.get(i)+9, 33, 33, null);
                }
            //drugače na nižemo eksplozijo še manjšo
```

Slika 40: Izrisovanje eksplozij sovražnikov tipa 1, 2 in 3:

Enako kot na sliki 40 se zgodi tudi pri risanju eksplozij boljših sovražnikov (funkcija vidna na sliki 41), le da imamo tu več eksplozij namesto ene same. Pri tem pa se izrisovanje zgodi natanko takrat, ko je boljši sovražnik v fazi uničenja in igra trenutno ni v načinu pavze.

```
//risanje eksplozij boljšega sovražnika --> wait nastavimo na true, da jih narišemo, ko pa  gre
//dol s screena oz. je število ponovitev eksplozij = 0, nastavimo wait nazaj na false
//za normalno delovanje igre
if(wait) {
    for(int i = 0; i < countExplosionsBoss; i++) {
        if(howManyRepaintsBoss.get(i) != 0) {
            if(howManyRepaintsBoss.get(i) > 4) {
                g.drawImage(explosion, explosionsBossX.get(i)+5,explosionsBossY.get(i)+5, 40, 40, null);
            }else if(howManyRepaintsBoss.get(i) > 2) {
                g.drawImage(explosion, explosionsBossX.get(i)+9,explosionsBossY.get(i)+9, 33, 33, null);
            }else{
                g.drawImage(explosion, explosionsBossX.get(i)+12,explosionsBossY.get(i)+12, 26, 26, null);
            }
            if(!pauseMenu) {
                int count = howManyRepaintsBoss.get(i);
                count--;
                howManyRepaintsBoss.remove(i);
                howManyRepaintsBoss.add(i, count);
            }
        }else {
                howManyRepaintsBoss.remove(i);
                explosionsBossX.remove(i);
                explosionsBossY.remove(i);
                countExplosionsBoss--;
                //i = -1;
                wait = false;
        }
    }
}
```

Slika 41: Eksplozije boljših sovražnikov

3.1.5 Timerji

»*Timer* je razred, ki sproži enega ali več *AcionEventov* v določenem intervalu.« (18) V igri sem uporabil 6 ralzičnih *Timerjev*, vidne na sliki 42, in sicer jih 5 uporabljam za pravilno delovanje igre (*t, shipTimer, difficultyTimer, bossAbilityTimer* in *powerUpTimer*) ter enega za menije (*menuTimer*), ki le skrbi za ponovno risanje grafike na zaslon s klicanjem funkcije *repaint*. Razred *Timer* kot parametre sprejme časovni interval, v katerem bo sprožal funkcijo *actionPerformed*, ki kot parameter potrebuje *ActionEvent* objekt, in objekt vmesnika *ActionListener* (moj razred ta vmesnik implementira).

```
Timer t = new Timer(15, (ActionListener) this);
Timer shipTimer = new Timer(time, (ActionListener) this);
Timer menuTimer =  new Timer(15, (ActionListener) this);
Timer difficultyTimer =  new Timer(20000, (ActionListener) this);
Timer bossAbilityTimer = new Timer(3000, (ActionListener) this);
Timer powerupTimer = new Timer(20000, (ActionListener) this);
```

Slika 42: Deklaracija in inicializacija Timerjev

3.1.5.1 ShipTimer

Timer z imenom *shipTimer*, skrbi za to, da se vsake toliko časa, kolikor smo v deklaraciji specificirali, ustvari nov sovražnik. Del algoritma, ki skrbi za to je tudi viden na sliki 43, kjer glede na naključno vrednost generira sovražnika tipa 1, 2 ali pa 3. Generira ga tako, da mu dodeli X in Y koordinato, ter druge potrebne vrednosti za pravilno delovanje in izrisovanje le-tega na zaslon. *Timer* deluje identično v primeru ko igralec igra neskončen način kot tudi način zgodbe, razlike je le v tem, da se v neskončnem načinu tipi sovražnikov generirajo glede na težavnost, ki se povečuje s časom, v načinu zgodbe pa glede na številko nivoja na katerem se trenutno igralec nahaja.

```
if(e.getSource() == shipTimer && !boss) {
    if(time == 20) {
        time = 2000;
        shipTimer.setDelay(time);
    }else {
        timerPauseAmount = 2000;
        if(level3) {
            System.out.println("making minions");
            int num = (int) (Math.random()*41);
            if(num >= 23) {
                enemyShip3X.add((int) (Math.random()*651));
                enemyShip3Y.add(-50);
                enemyShip3Health.add(1);
                enemyShip3Hit.add(0);
                enemyShip3AttackTime.add(0);
                countEnemySpaceships3++;
            }else if(num >= 10){
                enemyShip2X.add((int) (Math.random()*671));
                enemyShip2Y.add(-50);
                enemyShip2Health.add(1);
                enemyShip2Hit.add(0);
                countEnemySpaceships2++;
            }else {
                enemyShipX.add((int) (Math.random()*671));
                enemyShipY.add(-50);
                countEnemySpaceships++;
            }
```

Slika 43: Del algoritma za katerega je odgovoren shipTimer

3.1.5.2 BossAbilityTimer

Skrbi za to, da ko je v igri boljši sovražnik določenega tipa, lahko le-ta napada igralno figuro s 3 različnimi napadi na vsak časovni interval, ki je bil določen v ustvarjanju objekta *bossAbilityTimer*. Del algoritma, ki je viden na sliki 44, se zažene natanko takrat, ko ima sovražnik več kot 0 življenjskih točk in je funkcijo *actionPerformed* vzbudil ta *Timer*. Preden se izvede izbira napada počistimo laserje iz zaslona (primarni napad boljšega sovražnika tipa 2). Napad, katerega bo izkoristil pa je izbran naključno, saj s pomočjo funkcije *random* izberemo število od 0 do 2, te pa predstavljajo ustvarjanje navadnih sovražnikov (če na zaslonu ni nobenega navadnega sovražnika), primarni napad in pa sekundarni napad.

```
}else if(e.getSource() == bossAbilityTimer && bossHealth != 0) {//rather make a function
    bossAbilityTimeInitialAmount = 3000;
    bossAbilityTimer.setDelay(bossAbilityTimeInitialAmount);
    enemyLasersY.clear();enemyLasersX.clear();countEnemyLasers = 0;
    moveDrone0 = false;moveDrone1 = false;moveDrone2 = false;moveDrone3 = false;
    int num = (int) (Math.random()*3);//*3 here
    if(num == 2) {
        if(enemyShipY.isEmpty() && enemyShip2Y.isEmpty() && enemyShip3Y.isEmpty()) {//
            primary = false;
            secondary = false;
            createMinions();
        }else {
            num = (int) (Math.random()*2);
            if(num == 0) {
                if(level2) {
                    positions.clear();
                    for(int i = 1; i < 9; i++) {
                        positions.add(i);
                    }
                    byte num2 = (byte) (Math.random()*4);
                    positions.remove(num2);
                    num2 = (byte) (Math.random()*3);
                    positions.remove(num2);
                    num2 = (byte) (Math.random()*4 + 2);
                    positions.remove(num2);
                    num2 = (byte) (Math.random()*3 + 2);
                    positions.remove(num2);
                }
                primary = true;
                secondary = false;
```

Slika 44: Del algoritma, za katerega je odgovoren bossAbilityTimer:

3.1.5.3 PowerupTimer

Ta Timer potrebujem pri neskončnem načinu igranja kot pri končnem načinu, skrbi pa za algoritem, kjer vsak časovni interval, ki sem ga določil pri deklaraciji in inicializaciji objekt, ustvari novo začasno nadgradnjo, ki jo igralna figura lahko pridobi tako, da se vanjo »zaleti«. Na sliki 45 je prikazan del kode, kjer sprva naključno s pomočjo funkcije random izberemo število med od 0-3, te pa nato narekujejo, kater tip začasne nadgradnje se bo pojavil na zaslonu. Ker pa so boljši sovražniki tipa 1, 2 in 3 drugačne velikosti, se X komponenta nadgradnje nastavi na ustrezno lokacijo.

