Izrada 2D igrice koristeći programski jezik Java

Tutić, Mario

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Stručni studij

IZRADA 2D IGRICE KORISTEĆI PROGRAMSKI JEZIK JAVA

Završni rad

Mario Tutić

Osijek, 2023.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac Z1S: Obrazac za imenovanje Povjerenstva za završni ispit na preddiplomskom stručnom studiju

Osijek, 11.09.2023.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za završni ispit na preddiplomskom stručnom studiju

Ime i prezime Pristupnika:	Mario Tutić
Studij, smjer:	Stručni prijediplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	AR 4761, 19.07.2019.
OIB Pristupnika:	59359840360
Mentor:	doc. dr. sc. Tomislav Galba
Sumentor:	,
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	prof. dr. sc. Krešimir Nenadić
Član Povjerenstva 1:	doc. dr. sc. Tomislav Galba
Član Povjerenstva 2:	izv. prof. dr. sc. Alfonzo Baumgartner
Naslov završnog rada:	Izrada 2D igrice koristeći programski jezik Java
Znanstvena grana završnog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak završnog rada	Potrebno je proučiti i opisati postojeće biblioteke koje bi omogućile izradu 2D igrice. Zatim je potrebno implementirati jednostavnu 2D igru, opisati dobivene rezultate i korištene tehnologije.
Prijedlog ocjene pismenog dijela ispita (završnog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 2 razina
Datum prijedloga ocjene od strane mentora:	11.09.2023.

Potvrda mentora o predaji konačne verzije rada:	Mentor elektronički potpisao predaju konačne verzije.
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 29.09.2023.

Ime i prezime studenta:	Mario Tutić
Studij:	Stručni prijediplomski studij Računarstvo
Mat. br. studenta, godina upisa:	AR 4761, 19.07.2019.
Turnitin podudaranje [%]:	1

Ovom izjavom izjavljujem da je rad pod nazivom**: Izrada 2D igrice koristeći programski jezik Java**

izrađen pod vodstvom mentora doc. dr. sc. Tomislav Galba

i sumentora ,

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

Razvoj video igara je jedan od najboljih načina kako naučiti programirati zato što se pri razvoju često nailazi na logičke probleme koje je potrebno riješiti i tako pomaže razviti sposobnost "programerskog" načina razmišljanja. Također je vrlo zanimljiv pristup jer vizualno možemo vidjeti utjecaj "svake" nove linije koda, bilo to dodavanjem novog grafičkog detalja ili na primjer kroz "pametnije" odluke neprijatelja. Uz sve navedeno veliki dio funkcionalnosti koji se koristi za razvoj igara također je u nekom obliku primjenjiv i u drugim poljima softverskog inženjeringa.

Ideja završnog rada je napraviti 2D igricu u Javi bez korištenja programskog okvira za razvoj igara(*engl. game engine*) ili nekakvih vanjskih biblioteka, odnosno, koristeći samo Java ugrađene klase. Ovakav pristup izrade video igrice se ne koristi u današnje vrijeme, ali daje najbolji uvid kako video igrice funkcioniraju "ispod haube", odnosno, kakva se logika koristi. Kroz projekt izrade igrice bit će korišteni i objašnjeni razni temeljni koncepti od koji se većina igara sastoji kao što su petlja igrice, mapa, prihvaćanje unosa, interakcija između entiteta itd. Cilj ovog završnog rada je funkcionalna igrica, koja rješava nekakve standardne probleme klasične igrice ovakvog tipa. Rad se može ugrubo podijeliti na dvije cjeline. Prva bi bila grafički prikaz mape i animacija karaktera, a drugi dio logika iza grafičkog prikaza kao što je ograničavanje prolaska entiteta kroz zidove i detekcija napada igrača. Kao rezultat završnog rada se očekuje funkcionalna 2D igrica i razumijevanje općenitih mehanizama rada 2D igrice.

1.1. Zadatak završnog rada

Napraviti 2D igricu koristeći Java programski jezik bez uporabe programskih okvira(*engl. Framework*). Cilj igrice je da igrač što dulje preživi dok ga napadaju neprijatelji. Igrica se sastoji od mape, igrača i neprijatelja te koristi osnovne mehanike kao što je kretanje igrača, detekcija kolizije, kretanje kamere.

2. PREGLED PODRUČJA TEME

Ova tema spada u područje razvoja igara te se na prvu čini teško pronaći znanstvene dosege ili primjenu u praksi i stvarnom svijetu zato što su igrice “svijet za sebe”. Iako igrice spadaju u industriju zabave bez nekakvih praktično korisnih primjena u stvarnom svijetu, tržište video igara je do sada imalo konstantan rast i u 2022. godini je ostvarilo 220.79 milijardi dolara. Neke prognoze signaliziraju da bi do 2030. godine taj broj mogao dosegnuti i do 583 milijarde dolara, što svakako signalizira kako ova industrija ima svijetlu budućnost. Osim rasta tržišta i kvaliteta konstantno raste kako grafički tako i oni ne vizualni dijelovi kao što su npr. logika karaktera.

Projekt koji je rađen u ovom završnom radu koristi pločice za grafički prikaz. Super Mario Bros i Pacman neki su od primjera koji koriste ovu tehnologiju. Kroz ovaj projekt mogu se vidjeti problemi koji se susreću u razvoju, a te iste probleme potrebno je riješiti gotovo u svakoj današnjoj igrici. Tako se kroz razvoj nailazilo na probleme kao što su kolizija, animacija, petlja igrice itd.

Na kraju je važno napomenuti da bi mnoge funkcionalnosti i algoritmi korišteni u razvoju video igara mogli pronaći svoju primjenu i u stvarnome svijetu. Na primjer igrice koje umrežuju više računala i razmjenjuju podatke međusobno, algoritmi za traženje najbržeg puta, sigurnost itd.

3. POSTAVLJANJE PROJEKTA

Za početak je bilo potrebno postaviti i organizirati projekt na temelju čega se gradi ostatak projekta. Neke od temeljnih stvari su petlja igrice i prozor u kojem se iscrtava(*engl. Rendering*) grafika. Proces postavljanja projekta kod izrade ovakvog tipa igrice temeljenog na pločicama vrlo je uobičajen te se može pratiti kao popis stvari koji se mora napraviti. Može se reći da postoje dvije temeljne klase, Main i GamePanel.

Main klasa:

Kao i kod svakog programa tako i ova igrica zahtijeva glavnu klasu koja je startna točka izvođenja. U ovoj klasi je kreiran JFrame objekt koji je prozor igrice te su također određene početne postavke za prozor kao što su ime i njegova pozicija.

GamePanel klasa:

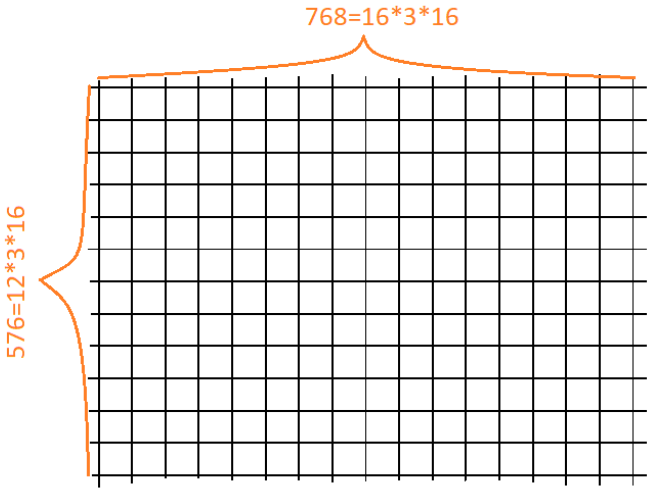
Ova klasa nasljeđuje JPanel. Ona predstavlja prozor igrice odnosno prostor na zaslonu na kojem se prikazuje igrica. Također određuje postavke zaslona koje će biti temelj i utjecati na cijeli daljnji razvoj ovog projekta.

Postavljene su sljedeće varijable/postavke:

-Veličina pločica(*engl. Tiles*): pločice se mogu smatrati temeljnim gradivnim blokovima grafike ove igrice te je njihova veličina u ovom projektu postavljena na 16*16 piksela. Veličina pločica je proizvoljna te ona može biti i veća npr. 32, ali budući da veliki broj retro igara koristi 16*16 lakše je pronaći već gotove sličice na internetu ili ih nacrtati jer su vrlo male. Također ovim odabirom dobiven je vrlo zanimljivi retro dizajn i bit će sasvim dovoljno za prikaz onoga što je potrebno za ovaj projekt.

-Varijabla za skaliranje: veličina slike od 16*16 piksela bi bila vrlo malena kada bi se prikazala na modernom zaslonu velike rezolucije, što nije bio problem u prošlosti. Iz tog razloga varijabla za skaliranje je postavljena na 3 što bi značilo da će svaka pločica(*engl.tile*) kada se prikaže biti uvećana 3 puta odnosno iznosit će 48*48 piksela. Ovakva praksa skaliranja je uobičajena pri izradi retro igara.

