UNIVERZA V MARIBORU
FAKULTETA ZA ELEKTROTEHNIKO,
RAČUNALNIŠTVO IN INFORMATIKO

Žiga Jelen

RAZVOJ MOBILNIH IGER S POMOČJO KNJIŽNICE ANDENGINE

Diplomsko delo

Maribor, avgust 2014

RAZVOJ MOBILNIH IGER S POMOČJO KNJIŽNICE ANDENGINE

Diplomsko delo

Študent:	Žiga Jelen
Študijski program:	Visokošolski študijski program
	Računalništvo in informacijske tehnologije
Mentor:	Doc. dr. Matej Črepinšek
Lektorica:	Tjaša Fajhtinger, dipl. slovenistka in anglistka

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

Številka: E1041446
Datum in kraj: 28. 03. 2014, Maribor

Na osnovi 330. člena Statuta Univerze v Mariboru (Ur. l. RS, št. 46/2012)
izdajam

SKLEP O DIPLOMSKEM DELU

1. **Žigi Jelenu,** študentu visokošolskega strokovnega študijskega programa RAČUNALNIŠTVO IN INFORMACIJSKE TEHNOLOGIJE, se dovoljuje izdelati diplomsko delo pri predmetu Programski jeziki.

2. **MENTOR:** **doc. dr. Matej Črepinšek**

3. **Naslov diplomskega dela:**
   RAZVOJ MOBILNIH IGER S POMOČJO KNJIŽNICE ANDENGINE

4. **Naslov diplomskega dela v angleškem jeziku:**
   DEVELOPING MOBILE GAMES WITH ANDENGINE

5. Diplomsko delo je potrebno izdelati skladno z "Navodili za izdelavo diplomskega dela" in ga oddati v treh izvodih (dva trdo vezana izvoda in en v spiralo vezan izvod) ter en izvod elektronske verzije do 30. 09. 2014 v referatu za študentske zadeve.

Pravni pouk: Zoper ta sklep je možna pritožba na senat članice v roku 3 delovnih dni.

Dekan:
red. prof. dr. Borut Žalik

Obvestiti:
- kandidata,
- mentorja,
- odložiti v arhiv.

ZAHVALA

Zahvaljujem se staršem, ki so mi omogočili študij in me podprli pri vsakem koraku. Posebne zahvale gredo pokojni materi, ki mi je pokazala, da z borbenostjo lahko pridemo do uspeha.

Za pomoč pri izdelavi diplomskega dela se zahvaljujem doc. dr. Mateju Črepinšku.

Razvoj mobilnih iger s pomočjo knjižnice AndEngine

Ključne besede: Android, AndEngine, igralni pogon, knjižnica, igra.

UDK: 004.354.7:004.412(043.2)

Povzetek

V diplomskem delu je predstavljen razvoj igre s pomočjo knjižnice AndEngine. Knjižnice vsebujejo različne funkcionalnosti, ki jih prikličemo oz. uporabimo pri samem razvoju. Ni nam treba vedeti, kako točno funkcionalnost deluje, ampak moramo vedeti samo, kako funkcionalnost priklicati oz. uporabiti. Zato je uporaba knjižnic za razvijalce priporočljiva, saj s tem dosežemo večjo stabilnost in zanesljivost igre, prav tako pa nam olajšajo delo.

Preučili bomo knjižnico AndEngine, v kateri bomo spoznali življenjski cikel, ustvarjanje različnih aktivnosti, delo z entitetami, modifikatorje, kamere in dodatne razširitve za samo knjižnico. Predstavili in analizirali bomo konkurenčne knjižnice in naredili preprost prototip igre, ki bo namenjena otrokom za učenje ločevanja odpadkov.

Developing Mobile Games with AndEngine

Key words: Android, AndEngine, Game engine, Library, Game.

UDK: 004.354.7:004.412(043.2)

Abstract

This diploma paper presents game development using the AndEngine library. Libraries include different functions, which we launch and use during the development. We do not have to be familiar with the functions in particular, we simply have to know how to launch and use the function. Therefore, the use of libraries is recommended for developers as we achieve greater stability and reliability of the game, and it also facilitates our work.

We will study the AndEngine library, in which we will get to know the life cycle, activity creating, working with entities, modifiers, cameras and different library extensions. We will present and analyze competitive libraries, and develop a simple game prototype intended to teach children waste sorting.

KAZALO VSEBINE

KAZALO SLIK

KAZALO TABEL

SEZNAM UPORABLJENIH SIMBOLOV IN KRATIC

2D	Dvodimenzionalen prostor	
3D	Tridimenzionalen prostor	
3R	Reduce, Reuse, Recycle	(zmanjšaj, ponovno uporabi, recikliraj)
ADT	Android Development Tools	(*razvojno orodje Android*)
CPE	Centralno procesna enota	
GPE	Grafično procesna enota	
HUD	Heads-up display	(*prikazovalnik*)
IDE	Integrated development environment	(*integrirano razvojno okolje*)
SDK	Software development kit	(*programski razvojni komplet*)
SVG	Scalable Vector Graphics	(*umerljiva vektorska grafika*)
WYSIWYG	What You See Is What You Get	(*kar vidiš, boš tudi dobil*)

1 UVOD

Igre ljudem že od nekdaj predstavljajo zabavo in užitek. Popularnost iger na pametnih napravah iz leta v leto narašča, posledično se povečuje tudi industrija iger. Raziskava je pokazala, da v povprečju igre igra kar 48 % Evropejcev, od tega 26 % igralcev igra igre na mobilnih napravah. V raziskavi izvedeni leta 2012 je bilo uspešno rešenih 15.142 spletnih anket v šestnajstih evropskih državah [9]. Igre naj bi bile za igralca preproste in zanimive, saj le na tak način dosežejo velik uspeh. Pomembno je, kako razvijemo igro in kakšen pristop uporabimo pri razvoju le-te. Prednost knjižnic je, da vsebujejo funkcionalnosti, ki jih uporabimo pri različnih projektih. Pri uporabi nam tako ni potrebno vedeti točnega delovanja same funkcionalnosti, ampak zgolj kako se nanjo sklicati. Uporaba knjižnic tako poveča stabilnost in zanesljivost ter zmanjša čas razvoja. Na spletu najdemo različne knjižnice in orodja. Najbolj priljubljene knjižnice za razvoj mobilnih iger so Unity, AndEngine, LibGDX, Cocos2D … Vsaka knjižnica ima prednosti in slabosti, ki jih bomo na kratko proučili v poglavju 2.1.

Cilj dela je razviti preprosto in poučno igro s pomočjo knjižnice AndEngine. Igra bi bila namenjena zmanjšanju in ločevanju odpadkov. Igralec bi tako lahko znanje, ki ga je usvojil skozi igro, uporabil tudi v realnem življenju. Za sam razvoj takšne igre bomo morali spoznati knjižnico AndEngine. Trend mobilnih iger je, da igralec znanje, ki ga usvoji s pomočjo igre, prenese v realni svet. Z igro povečamo motivacijo za izvajanje želene aktivnosti [16].

V diplomskem delu bom predstavil knjižnico AndEngine. Opisal bom njeno strukturo, v katero spadajo življenjski cikel, ustvarjanje različnih aktivnosti, delo z entitetami, modifikatorji, kamere ... Poleg osnovne knjižnice imamo na razpolago še dodatne razširitve, ki nam olajšajo delo pri samem razvoju in razširitvi igre. Za lažjo odločitev pri izbiri knjižnice za razvoj igre bom predstavil prednosti in slabosti konkurenčnih knjižnic. V nadaljevanju bom implementiral preprosto igro in opisal njen potek.

2 ANDROID IN OPENGL ES

Za optimalno in hitro delovanje iger na mobilnih napravah se moramo spustiti na nizko raven programiranja. Pri razvoju uporabljamo centralno procesno enoto (CPE), za izračune in obdelavo podatkov, ter grafično procesno enoto (GPE), za hitro obdelovanje in spreminjaje slik v pomnilniku. Za normalno delovanje igre morata biti ti dve enoti usklajeni, saj drugače čakata druga na drugo, s tem pa se zmanjša zmogljivost igre. Za lažji dostop do grafične procesne enote je Android podprl knjižnico Open Graphics Library ES (OpenGL® ES). Grafična knjižnica je posebej prilagojena za manjše naprave, kot so žepne konzole, mobilne naprave, navigacijske naprave …

Tabela 2.1: Tabela prikazuje relativno število naprav, na katerih je podprta določena različica OpenGL ES [4].

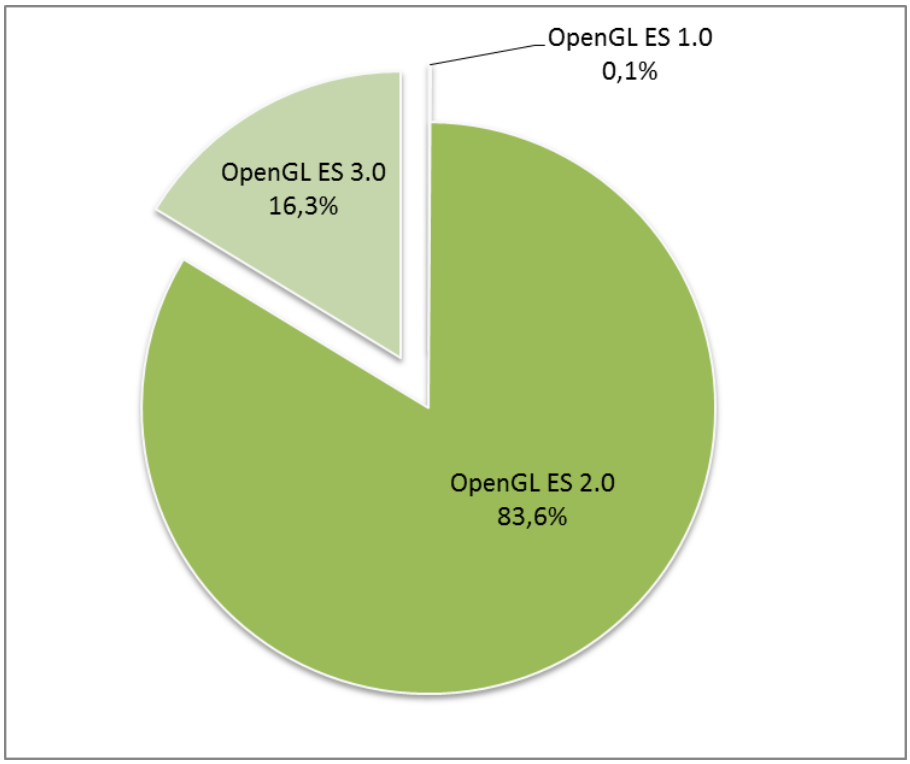
Različica OpenGL ES	Različica Android	Delež naprav
OpenGL ES 1.0	Android 1.0 in vse višje različice	0.1 %
OpenGL ES 2.0	Android 2.2 in vse višje različice	83.6 %
OpenGL ES 3.0	Android 4.3 in vse višje različice[1]	16.3 %

Prednost knjižnice je, da je nevtralna in dostopna vsakomur. V produkt jo lahko vključimo popolnoma brezplačno. Knjižnica predstavlja grafični standard za različne platforme. Uvrstimo jo v skupino OpenGL, v kateri so grafični elementi dobro strukturirani in načrtovani, kar razvijalcu omogoča preprosto uporabo funkcionalnosti same knjižnice. Prednost knjižnice je tudi ta, da je dobro dokumentirana [10]. Lastnik knjižnic OpenGL ES je neprofitna organizacija Khronos Group, ki je do zdaj izdala že tri različice. Vključujejo jo

[1] Sam proizvajalec naprave mora podpreti grafično knjižnico OpenGL ES 3.0, za katero ni nujno, da je podprta v vseh različicah, ki so višje od Androida 4.3.

vodilna podjetja, kot so Intel, Apple Inc., NVIDIA … Uporablja jo tudi podjetje Google Inc. v operacijskem sistemu Android, ki je že podprl vse tri različice. Delež podprtih knjižnic na napravah podaja tabela 2.1.

Največji delež Android naprav podpira knjižnica OpenGL ES 2.0, saj je že bila podprta pri različici 2.2 (Froyo), ki je bila glavna prelomnica za mobilne naprave (Slika 2.1).



Slika 2.1: Število naprav, ki podpirajo OpenGL različico [4].

