INSTITUT ZA MATEMATIKU I INFORMATIKU
PRIRODNO-MATEMATIČKI FAKULTET
UNIVERZITET U KRAGUJEVCU

SEMINARSKI RAD

WRAPPER

Mentor
dr Ana Kaplarević-Mališić

Student
Stefan Nestorović, 87/15

Jul 2017.

Sadržaj

1. Uvod

Seminarski rad je sačinjen u vidu dokumentacije za aplikaciju pravljenu kao deo ispitnih obaveza iz predmeta Objektno-orijentisano programiranje.

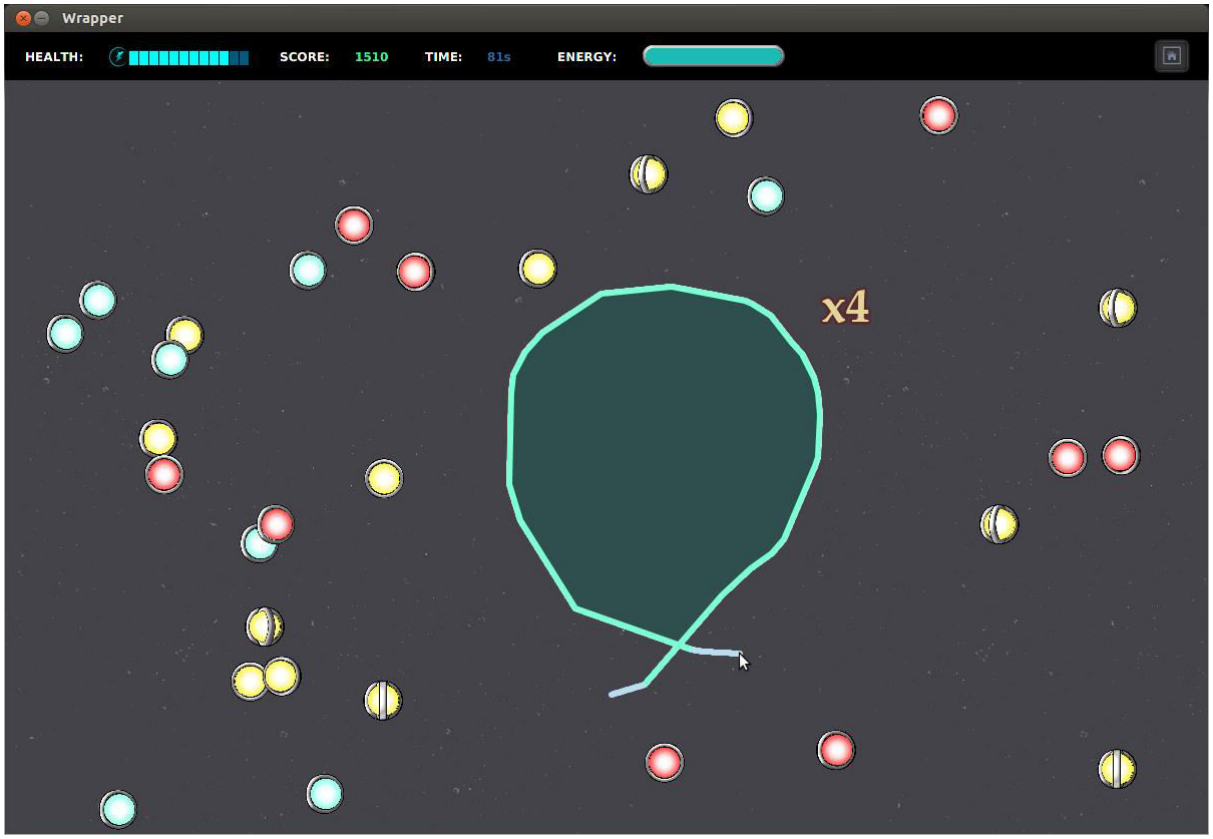
Tekst ima za cilj da sto jednostavnije i pomoću vizuelnih pomagala objasni čitaocu kako je izvedeno softversko rešenje za ovu aplikaciju.

Kompletna aplikacija je napisana u Java programskom jeziku i to u njenoj standardnoj biblioteci JavaFX.

1.1. Wrapper - pravila igre

Aplikacija je zamišljena kao igra u kojoj jedan igrač igra ulogu intergalaktičke zmije - Wrapper-a, sa ciljom da obuhvati što više sfera koje donose poene.

Igrač pomeranjem pozicije kursora u prozoru iscrtava telo zmije kojim treba da napravi poligon oko objekata, koji se na ekranu nasumično kreiraju tokom igre, kako bi sakupio poene. Cilj je sakupiti što vise dobrih sfera, koje se zajedno sa lošim sferama sve brze i više generišu što vreme igre dalje odmiče. Pritom igrač na početku igre ima 12 života, koje za vreme igre može da izgubi ili da vrati na više načina, a igra se završava kada igrač izgubi sve živote.



1. Slika igre

- Sfere koje se generišu mogu biti:

- Plave sfere
    - Cilj je sakupiti što više ovih objekata jer donose poene.
    - Imaju dva stadijuma, prvi traje 5 sekundi i tada sfera ima vrednost od 10 poena, nakon tih 5 sekundi sledi drugi period od još 5 sekundi kada plave sfere poprimaju žutu boju i animaciju čime označavaju da tada vrede 30 poena ako se uhvate.
    - Dobija se bonus ako se više sfera uhvati jednim poligonom.
        - Bonus se racuna kao:

          zbirVrednostiSvihUhvacenihSfera * brojUhvacenihSfera,

          tako da ako npr. uhvatimo jednu plavu (10) i jednu zutu (30), ukupno smo zaradili 80 poena!
        - Treba napomenuti da je najveći mogući bonus x7, dakle ako igrač uvhati 10 sfera, njihov zbir biće pomnožen sa 7 i dodat u rezultat.
    - Ako žutoj sferi istekne tajmer pre nego što je igrač uhvati, automatski se gube 2 života.

- Crvene sfere
    - Ove sfere treba izbegavati jer donose negativne poene i to -50, pri čemu je moguće otići u minus sa brojem poena.
    - Traju 10 sekundi ili dok se ne uhvate.
    - I kod njih važi multiplikacija kao rezultat hvatanja više sfera odjednom. Tako da ako npr. uhvatimo jednu žutu (30) i jednu crvenu (-50), na sumu poena se dodaje -40.
    - Za svaku crvenu sferu koju igrač uhvati, gube se 4 života.

Svojim kretanjem (iscrtavanjem svake nove linije) igrac generiše energiju. Kada se energija sasvim napuni (samo tada) moguce je potrositi je. Energija se troši pritiskom na levi taster misa (samo na njega, inace se ne registruje) i prevlačenjem (drag) po površini prozora. Na energy bar-u je prikazano u kojem periodu taster treba da se pusti.

- Sada razlikujemo dve mogućnosti:

1) Igrač je pustio taster na vreme:
    - Sa celog se prozora hvataju trenutno postojeće dobre sfere, pri cemu se dobijeni poeni računaju sa bonusom.
    - Igrač dobija 2 dodatna zivota, koja se ne računaju ako igrač vec ima maksimalnih 12 zivota.
2) Igrač je pustio taster prerano ili prekasno:
    - Dužina wrapper-a se skraćuje na polovinu od početne na 10 sekundi i sam wrapper dobija crvenu boju za taj vremenski perod.

Igrač može u bilo kojem trenutku igre da napravi pauzu i da otvori mini meni pritiskom na home dugme u gornjem desnom uglu, ili pritiskom na taster ESC, odakle može da se vrati u staru igru (RESUME), započne novu (NEW GAME), vrati se na glavni meni (MAIN MENU) ili da sasvim izađe iz igre (EXIT GAME).

1.2. Specifikacija zahteva

1) Laka navigacija kroz sadržaj putem glavnog menija, njegovih podmenija i više mini menija u toku igre.
2) Intuitivna modifikacija izgleda i osećaja aplikacije od strane korisnika.
3) Brza interakcija između prikazanog sadržaja i onoga što korisnik očekuje da bude prikazano.
4) Čuvanje rezultata igara radi lakog opažanja napretka u igri.

2. Arhitektura softverskog rešenja

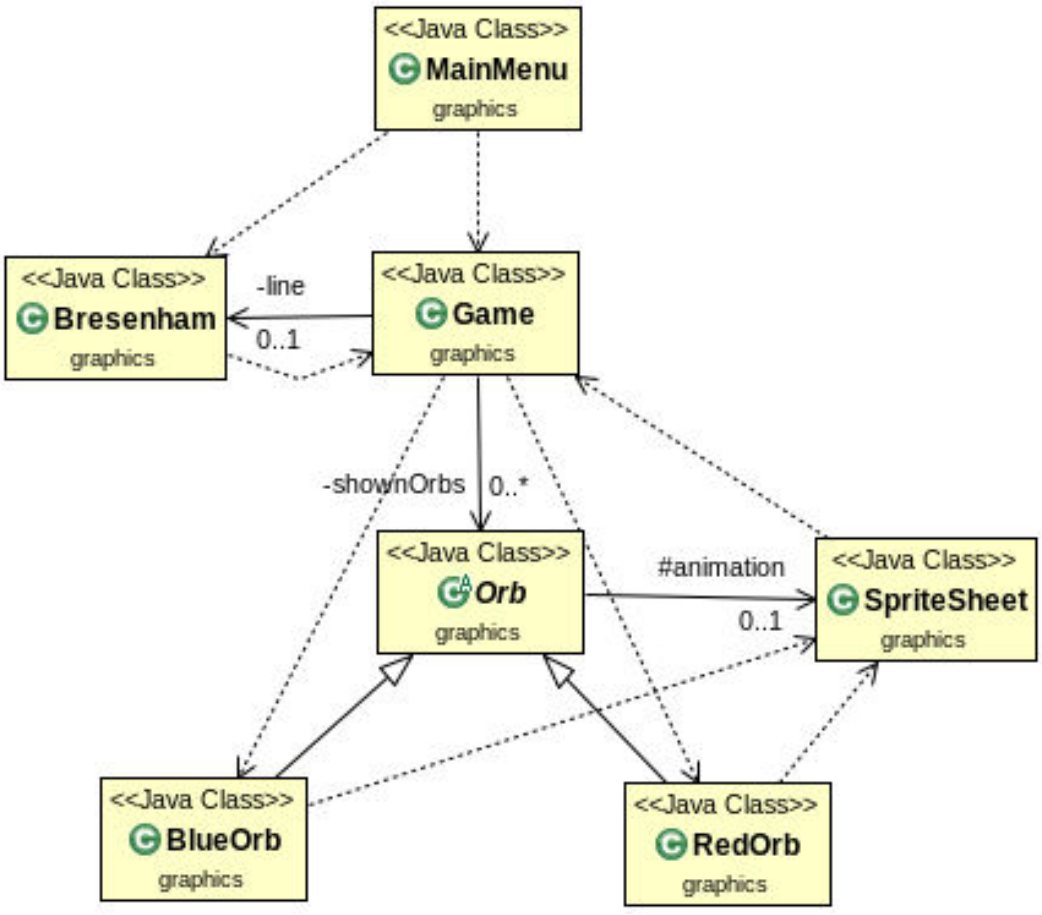
- Aplikacija je na visokom nivou apstrakcije funkcionalno podeljena na:

1) **Logički deo** - koji se bavi rešavanjem svih logičkih problema koji se javljaju i čini mozak aplikacije.
2) **Grafički deo** - koji se bavi samo čistim prikazom komponenti na ekranu.
3) **Dizajn** - čiji je temelj u CSS-u koji je novina u JavaFX-u, a koji služi da prilagodi izgled prikazanih komponenti.

