Razvoj video igre za Android platformu

Vlado Pajić

Sadržaj

Uvod

Industrija video igara u poslednjih nekoliko godina doživljava revoluciju. Tržište mobilnih igara koje se tek pojavilo beleži veliki rast sa očekivanjem da će u 2015. godini dostići 30% sveukupnog udela na tržistu igara. Za ovakav uspeh zaslužna je velika zastupljenost pametnih telefona i tablet uređaja kao i platformi za distribuciju aplikacija za pametne telefone.

Sa pojavom pametnih telefona pojavili su se i razvojni alati za pravljenje aplikacija za pametne telefone koji su smanjili prag ulaska na ovo tržište. Ovo je ogromna promena u odnosu na situaciju do pre nekoliko godina, jer sad programeri imaju mogućnost da sa dosta manje truda napišu igru koristeći dobro dizajnirane alate i objave igru koja će biti dostupna velikom auditorijumu.

Tema ovog rada biće orijentisana na razvoj igre za Android platformu za koju je korišćen osnovni set alata za razvoj aplikacija za Android platformu. Kao use-case biće prikazan na igri *Microorganisms* (igra se može se preuzeti sa https://play.google.com/store/apps/details?id=com.microorganisms.android).

Opis igre

Cilj igre

Microorganisms je dvodimenzionalna strateška igra u kojoj se igrač sa svojom vrstom nadmeće sa drugim vrstama kontrolisanim od strane računara. Igrač se proglašava pobednikom kada na određenoj mapi postane dominantna vrsta, odnosno kada igrač zauzme sve protivničke mikroorganizme. Pri svakoj pobedi igrač prelazi na sledeću mapu sve dok igrač ne postane dominantna vrsta na svim mapama unutar istog ciklusa, kada se smatra da je igrač završio ciklus.

Ciklus

U igri se može naći proizvoljan broj ciklusa, a svaki ciklus može sadržavati proizvoljan broj mapa. Smatra se da je igrač završio ciklus kada postane dominantna vrsta na svim mapama datog ciklusa. Ciklus postaje omogućen za igranje kada igrač završi sve zavisne cikluse. Igrač može u toku ciklusa da usavršava svoju vrstu u zamenu za bonus atribute osvojene na istom ciklusu.

Mapa

Mapa predstavlja prostor na kome se vrste međusobno bore za dominantnost. Na mapi može postojati proizvoljan broj mikroorganizama kao i igrača koji njima upravljaju. Mapa može imati nagradu u obliku bonus atributa koje igrač može da iskoristi za unapređenje karakteristika svoje vrste. Nagrada se dodeljuje samo prvi put kada igrač na mapi postane dominantna vrsta.

Igrač

Je učesnik igre. Svi mikroorganizmi koji pripadaju istom igraču čine vrstu. Na mapi se mogu naći tri tipa igrača:

- čovek – odnosno igrač koji predstavlja čoveka, korisnika igre
- računar – igrač koji je kontrolisan od strane računara
- NPC (eng. *non-player character*) igrač – igrač bez mogućnosti kontrolisanja svoje vrste

Mikroorganizam

Mikroorganizmi su osnovni i jedini element kojim igrač može upravljati. Igrač ima mogućnost kontrolisanja mikroorganizmima koji pripadaju isključivo njegovoj vrsti i njima može izdavati komande napadanja drugih mikroorganizama, koji ne moraju nužno da budu protivnički.

Atributi

Mikroorganizami poseduju atribute kojima se određuje njihova prilagođenost. Vrednosti nekih atributa mogu biti pridružene mikroorganizmu na osnovu mape ili na osnovu karakteristika vrste koje igrač unapređuje u toku ciklusa.

Populacija – određuje koliko se jedinica nalazi u sklopu mikroorganizma, odnosno određuje trenutnu *snagu* mikroorganizma. Mikroorganizmu se početna vrednost populacije dodeljuje u zavisnosti od mape i može se menjati u toku borbe.

Maksimalna populacija – određuje maksimalnu vrednost populacije koje mikroorganizam može imati pre nego što postane prenaseljen. Ukoliko mikroorganizam postane prenaseljen, jedinice počinju da

odumiru sve dok populacija ne opadne do vrednosti definisane ovim atributom. Maksimalna populacija se mikroorganizmu dodeljuje u zavisnosti od mape i ostaje nepromenjena tokom borbe.

Čvrstina – određuje izdržljivost mikroorganizma pri napadu neprijateljskih jedinica. Ovaj atribut se može unapređivati u toku ciklusa.

Brzina reprodukcije – kod mikroorganizama koji imaju sposobnost reprodukcije, određuje brzinu kojom se jedinice stvaraju. Ovaj atribut se može unapređivati u toku ciklusa.

Brzina jedinice – određuje brzinu kojom se jedince kreću. Ovaj atribut se može unapređivati u toku ciklusa.

Čvrstina jedinice – određuje izdržljivost jedinice pri napadu neprijateljskih jedinica ili jačinu pri napadu na neprijateljske mikroorganizme. Ovaj atribut se može unapređivati u toku ciklusa.

Napad

Svi mikroorganizmi imaju mogućnost napadanja. Tip napada zavisi od tipa mikroorganizma koji je inicirao napad. Prilikom napada, populacija mikroorgnizama se smanjuje za broj jedinica koji se nalazi u napadu. Broj jedinica koji će se naći u napadu opisuje se fomulom:

$$\begin{cases} 0, & kada\ populacija \leq 5 \\ populacija - 5, & kada\ 5 < populacija \leq 10 \\ \lfloor \frac{populacija}{2} \rfloor, & inače \end{cases}$$

gde je *populacija* trenutna vrednost populacije mikroorganizma koji inicira napad.

Tipovi

U igri postoje tri tipa mikroorganizama koji imaju različite uloge i karakteristike.

Virus – je jedini mikroorganizam koji ima mogućnost reprodukcije (reprodukcija je sposobnost mikroorganizma da uvećava populaciju). Prilikom napadanja neprijateljskih mikroorganizama, populacija napadnutog mikroorganizma se smanjuje za broj jedinica koji se nalazi u napadu. Prilikom napadanja mikroorganizama koji pripadaju istom igraču, populacija napadnutog mikroorganizma se povećava za broj jedinica koji se nalazi u napadu.

Parazit – je mikroorganizam koji prilikom napadanja šalje jedinice koje parazitiraju na neprijateljskom mikroorganizmu, zaustavljajući njegovu reprodukciju. Pri svakom napadu šalje se samo jedna jedinica. Ova jedinica ima *život* koji inicijalno ima vrednost koji je jednak broju jedinica koji se nalazi u napadu. Ukoliko se jedinica zakači na neprijateljski mikroorganizam koji ima sposobnost reprodukcije, jedinica će zaustaviti njegovu reprodukciju, a život te jedinice će se smanjivati.

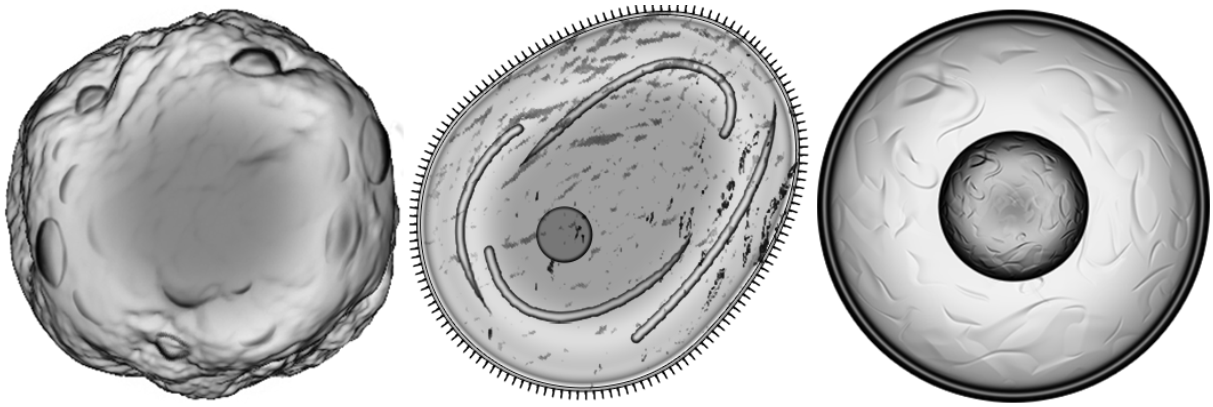
Membrana – je mikroorganizam koji prilikom napadanja šalje jedinice koje napadaju neprijateljkske parazitske jedinice. Kada stignu na napadnuti mikroorganizam, jedinice kruže oko njega sve dok se neprijateljski parazit ne nađe u neposrednoj blizini.

Predstava

Pri predstavljanju mikroorganizama, igraču se jednoznačno predstavlja tip mikroorganizma, vrstu kojoj pripada i trenutnu vrednost populacije. Tip mikroorganizma se razlikuje na osnovu slike, vrsta kojoj mikroorganizam pripada je određena bojom mikroorganizma dok je trenutna vrednost populacije prikazana na mikroorganizmu u tekstualnom obliku.

Pored ovih informacija, mikroorganizmi mogu imati različite veličine kojima se delimično određuje maksimalna populacija koju mikroorganizam može da ima, tako da će mikroorganizmi manjih dimenzija imati nižu vrednost maksimalne populacije od mikroorganizama koji imaju veće dimenzije.

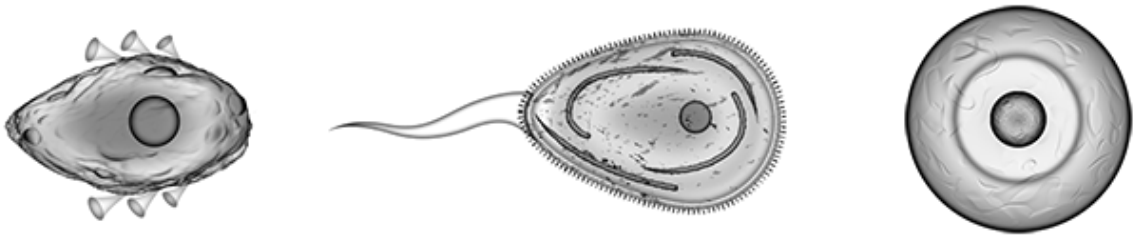
Mikroorganizmi koji pripadaju NPC igraču su uvek bele boje.



Slika 1. Izgled mikroorganizama; sa leva na desno virus, parazit, membrana

Jedinice

Jedinice se razlikuju po izgledu, koji jednoznačno određuje tip mikroorganizma koji ih je proizveo. Boja jedinice određuje vrstu kojoj pripadaju.



Slika 2. Izgled jedinica; sa leva na desno virus, parazit, membrana

Borba

Borba za dominantnost se odvija na mapi između bar dve vrste. Borba može imati dva ishoda:

- vrsta može postati dominantna ukoliko zauzme sve neprijateljske mikroorganizme i time postane jedina vrsta na mapi ili
- vrsta može biti istrebljena ukoliko svi njeni mikroorganizmi budu zauzeti od strane drugih vrsta

Da bi se zauzeo mikroorganizam koji pripada neprijateljskoj vrsti, neprijateljski mikroorganizam mora primiti napad koji će dovesti do nadjačavanja populacije koja se trenutno nalazi u mikroorganizmu. Ovakav napad je jedino moguće inicirati sa mikroorganizmom koji ima tip virusa.

Dizajn korisničkog interfejsa igre

Korisnički interfejs (eng. *user interface*) je sve ono što korisnik vidi i sa čime može da interaguje. Prilikom razvoja igara za Android platformu programeru se nameću dva pristupa za programiranje korisničkog interfejsa. Prvi pristup podrazumeva korišćenje komponenti baziranih na Androidovom *framework*-u, a drugi način je da se komponente korisničkog interfejsa prave uz pomoć biblioteke za programiranje grafike[1].

Grafički interfejs igre *Microorganisms* sadrži oba pristupa. Interfejs baziran na Androidovom *framework*u je korišćen na gotovo svim ekranima izuzev na ekranu za prikaz borbe čiji je interfejs pravljen uz pomoć biblioteke za programiranje grafike, o kojoj će biti reči u nastavku rada.

