Izrada multiplatformske igre pomoću mobilnog razvojnog okvira Flutter

Goran, Alković

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Izrada multiplatformske igre pomoću mobilnog razvojnog okvira Flutter

DIPLOMSKI RAD

Varaždin, kolovoz 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Goran Alković

JMBAG: 0016115937

Studij: Baze podataka i baze znanja

Izrada multiplatformske igre pomoću mobilnog razvojnog okvira Flutter

DIPLOMSKI RAD

Mentor:

Dr. sc. Mladen Konecki

Varaždin, kolovoz 2020.

Sadržaj

Goran Alković

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Izrada igre (koja se sastoji od tri manje igre) za više platformi (Android, iOS, web, Windows, macOS, Linux) u multiplatformskom razvojnom okviru *Flutter*. Opis platforme i usputnih dodataka i softvera korištenog za izradu. Opis tehnika, pojmova i softvera za izradu razina prve igre. Detaljan pregled i opis koraka implementacije svake od igara, uz prikaz razloga odabira određenih uzoraka i implementacija popraćen bogatim vizualnim prikazima. Ukratko su opisani bitni elementi razvojnog okvira *Flutter* korišteni za izradu dijela sučelja. Svaki dio opisa implementacije popraćen je i programskim kôdom koji je objašnjen dio po dio. Na kraju se rezimiraju iskustva rada s *Flutter*om i izrade projekta te savjeti potencijalnim budućim korisnicima *Flutter*a.

Ključne riječi: Flutter, igra, računalna igra, programiranje, višeplatformski

1. Uvod

Izrađivati igre nikad nije bilo lako. Na samim počecima to je zahtijevalo rad s jako niskim programskim jezicima i nekad jednostavno čuda za implementirati neke funkcionalnosti na tada ograničenim platformama. Igre poput *Dooma* u svoje su vrijeme napravile revoluciju.

Zbog brzog rasta i napretka tehnologije računala su ubrzo došla u kuće, što je dodatno proširilo tržište. Uvijek popularne konzole dobile su protivnika. Igranje igara na osobnim računalima je brzo postalo, a i još uvijek ostalo iznimno popularno i većini primaran način igranja igara. Tome je pridonijela relativno laka dostupnost hardvera, otvorenost platforme i velik broj igara na izbor.

Naravno, igranje na konzolama i osobnim računalima imalo je i nedostataka. Jedan od njih je fiksna lokacija uređaja. Naravno, u teoriji nije problem odspojiti, prenijeti i ponovno spojiti računalo, ali ipak je to jako spor i nepraktičan proces. Prijenosna računanal tadašnjeg vremena nisu bila dovoljno jaka za igranje *ozbiljnih* igara, pa većina ih nije uzimala u obzir za igranje. Pojavilo se nekoliko prijenosnih konzola, poput PSP-a (*PlayStation portable*) i Nintendo DS-a, ali nikada nisu uspjele doseći popularnost „običnih“ konzola, kamoli osobnih računala.

Razvoj pametnih telefona početkom 21. stoljeća pokrenuo je, tada za njih nepojmljivu, revoluciju. Izlaskom prvog iPhonea 2007. tržište mobilnih uređaja se potpuno promijenilo. Napretkom hardvera i rastom tržišta počele su se razvijati igre za mobilne uređaje. Ubrzo su došle igre poput *Flappy Bird*a koje su prodrmale svijet mobilnog igranja.

Broj platformi na kojima ljudi mogu igrati igre ima svoje prednosti i nedostatke. Prednost je da se zabava može naći na bilo kojem od uređaja, ali problem dolazi kada se razmatra izbacivanje jedne igre za više platformi. Svaka od ovih platformi ima svoj skup pravila, API-ja (eng. *Application Programming Interface*, aplikacijsko-programsko sučelje) i tehnologija koje omogućuju razvoj aplikacija, tj. igara. Sve te tehnologije su dovoljno drukčije da nekad tvrtke odluče ili ne izbaciti verziju za drugu platformu ili to traje jako dugo. Uzmimo za primjer *Grand Theft Auto V* – izrada verzije za osobna računala trajala je (barem) dvije godine – tj. PC verzija je izašla pune dvije godine nakon verzije za konzole.

S vremenom su nastali i alati koji su omogućili i olakšali razvoj igara za više platformi, poput *Unity*ja i *Unreal Engine*a. To su sada iznimno popularni i moćni alati, što može uplašiti manje iskusne programere i manje timove.

Uz razvoj samih igara postojao je i problem razvoja aplikacija za mobilne platforme općenito. Razvojni okviri poput *Xamarin*a pokušali su riješiti problem, međutim nikada nisu

stekli popularnost. *React Native* je uspio dostići popularnost, ali nije baš najoptimalniji alat za izradu performantnih aplikacija, što isključuje sve osim najjednostavnijih igara, jer koristi *međusustav* i sloj nad *nativnim* (sloj koji koristi *službene* tehnologije platforme za koju se razvija), što dodatno usporava izvršavanje.

Google je svojim (relativno) novim razvojnim okvirom Flutter započeo novu revoluciju razvoja višeplatformskih aplikacija, a time i dao mogućnost za novi razvojni okvir kojim se mogu izrađivati igre za više platformi.

Osim mogućnosti razvoja na više platformi Flutter sam odabrao zbog mogućnosti brzog i brzo-iterativnog razvoja. Promjene napravljene u kodu mogu se iznimno brzo vidjeti u samoj aplikaciji, što znatno ubrzava razvoj. Činjenicom da je razvijen od strane Googlea, da već duže vrijeme platforma mlađa od dvije godine generira poslove na tržištu i nevjerojatno brzim rastom zajednice i proširenja koja je napravila zajednica, htio sam istražiti kakve su mogućnosti za izradu igara u Flutteru.

2. Metode i tehnike rada

Rad je napravljen na način da se prvo krenulo s iterativnom izradom MVP-a (eng. *minimum viable product*, prve funkcionalne verzije) igre, zatim je počeo proces poliranja korisničkog sučelja i izrade dodatnih razina (eng. *level*), grafike i sličnih materijala, uz paralelno pisanje dokumentacije.

Od alata korišteni su *Microsoft Visual Studio Code* za pisanje programskog kôda igre, *Adobe Xd* za izradu dijela vizualnih elemenata te *Tiled* za izrađivanje razina za igru.

Razvojni okvir u kojem je rađena igra je Flutter (za više informacija pogledati sljedeće poglavlje), zajedno s nekoliko biblioteka. Korišteni programski jezik je *Dart*, također pisan od strane Googlea.

Svi vizualni elementi su autorsko djelo, osim ako je navedeno drukčije.

3. Korištene tehnologije i alati

3.1. Flutter

Flutter je Googleov razvojni okvir za izradu nativno-kompajliranih aplikacije za mobilne uređaje, stolna računala (eng. *desktop*) i web. [1]

Slika 1: Logo *Fluttera* [2]

Počeo je kao mini-projekt internog tima u Googleu, s ciljem da omoguće lak razvoj performantnih aplikacija za Android uređaje. Tadašnji nativni Android razvojni okvir u kombinaciji s XML-om i Javom se pokazao kao nespretan i spor za razvoj.

Flutter se oslanja na četiri glavna elementa:

- Platforma i programski jezik Dart
- Sloj niske razine (eng. *engine*) koji implementira iscrtavanje na zaslon i komunikaciju s API-jima platforme (npr. rad s datotekama, rad s mrežom, …)
- Dart bibiloteke za komunikaciju sa slojem niske razine - biblioteka *Foundation*
- Ugrađeni *widgeti* koji daju osnovne gradivne blokove sučelja

Iako je u pozadini Fluttera *Skia* podsustav za iscratavanje grafike koji bi u teoriji mogao iscrtavati 3D grafiku, Flutter je prvenstveno namijenjen izradi 2D aplikacija. Stoga, igre koje sam izradio u sklopu ovog rada su dvodimenzionalne. Naravno, ako postoji potreba elementi u sučelju mogu se rotirati u 3D prostoru korištenjem matričnih transformacija, međutim to nije najbolje rješenje za izradu igara.

Verzija 1.0 Fluttera izašla je u prosincu 2018. godine. Od tada platforma i njoj pripadajuća zajednica iznimno brzo rastu. Već od početka Google je koristio Flutter za izradu aplikacija popud AdWords aplikacije i kompletnog korisčnog sučelja Google Home Hub-a. S vremenom su se tom popisu počeli pridruživati i veliki klijenti poput BMW-a, Tencenta, eBay-a, Alibaba grupacije itd. [3]

Na početku je cilj bio podržati samo mobilne platforme, no već početkom 2019. razvojni tim počeo je istraživati podršku za web. Uskoro je počeo i razvoj podrške za *desktop* platformu, tj. stolna računala. Trenutno Flutter poptuno podržava Android i iOS, web podrška je trenutno u beta fazi razvoja, ali funkcionira jako dobro. Podrška za stolna računala je relativno nova, stoga nije preporučena za izradu produkcijskih aplikacija. Od operacijskih sustava za stolna računala podržani su Windows, macOS i Linux. Igru sam uspješno pokreno na Androidu, Windowsu i u web verziji.

3.2. Dart

Dart je programski jezik optimiziran za razvoj brzih klijentskih aplikacija na svim platformama. [4] To je objektno-orijentirani programski jezik sa sintaksom sličnom mnogim modernim jezicima poput JavaScripta, TypeScripta, Kotlina i Swifta.

Slika 2: Logo programskog jezika Dart [5]

Jedan od razloga zašto je Google odabrao Dart za Flutterov programski jezik je to što je iznimno fleksibilan s kompajliranjem. Može se prevesti ili u nativni kôd ili u JavaScript. Osim toga, postoje dva kompajlera: JIT (eng. *Just in Time*) i AOT (eng. Ahead of Time). JIT kompajler je prikladniji pri razvoju programa, gdje se promjene mogu vrlo brzo rekompajlirati i prikazati korisniku. AOT kompajler je prikladniji za produkcijske verzije aplikacija (verzije koje idu krajnjim korisnicima) jer su izvršne datoteke manje i izvršavanje je brže jer je sve već kompajlirano i *u hodu* ne treba ništa novo kompajlirati. Za web platforme postoje slične verzije kompajlera (Dart u JavaScript) za razvojnu i produkcijsku namjenu. [6]

3.3. Flame

Flame je modularni *game engine* (jezgra igre) napravljen da radi s Flutterom. Olakšava izradu igara implementacijom petlje igre (eng. *game loop*), uz pomoćne funkcionalnosti koje

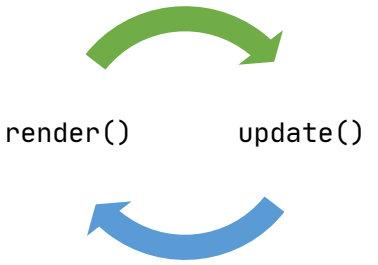
mogu dobro doći pri izradi igre, poput učitavanja slika, *spritesheet*ova i *sprite*ova iz njih, animacija, reprodukcije zvuka itd. [7]

Slika 3: Logo *Flame*a [7]

Iako je „tek“ na verziji 0.25, Flame je jako brzo postao popularna biblioteka za Flutter, što pokazuje kontinuirani napredak od izdavanja prve verzije.