Svakom od ovih slojeva je pridružen paket u aplikaciji i to respektivno:

paket logic, paket graphics i *style.css*

2.1. Grafički sloj

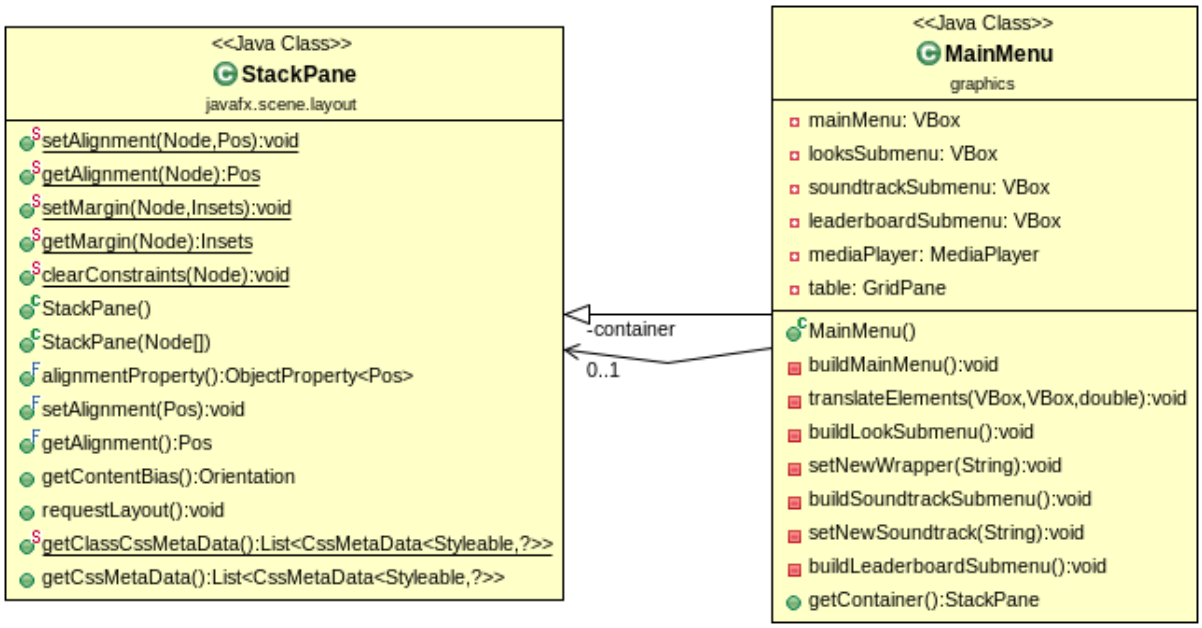


2. Dijagram klasa paketa graphics

Na osnovu ovog jednostavnog dijagrama koji samo prikazuje relacije, možemo primetiti da svaki entitet aplikacije koji ima svoje, specifično ponašanje ima sebi pridruženu klasu.

2.1.1. Glavni meni

Krenimo sada od prvog entiteta sa kojim se srećemo pri pokretanju aplikacije, od glavnog menija. Ispod se nalazi njegov UML dijagram.



3. Dijagram klase MainMenu

Sa dijagrama se vidi da je kompletan glavni meni u stvari proširenje klase *StackPane* iz paketa javafx.scene.layout.

java.lang.Object

javafx.scene.Node

javafx.scene.Parent

javafx.scene.layout.Region

javafx.scene.layout.Pane

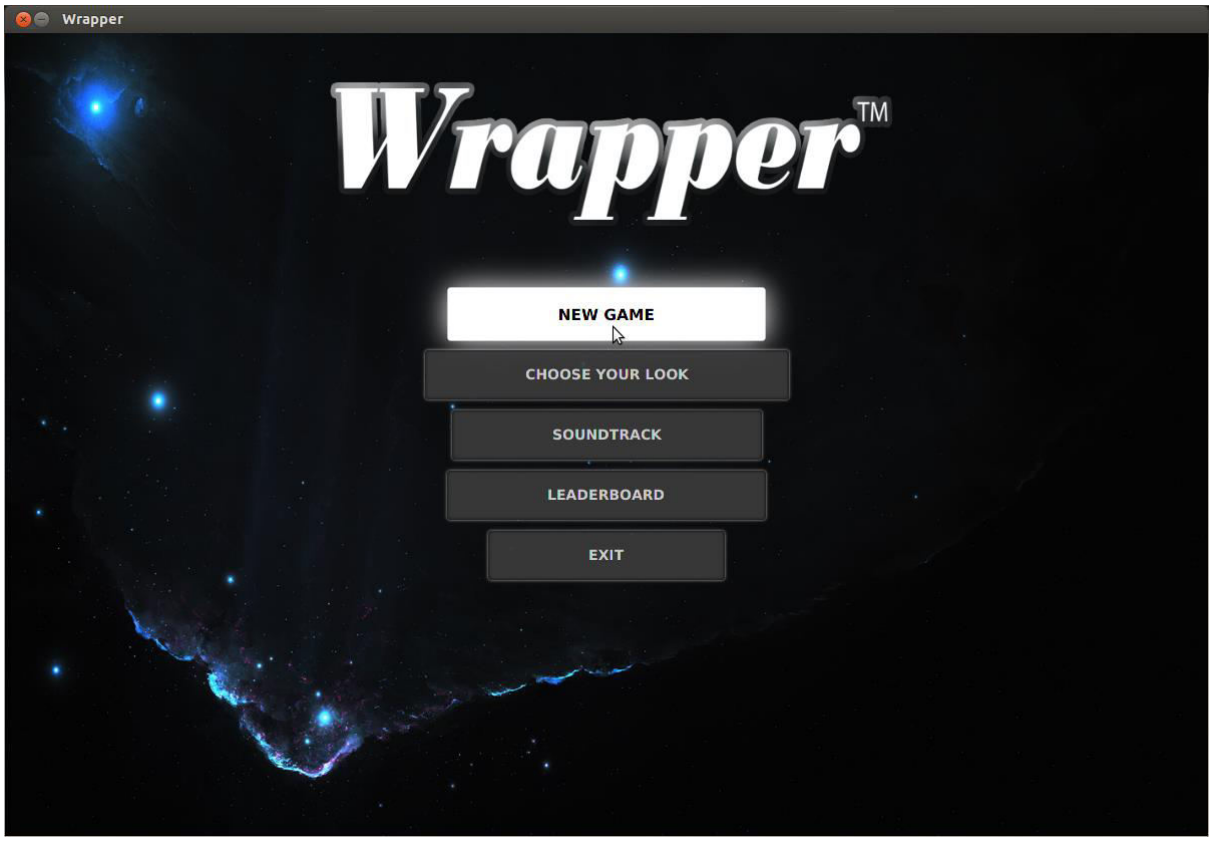
javafx.scene.layout.StackPane

Iznad je prikazan lanac nasledjivanja kojim se dolazi do klase *StackPane*. Ispod klase *Object* iz koje se izvode sve klase u Javi, primećujemo klasu ***Node*** koja predstavlja baznu klasu scenskog grafa čvorova. Scenski graf je skup struktura stabala gde svaki član ima nula ili jednog roditelja (scenski graf je uvek stablo), I svaki član je ili *list (leaf)* koji nema daljih potomaka ili je *grana (branch)* koji može imati nula ili vise podčlanova. *Klasa Node*, sama po

sebi je apstraktna klasa I nju nasleđuju sve vizuelne komponente u JavaFX-u. Sadrži podatke o svojem položaju na ekranu, poput X I Y koordinata, svoje okvire (Bounds), metode za promenu I preuzimanje istih, za rotaciju (Rotate), skaliranje (Scale) I slično.

Svaki član scenskog grafa se zove **Node**. Svaki tip koji igra ulogu *grane* u scenskom grafu jeste tipa **Parent**, cije su glavne podklase Group, Region I Control. Listovi grafa ne mogu sadržati druge komponente u sebi I oni su direktni potomci klase Node, kao što su Canvas, ImageView, MediaView, Shape I sve njihove podklase. I naravno, tu je I **Root**, koji je jedini čvor u stablu koji sigurno neće imati roditelja, a često je sam roditelj, pa je I tipa Parent.

Dakle, iz svega ovoga vidimo da je StackPane, u stvari roditeljska klasa I da je time I MainMenu klasa, ove aplikacije takođe roditeljska klasa, sto je sasvim logično, jer će naš meni sadržati druge elemente koji će u scenskom grafu biti njegova deca. StackPane sam po sebi, kao I sve podklase klase **Pane** (koja je sama kao null layout u Swing-u), se ponaša slično *LayoutManager*-ima iz JavaSwing-a, tj. Ima implementiran određen način rasporedjivanja elemenata unutar sebe, a to je u njegovom slučaju pakovanje svih novih elemenata jednih preko drugih (kao stack), što je u našem slučaju sasvim u redu, jer ćemo u jednom trenutku imati samo jedan meni/podmeni prikazan u prozoru I njega ćemo stavljati u objekat *container*, koji je StackPane.



4. Izgled glavnog menija

Ono sto je prikazano na slici iznad je u stvari sadržaj objekta mainMenu koji je tipa *VBox*, koji je takodje kontejnerski tip I to podklasa tipa Pane. On prikazuje svu svoju decu vertikalno, jedno ispod drugog, koja su u našem slučaju dugmići tipa *Button*.