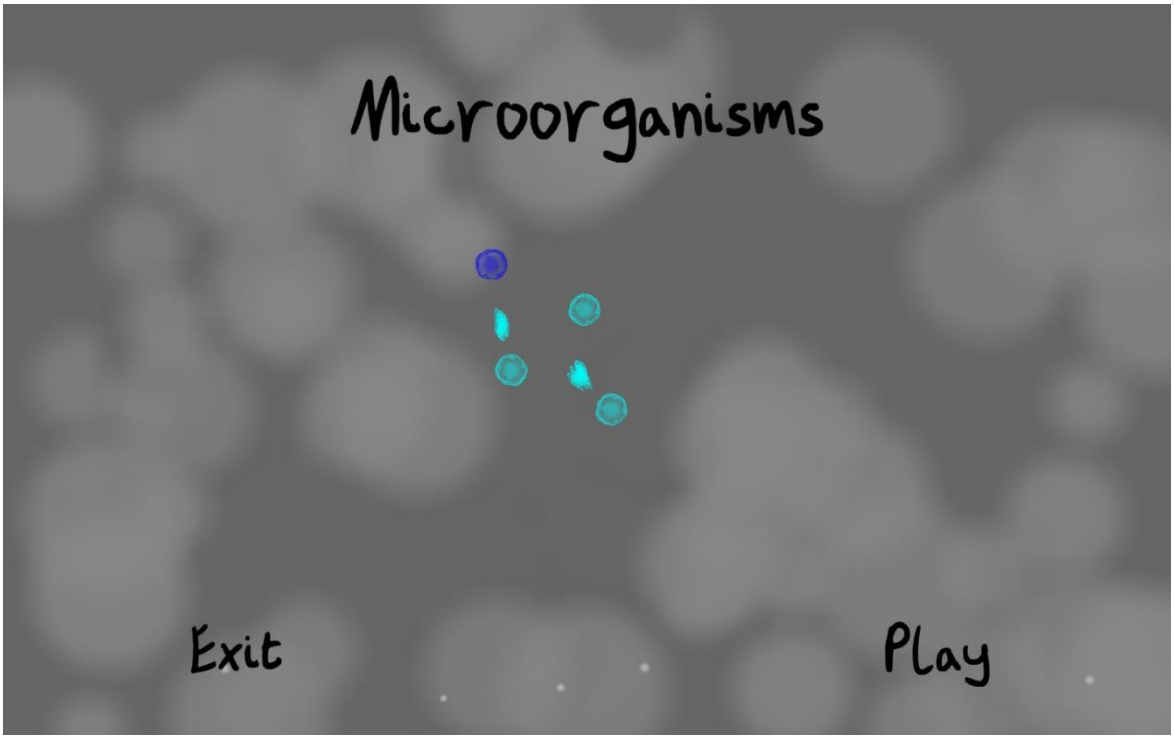
Razlog zbog koga se koriste oba načina je što Android već poseduje komponente koje se mogu prilagoditi temi igre, pa bi pisanje istih bespotrebno oduzelo vreme. Ekran za prikaz borbe je grafički daleko zahtevniji od ostalih ekrana. Na njemu se potencijalno može naći veliki broj objekata koje treba osvežavati brzo kako bi igra izgledala tečno. Ovo se ne može efikasno postići sa Androidovim komponentama pa se za programiranje korisničkog interfejsa u ovom slučaju koristi biblioteka za programiranje grafike.

Korisnički interfejs zasnovan na Android *framework*-u

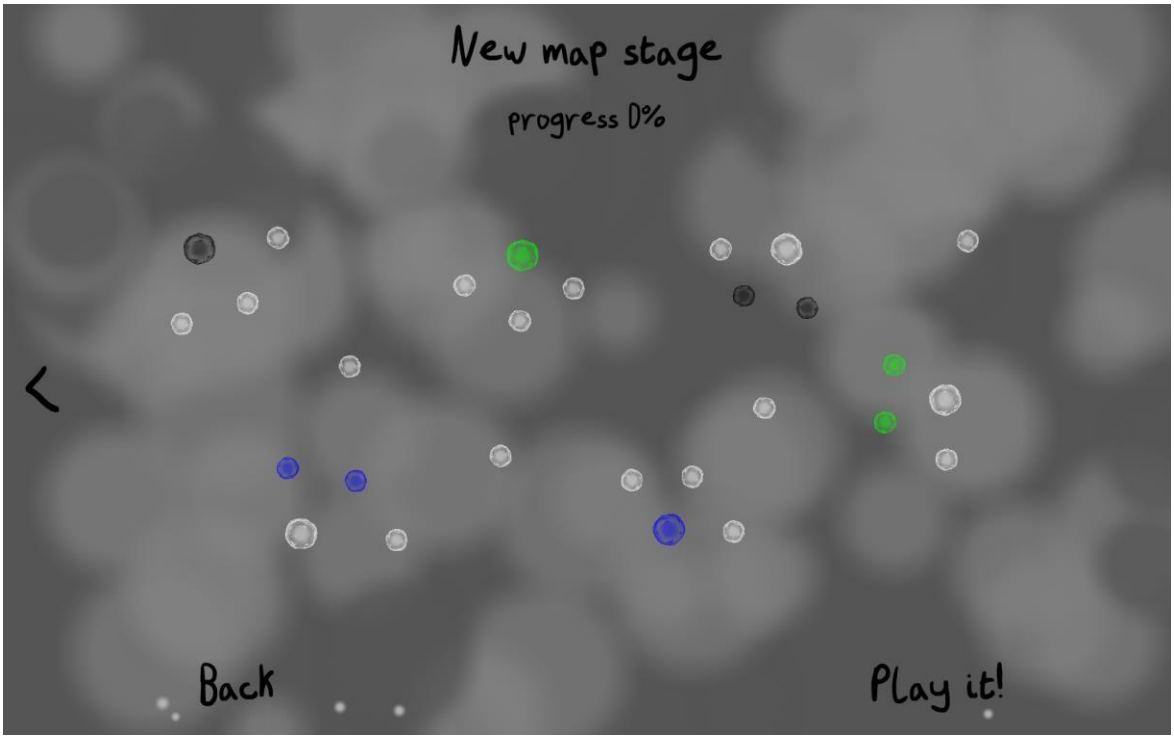
Korisnički interfejs zasnovan na Androidovom *framework*-u obuhvata sledeće ekrane: početni ekran, ekran za izbor ciklusa, ekran za izbor mape, ekran za usavršavanje vrste i ekran za izbor boje vrste.

Prilikom pokretanja igre otvara se početni ekran (slika 3) na čijem se centralnom delu nalazi prikaz žive borbe koju kontrolišu roboti. Igrač započinje igru prelazeći na ekran za izbor ciklusa (slika 4) pritiskom na dugme *Play*. Na ovom ekranu korisnik bira ciklus pritiskom na strelice koje se nalaze kod leve i desne ivice ekrana. Naziv trenutno izabranog ciklusa prikazan je na vrhu ekrana ispod koga se nalazi tekst koji pokazuje u kojoj je meri igrač prešao ciklus. U centralnom delu se nalazi prikaz mape koju korisnik treba da osvoji. Po izboru ciklusa otvara se ekran za izbor mape (slika 5). Na ovom ekranu igrač, pre nego što započne borbu na izabranoj mapi, može preći na ekran za usavršavanje vrste (slika 6). Na ovom ekranu prikazuju se mikroorganizmi koji su dostupni za usavršavanje na izabranom ciklusu. Selekcijom mikroorganizma prikazuju se trenutne vrednosti atributa i kontrole za uvećavanje i smanjivanje tih vrednosti. Sa ovog ekrana korisnik može preći na ekran za izbor boje koju će njegova vrsta imati u toku borbe.

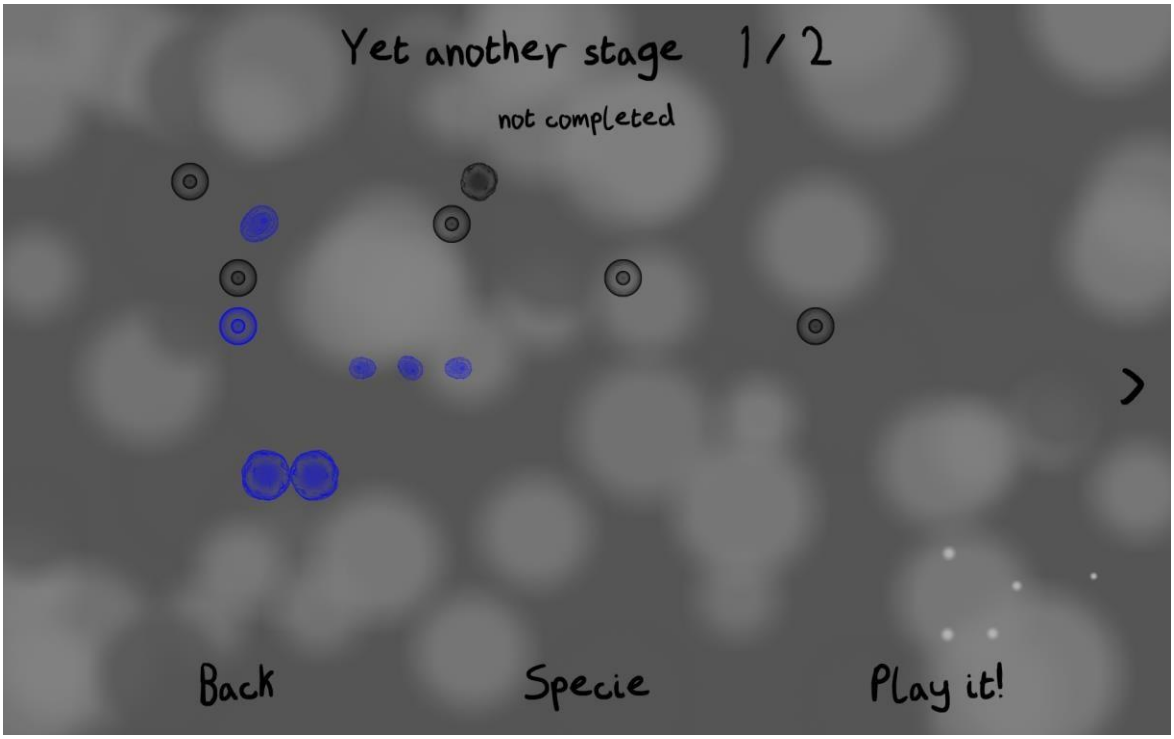
[1] Biblioteka za programiranje grafike (eng. *graphics library*) je biblioteka koja je dizajnirana da pomogne programeru u kreiranju grafičkih prikaz na ekranu monitora.