3.3.1. Petlja igre i komponente

Petlja igre (eng. *game loop*) je jedan od glavnih *Flame* modula. Nudi osnovnu funkcionalnost na kojoj se temelje gotovo sve igre: ***render-update* ciklus**.



Slika 4: *Render-update* ciklus (vlastita izrada)

Te metode se ponavljaju beskonačno i tako omogućavaju iscrtavanje elemenata igre na zaslon (metodom `render`) i same funkcionalnosti igre uz proces modificranja internog stanja igre (metoda `update`).

Za korištenje ovih elemenata dovoljno je proširiti implementaciju klase klasom `Game`. Osnovna implementacija prikazana je ispod:

```
class MojaIgra extends Game {
	void render(Canvas canvas) {
		...
	}

	void update(double t) {
		...
	}
}
```

Uz „manualnu“ implementaciju, *Flame* nudi i klasu `BaseGame` koja omogućava izradu igre radom s komponentama, gdje svaka komponenta ima svoje `render` i `update` metode koje se mogu proširiti ponašanjima koja želimo.

Postoji više vrsta komponenti, a osnovne su: [8]

- `PositionComponent`
  komponenta koja omogućava upravljanje x/y koordinatama, visinom, širinom i kutom nagiba komponente
- `SpriteComponent`
  komponenta koja omogućava iscrtavanje objekta ili lika iz slike
- `AnimationComponent`
  komponenta koja omogućava izradu animacija od kolekcije dvije ili više slike koje se prikazuju slijedno

Uz osnovne komponente, postoje i naprednije komponente od kojih vrijedi istaknuti `TiledComponent`, koji omogućava rad s *Tiled* razinama.

3.3.2. Interakcija s korisnikom

Flame je iznimno fleksibilan u vidu interakcije s korisnikom. U paket `gestures` ugrađena su proširenja za Flutterove detektora gesti koje omogućavaju detektiranje dodira, vučenja prsta po zaslonu i sl. Na desktop platformi ove geste mogu se izvršavati mišem.

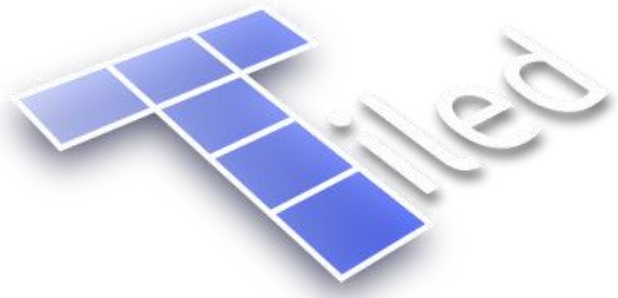
Uz dodir i miš podržani su tipkovnica i igrače palice (eng. *joystick*). Podrška za tipkovnice je vrlo fleksibilna i omogućava vrlo lako ubacivanje tipkovničke kontrole u igru. Podrška za igrače palice u Flame je ugrađena u vidu virtualnih *joysticka*, koji se mogu iscrtati na zaslon. Korištenje fizičkih *joysticka* i *gamepada* moguće je preko vanjske biblioteke, ali je trenutno podržano samo na Androidu.

3.3.3. Čestice

Čestice (eng. *particles*) mogu znatno vizualno obogatiti igru, a u *Flame*u ih je lako implementirati. Postoje implementacije sedam tipova čestica pomoću sustava komponenti, a moguće je i napraviti prilagođene čestice (`ComputedParticle`), što je korišteno u ovom projektu.

3.4. Tiled

Tiled je besplatni 2D uređivač razina igre (eng. *level editor*). [9] Služi za brzu i laku kreaciju razina za igre na način da nudi standardizirani format zapisa razine i pripadajućih objekata. Podržava klasične dvodimenzionalne tipove razina – pogled odozgo (eng. *top-down*, kao npr. u igri *Grand Theft Auto*) i pogled sa strane (eng. *side-scrolling*), kao i izometrijski pogled kojim se mogu simulirati trodimenzionalni prostori.



Slika 5: Logo aplikacije Tiled [10]

Svaka razina može se sastojati od više slojeva. Unutar razine može se dodati više vrsta pločica, svaki tip u svom sloju, uz mogućnost dodavanja više slojeva. Glavne su vrste slojeva **sloj pločica** i **sloj objekata**. Uz to, postoje i slikovni slojevi (eng. *image layer*), slojevi potamnjivanja (eng. *tinting layer*) i grupni slojevi (eng. *group layer*). [11]

3.4.1. Pločice

Pločice (eng. *tiles*) su osnovni gradivni element razine. Od glavnih svojstava imaju svoju poziciju u prostoru (X-Y koordinate) i teksturu, odnosno vrstu pločice. Dodatno se mogu se spremati i podaci o rotaciji pločice.

Postoji više načina dodavanja pločica u razinu. Najjednostavniji je crtanje pločice jedna po jedna. *Tiled* nudi alate koji olakšavaju izradu većih razina poput načina pravokutnog dodavanja, dodavanje pomoću četke (eng. *Stamp Brush tool*), alata za crtanje terena (eng. *Terrain Brush tool*), koji olakšava rad s elementima gdje postoje prijelazi između različitih vrsta pločica, poput zgrada, krajolika itd. [12]

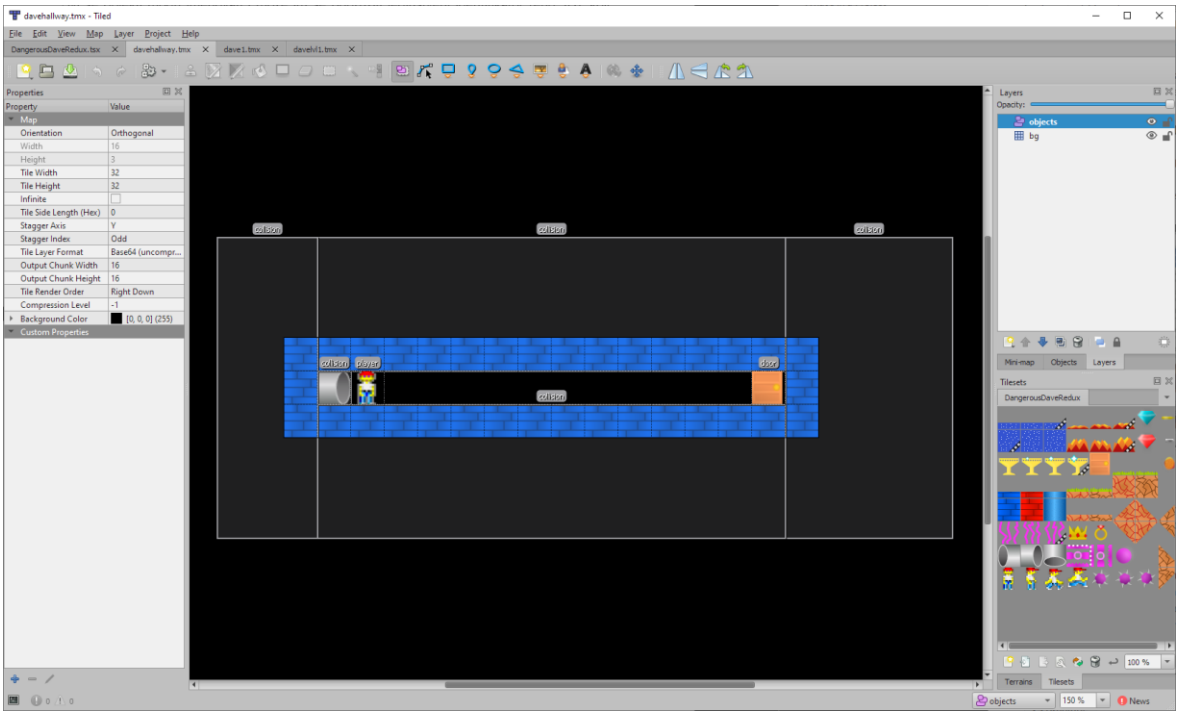
3.4.2. Objekti

Slojevi objekata koriste se za spremanje objekata obogaćenih dodatnim informacijama. Objekti nisu ograničeni mrežom pločica, pa se mogu slobodno pomicati, rotirati i povećavati, odnosno smanjivati. Korisni su kada želimo dodati elemente koji će u igri biti interaktivni, poput igrača, protivnika, elemenata koje skupljamo itd. [11]

Osnovni objekti su geomtrijski oblici:

- Pravokutnik
- Elipsa
- Točka
- Poligon
- Polilinija
- Pločica
- Tekst

Svi se objekti mogu imenovati i može im se pridružiti jedinstveni identifikator (eng. *ID*), koji poslije možemo čitati programski da bi dodali funkcionalnost i interaktivnost objektima.