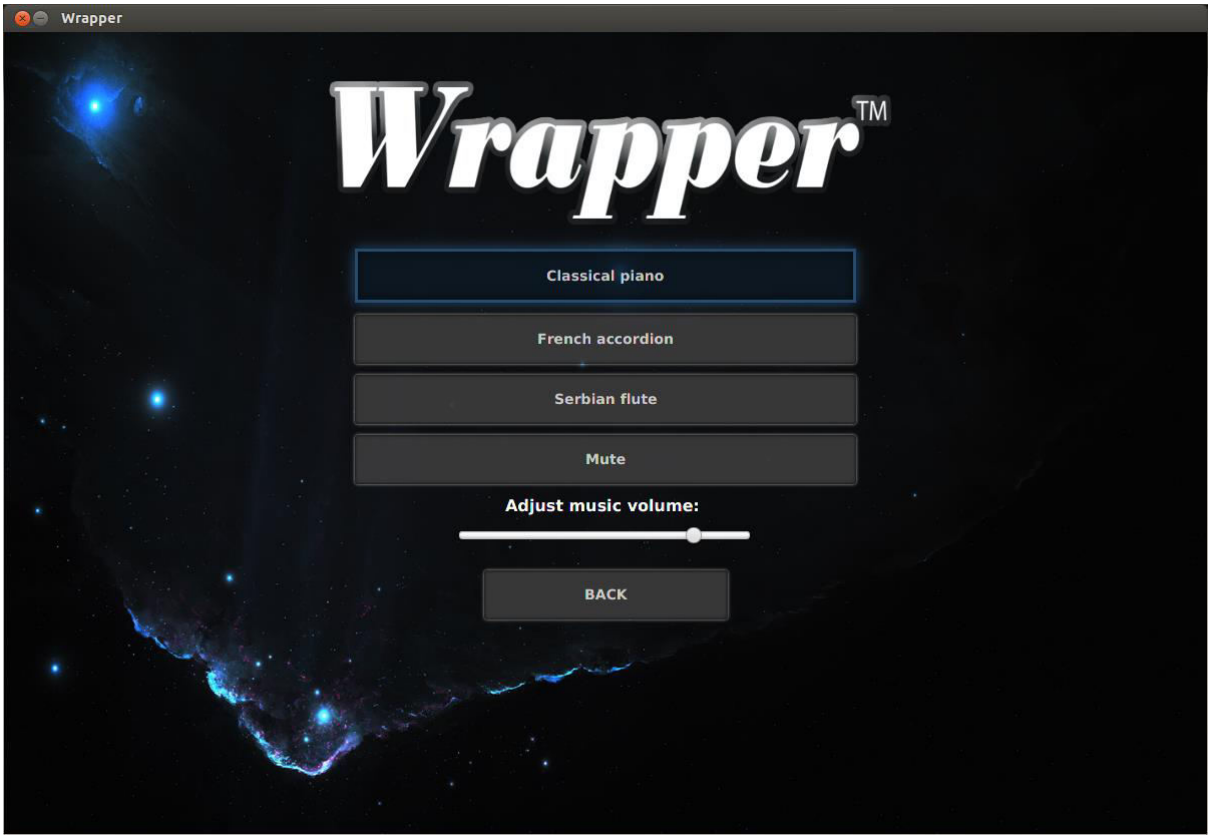
Klikom na dugme, koje ima implementiran metod *handle* osluškivača EventHandler, izvršava se metod, koji uz pomoć objekata tipa *TranslateTransition*, pravi animaciju kretanja I smene dva menija, a pritom se iz objekta *container* uklanja jedan meni, a postavlja drugi, komandama:

```
container.getChildren().remove(mainMenu);
container.getChildren().add(looksSubmenu);
```

5. Izgled podmenija CHOOSE YOUR LOOK

Kada se izvrše gornje dve komande, dobijamo ovaj podmeni, u njemu možemo da izaberemo izgled koji nam se najviše dopada, tj. boju traga I poligona koji nastaju našim kretanjem. Izbor se vrši jednostavnim klikom na sliku koja ilustruje mogući izgled.

Slike su u stvari stilizovani objekti tipa *RadioButton*, što nam govori da uvek mora postojati neki izbor, koji je u početnom slučaju postavljen na plavi wrapper. Ono što se zaista dešava kada kiknemo na sliku jeste okidanje dogadjaja *onAction,* koji poziva metod `setNewWrapper(String):void`, koja postojećem objektu tipa *Bresenham*, koji se bavi crtanjem linija I njihovim izgledom, javlja da za crtanje upotrebi drugu paletu. I ovo čini *relaciju zavisnosti* koju smo mogli da primetimo na slici 2.



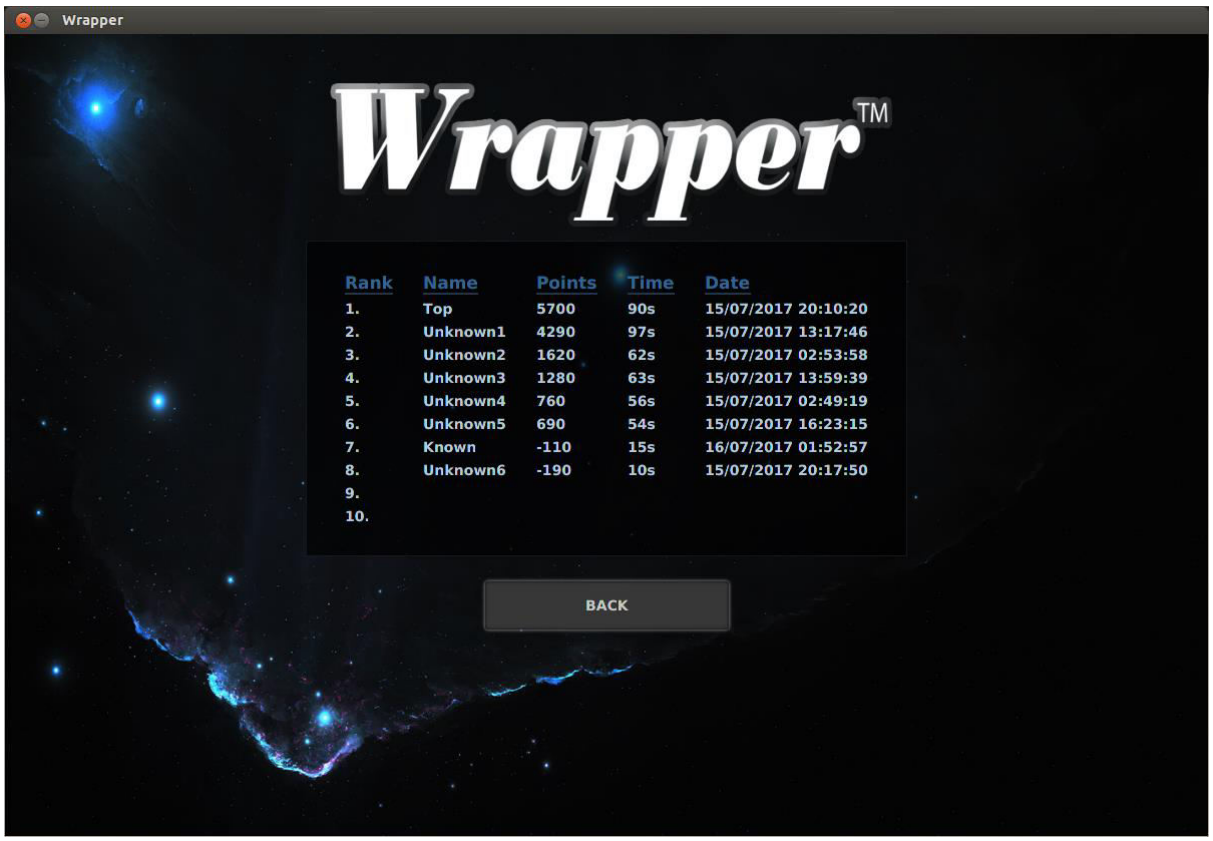
6. Izgled podmenija SOUNDTRACK

Izgled ovog podmenija u kome se može izabrati željena muzika kao I jačina iste, je veoma sličan onome što smo već do sad videli. Svaki od dugmića za izbor pesama su u stvari objekti tipa *RadioButton*, I klikom na svaki od njih se poziva metod `setNewSoundtrack(String):void`, koji za različite argumente menja atribut *mediaPlayer* i učitava poseban .mp3 fajl iz foldera *src/music*, I pušta ga da svira beskonačan broj ciklusa. Klikom na već izabrano dugme se, naravno, ništa ne dešava.

Jačina muzike se veoma lako može podesiti pomoću klizača. Klizač je objekat tipa *javafx.scene.control.Slider*, koji ima definisan izgled I koji pomeranjem kruga na sebi sam ažurira svoju vrednost I postavlja je u svoju promenljivu *value*. Ono što nama preostaje jeste da posmatramo tu vrednost I da njome ažuriramo jačinu u našem *MediaPlayer* objektu koji ima moć da kontroliše jačiu zvuka koji pušta. Ovo radimo pomoću **JavaFX binova** (beans). Binovi se razlikuju od običnih klasa I objekta, između ostalog I po tome sto drugi objekti mogu posmatrati (observe) polja binova I vezati se (bind) za njih, što znači da će automatski biti obavešteni kada neko polje promeni vrednost. Binovi nemaju obične atrubute, već imaju osobine (properties), tako da umesto int, imamo IntegerProperty, koji ima svoje pristupne metode I kome mogu da se pridruže osluškivači na promenu (*ChangeListener* – čiji metod dobija I staru I novu vrednost posmatranog property-a I *InvalidationListener*).

graphics/MainMenu.java - buildSoundtrackSubmenu():void

```
1  Slider volumeSlider = new Slider();
2  volumeSlider.valueProperty().addListener(new InvalidationListener()
3  {
4      public void invalidated(Observable ov)
5      {
6            mediaPlayer.setVolume(volumeSlider.getValue() / 100.0);
7      }
8  });
```



7. Izgled podmenija LEADERBOARD

Svrha ovog podmenija je da prikaže trenutnih Top10 igrača koji su odigrali igru. Tabela je sačinjena kao instanca objekta tipa *GridPane,* koji je proširenje klase Pane I slično GridLayout-u, elemente (u ovom slučaju instance klase Label) pakuje u redove I kolone.

Treba napomenuti da se ova klasa ne bavi cuvanjem I obradom podataka koji su prikazani u ovoj tabeli. Za to postoji posebna klasa iz paketa *logic,* koja se zove **FileEngine.** FileEngine ima više funkcija, o njima će biti više reči u sekciji za paket koji se bavi logikom aplikacije. Za sada je dovoljno reći to da FileEngine vrši jedno čitanje iz tekstualnog fajla koji čuva podatke o prethodnim igrama, I to na startu aplikacije. Te podatke smešta u listu tipa **ObservableList**, na koju se može postaviti osluškivač na promenu (*ListChangeListener*) tako da npr. promeni atribut *listChanged* na vrednost true, ako dodje do promene u sadržaju same liste, tj. kada neki takmičar obori record u postojećoj listi I želi sebe da doda u istu.

Tako da mi klikom na dugme LEADERBOARD iz glavnog menija, prvo pitamo da li je došlo do promene u listi rezultata, ako jeste, preuzimaju se novi podaci od klase FileEngine i gradi se novi GridPane koji se prikazuje, a vrednost promenljive *listChanged* se iznova setuje na false, tako označavajući da se čeka na sledeću promenu liste. Inače se prikazuje već postojeći objekat tabele iz klase MainMenu, jer je I dalje konzistentan sa realnim stanjem u listi rekorda.

Stigli smo I do dugmeta NEW BUTTON, koje sasvim očekivano pokreće novu igru. Ali kako se stvarno izvršava ta zamena dva sasvim različita entiteta poput same igre I glavnog menija? Odgovor leži u *strukturi stabla* koju JavaFX aplikacija ima. U korenu tog stabla se uvek nalazi **pozornica (stage)**, I ona predstavlja *top-level* kontejner, koji ako je aplikacija pokrenuta na desktop operativnom sistemu će biti *prozor* (window), ili ako je pokrenuta u okviru veb stranice će biti aplet (applet).