-Veličina prozora igrice je postavljena na 16*12 pločica. Pločice su temeljni blokovi i zbog tog razloga se veličina mjeri u pločicama. Budući da je pločica 16*16 veličine i da su uvećane za 3 puta dobivaju se sljedeće dimenzije prozora: širina=16*3*16=768 piksela i visina=16*3*12=576 piksela.



Sl. 2.1. Prozor igrice

Na kraju GamePanel klasa koja nasljeđuje JPanel se dodaje u JFrame objekt koji se nalazi u Main klasi kroz metodu add. Ovime je postavljen prozor i "panel" na kojem će se iscrtavati grafika. Kao što je već rečeno ovaj proces je generičan te nema potrebe za detaljnim razumijevanjem klasa korištenih za prikaz prozora. Najbitnije je razumjeti da ovakva implementacija daje mogućnost i prostor na kojem će se iscrtavati grafika igrice.

3.1. Veza niti i petlje igrice

Petlja igrice je osnovna mehanika, a najvažniji koncept igrice je postojanje vremena. Sve što se događa u prozoru igrice, odvija se u ovisnosti o vremenu kao što je na primjer pomicanje igrača, kretnje neprijatelja, mehanika pucanja itd. Svi ti događaji animiraju se pomoću niza statičkih sličica. One se izmjenjuju vrlo brzo dajući privid kao da je pokret stvaran, a ne da se u biti prikazuju statičke slike. Ako igrica ima 60 FPS to znači da se u sekundi izmjeni 60 sličica. Ovaj koncept zvuči vrlo jednostavno jer su se gotovo svi u prošlosti susretali s njim u drugim situacijama, ali je vrlo bitan i ako se ne implementira

ispravno igrica jednostavno ne bi radila ispravno. Primjer neispravnog grafičkog prikaza u ovisnosti o vremenu bi bio kada bi se statičke slike izmjenjivale prebrzo.

Za početak kako bi se pravilno moglo pratiti kako vrijeme u igrici prolazi i koliko puta u sekundi se ona ažurira potrebno je kreirati posebnu nit. Ako je igrica jedan proces koji se izvodi na računalu, nit možemo smatrati kao potproces te iste igrice. Kreiranjem projekta automatski već postoji jedna nit u kojoj je pokrenut program, ali dodaje se još jedna koja će "rukovoditi" samom igricom. Iako posebna nit za petlju igrice nije potrebna za jednostavnije projekte i u ovom slučaju projekt bi se vrlo vjerojatno mogao realizirati i bez nje, ona se preporučuje. U slučaju kada program izvodi puno procesiranja i kada se pri izvođenju osjeti loša osjetljivost na ulazne komande, npr. kašnjenje grafičkog odaziva pomicanja igrača tada je potrebna posebna petlja. Ukratko programeri koriste posebnu nit kako bi odvojili "teško" procesiranje podataka od interaktivnosti grafičkog prikaza.

Implementacija niti je također generičan proces kao i implementacija prozora i panela stoga će također biti objašnjen samo redoslijed implementacije bez objašnjavanja ugrađenih Java klasa koje se koriste.
Implementacija niti:

-Kreiranje objekta Thread klase, ovaj objekt je u biti nit i potrebno ju je pokrenuti.

-GamePanel klasa nasljeđuje Runnable interface te se automatski kreira i run metoda koju je potrebno prepisati. Metoda run se automatski izvršava u zasebnoj niti kada je nit pokrenuta te ona predstavlja srž cijelog projekta jer kao što je rečeno posebna nit će se koristiti za izvršavanje igrice, u ovoj metodi će biti napisana petlja igrice.

-Inicijalizira se Thread objekt odnosno poziva njegov konstruktor i prosljeđuje mu se GamePanel klasa koja implementira Runnable sučelje.

-Pokretanje niti u Main klasi.

3.2. Petlja igrice

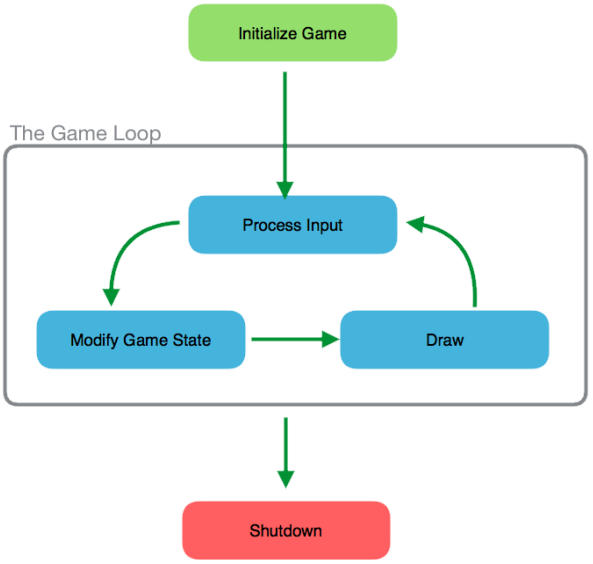
Unutar GamePanel klase nalazi se run metoda koja je naslijeđena implementiranjem Runnable klase te ju je potrebno prepisati. Run metoda će se izvesti pokretanjem niti u GamePanel klasi te je u njoj potrebno implementirati petlju igrice. Petlja igrice se metaforički može opisati kao srce igrice jer ona diktira koliko brzo se ažuriraju podaci kao što su

koordinate objekata te koliko brzo će se osvježavati njihov prikaz na zaslonu i ona se iznova izvršava tijekom cijelog životnog ciklusa rada igrice.

Unutar petlje pozivaju se dvije glavne metode:

-update za osvježavanje informacija o objektima poput igrača i neprijatelja

-repaint metodu koja osvježava grafički prikaz



Sl. 3.1. Shema petlje igrice

```
9   public void run(){
10      while(gameThread!=null){
11          update();
12          repaint();
13      }
14  }
15  public void update(){
16      //ažuriraj podatke objekata u igrici
17  }
18  public void paintComponent(Graphics g){
19      super.paintComponent(g);
20  }
```

Sl. 3.2. Pojednostavljena petlja igrice

Kod koji se vidi na slici 3.2. je pojednostavljen, s beskonačnom while petljom bez izračuna FPS-a i bez koda u update i paintComponent metodama, ali daje dobar prikaz kako petlja igrice funkcionira.

Metoda za crtanje grafike paintComponent je ugrađena Java metoda koja služi za crtanje po panelu odnosno JPanel-u. Na slici 3.2. se vidi da joj je proslijeđen Graphics parametar i unutar nje pozvana paintComponent metoda roditeljske klase koja nasljeđuje JPanel. Također ju pozivamo kao repaint metodu unutar same petlje što je malo zbunjujuće. Ovakva praksa je zahtijevana od strane Jave kako bi sve funkcioniralo kako treba i kako bi na kraju imali mogućnost crtanja na JPanel. Detaljno objašnjenje bi nadolazilo opseg ovog završnog rada te se više o načinu rada može pronaći u službenim Java dokumentima.

Na kraju preostaje pretvoriti beskonačnu while petlju u petlju koja svaki 60-ti dio sekunde poziva update i repaint metode. Petlja se temelji na matematičkom izračunu vremena koje je prošlo od zadnjeg poziva i koliko vremena treba proći do sljedećeg poziva te na temelju toga izvođenje prelazi u stanje "spavanja" do novog prolaska kroz petlju. Petlja koja je korištena je takozvana "spavajuća petlja", iako ona nema u potpunosti točne izračune, ta odstupanja neće biti vidljiva. Također zbog korištenja sleep metode možemo uštedjeti na procesorskim resursima iako je ovo vrlo jednostavna igrica.

```
85          while(gameThread!=null) {
86              update();
87              repaint();
88              try {
89                  double remainingTime=nextDrawTime-System.nanoTime();
90                  remainingTime=remainingTime/1000000;//because sleep accepts millis
91                  if(remainingTime<0) {
92                      remainingTime=0;
93                  }
94                  Thread.sleep((long)remainingTime);
95                  nextDrawTime+=drawInterval;
96              } catch (InterruptedException e) {
97                  // TODO Auto-generated catch block
98                  e.printStackTrace();
99              }
00          }
```

Sl. 3.3. Kod petlje igrice

3.3. Očitavanje unosa

Sastavni dio svake igrice je očitavanje unosa kako bi igrica bila interaktivna. Kreirana je nova klasa KeyHandler koja implementira KeyListener sučelje koje služi za očitavanje događaja na tipkovnici. KeyListener sučelje zahtijeva implementaciju tri metode od kojih se u ovom slučaju koriste dvije, a to je keyPressed koja se poziva kada je tipka na tipkovnici pritisnuta i keyReleased koja se poziva kada otpustimo tipku na tipkovnici.

```
16      //GamePanel kalsa
17      KeyHandler KeyH=new KeyHandler();
18      public void update(){
19          if(KeyH.upPressed==true){
20              playerY-=playerSpeed;
21          }
22      }
```