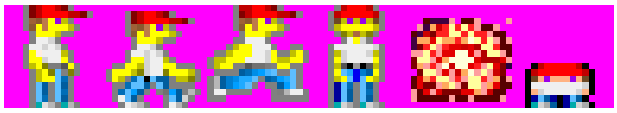
Slika 6: Sučelje izrade razine (vlastita izrada)

3.4.3. Setovi pločica

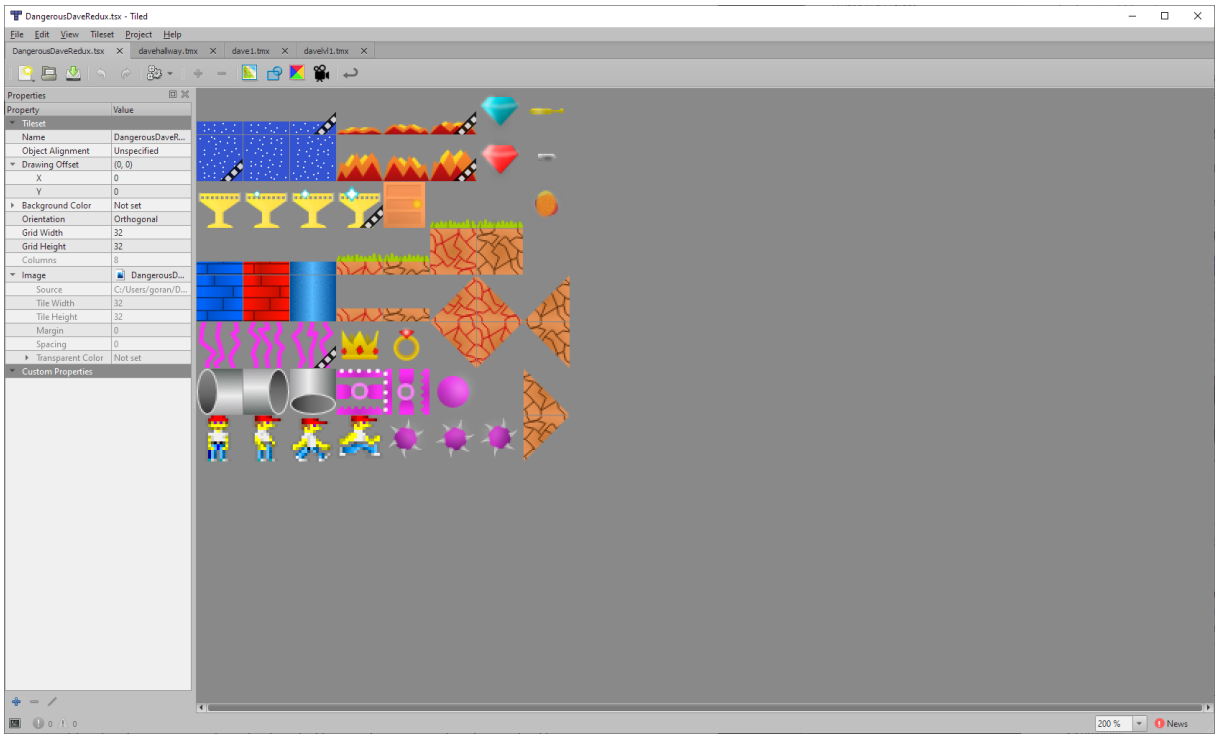
Teksture, odnosno grafika za pločice spremaju se u setove pločica (eng., u daljnjem tekstu *tileset*). Set pločica može biti baziran na jednoj slici ili na kolekciji više sličica. [13]

*Tileset*ovi bazirani na jednoj slici koriste fiksnu veličinu za svaku pločicu. Iz slike se izrezuju manje slike koje se koriste za prikaz pločice. Jedna takva slike zove se *spritesheet*.

Tilesetovi bazirani na više slika pogodni su kada želimo imati slike različitih dimenzija ili želimo poslije napraviti optimizirani *spritesheet*. Primjer jednog *spritesheet*a vidljiv je na slici ispod:



Slika 7: Primjer *spritesheet*a [14]



Slika 8: Sučelje za uređivanje *tileseta* (vlastita izrada)

3.4.4. Izvoz razine

Tiled omogućava izvoz (eng. *export*) u više formata zapisa razine, kao i mogućnosti izvoza razine u sliku.

Glavni formati izvoza su JSON (eng. *JavaScript Object Notation*), Lua, CSV (eng. *Comma-Seperated Values*), GameMaker XML i *.tmx* (*Tiled*ov format datoteka).

Za potrebe projekta korišten je *.tmx* format, koji zapisuje u XML format, koji se poslije lagano može čitati.

3.4.5. Tiled – Flame interoperabilnost

Flame nudi paket koji omogućava učitavanje *Tiled* razina u *.tmx* formatu. Ugrađeno je iscrtavanje statičnih pločica, odnosno sloja pločica, a za objekte postoji samo *parser* (funkcija koja čita i oblikuje sirove podatke iz *.tmx* datoteka), a sve ostalo se ručno implementira.

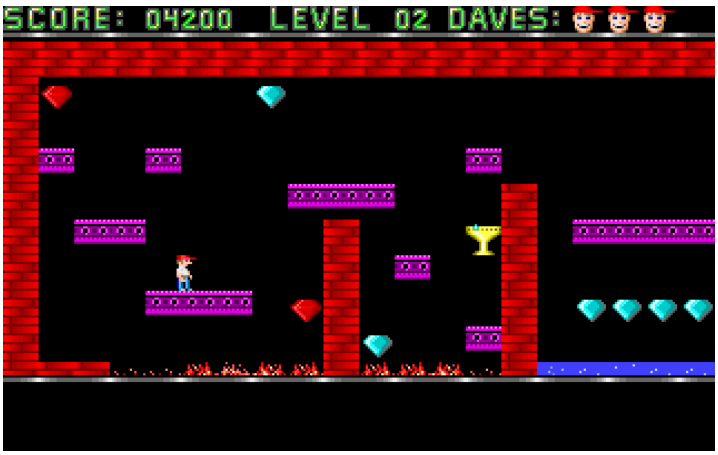
4. O igri

U ovom poglavlju ukratko će biti opisana ideja igre, kako se došlo do te ideje i pripadajući žanrovi.

4.1. Inspiracija

Kako sam igre počeo igrati u jako mladoj dobi, u sjećanje su mi se trajno zapisale igre poput *Doom*a, *Doom* a 2, *Tetris*a, *Dangerous Dave*a, *Supaplex*a, *Prehistorik*a i sličnih DOS igara. Zbog toga sam odlučio da će moj projekt sadržavati barem jednu retro igru. Uz to, većina *retro* igara nisu po funkcionalnosti previše kompleksne, pa su savršen kandidat za isprobavanje nove platforme.

Slika 9: Doom (1994) [15]



Slika 10: Dangerous Dave (1990) [16]

Slika 11: Tetris (1987) [17]

4.2. Ideja

Cilj ovog rada bio je istraživanje Fluttera kao razvojnog okvira za izradu igara. Da bi kroz praktični primjer što bolje istražio Flutter, odlučeno je da treba implementirati pet mehanika igre. Iako se to može napraviti kroz jednu igru, odlučio sam napraviti aplikaciju koja sadrži više manjih igara u sebi, u nekom vidu „virtualna kutija s igračkama“, pa sam zbog toga odlučio aplikaciju nazvati *GameBox*.

Odlučio sam rekreirati tri igre: *Dangerous Dave*, *Tetris* i *Minesweeper*.

Uz korištenje različitih mehanike igre, kroz ove tri igre odlučio sam pokazati da se za izradu igara ne mora nužno koristiti biblioteka poput Flamea, da se mogu koristiti i komponente Fluttera. Više u poglavlju *Implementacija*.

4.3. Žanrovi

Odabrane tri igre pokrivaju nekoliko žanrova:

- *Dangerous Dave*
    - Akcijska igra
        - Platformer
- *Tetris*
    - Slagalica

    - *Padajući blokovi*
- *Minesweeper*
    - Matematička slagalica

U nastavku će biti opisani pojedini žanrovi i istaknuti specifični podžanrovi kojima igre pripadaju, dok će ostali podžanrovi biti ukratko opisani.

4.3.1. Akcijska igra

Akcijska igra je žanr kod kojeg igrač upravlja virtualnim likom, najčešće uz prateće izazove koji zahtijevaju brzinu i dobre reflekse kod igrača. Ovo je najpopularniji žanr igara jer je vrlo lako početi igrati i potrošiti puno vremena. [18]

Uz brze poteze i reflekse u ovakvim se igrama često pojavljuju prepreke i mehanike borbe. Žanr *akcijska igra* sadrži mnoštvo podžanrova, a neki od njih su:

- Platformer
- Pucačina / u prvom licu (eng. *Shooter / First-person shooter, FPS*)
- Borbena igra (eng. *Fighting game*)
- Ritmične igre

i mnogi drugi.

4.3.1.1. Platformer

Platformeri su podžanr akcijskih igara koji karakterizira igra kod koje je dominantno skakanje između *plaftormi* različitih veličina. U pravilu lik kojim se igra može skakati relativno visoko i mijenjati svoju putanju u zraku.

Kretnje se najčešće izvršavaju slijeva nadesno, međutim postoji dosta igara koje omogućavaju i pomicanje ulijevo. Zamke i prepreke su tu da stoje igraču na putu do pobjede. [19]

Neki od prvih platformera su *Donkey Kong* i *Super Mario Bros*. *Super Mario Bros* je popularizirao način ubijanja neprijatelja skakanjem na njih. Većina drugih igara igraču daje određeno oružje kojim mogu ubijati neprijatelje.

Iako su s vremenom došli i 3D platformeri, od starijih poput *Crash Bandicoot*a, do novijih poput *Mirror's Edge*a, platformeri više nisu ni blizu popularni koliko su bili. Naravno, kako se tržište konstantno mijenja, nikad se ne zna što budućnost donosi.

4.3.1.2. Pucačina / u prvom licu

Pucačine su žanr koji se bazira na korištenju oružja bliskog ili dalekog dometa. Najčešća inačica su pucačine u prvom licu (eng. *First-person shooter, FPS*), gdje je perspektiva *iz očiju igrača*. Popularni primjeri su *Doom, Call of Duty* serijal, *Halo* serijal itd. [20]

4.3.1.3. Borbene igre

Borbene igre (eng. *Fighting game*) su igre gdje se igrač bori s neprijateljem u stilu *jedan-na-jedan*. Žanr su popularizirale igre *Mortal Kombat* i *Street Fighter*, a serijal *Tekken* je još uvijek jedan od najpopularnijih predstavnika ovog podžanra. [21]

4.3.1.4. Ritmičke igre

Ritmičke igre (eng. *Rhythm games*) traže od korisnika da prati određeni niz pokreta ili određene ritmove uz pratnju glazbe, svjetala ili sl. Ponekad su izvedene kao dio uređaja koji olakšavaju praćenje kretnje igrača. [21] Naslovi poput *Guitar Hero*a i *Just Dance*a su poznati primjeri ovih igara.

4.3.2. Slagalice

Slagalice (eng. *Puzzle*) su žanr koji primarno zahtjeva mentalne sposobnosti igrača, a malo ili uopće ne traži brzinu i dobre reflekse. [22]

Ovaj žanr je vrlo širok pa obuhvaća i igre koje su primarno slagalice, igre drugog žanra koje imaju elemente slagalice i igre koje po strogoj definicije nisu *fizičke slagalice*, već zahtijevaju razmišljanje kako određene elemente uklopiti da bi se došlo do rješenja.