Pozornica sadrži tačno jednu **scenu (scene)**, koja dalje sadrži čvorove (nodes), dakle scena je glavni kontejner za sadržaj scenskog grafa. Pri kreiranju scena očekuje da primi jedan objekat koji je tipa *Parent*, dakle to može biti bilo koji layout ili grupa koja dalje može da sadrži svoje potomke. Opcioni parametri su veličina scene I njena boja.

Sada je jasno da je ovde sistem takav da mi, kao I u pozorištu imamo pozornicu, I za vreme održavanja predstave (rada programa) se na njoj smenjuju različite scene. U našem slučaju imamo dve scene, to su glavni meni I sama igra. Tako da je sasvim logično da ćemo u klasi *Main*, čija je svrha da objedini različite delove naše aplikacije I da postavi pozornicu, napraviti dva objekta, jedan tipa MainMenu, drugi tipa Game I njih ćemo postaviti za korene dve scene koje ćemo imati.

Sve što je preostalo jeste sistem koji će znati kada da stavi koju scenu u pozornicu. To smo postigli korišćenjem jednog statičkog objekta tipa ***SimpleIntegerProperty*** u klasi Main, na koji smo definisali osluškivač promene, tako da:

- Ako je njegova nova vrednost 0 - treba da postavi scenu glavnog menija.
- Ako je njegova nova vrednost 1 - treba da inicijalizuje I započne novu igru.

Ovime smo postigli pouzdanu nezavisnost između različitih komponenti I obezbedili smo da mogu lako da se uvedu I nove scene u aplikaciju, bez ikakve promene sistema rada starih.

graphics/Main.java – start(Stage):void

```
9   toggleScenes = new SimpleIntegerProperty(0);
10  toggleScenes.addListener(new ChangeListener<Number>()
11  {
12      @Override
13      public void changed(ObservableValue ov, Number oldVal, Number newVal)
14      {
15          if (newVal.intValue() == 1)              // Postavi scenu za novu igru
16          {
17              game.initializeNewGame();
18              primaryStage.setScene(gameScene);
19              game.getGameLoop().play();
20          }
21          else if (newVal.intValue() == 0)         // Postavi scenu za meni
22          {
23              game.getGameLoop().stop();
24              primaryStage.setScene(menuScene);
25          }
26      }
27  });
```

graphics/MainMenu.java – buildMainMenu():void

```
28  Button newGame = new Button("NEW GAME");
29  newGame.setOnAction(e-> {
30        Main.setToggleScenes(1);       // Pokreni novu igru
31  });
```

2.1.2. Igra

8. Dijagram klase Game

Klasa *Game* je centralna klasa koja se bavi izgledom same igre. Osim nje, grafikom igre se bave I klase:

- *Bresenham* – bavi se iscrtavanjem linija I njihovim izgledom.
- *Orb, BlueOrb, RedOrb* – koje se bave izgledom I osobinama sfera.
- *SpriteSheet* – koja se bavi animacijom sfera.

U objekat tipa Game, koji je Pane, mi direktno stavljamo samo objekat *root* koji je tipa BorderPane. U severnom delu root-a, nalazi se meni sa bitnim podacima o trenutnoj igri, poput broja života, poena, vreme koje je proteklo od početka igre, status energije, dugme za mini meni. U centru root-a se nalazi jedan objekat tipa *Canvas* (platno). Canvas je slika po kojoj se može crtati korišćenjem skupa grafičkih komandi obezbeđenih od strane jednog objekta tipa *GraphicsContext*, koji se pridružuje kanvasu.

--- Problem1 --- Kako se vrši iscrtavanje linije pri pokretanju miša?

Kao što je već rečeno, klasa Bresenham je specijalizovana za crtanje linija I za odredjivanje izgleda istih. Tako da u klasi *Game* moramo imati jednu instancu te klase I to je u ovom slučaju objekat *line*. Kako znamo gde treba nacrtati liniju? Odgovor je pomoću osluškivača koji su pridruženi kanvasu.

graphics/Game.java – Game()

```
32  canvas.setOnMouseMoved(e-> {
33      registerMovement(e.getX(), e.getY());
34  });
```

Ono što se nalazi kao argument metodu *setOnMouseMoved* jeste **Lambda izraz**, koji je u suštini izraz u kojem implementiramo interfejs sa jednim metodom, a gde je e, u stvari argument koji taj metod dobija. U našem slučaju to e je argument tipa *MouseEvent* koji se šalje metodu *handle*, interfejsa *EventHandler*. Kako je e dogadjaj miša, on sigurno sadži koordinate pozicije kursora, I mi njih prosleđujemo našoj metodi koja će da obradi pokret I da ga prikaže

Kako se vrši samo iscrtavanje wrapper-a? Mi novodobijene koordinate prosleđujemo objektu klase *Engine,* koja se bavim celokupnom logikom igre. Engine te koordinate dodaje u listu koordinati koje čine našeg trenutnog wrapper-a, a pritom najstarije tačke izbacuje ako je dužina prevazišla njegovu maksimalnu, što čini srž osećaja kretanja u toku igre.

Zatim naš objekat klase Game, preuzima tu listu koja je sačinjena od objekata tipa Dot2D, I između tih tačaka crta linije pomoću metoda iz klase Bresenham koji se zove *bresenhamLine(double,double,double,double):void* našeg objekta *line*. Ovaj metod koristi jednostavnu verziju Bresenham-ovog algoritma za crtanje duži.

Kao argumente saljemo koordinate tačaka (x0,y0) I (x1,y1) između kojih se crta duž. Algoritam zatim određuje koje celobrojne vrednosti (1 ili -1) treba da dodaje na koordinate x0 I y0, kako bi na kraju završio u tački (x1,y1). Algoritam dalje određuje koje su tačke na putu najpovoljnije za crtanje I svakoj od njih pridružuje sliku 6x6 piksela, čiju smo boju odredili još u glavnom meniju izborom omiljene boje wrapper-a. Te tačke su u stvari tačke na kanvasu, tako da ovaj metod koristi *GraphicsContext* objekat kako bi proizveo željenu sliku na ekranu. Tako da je trag koji ostavljamo na kanvasu u stvari samo veliki broj veoma sitnih slika.[1]

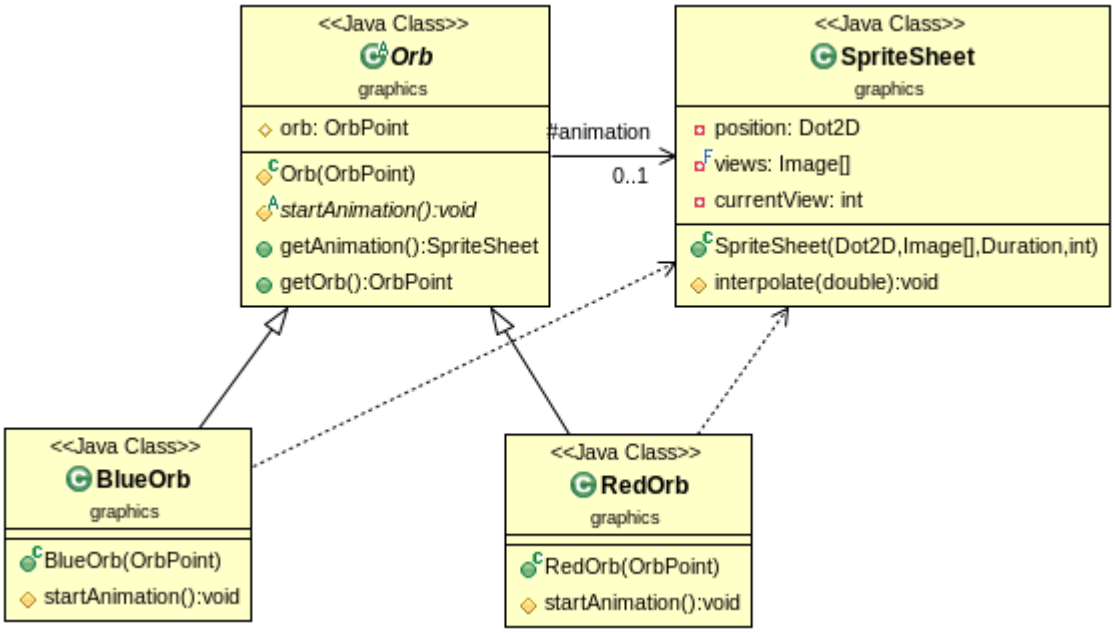
--- Problem2 --- Kako izvesti generisanje I postojanje sfera na platnu?

Problem je rešen uvođenjem posebnih klasa za sfere, tako da je od objekta klase Game, zahtevano samo to da ih generise u Game Loop-u (objekat tipa Timeline, čiji je broj ciklusa postavljen na beskonačnost) I da ih se reši kada su završile se radom. Tako da objekat sfere sam brine o svom izgledu I animaciji kroz svoj životni vek.

-Uvedeno je više klasa koje se bave sferama I to:

- **Orb** – apstraktna klasa koja okuplja osobine svih sfera, a to je da imaju svoju animaciju koja je tipa SpriteSheet, da imaju pristup svojim podacima, kao što su pozicija na kanvasu (kako bi znala gde da se iscrta), vrednost koju nosi (plava sfera menja svoju vrednost nakon nekog perioda), I stanje u kojem se trenutno nalazi (mrtva ili živa).
- **BlueOrb** – bavi se inicijalizacijom I nadgledanjem animacije plavih sfera I pokreće je sa određenim parametrima specifičnim za plavu sferu.
- **RedOrb** - bavi se inicijalizacijom I nadgledanjem animacije crvenih sfera I pokreće je sa određenim parametrima specifičnim za crvenu sferu.

[1] Autor je svestan da je umesto pravljenja cele klase mogao samo da upotrebi metod lineTo(double, double) objekta klase GraphicsContext… ali tako pokazani rezultati nisu bili zadovoljavajući.