2.1 Primerjava igralnih pogonov

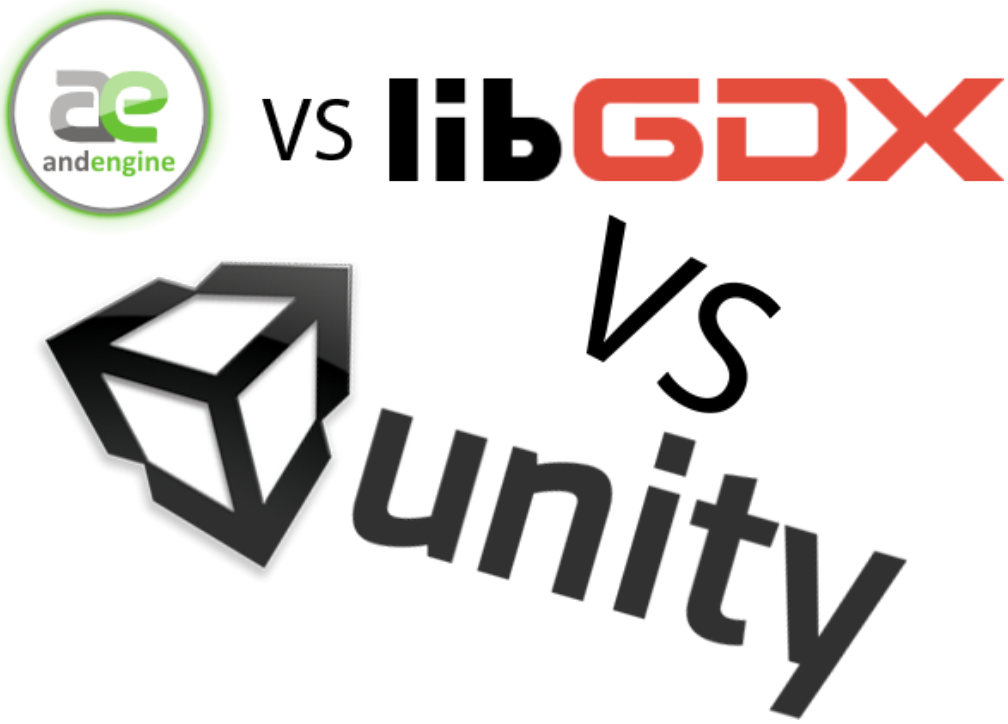
Na internetu lahko najdemo različne igralne pogone, zato bom v tem poglavju knjižnico AndEngine primerjal z igralnima pogonoma Unity in libGDX. Če primerjamo knjižnico in igralna pogona, lahko vidimo, da ima vsak določene prednosti in slabosti. Izbira je odvisna od same funkcionalnosti igre. Za primerjavo sem uporabil naslednje faktorje: ceno, podpiranje različnih prostorov in platform ter jezik razvoja igre.

Cena

Če primerjamo knjižnico in igralna pogona, lahko vidimo, da sta knjižnica AndEngine in igralni pogon libGDX brezplačna. Pogon Unity stane kar 1140 € oz. 57 € na mesec. Cena predstavlja pomemben faktor, saj si nekateri razvijalci ne morejo privoščiti dodatnih stroškov za sam razvoj.

Razvijanje igre v različnih prostorih

Za razvijanje iger poznamo dva prostora (2D in 3D). Razlika med knjižnico in igralnima pogonoma se pokaže pri podpori 3D prostora. Igralna pogona LibGDX in Unity imata, za razliko od knjižnice AndEngine, možnost za razvoj igre v 3D prostoru.



Slika 2.2: Logotipi knjižnice AndEngine (levo zgoraj), pogona libGDX (desno zgoraj) ter pogona Unity (spodaj) [1] [12] [17].

Podpiranje različnih platform

Knjižnica AndEngine je namenjena zgolj razvijanju na platformi Android. V tem se razlikuje od pogonov LibGDX in Unity, ki sta razvita z namenom, da z enako kodo podpreta več različnih platform. Pogon Unity podpira skoraj vse najbolj aktualne

računalniške, mobilne, spletne in konzolne platforme, za razliko od pogona LibGDX, ki podpira zgolj določene.

Razvojni jezik

Razvijanje v že poznanem programskem jeziku je velika prednost, saj s tem hitro usvojimo uporabo same knjižnice. Knjižnica AndEngine in pogon LibGDX uporabljata programski jezik Java. Enak jezik uporabljamo pri razvijanju Android aplikacij. Pri pogonu Unity pa za razvoj igre uporabljamo skripte, ki jih lahko pišemo v jezikih JavaScript, C# in Boo.

Tabela 2.2: Primerjava knjižnice in pogonov.

	AndEngine	**LibGDX**	**Unity**
Cena	Brezplačna	Brezplačna	1140 € ali 57 € na mesec
Podprte platforme	Android	Windows, Mac, Linux, Android, iOS, BlackBerry, HTML5	Windows, Mac, Linux, Android, iOS, BlackBerry, HTML5, Xbox ...
Podprti prostori	2D	2D in 3D	2D in 3D
Razvojni jezik	Java	Java	Skripte v jezikih JavaScript, C# ali Boo

Za razvoj igre sem potreboval knjižnico, ki je napisana v razvojnem jeziku Java, saj želim uporabiti že usvojeno znanje o samem jeziku. Odločil sem se, da uporabim knjižnico AndEngine. Igra, ki sem jo razvijal, je namenjena samo za platformo Android, zato nisem potreboval knjižnice oz. pogona, ki bi podpiral več različnih platform. Velika prednost knjižnice je ta, da razvijalec ne potrebuje kakršnega koli znanja o knjižnici OpenGL. Sama zgradba je tako trda, da se nam ni potrebno spustiti na nizko raven programiranja. Dodaten razlog za izbiro knjižnice AndEngine je tudi ta, da je enostavna za uporabo in razumevanje. Na spletu lahko najdemo literaturo, kjer so opisane različne funkcionalnosti,

ki jih knjižnica ponuja za razvoj igre. S pomočjo literature sem hitro usvojil znanje, ki sem ga potreboval za sam razvoj igre.

Za razliko od konkurence smo s knjižnico AndEngine omejeni pri podpiranju različnih platform. Ima pa veliko prednost pri hitrem razvoju 2D iger, saj je implementacija igre zelo preprosta. Za razvoj igre zadostuje že osnovno znanje programskega jezika Java in jezika Android. Zaradi podpiranja različnih platform ima pogon LibGDX veliko prednost pred knjižnico AndEngine. Njegova popularnost se iz meseca v mesec veča, kar pomeni večjo uporabo, večjo zanesljivost in boljšo dokumentacijo. Pogon Unity ima svoje programsko okolje, za razliko od knjižnice AndEngine in pogona LibGDX, ki uporabljata za razvoj programsko okolje Eclipse. Pri razvoju v programskem okolju že vidimo vse ustvarjene elemente, ki smo jih postavili v prostoru. Pogon Unity ima bogato trgovino z različnimi primeri, ki si jih lahko enostavno prenesemo na računalnik. Njihovo razvojno okolje in sam pogon Unity prevladata nad razvojnim okoljem Eclipse, knjižnico AndEngine in pogonom LibGDX.

Če razvijamo igro za platformo Android v 2D prostoru, je priporočljivo uporabiti knjižnico AndEngine. Namestitev je zelo preprosta, saj si prenesemo zgolj projekt, ki ga vključimo v igro. Težave se lahko pojavijo pri razvoju bolj zapletenih iger, saj smo pri določenih elementih omejeni. Zato je knjižnica AndEngine namenjena hitremu in enostavnemu razvoju preproste igre. Za razliko od pogona LibGDX, je uporaba bolj zapletena, saj potrebujemo poseben program za ustvarjanje razvojnega projekta. Tako se ustvari glavni projekt, ki povezuje manjše podprojekte, ki so namenjeni različnim platformam. Uporabnost pogona se vsako leto bistveno povečuje, saj z eno kodo ustvarimo igro na različnih platformah. Če nam pogon LibGDX ne zadostuje in želimo biti konkurenčni z največjimi proizvajalci mobilnih iger, priporočam pogon Unity. Razvite igre zasedajo prva mesta v samem svetovnem vrhu. Eden izmed uspešnejših razvijalcev mobilnih iger je podjetje Rovio Entertainment, ki za razvoj iger uporablja pogon Unity (Angry Birds Epic, Bad Piggies …) [25]. Primerjavo med knjižnicama in pogonom prikazuje tabela 2.2.

3 ANDENGINE

Razvijalci mobilnih iger se soočajo s problemom, kako razviti igro na najbolj učinkovit način. Nekateri razvijalci se odločijo razviti lastno knjižnico oz. igralni pogon (ang. *game framework*), ki pripomore k prikazovanju in izrisovanju elementov na zaslonu. Slabost takšnega početja je, da nam sam razvoj orodja vzame veliko časa, zato se večina razvijalcev odloči za knjižnice, ki so odprtokodne in so dostopne vsem. Ena izmed najbolj priljubljenih knjižnic za razvijanje 2D iger je knjižnica AndEngine, avtorja Nicolasa Gramlicha.

AndEngine je odprtokodna knjižnica za razvoj 2D iger na pametnih napravah Android. Knjižnica podpira OpenGL ES, ki je standardni programski vmesnik za grafično procesiranje na ravni strojne opreme [2]. Uporaba knjižnice nam prihrani veliko časa, saj ima že pripravljene funkcionalnosti, ki nam olajšajo delo pri samem razvoju igre.

3.1 Delovanje

Pomembno je, da razumemo življenjski cikel igre (od ustvarjanja do nalaganja in prikazovanja tekstur na zaslonu). V nadaljevanju bom opisal metode, ki se kličejo po zaporedju. Začetek našega cikla se začne s klicem metode *onCreate*.

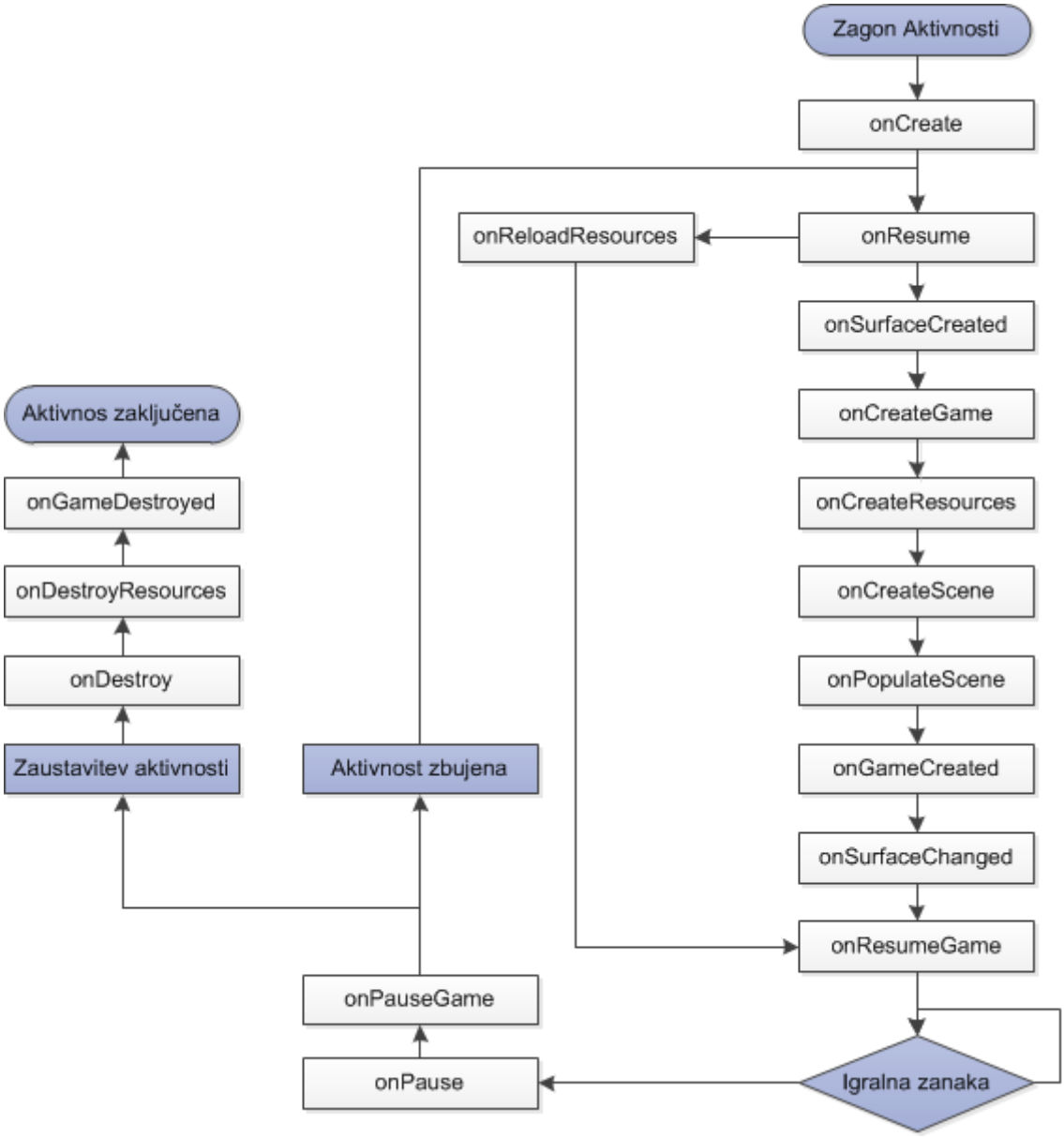
1. *onCreate*: V knjižnici se v tej metodi pokliče nova metoda *onCreateEngineOptions* razreda *BaseGameActivity*, kjer se nastavi pogled na zaslon (npr. velikost zaslona, budnost zaslona …) in ustvari površino (*RenderSurfaceView*).
2. *onResume*: Klic metode se izvede takrat, ko je aktivnost v ospredju in uporabnik dela z njo. Pokliče se tudi *onResume* metoda objekta *RenderSurfaceView*, ki predstavlja površino za izrisovanje teksture.
3. *onSurfaceCreated*: Preverimo, ali je treba znova naložiti sredstva. Če so že naložena, kličemo metodo *onReloadResources* in skočimo na *onResumeGame*.