```
}else if(e.getSource() == powerupTimer) {
    byte num = (byte) (Math.random()*4);
    if(num == 0) {
        selectedPowerup = speedPowerup;
    }else if(num == 1) {
        selectedPowerup = fireRatePowerup;
    }else if(num == 2) {
        selectedPowerup = healthPowerup;
    }else {
        selectedPowerup = shieldPowerup;
    }
    byte num2 = (byte) (Math.random()*2);
    if(boss && level3) {
        if(num2 == 1) {
            powerupX = (int) (Math.random()*((this.getWidth()-200)/2-8-40))
        }else {
            powerupX= (int) (Math.random()*((this.getWidth()-200)/2-8-40));
        }
    }else if(boss && level2) {
        if(num2 == 1) {
            powerupX = (int) (Math.random()*((this.getWidth()-230)/2-8-40))
        }else {
            powerupX = (int) (Math.random()*((this.getWidth()-230)/2-8-40))
        }
    } else if(boss) {
        if(num2 == 1) {
            powerupX = (int) (Math.random()*((this.getWidth()-270)/2-8-40))
```

Slika 45: Del algoritma, za katerega je odgovoren powerupTimer

3.1.5.4 Glavni Timer

Glavni *Timer* v igri je *Timer* z imenom *t*. Ta skrbi, da v primerih, ko igralec ni v nobenem izmed menijev, igra gladko poteka. To sem dosegel z zaporedjem ukazov, katerih majhen del je možno videti na sliki 46. Glavne funkcije, ki jih ta algoritem izvaja so: ponastavitev začasnih nadgradenj, zmanjšanje vrednosti spremenljivk, ki vsebujejo čas do izteka določenega dogodka (npr. čas streljanja za igralno figuro in sovražnikov), ustvarjanje boljših sovražnikov, klicanje drugih funkcij, ki opravljajo premike sovražnikov, funkcij za streljanje (če so pogoji izpolnjeni), za preverjanje trkov in pa najpomembnejše, posodabljanje grafike na zaslonu s klicem funkcije *repaint*. Nekatere izmed funkcij katere algoritem kliče je moč videti tudi na sliki 46, opisani pa so bili v sklopu 3.1.2 in 3.1.3.

```
                countEnemyBullets2--;
                countBullets2Explosions++;
            }
            moveEnemyBullets2();
            checkEnemyBullets2MissileCollision();
            checkEnemyBullets2Collision();
        }else {
            moveEnemyMissiles();
            if(attackTime <= 0 && secondary) {
                createEnemyBullet();
                attackTime = 300;
            }
            moveEnemyBullets();
            checkEnemyMissilesCollision();
            checkEnemyBulletsCollision();
        }
        moveSpaceImage();
        movePowerup();
        checkPowerupCollision();
        movePicture();
        createMissile();
        moveMissiles();
        moveEnemySpaceship();
        repaint();
        checkMissileCollision();
        checkEnemySpaceshipCollision();
    }
}
repaint();
```

Slika 46: Del algoritma, za katerega je odgovoren glavni timer t

3.1.5.5 DifficultyTimer

Ta *Timer* je pomemben samo takrat, ko je igralec v neskončnem načinu igre. Skrbi za to, da se stopnja težavnosti igre s časom povečuje, ter tako »popestri« igro kot je vidno na sliki 47. DifficultyTimer deluje tako, da se vsak časovni interval, določen pri deklaraciji in inicializaciji objekta, zmanjša čas med ustvarjanjem sovražnikov, pri tem pa se poveča časovni interval za 10s (da igra ne postane pretežka prehitro). Prav tako pa obstaja tudi najvišja stopnja, težavnosti, katero dosežemo, saj bi v nasprotnem primeru čas med kreiranjem dveh objektov sovražnikov postal prekratek in tako bi igra postala nemogoča

```
}else if(e.getSource() == difficultyTimer && amount <10) {
    amount++;
    time -= 150;
    shipTimer.setDelay(time);
    difficultyTimer.setDelay(difficultyTimer.getDelay()+10000);
    difficultyTimerInitialAmount = difficultyTimer.getDelay();
}else if(e.getSource() == powerupTimer) {
```

Slika 47: Algoritem, za katerega je odgovoren difficultyTimer

3.2 Izdelava s pomočjo igralnega pogona Unity

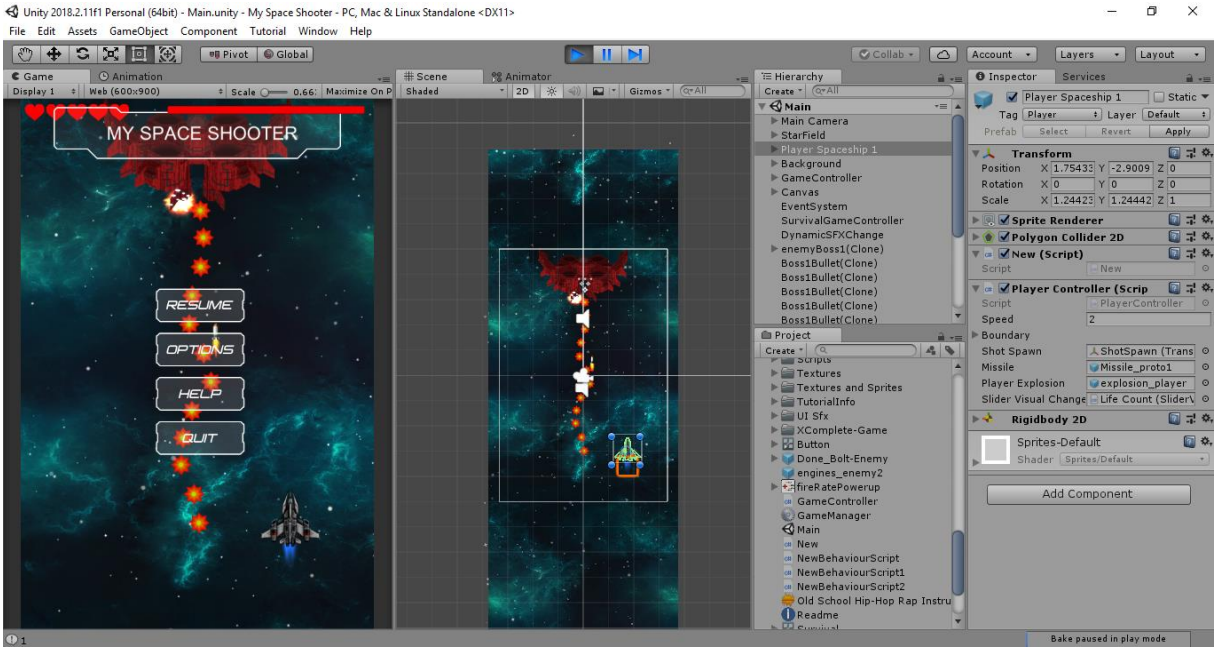
Igro, izdelano v igralnem pogonu Unity ne bom opisal tako podrobno, kot se opisal delovanje le-te v programskem jeziku Java, ampak se bom posvetil predstavitvi glavnih razlik, katere sem sam tudi izkusil, med izdelavo v igralnem pogonu in v programskem jeziku brez zunanjih knjižnic. Podobno kot pri opisu igre v Javi, bom tudi tukaj delovanje igre razdelil na manjše dele, katere bom okvirno predstavil, saj je logičen del bil predstavljen že pri opisu izdelave igre v programskem jeziku Java. Ti deli bodo bili:

- **Grafični urejevalnik Unity**
- **Objekt igralna figura**
- **Skrbniki igre in sovražniki**
- **Uporabniški vmesnik**
- **Zvočni in vizualni efekti**

3.2.1 Grafični urejevalnik Unity

Grafični urejevalnik Unity uporabnik nudi vizualno pomoč pri ustvarjanju, prav tako pa ima tudi veliko uporabnik funkcij, ki prihranijo veliko časa. Na sliki 48 je posnetek zaslona, kjer imam odpri urejevalnik Unity. Iz slike so razvidni okna pogled igranja (*Game*), pogled scene (*Scene*), hierarhija igralnih objektov, ki so znotraj scene, pod tem je okno, kjer je vsebina trenutno odprtega projekt (vse objekti igre, zvočni posnetki, slike in druge stvari), čisto na desni strani pa je še okno z imenom *Inspector*. Zgoraj na sredini so gumbi za zagon igralnega načina, za zaustavitev igralnega načina in pa gumb za delovanje korak za korakom.

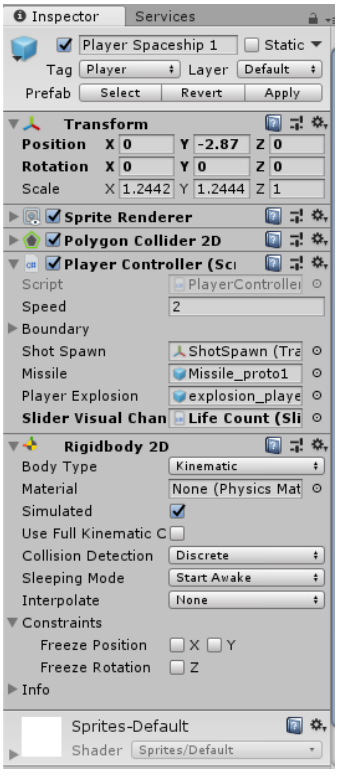
Igralni način je eden izmed najbolj uporabnih stvari pri testiranju igre, saj vidimo, kako je videti igra kot bi jo videl igralec. Igralni način tudi omogoča, da spreminjamo vrednosti spremenljiv v inšpektorju, ter tako hitreje in enostavneje pridemo do najboljše vrednosti za le-to. Hierarhija igralnih objektov nam omogoča dodajanje novih objektov, če pa jih hočemo shraniti kot igralne objekte, ki imajo že v naprej določene komponente oziroma v Unityju so ti poimenovani »*Prefabs*«, pa to storimo tako, da enostavno objekt igre dodamo v projektno okno. Inšpektor pa nam omogoča dodajanje veliko različnih komponent na igralne objekte (*Rigidbody, Collider, AnimatorController, AudiSource* itd.), omogoča tudi nastavljanje javnih spremenljivk skript, spreminjanje pozicije in rotacije igralnega objekta, urejanje dodanih komponent ter še veliko več.



Slika 48: Posnetek zaslona urejevalnika Unity

3.2.2 Objekt igralna figura

Objekt igralna figura, katerega informacije vidimo v inšpektorju na sliki 49, ima 5 komponent in sicer: *Transform* (lokacija in rotacija objekta), *Sprite Renderer* (slika, s katero ta objekt predstavljamo v igri), *Polygon Collider 2D* (je nastavljen kot *Trigger* in je obroba, s katero lahko ostali objekti s tem navidezno trčijo), *PlayerController* (razred, ki vsebuje algoritme figure in ima nekaj javnih spremenljivk, vidnih na sliki 49), ter *RigidBody 2D* (objektu dodelimo fizično prisotnost v igralnem pogonu – nanj zdaj igralni pogon lahko deluje, figura pa je nastavljena kot kinematično telo – nanj gravitacija ne deluje).



Slika 49: Lastnosti igralne figure

PlayerController vsebuje 3 pomembne funkcije, katere so klicane avtomatsko. Te so:

- **void Start** – poklicana samo enkrat, ko je objekt igre ustvarjen v sceni, uporabljamo za inicializacijo spremenljivk
- **void Update** – izvršena enkrat na vsako sliko (traja različno dolgo)
- **void FixedUpdate** – izvršena enkrat na fizični korak (traja enako časa)

Znotraj funkcije Start, vidne na sliki 50, nastavimo vrednosti dveh objektov tipa *SurvivalGameControler* in *GameController* tako, da uporabimo vgrajeno funkcijo, ki poišče objekt igre z določeno oznako in nato uporabimo še funkcijo, ki pridobi ustrezno komponento dobljenega objekta. Poleg tega pa pokličemo še funkcijo iz razreda *SliderVisualChange* (skrbi za prikazovanje drsnikov), ki nastavi vizualno vrednost življenjskih točk na vrednost, ki ga ji damo v parameter (beseda *ref* pomeni, da ji kot parameter posredujemo originalno spremenljivo – če se spremeni ta spremenljivka v tem razredu, se spremeni tudi v drugem razredu). Pridobimo tudi *RigidBody2D* komponento tega 2D objekta, ki jo nato uporabimo v funkciji *FixedUpdate*.

Znotraj funkcije *Update* v primeru da igralec tišči tipko za streljati in mu je dovoljeno streljati, ustvarimo nov objet v sceni in sicer s pomočjo funkcije *Instantiate*. Ta funkcija kot parametre vzame objekt igre, ki ga hočemo ustvariti, pozicijo ter rotacijo.