9. Dijagram klasa koje čine ponašanje sfere

Pokažimo sada kako izgleda životni vek jedne sfere. U Game Loop-u se 60 puta u sekundi (60 FPS) objekat engine kroz njegov metod pita da li je izgradio novu sferu (jer generisanje sfera radi po algoritmu verovatnoće), ako je engine izgradio novu sferu on vraća jedan objekat tipa *OrbPoint* u kome je između ostalog sadržan tip sfere I njena pozicija na kanvasu. U igri se zatim proverava kojeg je tipa I kreira se odgovarajuća sfera (plava ili crvena) I dodaje se u listu prikazanh sfera - *shownOrbs*. Kroz konstruktor same sfere se inicijalizuje objekat klase *SpriteSheet*, kome se prosleđuje pozicija na kojoj treba da se crta sfera, niz slika koje treba da formiraju animaciju, dužina trajanja jednog ciklusa kao I broj ciklusa. Zatim se pokreće animacija koja zove metod *interpolate(double):void*, koji smenjuje poslate slike I crta ih na kanvasu u određenim vremenskim intervalima, određen broj ciklusa. Na kraju animacije se javlja da je sfera završila sa radom I briše se sa kanvasa.

graphics/RedOrb.java

```
35  @Override
36  protected void startAnimation()
37  {
38      Image[] imagesRed = new Image[4];
39      for (int i=0 ; i<4 ; i++)
40         imagesRed[i] = new Image("images/Orbs/red"+i+".png");
41
42      animation = new SpriteSheet(new Dot2D(orb.getX(), orb.getY()), imagesRed,
    Duration.seconds(1), 10);
43      animation.play();
44
45      animation.setOnFinished(e-> {    // Cistim pravougaonik na kome se nalazila
46          Game.getGc().clearRect(orb.getX()-20, orb.getY()-20, 40, 40);
47          orb.setCurrentState(-1);     // Postavljam da je zavrsila sa radom
48      });
49  }
```

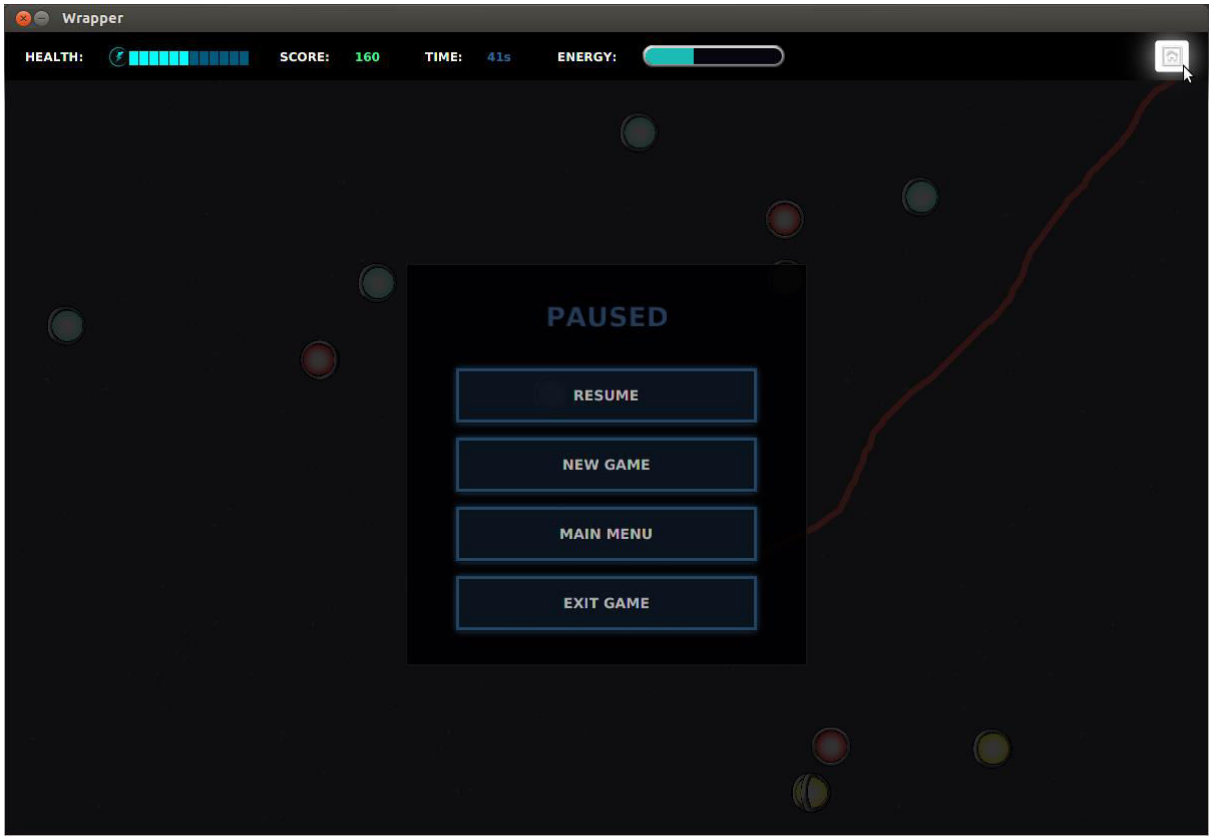
--- Problem3 --- Kako se kontroliše osvežavanje prozora?

Kako se neki događaji mogu desiti jedino kao posledica nekih drugih događaja, nema smisla da osvežavamo ceo prozor u Game Loop-u, jer bi bilo traćenje resursa kada bismo 60 puta u sekundi crtali istu sliku I još pored toga proveravali npr. da li je došlo do kolizije, a da mi nismo ni pokrenuli miša sa mesta.

Zato se prikaz menja sa više mesta, ali su svi oni objedinjeni u grafičkoj klasi *Game*.

- Kroz osluškivače pokreta miša na kanvasu:
  - Poziva se metod *registerMovement(double,double):void,* koji pri registrovanju svake nove validne tačke, čisti kanvas I ponovo iscrtava wrapper-a na pozicijama koje dobija od *engine*-a.
  - Provera kolizije se takodje dešava pri svakom novom pokretu miša, pri čemu engine proverava da li je došlo do nje, pa ako jeste javlja igri da boji novonastali poligon nekoliko koraka.
  - Označavanje uhvaćenih sfera se dešava kao posledica uspešne kolizije, I tada se u engine-u pronalaze sfere koje su se našle u napravljenom poligonu I zatim se kao njihovo stanje stavlja da su uhvaćene (moguća stanja su: živa, istekao tajmer I uhvaćena).
- Kroz GameLoop:
  - Ažurira se labela koja prikazuje vreme koje je proteklo od početka igre.
  - Generišu se nove sfere na već opisani način.
  - Vrši se provera života trenutno prikazanih sfera kroz metod *checkOrbLife():void*. Podaci o sferama se naravno, preuzimaju od engine-a, I ako ima sfera koje su ili zavrsile sa radom ili su uhvaćene, one se sklanjaju sa kanvasa I iz liste sfera I engine ažurira svoje stanje.
  - Ako je bilo mrtvih sfera, onda se ažurira I broj poena prikazanih u label-i, koji se mora promeniti zajedno sa nestankom sfera.
  - Ako je bilo mrtvih sfera, ažurira se I broj života koji su ostali, tj. menja se se prikazani objekat tipa *ImageView*.
  - Ako je bilo mrtvih sfera proverava se I da li je kraj igre.
- Kroz objekte sfera:
  - Vrši se inicijalizacija I pokretanje animacije sfere, kao I njeno iscrtavanje na kanvasu.
  - Smena animacija - npr. kada prodje period plave sfere I počne period žute.
  - Postavljanje novih vrednosti – plava sfera ima vrednost od 10 poena, dok žuta ima vrednost od 30. Objekat sam ažurira svoju novu vrednost na prelasku između animacija.
  - Na kraju animacije, kada istekne tajmer, sam objekat sfere postavlja svoje stanje tako da se zna da mu je istekao tajmer I čeka da ga GameLoop izbriše iz liste.

Za vreme igranja same igre se može pristupiti I mini meniju, koji je funkcionalno veoma sličan onom iz glavnog menija. Novo je jedino to što se njegovim pozivom (klikom dugmeta home ili pritiskom tastera ESC), igra privremeno pauzira. Svi tajmeri za sfere, kao I GameLoop se pauziraju, a preko kanvasa se postavlja jedan Pane objekat koji onemogućava da se crta po njemu.



10. Izgled mini menija

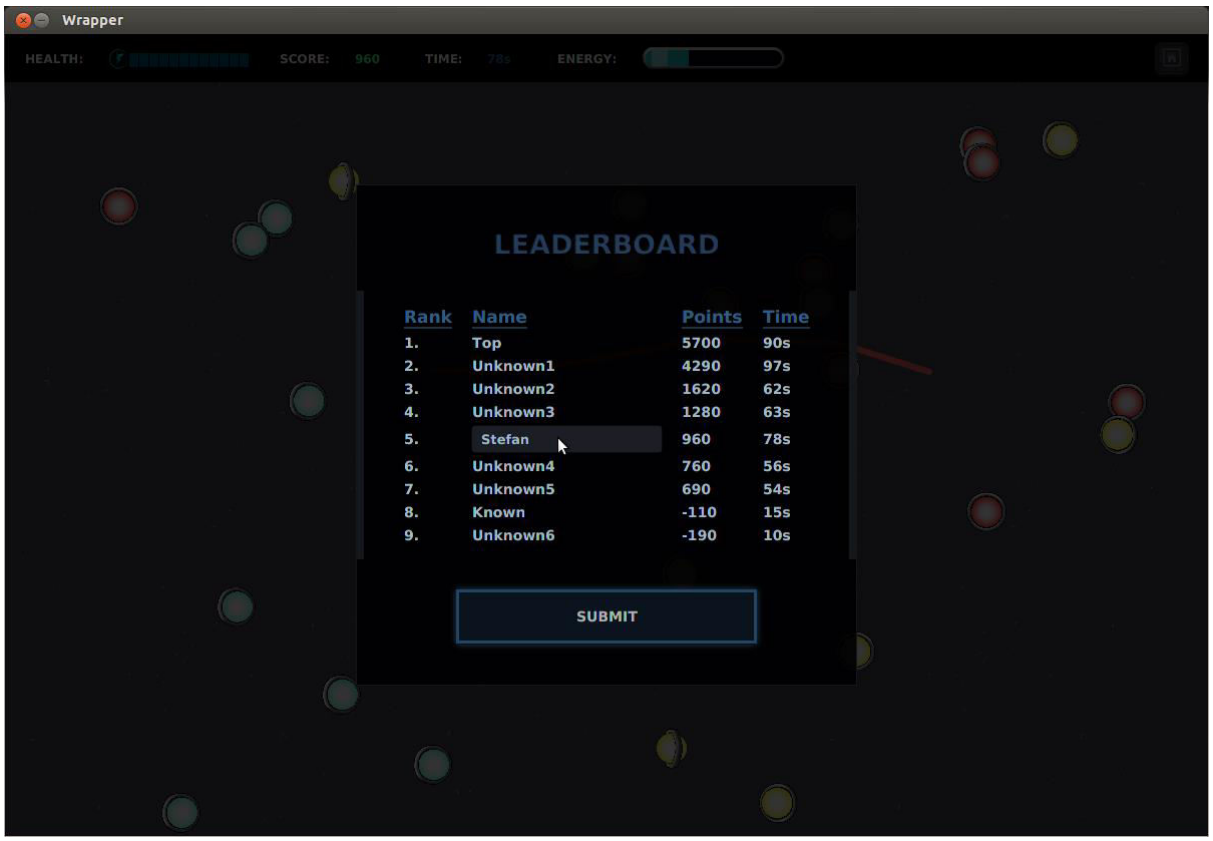
Klikom na dugme RESUME sklanja se Pane pod nazivom *miniMenuOverlayPause*, čime kanvas postaje aktivan I iznova se startuju svi tajmeri.

Klikom na dugme NEW GAME se ponovo inicijalizuju izgled I engine, I to metodom *initializeNewGame():void* I pokreće se nova igra.