4. *onReloadResources*: Ponovno naloži sredstva. To se zgodi v primerih, kadar aktivnost prebudimo in kadar je treba ponovno naložiti sredstva.
5. *onCreateGame*: Metoda skrbi za pravilni vrstni red klicev metod *onCreateResources*, *onCreateScene*, *onPopulateScene* in *onGameCreated*.
6. *onCreateResources*: Ustvari in prebere vsa potrebna sredstva, ki so uporabljena v igri. To so teksture, zvok, glasba in pisava.
7. *onCreateScene*: Ustvari in nastavi sceno ter pripne objekte nanjo. Scena predstavlja skupek objektov, ki so prikazani na zaslonu.
8. *onPopulateScene*: Lahko se zgodi, da imamo veliko število objektov pripetih na sceno. Vsi objekti pa še niso naloženi, zato je namen te metode, da prikažemo nalagalno sceno. Je zadnja metoda, ki upravlja sceno pred prikazom na zaslonu.
9. *onGameCreated*: Metoda *onCreateScene* je zaključena in po potrebi ponovno naloži sredstva. To se zgodi takrat, kadar druga nit ponovno pokliče metodo *onSurfaceCreated*.
10. *onSurfaceChanged*: Vsakokrat, ko se zamenja položaj naprave, se pokliče metoda (npr. kadar obrnemo napravo iz pokončnega v ležeč položaj).
11. *onResumeGame*: Ko so vse metode uspešno izvedene in ko je vse potrebno naloženo na pomnilnik, se začne izvajati posodobitvena nit, ki igro oživi.

Igro lahko zapremo ali pa jo zmotimo z drugo aktivnostjo, takrat se kličejo naslednje metode:

1. *onPause* – Klic metode se izvede takrat, kadar je aktivnost še delno vidna. V več primerih nas ta metoda opozori, da zapuščamo aktivnost, ki kmalu ne bo več vidna. Prav tako metoda zaustavi upodobitveno nit (ang. rendering thread).
2. *onPauseGame* – Namen metode je zaustaviti posodobitveno nit, ki zaustavi vse animacije in teksture na zaslonu.
3. *onDestroy* – Metoda se izvede takrat, ko bo igra popolnoma izbrisana iz pomnilnika. Tukaj se iz pomnilnika pobrišejo vsa sredstva, ki so bila prikazana na zaslonu.
4. *onDestroyResources* – Vsa sredstva so se že pobrisala v prejšnji metodi, zato se tukaj sprostijo vsi zvočni učinki in glasba.
5. *onGameDestroyed* – V tej metodi se konča naš življenjski cikel igre. Metoda nastavi zastavico *mGameCreated* na *false, kar* pomeni, da je igra dokončno izbrisana iz pomnilnika in jo je treba ponovno ustvariti.

Slika 3.1 prikazuje življenjski cikel igre. Kadar aktivnost ustvarimo, jo delno skrijemo ali izbrišemo iz pomnilnika.



Slika 3.1: Življenjski cikel igre.

Pogon (ang. Engine) je jedro knjižnice. Ustvari in zažene posodobitveno nit in skrbi za pravilno izvajanje funkcionalnosti. Za osnovno delovanje mora razvijalec določiti oz. nastaviti pogon. Ustvariti mora kamero, določiti politiko ločljivosti in usmerjenost zaslona.

Poleg osnovnih nastavitev lahko nastavimo še budnost zaslona, podamo zahtevo za uporabo zvoka … Politika ločljivosti določa, kako se bo površina za izrisovanje tekstur prilagajala zaslonu. Lahko jo prilagodimo površini celotnega zaslona ali omejimo na določeno velikost oz. razmerje. Zaslon je lahko usmerjen pokončno ali ležeče. Z določitvijo usmerjenosti zaslona igralcu točno določimo, v kateri smeri se mu naj prikazuje igralna površina.

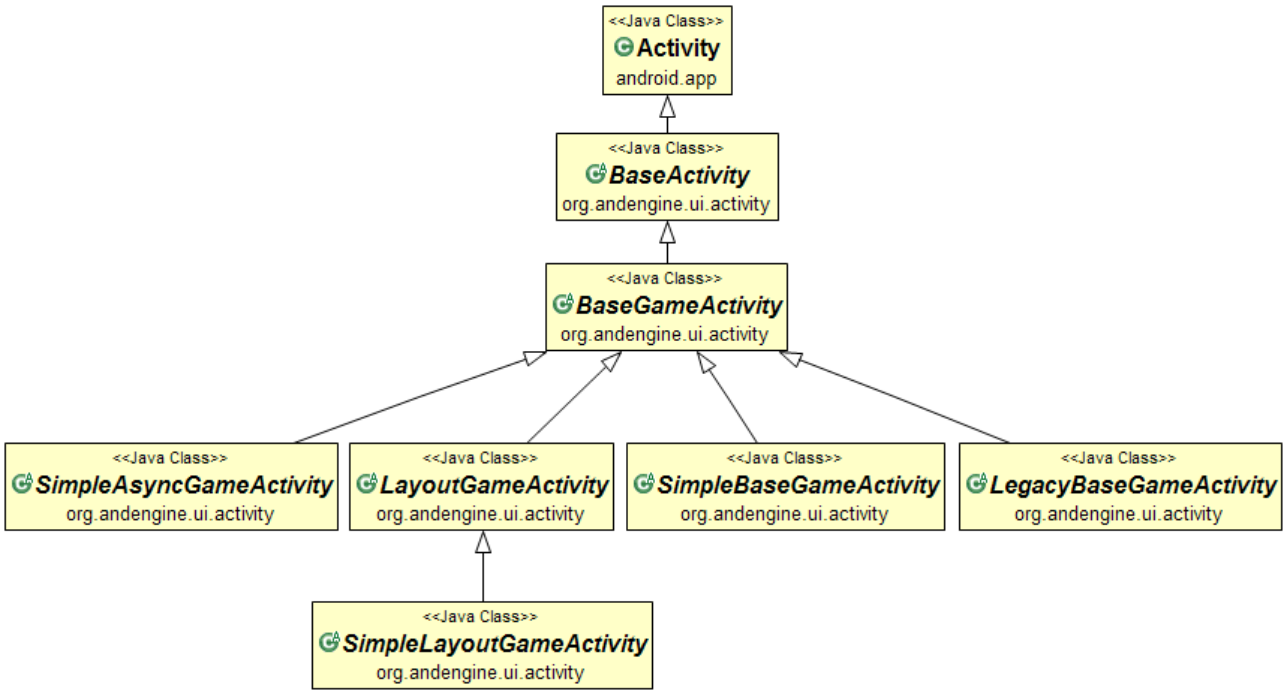
Tekstura v knjižnici predstavlja opis, kako se naj sličica naloži in prikaže na površino, ki je namenjena za izrisovanje. Pri ustvarjanju tekstur mora razvijalec določiti območje, v katerem pove, kateri del teksture se naj prikaže na zaslon. Tako lahko določimo eni teksturi več območij. To pomeni, da na grafični pomnilnik naložimo eno teksturo z različnimi sličicami, ki jih ločimo z območji.

3.2 Aktivnosti

Knjižnica nam ponuja že pripravljene abstraktne razrede, ki pripomorejo k ustvarjanju aktivnosti. Osnova za ustvarjanje aktivnosti v knjižnici AndEngine je razred *BaseGameActivity.* V knjižnici najdemo tudi druge razrede, ki ustvarijo aktivnost. Poznamo razred *LayoutGameActivity*, ki ga uporabljamo v igrah, v katerih želimo prikazovati več površin za izrisovanje. *SimpleAsyncGameActivity* se uporablja za poročanje napredka, in sicer ko želimo igralcu sporočiti, koliko elementov se je že naložilo in je pripravljenih za prikaz. Tako ima razvijalec na izbiro uporabo že pripravljenih abstraktnih razredov ali pa lahko ustvari svoj razred. Abstraktni razredi so:

- *LayoutGameActivity* – Razred, ki omogoča uporabo vizualne strukture XML android. Prednost tega razreda je, da lahko poleg površine za izris igre dodamo še druge poglede, ki jih ponuja sam android (gumbe, kazalce napredka …).
- *SimpleAsyncGameActivity* – Razred omogoča poročanje napredka. Ima tri dodatne metode, in sicer *onCreateResourcesAsync*, *onCreateSceneAsync* in *onPopulateSceneAsync*. Metode vsebujejo parameter, na katerega se sklicujemo in s pomočjo katerega poročamo o napredku nalaganja igre (*onProgressChanged*).
- *SimpleBaseGameActivity* in *SimpleLayoutGameActivity* – Sta poenostavljena razreda, ki izključujeta metodo *onPopulateScene*. Glavni namen je hitra implementacija in enostavna uporaba.

Iz slike 3.2 lahko razberemo, da so vsi razredi izpeljani iz razreda *BaseGameActivity*. Kar pomeni, da imajo vsi razredi enak življenjski cikel, ki je opisan v poglavju 3.1. Namen teh razredov je, da razvijalcu olajšajo delo, saj je njihova uporaba zelo enostavna.



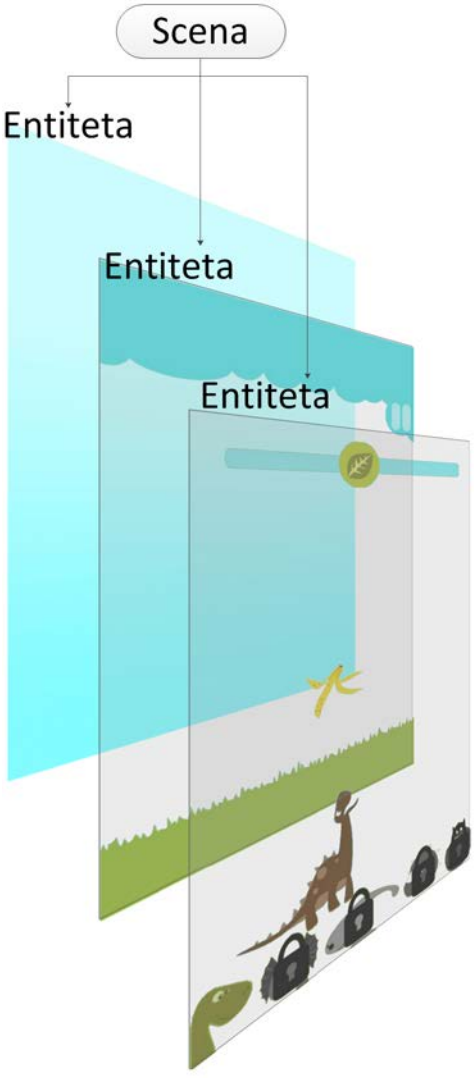
Slika 3.2: Razredna hierarhija aktivnosti.

3.3 Delo z entitetami

Entiteto si lahko zamislimo kot najbolj osnoven objekt, ki je postavljen v koordinatni sistem [15]. Vsi prikazani elementi, kot so sličice, teksti, kvadrati, animacije itd., predstavljajo entiteto. Tudi scene in plasti predstavljajo entitete, saj so izpeljane iz razreda *Entity*. Razred *Entity* oz. entiteta ravna zgolj z osnovnimi podatki, kot so položaj v koordinatnem sistemu, barva entitete, dodajanje in brisanje entitete s scene. V knjižnici AndEngine so vsi prikazani objekti pripeti na sceno in izpeljani iz entitete.

Entiteta se lahko predstavi kot objekt (sličice, kvadrat ...) ali kot plast (ang. layer). Razlika med plastjo in objektom se pokaže pri otrocih. To pomeni, da objekt nima otrok, za razliko od plasti, ki lahko ima neomejeno otrok.

Entitete se prikažejo na sceno v takšnem zaporedju, kot smo jih pripenjali nanjo. Pri razvoju igre uporabljamo različne plasti, ki so pripete na sceno (Slika 3.3). Na vsako plast pa lahko pripnemo več objektov ali plasti (Slika 3.4).



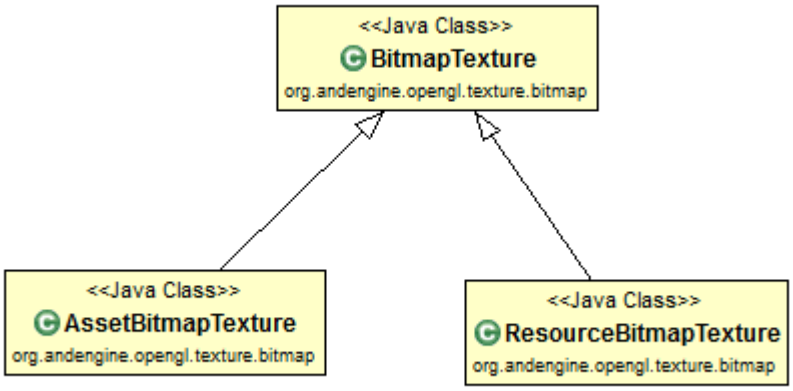
Slika 3.3: Plasti entitet v igri Junor 3R.

```
@Override
public void onCreateScene(OnCreateSceneCallback pOnCreateSceneCallback)
        throws IOException {

    Scene mScene = new Scene();

    Entity plast = new Entity();
    Entity objektA = new Entity();
    Entity objektB = new Entity();

    plast.attachChild(objektA);
    plast.attachChild(objektB);
    mScene.attachChild(plast);

    pOnCreateSceneCallback.onCreateSceneFinished(mScene);
}
```

Slika 3.4: Ustvarjanje scene in pripenjanje entitet na sceno.

3.4 Prikazovanje sličic in animacij

Za prikaz sličice ali animacije na zaslonu je potrebno najprej ustvariti teksturo, jo naložiti v pomnilnik in postaviti na sceno. Za ustvarjanje in nalaganje tekstur v pomnilnik imamo že pripravljen razred *BitmapTexture*, ki naloži teksturo v pomnilnik iz vhodnega toka podatkov. Ostali podrazredi se razlikujejo zgolj po vrsti vhodnega toka podatkov (Slika 3.5).



Slika 3.5: Hierarhija različnih razredov za odpiranje vhodnega toka.