Neki od glavnih podžanrova slagalica su:

- *Padajući blokovi* (eng. *Falling Blocks*)
- Akcije-slagalice
- *Čiste* slagalice
- Slagalice-platformeri
- Matematičke/logičke slagalice

i mnogi drugi.

4.3.2.1. *Padajući blokovi*

Padajući blokovi je podžanr slagalica koji podrazumijeva seriju blokova koji se pojavljuju na vrhu područja igre i padaju prema dolje. Dok padaju igrač ih može pomaknuti ili

rotirati s ciljem da ih posloži. Kada se posloži jedan cijeli red on nestane i svi ostali blokovi se pomaknu za jedan dolje. U nekim verzijama brzina padanja blokova ubrzava s vremenom. [20]

Tetris je započeo ovaj iznimno popularan žanr koji je i danas popularan, s relativno puno klonova *Tetrisa* i njegovih varijacija.

4.3.2.2. Matematičke/logičke slagalice

Ovaj žanr traži od igrača brzinsko rješavanje jednostavnih matematičkih problema i razradu strategije. [18]

Iako se ovaj žanr može povezati i sa žanrom *padajući blokovi*, ovaj je žanr malo apstraktniji po specifičnim detaljima izvedbe igre.

Jedan od popularnih predstavnika ovog žanra je *Minesweeper*, koji traži od igrača konstantno kalkuliranje situacije u igri jer jedan pogrešan proračun dovodi do gubitka igre.

5. Mehanike

U ovom poglavlju će biti navedene i opisane mehanike prisutne u svakoj od tri igre.

5.1. *Dangrous Dave*

Dangerous Dave je klasični platformer s akcijskim elementima. Igrač mora prolaziti kroz različite razine izbjegavajući prepreke (vatru, kiselinu i sl.) i ubijajući neprijatelje. Cilj svake razine je skupiti trofej/pehar i doći do vrata. Igrač je opremljen pištoljem koji mu pomaže pri savladavanju razina.

U ovoj igri prisutne su sljedeće mehanike:

- Skakanje + gravitacija
- Pucnjava od strane protivnika
- Objekti u svijetu koji uzrokuju trenutnu smrt

5.2. *Tetris*

Tetris je igra slaganja blokova. Iako šareni blokovi izgledaju jednostavno, situacija se brzo zakomplicira. Iako striktnog cilja nema, možemo reći da je cilj popuniti red blokovima, nakon čega on nestaje i blokovi iznad padaju. Vrh ploče za igru je naš neprijatelj, ako ga dosegnemo, tu je igri kraj!

U ovoj igri prisutne su sljedeće mehanike:

- Blok-slagalice
- *Igra se ponavlja beskonačno do smrti igrača*

5.3. *Minesweeper*

Minesweeper je igra za koju svi znaju. Sve što treba napraviti je zastavicom označiti polja s minama. Ali treba paziti – jedan pogrešan klik i igra je gotova! Ploča za igru je tu da nam pomogne. Ako u krugu polja koje smo otvorili ima mina, na tom polju će se pojaviti broj koji nam govori koliko ih ima. Još bolje, ako na tom polju nema mina, lančano se traže i otvaraju polja u okolici koja nemaju mina! Često par klikova zna otvoriti većinu ploče, ali onda težak posao počinje.

U ovoj igri prisutne su mehanike:

- *Potreba za konstantnim razmišljanjem i pamćenjem*
- *Igra se ponavlja beskonačno do smrti igrača*

5.4. Kratki opisi mehanika

5.4.1. Skakanje + gravitacija

Na igrača djeluje konstantna sila u jednom smjeru (najčešće prema dolje). Moguće verzije su i nedostatak gravitacije ili objekti koji privlače stvari prema sebi. Mehanika skakanja, kombinirana uz mehaniku gravitacije, omogućava igraču da skače između objekata ili platformi da bi nastavio prolaziti kroz igru. [23]

5.4.2. Pucnjava od strane protivnika

Ova mehanika podrazumijeva čestu ili konstantnu pucnjavu od strane jednog ili više protivnika, što ubrzava tijek igre i zahtijeva brže reakcije igrača. Može biti implementirana kao jedan neprijatelj koji vrlo brzo puca ili kao više neprijatelja. [23]

5.4.3. Objekti u svijetu koji uzrokuju trenutnu smrt

Da bi ova mehanika bila implementirana u svijetu moraju postajati objekti ili zamke koje na dodir igrača uzrokuju trenutnu smrt, odnosno gubitak života ili kraj igre. [23]

5.4.4. Blok-slagalice

Ovakva igra uključuje niz objekata slične veličine koji se moraju pomaknuti i složiti na isključivo jedan način. [23]

5.4.5. *Igra se ponavlja beskonačno do smrti igrača*

Ovo je relativno jednostavna mehanika, igra nema posebnog cilja, već se ponavlja dok igrač ne izgubi život (ili sve živote). [23]

5.4.6. Potreba za konstantnim razmišljanjem i pamćenjem

Igrač mora stalno pratiti tijek igre i imati na umu svoj sljedeći potez.

6. Implementacija

U ovom poglavlju opisat će se implementacija igara i njihovih mehanika s pripadajućim programskim kodom i vizualima.

6.1. Dangerous Dave

Od tri odabrane igre ova je najkompleksnija za implementaciju.

Opis implementacije ići će kroz ove korake:

- Izrada razina
- *Tileset*
- Izrada probne razine
- Implementacija osnovne igre i učitavanja razine
- Kolizije i sudarači
- Upravljanje igračem
- Sustav adaptivnog učitavanja razine
- Integracija u igru
- Dodavanje funkcionalnosti igri
- Dodatne funkcionalnosti

6.1.1. Izrada razina

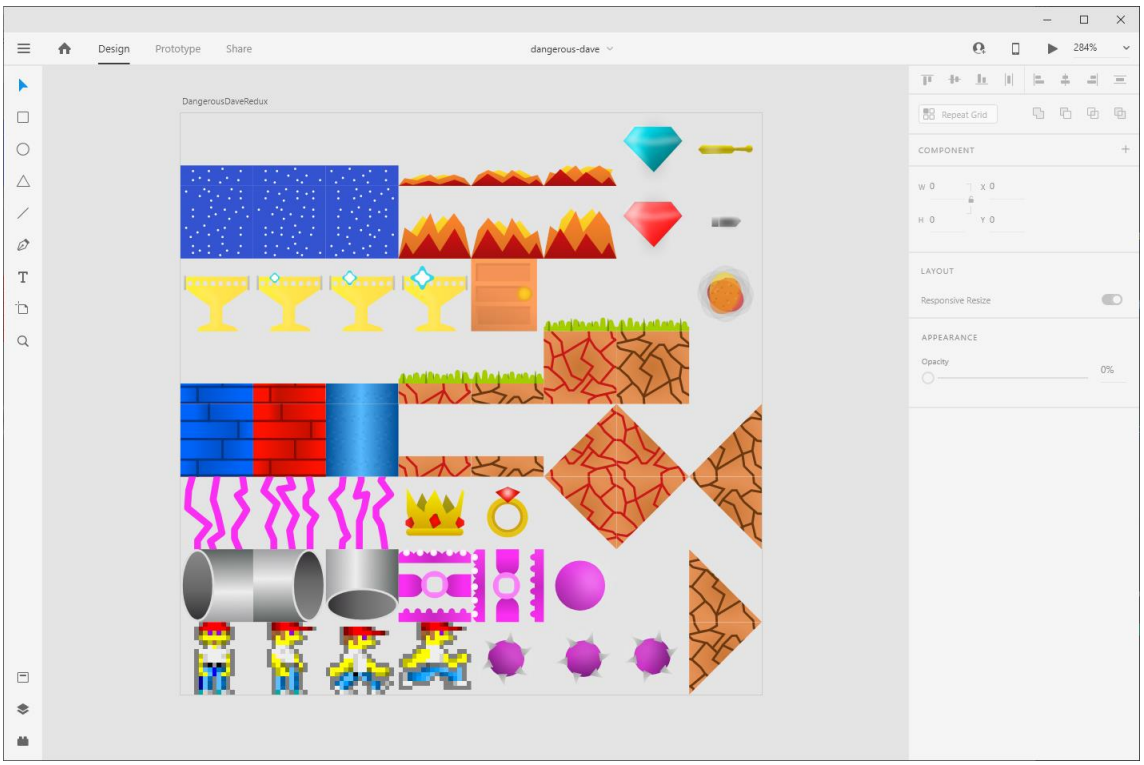
Prije početka izrade igre bilo je potrebno odlučiti na koji način zapisivati i spremati razine koje će se koristiti u igri. Prvotni je plan bio ručno zapisivati podatke u format poput JSON-a, pa onda programski učitavati i ručno *parse*ati (čitati i obrađivati) iste. To bi na kraju radilo, ali bi uzelo jako puno vremena.

Pronalaskom alata *Tiled* plan je promijenjen. Kako *Tiled* daje format zapisa razina i dosta napredan uređivač bio je logičan izbor.

6.1.2. Tileset

Prije izrade razine trebalo je napraviti *tileset*. Planirao sam uzeti neki gotovi *tileset*, ali nisam uspio naći nijedan koji daje istu atmosferu kao i original, pa sam ga odlučio izraditi sam. Vizuali su izrađeni u *Adobe Xd*-u, ukupno je potrošeno oko 2 sata na izradu vizuala. Jedini

vizual koji je preuzet (sa [14]) iz originalne igre je sâm glavni lik jer nisam bio zadovoljan vlastitom verzijom istog.