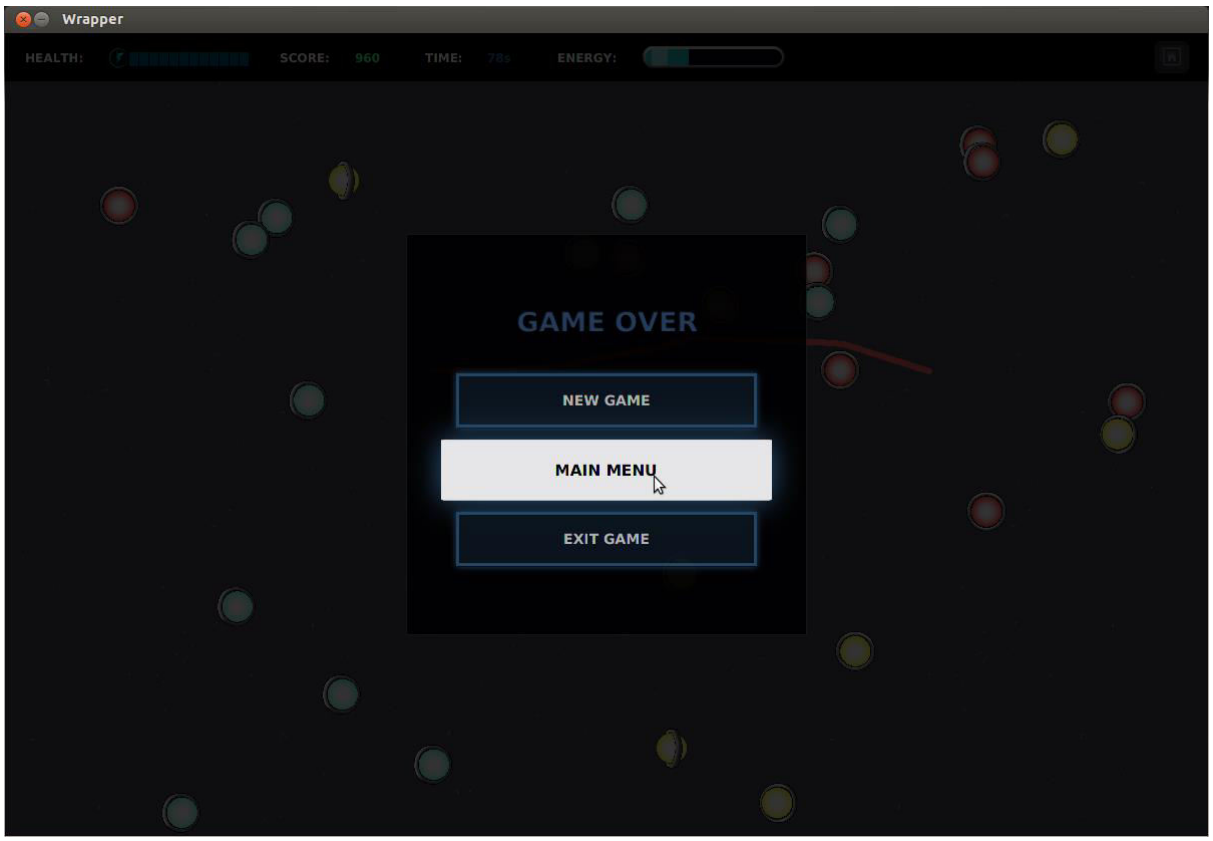
Klikom na dugme MAIN MENU se kao što je već pokazano vrši zamena scena I to promenom vrednosti nadgledanog proprty-a.

Klikom na dugme EXIT GAME se izvršava komanda System.exit(0).

Pored ovog mini menija, postoje još dva koja se prikazuju tek kada igrač izgubi sve živote. Na kraju svake partije se proverava da li je igrač probio neki rekord, ako jeste prikazuje se tabela najboljih učesnika igre, kao I njegov rezultat medju njihovim, u sortiranom redosledu, od prvog do desetog mesta. Tu se traži od igrača da ukuca svoje ime u *TextField*, kako bi bio zapamćen njegov rezultat. Ako igrač ne ukuca svoje ime, biće zapamćen kao *ŧUnknownŧ*. Nakon klika na dugme SUBMIT, cela nova tabela se upisuje u tekstualni fajl, kako bi rezultati sigurno bili sačuvani. Zatim se prikazuje mini meni, ali bez RESUME dugmeta. Ako takmičar ne probije rekord, neće se ni prikazati meni za rekorde, već će odmah dobiti ovaj mini meni.

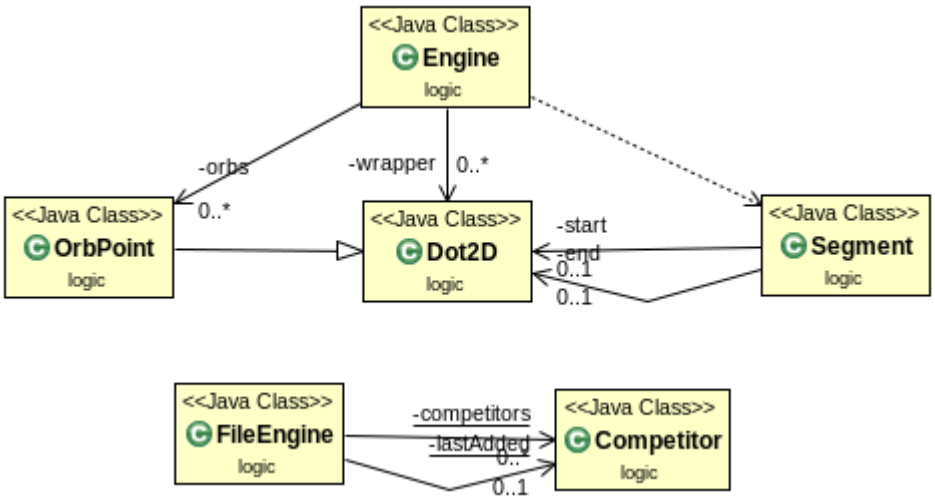


11. Izgled LEADERBOARD mini menija



12. Izgled GAME OVER mini menija

2.2. Logički sloj

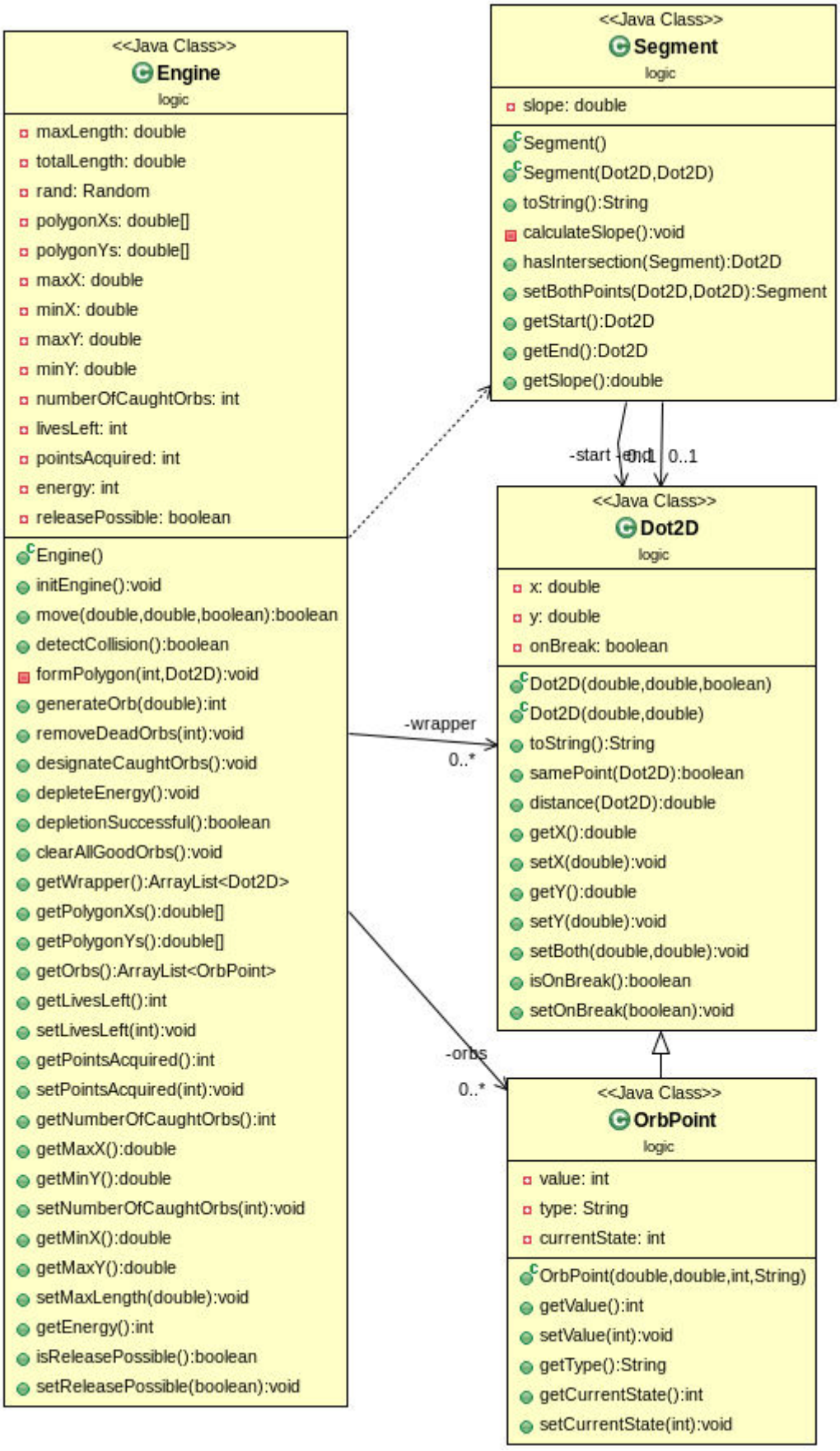


-Na ovom dijagramu se jasno izdvajaju dve logičke celine:

- Klasa Engine, I klase okupljene oko nje koje čine logiku igre.
- Klasa FileEngine koja služi da olakša manipulisanje sadržajem tekstuelne datoteke koja čuva najbolje rekorde.

Zbog kompleksnosti I količine podataka koje ove klase zajedno razmenjuju, u ovom delu će fokus biti na razumevanju kako ovaj sastav kroz algoritme funkcioniše u zajednici kako bi se rešili osnovni logički problemi koje ova aplikacija postavlja.

2.2.1. Game Engine



13. Dijagram koji prikazuje relacije osnovnih logičkih klasa

--- Problem1 --- Kako se logički definiše kretanje wrapper-a?

Glavni metod koji se na ovom nivou bavi **kretanjem** jeste metod *move(double,double,boolean):boolean*. On se poziva iz objekta klase Game, kroz kontrolere za pokretanje miša. Kao argumente dobija X I Y koordinatu trenutne pozicije kursora, kao I informaciju o tome da li je ta tacka nastala posle preseka.