```
void Start() {
    resetSpeed = speed;
    resetPosition = this.transform.position;

    survivalGameController = GameObject.FindGameObjectWithTag("SurvivalGameController").GetComponent<SurvivalGameController>();
    gameController = GameObject.FindGameObjectWithTag("GameController").GetComponent<GameController>();
    sliderVisualChange.CallActivateLifeCount(ref health);
    rb = GetComponent<Rigidbody2D>();
}
private void OnEnable() {
    sliderVisualChange.CallActivateLifeCount(ref health);
}
// Update is called once per frame
void Update() {
    if (Input.GetButton("Fire1") && Time.time >= nextFire && Time.timeScale == 1) {
        nextFire = Time.time + fireRate;
        Instantiate(missile, shotSpawn.position, shotSpawn.rotation);
    }
}
//will be executed once per physics step
void FixedUpdate() {
    float moveHorizontal = Input.GetAxis("Horizontal");
    float moveVertical = Input.GetAxis("Vertical");
    rb.velocity = new Vector2(moveHorizontal, moveVertical) * speed;
    rb.position = new Vector2(
        Mathf.Clamp(rb.position.x, boundary.xmin, boundary.xmax),
        Mathf.Clamp(rb.position.y, boundary.ymin, boundary.ymax)
    );
}
```

Slika 50: Funkcije Start, Update in FixedUpdate v razredu PlayerController

V *FixedUpdate* (na sliki 50), pa igralca premikamo v smeri X in Y s tem, da beremo podatke s pomočjo razreda *Input* (dalj časa kot se tišči tipka, večja je vrednost), objekt pa premikamo tako, da *RigidBody2D* komponenti nastavimo hitrost, pri tem pa ne želimo, da bi igralec lahko šel izven igralnega območja, zato vrednosti lokacij omejimo s funkcijo *Clamp*, ki vzame 3 parametre – trenutna lokacija, najmanjša dovoljena ter največja dovoljena lokacija.

Funkcije, ki so del razreda, razvidne iz slike 51 spremenijo vrednost življenjskih točk figure oziroma število ščitov, ter igralcu prikažemo indikator da je bil zadet (če je bil) ter igralno figuro naredimo neranljivo za določen čas v funkciji *CallInvurnerable*, ki zažene Coroutino z imenom *Invulnerable*. »Coroutina je kot funkcija, ki ima možnost prekiniti izvajanje in vrniti nadzor Unityju, nato pa nadaljevati tam, kjer je končala v naslednjem okviru.« (19) V mojem primeru se le-ta zažene, nastavi neranljivost na *true*, nato vrne nadzor Unityju za 1 sekundo in po 1 sekundi nastavi neranljivost nazaj na *false* in se zaustavi. Funkciji *CallTintChange* in *TintChange* delujeta identično kot funkciji za neranljivost, le da se tu spreminja barvni odtenek slike objekta.

```
public void ChangeHealth(float amount) {
    if (!gameObject.activeInHierarchy) { return; }
    //float health = sliderChange(ref health, amount);
    sliderVisualChange.CallHealthChange(ref health, amount, ref isInvulnerable, enemyDamage, ref shieldCount);
    if (health <= 0) {
        DestroySelf();
    }
}
public void ChangeShield(int amount) {
    if (!gameObject.activeInHierarchy) { return; }
    sliderVisualChange.CallShieldChange(ref shieldCount, amount);
}

public void CallInvulnerable() {
    if (!gameObject.activeInHierarchy) { return; }
    if (!isInvulnerable) {
        StartCoroutine("Invulnerable");
    }
}

IEnumerator Invulnerable() {
    isInvulnerable = true;
    yield return new WaitForSeconds(1);
    isInvulnerable = false;
}
```

Slika 51: Funkcije, ki se izvedejo ob trku igralne figure z drugim objektom igre

V primeru, da igralni figuri zmanjka število življenjskih točk, se zažene funkcija na sliki 52 z imenom *DestroySelf*, ki objekt nastavi na neaktivnega, saj tako omogočimo, da lahko igralec ponovno poskusi z igranjem. Igro zaustavimo tako, da ustavimo vse Coroutine, resetiramo pozicijo igralne figure, ponastavimo njegovo statistiko (potrebno, saj je igralec lahko igral neskončni način) in onemogočimo prikaz življenjskih točk (boljšega sovražnika kot tudi igralne figure). Obenem pa igralcu pokažemo tudi indikator, da je bil poražen tako, da ustvarimo eksplozijo na mestu igralne figure.