Slika 3. Izgled početnog ekrana

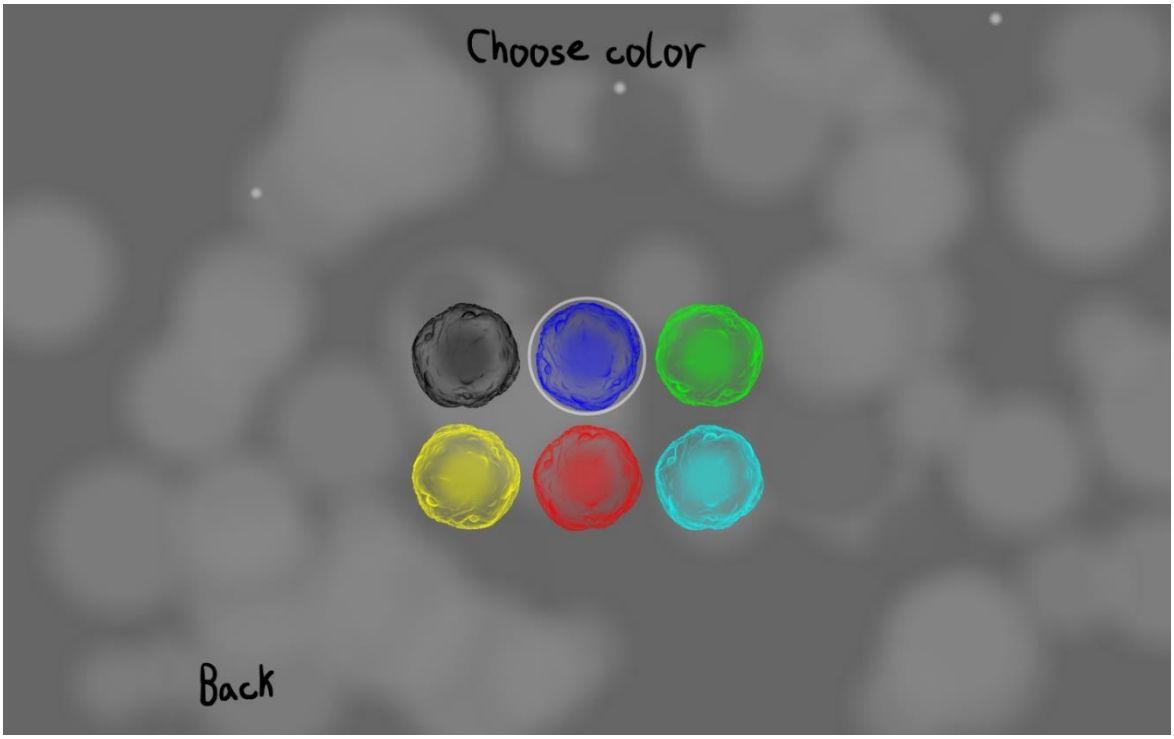


Slika 4. Izgled ekrana za biranje ciklusa



Slika 5. Izgled ekrana za biranje mape

Slika 6. Izgled ekrana za usavršavanje vrste



Slika 7. Izgled ekrana za izbor boje vrste

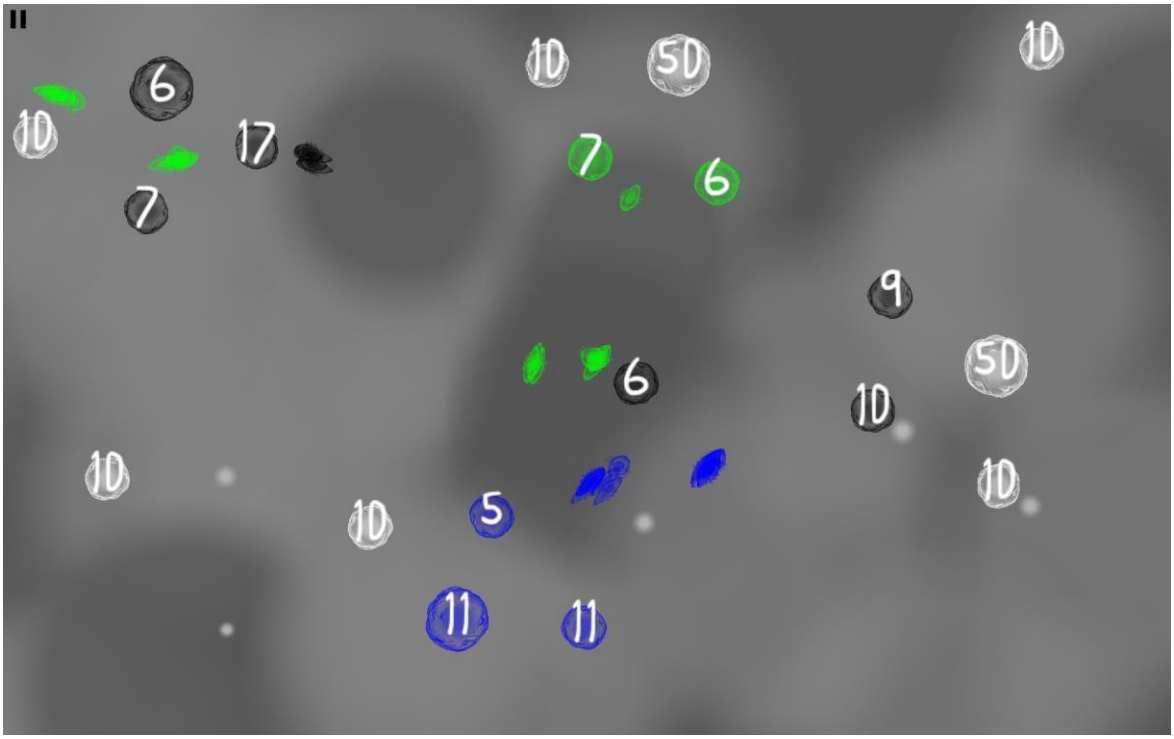
Korisnički interfejs zasnovan na biblioteci za programiranje grafike

Kao što je već pomenuto, korisnički interfejs zasnovan na biblioteci za programiranje grafike se jedino primenjuje na ekranu za prikaz borbe.

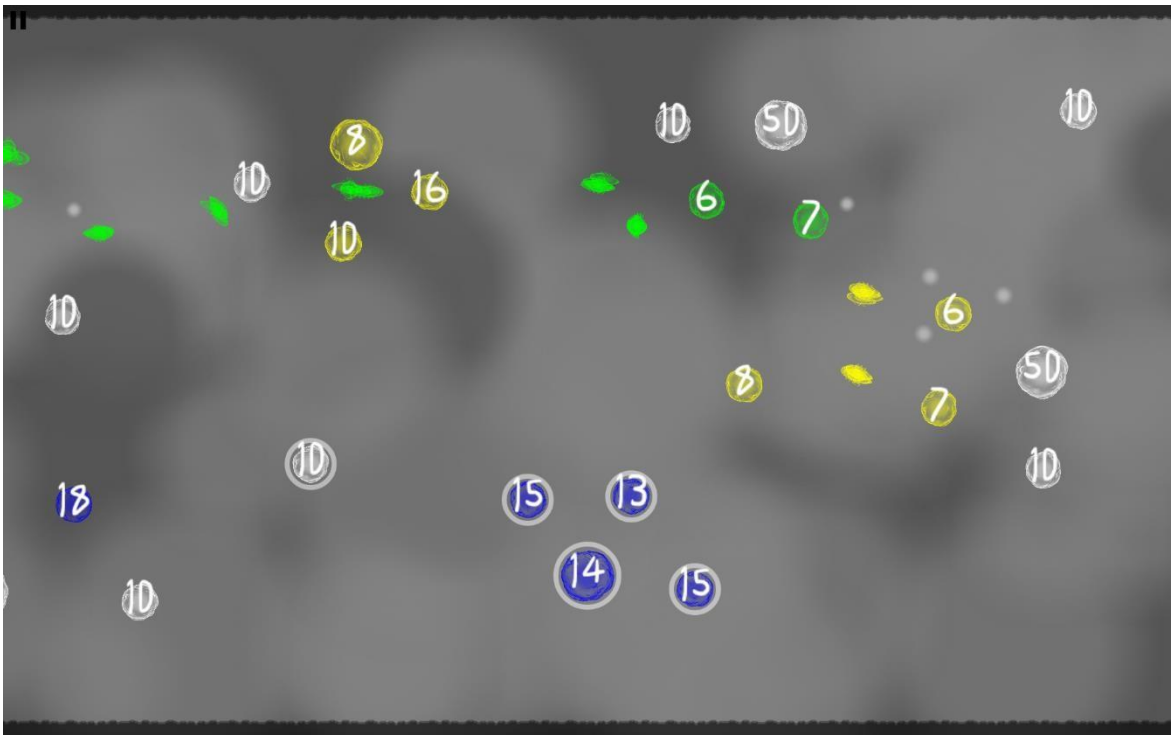
Ekran borbe predstavlja “bojno polje” na kome se igrač bori za dominantnost. Izgled ekrana borbe prikazan je na slici 8. Na ovoj slici prikazane su tri vrste čije mikroorganizme i jedinice možemo raspoznati po boji vrste kojoj pripadaju, a to su plava, zelena i crna.

Iniciranje napada

Da bi izvršio napad, igrač treba da izabere mikroorganizam/e sa kojima želi da izvrši napad i mikroorganizam koji želi da napadne. Izbor mikroorganizma se započinje dodirom na mikroorganizam sa kojim se želi izvršiti napad i zatim pomeranjem do ciljanog mikroorganizma nad kojim se izvršava napad kada se dodir otpusti. Igrač može napasti sa više mikroorganizama (slika 9) tako što će u istom pokretu, prevući dodir preko drugih mikroorganizama sa kojima želi da napadne.



Slika 8. Izgled ekrana za prikaz borbe



Slika 9. Prikaz napada sa više mikroorganizama

Programiranje grafike

Android pruža dva API-a za programiranje grafike. Prvi pristup se zasniva na crtanju 2D grafike pomoću Canvas API-a, a drugi pristup se koristi za hardverski ubrzano crtanje 2D i 3D grafike pomoću OpenGL ES API-a.

U zavisnosti od tipa igre jedan od pristupa može biti pogodniji od drugog. Recimo, za jednostavne 2D igre u kojima nije potrebna velika brzina smenjivanja slika[2] daleko je pogodniji pristup pomoću Canvas API-a zbog svoje lake upotrebe. Nasuprot ovom pristupu, OpenGL ES je jedina opcija za programiranje grafike u 3D igrama ili 2D igrama sa velikom brzinom smenjivanja slika.

Petlja igre

Petlja igre (eng. *game loop*) je, iz aspekta programiranja, osnovna komponenta svake igre. To je petlja koja je zadužena za osvežavanje igre bez obzira na akcije korisnika. Kako se ove petlje dosta razlikuju u zavisnosti od platformi za koje se igre pišu u radu će biti izložene dve petlje od kojih će jedna biti dizajnirana za iscrtavanje sa Canvas API-om dok će druga za iscrtavanje koristiti OpenGL ES API.

U najosnovnijem obliku petlja igre je *while* petlja koja se izvršava dokle god je igra aktivna. Unutar ove petlje treba da se osveži stanje igre i ponovo da se prikaže na ekranu. Imajući u obzir da brzina iteracija ove petlje diktira brzinu smenjivanja slika jedna iteracija se često naziva *frame*, a broj iteracija koji se izvrši u sekundi se naziva *frame rate*. Jedna takva petlja prikazana je u narednom primeru u kome objekat `world` predstavlja stanje igre, odnosno Svet igre.

```
while (running) {
world.update();
   world.draw();
}
```

Prikazana petlja igre u igrama u kojima je fizika Sveta modelovana u odnosu na *frame rate* neće imati stabilno napredovanje na svim uređajima. Razlog je što će uređaji sa većom procesorskom moći datu petlju izvršavati brže nego što bi to inače trebalo, dok će slabiji uređaji istu petlju izvršavati sporije. Zbog toga se igra može odvijati brže ili sporije u zavisnosti od procesorske moći uređaja.

Da bi prevazišli problem brzine uređaja, stanje igre treba da se osvežava na osnovu vremena koje je proteklo od prethodnog osveženja. Ovo vreme je u narednom primeru označeno kao `deltaTime`.

```
long lastUpdateTime = System.currentTimeMillis(); while
(running) {
   long currentTime = System.currentTimeMillis();
long deltaTime = currentTime - lastUpdateTime;
lastUpdateTime = currentTime;


   world.update(deltaTime);
world.draw();
}
```

Kod brzih uređaja broj iteracija petlje igre, odnosno *frame rate,* će višestruko prelaziti vrednosti koje su dovoljne ljudskom oku kako bi promena slika izgledala tečno. Na prvi pogled ovo može da izgleda kao prednost brzih uređaja jer će animacije izgledati bolje. Imajući na umu da pod velikim opterećenjem procesor troši više struje, a samim tim i struju baterije mobilnog uređaja, brzinu izvršavanja petlje igre

[2] Brzina smenjivanja slika ili broj slika po sekundi (eng. *FPS - frames per second*) je mera frekventnosti promene slika na animacijama, odnosno koliko se slika prikazuje u sekundi.

treba ograničiti na neku vrednost koja nije manja od brzine koja je potrebna ljudskom oku kako bi animacija izgledala tečno. Ta vrednost ne bi trebala da bude manja od 25, a veća od 60.

Programiranje grafike sa Canvas API-om

U Androidovom paketu `android.graphics` se nalazi skup klasa za rad sa Canvas API-om. Osnovna klasa je *Canvas* koja služi kao interfejs za crtanje grafike na nekoj površini. Sve metode za iscrtavanje sadržane u *Canvas*-u crtaju po bitmapi koji predstavlja sadržaj neke površine.

Crtanje na SurfaceView površini

SurfaceView je podklasa klase *View* koja obezbeđuje površinu za crtanje u okviru *View* hijerarhije. Cilj je da omogući površinu po kojoj se može crtati iz drugih niti koje nisu deo sistemske UI niti.

Iscrtavanje po *Canvas*-u nekog *SurfaceView*-a se može postići kao u sledećem primeru.

```
surfaceHolder = surfaceView.getHolder();

// …

Canvas canvas = null; try
{
   canvas = surfaceHolder.lockCanvas();
synchronized (surfaceHolder) {
      canvas.drawText("Hello world", 0, 0, null);
   } } finally {    if
(canvas != null) {
      surfaceHolder.unlockCanvasAndPost(canvas);
   }
}
```

U ovom primeru `surfaceHolder` objekat je referenca na *SurfaceHolder* od *SurfaceView*-a preko koga se može pristupiti površini za crtanje. Kada se jednom dohvati *SurfaceHolder* sva naredna crtanja se mogu inicirati preko istog objekta.