Na šta se misli po presekom? Kako prozor igre nije fullscreen, moguće je kursorom jednostavno napustiti prozor igre, a trag ne može ići dalje od veličina prozora. Tako da kada se kursor vrati u prozor, poslednja tačka pre napuštanja prozora biće povezana sa novom tačkom, a to je nešto što ne želimo da dopustimo, jer bi u slučaju da napustimo prozor sa jedne strane, a vratimo se na sasvim drugoj strani, trag prekoračio svoju maksimalnu dužinu I došlo bi do nagle promene dužine.

Iz tog razloga uvodimo da sve tačke koje su nastale na *mouseEntered,* budu prekidne tačke, koje se neće spojiti sa prethodnom svojom tačkom I tada kao argument šaljemo true.

Ovaj metod, pre nego što sačuva novu tačku, prvo proverava da li je nova tačka ista kao I prethodna, I u tom slučaju je neće sačuvati. Isto, ako nova tačka ima istu X ili Y koordinatu kao I prethodna, neće je sačuvati kao poseban element liste, već će se samo dodati na prethodnu tačku u listi objekata Dot2D – wrapper lista.

Pored dodavanja nove tačke, ovaj metod se bavi I brisanjem stare, kako bi se održala konstantna dužina traga. A kako bi kretanje bilo što prirodnije, ovaj metod skraćuje duge linije na delove I to presecanjem linije na pola I čuvanjem koordinate centra.

Povratna vrednost je boolean, koji govori igri da li je potrebno da izvrši iscrtavanje wrapper-a, što u nekim slučajevima nije potrebno, pa se tada vraća false.

logic/Engine.java

```
50  public boolean move(double x, double y, boolean onBreak)
51  {
52      Dot2D newPoint = new Dot2D(x, y, onBreak);
53      int n = wrapper.size()-1;
54
55      if (n <= 0)
56      {
57              wrapper.add(newPoint);
58              return false;
59      }
60
61      if (wrapper.get(n).samePoint(newPoint)) return false;       // Ista tacka?
62
63      // Da li je X ili Y koordinata nove tacke ista kao kod prethodne?
64      if (wrapper.get(n).getX() == newPoint.getX())
65      {
66              totalLength -= wrapper.get(n).distance(wrapper.get(n-1));
67              wrapper.get(n).setY(newPoint.getY());
68      }
69      else if (wrapper.get(n).getY() == newPoint.getY())
70      {
71              totalLength -= wrapper.get(n).distance(wrapper.get(n-1));
72              wrapper.get(n).setX(newPoint.getX());
73      }
74      else
75      {
76              wrapper.add(newPoint);
77              n++;
78      }
```

```
79
80      totalLength += wrapper.get(n).distance(wrapper.get(n-1));
81
82      while (totalLength > maxLength)
83      {
84          if ((wrapper.get(0).distance(wrapper.get(1)) > 20) &&
    (!(wrapper.get(1).isOnBreak())))
85          {
86              wrapper.get(0).setBoth(ceil((wrapper.get(0).getX() +
    wrapper.get(1).getX()) / 2.0), ceil((wrapper.get(0).getY() +
    wrapper.get(1).getY()) / 2.0));      // Kratim liniju na pola
87              totalLength -= wrapper.get(0).distance(wrapper.get(1));
88          }
89          else
90          {
91              totalLength -= wrapper.get(0).distance(wrapper.get(1));
92              wrapper.remove(0);  //Brisem najstariju liniju koja nije duza od 20px
93          }
94  }
```

--- Problem2 --- Kako se proverava kolizija u ostavljenom tragu?

Ovo je u suštini jednostavan problem I predstavlja proveru preseka između dve duži, tj. između najnovije duži, koja je nastala od najnovije registrovane tačke I njene direktne prethodnice I svih ostalih duži koje se u tom trenutku grade izgled wrapper-a. Naravno provera se ne vrši za duž koju pretposlednja tačka gradi sa svojom prethodnicom I kolizija se ne proverava ako je tačka na preseku, odnosno ako joj je promenljiva isOnBreak == true.

Kako su nam ovde potrebne duži, a mi imamo samo tačke, uvodimo novu klasu **Segment**, koja ima osobine duži, kao I metod za proveru preseka koji vraća presečnu tačku, ako presek postoji, ili null ako ne postoji: *hasIntersection(Segment):Dot2D*.

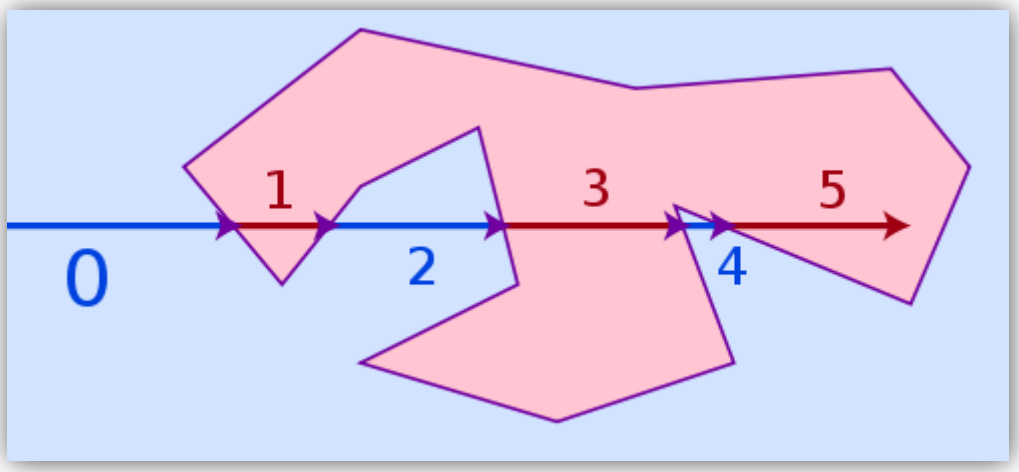
logic/Engine.java

```
95  public boolean detectCollision()
96  {
97      int n = wrapper.size()-1;
98      Segment newSegment = new Segment(wrapper.get(n-1), wrapper.get(n));
99      Segment otherSegment = new Segment();
100
101     int i;
102     for (i=0 ; i<n-2 ; i++)
103     {
104          Dot2D collisionPoint =
    newSegment.hasIntersection(otherSegment.setBothPoints(wrapper.get(i),
    wrapper.get(i+1)));
105
106         if (collisionPoint != null)        // Postoji presecna tacka
107         {
108              formPolygon(i, collisionPoint);     // Sacuvaj tacke poligona
109              return true;    // Doslo je do kolizije, oboji poligon
110         }
111     }
112
113     return false;       // Nema kolizije
114 }
```

--- Problem3 --- Kako se proverava koje sfere su se našle u obuhvaćenom prostoru?

Metod koji se bavi ovom proverom jeste *designateCaughtOrbs():void*. On se poziva jedino ako je došlo do kolizije. Dakle u tom trenutku znamo da su se neke dve prave presekle I znamo sve tačke tog poligona, samo je potrebno da odredimo da li se nešto I šta nalazi u tom prostoru. Ima više algoritama koji se bave proverom da li se odredjena tačka nalazi u poligonu, moj lični izbor bio je **Ray casting algoritam**, zato što može bez ikakvih dodataka da radi sa konveksnim, konkavnim i sa šupljim poligonima.

Ideja je veoma jednostavna, povlačimo zamišljenu duž od neke tačke sasvim van poligona, do tačke za koju želimo da znamo da li pripada poligonu ili ne. I samo izbrojimo broj preseka koje je ta duž postigla sa stranicama poligona. Ako je broj preseka paran, tačka se ne nalazi u poligonu. Ako je broj preseka neparan broj, tačka pripada poligonu.



14. Ray casting algoritam

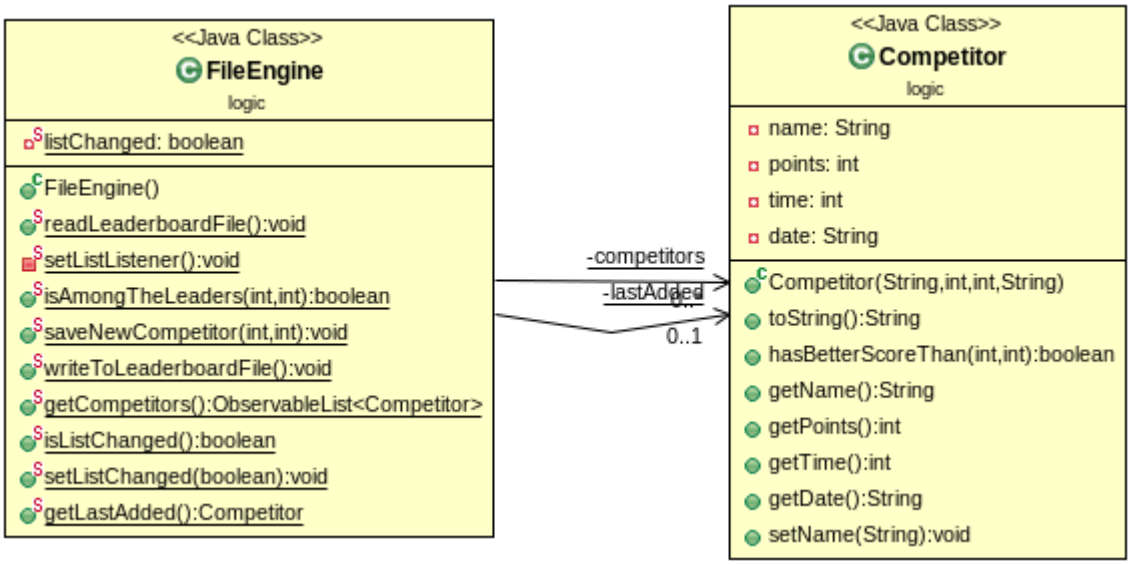
Dakle, ovde je ideja da za svaku sferu, koje čuvamo u listi *orbs*, kao objekte tipa **OrbPoint**, koji u suštini samo sadrže poziciju, tip, stanje I vrednost sfere, proverimo da li pripada poligonu, povlačeći zamišljenu duž od sasvim leve tačke poligona na istoj visini kao I ta sfera - (minX - e, sfera.Y) do koordinata same sfere (sfera.X, sfera.Y) brojeći pritom broj preseka sa poligonom. Promenljiva e će biti 1% širine poligona, kako ne bismo u nekom slučaju počeli sa crtanjem od same strane poligona.