```
private void DestroySelf() {
    if (survivalGameController.survival) {
        survivalGameController.EndGame();
    } else {
        gameController.endGame();
    }
    gameController.bossHealthSlider.gameObject.SetActive(false);
    Instantiate(playerExplosion, transform.position, transform.rotation);
    StopAllCoroutines();

    gameObject.SetActive(false);

    speed = resetSpeed;
    health = 5f;
    isInvulnerable = false;
    enemyDamage = 0f;
    nextFire = 0f;
    fireRate = 0.7f;
    transform.position = resetPosition;

    sliderVisualChange.CallDeactivateLifeCount();
    //shieldCount = 0; //not needed
    //Destroy (gameObject);
}
```

Slika 52: Funkcija, ki uniči dezaktivira igralno figuro

Igralna figura vsebuje tudi še funkcije za začasno nadgradnjo in trajno nadgradnjo (neskončni način igre). Na sliki 53 sta vidni funkciji, ki za 10 sekund omogočita izboljšavo za igralno figuro, nato pa spet vrednost vrneta na osnovno. Vredno je omeniti tudi, da razred PlayerController ne vsebuje funkcij za zaznavanje navideznih trkov, saj to preverjajo objekti sovražnikov.

```
public IEnumerator TemporarySpeedChange() {
    speed += 1f;
    yield return new WaitForSeconds(10f);
    speed -= 1f;
}
public IEnumerator TemporaryFireRateChange() {
    fireRate -= 0.35f;
    yield return new WaitForSeconds(10f);
    fireRate += 0.35f;
}
```

Slika 53: Algoritma za začasne nadgradnje

3.2.3 Skrbniki igre in sovražniki

Skrbniki igre so razredi, ki nadzirajo potek celotne igre. V moji igri imam 3 take razrede in sicer so to:

- ***GameManager*** (skripta, ki nadzira prikaz menije in ponovni zagon igre → neskončni način ali način zgodbe)
- ***GameController*** (nadzira delovanje igre načina zgodbe)
- ***SurvivalGameController*** (nadzira delovanje igre v neskončnem načinu)

3.2.3.1 GameManager

Kot že omenjeno, razred *GameManager* nadzira prikazovanje menijev (slika 54). To opravlja znotraj funkcije *Update*, kjer glede na trenutno stanje igre nastavlja menije aktivne v hierarhiji, ali pa jih dezaktivira. Prav tako skrbi, da v primeru da je aktiven končni meni in igralec pritisne tipko 'R' se igra ponastavi (če igralec igra neskončen način, se ponastavi le-ta podobno se zgodi tudi če igra način zgodbe).

```
void Update () {
    if (gameController.GameWon) { gameWonMenu.SetActive(true); return; }
    if ((gameController.GameOver || SurvivalGameController.GameOver) && Input.GetKeyDown(KeyCode.R) && canRestart) {
        if (survivalGameController.survival){
            survivalGameController.RestartGame();
        }
        else{
            gameController.restartGame();
        }
        gameOverMenu.SetActive(false);
        return;
    }
    if (gameController.GameOver || SurvivalGameController.GameOver) {
        if (gameOverMenu.activeInHierarchy) {
            return;
        }
        canRestart = false;
        StartCoroutine(Restart());
        gameOverMenu.SetActive(true);
        return;
    }
    if ((Input.GetKeyDown(KeyCode.P) || Input.GetKeyDown (KeyCode.Escape)) && !pause && !upgrade && !mainMenu.activeInHierarchy
        pauseMenu.SetActive (true);
        Time.timeScale = 0;
        pause = true;
        audios[0].Pause ();
        audios[1].UnPause ();
    } else if((Input.GetKeyDown (KeyCode.P) || Input.GetKeyDown (KeyCode.Escape)) && pauseMenu.activeInHierarchy){
        pauseMenu.SetActive (false);
        Time.timeScale = 1;
        pause = false;
```

Slika 54: Naloge razreda GameManager

3.2.3.2 GameController in SurvivalGameController

Razred *GameController*, viden na sliki 55, ob aktiviranju aktivira igralca in zažene *coroutini* z imenom *Spawner* in *PowerUpSpawner*. Prva izmed naštetih gre skozi en cikel v katerem ustvarja sovražnike in nato še boljšega sovražnika. Če ima igralna figura po izteku tega cikla še vedno nekaj življenjskih točk, je igralec igro premagal. Na spodnji sliki je videl del algoritma, ki ustvarja sovražnike s pomočjo zanke, časovni interval pa dodelim tako, da kontrolo vrnem nazaj Unityju za toliko časa, kolikor hočem da je časovne razlike med kreiranjem dveh objektov sovražnika.

```
public void Activate(){
    player.SetActive (true);
    StartCoroutine ("Spawner");
    StartCoroutine ("PowerUpSpawner");
}

IEnumerator Spawner(){
    audios[1].Pause ();
    audios[0].Play();
    yield return new WaitForSeconds (startWait);
    for (int i = 0; i < 1/*was 4 before*/; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            Instantiate(enemy1, new Vector3(Random.Range(minX, maxX), positionY, 0), transform.rotation);
            yield return new WaitForSeconds(enemyWait);
        }
    }
    Instantiate(boss1, boss1Spawn.position, boss1Spawn.rotation);//instantiate boss1
    bossHealthSlider.maxValue = 50;
    bossHealthSlider.value = 50;
    bossHealthSlider.gameObject.SetActive(true);
    boss1Alive = true;
    yield return new WaitUntil(() => !boss1Alive);
    bossHealthSlider.gameObject.SetActive(false);
    yield return new WaitForSeconds(3);
    for (int i = 0; i < 1/*4*/; i++)...//USTVARJANJE
    Instantiate(boss2, boss2Spawn.position, boss2Spawn.rotation);//instantiate boss2
    bossHealthSlider.maxValue = 75;
    bossHealthSlider.value = 75;
    bossHealthSlider.gameObject.SetActive(true);
    boss2Alive = true;
```

Slika 55: Funkcija, ki ustvarja sovražnike

PowerUpSpawner, na sliki 56, pa deluje natanko tako kot pri igri, narejeni v Javi, le da tukaj namesto *Timerjev* uporabljamo *Coroutino* in ključno besedo *yield return* new *WaitForSeconds*(čas) - vrne nadzor glavni niti in nato prevzame kontrolo nazaj po poteku specificiranega časa.