Kako bi se započelo iscrtavanje poziva se metoda *lockCanvas* koja vraća *Canvas* nad kojim se pozivaju metode za iscrtavanje po bitmapi koja predstavlja sadržaj površine *SurfaceView*-a. Po završetku iscrtavanja izmene se prikazuju na ekranu pozivom metode `unlockCanvasAndPost(Canvas)` kojoj se prosleđuje *Canvas* koji je dobijen pozivom metode *lockCanvas*.

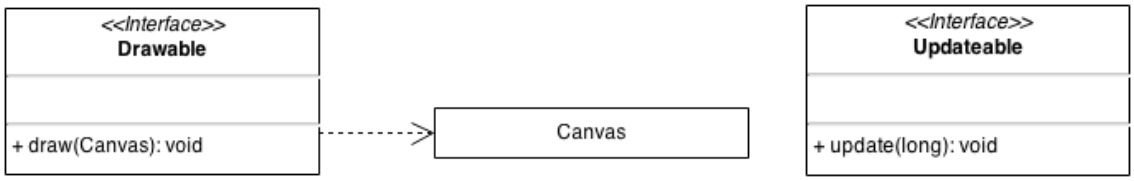
Klasa *Canvas* sadrži veliki broj metoda za iscrtavanje grafike pomoću kojih se može iscrtavati tekst, linije, pravougaonici, krugovi i druge grafičke primitive. Za 2D igre klasa *Canvas* sadrži sve što je potrebno za programiranje grafike.

Dizajn petlje igre za Canvas API

Pre nego što predstavimo dizajn petlje igre potrebno je formalno definisati interfejse koji će biti implementirani od strane Sveta igre kako bi se on osvežio i iscrtao.

Osvežavanje igre će biti ostvareno interfejsom *Updateable,* koji će pored Sveta igre biti implementiran i od strane svih objekata koje je neophodno osvežiti u svakoj iteraciji petlje igre. Ovaj intefejs poseduje samo jednu metodu `update(long)` koja kao jedini argument prima vreme u milisekundama koje je proteklo od prethodnog osveženja.

Iscrtavanje Sveta igre će biti ostvareno interfejsom *Drawable*. Ovaj interfejs poseduje samo jednu metodu `draw(Canvas)` koja kao jedini argument prima *Canvas* koji predstavlja površinu ekrana. Objekti koji žele da se iscraju na ekranu trebaju da pruže implementaciju ovog interfejs.



Slika 10. Klasni dijagrami interfejsa *Drawable* i *Updateable*

Interfejsima *Updateable* i *Drawable* se mogu lako spregnuti petlja igre i Svet igre, pa će se u narednom primeru na Svet igre odnositi preko ova dva interfejsa.

Ova petlja igre ima *frame rate* koji je ograničen na 50 *frame*-ova u sekundi, pa je perioda predviđena za jednu iteraciju petlje igre 20 milisekundi. Ukoliko se iteracija petlje igre izvrši pre završetka trenutne periode, petlja igre se uspavljuje do početka sledeće periode.

```
private final static int MAX_FPS = 50;
private final static int FRAME_PERIOD = 1000 / MAX_FPS;

private SurfaceHolder surfaceHolder; private
boolean running;

 private Updateable updateable;
 private Drawable drawable;

 public final Object updateLock = new Object();
  @Override public
 void run() {
 Canvas canvas;
 long sleepTime;
 long beginTime;
 long currentTime;
    long lastUpdateTime = System.currentTimeMillis();

    while (running) {
 canvas = null;
       beginTime = System.currentTimeMillis();
       try
{
         synchronized (updateLock) {
             // Update game
            currentTime     =     System.currentTimeMillis();
updateable.update(currentTime       -       lastUpdateTime);
lastUpdateTime = currentTime;

             // Draw game
             canvas = surfaceHolder.lockCanvas();
 synchronized (surfaceHolder) {
 drawable.draw(canvas);
             }
```

```
            }

            sleepTime = FRAME_PERIOD - (System.currentTimeMillis() - beginTime);
    if (sleepTime > 0) {
                try {
                    Thread.sleep(sleepTime);
                } catch (InterruptedException e) {
                }
            }       } finally {
    if (canvas != null) {
                surfaceHolder.unlockCanvasAndPost(canvas);
            }
        }
    }
}
```

Programiranje grafike sa OpenGL ES API-om

OpenGL je industrijski standard za programiranje 2D i 3D grafike. OpenGL ES je podskup OpenGL API-a dizajniran za ugrađene sisteme (eng. *embedded systems*) kao što su pametni telefoni ili tablet uređaji.

Android platforma ima ugrađenu podršku za OpenGL ES API. Verzija 1.0 OpenGL ES API-a je prisutna od verzije 1.0 Androida, dok novije verzije Androida podržavaju i novije verzije OpenGL ES API-a. Ovaj rad će se baviti isključivo OpenGL ES verzijom 1.0, koji je i korišćen u igri *Microorganisms*.

Crtanje na GLSurfaceView površini

Analogno Canvas API-u, Android *framework* takođe poseduje površinu po kojoj se može crtati pozivima OpenGL ES API-a. Klasa koja predstavlja ovu površinu zove se *GLSurfaceView*.

Da bi se iscrtali objekti po *GLSurfaceView* potrebno je napisati implementaciju interfejsa *GLSurfaceView.Renderer* i dodati ga na *GLSurfaceView*. Ovaj interfejs poseduje definicije metoda koje su neophodne za iscrtavanje grafike. Definicije tih metoda su sledeće:

- `onSurfaceCreated(GL10 gl, EGLConfig config)` – Metoda koja će biti pozvana samo jednom, pri kreiranju *GLSurfaceView* površine. Ova metoda je namenjena za inicijalizaciju OpenGL ES okruženja.
- `onSurfaceChanged(GL10 gl, int width, int height)` – Metoda koja će biti pozvana pri promeni dimenzija površine.
- `onDrawFrame(GL10 gl)` – Metoda koja će biti pozvana svaki put da bi se sadržaj prikazao na ekranu.

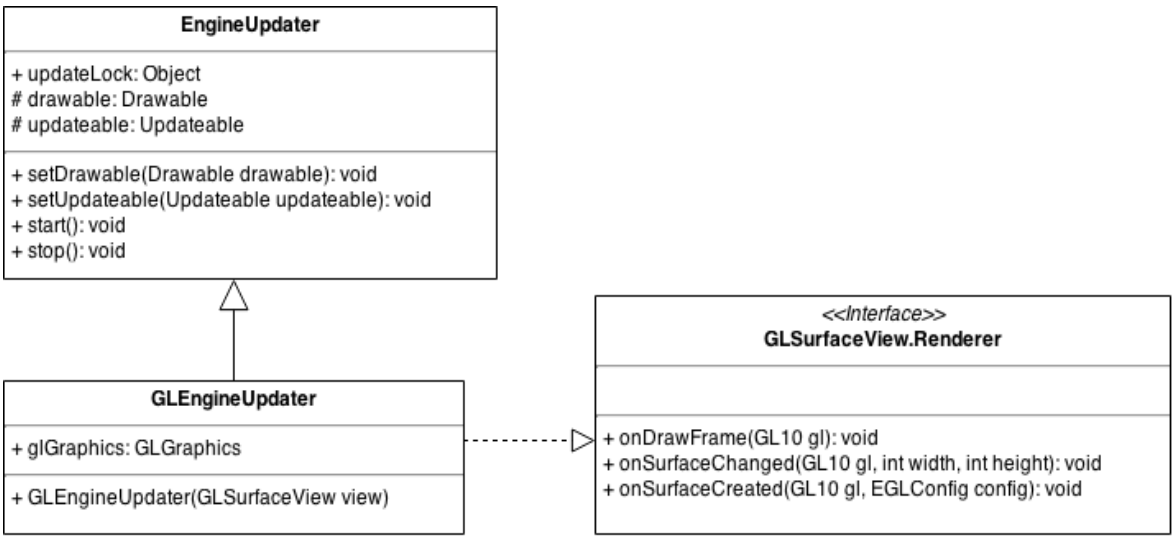
Drugim rečima, *GLSurfaceView* interno poseduje mehanizam kojim se sadržaj ekrana osvežava. Detalji ove implementacije su sakrivene, a jedino što ostaje na korisniku je da napiše implementaciju *GLSurfaceView.Renderer*, odnosno da opiše šta će biti prikazano na ekranu. Ovaj mehanizam je upravo ono što je kod Canvas API-a radila petlja igre. Metodu *onDrawFrame* možemo posmatrati kao deo koda koji je potrebno izvršiti unutar jedne iteracije petlje igre.

GLSurfaceView ima dva načina osvežavanja sadržaja površine. Prvi način je renderovanje samo u slučaju kada je površina prljava. U ovom slučaju od *GLSurfaceView* se mora eksplicino zatražiti da osveži

površinu pozivom metode *requestRender*. Drugi način je kontinuirano osvežavanje površine. Kod ovog načina *GLSurfaceView* će sam osvežavati površinu brzinom približnom 60 FPS[3].

Ova dva načina osvežavanja ekrana nameću dve varijacije u realizaciji glavne niti. U prvoj realizaciji, kod renderovanja u slučaju kada je površina prljava, potreban je dodatni mehanizam koji bi u kontinualnim vremenskim intervalima pozivao metodu *requestRender* kao bi se igra osvežila. Ovaj mehanizam je dovoljan ako igra ne zahteva veliki FPS, dok bi u protivnom pisanje ovakvog mehanizma bilo redundantno sa mehanizmom koji je interno implementiran sa metodom kontinualnog osvežavanja.

U realizaciji igre *Microorganisms* klasa koja implementira interfejs *GLSurfaceView.Renderer* naziva se *GLEngineUpdater*. Ova klasa osvežava i iscrtava svet igre u pozivima metode *onDrawFrame*.



Slika 11. Klasni dijagram *GLEngineUpdater*-a

Biblioteka za programiranje 2D grafike sa OpenGL ES-om

Programiranje grafike sa OpenGL ES API-em je daleko komplikovanije od Canvas API-a. Kako bi se olakšalo programiranje grafike za potrebe igre razvijena je biblioteka koji nudi set alata za jednostavan razvoj 2D igara. Ovaj biblioteka je nazvan TinyLib.

Iscrtavanje *sprite*-a

Iscrtavanje *sprite*[4]-a je elementarna grafička operacija u svim 2D igrama. Sa ovom operacijom se mogu napraviti gotovo sve 2D igre. Za razliku od Canvas API-a, OpenGL ES API ne poseduje jednostavan mehanizam za iscrtavanja *sprite*-ova.

U ovom delu rada biće opisan najjednostavniji način na koji se *sprite* može iscrtati korišćenjem OpenGL ES API-a. Zatim će biti predstavljen mehanizam iscrtavanja u TinyLib biblioteci.

Trouglovi

Trouglovi su osnovna primitiva iscrtavanja u OpenGL ES-u. Iscrtavanje kompleksnijih oblika svodi se na kombinovanje više trouglova. Tako da ako želimo da nacrtamo pravougaonik biće potrebna bar dva

[3] Brzina osvežavanja može zavisiti od hardvera i od verzije Androida.

[4] *Sprite* je dvodimenzionalna slika ili animacija koja je deo scene.

trougla. Kako bi lakše pojmili, OpenGL ES možemo posmatrati kao mehanizam za iscrtavanje trouglova, odnosno oblika sastavljenih od trouglova.

Da bi OpenGL ES znao kakav trougao treba da nacrta potrebno je definisati temena trougla. Osnovni atribut temena je njegova pozicija u trodimenzionalnom prostoru. Ove pozicije definišemo kao niz uzastopnih vrednosti x, y i z komponenti koordinata trodimenzionalnog sistema za svaku poziciju temena trougla. Pošto se temena definišu možemo izvršiti poziv OpenGL ES API-a da nacrta trougao sa datim temenima.

Ovde se stvari malo komplikuju kod OpenGL ES API-a pisanim za programiski jezik Java, jer je OpenGL ES pisan u programskom jeziku C, pa je API pisan u Javi zapravo poziv odgovarajućih C funkcija. Imajući ovo na umu, nije moguće definisati standardni niz u Javi i zatim taj niz proslediti pri pozivu OpenGL ES API-a. Razlog je što će standardnim definisanjem niza u Javi memorija biti alocirana na hipu virtuelne mašine, a ne u nativnom hipu kome može da se pristupi iz nativnog koda. Ovaj problem se može lako rešiti uz pomoć Javinih NIO buffer-a, koji se koriste za alociranje memorijskih blokova.