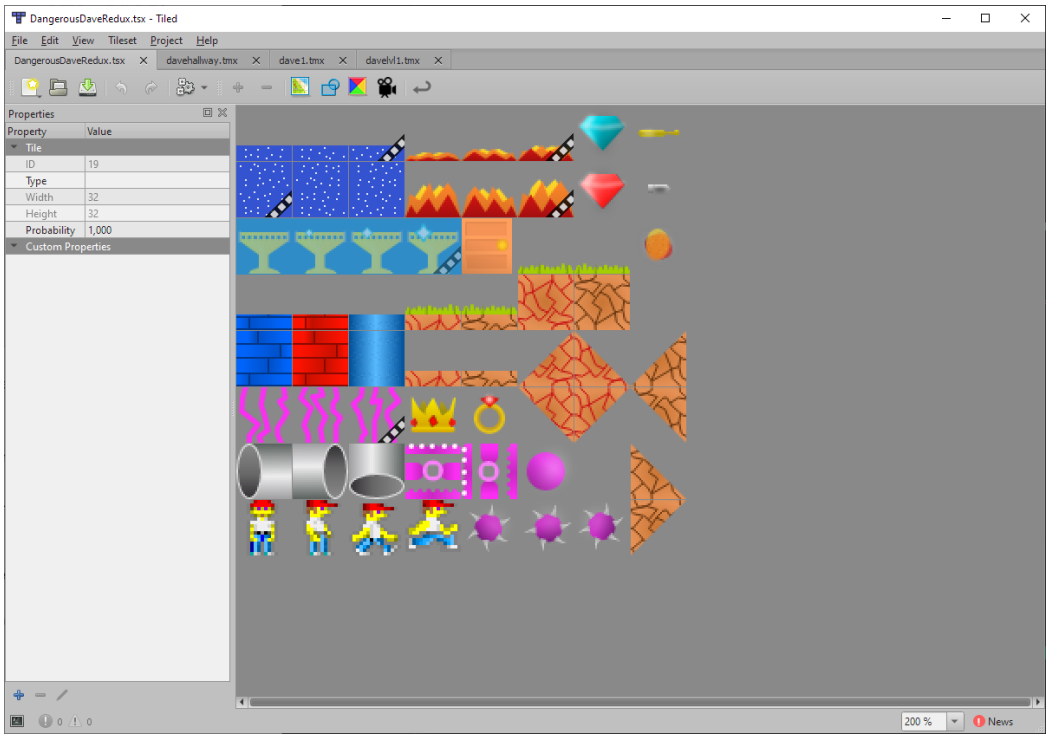


Slika 12: Izrada vizuala u *Adobe Xd*-u (vlastita izrada)

Svi su vizuali izrađeni kao vektori pa bi se ako bude potrebno lako mogli skalirati na veće.

Nakon kreiranja vizuala bilo je potrebno napraviti izvoz (eng. *export*) u neki od slikovnih formata, (odabran je PNG jer podržava prozirnost) da bi se mogao uvesti (eng. *import*) u *Tiled*.

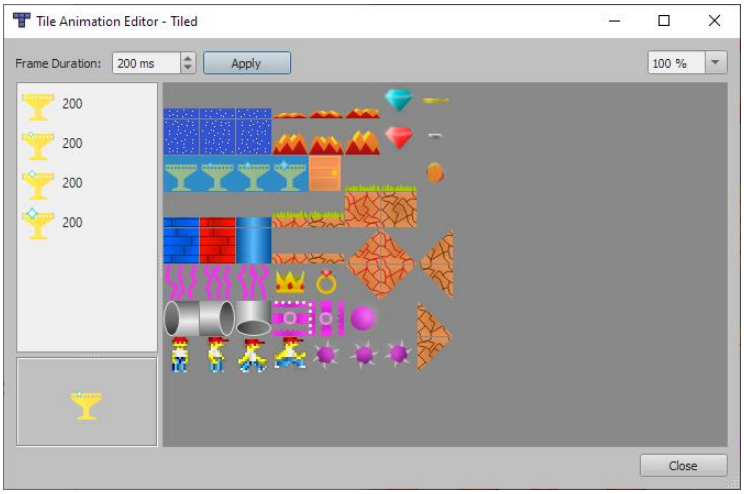
U *Tiled*u je trebalo definirati veličinu svakog polja i *tileset* je bio spreman.



Slika 13: Uvezeni *tileset* u *Tiled*u (vlastita izrada)

Na slici je vidljivo da se neke slike pojavljuju više puta, samo s malim varijacijama. Te slike tako grupirane služe za izradu animacija.

Animacije je lako kreirati. Označimo pločice koje želimo da se animiraju, otvorimo *Tile animation editor* i dodamo slike pločica u željenom redosljedu (vidi sliku ispod). Treba imati na umu da za svaku animaciju posebno treba označiti pločice i stvoriti animaciju i između kreiranja zatvoriti i ponovno otvoriti dijaloški okvir.



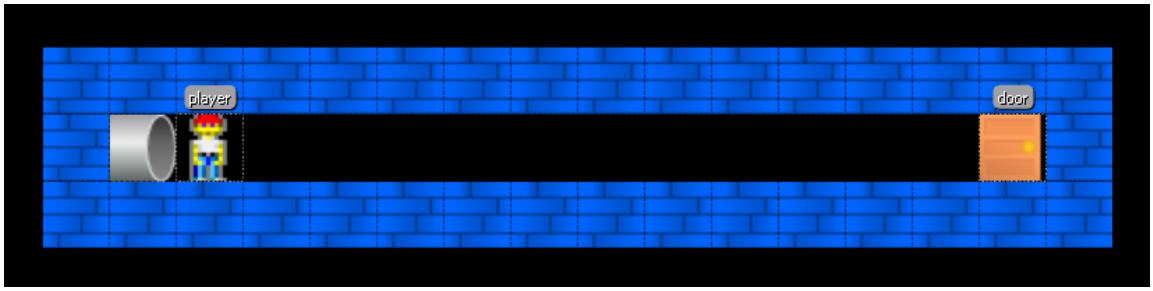
Slika 14: Izrada animacije od više pločica u *Tiled*u (vlastita izrada)

Animirane pločice označene su ovakvom ikonom uz pločicu:

Slika 15: Prikaz animirane pločice u *Tiled*u (vlastita izrada)

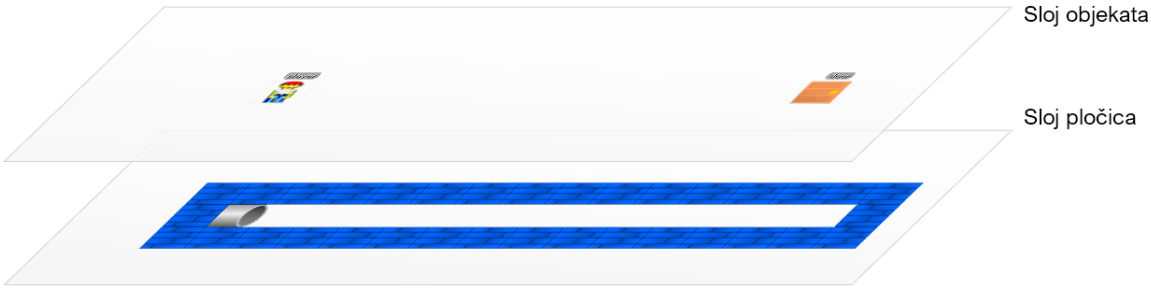
6.1.3. Izrada probne razine

Nakon što je tileset pripremljen, bilo je vrijeme napraviti jednostavnu probnu razinu da vidim kako ju mogu učitati u Flutter igru. Probna razina – *Hodnik* – prikazana je na slici ispod, a predstavlja „međurazinu“ koja je u originalnoj igri služila kao interaktivni zaslon učitavanja, dok su se u pozadini učitavale datoteke sljedeće razine.



Slika 16: *Hodnik* razina (vlastita izrada)

Ova se razina kao i sve druge sastoji od dva sloja: sloj objekata i sloj pločica. Na sloju pločica nalaze se neinteraktivni objekti, odnosno cigle u pozadini, dok su u sloju objekata nalaze igrač (`player`) i vrata (`door`) koja označavaju završetak razine i prijelaz na sljedeću.



Slika 17: 3D prikaz slojeva *Hodnik* razine (vlastita izrada)

Slojevi su direktno iznad drugoga što ne kvari iluziju prostora, ali daje mogućnost implementacije dodatnih funkcionalnosti iz objekta, bez dodatnog kôda koji odvaja statične od dinamičnih dijelova razine.

6.1.4. Implementacija osnovne igre i učitavanje razine

Prije učitavanja razine trebalo je inicijalizirati osnovnu Flame igru da bi mogao testirati sustav učitavanja. To je učinjeno implementiranjem nove klase koja proširuje klasu `BaseGame`.

Nakon implementacije osnove igre bilo je vrijeme za istražiti kako učitati razinu napravljenu i uređivaču *Tiled*. Pronašao sam paket `flame_tiled` od istih autora kao i same biblioteke *Flame*. Paket je lako učitati, kao i svaki drugi Flutter/Dart paket. U projektu postoji datoteka `pubspec.yaml` koja sadrži popis svih paketa koji se koriste u projektu. U datoteku samo dodamo zapis prikazan ispod:

```
flame_tiled: ^0.1.0
```

Nakon toga spremanjem datoteke uređivač automatski pokrene naredbu `flutter pub get` koja čita datoteku specifikacija paketa, preuzima ga s centralnog repozitorija i uključuje u projekt. Sljedeće što treba napraviti je uključiti paket u datoteku koju koristimo dodavanjem ove linije na vrh:

```
import 'package:flame_tiled/flame_tiled.dart';
```

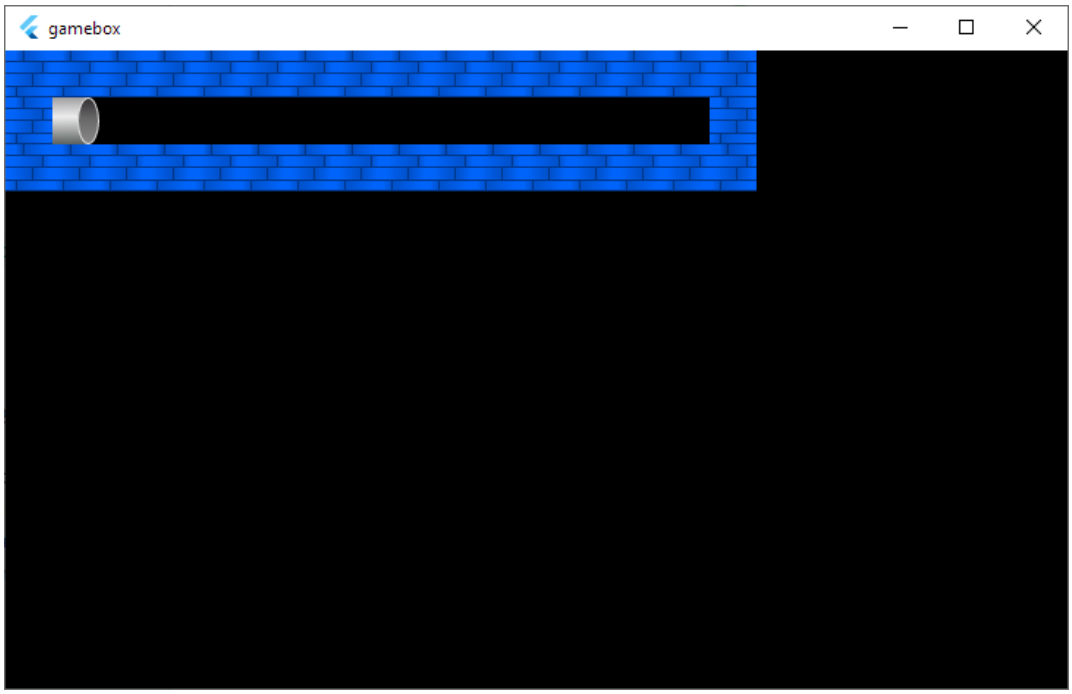
Deklarirat ćemo konstantu TILE_SIZE koje će nam označavati dimenzije pločice (u pikselima).

```
const double TILE_SIZE = 32;
```

Nakon toga možemo instancirati objekt koji će nam biti referenca sustavu za *parsing* (čitanje i obradu) datoteke *Tiled* razine iz paketa `flame_tiled`.

```
TiledComponent _map = TiledComponent("razina.tmx", Size(TILE_SIZE, TILE_SIZE));
```

Ako sada pokrenemo igru možemo vidjeti da nam je sloj pločica učitan i prikazan u gornjem lijevom kutu zaslona.