```
private IEnumerator PowerUpSpawner(){
    yield return new WaitForSeconds (30f);
    while(true){
        int which = Mathf.FloorToInt (Random.Range(0f, 3.99f));
        if (boss1Alive) {
            int which1 = Mathf.FloorToInt (Random.Range(0f, 1.99f));
            if (which1 == 0) {
                Instantiate (powerUps [which], new Vector2(Random.Range (boss1PosXMin1, boss1PosXMax1), positionY),
            } else {
                Instantiate (powerUps [which], new Vector2(Random.Range (boss1PosXMin2, boss1PosXMax2), positionY),
            }
        } else if (boss2Alive) {
            int which1 = Mathf.FloorToInt (Random.Range(0f, 1.99f));
            if (which1 == 0) {
                Instantiate (powerUps [which], new Vector2(Random.Range (boss2PosXMin1, boss2PosXMax1), positionY),
            } else {
                Instantiate (powerUps [which], new Vector2(Random.Range (boss2PosXMin2, boss2PosXMax2), positionY),
            }
        } else if (boss3Alive) {
            int which1 = Mathf.FloorToInt (Random.Range(0f, 1.99f));
            if (which1 == 0) {
                Instantiate (powerUps [which], new Vector2(Random.Range (boss3PosXMin1, boss3PosXMax1), positionY),
            } else {
                Instantiate (powerUps [which], new Vector2(Random.Range (boss3PosXMin2, boss3PosXMax2), positionY),
            }
        } else {
            Instantiate (powerUps [which], new Vector2(Random.Range (minX, maxX), positionY), powerUps[which].trans
        }
        yield return new WaitForSeconds (30f);
    }
```

Slika 56: Funkcija, ki na določene časovne intervale ustvari začasno nadgradnjo

SurvivalGameController pa deluje skoraj natanko tako kot razred *GameController* le da imata nekolikšne variacije glede funkcij, ki jih vsebujeta. Med funkcijami za skrbnika neskončnega načina igre je tudi funkcija *UpgradeDifficulty*, ki tudi deluje na enak način kot v igri napisani v Javi. Poleg le-te pa vsebuje tudi funkcije za dodajanje točk in pridobitev teh. Na sliki 57 sta prikazani funkciji za končanje in ponastavitev igre za način zgodbe, pri neskončnemu načinu pa je razlika le v tem, da nam ni potrebno ponastaviti spremenljivk, ki so vezane na boljše sovražnike. Funkcija *endGame* deluje tako, da zaustavimo vse *coroutine* in napade boljših sovražnikov, funkcija za ponovno igranje igre pa kot ime samo pove, ponastavi vse elemente, ki so bili spremenjeni, na začetno stanje in izbriše vse objekte, ki so trenutno prisotni na sceni in sprva na njej niso bili (ustvarjeni objekti preko skript).

```
public void endGame(){
    GameOver = true;
    GameWon = false;
    StopAllCoroutines ();
    //audios[1].Stop();
    if (boss1Alive) {
        GameObject.FindGameObjectWithTag("Boss1").GetComponent<Boss1Script>().StopAttacking();
    } else if (boss2Alive) {
        GameObject.FindGameObjectWithTag("Boss2").GetComponent<Boss2Script>().StopAttacking();
    } else if (boss3Alive) {
        GameObject.FindGameObjectWithTag("Boss3").GetComponent<Boss3Script>().StopAttacking();
    }
}
public void restartGame() {
    GameOver = false;
    GameWon = false;
    boss1Alive = false;
    boss2Alive = false;
    boss3Alive = false;
    player.SetActive(true);
    StartCoroutine("Spawner");
    StartCoroutine("PowerUpSpawner");
    List<GameObject> objectsToBeRemoved = new List<GameObject>();
    objectsToBeRemoved.AddRange(GameObject.FindGameObjectsWithTag("Enemy Mine"));
    objectsToBeRemoved.AddRange(GameObject.FindGameObjectsWithTag("Enemy"));
    objectsToBeRemoved.AddRange(GameObject.FindGameObjectsWithTag("Enemy Bolt"));

    foreach (GameObject objectToBeRemoved in objectsToBeRemoved) {
        Destroy(objectToBeRemoved);
    }
```

Slika 57: Algoritma za ponovno igranje in končanje igre

3.2.3.3 Boljši sovražnik

Boljši sovražnik tipa 1, 2 in 3 delujejo zelo podobno, razlikujejo se le v tem, da imajo drugačne napade. Znotraj razreda *Boss1Script* so funkcije, ki upravljajo z objektom boljšega sovražnika tipa 1. Slika 58 prikazuje funkcije *Start*, *Update* in *FixedUpdate*. V *Start* pridobim objekt razreda *GameController* (uporabim, ko želim končati igro), ki je trenutno kot komponent objekta v sceni, nastavim vrednost življenjskih točk sovražnika in pridobim *RigidBody2D* komponento tega objekta.

```
void Start () {
    health = 50;
    gameController = GameObject.FindGameObjectWithTag("GameController").GetComponent<GameController> ();
    rb = GetComponent<Rigidbody2D> ();
}

// Update is called once per frame
void Update () {
    if(transform.position.y <= 4 && move){
        move = false;
        StartCoroutine ("Attack");
    }
    if(health <= 0 && !moveUp){
        moveUp = true;
        moveSpeed *= -1;
    }else if(moveUp && health <= 0 && transform.position.y >= 6.2){
        Destroy (gameObject);
    }
}

void FixedUpdate(){
    if (move) {
        rb.velocity = new Vector2 (0f, moveSpeed);
        return;
    } else if (moveUp) {
        rb.velocity = new Vector2 (0f, moveSpeed);
        return;
    }
        rb.velocity = new Vector2 (0f, 0f);
}
```

Slika 58: Funkcije Start, Update in FixedUpdate razreda Boss1Script

To nato potrebujem pri *FixedUpdate* funkciji, kjer objekt premikam, če je še izven zaslona oziroma če je število njegovih življenj 0 ga premikam izven zaslona. *Update* pa vsebuje zaporedje ukazov, ki, ko je objekt na končni poziciji, boljšemu sovražniku dovolijo napadanje (funkcija vidna na sliki 58) , v primeru, da nima več življenjskih točk, spremenijo smer gibanja in objektu dovolijo, da se začne premikati ter ko je izven zaslona, se objekta znebim tako, da uporabim funkcijo *Destroy(objekt),* ki le-tega odstrani iz scene.

Coroutina Attack objektu omogoča, da na določene časovne intervale uporabi enega izmed svojih treh napadov (slika 60). To naredi natanko tako, kot algoritem, napisan v programskem jeziku Java, in sicer naključno izbere število od 0 do 2 in glede na to pokliče funkcijo, kateri pripada to število (slika 59). V primeru, da je klicana funkcija *PrimaryAttack*, v hierarhiji dodamo 5 objektov tipa sovražnikovih izstrelkov. *SecondaryAttack* pa vsak časovni interval ustvarja drugačne objekte (še vedno so izstrelki), če zadošča pogoj, da boljši sovražnik trenutno koristi sekundarni napad. *CreateMinions* deluje natanko tako kot deluje algoritem za ustvarjanje sovražnikov znotraj *GameController* razreda le da so tukaj omejitve glede začetne lokacije (ne sme se ustvariti tam, kjer je trenutno boljši sovražnik). V primeru ko je igralec poražen, se izvede funkcija *StopAttacking* (del algoritma viden na sliki 59), ki skrbi, da se vse trenutne coroutine zaustavijo in se tako boljši sovražnik odstrani iz hierarhije (premakne se izven scene in se uniči).

```
private IEnumerator PrimaryAttack(){// do we actually want it to be a coroutine?
    Instantiate(missile, attack1Spawn1.position, attack1Spawn1.rotation);
    yield return new WaitForSeconds (0.2f);
    Instantiate(missile, attack1Spawn2.position, attack1Spawn2.rotation);
    Instantiate(missile, attack1Spawn3.position, attack1Spawn3.rotation);
    Instantiate(missile, attack1Spawn4.position, attack1Spawn4.rotation);
    Instantiate(missile, attack1Spawn5.position, attack1Spawn5.rotation);
}
private IEnumerator SecondaryAttack(){//has to be an IEnumerator
    while(secondaryAttack){
        Debug.Log("Shooting");
        Instantiate (bullet, bulletSpawn.position, bulletSpawn.rotation);
        yield return new WaitForSeconds (secondaryAttackRate);//maybe change this
    }
}
private IEnumerator CreateMinions(){
    for(int i = 0; i < 5/*maybe hange this value dynamically -> based on the health of the boss??*/; i++){
        int which = Mathf.FloorToInt (Random.Range(0, 1.99f));
        if (which == 0) {
            Instantiate (minion, new Vector2(Random.Range(spawnMin1, spawnMax1), 5.5f), gameObject.transform.rotation);
        } else...
        yield return new WaitForSeconds (minionsWait/*change dynamically?*/);
    }
    minions = false;
}

public void StopAttacking() {
    if (gameController.GameOver)
    {
        health = 0;
        secondaryAttack = false;
        StopCoroutine("Attack");
```

Slika 59: Funkcije za primarni in sekundarni napad ter ustvarjanje sovražnikov:

```
private IEnumerator Attack(){
    while(!gameController.GetBoss1Destroyed()){
        int which = Mathf.FloorToInt(Random.Range (0, 2.99f)); Debug.Log(which);
        if (which == 0){
            secondaryAttack = false;
            StartCoroutine ("PrimaryAttack");
        }else if(which == 1){
            if(!secondaryAttack){
                secondaryAttack = true;
                StartCoroutine ("SecondaryAttack");
            }
        }else{
            if (!minions) {
                minions = true;
                secondaryAttack = false;
                StartCoroutine ("CreateMinions");
            } else {
                which = Mathf.FloorToInt(Random.Range (0, 1.99f));
                if(which == 0){
                    if(!secondaryAttack){
                        secondaryAttack = true;
                        StartCoroutine ("SecondaryAttack");
                    }
                }else{
                    secondaryAttack = false;
                    StartCoroutine ("PrimaryAttack");
                }
            }
        }
        yield return new WaitForSeconds (attackWait);
    }
}
```

Slika 60: Funkcija za napadanje boljših sovražnikov

OnTriggerEnter2D je funkcija, vidna na sliki 61, ki je vzbujena vsakič, ko objekt navidezno trči z drugim objektom (če je njegov *Collider* nastavljen kot *Trigger*). Vsebina le-te znotraj razreda *Boss1Script* skrbi za to, da v primeru, ko izstrelek igralne figure zadane ta objekt, se zmanjšaj število življenjskih točk za določeno število. Če se to število zmanjša na vrednost 0, se zažene *Explode*, ki ustvarja nove objekte – eksplozije na sceni dokler je boljši sovražnik znotraj kamere (tam, kjer se odvija igra).

```
void OnTriggerEnter2D(Collider2D other){
    if(other.CompareTag("Bolt")){
        Instantiate (smallExplosion, other.transform.position, other.transform.rotation);
        health--;
        gameController.ChangeBossHealth(-1, health);
        Destroy (other.gameObject);
        StartCoroutine("HitMarker");
        //maybe a sound and explosion?
    }
    if(health == 0 && !gameController.GetBoss1Destroyed() && other.CompareTag("Bolt")){
        Instantiate (smallExplosion, other.transform.position, other.transform.rotation);
        secondaryAttack = false;
        StopCoroutine ("Attack");
        StopCoroutine ("PrimaryAttack");
        StopCoroutine ("SecondaryAttack");
        StopCoroutine("CreateMinions");
        StartCoroutine ("Explode");
        gameController.Boss1Destroyed ();
        //destroy boss
        //add sound to those explosion -- mujltiple??
    }
}
private IEnumerator Explode(){
    while(transform.position.y <= 5.7){//i think i like it
        //could have done publlic floats(x only) and determined it like that.. maybe do it??
        Instantiate (bossExplosion, new Vector3(Random.Range(transform.position.x-0.9f, transform.position.x+0.9f),
        //play sound -- make sound first
        yield return new WaitForSeconds (0.3f);//could have explosionWait instead of fixed delay??
    }
    yield break;
}
```

Slika 61: Funkcija za preverjanje navideznih trkov

Ustvarjanje sovražnikovih izstrelkov in preverjanje kolizij z igralčevo figuro poteka na nekoliko drugačen način kot pri ustvarjanju le-teh v programskem jeziku Java. Razlikuje se predvsem le v sintaksi, saj sem v Unityju uporabljal v naprej določene objekte (*prefab*) , medtem ko sem v Javi imel X in Y koordinate in sliko, katero sem na tisti lokaciji narisal. Algoritem, ki je vsebovan v funkciji *Start* na sliki 62 prikazuje, da je način izračuna hitrosti v smeri X in Y enak kot v Javi, le da tukaj enostavno to le uporabimo s pomočjo lastnosti *velocity* v *RigidBody2D* komponenti objekta izstrelka. V primeru navideznega trka z igralčevo figuro (*OnTriggerEnter2D*) pa se objekt uniči, figuri pa odštejemo določeno število točk ter vzbudimo funkciji, ki skrbita za vizualni prikaz trka in za omogočanje začasne neranljivosti.

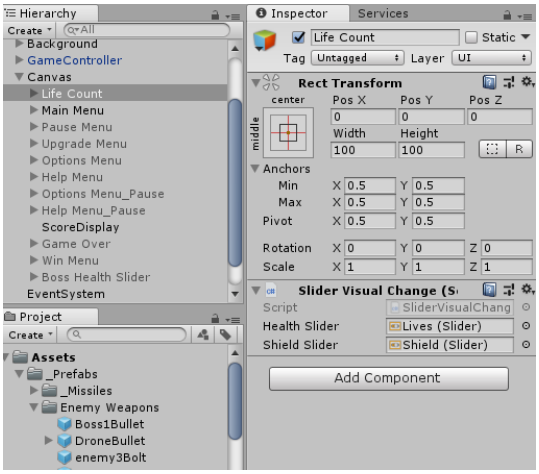
```
void Start () {
    rb = GetComponent<Rigidbody2D> ();
    t = GameObject.FindGameObjectWithTag ("Player").GetComponent<Transform> ();
    playerController = GameObject.FindGameObjectWithTag ("Player").GetComponent<PlayerController> ();