U sledećem primeru prikazano je kako se definišu temena sa pozicijama u tačkama (0,0), (10,0) i (0, 10).

```
float[] jVertices = new float[] { 0.0f, 0.0f, 10.0f, 0.0f, 0.0f, 10.0f };
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(jVertices.length * Float.SIZE /
8); byteBuffer.order(ByteOrder.nativeOrder()); FloatBuffer vertices =
byteBuffer.asFloatBuffer(); vertices.put(jVertices); vertices.flip();
```

Kako je *z* komponenta izostavljena OpenGL ES će je tretirati kao da ima vrednost 0.

Zatim, ovako definisana temena trougla možemo iscrtati pozivom sledećih funkcija.

```
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY); gl.glVertexPointer(2,
GL10.GL_FLOAT, 0, vertices); gl.glDrawArrays(GL10.GL_TRIANGLES, 0,
3);
```

Gde poziv funkcije *glEnableClientState* govori OpenGL ES-u da temena sadrže pozicije, zatim sa pozivom *glVertexPointer* naznačavamo OpenGL ES-u gde se nalaze temena i konačno *glDrawArrays* iscrtava trouglove.

Pravougaonik

Već se može zaključiti da se iscrtavanje pravougaonika može postići iscrtavanjem dva trougla koja će imati dva temena na istim pozicijama. Ovo je svakako tačno, ali ne i najoptimalnije rešenje koje se može postići sa OpenGL ES-om.

Da bi iscrtali dva trougla potrebno je definisati temena za svaki trougao, što je ukupno šest temena. Od ovih šest temena postoje četiri temena koja imaju različite pozicije. OpenGL ES omogućava da se ista temena ponovo upotrebe, umesto da se njihovo definisanje ponavlja.

Način na koji OpenGL ES ovo omogućava je indeksiranje temena. Što znači da bi prilikom iscrtavanja, pored definicija temena, trebalo biti naznačeno koja temena pripadaju kom trouglu.

```
// Define indices
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(6 * Short.SIZE / 8);
byteBuffer.order(ByteOrder.nativeOrder()); ShortBuffer
indices = byteBuffer.asShortBuffer(); indices.put(new
short[] { 0, 1, 2, 2, 3, 0 }); indices.flip();

// Draw using indices
```

```
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY); gl.glVertexPointer(2,
GL10.GL_FLOAT, 0, vertices); gl.glDrawElements(GL10.GL_TRIANGLES, 6,
GL10.GL_UNSIGNED_SHORT, indices);
```

Dodavanje teksture oblicima

Do sada su temena bila posmatrana sa samo jedinim atributom – pozicijom u trodimenzionalnom sistemu. Ali pored ovog atributa temenima se mogu pridružiti i drugi atributi kao što je boja ili koordinate teksture. Za sada, najviše će biti reči o teksturama.

Koordinatni sistem tekstura se predstavlja preko normalizovanih koordinata. Što znači da komponente koordinata mogu imati vrednosti između 0 i 1 (uključivo), gde 1 odgovara širini, odnosno visini teksture. Koordinatni početak se nalazi u gornjem levom uglu teksture, pa će donji desni ugao teksture imati vrednost (1,1).

Kako bi se iscrtao oblik sa teksturom, potrebno je specificirati OpenGL ES-u koju teksturu treba da koristi i definiciji temena pridružiti koordinate teksture.

Specificiranje koju teksturu treba dodeliti obliku radi se pomoću njenog *handler*-a, odnosno *id*-a. Handler teksture se generiše svaki put kada se uvodi nova tekstura. Tom prilikom se tekstura učitava u VRAM i

```
int textureIds[] = new int[1]; gl.glGenTextures(1,
textureIds, 0);
gl.glBindTexture(GL10.GL_TEXTURE_2D, textureIds[0]);
// Load bitmap to VRAM
GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
```

automatski se veže za generisani handler.

Od ovog trenutka ovoj teksturi se obraća preko njenog *handler*-

a. Za jednostavno kreiranje teksture, Android *framework* poseduje alat koji na osnovu bitmape pravi i učitava teksturu u VRAM.

Tekstura, koja je prethodno učitana na ovaj način, se može koristiti prilikom iscrtavanja na sledeći način.

```
gl.glBindTexture(GL10.GL_TEXTURE_2D, textureIds[0]);
```

Serijsko iscrtavanje

Obično je potrebno iscrtati više istih *sprite*-ova na različitim pozicijama, različitih dimenzija ili rotacija. Jedno rešenje za ovakav scenario je primeniti niz 2D transformacije nad OpenGL ES-ovom *model-view* matricom neposredno pre iscrtavanja *sprite*-a.

Model-view matrica se može posmatrati kao stanje OpenGL ES-a. Ova stanja će biti primenjena nad temenima trouglova prilikom iscrtavanja. Model-view matrica je zadužena za održavanje dva stanja, stanje modela koja primenjuje transformacije kao što su translacija, rotiranje i skaliranje nad temenima trougla i view matrica koja primenjuje transformacije nad kamerom.

U narednom primeru dat je deo koda koji podešava translaciju i skaliranje sprite-a neposredno pre iscrtavanja, takođe učitava jediničnu matricu kako bi se resetovalo trenutno stanje koje je možda menjano prilikom nekog ranijeg iscrtavanja.

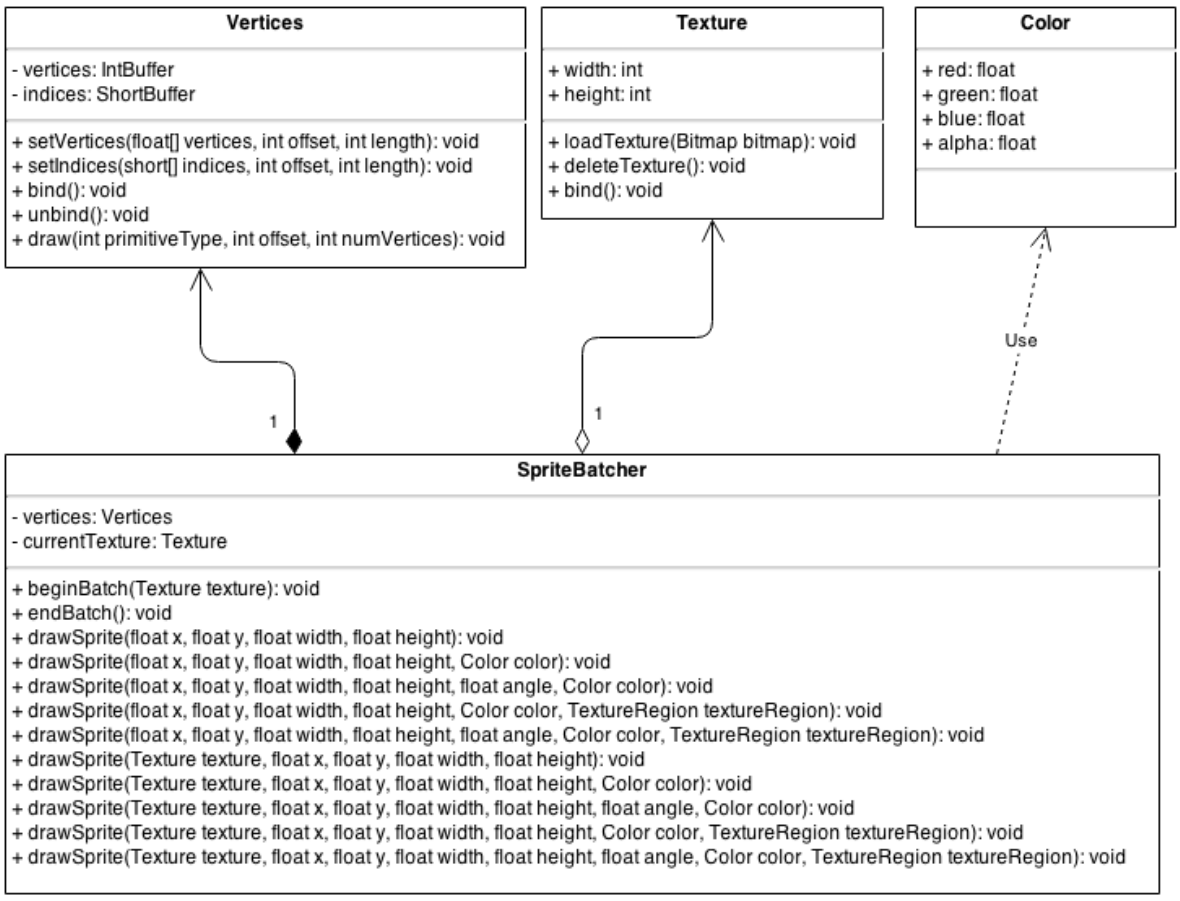
```
// Apply 2D transformations to the model view
matrix gl.glLoadIdentity(); gl.glTranslatef(posX,
posY, 0f); gl.glScalef(width, height, 1.0f);
```

```
// Draw sprite ...
```

Mana ovog pristupa je što zahteva veliki broj JNI poziva, odnosno poziva OpenGL ES API-a. Pored toga, *model-view* matrica se transformiše prilikom svakog iscrtavanja.

Opisani problem se može prevazići tehnikom serijskog iscrtavanja *sprite*-ova (eng. *sprite batching*) kojom se u jednom potezu iscrtava veći broj sprite-ova. Kod ove tehnike se prilikom iscrtavanja samo konstruišu temena trouglova koja se zatim čuvaju u *buffer*-u. U ovom trenutku nema poziva OpenGL ES API-a. Trouglovi će biti iscrtani tek onda kada se serija iscrtavanja završi i tom prilikom se izvršava poziv OpenGL ES API-a da se konstruisana temena prikažu na ekranu.

Tehnika serijskog iscrtavanja je implementirana u TinyLib biblioteci sa klasom *SpriteBatcher*. Ova klasa se oslanja na klasu *Texture* koja predstavlja teksturu *sprite*-a koji se iscrtava kao i na klasu *Vertices* koja ja zadužena za rad sa temenima. Klasni dijagram koji opisuje zavisnost ovih klasa dat je na slici 12.



Slika 12. Klasni dijagram *SpriteBatcher* klase

Kod klase *SpriteBatcher* serija iscrtavanja počinje pozivom metode `beginBatch()` kojoj se prosleđuje tekstura koja će biti primenjena na temenima kada se serija iscrtavanja završi. Serija iscrtavanja se završava pozivom metode `endBatch()`. Takođe, serija iscrtavanja se implicitno može završiti u dva slučaja: kada se prekorači maksimalan broj *sprite*-ova koji se iscrtavaju u jednoj seriji ili kada dođe do promene teksture koja se iscrtava.

Klasa *SpriteBatcher* ima veliki broj *draw* metoda sa kojima se mogu iscrtavati *sprite*-ovi nad kojima će biti primenjene različite transformacije. U svim slučajevima *draw* metode kreiraju četiri temena koja se smeštaju u bafer. Ova četiri temena predstavljaju pravougaonik sastavljen od dva trougla. Prilikom završetka serije iscrtavanja, *SpriteBatcher* vezuje trenutnu teksturu i prosleđuje temena *Verties* klasi da bi ih iscrtao.

U primeru koji sledi je prikazan način na koji se iscrtavaju jedinice u igri *Microorganisms*.

```
spriteBatcher.beginBatch(texture);
for (int i = 0, len = attackUnits.size(); i < len; i++) {
Unit unit = attackUnits.get(i);
   spriteBatcher.drawSprite(unit.center.x, unit.center.y,
         unit.size.width, unit.size.height, unit.rotation * MathUtils.degRad, color);
}
```

U ovom primeru serija iscrtavanja se eksplicitno pokreće za datu teksturu, a zatim se vrši iscrtavanje spriteova. Eksplicitno pokretanje serije iscrtavanja dolazi do izražaja kada ima više sprite-ova koja treba iscrtati. Ali, ukoliko postoji samo jedan sprite koji treba iscrtati, može se koristiti *draw* metoda koja kao prvi argument prima teksturu. U ovom slučaju serija iscrtavanja se implicitno pokreće za specificiranu teksturu, a zatim se poziva *draw* metoda koja odgovara potpisu pozvane metode samo bez prvog argumenta. Ovaj princip se koristi za iscrtavanje mikroorganizama i dat je u sledećem primeru.

```
spriteBatcher.drawSprite(cellTexture, cell.center.x, cell.center.y, cell.size.width,
cell.size.height, cell.rotation * MathUtils.degRad, color);
```

Važno je napomenuti da u oba primera serija iscrtavanja nije eksplicitno okončana od strane programera. Razlog je to što, ukoliko se nad istim *SpriteBatcher* objektom ponovo pokrene serija iscrtavanja za istu teksturu, neće doći do okončanja trenutne serije pa će se tako smanjiti ukupan broj poziva OpenGL ES API-
a. Ipak, eksplicitno okončanje će biti neophodno izvršiti neposredno pre završetka svih iscrtavanja, jer se u protivnom može desiti da nešto ostane u buffer-u *SpriteBatcher* objekta.