Slika 18: Djelomično učitana razina (vlastita izrada)

Naravno, ovo je tek dio posla. Sada treba pronaći način kako učitati objekte sa sloja objekata.

Kratkim istraživanjem otkrio sam da postoji metoda `getObjectGroupFromLayer` koja omogućava učitavanje osnovnih metapodataka iz datoteke razine. To je asinkrona metoda, pa je dio koda za učitavanje trebalo prebaciti u odvojenu metodu budući da konstruktor klase ne može biti asinkron. Stoga, za sada naša klasa igre izgleda ovako:

```
class _TempGame extends BaseGame {
  _TempGame() {
    initGame();
  }
  void initGame() async {
    TiledComponent _map =
      TiledComponent("davehallway.tmx", Size(TILE_SIZE, TILE_SIZE));

    add(_map);
    final ObjectGroup objGroup = await _map.getObjectGroupFromLayer("objects");
  }
}
```

Također, moramo uključiti novu datoteku, dodavajući na vrh datoteke liniju:

```
import 'package:tiled/tiled.dart';
```

Sada kada imamo metapodatke o objektima moramo ih dodati u zaslon igre. Gledajući implementaciju objekta `ObjectGroup` možemo vidjeti da je on u suštini lista objekata `TmxObject` s nekoliko dodatnih informacija koje nam trenutno nisu ključne, poput prozirnosti sloja, boje sjenčanja, vidljivosti itd.

Nama zanimljivija klasa je `TmxObject`, koja sadrži podatke o svakom objektu na danom sloju. Podaci koji su zapisani su x/y koordinate, širina, visina, rotacija, jedinstvena oznaka (eng. *ID*) i vidljivost objekta.

Budući da su nam sve informacije o objektima koje nam trebaju tu, možemo `for…in` petljom proći kroz popis `TmxObject` objekata iz elementa `objGroup`:

```
for (TmxObject obj in objGroup.tmxObjects) { ... }
```

Sada dolazimo do pitanja *kako razlikovati različite vrste objekata?* Postoje dva puta kojima možemo ići:

1. Staviti sve objekte iste vrste na jedan sloj
2. Jednako imenovati sve objekte iste vrste pa ih kondicionalno na različit način učitavati u kôdu

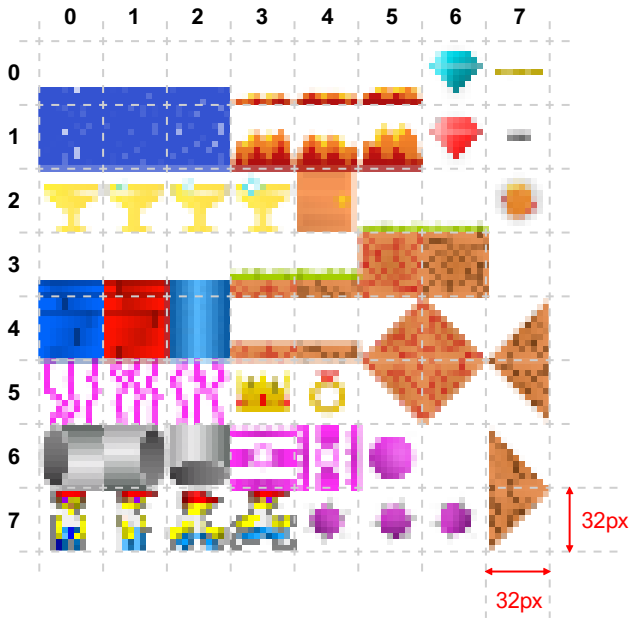
Odlučio sam se za drugi pristup jer ne moram brinuti o količini slojeva i uvoditi dodatne petlje za prolaske kroz sve slojeve, što uvodi dodatnu kompleksnost. Sada možemo pokušati iscrtati objekte sa sloja objekata.

Za početak ćemo iscrtati sve objekte jednom grafikom, pa onda poslije možemo lako dodati mijenjanje grafike ovisno o objektu. Unutar petlje instanciramo objekt tipa *Sprite*, koji služi za spremanje podataka o (statičnoj) grafici, to u kôdu izgleda ovako:

```
Sprite _sprite = Sprite(
        'DangerousDaveRedux.png',
        x: 1 * TILE_SIZE,
        y: 4 * TILE_SIZE,
        width: TILE_SIZE,
        height: TILE_SIZE,
);
```

Prvi i jedini obvezni parametar je ime datoteke *spritesheet*a, u ovom slučaju *DangerousDaveRedux.png*. Tu datoteku moramo u strukturi projekta spremiti u mapu

`assets/images`. Sljedeći parametri su `x` i `y`. Oni označavaju gornji lijevi piksel pločice koju želimo iscrtati, odnosno *izvući* iz *spritesheeta*. Da bi olakšali kalkulacije, uzimamo redni broj retka i stupca i množimo ga s konstantom `TILE_SIZE` (koja je vrijednosti 32).



Slika 19: Prikaz koordinatne mreže *spritesheeta* (vlastita izrada)

U primjeru vidimo da je uzeta koordinata (1, 4) što odgovara bloku kao na slici ispod:

Slika 20: Blok crvene cigle (vlastita izrada)

Sada kada imamo *Sprite*, možemo od njega napraviti komponentu:

```
SpriteComponent sprite =
    SpriteComponent.fromSprite(TILE_SIZE, TILE_SIZE, _sprite);
```

Komponenta nam treba jer u osnovnoj implementaciji igre koristimo `BaseGame`, koji je namijenjen radu s komponentama, što je praktično jer tako lakše možemo u kodu raditi sa svim elementima na zaslonu.

Sada kada imamo komponentu, možemo ju dodati na zaslon, ali prije toga moramo biti sigurni da je ta komponenta učitana da ne bi došlo do iznimke. Budući da smo unutar asinkrone funkcije možemo dodati ovu liniju kôda:

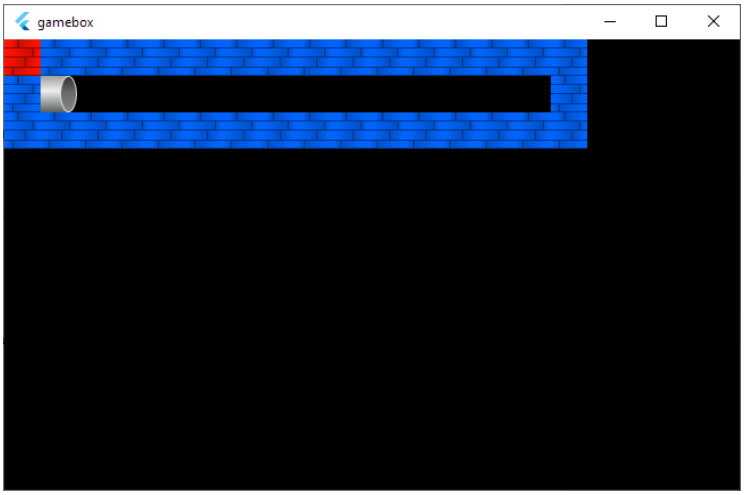
```
while (!sprite.loaded()) await Future.delayed(Duration(milliseconds: 50));
```

Ova će linija pokrenuti *radno čekanje* od 50 milisekundi sve dok uvjet `sprite.loaded()` – funkcija koja vraća `true` ili `false` ovisno o tome je li tražena slika istina – ne postane `true`. Važno je napomenuti da ovo funkcionira samo s asinkronim radom, u sinkronom radu ostatak petlje igre i sučelja bi se *zamrznuo* sve dok se slika ne učita. S asinkronim radom samo ta funkcija privremeno usporava, odnosno privremeno zaustavlja izvođenje.

Sada kada imamo spreman *Sprite* pretvoren u komponentu s učitanom slikom, lako ju možemo dodati na zaslon:

```
add(sprite);
```

Ako spremimo datoteku i pogledamo kako naša igra sad izgleda, vidjet ćemo sljedeće:



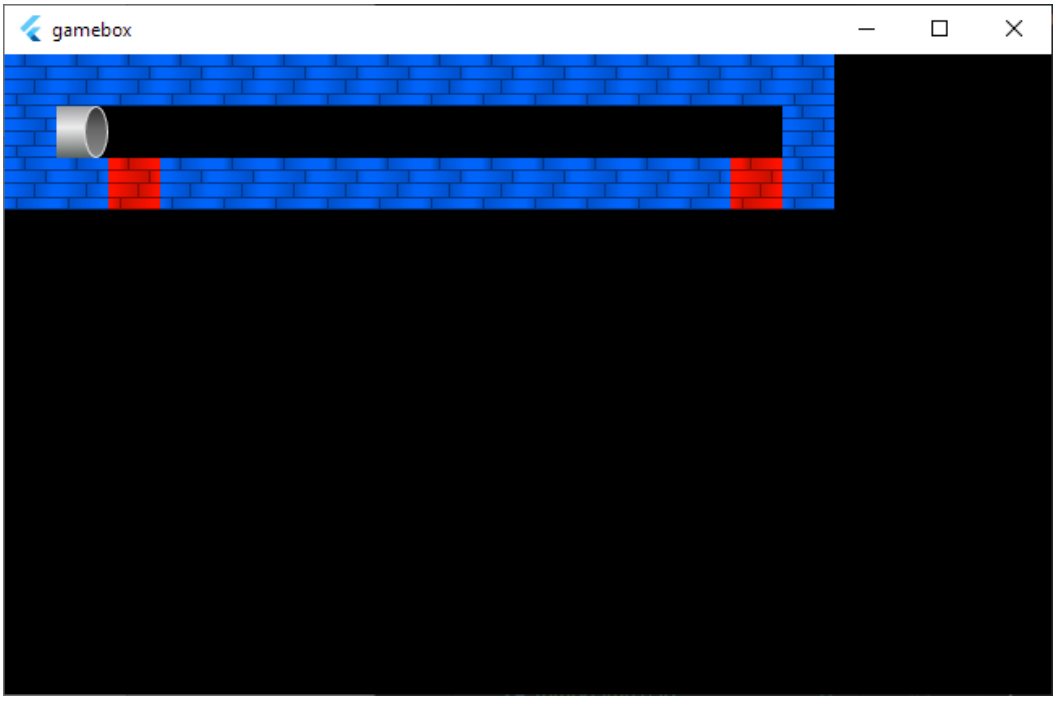
Slika 21: Prikaz igre s dodanim objektima (vlastita izrada)

Možemo primijetiti da su naši objekti dodani, ali svi su u gornjem lijevom kutu. Po *defaultu*, `Sprite` objekti su pozicionirani na (1, 1) koordinatu u svijetu igre. U objektu `obj` iz `for..in` petlje možemo dohvatiti x/y koordinate objekta (direktno u pikselima, pa nije potrebno množiti veličinom pločice).