    float distanceX = t.position.x - rb.position.x;
    float distanceY = t.position.y - rb.position.y;
    if(Mathf.Abs(distanceX) > Mathf.Abs(distanceY)){
        float x = Mathf.Abs(distanceY) / Mathf.Abs(distanceX);
        float res = bulletSpeed/Mathf.Sqrt (Mathf.Pow(x,2) + 1);
        float velocityX = 0;
        if (distanceX >= 0) {
            velocityX = res;
        } else {
            velocityX = res*-1;
        }
        float velocityY = res * x * -1;
        rb.velocity = new Vector2 (velocityX, velocityY);
    }else...
}
void OnTriggerEnter2D(Collider2D other){
    if(other.CompareTag("Player")){
        Destroy (gameObject);
        playerController.ChangeHealth (-1);
        playerController.CallTintChange ();
        playerController.CallInvulnerable ();//try again if it is good
    }
}
```

Slika 62: Algoritma znotraj razreda izstrelka sovražnika

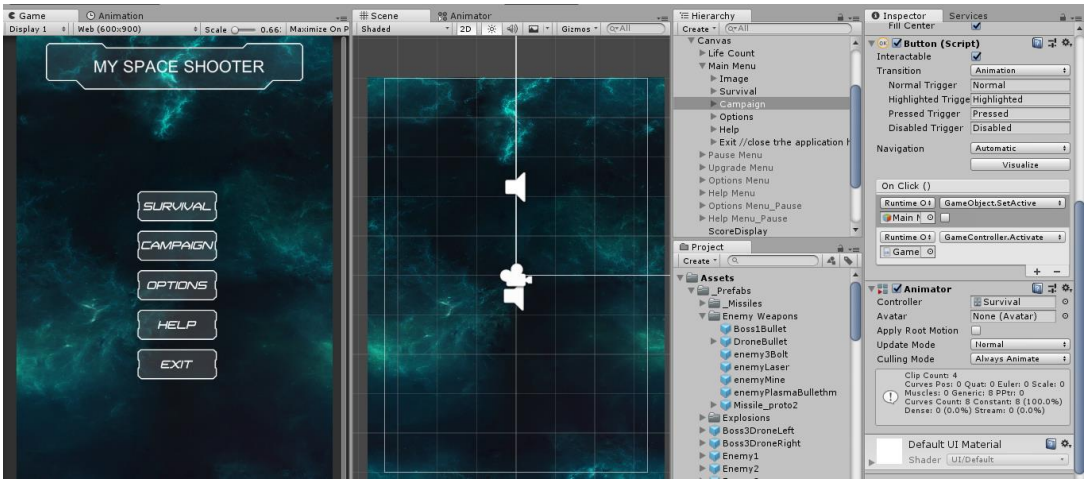
3.2.4 Uporabniški vmesnik

Uporabniški vmesnik je podobno kot pri igri v Javi sestavljen iz več menijev, vidnih na sliki 63: glavni meni, meni pavze, meni nadgradenj, meni opcij, meni za pomoč ter dva končna menija (igralec izgubi igro in igralec premaga igro v načinu zgodbe). Vsi ti objekti so v hierarhiji »otroci« platna (*Canvas*). Tako igralni pogon ve, kaj je del vmesnika in kaj je del igre. »Komponenta *Canvas* predstavlja abstraktni prostor, v katerem je uporabniški vmesnik postavljen in upodobljen. Vsi elementi uporabniškega vmesnika morajo biti otroci predmeta *GameObject*, ki ima priloženo komponento *Canvas*.« (20) Za prikaz menijev pa skrbi že opisan razred – *GameManager* na sliki 54.



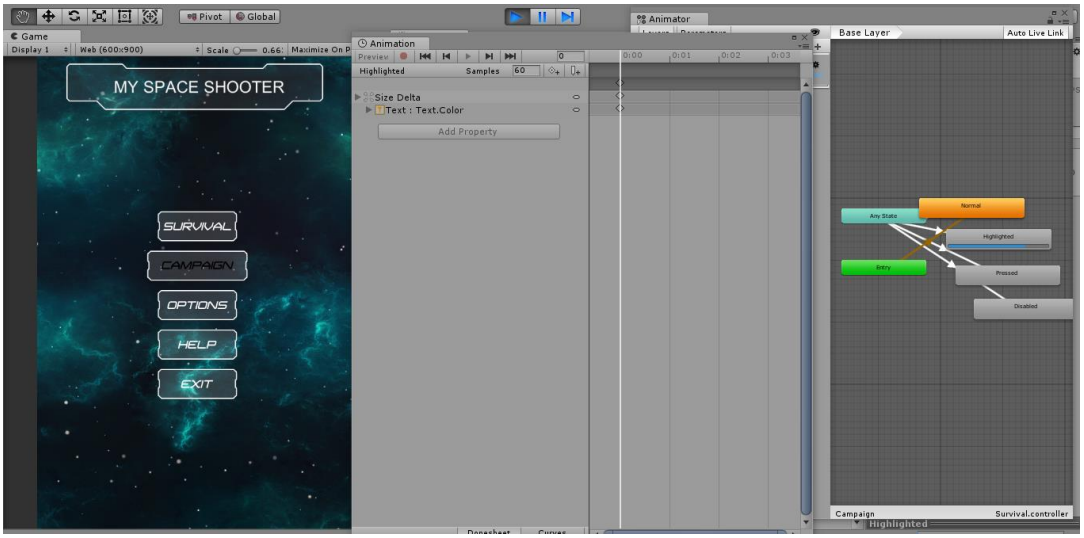
Slika 63: Posnetek zaslona prikazuje menije v urejevalniku Unity

Posnetek zaslona na sliki 64 predstavlja sestavo glavnega menija. Objekt vsebuje komponente tipa *Button* (gumbi) in komponente tipa *Text* (izpis besedila za posamezen gumb). Gumbom lahko funkcionalnost dodamo kar že znotraj inšpektorja in sicer desno na sliki 66 vidimo, da ima dogodek *On Click ()*, ki se izvrši natanko takrat, ko igralec klikne na ta gumb. Le-ta pa nato vzbudi tiste funkcije, katere smo določili znotraj okenca pod dogodkom *On Click ()*. Na posnetku zaslona je trenutno izbran objekt *Campaign*, ta pa ima tudi komponento tipa *Animator* (lahko ga animiramo). Znotraj te komponente mu lahko dodamo *AnimatorController*, ki vsebuje vse animacije.



Slika 64: Sestava glavnega menija

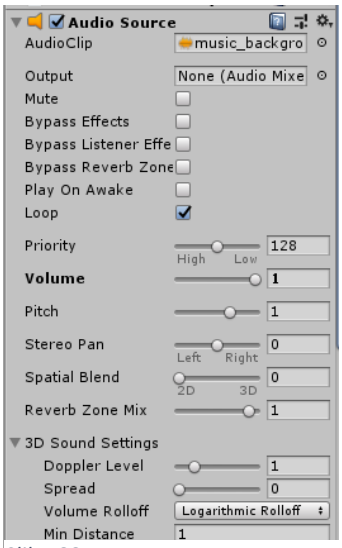
Slika 65 prikazuje pogled igre, okno animacije in okno animatorja, ko je urejevalnik v igralnem načinu. Znotraj okna *Animator* je odprt *AnimatorController*, ki skrbi to katera animacija se trenutno predvaja. Na spodnji sliki je primer, ko se predvaja animacija *Highlighted*, kar pomeni, da ima igralec miško na gumbu *Campaign* (levo na sliki 65). V oknu *Animation* pa je odprta animacija *Highlighted*, ki gumbu začasno spremeni velikost in barvo napisa (dokler ima igralec miško na gumbu).



Slika 65: Prikaz pogleda igre, okna animatorja, okna animacije v igralnem načinu urejevalnika Unity

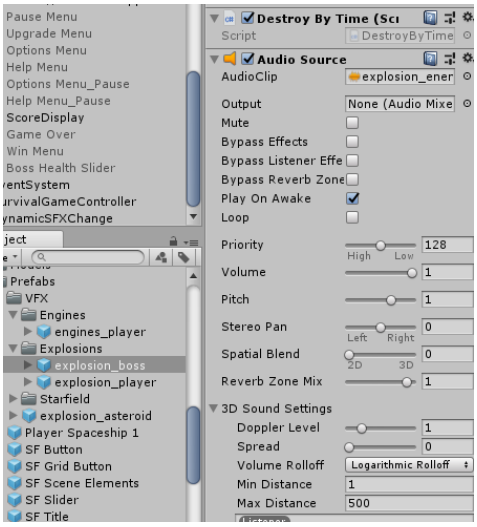
3.2.5 Zvočni in vizualni efekti

Dodajanje zvoka znotraj igre s pomočjo urejevalnika Unity je nadvse preprosto, saj objektu dodamo komponento Audio Source, vidno na sliki 66, nato pa izberemo še zvočni posnetek, katerega bi radi predvajali. V inšpektorju je možno nastaviti tudi ali se zvok začne predvajati tako, ko objekt ustvarimo v sceni, ali se ponavlja, lahko ga utišamo in pa spreminjamo različne lastnosti kot so višina tona, okolica, kjer je ta zvok slišen in njegova prioriteta (če je v igri več zvokov, se tistega, ki ima najvišjo prioriteto sliši bolje) in še veliko več nastavitev za 3D zvok (jih ne uporabljam ker je to 2D igra).



Slika 66: Komponenta AudioSource in njene nastavitve

Nekatere zvočne efekte predvajam znotraj skrip, druge pa predvajam takoj, ko se objekt ustvari. Pri ustvarjanju v že shranjenih objektov (primer na sliki 67 – explosion_boss), v *Audio Source* komponenti le nastavimo, da se zvočni efekt predvaja takoj, ko je objekt narejen. Za predvajanje, nadaljevanje in začasno ustavitev glasbe v ozadju pa uporabim funkcije *Play*, *Pause* in *UnPause* na objektu tipa *AudioSource*. Uporaba funkcij je vidna na sliki 68, znotraj razreda *GameManager*, ki je bil opisan v poglavju 3.2.3.1.

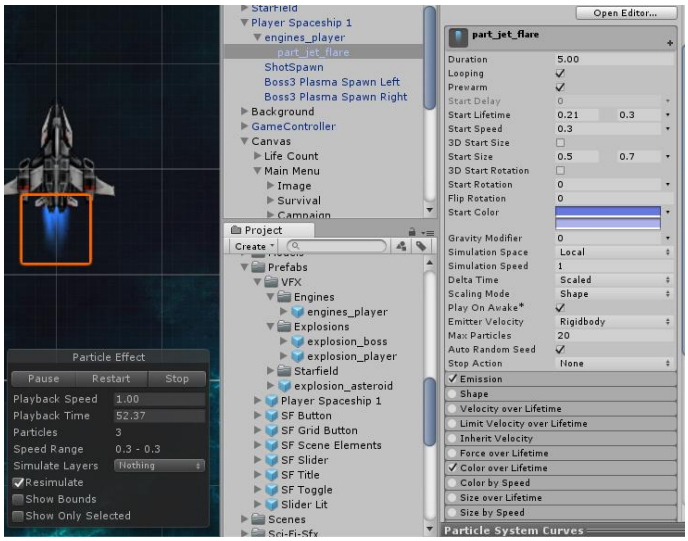


Slika 67: Audio Source komponenta za vnaprej definiran objekt eksplozije