Pored svega ovoga imaćemo I uslove koji će odbaciti neke kandidate ako očigledno nisu ni blizu regiona u kome se poligon nalazi, kao I uslov da proveravamo samo strane poligona koje su levo od naše sfere, jer nema smisla proveravati one koje su desno od nje.

logic/Engine.java

```
115 public void designateCaughtOrbs()
116 {
117     Segment ray = new Segment();
118     Segment polygonSide = new Segment();
119     Dot2D currentOrb = new Dot2D(0,0);
120     double startPadding = ((maxX - minX) / 100.0);  // 1% sirine poligona
121 
122     for (int i=0 ; i<orbs.size() ; i++)
123     {
124             currentOrb.setBoth(orbs.get(i).getX(), orbs.get(i).getY());
125 
126             // Proveravam da li se data sfera ne nalazi u granicnom okviru
127             if (currentOrb.getX() > maxX || currentOrb.getX() < minX ||
    currentOrb.getY() > maxY || currentOrb.getY() < minY) continue;
128 
129             // Ray-casting algorithm
130             // Segment... (minX - padding, p.y) : (p.x, p.y)
131             ray.setBothPoints(new Dot2D(minX - startPadding, currentOrb.getY()),
    currentOrb);
132 
133             int numOfIntersections = 0;
134 
135             for (int j=0 ; j<polygonXs.length-1 ; j++)
136                     if (((polygonXs[j] <= currentOrb.getX()) ||
    (polygonXs[j+1] <= currentOrb.getX())) && (Math.max(polygonYs[j],
    polygonYs[j+1]) >= currentOrb.getY()) && (Math.min(polygonYs[j],
    polygonYs[j+1]) <= currentOrb.getY()))
137             {
138                     if (ray.hasIntersection(polygonSide.setBothPoints(new
    Dot2D(polygonXs[j], polygonYs[j]), new Dot2D(polygonXs[j+1], polygonYs[j+1])))
    != null)
139                     {             // Ako se segment nalazi levo od tacke na kojoj
    se nalazi sfera i ako nije sasvim iznad/ispod zraka i ako se sece sa zrakom...
140                             numOfIntersections++;
141                     }
142             }
143 
144             if (numOfIntersections % 2 != 0)
145                     orbs.get(i).setCurrentState(1);  // Uhvacen
146     }
147 }
```

2.2.2. File Engine



15. Dijagram klasa za upravljanje fajlovima

Smisao klase FileEngine jeste da učitava, čuva I zapisuje podatke o najboljim učesnicima igre. Svaki takmičar je predstavljen jednim objektom klase Competitor, koja sadrži sve njegove podatke: ime, broj poena, vreme igranja, datum I vreme kada je igra odigrana.

Pri prvom pokretanju aplikacije se poziva statička metoda ove klase *readLeaderboardFile():void*, kojom se učitavaju podaci o dosadašnjim takmičarima iz tekstualnog fajla *text/leaderboard.txt*, I smeštaju u statičku listu *competitors*, koja može da se posmatra. Na ovu listu se I postavlja osluškivač na promenu, o kome je već bilo reči u delu za glavni meni. Kako ovu klasu koriste I klasa *Game* I *MainMenu*, sa potrebom za istim podacima, tako su svi atributi I metodi ove klase statički I zato nema potrebe bilo gde instancirati ovu klasu.

Već smo upoznati kako glavni meni koristi ovu klasu, igra je koristi na veoma sličan način uz par dodataka. Na kraju svake partije poziva se metod *isAmongTheLeaders(int,int):boolean*, kojoj se šalju poeni I vreme koje je takmičar postigao, a vraća se true, ako je probio neki rekord, u suprotnom false. Tu se rezultat jednostavno, pomoću metode klase Competitor – hasBetterScoreThan(int,int) poredi sa rezultatom poslednjeg takmičara, ako je lista puna, ako nije, odmah se vraća true.

Ako je takmičar oborio rekord, on se smešta na svoje mesto u listi, metodom saveNewCompetitor(int,int):void. Zatim igra preuzima podatke iz liste takmičara I ispisuje ih, na mestu gde naiđe na null vrednost za ime, stavlja TextField, kako bi takmičar uneo svoje ime. Klikom na SUBMIT dugme se upisuje ime takmičaru, I kompletna lista se, metodom *writeToLeaderboardFile():void*, ispisuje nazad u tekstualni fajl.

2.3. Dizajn aplikacije

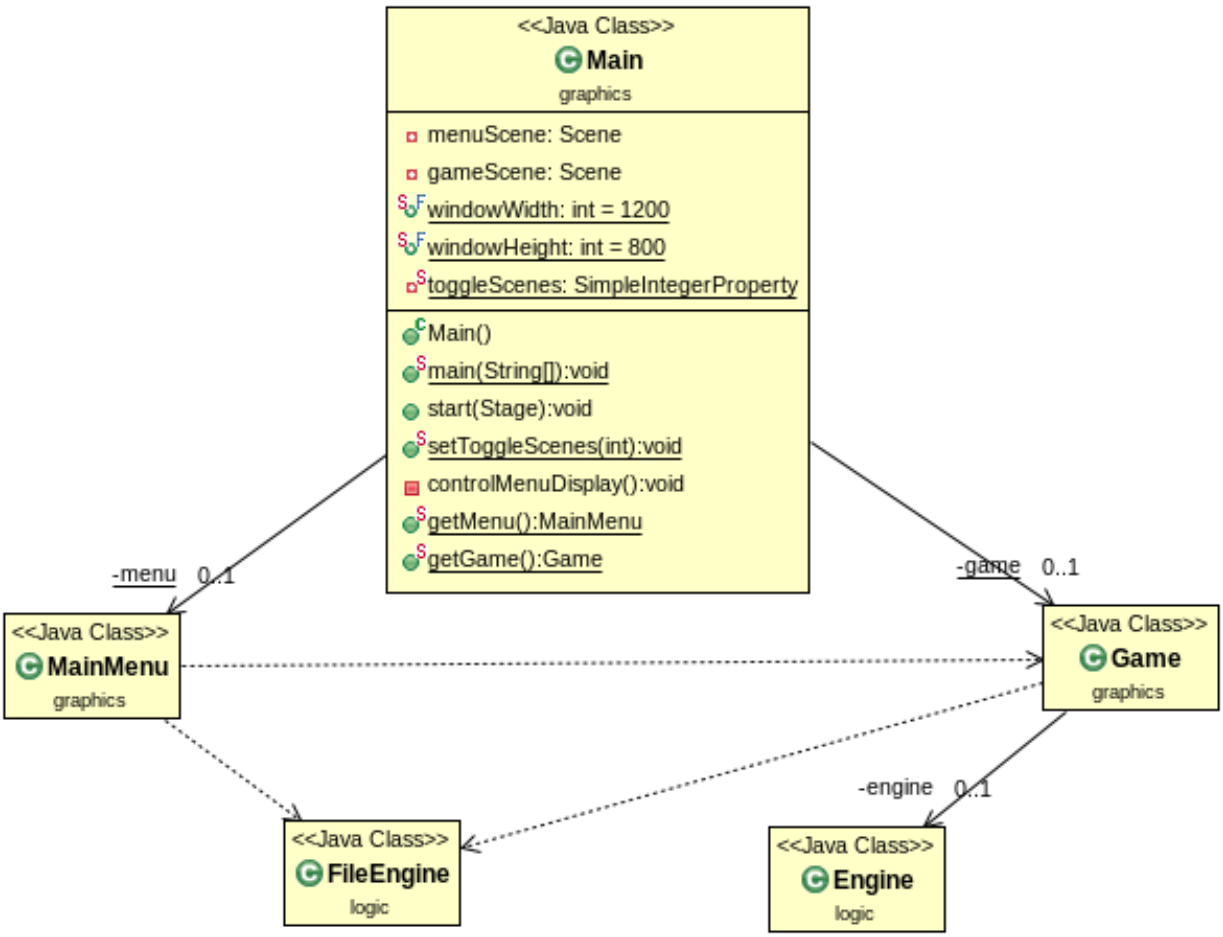
JavaFX omogućava korišćenje CSS-a pri određivanju izgleda vizuelnih komponenti. Upoteba CSS-a u javaFX-u je veoma slična njenoj upotrebi u web tehnologijama. Svodi se na dodeljivanje CSS klasa i ID-eva komponentama I zatim u CSS fajlu navodimo kakav će izgled komponenta sa tom klasom imati. Ovako se prikaz I izgled aplikacije sasvim razdvajaju.

Na primer ako objektu dodamo ID: *miniMenuButton.setId("miniMenuButton");*
U CSS fajlu ga uređujemo sa:

style.css

```
148 #miniMenuButton
149 {
150     -fx-background-image: url("images/Menu/home.png");
151     -fx-background-repeat: no-repeat;
152     -fx-background-position: center;
153     -fx-padding: 5px 15px 5px 15px;
154 }
155
156 #miniMenuButton:hover  // moguce je definisati i pseudoklase
157 {
158     -fx-background-image: url("images/Menu/homeHovered.png");
159 }
```

2.4. Dijagram ključnih klasa aplikacije



16. Dijagram ključnih klasa aplikacije

3. Finalne napomene

-Osnovni sistemski zahtevi:

- Linux ili Windows operativni sistem – aplikacija nije isprobana na Mac operativnim sistemima.
- JRE (Java Runtime Environment) 1.8.0

-Moguća unapređenja:

- Veći izbor nivoa, sa određenim preprekama koje bi naš trag morao da zaobiđe zid na najbolji mogući način.
- Sfere koje mogu da se kreću po prozoru I nose više poena.
- Više 'moći' na koje se može trošiti energija. Da postoji mogućnost da se izabere pre početka igre ili da se izabere opcija Random, koja će davati nasumičnu moć na uspešno iskorišćenu energiju. Neke od ideja jesu magnet, koji privlači dobre, a odbija loše sfere ili moć da za određeni vremenski period nije moguće izgubiti živote ili da se sa ekrana počiste sve loše sfere, bez posledica…
- Uvodjenje I drugačijih kazni za pogrešno pražnjenje energije. Na primer da prikaz na ekranu postane crno-beo na određeni vremenski period I tada igrač može da razlikuje dobre od loših sfera jedino po njihovoj drugačijoj animaciji.
- Nadogradnje, u vidu više života ili bržeg generisanja energije, ako igrač dostigne određeni breakpoint u igri.
- Uvođenje palete tako da igrač može sam da napravi izgled svog wrapper-a, kakav god želi.
- Mogućnost importovanja omiljene muzike sa računara pomoću FileChooser klase I puštanje iste za vreme igranja.
- Da se umesto sa linijama, radi sa krivama, radi lepšeg izgleda traga koji ostavljamo.
- Mogućnost izbora veličine ekrana.
- Uvođenje moći koje se mogu iskoristiti samo jednom za vreme partije, a da se pritom različite moći generišu na različite načine. Na primer različitim kombinacijama levog, desnog klika I scroll-a.

Literatura

[1] https://imi.pmf.kg.ac.rs/moodle/course/view.php?id=33

[2] https://stackoverflow.com/questions/217578/how-can-i-determine-whether-a-2d-point-is-within-a-polygon

[3] https://imi.pmf.kg.ac.rs/moodle/pluginfile.php/8466/mod_resource/content/1/seminarskiOOP%20Jovica.pdf

[4] D. Poo, D. King, S. Ashok, *Object-oriented programming in Java*, Springer-verlag, 2008.

[5] http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#JFXST784

[6] https://gamedevelopment.tutsplus.com/tutorials/introduction-to-javafx-for-game-development--cms-23835

[7] https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html

[8] https://www.tutorialspoint.com/javafx/index.htm

[9] http://www.javafxtutorials.com/

[10] https://www.youtube.com/channel/UCmjXvUa36DjqCJ1zktXVbUA

[11] …