Sl. 3.4. GamePanel klasa

```
28  //KeyHandler Klasa
29  public boolean upPressed;
30
31  public void keyPressed(KeyEvent e){
32      int code=e.getKeyCode();
33      if(code==KeyEvent.VK_W){
34          upPressed=true;
35      }
36  }
37  public void keyReleased(KeyEvent e){
38      int code=e.getKeyCode();
39      if(code==KeyEvent.VK_W){
40          upPressed=false;
41      }
42  }
```

Sl. 3.5. KeyHandler klasa

Na slikama 3.4. i 3.5. se vidi primjer kako bi izgledala implementacija za pomicanje igrača prema gore, a ista logika se koristi i za ostale smjerove. KeyPressed metoda se izvodi samo onda kada je tipka pritisnuta prvi put te zbog toga je dodana kontrolna boolean varijabla koja ima vrijednost true sve dok se tipka drži pritisnutom. Iako za sada nema igrača niti bilo kakvih drugih objekata, svaki od njih će imati svoje koordinate unutar prozora. Kod u primjeru sa slike 3.5. mijenja koordinate igrača kada se tipka za kretanje drži pritisnutom, za npr. 4 piksela prema gore. Ako se tipka za smijer gore drži pritisnutom jednu sekundu, a unutar update metode ona se poziva 60 puta u sekundi to bi značilo da se u sekundi igrač pomaknuo 240 piksela prema gore. Iz ovog primjera možemo vidjeti važnost kontrole vremenskog izvođenja update i repaint metoda odnosno zaključavanja broja izvođenja ovih metoda u vremenu.

3.4. Korisničko sučelje

Igrica će imati nekoliko mogućih stanja npr. nova igra, izgubljena igra, pauza itd., a korisničko sučelje će omogućavati kretanje kroz ta stanja.

U GamePanel klasi određena su stanja u kojima se igrica može nalaziti, a to su:
stanje menija(menuState), stanje igranja(playState), stanje pauze(pauseState), stanje izgubljene igre(gameOverState), inicijalizacija ovih varijabli je prikazana na slici 3.6.

```
public final int menuState=0;
public final int playState=1;
public final int pauseState=2;
public final int gameOverState=3;
```

Sl. 3.6. Stanja igrice

Svako od stanja prikazuje određeno sučelje na zaslonu kao što se vidi na primjerima slika 3.7.-3.10.

Sl. 3.7. Stanje menija

Sl. 3.8. Stanje igranja

Sl. 8.9. Stanje pauze

Sl. 3.10. Stanje izgubljene igre

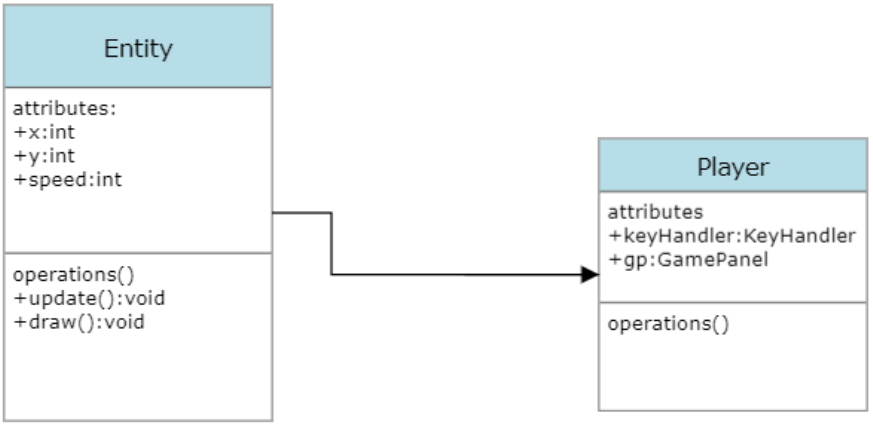
Sučelje je kreirano pomoću Graphics2D ugrađene Java klase. Pomoću tipki ‘w’ i ‘s’ možemo pomicati strelicu gore dolje, a tipkom enter ovisno o poziciji strelice mijenja se stanje igrice.

4. PRIKAZ IGRAČA, SVIJETA IGRICE I KRETANJE KAMERE

Svi elementi osim sučelja koji su do sada objašnjeni, rade u pozadini, a u nastavku je objašnjen dio projekta koji će biti vidljiv korisnicima tijekom stanja igranja. Igrač je glavni lik u igrici s kojim će se biti moguće kretati po mapi, a budući da mapa izlazi izvan okvira prozora programa, potrebno je implementirati kameru koja će ga pratiti kada se on kreće.

4.1. Prikaz igrača

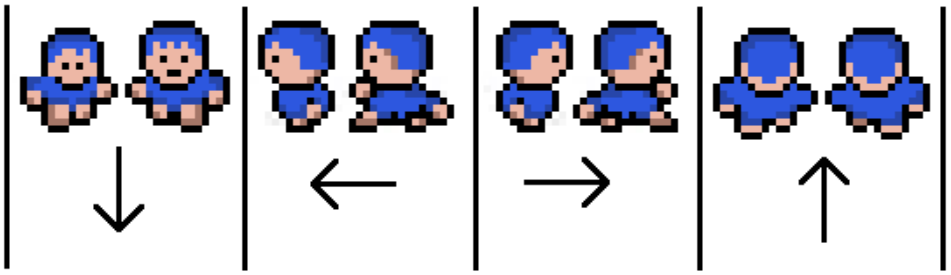
Kreiran je poseban paket Entity kako bi se grupirale međusobno povezane klase. Unutar Entity paketa kreirana je Entity klasa koja će sadržavati varijable kao što su koordinati i brzina koje će nasljeđivati svi ostali karakteri u igrici. Kreirana je klasa Player koja nasljeđuje Entity te za sada ima referencu na GamePanel i KeyHandler klasu. Razlog tome je što GamePanel sadrži temeljne podatke o igrici kao npr. širina prozora kojima će se često pristupati i također sadrži referencu na KeyHandler klasu jer će se Player objekt kretati ovisno o pritisnutim tipkama w,a,s i d. Na slici 4.1. je UML klasni dijagram na kojem se vidi nasljeđivanje Entity klase i kompozicija KeyHandler i GamePanel klase. Također se vide i dvije vrlo bitne update i draw metode koje će se provlačiti kroz sve objekte i pozivati na kraju u update i repaint metodama GamePanel klase, ovakvom strukturom kod je modularan. Ovo nije konačna struktura i s vremenom će se dodavati varijable i metode kao i neke stvari mijenjati kada bude potrebno jer je ne moguće istovremeno graditi “krov i temelje”.



Sl. 4.1. Odnos Entity i Player klase

Nakon inicijalizacije Player klase slijedi implementacija update i draw metoda. Update metoda u ovom poglavlju nije objašnjavana u detalje jer se u njoj za sada procesira samo kretanje igrača pomoću tipki, što je objašnjeno u poglavlju 3.3.

Draw metoda se koristi za "crtanje" na prozoru igrice. Kao što je ranije objašnjeno pokreti su u biti brza izmjena statičkih sličica te će za svaki od 4 smjera biti 2 sličice koje će se naizmjenično mijenjati kako bi se dobio dojam pokreta.



Sl. 4.2. Animacija glavnog lika

Implementacija koda:

-Kreiranje res foldera za slike, ali i sve buduće resurse koji će se koristiti. Također kreiran je i player paket u kojem će biti samo slike glavnog lika.

-Kreiranje varijabli tipa BufferdImage za svaku pojedinu sliku u Entity klasi kako bi ih i svi budući entiteti mogli naslijediti.

-Slike iz res foldera je potrebno učitati i spremiti ih u varijablu. Napisana je metoda getPlayerImage koja pomoću ImageIO.read metode učitava svih osam slika i dodjeljuje vrijednost varijablama kako bi se kasnije mogle prikazati. Ova metoda se poziva u konstruktoru Player klase kao inicijalizacija. Ovaj isti proces će se ponavljati više puta kroz izradu projekta.

```
public void getPlayerImage() {
    try {
        up1=ImageIO.read(getClass().getResourceAsStream("/player/boy_up_1.png"));
        up2=ImageIO.read(getClass().getResourceAsStream("/player/boy_up_2.png"));
        down1=ImageIO.read(getClass().getResourceAsStream("/player/boy_down_1.png"));
        down2=ImageIO.read(getClass().getResourceAsStream("/player/boy_down_2.png"));
        left1=ImageIO.read(getClass().getResourceAsStream("/player/boy_left_1.png"));
        left2=ImageIO.read(getClass().getResourceAsStream("/player/boy_left_2.png"));
        right1=ImageIO.read(getClass().getResourceAsStream("/player/boy_right_1.png"));
        right2=ImageIO.read(getClass().getResourceAsStream("/player/boy_right_2.png"));
    }catch(IOException e){
        e.printStackTrace();
    }
}
```

Sl. 4.3. getPlayerImage metoda za učitavanje slika

Nakon što su slike učitane, na temelju smjera u kojem se igrač kreće prikazuje se određena slika. Ovaj problem je riješen tako da je u Player klasi dodana varijabla direction tipa String koja na temelju smjera poprima određenu vrijednost koja se kroz switch uvjet ispituje i ovisno o uvjetu koji je zadovoljen iscrtava određenu sliku.