Ispisivanje teksta

Ispisivanje proizvoljnog teksta je potrebno u gotovo svim igrama. Kako OpenGL ES ne poseduje mehanizam ispisivanja teksta u TinyLib biblioteci je napisana klasa sa kojom se proizvoljan tekst može ispisivati korišćenjem OpenGL ES API-a.

TextPainter je klasa kojom se ispisivanje teksta postiže serijskim iscrtavanjem *sprite*-ova na kojima se nalaze znakovi teksta. Zbog toga je pre početka ispisivanja potrebno kreirati teksturu koja će sadržavati sve znakove alfabeta pozivom metode *createTexture*. Kako bi *TextPainter* podržavao proizvoljan font, odnostno stil teksta, prilikom kreiranja teksture potrebno je proslediti i objekat klase `android.graphics.Paint` koji će se koristiti pri pravljenju teksture. Sa ovako kreiranim *TextPainter*-om se može iscrtavati proizvoljan tekst sastavljen isključivo od alfabeta definisanog pri pozivu metode *createTexture*.

U igri *Microorganisms* se prilikom iscrtavanja mikroorganizma ispisuje i tekst preko mikroorganizma koji označava njegovu trenutnu populaciju. Kôd za ispisivanje teksta je dat u narednom primeru.

```
String text = texts[(int) cell.population];
textPainter.drawText(textBatcher, text, cell.center.x, cell.center.y, textColor);
```

U datom primeru nalazi se nalazi nekoliko stvari koje treba napomenuti. Treba primetiti da tekst koji se iscrtava nalazi u unapred definisanom nizu koji sadrži sve stringovne vrednosti populacije. Ovo je

trivijalna optimizacija sa ciljem da se smanji broj dinamičkih alokacija koje se dešavaju unutar *update* i *draw* metoda. Drugo, u *engine*-u igre *Microorganisms* se koriste dva nezavisna *SpriteBatcher* objekta. Jedan od njih se koristi za generalno iscrtavanje *sprite*-ova (iscrtavanje jedinica i mikroorganizama), a drugi se koristi za iscrtavanje teksta. Ukoliko bi se koristio isti *SpriteBatcher* objekat, serija iscrtavanje bi se završavala češće usled česte promene teksture. Na kraju treba napomenuti da *drawText* metoda ispisuje tekst koji je vertikalno i horizontalno centriran u tački prosleđenoj pri pozivu metode.

Logika igre

Implementacija igre *Microorganisms* podeljena je u dva paketa. U prvom paketu `com.microorganisms.game` se nalazi implementacija *engine* borbe igre. U ovom paketu se nalazi osnovna logika igre koju pokreće e*ngine* borbe, pa je sa aspekta logike igre ovo osnovni paket. U drugom paketu `com.microorganisms.android` se nalazi implementacija korisničkog interfejsa igre u kome se *engine* borbe igre koristi za simulaciju borbe.

Engine borbe igre

U igri *Microorganisms*, *engine* borbe igre, ili samo *engine*, je ujedno i svet igre. Zbog ovoga je logiku borbe, vrlo lako moguće ubaciti i pokrenuti na skoro svakom mestu. Ova prednost *engine*-a se u igri *Microorganisms* koristi na početnom ekranu gde se nalazi živ prikaz borbe između igrača kontolisanim od strane uređaja. Veoma sličana primena *engine*-a nalazi se na ekranu borbe kada igrač kontroliše svojom vrstom.

Klasa koja predstavlja *engine* borbe igre zove se *BattleEngine*. *BattleEngine* je u osnovi veoma jednostavan. On poseduje listu objekata koje treba iscrtavati i listu objekata koje treba osvežavati. U toku osvežavanja, objekat može menjati svoje stanje što će rezultovati promenom prikaza na ekranu prilikom narednog iscrtavanja. Ovim je logika igre implementirana u klasama čiji se objekti nalaze u navedenim listama.

Kako bi se instancirala klasa *BattleEngine* potrebno je pozvati konstruktor kome se prosleđuju objekti klasa *BattleEngineHolder*, *EngineUpdater*, *BattleInfo*, *Background*, *SoundManager* i *AssetsProvider*.

BattleEngineHolder – Interfejs kojim se klasa, odnosno objekat koji koristi *BattleEngine*, skriva od njega. Drugim rečima, ovim interfejsom se *BattleEngine* objektu pružaju usluge okruženja potrebne u toku izvršavanja *engine*-a.

EngineUpdater - Objekat klase *EngineUpdater* je interfejs između *BattleEngine*-a i mehanizma koji inicira njegovo iscrtavanje i osvežavanje. U slučaju igre *Microorganisms* konkretna implementacija je klasa *GLEngineUpdater* čiji je klasni dijagram prethodno prikazan na slici 11.

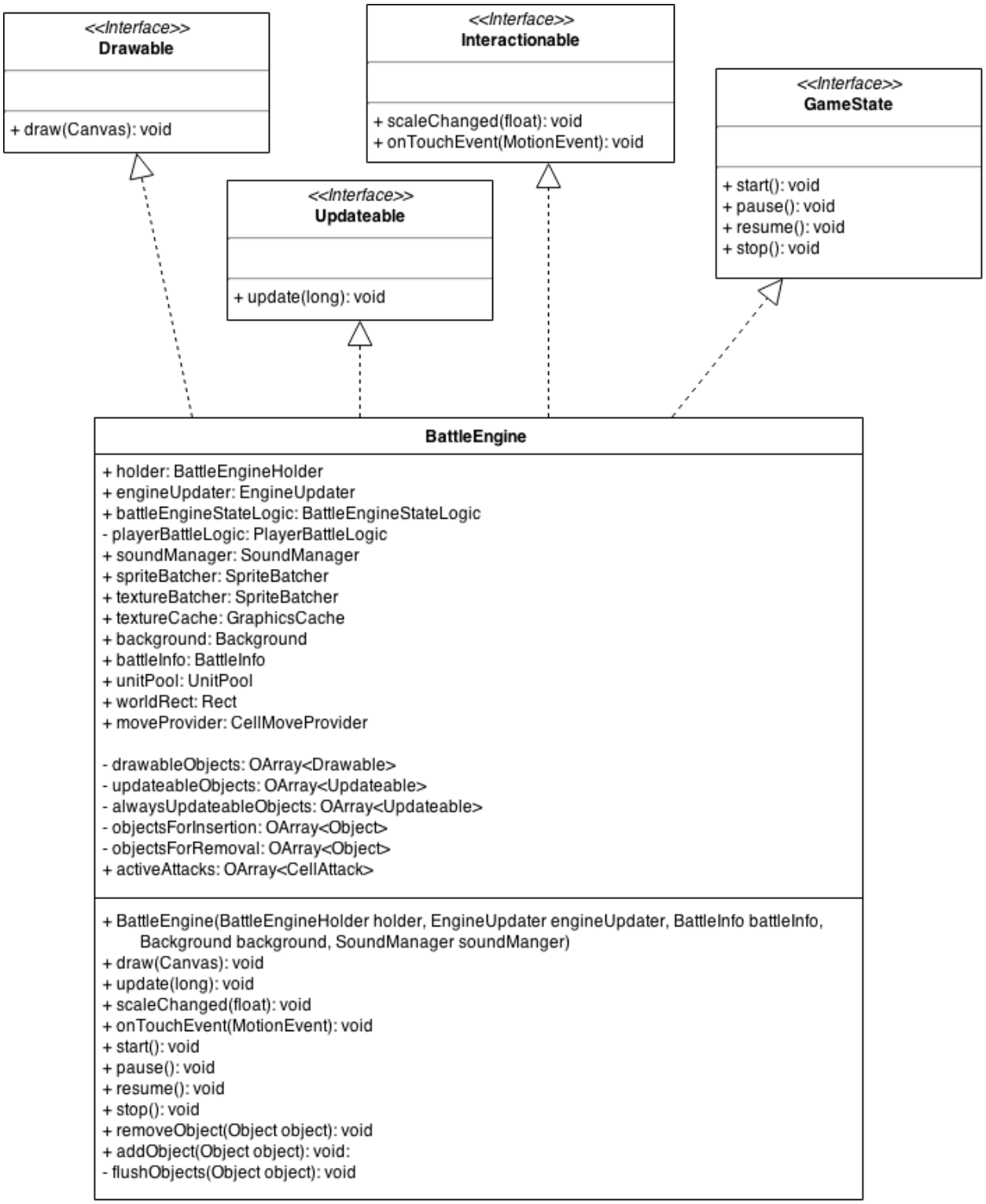
BattleInfo – Objekat klase *BattleInfo* sadrži podatke o borbi koju treba simulirati sa *BattleEngine*-om. Ove podatke čine podaci o igračima i mikroorganizmima, trenutnom igraču, maksimalnim vrednostima atributa mikroorganizama i dimenziji mape i ekrana.

Background – Objekat klase *Background* predstavlja pozadinu koja se prva iscrtava.

SoundManager – Objekat klase *SoundManager* predstavlja menadžer zvukova u igri.

AssetsProvider – Objekat koji implementira interfjes *AssetsProvider* se koristi za snadbevanje *BattleEngine*-a resursima, kao što su bitne mape (eng. *bitmap*) i fontovi.

Ovako kreiran *engine* potrebno je pokrenuti pozivom metode *start* kada počinje simulacija borbe. Pozivom metode *start*, *engine* borbe izvršava inicijalizaciju listi objekata koje treba iscrtavati i/ili osvežavati tokom izvršenja. Potom, *engine* prosleđuje sebe *EngineUpdater* objektu kako bi bio pozivan svaki put kada bi mehanizam *EngineUpdater*-a inicirao iscrtavanje i osvežavanje.



Slika 13. Klasni dijagram *BattleEngine* klase

Logika igre sadržana u *Updateable* i *Drawable* objektima

Kao što je već pomenuto, sveukupna logika igre je sadržana u listi objekata koja se periodično osveža i iscrtava od strane *engine*-a. Konkretno, objekti koji se periodično osvežavaju od strane *engine*-a se nalaze u listi `updateableObjects`, čiji elementi implementiraju interfejs *Updateable*. Dok se objekti koji se periodično iscrtavaju nalaze u listi `drawableObjects`, čiji elementi implementiraju interfejs *Drawable*. U ovom delu rada će biti opisane klase čiji se objekti nalaze, odnosno mogu naći, u ovim listama u toku izvršenja *engine*-a.

Interakcija igrača sa objektima igre

Klasa *PlayerBattleLogic* enkapsulira logiku koju igrač može izvršavati pri interakciji sa igrom. Njena podklasa koja implementira logiku koju može izvršavati čovek, sadržana je u klasi *HumanBattleLogic*. Klasa *HumanBattleLogic* implementira logiku za selektovanje mikroorganizama (u specijalnom slučaju kreiranje napada), podešavanje kamere (zuma i translacije) i logiku za detekciju završetka igre. Prilikom pokretanja *engine*-a se proverava da li u *BattleInfo* objektu postoji igrač koga kontroliše čovek u kom slučaju se kreira logika koju implementira klasa *HumanBattleLogic*, dok se u protivnom kreira logika koju implementira klasa *BotBattleLogic*. Ovom objektu se kasnije u toku izvršavanja *engine*-a delegiraju pozivi metoda interfejsa *Interactionable* koji opisuje događaje koje korisnik generši prilikom interakcije sa ekranom.

Logika mikroorganizama

Klasa *BaseCell* [5] je osnovna klasa koja implementira logiku svih mikroorganizama. Objekti ove klase se kreiraju prilikom inicijalizacije *BattleInfo* objekta na osnovu njihovih deskriptora, a zatim se prilikom startovanja *engine*-a dodaju u `updateableObjects` i `drawableObjects` liste.

Klasa *BaseCell* delegira pozive metoda interfejsa *Drawable* na objekat klase *CellPainter* koji je zadužen za iscrtavanje mikroorganizma.

Klasa *BaseCell* interno poseduje listu objekata koji implementiraju logiku ponašanja mikroorganizma. Tokom faze osvežavanja klasa *BaseCell* delegira pozive *update* metode interfejsa *Updateable* na elemente ove liste. Klase čiji se objekti mogu naći u ovoj listi su:

IncreasePopulationAffect – Implementira logiku reprodukcije mikroorganizma.