Ko je tekstura naložena, ji je treba določiti območje, ki ga želimo prikazovati. Za določitev območja nam je razvijalec že pripravil statične metode, ki iz teksture izvlečejo območje. Metode najdemo v razredu *TextureRegionFactory*. Postopek nalaganja in določanja območja teksture implementiramo v metodi oz. ciklu *onCreateResources*. Tekstura se prikaže na zaslonu takrat, ko ji ustvarimo entiteto, določimo položaj in jo pripnemo na sceno. Pripenjanje na sceno izvedemo v ciklu *onCreateScene* ali pa v ciklu *onPopulateScene* (Slika 3.6).

```
private TextureRegion mObmocjeTeksture;

@Override
public void onCreateResources(
        OnCreateResourcesCallback pOnCreateResourcesCallback)
        throws IOException {

    Texture tekstura = new ResourceBitmapTexture(getTextureManager(), getResources(), R.drawable.image);
    mObmocjeTeksture = TextureRegionFactory.extractFromTexture(tekstura);
    tekstura.load();

    pOnCreateResourcesCallback.onCreateResourcesFinished();
}

@Override
public void onCreateScene(OnCreateSceneCallback pOnCreateSceneCallback)
        throws IOException {

    Scene scena = new Scene();

    Sprite slicica = new Sprite(15, 100, mObmocjeTeksture, getVertexBufferObjectManager());
    scena.attachChild(slicica);

    pOnCreateSceneCallback.onCreateSceneFinished(scena);
}
```

Slika 3.6: Ustvarjanje, nalaganje in pripenjanje sličice na zaslon.

Za prikazovanje animacij potrebujemo teksturo, ki združuje manjše sličice, ki tvorijo animacijo. Pri določanju območja je treba paziti, da teksturi določimo več manjših območij (Slika 3.7). Za postavitev in prikaz animacije na zaslonu uporabimo že pripravljeno animacijsko entiteto (ang. *AnimatedSprite*). Vsaki sličici v animaciji je treba določiti čas izvajanja. Entiteto pripnemo na sceno in jo zaženemo z metodo *animate*. Avtor knjižnice je razvijalcem pripravil vmesnik *IAnimationListener*, s katerim lahko poslušamo spremembe animacije ter spremljamo kdaj se je animacija začela, spremenila in zaključila (Slika 3.8). S tem ima razvijalec nadzor nad vsako sličico, ki se prikaže na zaslonu.

```
private TiledTextureRegion mObmocjeTeksture;

@Override
public void onCreateResources(
        OnCreateResourcesCallback pOnCreateResourcesCallback)
        throws IOException {

    Texture tekstura = new ResourceBitmapTexture(getTextureManager(),
            getResources(), R.drawable.image);
    mObmocjeTeksture = TextureRegionFactory.extractTiledFromTexture(tekstura, 2, 3);
    tekstura.load();

    pOnCreateResourcesCallback.onCreateResourcesFinished();
}
```

Slika 3.7: Ustvarjanje in nalaganje teksture z različnimi manjšimi sličicami.

```
@Override
public void onCreateScene(OnCreateSceneCallback pOnCreateSceneCallback)
        throws IOException {

    Scene scena = new Scene();

    AnimatedSprite animacija = new AnimatedSprite(15, 100, mObmocjeTeksture,
            getVertexBufferObjectManager());

    scena.attachChild(animacija);

    animacija.animate(100, new IAnimationListener() {

        @Override
        public void onAnimationStarted(AnimatedSprite pAnimatedSprite,
                int pInitialLoopCount) {
            Log.i("PoslusanjeAnimacije", "Začetek animacije");
        }

        @Override
        public void onAnimationLoopFinished(AnimatedSprite pAnimatedSprite,
                int pRemainingLoopCount, int pInitialLoopCount) {
            Log.i("PoslusanjeAnimacije", "Zaključek zanke animacije");
        }

        @Override
        public void onAnimationFrameChanged(AnimatedSprite pAnimatedSprite,
                int pOldFrameIndex, int pNewFrameIndex) {
            Log.i("PoslusanjeAnimacije", "Sprememba animacije");
        }

        @Override
        public void onAnimationFinished(AnimatedSprite pAnimatedSprite) {
            Log.i("PoslusanjeAnimacije", "Zaključek animacije");
        }
    });

    pOnCreateSceneCallback.onCreateSceneFinished(scena);
}
```

Slika 3.8: Ustvarjanje animacije in poslušanje spremembe.

3.5 Posodabljanje entitet

Vsaka entiteta vsebuje metodo *onManagedUpdate*, ki jo lahko prekrijemo. Klic metode izvaja posodobitvena nit. V njej lahko animiramo entiteto, preverjamo trke, izvedemo časovno-intervalne dogodke … Metoda nam prihrani veliko časa, saj nam ni treba ustvariti dodatnega časovnega krmilnika za vsako posamezno entiteto. Vhodni parameter metode je razlika med časom prejšnjega in trenutnega klica metode. Jedro pogona, ki zažene posodobitveno nit, poskrbi, da si ob zagonu shrani trenutni čas. Preden pogon pokliče metodo za posodobitev scene, si ponovno pridobi trenutni čas in ga odšteje shranjenemu (Slika 3.9).

```
private long getNanosecondsElapsed() {
    final long now = System.nanoTime();

    return now - this.mLastTick;
}
```

Slika 3.9: Pridobivanje in izračun preteklega časa.

Časovno razliko poda kot vhodni parameter metode. Scena je vrsta entitete, ki poskrbi, da se vsem otrokom pokliče posodobitvena metoda. Ko se zaključijo posodobitve scene, se ob koncu prišteje razlika k shranjenim časom. Tako lahko ob naslednjem ciklu ponovno izračunamo pretečeno razliko med časoma.

Iz opisa ugotovimo, da lahko sceni prekrijemo metodo *onManagedUpdate*, ki ji z drugo besedo rečemo tudi posodobitvena metoda. S prekrivanjem lahko začasno zaustavimo ali pa upočasnimo vse entitete (Slika 3.10). To nam lahko pride prav takrat, kadar želimo začasno zaustaviti ali upočasniti igro.

```
public class MojaScena extends Scene {
    float pretecenCas = 0;

    @Override
    protected void onManagedUpdate(float pSecondsElapsed) {
        pretecenCas += pSecondsElapsed;
        if (pretecenCas >= 5) {
            super.onManagedUpdate(pretecenCas);
            pretecenCas -= 5;
        }
    }

}
```

Slika 3.10: Prekrivanje metode *onManagedUpdate*.

3.6 Modifikatorji

Modifikatorji so pripravljeni razredi, ki spremenijo lastnosti entitete v določenem časovnem obdobju. Z njimi lahko spreminjamo barvo, vrednost alfe, premikamo entitete po zaslonu, pomikamo entitete po določeni poti itd. Lahko rečemo, da modifikatorji oživijo entiteto, kar daje videz animacije. S poslušanjem modifikatorja razvijalcu omogočimo popoln nadzor, saj s tem točno vemo, kdaj se njegovo izvajanje začne ter zaključi in kdaj se sprememba zgodi. V knjižnici imamo dve vrsti modifikatorjev, prva vrsta je namenjena spreminjanju lastnosti, druga pa za izvajanju modifikatorjev. Za spreminjanje lastnosti poznamo:

- *AlphaModifier* – Entiteti spremeni vrednost alfe (transparentnost). Konstruktor prejme tri parametre, in sicer čas trajanja modifikatorja, začetno in končno vrednost alfe.
- *ColorModifier* – Spremeni barvo entitete čez čas. Konstruktorju podamo čas trajanja, začetno in končno vrednost rdeče, zelene in modre barve.
- *DelayModifier* – Uporablja se pri zaporednem izvajanju modifikatorjev, saj zakasni izvajanje med enim in drugim modifikatorjem. Podamo zgolj čas zakasnitve.
- *FadeInModifier* in *FadeOutModifier* – Modifikatorja sta podrazreda razreda *AlphaModifier* in entiteto prikažeta iz nevidnega v vidno ali obratno. To pomeni, da je začetna vrednost alfe 0 in končna 1 ali pa začetna 1 in končna 0. Konstruktorju podamo samo čas trajanja.
- *JumpModifier* – Simulira skok entitete. Konstruktorju podamo informacijo, od kod in do kod bo entiteta skočila, prav tako podamo še višino skoka. Entiteta linearno

skoči in pada. Razvijalec ima možnost, da v konstruktorju poda svojo enačbo ali pa uporabi že napisane.

- *MoveByModifier* – Premaknemo entiteto za določeno vrednost. V konstruktor podamo vrednost premika v smeri x in y osi. Podamo tudi čas izvajanja premika.
- *MoveXModifier* in *MoveYModifier* – Premaknemo entiteto iz ene točke v drugo v smeri x ali y osi. V primeru, da premik naredimo v x ali v y smeri, konstruktorju podamo začetni in končni položaj ter čas premika.
- *RotationAtModifier* – Entiteto zavrti v podani točki za določeno število stopinj. Razvijalec mora določiti točko rotacije, kjer se bo le-ta izvedla. Določiti mora, za koliko stopinj želi zavrteti entiteto in v kakšnem časovnem obdobju se naj izvede rotacija.
- *RotationByModifier* – Modifikator zavrti entiteto za število stopinj, ki jih poda razvijalec. Konstruktorju podamo število stopinj in trajanje rotacije.
- *ScaleAtModifier* – Entiteto raztegne ali skrči v določeni točki, ki jo poda razvijalec. Konstruktorju podamo časovno obdobje in faktor raztega oziroma krčenja.
- *SkewModifier* – Zamakne entiteto v določenem časovnem obdobju v smeri x in y. Smer zamika je lahko pozitivna ali negativna. V konstruktor podamo čas trajanja in smeri zamika.
- *PathModifier* – Modifikator premika entiteto po poti, ki jo določimo s točkami. Konstruktor prejme čas izvajanja in pot. Čas trajanja se šteje od prve do zadnje točke premika. Pot definiramo s pripravljenim razredom *Path*. V njem definiramo število točk, kjer želimo voditi entiteto. Z metodo *to* povemo, kje se nahaja točka v koordinatnem sistemu.

Knjižnica je dostopna vsem, zato se vsako leto poveča število modifikatorjev. Opisal sem zgolj nekaj pogosteje uporabljenih modifikatorjev, ki spreminjajo lastnosti entitete. Poznamo tudi drugo skupino modifikatorjev, ki je namenjena izvajanju modifikatorjev iz prve skupine. Razlikujejo se v tem, da ne spreminjajo lastnosti entitete, vendar skrbijo za izvajanje modifikatorjev, ki spreminjajo lastnost. Tako lahko vzporedno ali zaporedno izvedemo več različnih modifikatorjev. Imamo pa tudi modifikator, ki ponovno izvaja že zaključen modifikator. Tako poznamo:

- *LoopEntityModifier* – Omogoča, da se lahko modifikator izvaja v nedogled ali za določeno število ciklov. V konstruktor podamo modifikator za spreminjanje lastnosti, ki ga želimo izvajati v zanki.

- *ParallelEntityModifier* – Modifikator izvaja vzporedno več drugih modifikatorjev hkrati. V konstruktor lahko podamo neomejeno število drugih modifikatorjev. Pomen vzporednega izvajanja je, da modifikatorje združi v eno samo animacijo.
- *SequenceEntityModifier* – Nasprotno od vzporednega izvajanja imamo zaporedno izvajanje. V tem primeru se izvajanje modifikatorjev izvede zaporedno, ko se prvi zaključi se drugi začne izvajati.

```
@Override
public void onCreateScene(OnCreateSceneCallback pOnCreateSceneCallback)
        throws IOException {

    Scene scena = new Scene();
    Entity entiteta = new Entity();
    FadeInModifier modifikator1 = new FadeInModifier(100);
    DelayModifier zakasnitev = new DelayModifier(500);
    FadeOutModifier modifikator2 = new FadeOutModifier(100);
    SequenceEntityModifier zaporednoIzvajanje =
            new SequenceEntityModifier(modifikator1, zakasnitev, modifikator2);
    entiteta.registerEntityModifier(zaporednoIzvajanje);

    pOnCreateSceneCallback.onCreateSceneFinished(scena);
}
```

Slika 3.11: Zaporedno izvajanje različnih modifikatorjev.

Kompleksne animacije ustvarimo s kombinacijo različnih modifikatorjev (Slika 3.11). Vsem zgoraj naštetim modifikatorjem lahko v konstruktor podamo vmesnik, s katerim poslušamo izvajanje (Slika 3.12). Vmesnik ima pripravljeni dve metodi, ena se kliče pred začetkom izvajanja modifikatorja, druga pa se kliče šele takrat, ko se modifikator zaključi.

```
IEntityModifierListener modifierListener = new IEntityModifierListener() {

    @Override
    public void onModifierStarted(IModifier<IEntity> pModifier,
            IEntity pItem) {
        Log.i("EntityModifierListener", "Pričetek modifikatorja");
    }

    @Override
    public void onModifierFinished(IModifier<IEntity> pModifier,
            IEntity pItem) {
        Log.i("EntityModifierListener", "Modifikator je zaključen");
    }
};
```

Slika 3.12: Poslušanje spremembe modifikatorja.

3.7 Kamere

V igri lahko kamere igrajo različno vlogo. Znajdemo se lahko v situaciji, ko potrebujemo več kot le eno kamero. Glavni namen kamere je, da določeno območje v igri prikaže na zaslonu naprave. V knjižnici imamo štiri vrste kamer, vključno z osnovno kamero (Slika 3.14). Metode, ki so pomembne pri razvoju igre in so osnova vseh kamer (Slika 3.13):

Postavitev kamere

Kamera ima enak koordinatni sistem kot entitete. Položaj kamere se določi v točki, ki je postavljena v sredini. To je polovica širine in višine, ki jo podamo pri inicializaciji. Premikanje kamere nam lahko pride prav takrat, kadar skozi igro premikamo kamero in se v določenem trenutku želimo vrniti v prvotni položaj.