Ovo nije potpuno rješenje te ostaje problem naizmjenične izmjene slika. Problem je riješen tako da je dodana varijabla koja svakih 10 prolaza kroz update metodu odnosno svakih 0.1666 sekundi mijenja svoju vrijednost u 1 ili 2 gdje broj 1 označava jednu a broj 2 drugu sliku. Ako se igrica osvježava 60 puta u sekundi, a svakih 10 prolaza kroz update metodu se mijenja slika koja se prikazuje dobije se da je 60- dio sekunde puta 10 približno 0.1666.

```
spriteCounter++;
if(spriteCounter>10) {
    if(spriteNumber==1) {
        spriteNumber=2;
    }
    else if(spriteNumber==2) {
        spriteNumber=1;
    }
    spriteCounter=0;
}
```

Sl. 4.4. Izmjena slika

```
public void draw(Graphics2D g2) {
    BufferedImage image = null;
    if(attacking ==false) {
        switch(direction) {
        case "up":
            if(spriteNumber==1)
                image = up1;
            else
                image=up2;
            break;
        case "down":
            if(spriteNumber==1)
                image = down1;
            else
                image=down2;
            break;
        case "left":
            if(spriteNumber==1)
                image = left1;
            else
                image=left2;
            break;
        case "right":
            if(spriteNumber==1)
                image = right1;
            else
                image=right2;
            break;
        }
        g2.drawImage(image,screenX,screenY,gp.tileSize,gp.tileSize,null);
```

Sl. 4.5. Crtanje slike u ovisnosti o smjeru i brojaču slike

Za prikaz slike korištena je ugrađena Java Graphics2D klasa i njena metoda drawImage.

4.2. Stvaranje i učitavanje mape

Cilj ovog poglavlja je napraviti svijet po kojem će se igrač kretati. Iako već postoji prostor po kojem se on može kretati, cilj je napraviti ga ljepšim te umjesto crne pozadine označiti gdje su granice svijeta/mape te također dodati zidiće. U inicijalizaciji projekta objašnjeno je kako su pločice “temeljni gradivni” elementi grafike te da su veličine 16*16 tako da će se pozadina i svi ostali elementi sastojati od sličica te veličine. U res folderu je

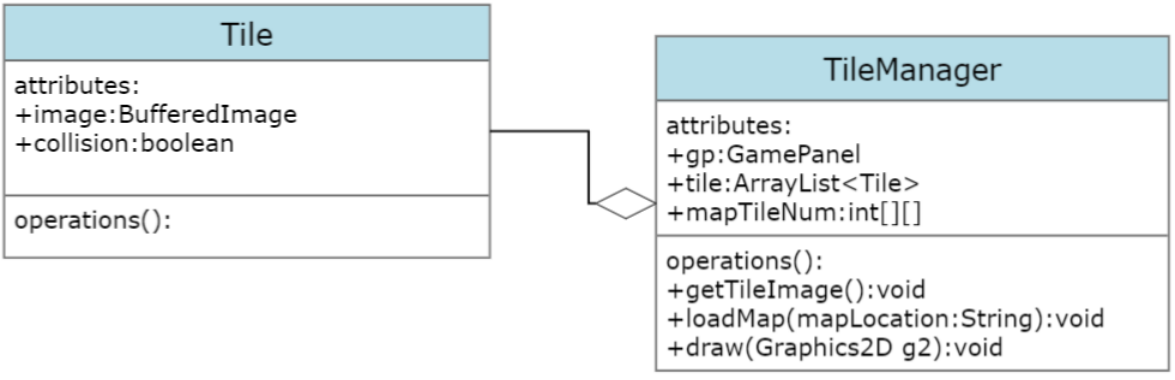
kreiran novi paket tiles u kojemu će se nalaziti sličice, a na slici 4.6. se vidi primjer kako one mogu izgledati.

Sl. 4.6. Slike pločica

Implementacija:

-Kreiran je paket tile te klasa Tile koja reprezentira jednu pločicu. Svaka pločica mora imati svoju sliku i boolean varijablu koja određuje dali pločica ima uključenu koliziju ili ne. Na primjer zid će imati uključenu koliziju i igrač neće imati mogućnost prolaziti kroz zidove.

-Kreiranje TileManager klase koja će učitavati pločice i učitavati mapu kao tekstualnu datoteku te na temelju podataka u mapi na zaslon crtati određenu pločicu to jest sličicu.



Sl. 4.7. UML dijagram odnosa Tile i TileManager klase

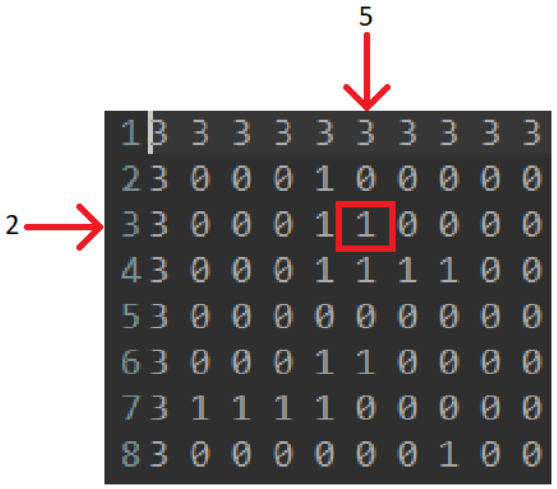
Na slici 4.7. se vide metode i operacije koje te dvije klase sadrže. U nastavku su dodatno objašnjene metode i atributi TileManager klase.

Metoda getTileImage učitava sve slike iz res/tiles paketa i sprema ih u image varijablu klase Tile.

```
tile[1]= new Tile();
tile[1].image=ImageIO.read(getClass().getResourceAsStream("/tiles/wall.png"));
tile[1].collision=true;
```

Sl. 4.8. Primjer učitavanja slike zida

Atribut mapTileNum služi za učitavanje tekstualne datoteke u kojoj je zapisan prostorni raspored pločica.



Sl. 4.9. Tekstualna datoteka u kojoj je zapisan raspored slika

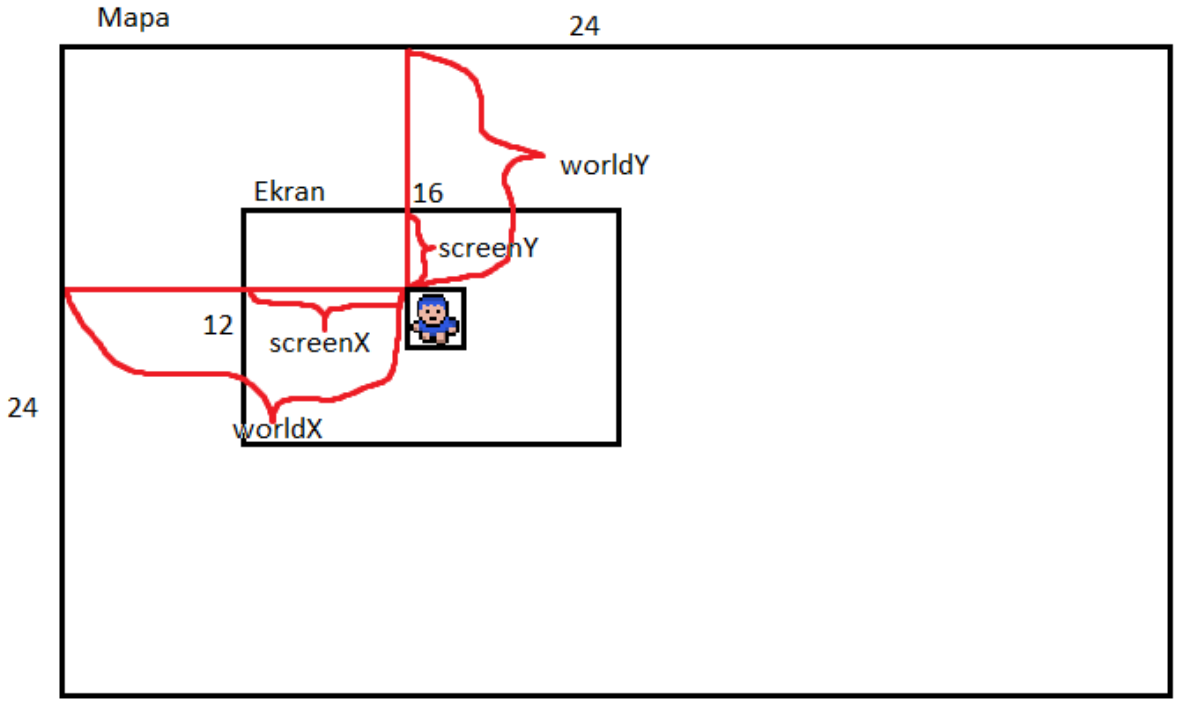
Slika 4.9. prikazuje primjer tekstualne datoteke u kojoj se broj u crvenom kvadratiću nalazi na poziciji x=5 i y=2 i ima vrijednost 1. X i y translacijski određuju poziciju u igrici, a vrijednost određuje koja će se slika prikazati npr. 0 označuje travu, a 1 označuje zid. Važno je reći kako broj stupaca i redaka u tekstualnoj datoteci predstavlja veličinu svijeta, a ne prozora igrice koji je veličine 16*12 pločica. U sljedećem poglavlju će se implementirati kamera koja će pratiti igrača kada izlazi izvan okvira dijela mape koji je trenutno prikazan u prozoru. Metoda loadMap ne radi ništa drugo osim iščitavanja tekstualne datoteke i zapisivanje njenih vrijednosti istim redoslijedom u mapTileNum dvodimenzionalni niz.