MaxPopulationExceededAffect – Implementira logiku kojom se populacija mikroorganizma smanjuje ukoliko populacija nadmaši maksimalnu vrednost koju mikroorganizam može da ima.

SizeChangeAnimation – Animacija kojom se uvećavaju ili smanjuju dimenzije mikroorganizma. Ova animacija se pokreće da uveća mikroorganizam ukoliko trenutna vrednost populacije nadmaši maksimalnu vrednost. Takođe, se pokreće da smanji mikroorganizam kada jedinice počnu da odumiru.

ColorChangeAnimation – Animacija kojom se interpolira promena boje mikroorganizma kada mikroorganizam zauzme drugi igrač.

Prilikom inicijalizacije objekta klase *BaseCell*, kreiraju se objekti klasa *IncreasePopulationAffect* i *MaxPopulationExceededAffect* i dodaju u listu `updateables`. Objekti klasa *SizeChangeAnimation* i *ColorChangeAnimation* se kreiraju lenjo (eng. *lazy*) prvi put prilikom pokretanja animacije.

[5] Radni naziv za mikroorganizme je ćelija, stoga se u kodu koristi engleska reč *Cell*.

Kretanje mikroorganizama

Do kretanja mikroorganizma može doći prilikom iniciranja napada, u kom slučaju se mikroorganizam kreće u pravcu mikroorganizma koji se napada, i prilikom primanja napada, u kom slučaju se mikroorganizam pomera u pravcu kretanja jedinice koja ga je napala. Pored ovog, do kretanja može doći i ukoliko se dva mikroorganizma sudare.

Logika kretanja mikroorganizama implementirana je pomoću dve klase *CellMoveProvider* i *CellMover.* Objekat klase *CellMover* je zadužen da pomera mikroorganizam pazeći pritom na ivice mape, dok se kolizija između mikroorganizama rešava u objektu klase *CellMoveProvider.*

Objekat klase *CellMoveProvider* se kreira prilikom pokretanja *engine*-a i dodaje se u listu objekata za osvežavanje. Kako bi klasa *CellMoveProvider* omogućila kretanje, objekti klase *BaseCell* treba da pozovu njenu *factory* metodu *createCellMover* kojom se kreira objekat klase *CellMover*. Ovom prilikom klasa *CellMoveProvider* interno čuva sve objekte klase *BaseCell* koji su izvršili kreiranje objekata klase *CellMover*. Nad objektima klase *BaseCell* se u toku pokretanja *engine*-a poziva metoda *createMover* koja kreira *CellMover* kojim se inicijalizuje atribut klase `mover` koji se ubuduće koristi za kretanje tog mikroorganizma.

Kôd koji pomera objekat klase *BaseCell* prilikom napada je prikazan u narednom primeru. Kako je već

```
float angle = center.getAngle(destinationCell.center);
float velocity = cellUnitsCount * (1f - (population / CELL_MAX_POPULATION));
mover.moveVec.addVector(angle, velocity);
```

pomenuto, mikroorganizam se prilikom napada kreće u pravcu mikroorganizma koga napada što određuje ugao novog kretanja koji će mikroorganizam imati posle napada. Brzina novog kretanja je proizvod broja jedinica sa kojima će se izvršiti napad i inverzne vrednosti trenutne mase populacije. Ovako dobijeni vektor novog kretanja se dodaje na trenutni vektor kretanja kojim se mikroorganizam kreće.

Veoma slično se mikroorganizam pomera prilikom primanja napada. Kôd koji ovo ilustruje dat je u narednom primeru. U ovom slučaju je ugao novog kretanja određen uglom kretanja pod kojim se jedinica

```
float velocity = attackStruct.damage * (1f – (preAttackPopulation / CELL_MAX_POPULATION));
mover.moveVec.addVector(attackStruct.angle, velocity);
```

sudarila sa mikroorganizmom, dok se za brzinu novog kretanja koristi proizvod štete koja se nanosi mikroorganizmu i inverzne vrednosti mase populacije pre napada.

Logika napada i jedinica

Napad se kreira pozivom *factory* metode *createAttack* nad objektom klase *BaseCell*. Prilikom poziva metode potrebno je proslediti objekat klase *BaseCell* koji predstavlja mikroorganizam koji se napada. Metoda *createAttack* vraća objekat tipa *UnitAttack* koji predstavlja osnovnu klasu svih napada. Kako je metoda *createAttack* apstraktna konkretni tipovi koji nasleđuju klasu *BaseCell* su u obavezi da vrate odgovarajuć tipa *UnitAttack*-a kojim se koristi taj mikroorganizam. Tako npr. klasa *VirusCell* kreira objekat tipa *VirusAttack, ParasiteCell* kreira objekat tipa *ParasiteAttack*, a klasa *VelumCell* kreira objekat tipa *VelumAttack*.

Po kreiranju napada potrebno je započeti izvršenje napada pozivom metode *performAttack.* Metoda *performAttack* kreira jedinice i potom dodaje objekat kome pripada *engine*-u u listu objekata za osvežavanje i iscrtavanje.

Maksimalni broj jedinica koji se može kreirati napadom je određen vrednošću koju vraća metoda *getMaxUnitCount* koju su dužne da implementiraju klase koje nasleđuju *UnitAttack*. Ukoliko je mikroorganizam izveo napad koji je sačinjen od većeg broja jedinica nego što je vrednost definisana metodom *getMaxUnitCount*, jedinicama se podjednako uvećava vrednost života tako da suma vrednosti života jedinica bude jednaka vrednosti sa kojom je napad iniciran.

Prilikom iscrtavanja, objekat klase *UnitAttack* iscrtava sve jedinice u jednoj seriji, a prilikom osvežavanja delegira poziv *update* metode na jedinice kako bi osvežio njihovo stanje i naposletku proverava da li je napad završen kada se briše iz *engine*-ovih listi objekata za osvežavanje i iscrtavanje.

Klasa koja implementira osnovnu funkcionalnost svih jedinica se zove *Unit*. Ponašanje koje se simulira klasom *Unit* se može razlikovati u zavisnosti od tipa napada koji je kreirao jedinice, odnosno konkretne implementacije klase *Unit*. Tako postoje sledeći tipovi jedinica predstavljeni klasama:

Virus – Definiše ponašanje jedinica koje kreira klasa *VirusAttack.* Jedinice ovog tipa se kreću do mikroorganizma kojeg napadaju gde završavaju svoj život.

Parasite – Definiše ponašanje jedinica koje kreira klasa *ParasiteAttack*. Jedinice ovog tipa se kreću do mikroorganizma kojeg napadaju gde ostaju zalepljene dok ne izgube život.

Velum – Definiše ponašanje jedinica koje kreira klasa *VelumAttack.* Jedinice ovog tipa se kreću do mikroorganizma kojeg napadaju oko koga kruže dok se u njihovoj blizini ne pojavi parazitska jedinica kada gube život napadom nju.

Kretanje jedinica

Logika kretanja je u osnovi ista za sve jedinice. One se kreću slobodno po mapi zaobilazeći mikroorganizme koje ne napadaju, a za razliku od mikroorganizama se ne mogu sudarati. Ovo kretanje je implementirano u klasi *Unit* metodom *moveToDestination* koja ima potpis: `void moveToDestination(PointF destPoint, float moveLenght, float moverAngle)`

Metoda *moveToDestination* pomera poziciju jedinice ka odredišnoj tački koja je definisana prvim argumentom metode. Prilikom kretanja jedinica proverava da li se približila nekom mikrooganizmu kada započinje zaobilaženje stranom suprotnom od strane na kojoj se nalazi mikroorganizam (u odnosu na vektor kretanja jedinice). U ovoj proveri se proveravaju samo mikroorganizmi koji se nalaze u listi bliskih mikroorganizama koja se periodično osvežava. Takođe, sama provera zaobilaženja se ne radi stalno, već samo kada je pri prethodnom kretanju jedinica zaobilazila mikroorganizam ili kada od prethodnog zaobilaženja protekne `Constants.UNIT_BYPASS_INTERVAL` milisekundi. Kôd koji implementira logiku zaobilaženja u metodi *moveToDestination* je dat u narednom primeru.

```
long currentTime = System.currentTimeMillis();

// Check if list of close cells should be updated
if (currentTime > lastCloseCellsSearchTime + CLOSE_CELLS_REFRESH_INTERVAL) {
lastCloseCellsSearchTime = currentTime;    findCloseCells();
}

// Do not perform cell bypassing check at every move. Instead, bypassing
```

```
// check is performed every time while unit's last move was bypassing //
or after bypass interval has passed
boolean doBypass = lastMoveWasBypass || (currentTime > lastBypassTime +
UNIT_BYPASS_INTERVAL);

// Bypass cells that are on the way
if (doBypass) {
   unitMoveVec.setVector(angle, moveLenght);
float unitRadius = size.width / 2;
   // Minimum distance from unit's closest point to cell's closest
   // point when bypassing
   float minCellToUnitDistance = Math.max(unitRadius, UNIT_MIN_BYPASS_DISTANCE);
   // Distance from cell's edge when unit will start to bypass cell
float avoidDistance = unitRadius + minCellToUnitDistance + moveLenght;

   bypassingCells.clear();
   for (int i = 0, len = closeCells.size(); i < len; i++) {
BaseCell closeCell = closeCells.get(i);
      positionDiffVec.init(closeCell.center.x - center.x, closeCell.center.y - center.y);
float cellToUnitDistance = positionDiffVec.getLength();
      float cellRadius = closeCell.size.width / 2;

      // Units is not close enough to the cell
      if (cellToUnitDistance > cellRadius + avoidDistance) {
continue;
      }
      // Unit is moving away from cell
      if (GeomUtil.distPointToLineSegment(destPoint, center, closeCell.center) > cellRadius
            + minCellToUnitDistance) {
continue;
      }
      float correction = unitMoveVec.cross(positionDiffVec) >= 0 ? -1 : 1;
      correction *= (cellRadius + unitRadius + minCellToUnitDistance) / cellToUnitDistance;

      angle += correction;
      angle = angle < 0 ? angle + MathUtils.PI2 : angle;

      isInBypass = true;
lastBypassTime = currentTime;
bypassingCells.add(closeCell);
   }
   lastMoveWasBypass = isInBypass;
}
```

Jedinice, u zavisnosti od tipa, imaju različitu manifestaciju kretanja. Tako npr. jedinice klasa *Virus* i *Parasite* gmižu, dok se jedinice klase *Velum* kreću cik-cak. Oba tipa manifestacije kretanja se nadograđuju na pokret koji se izvodi metodom *moveToDestination* tako što izvršavaju deformaciju osnovnog pokreta menjajući mu ugao kretanja i dužinu pokreta. Tako da prilikom poziva metode *moveToDestination* se kao drugi argument prosleđuje osnovna dužina pokreta na koju je dodata dužina pokreta deformacije, a kao treći argument ugao deformacije koji se dodaje na osnovni ugao ukoliko jedinica nije u procesu zaobilaženja. Klasa koja računa deformaciju koja simulira gmizanje se zove *CreepDeformation*, a klasa koja simulira cikcak deformaciju se zove *ZigZagDeformation*.

U narednom primeru je prikazano kako je u klasi *Virus* implementirano kretanje jedinice. Prilikom gmizanja jedinica ne menja pravac, pa je ugao deformacije uvek jednak nuli. Kako bi se jedinici dao verniji utisak

```
@Override
public void update(long diffTime) {
   // Check if unit has come close enough to cell to attack it
float mutualRadius = (destinationCell.size.width + size.width) / 2;
   if (center.getSquareDistance(destinationCellCenter) <= mutualRadius * mutualRadius)
{       attack.attackWithUnit(this);       return;
   }

   // Update moveDeform
moveDeform.update(diffTime);

   // Set width and height
   size.width = defaultSize.width * moveDeform.scale;
size.height = defaultSize.height / moveDeform.scale;

   // Move to destination
   destPoint.init(destinationCellCenter).sub(destCenterDiffer);
   float moveLenght = moveDeform.moveLenght * singleMoveLength * ((float) diffTime /
singleMoveDuration);
   moveToDestination(destPoint, moveLenght, 0); }
```

gmizanja vrši se neizmenično skaliranje njenih ivica pomoću vrednosti atributa `scale` klase *CreepDeformation.* Konkretno, klasa *CreepDeformation* vrši interpolaciju vrednosti atributa `scale` između dve granične vrednosti u zadatom vremenskom intervalu. Ova logika se nalazi u *update* metodi i prikazana je u narednom primeru.

```
@Override
public void update(long diffTime) {
   // Update scale
   float scaleDeltaTime = (float) diffTime /
scaleTransitionDurration;    float deltaScale = scaleDeltaTime *
scaleDiffer;    scale += deltaScale * scaleDirection;    if (scale >=
maxScale) {       scale = maxScale;       scaleDirection = -1;    }
else if (scale <= minScale) {       scale = minScale;
scaleDirection = 1;
   }
   moveLenght = scale;
}
```

Logika veštački inteligentnih igrača

Klasa *AIPlayer* je osnovna klasa svih igrača kontrolisanih od strane uređaja, odnosno igrača koji poseduju veštačku inteligenciju. Objekti ove klase se kreiraju prilikom inicijalizacije *BattleInfo* objekta sa njihovih deskriptora, a zatim se prilikom pokretanja *engine*-a dodaju u `updateableObjects` listu.