```
if ((Input.GetKeyDown(KeyCode.P) || In
    pauseMenu.SetActive (true);
    Time.timeScale = 0;
    pause = true;
    audios[0].Pause ();
    audios[1].UnPause ();
} else if((Input.GetKeyDown (KeyCode.P
    pauseMenu.SetActive (false);
    Time.timeScale = 1;
    pause = false;
    audios[1].Pause ();
    audios[0].UnPause ();//the !pause
}else if(Input.GetKeyDown (KeyCode.U)
    upgradeMenu.SetActive (true);
    Time.timeScale = 0;
    upgrade = true;
    audios[0].Pause ();
    audios[1].UnPause ();
}else if((Input.GetKeyDown (KeyCode.U)
    upgradeMenu.SetActive (false);
    Time.timeScale = 1;
    upgrade = false;
    audios[1].Pause ();
    audios[0].UnPause ();
}
```

Slika 68: Del algoritma za predvajanje zvoka

Prav tako kot uporaba zvočnih je bila tudi uporaba vizualnih efektov nadvse enostavna. Na objektu, kjer hočemo imeti vizualni efekt preprosto le dodamo sistem delcev (*Particle System*). Posnetek zaslona 69 prikazuje sistem delcev za igralčevo figuro (simuliranje reaktivnega motorja). Sistem deluje tako, da izberemo sliko, ki bo služila kot delec, nato pa skozi različne nastavitve ustvarimo vizualni efekt. Nekatere nastavitve so začetna velikost, začetna hitrost, čas trajanja, omogočanje ponavljanja, življenjski čas delca (da ne preobremenimo računalnika z delci), omogočanje da simulacijo začnemo takoj ko je objekt, ki vsebuje sistem delcev, ustvarjen, omogočanje spreminjanje rotacije, velikosti, barve, oblike, moči sevanja skozi čas itd.



Slika 69: Prikaz komponente Particle System in njenih lastnosti

4 Zaključek

Ustvarjanje videoigre še zdaleč ni enostavno delo, saj se marsikateri ustavi že v prvem koraku: pridobitev ideje, in kasneje tudi v samem izdelovanju igre. Izdelovanje kompleksne igre, kjer se programer lahko zanese le nase, je proces, pri katerem je potrebno vložiti veliko ur in še več dela. Še posebej, če ustvarja igro brez kakršnih koli orodij, ki so namenjena olajšavi dela. To sem izkusil tudi sam, saj sem po izdelanem duplikatu igre v igralnem pogonu Unity spoznal, da sem ob tem prihranil veliko ur »trpinčenja«, ki sem jih prebil pri izdelavi igre v višjem programskem jeziku Java, pri tem pa sem lahko dodajal še veliko drugih elementov, ki so igro popestrile.

Osebno menim, da sem cilj seminarske naloge gladko dosegel, saj sem predstavil celotno pripravo na izdelavo igre in tudi opisal njeno izdelavo in delovanje v igralnem pogonu Unity kot tudi v programskem jeziku Java. Najpomembnejše pa se mi zdi to, da sem predstavil kako nam kakršnokoli orodje za pomoč pri kreiranju videoigre znatno zmanjša število ur, ki jih bodisi posameznik ali podjetja potrebujejo za izdelovanje kvalitetne igre.

5 Zahvala

Za pomoč pri izdelavi seminarske naloge se zahvaljujem svojem mentorju, Albertu Zorku, podjetju Trac d.o.o in vsem zaposlenim, kjer sem med poletnimi počitnicami 2018 opravljal počitniško delu ter tudi svojim staršem in vsem drugim, ki so mi pri izdelavi seminarske naloge na kakršen koli način pripomogli.

6 Stvarno kazalo

7 Viri in literatura

1. Krhun, Andrej. Programski jezik java. *Wikipedija.* [Elektronski] 1. februar 2019. [Navedeno: 18. marec 2019.]

2. Ramoliya, Hitesh. Popular programming languages. *Amar InfoTech.* [Elektronski] 12. december 2018. [Navedeno: 18. marec 2019.] https://www.amarinfotech.com/top-6-popular-programming-languages-in-2019.html.

3. Integrirana razvojna okolja (IDE). *E gradiva za računalništvo in informatiko.* [Elektronski] 24. maj 2006. [Navedeno: 18. marec 2019.] http://colos.fri.uni-lj.si/ERI/RACUNALNISTVO/PROG_JEZIKI_ORODJA/prog_ide.html.

4. Eclipse (Software). *Wikipedia.* [Elektronski] 17. februar 2019. [Navedeno: 18. marec 2019.] https://en.wikipedia.org/wiki/Eclipse_(software).

5. Java AWT . *javaTpoint.* [Elektronski] 10. september 2012. [Navedeno: 19. marec 2019.] https://www.javatpoint.com/java-awt.

6. Java Swing. *javaTpoint.* [Elektronski] 25. september 2012. [Navedeno: 19. marec 2019.] https://www.javatpoint.com/java-swing.

7. Godbold, Ashley in Jackson, Simon. *Mastering Unity 2D Game Development.* s.l. : Packt Publishing, 2016. ISBN-10 : 1786463458.

8. Unity Technologies. *Wikipedia.* [Elektronski] 6. marec 2019. [Navedeno: 20. marec 2019.] https://en.wikipedia.org/wiki/Unity_Technologies.

9. Unity technologies delivers Unity 3. *Marketwire.* [Elektronski] 27. september 2010. [Navedeno: 20. marec 2019.] http://www.marketwired.com/press-release/Unity-Technologies-Delivers-Unity-3-1325564.htm.

10. Matney, Lucas. Unity raises $181M monster round at a reported $1.5B valuation. *TechCrunch.* [Elektronski] 13. julij 2016. [Navedeno: 20. marec 2019.] https://techcrunch.com/2016/07/13/unity-announces-181-million-monster-round-led-by-dfj-growth/?guccounter=1&guce_referrer_us=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnLw&guce_referrer_cs=r9rcX1keUax7eagIXIK7tQ.

11. Unity (game engine). *Wikipedia.* [Elektronski] 3. marec 2019. [Navedeno: 20. marec 2019.] https://en.wikipedia.org/wiki/Unity_(game_engine).

12. C sharp (programming language). *Wikipedia.* [Elektronski] 23. februar 2019. [Navedeno: 20. marec 2019.] https://en.wikipedia.org/wiki/C_Sharp_(programming_language).

13. Guthrie, Scott. ScottGu's Blog. *Nice VS 2008 Code Editing Improvements.* [Elektronski] 28. julij 2007. [Navedeno: 20. marec 2019.] https://weblogs.asp.net/scottgu/nice-vs-2008-code-editing-improvements.

14. Microsoft Visual Studio. *Wikipedia.* [Elektronski] 7. marec 2019. [Navedeno: 20. marec 2019.] https://en.wikipedia.org/wiki/Microsoft_Visual_Studio.

15. Facchetti, Clara. How to edit photos online. *Tech Advisor.* [Elektronski] 10. oktober 2018. [Navedeno: 2019. marec 2019.] https://www.techadvisor.co.uk/how-to/internet/edit-photos-online-3663773/.

16. Java MouseListener Interface. *javaTpoint.* [Elektronski] [Navedeno: 21. marec 2019.] https://www.javatpoint.com/java-mouselistener.

17. A Close Look at the Paint Mechanism. *Oracle.* [Elektronski] [Navedeno: 22. marec 2019.] https://docs.oracle.com/javase/tutorial/uiswing/painting/closer.html.

18. Class Timer. *Oracle.* [Elektronski] [Navedeno: 22. marec 2019.] https://docs.oracle.com/javase/7/docs/api/javax/swing/Timer.html.

19. Coroutines. *Unity Documentation.* [Elektronski] [Navedeno: 22. marec 2019.] https://docs.unity3d.com/Manual/Coroutines.html.

20. Canvas. *Unity Documentation.* [Elektronski] [Navedeno: 22. marec 2019.] https://docs.unity3d.com/Manual/class-Canvas.html.

21. Java Introduction. *w3schools.* [Elektronski] 8. april 2015. [Navedeno: 18. marec 2019.] https://www.w3schools.com/java/java_intro.asp.

8 Priloge

Zraven seminarske naloge prilagam tudi zgoščenko, na kateri so:

- **Dokument .docx**: Vesoljski strelec
- **Stisnjene datoteke:** Slike in koda napisana v Javi ter projekt, ustvarjen v igralnem pogonu Unity