Za kraj preostaje prikazati mapu na zaslonu u draw metodi. Budući da je mapa pohranjena u 2D niz, potrebna je petlja u petlji kako bi se prošlo kroz dvodimenzionalni niz i u ovisnosti o vrijednosti svakog od elementa na zaslonu iscrtala određena slika.

4.3. Kretanje kamere u skladu s pokretima igrača

U slučaju da se za igricu koristi mapa koja je iste veličine kao i prozor igrice tada nema potrebe za implementiranjem pokreta kamere. Budući da će mapa u ovoj igrici biti veća od veličine prozora koji iznosi 16*12 pločica bit će potrebno implementirati pokret kamere. Cilj kamere je da se igrač cijelo vrijeme nalazi u fokusu odnosno da se nalazi na sredini prozora igrice.

Iz tog razloga varijable y i x koje se nalaze u Entity klasi su zamijenjene sa worldY i worldX varijablama te one nisu vrijednosti koje označuju poziciju na prozoru igrice već na mapi. U Player klasi su dodane varijable screenX i screenY koje zamjenjuju stare varijable x i y te one označuju poziciju gdje se igrač nalazi na prozoru.



Sl. 4.10. Odnos mape i prozora igrice

Na skici 4.10. se može vidjeti razlika između varijabli. Također može se primijetiti kako su screenX i screenY pola visine i širine prozora s time da se u obzir mora uzeti kompenzacija veličine sličice igrača koja iznosi 16*16 piksela kako bi se ona iscrtavala točno na sredini budući da drawImage metoda kao početnu poziciju crtanja slike uzima gornji lijevi kut.

Pozicija igrača unutar prozora je uvijek ista odnosno igrač će uvijek biti na sredini stoga su one deklarirane kao final varijable.

Sada kada se igrač prikazuje u sredini sve je spremno za implementiranje kamere. Najbitnije informacije pri crtanju mape su koja sličica se mora prikazati što je već implementirano te također na kojem mjestu se treba prikazati u odnosu na igrača. U draw metodi klase TileManager je dodan kod koji računa koji dio mape se prikazuje unutar prozora.

```
screenX=worldX-gp.player.worldX+gp.player.screenX;
screenY=worldY-gp.player.worldY+gp.player.screenY;
```

Sl. 4.11. Određivanje koordinata u odnosu na poziciju igrača

Na slici 4.11. se može vidjeti "krucijalni" dio koda odnosno formula koja računa poziciju unutar prozora gdje su varijable: screenX - pozicija pločice u odnosu na sredinu prozora

worldX - pozicija pločice na mapi

gp.player.worldX - pozicija igrača na mapi

gp.player.screenX - pozicija igrača u prozoru

Također dodan je kod koji osigurava da se nepotrebni dio mape ne crta radi uštede resursa, ovo je riješeno s if uvjetom.

```
if( worldX+gp.tileSize>gp.player.worldX-gp.player.screenX&&
    worldX-gp.tileSize<gp.player.worldX+gp.player.screenX&&
    worldY+gp.tileSize>gp.player.worldY-gp.player.screenY&&
    worldY-gp.tileSize<gp.player.worldY+gp.player.screenY)
{
    g2.drawImage(tile[tileType].image,screenX,screenY,gp.tileSize,gp.tileSize,null);

}
```

Sl. 4.12. Ispitivanje nalazi li se pločica unutar prozora

4.4. Detektiranje kolizije

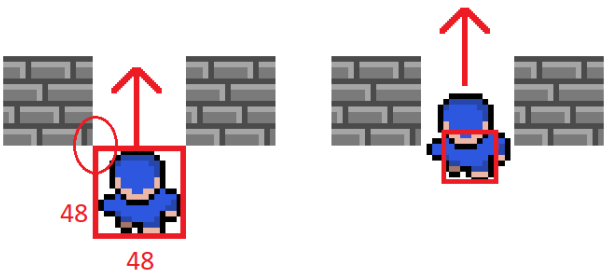
Detekcija kolizije u ovoj igrici je vrlo bitan segment. Koliziju će biti potrebno detektirati u više slučajeva nego što se to na prvu čini, a to su kolizija igrača s neprijateljem, te obratno neprijatelja s igračem, neprijatelja sa zidom, tijekom napada igrača na neprijatelja. U ovom potpoglavlju bit će objašnjena kolizija igrača s pločicama, a sve ostale kolizije radit će na isti način s manjim promjenama i bit će objašnjene u poglavljima koja će slijediti.

Neke od pločica kao što je zid imat će uključenu koliziju odnosno igrač neće biti u mogućnosti "proći kroz njih". U Tile klasi već ranije je kreirana boolean varijabla collision koja upravo određuje dali je kolizija uključena ili ne.

```
tile[1]= new Tile();
tile[1].image=ImageIO.read(getClass().getResourceAsStream("/tiles/wall.png"));
tile[1].collision=true;
```

Sl. 4.13. Uključena kolizija za sliku zida

Za svaki objekt u igrici kojemu se testira kolizija mora imati određenu površinu na kojoj će se testirati kolizija. Na primjer za igrača bi bilo logično da se kao površina gleda obris igrača, ali za potrebe ove igrice to bi bilo prekompleksno za izvesti stoga će se kao površina kolizija gledati imaginarni pravokutnik. Pravokutnik je nešto manjih dimenzija nego sama sličica igrača jer ako bi dimenzija bila 48*48 piksela tada bi nastao problem prilikom prolaska igrača između dva zida jer u tom slučaju igrač bi morao biti "namješten točno na piksel" kako bi mogao proći kroz prolaz.



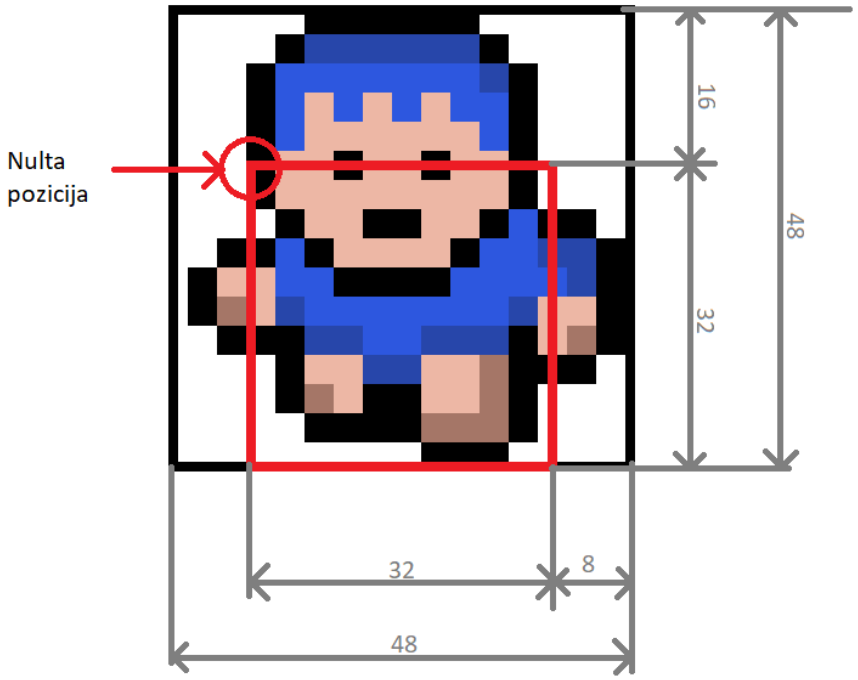
Sl. 4.14. Problem površine kolizije

Implementacija pravokutnika za koliziju:

-U Entity klasi kreirana je varijabla solidArea tipa Rectangle klase koja je dio java.awt API-ja. Rectangle klasa predstavlja pravokutnik i većinom se koristi za prosljeđivanje draw metodi zbog crtanja na zaslon, ali u ovom slučaju će se koristiti kao zamišljeni pravokutnik koji će biti nevidljiv i služit će za testiranje kolizije.