Prilagajanje višine in širine kamere

Konstruktor kamere kliče metodo *set*, ki nastavi višino in širino kamere. Metoda je javna in prejme najnižjo in najvišjo vrednost osi x in y. Kadar spreminjamo višino in širino kamere, moramo vedeti, da se elementi, ki so izven nastavljenega območja, ne bodo prikazali oz. se lahko tudi pomanjšajo.

Preverjanje vidljivosti

Kamera nam lahko pove, ali je določena entiteta v vidnem območju. Vidnost entitete preverimo s klicem metode *isEntityVisible*, ki prejme kot parameter entiteto. Preverjanje vidljivosti entitete nam lahko zelo koristi. Entitete, ki odidejo iz vidnega območja kamere, lahko ponovno uporabimo in nam ni treba ustvarjati novega objekta.

Sledenje entiteti

Pri določenih igrah kamera sledi objektu, ki se premika po zaslonu. Avtor knjižnice nam je sledenje entiteti omogočil s klicem metode *setChaseEntity*. Kamera sledi entiteti, ki se premika po koordinatnem sistemu. Vhodni parameter metode je entiteta, ki ji sledimo.

```
private final float SIRINA = 800, VISINA = 480;
private Camera kamera;

@Override
public EngineOptions onCreateEngineOptions() {

    kamera = new Camera(0, 0, SIRINA, VISINA);

    EngineOptions mEngineOptions = new EngineOptions(true,
            ScreenOrientation.PORTRAIT_FIXED,
            new FillResolutionPolicy(), kamera);
    return mEngineOptions;
}

@Override
public void onCreateScene(OnCreateSceneCallback pOnCreateSceneCallback)
        throws IOException {

    Scene scena = new Scene();
    Entity entiteta = new Entity();

    kamera.setCenter(0, 0);
    kamera.set(-SIRINA/2, -VISINA/2, SIRINA/2, VISINA/2);
    kamera.isEntityVisible(entiteta);
    kamera.setChaseEntity(entiteta);

    pOnCreateSceneCallback.onCreateSceneFinished(scena);
}
```

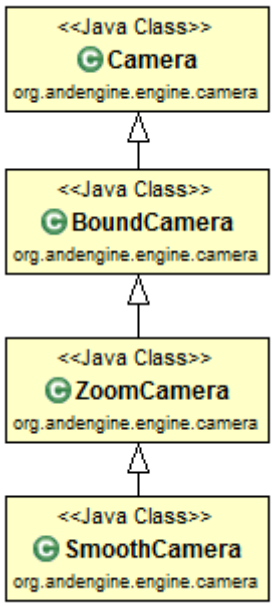
Slika 3.13: Osnovne funkcionalnosti vseh kamer.

3.7.1 Razlike med kamerami

Osnovna kamera ima podkamero, ki jo imenujemo kamera z mejo (ang. *BoundCamera*). Kameri natančno določimo meje območja, kjer lahko potuje po koordinatnem sistemu. Pri določanju mej kamere kličemo metodo *setBounds*, ki na vhod prejme najnižjo in najvišjo vrednost meje osi x in y.

Če raziskujemo globlje, lahko vidimo, da je avtor knjižnice vključil kamero, ki enostavno poveča ali zmanjša objekte (ang. *ZoomCamera*). S klicem metode *setZoomFactor* kameri nastavimo faktor večanja ali manjšanja. Na primer, če nastavimo faktor na 1˙5, se bodo entitete povečale, če nastavimo faktor na 0˙5, se bodo entitete zmanjšale.

SmoothCamera kamera je najbolj napredna kamera, saj podeduje vse funkcionalnosti drugih kamer. Poleg zgoraj že omenjenih funkcionalnosti ima kamera še dodatno funkcionalnost, ki omogoča gladko pomikanje kamere. To pomeni, da se premik kamere zgodi z določeno hitrostjo. Prav tako lahko določimo hitrost povečave in pomanjšave kamere. V konstruktorju podamo poleg velikosti kamere še prej omenjeno hitrost.



Slika 3.14: Hierarhija kamer.

3.7.2 Prikazovalnik HUD

HUD (ang. Head-up display) je vrsta prikazovalnika, ki nam je vedno vidna. Namen takšnega prikazovalnika je prikaz gumbov, teksta ali katere koli druge entitete. Ključni pomen prikazovanja je ta, da so entitete vedno vidne na zaslonu ne glede na položaj kamere. Prav tako bodo entitete vedno v ospredju vsake scene. Uporaba prikazovalnika je v knjižnici zelo preprosta (Slika 3.15).

```
private final float SIRINA = 1080, VISINA = 1920;
private Camera kamera;

@Override
public EngineOptions onCreateEngineOptions() {

    kamera = new Camera(0, 0, SIRINA, VISINA);

    EngineOptions mEngineOptions = new EngineOptions(true, ScreenOrientation.PORTRAIT_FIXED,
            new FillResolutionPolicy(), kamera);
    return mEngineOptions;
}

@Override
public void onCreateScene(OnCreateSceneCallback pOnCreateSceneCallback)
        throws IOException {

    Scene scena = new Scene();

    Entity gumb = new Entity();
    HUD hud = new HUD();

    hud.attachChild(gumb);

    kamera.setHUD(hud);
    pOnCreateSceneCallback.onCreateSceneFinished(scena);
}
```

Slika 3.15: Ustvarjanje in pripenjanje entitete na prikazovalnih HUD.

Dobro je vedeti:

1. Entitete se ne bodo pomikale s kamero. Ko je položaj nastavljen, bo entiteta ostala na mestu, razen če ji spremenimo položaj.
2. Entitete bodo vedno prikazane pred vsako pripeto entiteto na sceni ne glede na lastnost, ki določa vrstni red entitet (ang. z-index). To pomeni, da se najprej izriše scena in vse njene pripete entitete. Na koncu, ko je kamera tista, ki določi pogled na sceno, se izvedejo oz. izrišejo še pripete entitete prikazovalnika HUD.

3.8 Razširitve za knjižnico

Za knjižnico AndEngine obstajajo različne razširitve, ki vsebujejo dodatne funkcionalnosti in izboljšajo sam razvoj igre. V tem poglavju bom predstavil razširitev za vključevanje fizike v igro, razširitev za živa ozadja in razširitev za vključevanje vektorskih slik. Obstajajo še druge razširitve, kot so:

- **Večigralska razširitev** – Z razširitvijo razvijalec enostavno implementira igro namenjeno več igralcem. Ti lahko igro igrajo istočasno na različnih napravah. Naprava se tako obnaša kot strežnik ali kot odjemalec.
- **Razširitev za ustvarjanje igre po konceptu WYSIWYG** – To pomeni, da uporabnik preko uporabniškega vmesnika vidi končni izgled igre. S pomočjo programa *CocosBuilder* lahko zgradimo strukturo igre in celotno strukturo izvozimo. Z razširitvijo lahko izvožene datoteke vključimo v igro.
- **Razširitev za pripenjanje entitet na živo sliko** – Razvijalcu omogoča, da lahko uporabi že pripravljeno aktivnost. Na njej se v ozadju prikazuje živa slika kamere, ki ji lahko pripnemo različne entitete.
- **Razširitev za vključitev tekstur izvoženih s programom *TexturePacker*** – Programom omogoča, da lahko uporabnik različne sličice združi v eno samo in jo izvozi. Razširitev omogoča, da lahko izvožene datoteke vključimo v igro.
- **Razširitev za vključevanje sestavljenega okolja** – Razširitev razvijalcu omogoča, da lahko sestavljeno okolje, ki je shranjeno v datoteki tipa *tmx*, vključi v svojo igro. Na spletu lahko najdemo različna orodja, ki omogočajo grajenje sestavljenega okolja. Takšen pristop lahko razvijalcu olajša delo, saj mu okolja ni treba graditi s kodo.

3.8.1 Vključevanje fizikalnih lastnosti

Igre, ki temeljijo na fiziki, so danes ene izmed najbolj priljubljenih vrst iger na mobilnih napravah. Sama razširitev omogoča ustvarjanje iger, ki temeljijo na fiziki. Avtor je uporabil knjižnico imenovano Box2D (avtorja Erina Cattoja), ki je namenjena simulaciji togih teles v 2D okolju. Z razredom *PhysicsWorld* zgradimo realistično 2D okolje.

Knjižnica Box2D nam daje možnost ustvarjanja različnih vrst teles, ki vplivajo na simulacijo (Slika 3.16). Ustvarimo lahko **dinamična telesa**, ki reagirajo na sile drugih teles. Nasprotno od dinamičnih teles lahko ustvarimo **statična telesa**, ki jih ni moč premikati. Zadnja vrsta teles, ki jih lahko ustvarimo, so **kinematična telesa**, prednost teh teles je, da jih je moč premikati, vendar na njih ne vpliva nobena druga sila ali telo.

```
mPhysicsWorld = new PhysicsWorld(
        new Vector2(0f, -SensorManager.GRAVITY_EARTH * 2), false);

FixtureDef BoxBodyFixtureDef = PhysicsFactory.createFixtureDef(20f, 0f, 0.5f);

mScene.registerUpdateHandler(mPhysicsWorld);

Rectangle staticni = new Rectangle(cameraWidth / 2f, 75f, 400f, 40f,
        this.getVertexBufferObjectManager());
Body staticnoTelo = PhysicsFactory.createBoxBody(mPhysicsWorld, staticni,
        BodyType.StaticBody, BoxBodyFixtureDef);
mScene.attachChild(staticni);

Rectangle dinamicni = new Rectangle(400f, 120f, 40f, 40f,
        this.getVertexBufferObjectManager());
Body dinamicnoTelo = PhysicsFactory.createBoxBody(mPhysicsWorld, dinamicni,
        BodyType.DynamicBody, BoxBodyFixtureDef);
mScene.attachChild(dinamicni);

Rectangle kinematicno = new Rectangle(600f, 180f, 40f, 40f,
        this.getVertexBufferObjectManager());
Body kinematicnoTelo = PhysicsFactory.createBoxBody(mPhysicsWorld, kinematicno,
        BodyType.KinematicBody, BoxBodyFixtureDef);
mScene.attachChild(kinematicno);
```

Slika 3.16: Ustvarjanje različnih tipov teles.

Telesa lahko kategoriziramo. To pomeni, da jim lahko določimo kategorijo in da reagirajo zgolj na telesa, ki so znotraj iste kategorije. Včasih potrebujemo telo, ki ima različne atribute fizike na različnih delih telesa. Z uporabo razreda *Body* lahko enostavno sestavimo telo z različnimi fizikalnimi lastnostmi (Slika 3.17). Telesom poleg osnovnih oblik lahko določimo svojo obliko, ki jo opišemo s točkami. Za premikanje telesa v prostoru imamo na izbiro uporabo linearne ali kotne sile, nastavitev linearne ali kotne hitrosti in uporabo kotne sile v obliki navora.

V razširitvi lahko dve telesi povežemo oz. ustvarimo spoj med njima (Slika 3.18). Spoji se lahko med simulacijo ustvarijo ali uničijo. Poznamo različne spoje, ki imajo različne fizikalne lastnosti. Eden izmed pogosteje uporabljenih spojev v knjižnici je spoj *Revolute Joint.* Na sliki 3.19 lahko vidimo telesa, ki so speta z dvema spojema *Revolute Joint*. Glavna lastnost tega spoja je, da vsebuje pogon in rotacijsko točko (Slika 3.18). Pogonu lahko določimo hitrost in navor. Speto telo se bo tako vrtelo okoli določenega telesa. Smer vrtenja teles je odvisna od lastnosti *motorSpeed*. Če je število negativno, bo vrtenje v smeri urinega kazalca, pri pozitivnem številu se bo telo vrtelo v nasprotni smeri urinega kazalca.