Koordinate svijeta Spriteu možemo dodati na dva načina:

1. Prije dodavanja Spritea u igru pridružiti mu koordinate:
   ```
   sprite.x = obj.x;
   sprite.y = obj.y;
   add(sprite);
   ```
2. Značajkom Dart programskog jezika *cascade* odmah u funkciji dodavanja pridružiti koordinate objektu:
   ```
   add(sprite..x = obj.x..y = obj.y);
   ```

Odabran je drugi način, ali ishod za oba načina je isti:

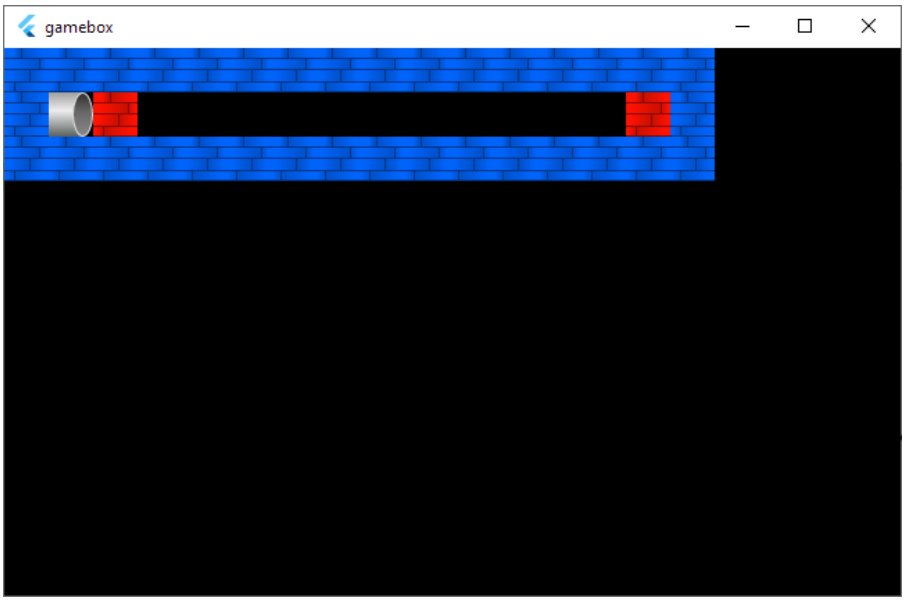


Slika 22: Objekti dodani u razinu (vlastita izrada)

Možemo primijetiti da su nam objekti dodani ispravno i da x koordinata odgovara, međutim y koordinata je pomaknuta za 1 blok dolje previše. To možemo lako popraviti oduzimanjem veličine pločice od y koordinate:

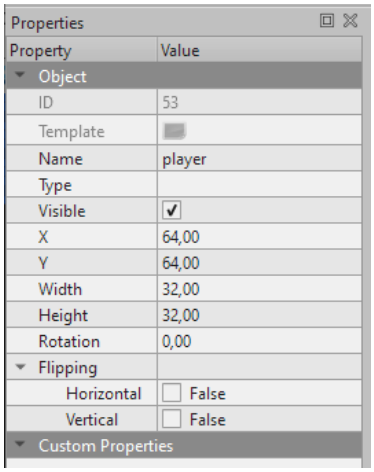
```
add(sprite..x = obj.x..y = obj.y - TILE_SIZE);
```

Ako sada pogledamo našu igru, možemo vidjeti da su nam objekti ispravno dodani:



Slika 23: Objekti dodani na ispravne pozicije (vlastita izrada)

Sada kada su nam objekti na ispravnim mjestima možemo se pozabaviti učitavanjem ispravnih grafika za svaki objekt. Pri kreiranju razine u *Tiled*u postavili smo atribut `name` u prozoru *Properties*:



Slika 24: Ime dodano u prozor *Properties* u *Tiled*u (vlastita izrada)

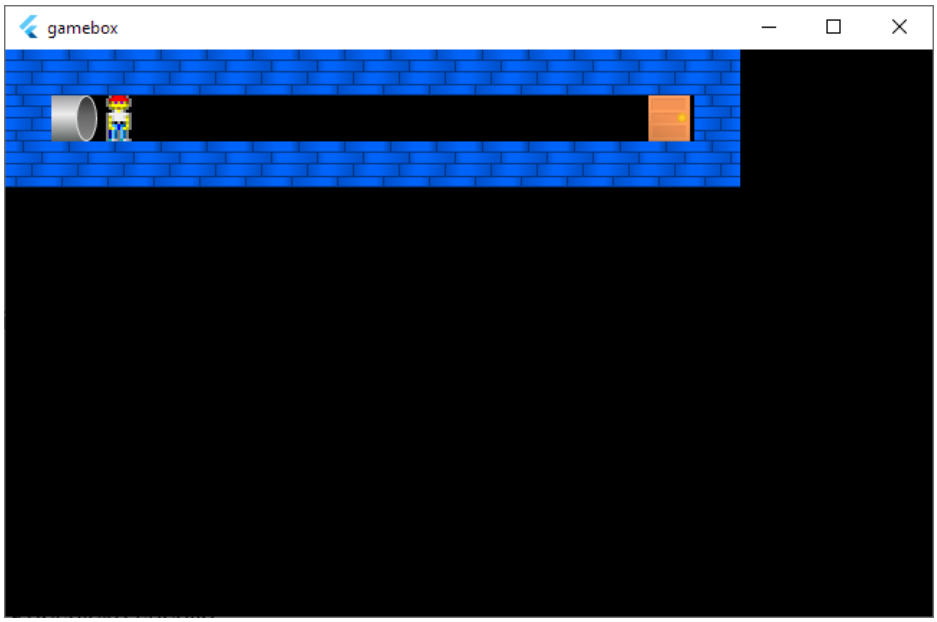
Parser nam je dao pristup tom imenu kroz `TmxObject` tip podataka koji koristimo za dohvaćanje ostalih podataka o objektu, pa tako jednostavnom `switch` selekcijom možemo ovisno o različitim nazivima odabrati različite koordinate koje ćemo koristiti pri kreiranju `Sprite` objekta.

Prije toga ćemo na početku bloka kôda `for..in` petlje dodati varijablu `offset` tipa `Offset` kojom ćemo izbjeći kopiranje i lijepljenje istog dijela koda za kreiranje `Sprite`a. Također,

moramo promijeniti kreiranje `Sprite`a da uključuje podatke iz varijable `offset`. Naša `for..in` petlja sada izgleda ovako:

```
for (TmxObject obj in objGroup.tmxObjects) {
      Offset offset;

      switch (obj.name) {
        case "player":
          offset = Offset(1, 8);
          break;
        case "door":
          offset = Offset(5, 3);
          break;
      }

      Sprite _sprite = Sprite(
        'DangerousDaveRedux.png',
        x: (offset.dx - 1) * TILE_SIZE,
        y: (offset.dy - 1) * TILE_SIZE,
        width: TILE_SIZE,
        height: TILE_SIZE,
      );
      SpriteComponent sprite =
          SpriteComponent.fromSprite(TILE_SIZE, TILE_SIZE, _sprite);

       while (!sprite.loaded()) await Future.delayed(
           Duration(milliseconds: 50));

      add(sprite
        ..x = obj.x
        ..y = obj.y - TILE_SIZE);
    }
```

Nakon što spremimo promjene i pogledamo igru, možemo vidjeti da su sada učitane ispravne teksture:

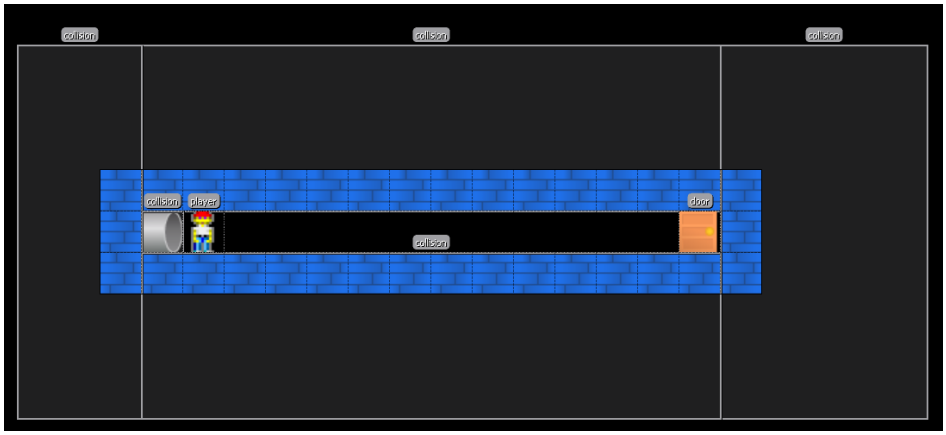


Slika 25: Ispravno učitane teksture objekata (vlastita izrada)

Sada kada možemo ispravno učitati razinu, vrijeme je da igri dodamo interaktivnosti.