-Također u Entity klasi dodana je boolean varijabla collisionOn koja je inicijalizirana na false, a kada je Entity objekt u koliziji s drugim objektima varijabla će se postavljati na vrijednost true.

-U Player klasi instanciran je objekt tipa Rectangle i referenca je dodijeljena solidArea varijabli. Konstruktor Rectangle klase prima 4 parametra a to su koordinati x i y, te visina i širina. Pravokutnik će se nalaziti u sredini te kao što je objašnjeno iz "tehničkih" razloga on je manjih dimenzija od 48*48.



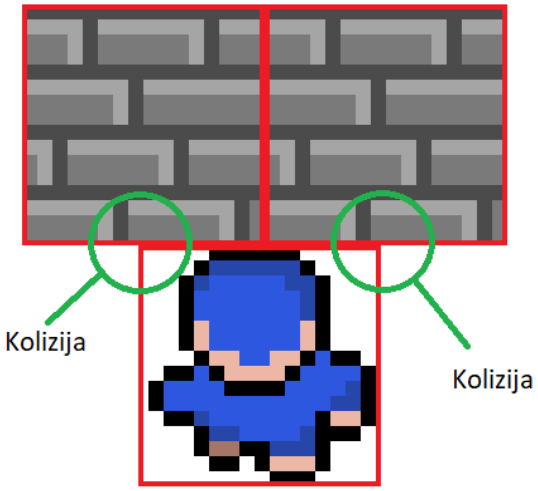
Sl. 4.15. Odnos pravokutnika za koliziju i slike igrača

Na temelju dimenzija sa slike 4.15. pravokutnik je inicijaliziran na sljedeće vrijednosti: x=8, y=16, height=32, width=32

Implementacija klase za koliziju:

-Kreirana je CollisionChecker klasa koja će sadržavati metode za sve vrste kolizija u nastavku projekta

-Prilikom ispitivanja kolizije potrebno je raditi provjeru samo u onom smjeru u kojem se igrač kreće. U nastavku će biti objašnjen princip rada za smjer kretanja prema gore, a isto se može primijeniti i za ostale smjerove.



Sl. 4.16. Očitavanje kolizije na dvije pločice

U bilo kojem smjeru se igrač kretao uvijek će biti potrebno ispitati dvije pločice kao što vidimo na primjeru sa slike 4.16. kada se on kreće prema gore.

-Računanje funkcija koje omeđuju pravokutnik za koliziju:

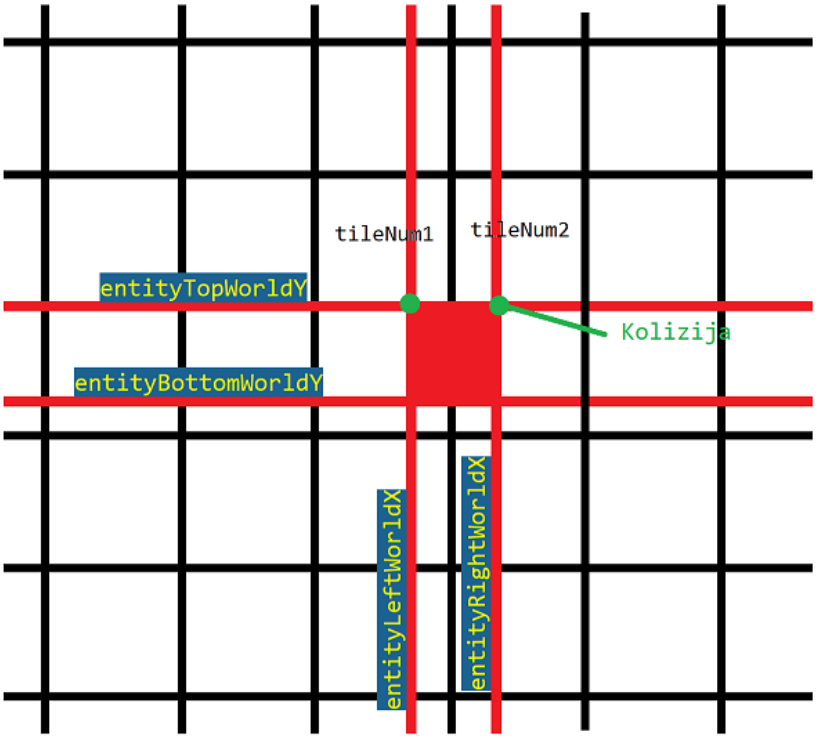
Cilj je pronaći dvije susjedne pločice prema kojima se igrač kreće. Iz koordinata igrača na mapi i podacima o pravokutniku za koliziju pronađene su funkcije osi koje omeđuju pravokutnik za koliziju. Kada se vrijednost funkcije osi podijeli s veličinom pločice dobije se redak ili stupac koji ta određena funkcija siječe.

```
//in pixels
int entityLeftWorldX=entity.worldX+8;//coordinate of solid area in
int entityRightWorldX=entity.worldX+8+entity.solidArea.width;//coor
int entityTopWorldY=entity.worldY+16;
int entityBottomWorldY=entity.worldY+16+entity.solidArea.height;
//in tiles 48*48
int entityLeftCol=entityLeftWorldX/gp.tileSize;//this number tells
int entityRightCol=entityRightWorldX/gp.tileSize;
int entityTopRow=entityTopWorldY/gp.tileSize;
int entityBottomRow=entityBottomWorldY/gp.tileSize;

/*this are possible 2 tiles player can touch when moving in whateve
 * direction one with left and one with right "shoulder" for exampl
int tileNum1;
int tileNum2;
```

Sl. 4.17. Računanje pozicije susjednih pločica

Na slici 4.17. se vide varijable i kod za računanje spomenutog, gdje su prve četiri varijable iznos funkcija u pikselima a druge četiri predstavljaju redak ili stupac. varijable tileNum1 i tileNum2 predstavljaju dvije pločice s kojima se kolizija mora provjeravati. Slika 4.18. grafički prikazuje objašnjeno odnosno što koja varijabla predstavlja.



Sl. 4.18. Traženje funkcija koje omeđuju pravokutnik za koliziju

-Algoritam koji provjerava dali je došlo do kolizije je nešto kompleksniji dio implementacije, a njegova glavna zadaća je da boolean varijablu collisionOn postavi na true ako pločica koju igrač dodiruje ima varijablu collision postavljenu na true. Pravilnije bi bilo reći da ovaj kod predviđa koliziju jer baš kao i stvarnom životu zid neće zaustaviti udarac rukom kad je ruka već unutar zida. Iz tog razloga, ali i zbog same implementacije igrice odnosno petlje igrice u formuli se koristi brzina igrača. Kod na slici 4.19. je za primjer kretanja prema gore, a analogno je za sve ostale smjerove.

```
switch(entity.direction) {
case "up":
    entityTopRow=(entityTopWorldY-entity.speed)/gp.tileSize;
    tileNum1=gp.tileManager.mapTileNum[entityLeftCol][entityTopRow];
    tileNum2=gp.tileManager.mapTileNum[entityRightCol][entityTopRow];
    if(gp.tileManager.tile[tileNum1].collision||gp.tileManager.tile[tileNum2].collision) {
        entity.collisionOn=true;
    }
    break;
```

Sl. 4.19. Kod za ispitivanje kolizije prema gore

-Za kraj preostaje instanciranje CollisionChecker-a u GamePanel klasi te pozivanje metode za provjeru koliziju unutar update metode Player klase.

5. IMPLEMENTACIJA NEPRIJATELJA

Neprijatelji dodaje smisao ovoj igrici jer će cilj igrača biti preživjeti što dulje dok ga neprijatelji napadaju. Baš kao i Player klasa i neprijatelj će nasljeđivati Entity klasu budući da će imati svoje koordinate na mapi, brzinu kojom će se kretati, slike koje će se koristiti u animaciji pokreta, te površinu za koliziju. Veći dio implementacije je sličan implementaciji igrača zbog toga neki dijelovi koji su slični nisu posebno objašnjavani u nastavku.

5.1 Kreiranje neprijatelja

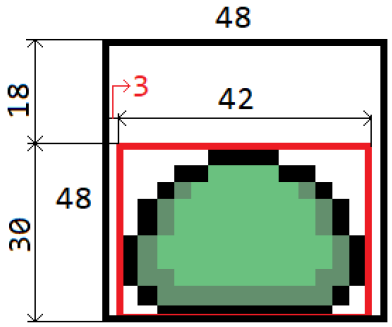
Početna inicijalizacija je slična kao i kod igrača stoga su ukratko samo nabrojani zadaci koji su odrađeni:

-Stvoren je paket enemy u kojem će biti klase svih neprijatelja, iako ovaj projekt ima samo jedan tip neprijatelja. Unutar enemy paketa kreirana je klasa MON_GreenSlime koja nasljeđuje Entity.

-Kreiran je folder u mapi res koji će sadržavati slike za animaciju. U ovom slučaju koriste se samo dvije slike za sve smjerove kretanja.