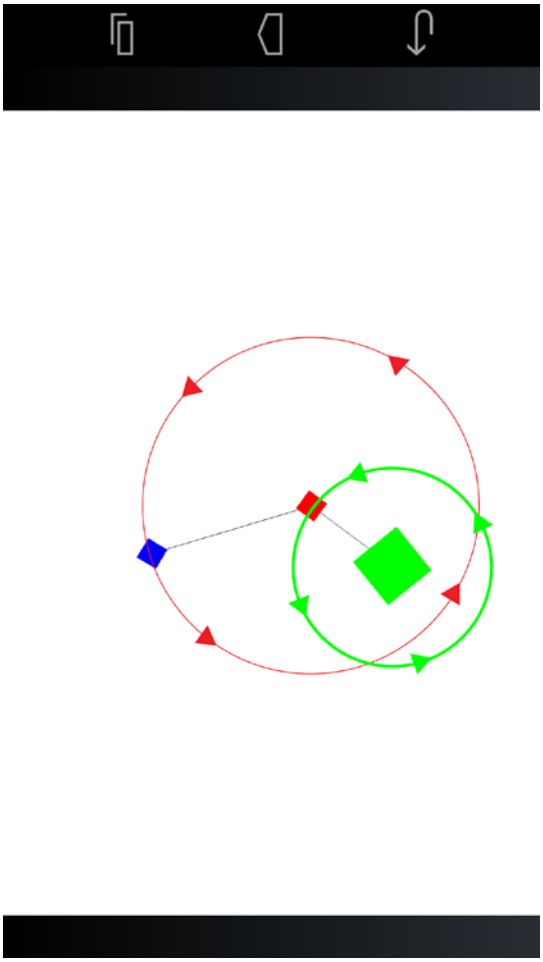
```
Body sestavljenoTelo = mPhysicsWorld.createBody(new BodyDef());
sestavljenoTelo.setType(BodyType.DynamicBody);

FixtureDef teloADef = PhysicsFactory.createFixtureDef(20, 0.0f, 0.5f);
final PolygonShape teloAOblika = new PolygonShape();
teloAOblika.setAsBox(1, 1, new Vector2(0.5f, 0.5f), 0f);
teloADef.shape = teloAOblika;
sestavljenoTelo.createFixture(teloADef);

FixtureDef teloBDef = PhysicsFactory.createFixtureDef(20, 2.0f, 0.5f);
final PolygonShape teloBOblika = new PolygonShape();
teloBOblika.setAsBox(1, 1, new Vector2(1.5f, 1.5f), 0f);
teloBDef.shape = teloBOblika;
sestavljenoTelo.createFixture(teloBDef);

FixtureDef teloCDef = PhysicsFactory.createFixtureDef(10, 0.0f, 0.5f);
final PolygonShape teloCOblika = new PolygonShape();
teloCOblika.setAsBox(1, 1, new Vector2(-1.5f, -1.5f), 0f);
teloCDef.shape = teloCOblika;
sestavljenoTelo.createFixture(teloCDef);
```

Slika 3.17: Sestavljeno telo z različnimi fizikalnimi lastnostmi.

```
final RevoluteJointDef revoluteJointDef = new RevoluteJointDef();
revoluteJointDef.initialize(staticnoTelo, vrtljivoTelo, staticnoTelo.getWorldCenter());
revoluteJointDef.enableMotor = true;
revoluteJointDef.motorSpeed = 10;
revoluteJointDef.maxMotorTorque = 20;

this.mPhysicsWorld.createJoint(revoluteJointDef);
```

Slika 3.18: Spenjanje dveh teles.

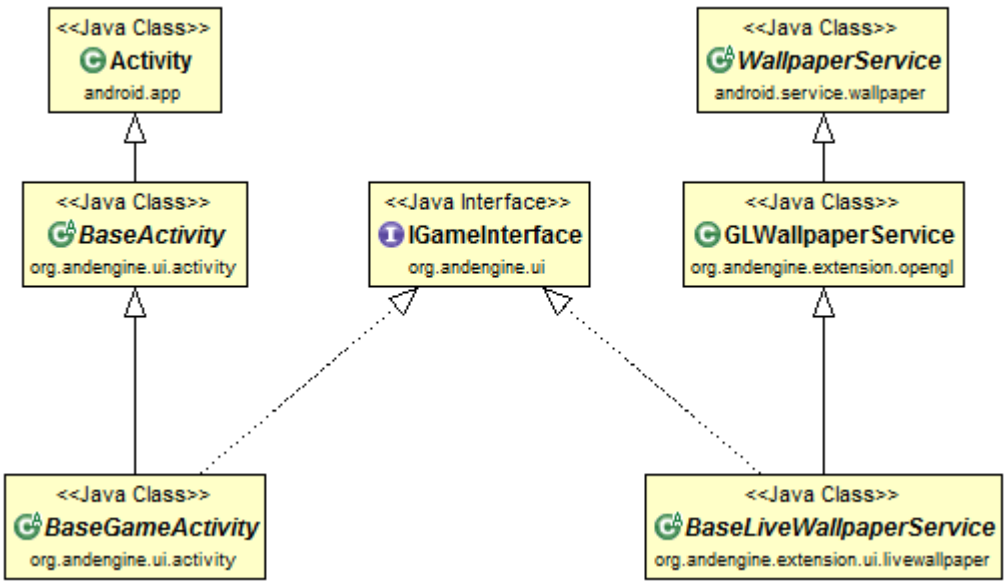
Vidimo lahko, da nam razširitev omogoča različne funkcionalnosti nad telesi. S kombiniranjem različnih funkcionalnosti lahko simuliramo lutko, ki je sestavljena iz različnih teles in je speta s spoji. Prav tako lahko simuliramo vrv. Razvijalec ima v tej razširitvi odprte možnosti. Sama fizika se dopolnjuje in izboljšuje.



Slika 3.19: Vrtenje spetih teles.

3.8.2 Razširitev za živa ozadja

Živa ozadja so v mobilnih napravah Android interaktivna animirana ozadja. Z razširitvijo lahko enostavno ustvarimo živa ozadja z vsemi funkcionalnostmi, ki jih ponuja knjižnica. Uporaba je identična kot pri razvijanju igre. Razlika je samo v tem, da pri živem ozadju ne ustvarimo aktivnosti, ampak uporabimo storitev. Pri živem ozadju uporabljamo aktivnost zgolj za nastavljanje določenih parametrov. Razširitev prekrije in preuredi metode razreda *WallpaperService*, ki so potrebne za normalno delovanje knjižnice. Za implementacijo uporabimo pripravljen razred *BaseLiveWallpaperService*, ki uporablja vmesnik *IGameInterface*. Enak vmesnik se uporabi tudi v razredu *BaseGameActivit*, uporabljamo ga pri ustvarjanju aktivnosti (Slika 3.20). Ker oba razreda uporabljata isti vmesnik, se sama implementacija živega ozadja in igre ne razlikuje.



Slika 3.20: Razlika med aktivnostjo in storitvijo.

3.8.3 Razširitev za vključevanje vektorske grafike

Velika težava pri razvijanju igre za mobilne naprave je različna velikost in ločljivost zaslona. Pri razvijanju igre se s tem problemom soočimo tako, da celoten pogled povečamo ali pomanjšamo. Pri tem dejanju trpi rastrska grafika, saj pri dovolj veliki povečavi izgubimo kakovost grafike (Slika 3.21). Ena izmed možnosti, ki jih ponuja sam Android je, da ustvarimo različne velikosti grafike in jo kategoriziramo v določene mape. Druga možnost pa je, da uporabimo vektorsko grafiko. Prednost vektorske grafike je, da se kakovost grafike ohrani ne glede na njeno velikost. Razširitev razvijalcu omogoča, da lahko ustvari teksture, ki vključujejo vektorsko grafiko tipa svg. Slabost razširitve je ta, da ne podpira vseh elementov, ki jih ponuja sam standard SVG. Razširitev podpira elemente, kot so poti, gradienti, barvna polnila in še nekatere oblike. Vsi elementi v razširitvi, ki niso podprti, vplivajo na čas nalaganja, zato jih je pametno izločiti iz datoteke. Nalaganje teksture traja veliko dlje kot običajno nalaganje, saj grafiko pretvori v rastrsko in jo naloži v pomnilnik. Da bi zmanjšali čas nalaganja, si ob prvem zagonu shranimo pretvorjeno grafiko na napravo.

Slika 3.21: Razlika med rastrsko (levo) in vektorsko (desno) sliko.

Za določitev območja uporabljamo pripravljene metode, ki se nahajajo v razredu *SVGBitmapTextureAtlasTextureRegionFactory*. Uporaba metod ni nič drugačna kot pri običajnih teksturah. Razlikuje se v tem, da je treba metodam dodati dva parametra, ki sta namenjena za širino in višino. Pri rastrskih slikah sta višina in širina, za razliko od vektorskih slik, že v osnovi določeni. Zato je treba podati višino in širino, saj lahko sliko pomanjšamo ali povečamo. Grafiko prikažemo na zaslonu po enakem postopku, kot smo ga že omenili v poglavju 3.4.

```
ITextureRegion mSVGslicica = SVGBitmapTextureAtlasTextureRegionFactory
        .createFromAsset(atlasObmocij, this, "slicica.svg", 100, 100,
                new ISVGColorMapper() {

    @Override
    public Integer mapColor(Integer pColor) {
        int alfa = Color.alpha(pColor);
        int rdeca = Color.red(pColor);
        int zelena = Color.green(pColor);
        int modra = Color.blue(pColor);

        return Color.argb(alfa, zelena, modra, rdeca);
    }
});
```

Slika 3.22: Preslikanje barve.

Pri določanju območja teksture lahko teksturi preslikamo barvo. Preslikanje nam pride prav takrat, ko želimo uporabniku omogočiti, da določenemu delu grafike spremeni barvo. Razvijalec ima podan vmesnik *ISVGColorMapper,* v katerem lahko preslika barvo (Slika 3.22).

4 IGRA JUNIOR 3R

V sodelovanju z Okoljskim raziskovalnim zavodom, ki izvaja storitve na področju ravnanja z odpadki [19], smo se odločili, da razvijemo poučno igro za otroke. Igro smo poimenovali Junior 3R. Kratica 3R predstavlja strategijo za ravnanje z odpadki, s katero želimo zmanjšati, ponovno uporabiti in reciklirati odpadke. Pri projektu so sodelovali grafična oblikovalka Mojca Kranjc, ki je poskrbela za vizualno podobo igre, dr. Marinka Vovk, ki je vodila sestanke in nam nudila strokovne nasvete o pravilnem ravnanju z odpadki, ter Aleš Kos, tehnični svetovalec.

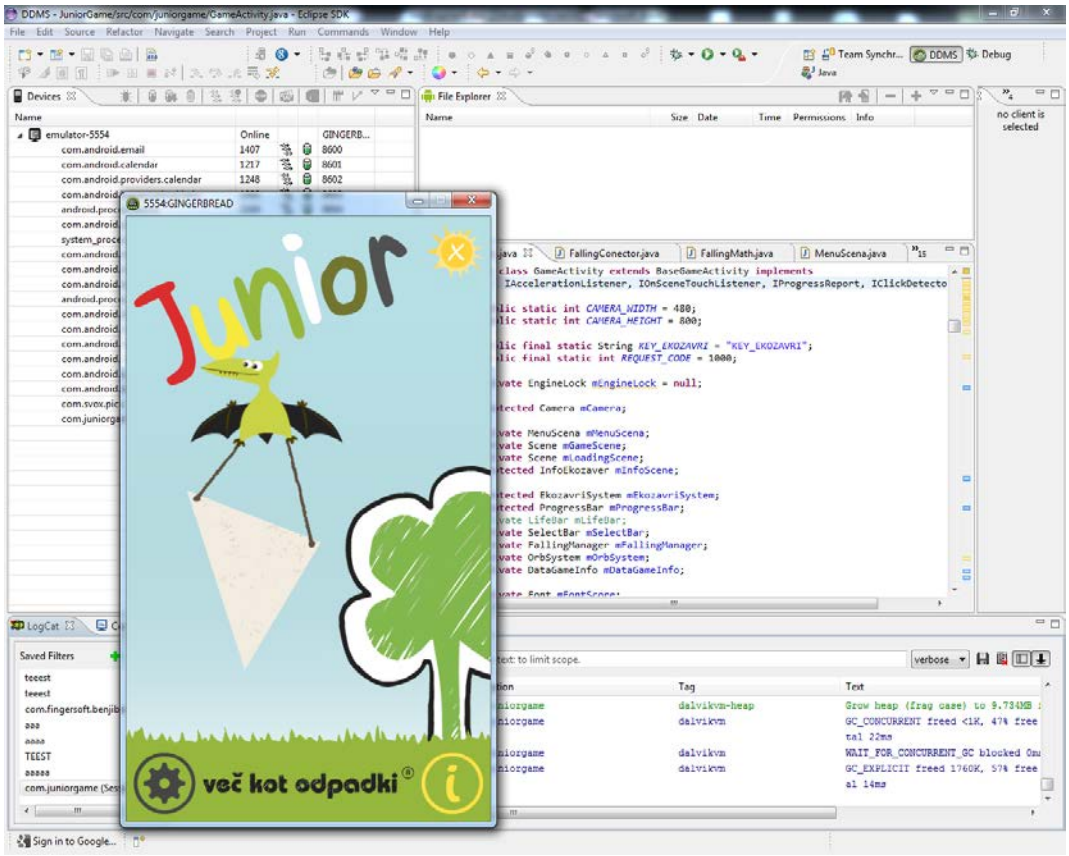
Glavni namen igre je, da se otroci na zabaven način naučijo ločevati odpadke. Osnovna ideja igre je, da ekozaver poje določeno vrsto odpadkov. Ekozaver je neke vrste dinozaver, ki je le določene odpadke. V igri je tako osem različnih ekozavrov in vsak se prehranjuje z določeno vrsto odpadkov. Ko ekozaver zaužije določeno število pravilnih odpadkov, mora oditi na stranišče, kjer iz zaužitih odpadkov nastane nov produkt. Katere odpadke lahko ekozaver poje in kakšen produkt nastane, prikazuje tabela 4.1.

Tabela 4.1: Hrana ekozavrov in nastali produkti

Ekozaver	Odpadki	Produkt
Biozaver	biološki odpadki	gnojilo
Papirozaver	papir	škatla
Embozaver	embalaža	plastični stol
Steklozaver	steklo	steklena volna
Kosozaver	kosovni odpadki	nova omara
Ostalozaver	ostali odpadki	električna energija, odlagališče
Elektrozaver	električni aparati in električna oprema	nove električne naprave
Nevarnozaver	nevarni odpadki	električna energija

4.1 Orodja za izdelavo igre

Pri razvijanju igre sem uporabljal različna orodja, ki sem jih razdelil v dve skupini, in sicer razvojna orodja programske opreme in grafična orodja. Za razvoj sem uporabljal programsko okolje Eclipse z vtičnikom Android Developer Tools (ADT) (Slika 4.1). Eclipse je odprtokodno integrirano razvojno okolje (IDE). Namenjeno je predvsem razvoju aplikacij v programskem jeziku Java. S pomočjo vtičnega sistema pa lahko razvijamo aplikacije tudi v drugih programskih jezikih, kot so C, C++, Fortran, JavaScript, Perl, PHP ... Vtičnik ADT vključuje razvojni programski komplet (SDK) za Android, ki je ključnega pomena pri razvoju Android aplikacij. SDK vsebuje poleg osnovnih knjižnic za razvoj aplikacij še dodatna orodja, ki pripomorejo k lažjemu in hitrejšemu razvoju. Celotna grafična podoba igre je izrisana vektorsko, zato je orodje Adobe Illustrator najbolj primerno za ustvarjanje in urejanje grafike. Ker pa težko najdemo igro brez animiranih slik, je družina Adobe priskrbela orodje Adobe Flash, ki je namenjeno ustvarjanju interaktivne večpredstavne vsebine, ki se pogosto pojavi na spletnih straneh. Prednost orodij Adobe je ta, da se med seboj dobro povezujejo in dopolnjujejo.