Svi konkretni tipovi VI (veštačko inteligentnih) igrača su potklase klase *StrategyAIPlayer.* Klasa *StrategyAIPlayer* implementira logiku upravljanja na principu strategija koje se izvršavaju prilikom osvežavanja VI igrača. Konkretni tipovi VI igrača se razlikuju po strategijama koje koriste, koje mogu biti:

DummyVirusStrategy – Strategija kojom se kontrolišu mikroorganizmi tipa virus. Mikroorganizam kojim se napada se bira ponderisanjem populiacije od prijateljskih mikroorganizama, a mikoorganizam koji se napada se bira ponderisanjem udaljenosti između izabranog mikroorganizma i neprijateljskih mikroorganizma.

LegitVirusAttackingStrategy – Strategija kojom se kontrolišu mikroorganizmi tipa virus. Strategija prvo proverava da li postoji prijateljski mikroorganizam koga treba ojačati u kom slučaju se on ojačava kreiranjem napada iz mikroorganizama koji imaju najveću populaciju. Strategija smatra da mikroorganizam treba ojačati kada njegova populacija postane bar pet puta manja od prijateljskog

mikroorganizma koji ima najveću populaciju. Ukoliko ne postoji prijateljski mikroorganizam koga treba ojačati strategija izvšava napad na neprijateljski mikroorganizam. U ovom slučaju strategija bira prijateljske mikroorganizme sa najvećom populacijom sa kojim može izvršiti napad na neprijateljski mikroorganizam koji bi rezultirao njegovim zauzmanjem. Ova provera napada se vrši za sve neprijateljske mikroorganizme birajući onaj napad čija je prosečna udaljenost između mikroorganizama koji napadaju i mikroorganizma kojeg napadaju manja.

LegitVirusDefenderStrategy – Strategija kojom se kontrolišu mikroorganizmi tipa virus. Strategija uvek ojačava prijateljske mikroorganizme sa najmanjom populacijom pomoću mikroorganizama sa najvećom populacijom.

ParasiteStrategy – Strategija kojom se kontrolišu mikroorganizmi tipa parazit. Strategija napada mikroorganizmom sa najvećom populacijom na neprijateljske virus mikroorganizme birajući prvo one najbliže koji na sebi već nemaju parazite.

VelumStrategy – Strategija kojom se kontrolišu mikroorganizmi tipa membrane. Strategija napada mikroorganizmima sa najvećom populacijom na prijateljske virus mikroorganizme birajući prvo one koji na sebi imaju parazite sa najmanjom vrednošću života.

Kako VI igrači ne bi bili previše jači od korisnika, svim strategijama se dodeljuje tempo kojim kreiraju napade. Tempo kreiranja napada je definisan u konfiguraciji VI igrača i može biti različit u zavisnosti od mape.

Optimizacija igre

Igre za Android platofrmu mogu biti veoma jednostavne, a opet mogu imati loše performanse ako se ne obrati pažnja na specifičnosti diktirane od strane platforme i programskog jezika. Ovaj deo rada se bavi dobrom programerskom praksom koju treba slediti kako bi se performanse igre uvećale. Takođe će biti reči o načinu na koji su ove optimizacije sprovedene u delo u igri *Microorganisms*.

Pozivi metoda

Pozivi metoda su daleko sporiji od direktnog pristupa poljima klasa, čak i do sedam puta kod virtuelnih mašina sa JIT kompajlerom. Zbog ovoga treba izbegavati *getter* i *setter* metode bar na klasama čijim se poljima intezivno pristupa.

Pristup lokalnim promenljivama

Pristup lokalnim varijablama unutar metoda se izvršava brže nego pristup poljima klasa. Zbog toga vrednosti kojima se često pristupa treba keširati unutar metoda.

Garbage collector

Javin *garbage collector* (GC) može praviti najviše problema jer njegovo izvršavanje može oduzeti nekoliko desetina do nekoliko stotina milisekundi procesorskog vremena. Ovo vreme nekad može biti dovoljno da igrač primeti usporenje. Zbog ovoga dinamičku alokaciju treba izbegavati, pogotovu unutar *update* i *draw* metoda jer se ove metode izvršavaju i do 60 puta u sekundi. Dobra praksa je da se sva alociranja memorije izvrše pre početka igre i da u toku igre nema dodatnih alociranja memorije.

Recikliranje objekata

Kako bi se u toku izvršavanja smanjio broj nedostižnih objekata, a samim tim i učestalost pokretanja GCa, objekte treba kreirati samo jednom i zatim ih reciklirati umesto da se objekti iznova kreiraju. U zavisnosti od životnog veka objekta mogu se izdvojiti dva načina pomoću kojih se objekti mogu reciklirati:

a. Uslužne objekte koji se kreiraju unutar metoda treba „podići“ na nivo polja klase i kreirati samo jednom kada se klasa instancira. Zatim je na mestima na kojima se izvršavalo kreiranje ove objekte je potrebno samo ponovo incijalizovati.
b. Objekti čiji je životni vek unapred nepoznat mogu se reciklirati korišćenjem tehnike bazena objekata. Kod ove tehnike se konstruiše bazen iz koga se uzimaju objekti kada postanu potrebni i vraćaju se u bazen kada se korišćenje objekata završi. Ukoliko se bazen isprazni potrebno ga je napuniti sa novim objektima.

Tehnika bazena objekata se u igri *Microorganisms* uspešno koristi za kreiranje jedinica. Pogodnost primene bazena objekata za kreiranja jedinica se ogleda u tome što je veoma lako odrediti mesta na kojima je potrebno jedinicu uzeti i vratiti u bazen. Naime, jedinice se kreiraju prilikom iniciranja napada što predstavlja trenutak kada bi se jedinice uzimale iz bazena. Jedinica završava svoj životni vek kada biva uništena i ovo je trenutak u kojem se jedinica vraća natrag u bazen.

Klasa koja predstavlja bazen za čuvanje jedinica se zove *UnitPool*. Instanca ove klase se kreira prilikom pokretanja *engine*-a borbe sa inicijalno popunjenim bazenima za svaki tip jedinica koji se može naći u borbi. Metoda kojom se jedinica može uzeti iz ovog bazena je *getUnit*, a metoda kojom se jedinica može vratiti u bazen je *returnUnit* odnosno *returnUnits* ako se radi o listi jedinica.

Rad sa klasom *UnitPool* je prikazan u narednom primeru u kome se prvo izvršava popunjavanje bazena prilikom poziva *start* metode klase *BattleEngine*. Zatim se u klasi *UnitAttack* u *factory* metodi za kreiranje

```
// Fill unit pools
unitPool.createPools(battleInfo.getAllUnitTypes());
```

jedinica pristupa ovom bazenu kako bi se jedinica reciklirala. Naposletku, kada se jedinica ukloni iz napada

```
protected Unit createUnit(int strength) { Unit unit = unitPool.getUnit(unitType); unit.init(this, strength); return unit; }
```

ona se tom prilikom vraća u bazen što je prikazano u narednom primeru.

```
// Remove and return units to the pool if
(!unitsForRemoval.isEmpty()) {
attackUnits.removeAll(unitsForRemoval);
unitPool.returnUnits(unitsForRemoval);
unitsForRemoval.clear();

   if (attackUnits.isEmpty()) {
endAttack();
   }
}
```

Skrivene alokacije

Dinamička alokacija se može desiti na mestima koja nisu očigledna, najčešće prilikom rada sa kolekcijama. Recimo, *foreach* prilikom iteracije instancira iterator kolekcije, zatim neke implementacije metoda za sortiranje kolekcija mogu kreirati pomoćne nizove, itd.

Kako bi se skrivene alokacije prilikom rada sa kolekcijama eliminisale u TinyLib biblioteci napisana je klasa *OArray* za rad sa kolekcijom sa promenljivom veličinom elemenata. Klasa *OArray* garantuje da prilikom rada neće izvršavati alokacije osim u slučaju kada je potrebno proširiti kolekciju. Kako ova klasa ne implementira interfejs *Iterable* ne može se koristiti u *foreach* petljama. Sa ovim svi izvori skrivenih alokacija pri radu sa kolekcijama su eliminisani.

Pored klase *OArray*, TinyLib biblioteka poseduje još jednu klasu za rad sa kolekcijama koja se zove *ReusableArray*. Specifičnost ove klase u odnosu na *OArray* je u tome da se prilikom pražnjenja kolekcije pozivom metode *clear* ne brišu reference elemenata koje se nalaze u kolekciji, odnosno vrši se logičko pražnjenje. Razlog čuvanja referenci je da se elementima može ponovo pristupiti kako bi se reciklirali. Recikliranje elementa ove kolekcije postiže se pozivom metode *extend* kada se kolekcija proširuje elementom koji se nalazi na lokaciji posle lokacije trenutno poslednjeg elementa.

Matematičke operacije

U zavisnosti od implementacije Jave kao i samog hardware-a, neke matematičke funkcije se mogu izvršavati sporije odnosno brže. Razlog je što u implementaciji matematičke funcije može biti poziv mašinske instrukcije ili se implementacija matematičke funkcije radi u softveru.

Računanje sinusa i kosinusa ugla

Računanje sinusa i kosinusa ugla je veoma česta operacija u brojnim igrama. Ove metode računaju vrednost sinusa odnosno kosinusa korišćenjem nekog algoritma koji daje aproksimativnu vrednost. Uglovi, kao ulaz ovih funkcija, se izražavaju u radijanima koji se predstavljaju u pokretnom zarezu. Ovo znači da će metode za računanje sinusa i kosinusa ugla imati veliki broj operacija sa vrednostima predstavljenim u pokretnom zarezu. Na nekim uređajima i VM operacije u pokretnom zarezu su implementirane u softveru pa bi izvršavanje ovih matematičkih funkcija bilo dosta sporo.

Kako bi se ubrzalo računanje ovih funkcija u TinyLib biblioteci kreirana je klasa *MathUtil* koja poseduje metode za brzo računanje sinusa i kosinusa ugla. Ove funkcije koriste tabele sa unapred izračunatim vrednostima.

Zaključak

Za razvoja igara za Android platformu programeru se nude gotova rešenja koja se vrlo efikasno mogu koristiti pri izradi igara.

Standardne Android komponente se mogu prilagoditi temi igre pa se mogu koristiti za izradu ekrana sa menijima. Ovo daleko olakšava razvoj jer se komponente zasnovane na Androidovom *framework*-u mogu kombinovati u složenije komponente.

Prilikom programiranja grafike programeru se pružaju dva API-a koja dolaze us Android platformu. Canvas API je veoma trivijalan za korišćenje, ali ima lošije performase od OpenGL ES API-a. Programer se može odlučiti da koristi gotove bibliote za programiranje grafike zasnovane na OpenGL ES API-u koje bi daleko pojednostavile programiranje grafike.

Na kraju, treba obratiti pažnju i na specifičnosti diktirante od strane platforme i programskog jezika kako bi igra ostvarila što bolje performanse.

Literatura

[1] Mike Smithwick, Mayank Verma, *Pro OpenGL ES for Android*, Apress 2012

[2] Mario Zechner, Robert Green, *Beginning Android Games, Second Edition*, Apress 2012

[3] Yevgen Karpenko, *Android 2D Graphics with Canvas API*, USA 2013

[4] Richard A. Rogers, *Learning Android Game Programming*, Addison-Wesley 2012

[5] J. F. DiMarzio, *Practical Android 4 Games Development*, Apress 2011

[6] *Animation and Graphics* [Online] http://developer.android.com/guide/topics/graphics/index.html

[7] *Global Games Market Will Reach $102.9 Billion in 2017* [Online] http://www.newzoo.com/insights/global-games-market-will-reach-102-9-billion-2017-2/