Sl. 5.1. Animacija neprijatelja

-Inicijalizirana je površina kolizije koja je nešto drugačijih dimenzija nego kao kod igrača.



Sl. 5.2. Površina kolizije neprijatelja

-Učitane su slike iz foldera sa slikama u program

-Potrebno je implementirati ažuriranje neprijatelja odnosno kretanje i ovo je dio koda koji se razlikuje od igrača budući da će se neprijatelj pomoću A* algoritma za traženje staze automatski kretati prema igraču. A* algoritam bit će detaljnije objašnjen u posebnom poglavlju.

-Kreiran je niz tipa klase MON_GreenSlime u GamePanel klasi. Ovaj niz je potrebno inicijalizirati te je za potrebe inicijalizacije objekata u igrici i bolje organizacije koda kreirana posebna klasa AssetSetter. Ova klasa će se s vremenom zbog balansiranja igre vjerojatno mijenjati, ali primjer se može vidjeti na slici 6.3. gdje su kreirana tri neprijatelja u metodi setEnemy. Metoda setEnemy se poziva u GamePanel klasi unutar metode setupGame, a ona se ponovno poziva u Main klasi koja predstavlja glavnu nit.

```
public class AssetSetter {
    GamePanel gp;
    public AssetSetter(GamePanel gp){
        this.gp = gp;
    }
    public void setEnemy() {
        gp.enemy[0]=new MON_GreenSlime(gp);
        gp.enemy[1]=new MON_GreenSlime(gp);
        gp.enemy[1].worldY=184;
        gp.enemy[2]=new MON_GreenSlime(gp);
        gp.enemy[2].worldY=230;
    }
}
```

Sl. 5.3. AssetSetter klasa

5.2. Kolizija između entiteta

Prvi put igrica sada sadrži više entiteta i potrebno je testirati postoji li kolizija između dva entiteta. Postoje dva posebna slučaja koja su vrlo slična i potrebno je ispitati oba, a to su kada neprijatelj dotakne igrača i kada igrač dotakne neprijatelja.

Kolizija entiteta s igračem:

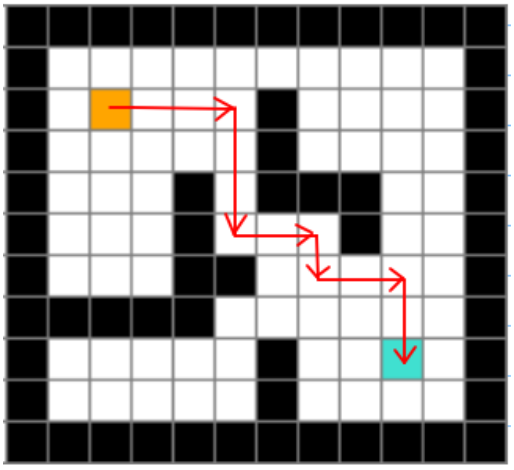
Budući da svaki entitet ima svoju površinu kolizije odnosno sadrži klasu Rectangle bit će jednostavnije testirati koliziju jer Java ima ugrađenu metodu intersects koja je dio klase Rectangle. Metoda za testiranje kolizije entiteta s igračem je nazvana checkPlayer i napisana je unutar klase CollisonChecker. Glavni zadatak ove metode je postaviti collisionOn varijablu na true, ako se dva pravokutnika preklapaju. Metoda checkPlayer ima povratnu vrijednost

tipa boolean i ako je ona true to bi značilo da entitet odnosno neprijatelj dodiruje igrača i tada se u posebnoj metodi igračev život smanjuje. Ova metoda se poziva unutar update metode Entity klase te baš kao i kod igrača, ako je collisionOn postavljen na true on se ne može više kretati sve dok se igrač ne pomakne ili neprijatelj ne promijeni smjer.

Drugi slučaj je kolizija igrača s drugim entitetom te je ista implementacija potrebna i sa strane igrača. Ako se neprijatelj nađe na putu igrača tada se on više ne može kretati u istom smjeru, a i život mu se u tom slučaju također oduzima. Za provjeru ovog slučaja koji je vrlo sličan prethodnom napisana je posebna metoda jer ipak postoje neke razlike. Na primjer kada provjeravamo koliziju entiteta s igračem potrebno je provjeriti samo jednog igrača, a u ovom slučaju provjeravamo cijeli niz neprijatelja. Također ova metoda vraća indeks neprijatelja koji će se koristiti pri napadu igrača za detekciju neprijatelja kojeg je igrač uspješno udario. Ovu metodu potrebno je pozvati unutar update metode klase Player te ograničiti kretanje.

5.3. Ažuriranje i iscrtavanje

Neprijatelj će za kretanje koristiti A* algoritam za traženje puta do igrača. Princip rada A* algoritma neće biti objašnjen, ali će ukratko biti objašnjena njegova primjena u projektu. U projektu je kreirana klasa PathFinder koja implementira A* i u sebi sadrži metodu search koja prima koordinate pozicije neprijatelja i igrača te na temelju njih ona nam vraća listu čvorova. Lista čvorova je ubiti lista koja sadrži stazu sastavljenu od čvorova po kojoj se neprijatelj treba kretati da bi došao do igrača. Čvor predstavlja jednu pločicu i neprijatelj se kreće po čvorovima odnosno pločicama da dođe igrača. Na slici 5.4. se može vidjeti početna točka(žuto), cilj(plavo) i lista čvorova(crvena linija).



Sl. 5.4. A* algoritam

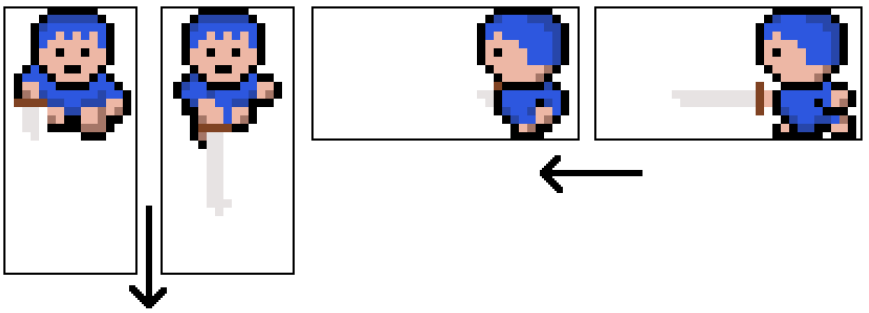
Kada se izračuna lista čvorova odnosno staza kretanja koristeći A* algoritam, potrebno je na temelju staze odrediti smjer kretanja unutar update metode za ažuriranje i implementacija je gotova. Update i draw metode su implementirane unutar klase Entity kako bi ih i neki budući entiteti/neprijatelji mogli naslijediti. Draw metoda je ista kao i kod igrača, ali važno je napomenuti kako sada postoji niz entiteta i potrebno je pomoću petlje proći kroz svaki entitet i pozvati update i draw metodu. Kao i za igrača update i draw metode se pozivaju unutar GamePanel klase.

6. SUSTAV BORBE

Sada kada u igrici postoje neprijatelji koji napadaju igrača i oduzimaju mu život kada ga dotaknu, na redu je da se implementira i napad igrača na neprijatelja. Napad igrača je nešto kompliciraniji te će se u trenutku napada prikazati posebna animacija igrača s mačem i bit će potrebno postaviti posebnu površinu za detekciju kolizije koja će obuhvaćati sliku mača. U ovom poglavlju također će se implementirati i grafički prikaz života igrača u gornjem lijevom kutu.

6.1. Animacija napada

Za animaciju napada korišten je novi set slika s po dvije slike za svaki smjer. Slike su dodane u res/player mapu te učitane unutar Player klase pomoću posebne metode za učitavanje slika. Za razliku od svih prethodnih slika koje su bile veličine 16*16 piksela sad su slike veličine 16*32 ili 32*16 ovisno o smjeru. Slika 6.1. prikazuje primjere za animaciju napada prema dolje i lijevo.



Sl. 6.1. Animacija napada

Zbog specifične veličine slika prilikom crtanja stvara se problem pomaka slike te je potrebno namjestiti nulte koordinate od koji kreće crtanje. Problem se može vidjeti na slici 6.2. gdje zbog veće širine slike napada i iste nulte točke crtanja dolazi do pomaka igrača u desno.

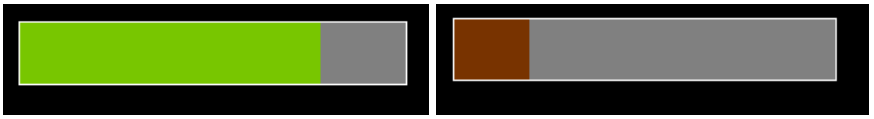
Sl. 6.2. Problem animacije napada

Problem je riješen tako da se u draw metodi nulta pozicija u primjeru sa slike 6.2. pomiče za 48 piksela ili jednu pločicu u lijevo. Ovaj problem se događa samo kod napada lijevo i prema gore.