Slika 4.1: Programsko okolje Eclipse z Android orodji.

4.2 Potek igre

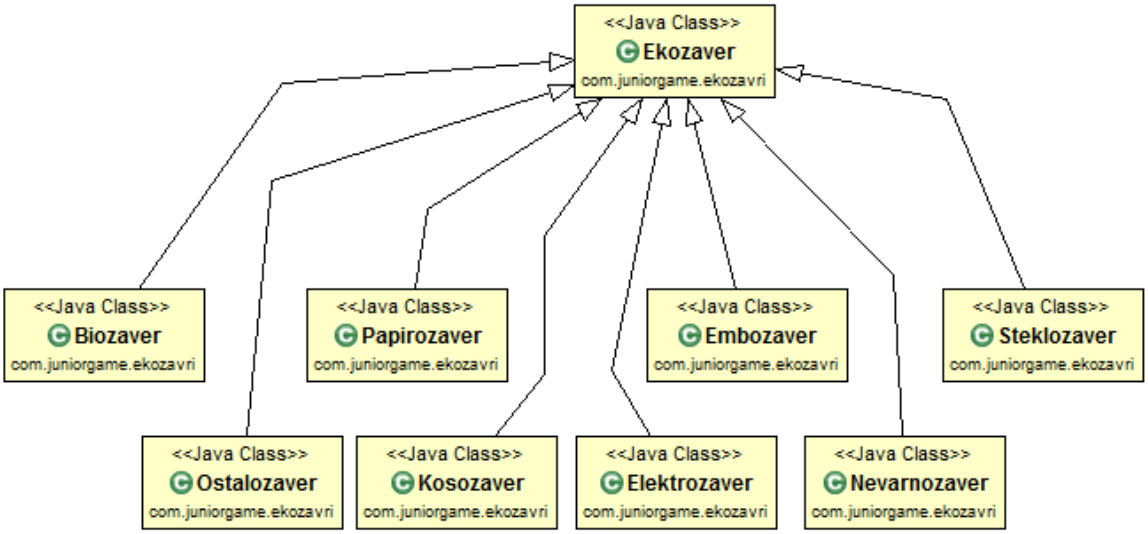
Da bi bila igra lažje razumljiva, sem jo razdelil na pet scen. Vsaka scena ima različne funkcionalnosti, ki si sledijo po zaporedju od samega zagona do prekinitev in zaključka igre.

4.2.1 Vstopna scena

Ob zagonu igre prileti Ostalozaver, ki nosi gumb za začetek igre. V levem kotu spodaj vidimo gumb, ki nas vodi v nastavitve, v desnem kotu spodaj pa je gumb, ki nas vodi do informacij o sami igri. Sonček s križcem je namenjen izhodu iz igre (Slika 4.2). V prvi prikazani sceni lahko vidimo, da imamo zelo malo objektov, saj želimo igro ob zagonu čim hitreje prikazati.

Slika 4.2: Videz vstopne scene na napravi.

Vsak prikazan ekozaver se predstavlja kot entiteta, ki je pripeta na sceno. Ker pa imajo vsi ekozavri podobne funkcionalnosti, kot so na primer animacija hoje, odpiranje ust itd., sem jim dodelil nadrazred imenovan *Ekozaver* (Slika 4.3). Na sceni so tako pripeti gumbi, entiteta tipa Ostalozaver in sličici za prikaz drevesa ter travnate površine. Gumbi so nekakšne vrste entitete, ki vsebujejo dodatno funkcionalnost za klik. Da se lahko Ostalozaver premika po sceni, sem uporabil modifikator tipa *PathModifier*.



Slika 4.3: Hierarhija ekozavrov.

4.2.2 Scena namenjena prikazu ekozavrov

Po pritisku na gumb za začetek igre se nam prikaže druga scena, ki je namenjena prikazu ekozavrov v svojem naravnem okolju. Cilj igre je, da z zbiranjem žetonov igralec odklene zaklenjene ekozavre. Sličice se samodejno premikajo levo in desno, lahko pa jih premikamo tudi sami. Pod sličico so prikazani žetoni. Sivi žetoni ponazarjajo še neosvojene žetone, osvojeni žetoni pa so prikazani z zeleno barvo. Zgoraj lahko vidimo igralčev najboljši rezultat. Za vrnitev na prejšnjo sceno pritisnemo na gumb v obliki hišice (Slika 4.4). Igralec začne z igro šele takrat, ko pritisne na gumb v obliki trikotnika ali ko pritisne na sličico.

Število žetonov in število točk je shranjeno v za to posebej namenjenemu prostoru (ang. SharedPreferences), do katerega ni mogoče dostopati z drugimi aplikacijami. S tem igralcu preprečimo preurejanje rezultata. Entiteta za prikaz ekozavrov v okolju ima otroke, ki predstavljajo sličice posameznih ekozavrov. Vsaka sličica ima dve stanji (odklenjeno in zaklenjeno). Vsaki sličici, ki odide s sredine zaslona, se preuredi entiteta žetonov na prihajajočo sličico.

Slika 4.4: Scena namenjena prikazu ekozavrov v svojem naravnem okolju.

4.2.3 Igralna scena

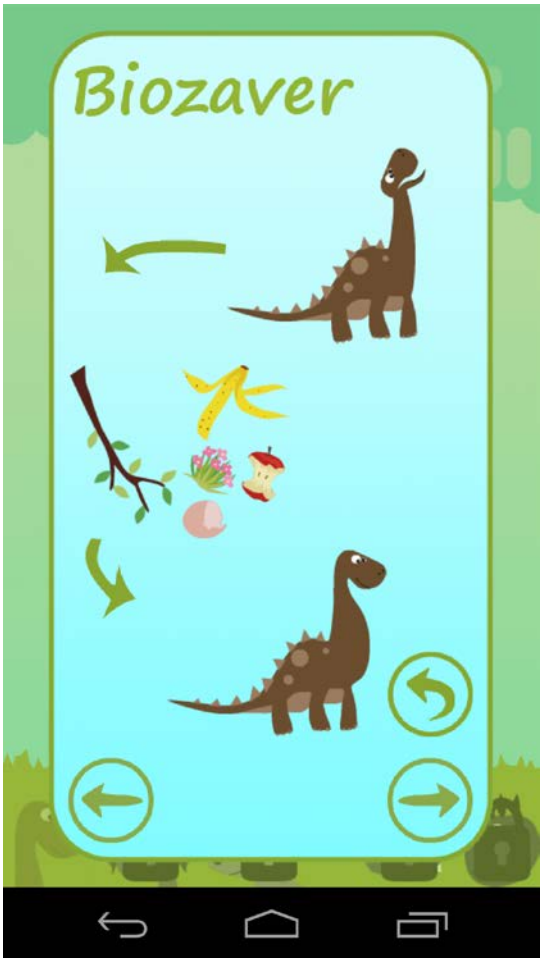
Pred začetkom igre je treba počakati, da se v pomnilnik naložijo vsa sredstva, zato se za kratek čas prikaže nalagalna scena. Ob prvem zagonu igralca z animacijami poučimo o načinu igranja igre. Animacija prsta igralcu nakaže poteg ekozavra na travnato površino (Slika 4.5). Ko igralec opravi poteg, ga moramo poučiti, kako se ekozaver premika levo in

desno. To se prikaže z animacijo naprave. Po nagibu naprave levo in desno, kot to prikazuje animacija, igralcu podamo še informacijo o prehrani ekozavrov (Slika 4.6).

Odpadki naključno in počasi začnejo padati. Ekozaver poje odpadek samo takrat, ko ima odprta usta (Slika 4.7). V primeru, da ekozaver poje pravilni odpadek, se prikaže simbol za pravilno dejanje, v nasprotnem primeru se prikaže simbol za nepravilno dejanje.

Slika 4.5: Simuliranje prsta.

V zgornjem delu zaslona je prikazan indikator, ki nam kaže razmerje med pravilnimi in nepravilnimi dejanji. Ko je igralec uspešen in pride do maksimalne pravilnosti, osvoji žeton. V nasprotnem primeru se igra zaključi in se prikaže rezultat. Igralec tako zbira žetone. Z določenim številom žetonov lahko igralec odklene novega ekozavra.

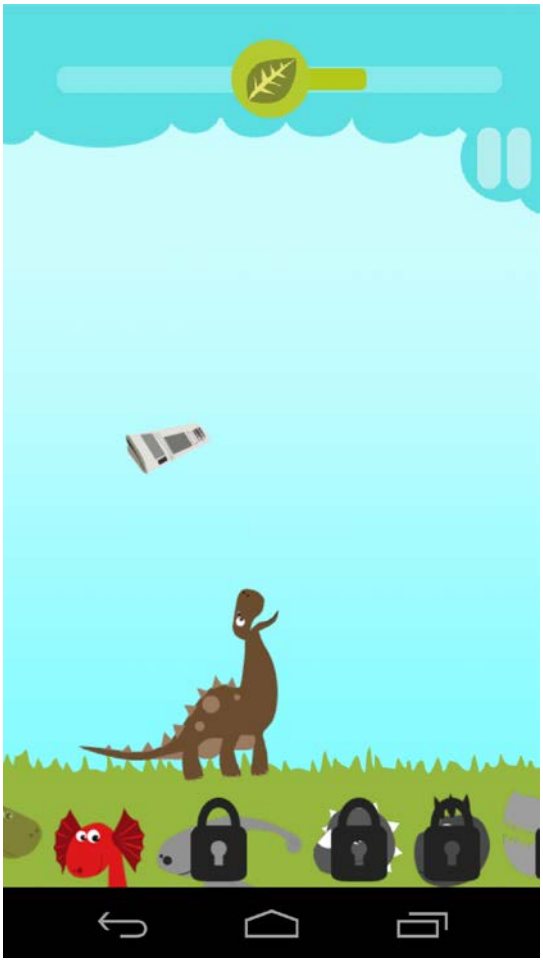


Slika 4.6: Informacije o prehrani ekozavra.

Nalagalna scena vsebuje zgolj ozadje, saj jo želimo čim hitreje prikazati na zaslonu. Takoj ko se vsi elementi naložijo in je igra pripravljena, se nalagalna scena odstrani in se nastavi igralna scena. Simulacija se izvede samo takrat, kadar je postavljena zastavica za simuliranje. Ko igralec uspešno izvede nalogi, ki prikazujeta simulacijo, se zastavica spremeni in shrani v poseben prostor. Ko sta nalogi opravljeni, se prikažejo informacije o prehrani ekozavra. Informacije prikažemo tako, da igralni sceni pripnemo novo sceno. Izvajanje metode *onManagedUpdate* se v trenutku, ko pripnemo drugo sceno, zaustavi. V tem trenutku se prav tako ustavi izvajanje pripetih entitet. Vedno, ko posodobimo igralno sceno, preverimo, ali imamo pripeto drugo sceno.

Ekozavra premikamo zgolj s senzorjem, ki je vgrajen v napravo. Senzorju pravimo pospeškometer. Metoda *onAccelerationChanged* se kliče samo takrat, ko senzor zazna spremembo. V igri uporabljamo samo spremembe, ki se zgodijo na x osi. Večji bo nagib naprave, hitreje se bodo pomikali ekozavri in večja bo vrednost odklona.

Z metodo *nextInt* razreda *Random* pridobimo naključno število, ki je v območju našega zaslona. Število uporabimo pri položaju odpadka. Frekvenca in hitrost padanja se stopnjujeta glede na količino pravilno zaužitih odpadkov. S pomočjo interpolacijskega polinoma sem izračunal pravilno razmerje med frekvenco in hitrostjo padanja.



Slika 4.7: Izgled igre Junior 3R.

4.2.4 Scena namenjena prikazu produkta

Vsak ekozaver je sit takrat, ko zaužije določeno količino pravilnih odpadkov. Odpadki v tem trenutku padajo mimo ekozavra. Ekozavra, ki je sit, odpeljemo na stranišče, kjer se izprazni. Z animacijo ponazorimo izpraznitev ekozavra ter prikažemo novi produkt (Slika 4.8). S tem želimo igralcem pokazati, da iz odpadkov ustvarimo nov produkt, ki ga ponovno uporabimo.

Slika 4.8: Prikaz produkta.

Ko je ekozaver sit, se pokliče metoda *toMainMove*, ki se nahaja v glavni aktivnosti. Metoda zažene novo aktivnost, v kateri se zažene animacija. Animacijo sem opisal z jezikom XML. Datoteka z opisom je shranjena na napravi, ki vsebuje vrsto entitete in informacije o modifikatorjih, ki jih naj izvede oz. pripne na entiteto.

4.2.5 Prekinitvena scena

Igralec lahko igro v vsakem trenutku zaustavi in nadaljuje. To stori tako, da pritisne na gumb, ki je desno zgoraj (ang. *pause)*. Po pritisku se prikaže scena, na kateri igralec vidi število doseženih točk in število zbranih žetonov (Slika 4.9). Na sceni je tudi knjiga, ki predstavlja gumb oz. leksikon ekozavrov in vsebuje informacije o njihovi prehrani.

Slika 4.9: Videz prekinitvene scene.

Prekinitvena scena se pripne na igralno sceno, ki avtomatično zaustavi izvajanje. Metodi *setChildScene* podamo sceno in parametre, ki določajo izvajanje same scene. Eden izmed parametrov je namenjen izrisovanju očetovske scene. S tem lahko pripeti sceni nastavimo prozorno ozadje, na katerem bo vidna tudi očetovska scena.

4.2.6 Igra v praksi

Sam razvijalec točno ve, kako igrati igro in ima občutek, da so vse vključene funkcionalnosti popolnoma razumljive uporabnikom. Hitro pa se lahko izkaže, da so te funkcionalnosti za navadnega uporabnika prezapletene ali nerazumljive. Uporabniki so bili navdušeni nad izgledom igre Junior 3R, vendar pa so že na začetku naleteli na manjše težave. Izkazalo se je, da se gumb za začetek igre preveč zliva z okoljem, kar posledično zmede uporabnika, ki ne ve točno, kje začeti z igranjem. Skozi igro naletijo na težave

povezane s premikanjem ekozavrov na travnato površino, kar na začetku ne predstavlja večjih problemov, saj je padanje odpadkov počasnejše in je na razpolago manj ekozavrov. Uporabnikom je bila všeč celotna grafična podoba igre, ki jih je tudi pritegnila k igranju. Z zbiranjem žetonov so želeli čim hitreje odkleniti ostale ekozavre, kar jih je spodbudilo k tekmovalnosti. Navdušila jih je animacija, ko ekozavra odpeljemo na stranišče, saj je prikazana na duhovit način. To pa je igralce motiviralo za nadaljnjo igranje.

5 SKLEP

V diplomskem delu sem predstavil knjižnico AndEngine. Poleg osnovnih elementov, ki jih ponuja knjižnica, sem opisal in predstavil tudi razširitve same knjižnice. Vso znanje, ki sem ga o knjižnici usvojil, sem prenesel na praktični primer. Tako je nastala igra Junior 3R.

Projekt oz. igra se je razvila v sodelovanju z Okoljskim raziskovalnim zavodom in je namenjena otrokom, ki se tako lahko na zabaven način naučijo ločevati odpadke. Bistvo igre je, da otrokom pokažemo, kako pravilno ravnati z odpadki in kako določene odpadke ponovno uporabiti. S tem jih posledično naučimo, da lahko tudi sami vplivajo na zmanjšanje nastalih odpadkov. V igri imamo tako imenovane ekozavre, ki predstavljajo zabojnike, v katere padajo oz. v njih odvržemo odpadke. S pravilnim razvrščanjem odpadkov igralec odkriva nove vrste ekozavrov.

Delo razkriva, da bi razvoj igre moral biti dokaj enostaven, vendar se stvari kljub temu lahko hitro zapletejo. Čas razvoja se s kompleksnostjo stopnjuje, to pomeni, da bolj kot bo kompleksna igra, več časa bomo porabili za razvoj. Zato je pomembno, da se držimo pravila *»Najboljše stvari so preproste, a ne enostavne«*[13]. Za razvoj igre ni dovolj le programersko znanje, saj sem naletel tudi na matematične in na fizikalne probleme, pridobiti pa sem moral tudi znanje o pravilnem ločevanju odpadkov. Hitrost razvijanja je velikokrat odvisna od dokumentacije, saj dobro dokumentirane knjižnice hitro privedejo do pravih rešitev. Na internetu je poleg osnovne dokumentacije dostopna še knjiga, v kateri je predstavljena celotna knjižnica AndEngine. Obstajajo tudi forumi, ki so namenjeni zgolj knjižnici AndEngine. Na teh forumih lahko zastavimo vprašanja o problemih povezanih s knjižnico AndEngine in poiščemo rešitve za različne težave, ki se pojavijo pri samem razvoju. Vedeti moramo, da je knjižnica odprtokodna in omogoča hitro odkrivanje morebitnih napak. Uporabnikov, ki so zainteresirani za knjižnico in želijo slediti vsem novostim, je 2.227. Uporabnikov, ki gradijo samo knjižnico (izboljševanje, popravljanje in vključevanje novih idej), je 1.223 [7]. Vidimo lahko, da je knjižnica pritegnila veliko zanimanja.

V sami knjižnici se lahko pojavijo tudi nepravilnosti (npr. pri prikazovanju sličice na zaslonu). Ugotovil sem, da se napaka pojavi pri nalaganju slike, ki je kategorizirana v različne mape z različnimi velikostmi. Pri nalaganju oz. dekodiranju slike Android samodejno pomanjša ali poveča sliko glede na tip zaslona. Prav tako nam knjižnica samodejno poveča ali pomanjša sličico glede na določeno politiko ločljivosti. Posledično nam knjižnica nepravilno določi območje slike in jo prikaže napačno. Napako sem odpravil tako, da sem pri nalaganju sličice Androidu onemogočil samodejno pomanjšanje ali povečanje slike.

Način igranja igre Junior 3R vključuje gibanje prstov po zaslonu, zato igra ni primerna za otroke, saj še nimajo tako razvite motorike kot odrasla oseba. Posledično bi bilo treba premike poenostaviti ali pa povsem spremeniti način igranja igre. Tako ne bi več pobirali padajočih odpadkov, ampak bi bili odpadki postavljeni v okolje, kjer bi igralec zgolj nahranil ekozavre (igralec bi s prstom odpadek premaknil ekozavru v usta).

Igra je sama po sebi zanimiva, na kar kažejo zadovoljni uporabniki. Z delom sem prikazal, da lahko z uporabo knjižnic hitro in učinkovito razvijemo igro. Prav njihova uporaba pa je lahko ključ do našega uspeha.

Uspel sem razviti poučno igro, ki je namenjena otrokom. Z animacijami mi je uspelo nasmejati različne uporabnike, naročniki pa so bili z mojim delom zadovoljni. Vse težave, ki sem jih imel pri samem razvoju, sem uspel premostiti s svojim znanjem in s pomočjo forumov. Želim si, da bi igra naredila dober vtis na uporabnike in da bi pripomogla k pravilnem ravnanju z odpadki.

SEZNAM VIROV

[1] AndEngine. *AndEngine Logo*. Dostopno na: http://www.andengine.org/forums/styles/dark-grunge/theme/images/andengine_badge.png [26. 7. 2014].

[2] AndEngine. *Android Game Engine*. Dostopno na: http://www.andengine.org/blog/ [26. 7. 2014].

[3] Erin Catto. *About*. Dostopno na: http://box2d.org/about/ [24. 7. 2014].

[4] Google Inc. *Dashboards* | *Android Developers*. Dostopno na: https://developer.android.com/about/dashboards/index.html#OpenGL [24. 7. 2014].

[5] Google Inc. *Android Developers*. Dostopno na: http://developer.android.com/index.html [24. 7. 2014].

[6] Google Inc. *OpenGL ES* | *Android Developers*. Dostopno na: http://developer.android.com/guide/topics/graphics/opengl.htmlhttp://developer.android.com/index.html [24. 7. 2014].

[7] GitHub, AndEngine. *AndEngine.* Dostopno na: https://github.com/nicolasgramlich/AndEngine [26. 7. 2014].

[8] GitHub, Libgdx. *Setting up your Development Environment (Eclipse, Intellij IDEA, NetBeans)*. Dostopno na: https://github.com/libgdx/libgdx/wiki/Setting-up-your-Development-Environment-%28Eclipse%2C-Intellij-IDEA%2C-NetBeans%29 [26. 7. 2014].

[9] Interactive Software Federation of Europe. *Videogames in Europe: 2012 Consumer Study*. Dostopno na: http://www.isfe.eu/sites/isfe.eu/files/attachments/euro_summary_-_isfe_consumer_study.pdf [28. 7. 2014].

[10] Khronos Group. *OpenGL ES - The Standard for Embedded Accelerated 3D Graphics*. Dostopno na: http://www.khronos.org/opengles/ [24. 7. 2014].

[11] Lars Vogel. *Android Live Wallpaper - Tutorial*. Dostopno na: http://www.vogella.com/tutorials/AndroidLiveWallpaper/article.html [24. 7. 2014].

[12] Libgdx, Badlogic Games. *Goals and Features* Dostopno na: http://libgdx.badlogicgames.com/features.html [26. 7. 2014].

[13] Matej Mušič. *Najboljše stvari so preproste, a ne enostavne*. Dostopno na: http://mladipodjetnik.si/novice-in-dogodki/novice/najboljse-stvari-so-preproste-a-ne-enostavne [24. 7. 2014].

[14] Mateusz Myśliwiec. *AndEngine Tutorials*. Dostopno na: http://www.matim-dev.com/tutorials.html [26. 7. 2014].

[15] Schroeder J., Broyles B. AndEngine for Android Game Development Cookbook. Livery Place: Packt Publishing Ltd., 2013.

[16] Štrok, T. Igrifikacija: raba igralnih mehanik v neigralnih kontekstih. Ljubljana: Fakulteta za družbene vede, 2011. Dostopno na: http://dk.fdv.uni-lj.si/diplomska/pdfs/strok-tadej.pdf [1. 8. 2014].

[17] Unity Technologies. *Unity – Brand.* Dostopno na: http://unity3d.com/public-relations/brand [26. 7. 2014].

[18] Unity Technologies. *Unity - Multiplatform - Publish your game to over 10 platforms*. Dostopno na: https://unity3d.com/unity/multiplatform [26. 7. 2014].

[19] Okoljsko raziskovalni zavod. *Kdo smo?*. Dostopno na: http://www.orz.si/o-nas/kdo-smo.html [27. 7. 2014].

[20] Wikipedia. *Android (operating system)*. Dostopno na: http://en.wikipedia.org/wiki/Android_(operating_system) [24. 7. 2014].

[21] Wikipedia. *Eclipse (software)*. Dostopno na: http://en.wikipedia.org/wiki/Eclipse_(software) [26. 7. 2014].

[23] Wikipedia. *Entity component system*. Dostopno na: http://en.wikipedia.org/wiki/Entity_component_system [23. 7. 2014].

[24] Wikipedia. *Head-up display*. Dostopno na: http://sl.wikipedia.org/wiki/Head-up_display [23. 7. 2014].

[25] Wikipedia. *Rovio Entertainment*. Dostopno na: http://en.wikipedia.org/wiki/Rovio_Entertainment [24. 7. 2014].

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

FERI

IZJAVA O AVTORSTVU

Spodaj podpisani **Žiga Jelen**

z vpisno številko **E1041446**

sem avtor diplomskega dela z naslovom:

Razvoj mobilnih iger s pomočjo knjižnice AndEngine

(naslov diplomskega dela)

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom (naziv, ime in priimek)

doc. dr. Matej Črepinšek

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela.
- soglašam z javno objavo elektronske oblike diplomskega dela v DKUM.

V Mariboru, dne 8. 8. 2014

Podpis avtorja:

Žiga Jelen

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

FERI

IZJAVA O USTREZNOSTI ZAKLJUČNEGA DELA

Podpisani mentor :

doc. dr. Matej Črepinšek

(ime in priimek mentorja)

Izjavljam, da je študent

Ime in priimek: ***Žiga Jelen***

Št. indeksa: ***E1041446***

Na programu: **Računalništvo in informacijske tehnologije**

izdelal zaključno delo z naslovom:

Razvoj mobilnih iger s pomočjo knjižnice AndEngine

(naslov zaključnega dela v slovenskem in angleškem jeziku)

Developing Mobile Games with AndEngine

v skladu z odobreno temo zaključnega dela, Navodilih o pripravi zaključnih del in mojimi (najinimi oziroma našimi) navodili.

Preveril in pregledal sem poročilo o plagiatorstvu.

Datum in kraj:
11.8.2014, Maribor

Podpis mentorja:

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

FERI

IZJAVA O ISTOVETNOSTI TISKANE IN ELEKTRONSKE VERZIJE ZAKLJUČNEGA DELA IN OBJAVI OSEBNIH PODATKOV DIPLOMANTOV

Ime in priimek avtorja: **Žiga Jelen**

Vpisna številka: **E1041446**

Študijski program: **Računalništvo in informacijske tehnologije**

Naslov zaključnega dela: **Razvoj mobilnih iger s pomočjo knjižnice AndEngine**

Mentor: **doc. dr. Matej Črepinšek**

Podpisani **Žiga Jelen** izjavljam, da sem za potrebe arhiviranja oddal elektronsko verzijo zaključnega dela v Digitalno knjižnico Univerze v Mariboru. Zaključno delo sem izdelal-a sam-a ob pomoči mentorja. V skladu s 1. odstavkom 21. člena Zakona o avtorskih in sorodnih pravicah dovoljujem, da se zgoraj navedeno zaključno delo objavi na portalu Digitalne knjižnice Univerze v Mariboru.

Tiskana verzija zaključnega dela je istovetna z elektronsko verzijo elektronski verziji, ki sem jo oddal za objavo v Digitalno knjižnico Univerze v Mariboru.

Podpisani izjavljam, da dovoljujem objavo osebnih podatkov, vezanih na zaključek študija (ime, priimek, leto in kraj rojstva, datum zaključka študija, naslov zaključnega dela), na spletnih straneh in v publikacijah UM.

Datum in kraj: 8.8.2014, Maribor Podpis avtorja: Žiga Jelen