Sl. 6.3. Rješenje problema animacija napada

6.2. Prikaz života igrača

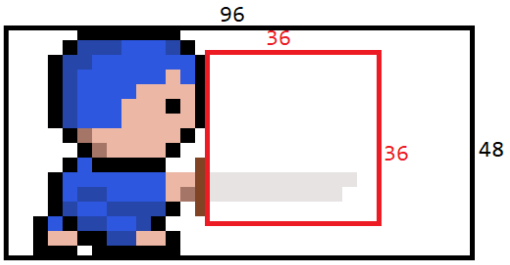
Pri koliziji igrača i neprijatelja igraču se oduzima život stoga je potrebno i prikazati koliko života je igraču preostalo. Kreirana je klasa HUD(engl. *head-up display*) unutar main paketa koja će sadržavati metode za prikaz i ažuriranje života. Kako izgleda prikaz života vidi se na slici 6.4. , a cijeli dizajn je kreiran pomoću Graphics Java klase crtajući pravokutnike. Možemo vidjeti da se sastoji od bijelog obruba, sivo obojenog pravokutnika i pravokutnika koji ovisno o preostalom životu igrača mijenja svoju veličinu i boju.



Sl. 6.4. Prikaz života igrača

6.3. Detekcija udarca

Kada se prisne tipka enter animira se udarac, ali neprijatelju se ne oduzima život. Za detekciju udarca potrebno je kreirati posebnu površinu kolizije koja će biti nešto veća u odnosu na površinu koja se koristila do sada. Iako je slika veličine 32*16 ili 16*32, a nakon skaliranja 96*48 odnosno 48*96, površina za detekciju udarca bit će veličine 36*36. Razlog ovome je što se za detekciju udarca uzima samo dio slike gdje se nalazi mač.



Sl. 6.5. Površina za detekciju udarca

Metoda za detekciju kolizije već postoji i ona je korištena ranije, to je metoda checkEntity klase CollisionChecker. Problem je što ona koristi staru površinu za detekciju kolizije stoga je kreiran novi objekt klase Rectangle koji će služiti kao površina detekcije udarca. U kodu na slici 6.6. se vidi implementacija metode attacking koja se poziva kada je enter pritisnut unutar update metode Player klase.

Implementacija koda na slici 6.6. redom radi sljedeće:

-Spremanje veličine i nultih koordinata površine u privremene varijable kako bi se vrijednosti mogle kasnije vratiti.

-Ovisno o smjeru postavljaju se nove vrijednosti koordinata na mapi na kojima se nalazi igrač.

-Postavljaju se nove vrijednosti visine i širine površine za koliziju.

-Poziva se metoda checkEntity s novom površinom za koliziju.

-Vraćanje starih vrijednosti.

```
//Save the current worldX,worldY, solidArea
int currentWorldX=worldX;
int currentWorldY=worldY;
int solidAreaWidth=solidArea.width;
int solidAreaHeight=solidArea.height;

//adjust players world X/Y for the attackArea
switch(direction) {
    case "up":worldY-=attackArea.height; break;
    case "down":worldY+=attackArea.height; break;
    case "left":worldX-=attackArea.width; break;
    case "right":worldX+=attackArea.width; break;
}
//attackArea becomes solidArea
solidArea.width=attackArea.width;
solidArea.height=attackArea.height;
monsterIndex=gp.collisionChecker.checkEntity(this, gp.enemy);
//restoring original data to solid area of player
worldX=currentWorldX;
worldY=currentWorldY;
solidArea.width=solidAreaWidth;
solidArea.height=solidAreaHeight;
```

Sl. 6.6. Kod detekcije udarca

Ako metoda checkEntity vrati indeks koji nije 999 tada se neprijatelju s tim indeksom oduzima život. Metoda checkEntity vraća indeks prvog neprijatelja s kojim se dogodi kolizija što znači da igrač ne može istovremeno zadati udarac dvama ili više neprijatelja.

6.4. Balansiranje igrice

Balansiranje igrice nije neophodan dio ovog projekta, ali radi bolje igrivosti i doživljaja potrebno je izbalansirati igricu. Cilj igrice je da igrač preživi vremenski što duže u svijetu u kojem ga napadaju neprijatelji. Balansiranje bi značilo namještanje određenih varijabli kao što su život igrača ili neprijatelja tako da igrica ima smisla. Na primjer šteta koju jedan udarac igrača zadaje je postavljena na 1/4 ukupnog života neprijatelja tako da trebaju 4 udarca da ga se uništi. Potrebno je balansirati i brzinu kretanja karaktera, smisla bi imalo da igrač bude nešto brži od neprijatelja da ima realne šanse za preživljavanje. Važno je razmišljati i o ponovnom stvaranju neprijatelja kada su uništeni.

7. ZAKLJUČAK

Cilj završnog rada bio je napraviti funkcionalnu igricu u kojoj je cilj da igrač preživi što duže dok ga napadaju neprijatelji. Projekt koristi pločice za stvaranje mape i grafički prikaz elemenata te je implementirana petlja igrice koja diktira ažuriranje i iscrtavanje(*engl. Rendering*). Ideja je bila da mapa bude većih dimenzija od veličine prozora i da sadrži elemente poput zidova tako da je neophodno bilo implementirati pokret kamere i koliziju. Iz teme možemo zaključiti da igrica sadrži dva karaktera a to su igrač i neprijatelj. Budući da je cilj igrača da preživi što duže logično je bilo dodati napad igrača kako bi mogao uništiti neprijatelje, a za implementaciju bilo je potrebno napisati novu metodu kolizije te napraviti posebnu logiku za animaciju kada se događa napad. Kao i u svakom programu dodano je korisničko sučelje kako bi se igrica mogla pauzirat, ugasiti ju ili pokrenuti novu kada igrač bude poražen.

Projekt ima još dosta prostora za napredak kako bi se igranje napravilo zanimljivijem. Neke od ideja su dodavanje različitih vrsta neprijatelja, mogućnosti mijenjanja oružja na primjer puška ili spremanje rezultata u tekstualnu datoteku gdje bi trebalo dodati i nekakvu sigurnost od mijenjanja te datoteke. Igrica je sada gotova i “igriva” i podosta elemenata bi se moglo dodati na do sada napravljeno kako bi ona bila što zanimljivija. Kako će igrica izgledati i kada će se ona smatrati gotovom ovisi o mašti programera i njegovoj viziji.

LITERATURA

[1]https://www.youtube.com/playlist?list=PLlrATfBNZ98eOOCk2fOFg7Qg5yoQfFAdf

[2]https://www.youtube.com/playlist?list=PL_QPQmz5C6WUF-pOQDsbsKbaBZqXj4qSq

[3]https://www.youtube.com/playlist?list=PLWms45O3n--6TvZmtFHaCWRZwEqnz2MHa

[4]https://docs.oracle.com/javase/8/docs/api/java/awt/Graphics.html

[5]https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/Game%20Programming%20Algorithms%20and%20Techniques_%20A%20Platform-Agnostic%20Approach%20%5BMadhav%202013-12-29%5D.pdf

SAŽETAK

Cilj završnog rada je bio napraviti 2D igricu u Javi bez korištenja vanjskih biblioteka za izradu igara. Cilj igrice je da igrač preživi što dulje dok ga napadaju neprijatelji dok on ima mogućnost bježanja ili uništavanja neprijatelja. Igrica je temeljena na pločicama veličine 16*16 piksela koji su temeljni gradivni element. Mapa/svijet po kojoj se kreću igrač i neprijatelj je veća od prozora igrice stoga je bilo potrebno implementirati pokret kamere. Na mapi se nalaze i zidovi koji ne omogućuju prolazak neprijateljima i igračima stoga je bilo neophodno detektirati kada dolazi do kolizije, ali također kolizija se koristi i između igrača i neprijatelja. Igrač može napadati neprijatelji te je to također jedan od slučajeva kada je korištena detekcija kolizije. Napravljen je sustav menija za kretanje između stanja igrice kao što su završena igra ili pauza. Za kraj je implementiran A* algoritam koji omogućuje da neprijatelj prati kretanje igrača po mapi .

Ključne riječi: 2D, igrica, petlja igrice, Java, pločice

ABSTRACT

Title: 2D game in Java

The goal of this final work was to make a 2D game in Java from scratch without using external libraries or game engines for game development. Theme of the game is about the player trying to survive as long as possible while being attacked by the enemies. Player can survive by running away and also attacking and destroying enemies. Game is based on tiles 16*16 pixels size which are fundamental building elements of this project. Map/world on which characters are placed is bigger than the window of the game so it was necessary to implement camera movement. Map contains walls as well which are solid and do not allow characters to go through to it. Collision detection was used to solve this problem as well for some other cases like player and enemy collision or enemy attack hit detection. Game has a menu system as well to move between different game states like game over state or pause. At the end, A* an algorithm for path searching was implemented so the enemy can follow player around the map.

Key words: 2D, game, game loop, Java, tiles