6.1.5. Kolizije i sudarači

Prije nego što nastavimo s izradom igre moramo dodati još jedan element našoj razini, a time i igri – kolizije. Ovo će biti niz pravokutnika koji definiraju mjesta na razini kroz koja igrač neće moći proći, odnosno s kojima će se sudarati. Kolizije ćemo dodati u *Tiled*u na sloj objekata i imenovati ih `collision`.



Slika 26: Kolizije dodane u razinu

Iako smo pravokutnike kolizije mogli dodati samo na mjesta koja zauzimaju pločice, oni pokrivaju malo veće područje da bi se izbjegli mogući *glitchevi* uslijed sitnih odstupanja u kalkulacijama, koji mogu dovesti do neželjenih ponašanja unutar igre.

Sudarače nećemo prikazivati, tako da je unutar petlje za učitavanje objekata potrebno dodati liniju

```
if (obj.name == "collision") continue;
```

kako bismo ignorirali objekte s imenom `collision`.

6.1.6. Upravljanje igračem

Da bismo smanjili kompleksnost kôda u odvojenoj datoteci napravit ćemo klasu `Player`, koja proširuje klasu `PositionComponent` koja će nam olakšati upravljanje pozicijom igrača dalje u kôdu.

Prvo ćemo definirati nekoliko konstanti i varijabli koje ćemo koristiti. Konstante koje će nam trebati su `SPEEDX` i `SPEEDY` koji definiraju brzinu kretanja u horizontalnom i vertikalnom smjeru. `SPEEDX` je definiran kao 1, što će označavati pomak za 1 piksel po svakom pomaku, a `SPEEDY` je poprimio vrijednost konstante `TILE_SIZE`, što je jednako dimenziji pločice. Od ostalih varijabli koje nam trebaju je referenca na glavnu igru `gameRef` tipa `BaseGame`, koja će nam trebati za namještanje pozicije kamere, varijabla `animation` tipa `Animation` koja će nam služiti za mijenjanje okvira animacije glavnog lika te varijabla `colliders` tipa `List<Rect>` koja sadrži popis pravokutnika sudarača. Svaki `Rect` ima definirane x/y kordinate gornje lijeve točke, širinu i visinu.

Zadnje varijable koje ćemo definirati su privatne varijable `_directionX` i `_speedY` te varijabla `collisionResults` koja će spremati podatke o kolizijama igrača. Varijabla `_directionX` nam govori u kojem smjeru igrač ide:

- `-1` – kreće se lijevo
- `0` – stoji na mjestu
- `1` – kreće se desno

Varijabla `_speedY` sadrži trenutnu vertikalnu brzinu igrača:

- `<0` – pada
- `0` – stoji na mjestu
- `>0` – skače (ide gore)

Kako bismo dodali funkcionalnost igraču moramo nadjačati metodu `update` bazne klase. Ona će se izvršavati 60 puta u sekundi i ažurirati sve podatke o trenutnom stanju igre.

Prvo što moramo napraviti je provjeriti kolizije. Na primjer, ako je igrač na zemlji u koliziji je s donjom stranom svog lika, pa onda znamo da ne može ići niže od toga. Ako je igrač u koliziji s lijeve strane, ne smijemo dopustiti kretanje ulijevo. Kolizije provjeravamo sljedećom metodom:

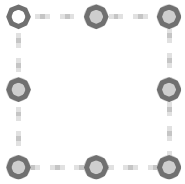
```
Map<CollisionPos, bool> isColliding(Rect r, Offset topLeft,
    {double offset = TILE_SIZE}) {
  var output = Map<CollisionPos, bool>();

  var topRight = Offset(topLeft.dx + offset, topLeft.dy);
  var bottomLeft = Offset(topLeft.dx, topLeft.dy + offset);
  var bottomRight = Offset(topLeft.dx + offset, topLeft.dy + offset);

  var top = Offset(topLeft.dx + offset / 2, topLeft.dy);
  var left = Offset(topLeft.dx, topLeft.dy + offset / 2);
  var bottom = Offset(topLeft.dx + offset / 2, topLeft.dy + offset);
  var right = Offset(topLeft.dx + offset, topLeft.dy + offset / 2);

  output[CollisionPos.topLeft] = r.contains(topLeft);
  output[CollisionPos.topRight] = r.contains(topRight);
  output[CollisionPos.bottomLeft] = r.contains(bottomLeft);
  output[CollisionPos.bottomRight] = r.contains(bottomRight);
  output[CollisionPos.top] = r.contains(top);
  output[CollisionPos.bottom] = r.contains(bottom);
  output[CollisionPos.left] = r.contains(left);
  output[CollisionPos.right] = r.contains(right);

  return output;
}
```

Ona kao parametre prima pravokutnik `r` koji predstavlja pravokutnik kolizije i točku `topLeft` koja predstavlja x/y koordinatu igrača. Postoji i treći opcionalni parametar `offset`, koji služi za fino ugađanje u posebnim slučajevima, a inicijalno je postavljen na vrijednost `TILE_SIZE`. Metoda vraća *mapu*, odnosno kolekciju uređenih parova (strana, `bool`) iz koje možemo iščitati s koje strane objekt je ili nije u koliziji sa zadanim pravokutnikom.

`CollisionPos` je jednostavna enumeracija koja sadrži sve strane koje testiramo za koliziju:

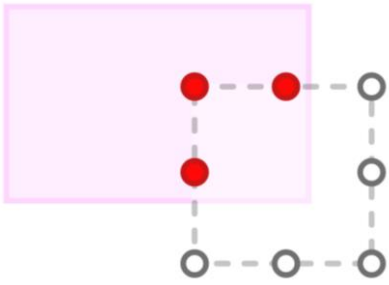
```
enum CollisionPos {
  top,
  left,
  right,
  bottom,
  topLeft,
  bottomLeft,
  topRight,
  bottomRight,
  center
}
```

Metoda radi tako da iz gornje lijeve točke izračuna koordinate ostalih točaka:



Slika 27: Točke koje se gledaju u koliziji, izračunate iz izvorišne točke (vlastita izrada)

Nakon generiranja točaka prolazi kroz svaka i *proširenom metodom* (eng. *extension method*) `contains` provjerava nalazi li se ta točka unutar pravokutnika `r`. Ako se nalazi tu poziciju označava kao `true`, a inače `false`.



Slika 28: Provjera kolizije nad pravokutnikom (vlastita izrada)

Sada kada imamo metodu za provjeru rezultata kolizije možemo prvo *resetirati* vrijednost varijable a zatim proći kroz sve pravokutnike kolizije učitane iz datoteke razine i ažurirati vrijednost varijable collisionResults:

```
collisionResults = isColliding(Rect.zero, Offset(10, 10));
    for (var collider in colliders) {
      var _res = isColliding(collider, Offset(this.x, this.y));
      _res.forEach((key, value) {
        if (value == true) {
          collisionResults.update(key, (value) => true);
        }
      });
    }
```

Nakon prikupljenih i spremljenih rezultata analize kolizije možemo ih odmah početi i koristiti. Prvo što ćemo napraviti je blokirati mogućnost kretanja ulijevo, odnosno udesno ako je igrač u koliziji na tim stranama:

```
if (_directionX == -1 && collisionResults[CollisionPos.left])
    _directionX = 0;
if (_directionX == 1 && collisionResults[CollisionPos.right])
    _directionX = 0;
```

Sljedeće što ćemo napraviti je promijeniti trenutni okvir (eng. *frame*) naše animacije. Da se prisjetimo, imamo ukupno četiri okvira animacije za igrača – jedan za stajanje na mjestu, dva za hodanje i jedan za skakanje. Ako nismo u koliziji s donjom stranom znači da smo u zraku pa ćemo trenutni okvir prebaciti na taj. Ako smo u koliziji s donjom stranom i vrijednost varijable `_directionX` je `0` znači da stojimo na mjestu pa možemo postaviti okvir na pripadajuću vrijednost. Ako nijedan od ovih uvjeta nije ispunjen, možemo naizmjenice postavljati dva okvira animacije da imamo efekt hodanja. To u kôdu izgleda ovako:

```
if (!collisionResults[CollisionPos.bottom]) {
      animation.currentIndex = 3;
    } else if (_directionX == 0) {
      animation.currentIndex = 0;
    } else {
      animation.currentIndex = 1;
      animation.update(t);
      if (animation.currentIndex > 2) animation.currentIndex = 1;
    }
```

Nakon promjene okvira animacije počinjemo s kalkulacijom pokreta igrača. Inicijalizirat ćemo dvije varijable – `stepY` i `stepX`, koje nam služe kao kalkulacija trenutne brzine. Da kretnja po y osi bude realističnija umjesto konstantne akceleracije koristit ćemo prirodniju. Prvotna ideja je bila da se i kretanje po x osi napravi na isti način, ali osjećaj kretanja je bio previše drugačiji od originalne igre, pa je odlučeno da će x os ići po konstantnoj akceleraciji. Ove varijable definirane su ovako:

```
final stepY = _speedY * t - SPEEDY * t * t / 2;
var stepX = SPEEDX * _directionX;
```

`t` je varijabla koju nam pruža *Flame*. To je vrijednost tipa `double` koja predstavlja vrijeme od zadnjeg poziva metode `update`. Nakon ovih kalkulacija možemo igračevoj x, odnosno y koordinati dodati ili oduzeti izračunate vrijednosti, što će igraču dati prikaz kretnje:

```
  this._speedY += SPEEDY * t;
    x += stepX;
    if (!collisionResults[CollisionPos.bottom]) y += stepY;
    if (collisionResults[CollisionPos.top] &&
        !collisionResults[CollisionPos.bottom]) y -= stepY - 2;
```

Zadnje dvije `if` selekcije nam služe da pretvorimo kretnju prema gore u kretnju prema dolje kada igrač udari u prepreku iznad.

Na kraju sve što je preostalo je centrirati kameru na igrača, a to možemo učiniti vrlo jednostavno:

```
gameRef.camera.x = this.x - gameRef.size.width / 2;
gameRef.camera.y = this.y - gameRef.size.height / 2;
```

Da bi kompletirali implementaciju igrača, moramo nadjačati metodu `render` koja će iscrtati igrača na zaslon. Ona je u ovom slučaju vrlo jednostavna, sve što treba napraviti je pozvati metodu `renderPosition` i dati joj pripadajuće parametre da bi se igrač iscrtao na zaslonu:

```
@override
  void render(Canvas canvas) =>
      animation.getSprite().renderPosition(canvas,
          Position(this.x, this.y));
```

Sada imamo igrača koji se može kretati po razini možemo se pozabaviti dodavanjem još vrsta objekata i mehanika. Prvo što ćemo napraviti je proširiti sustav iscrtavanja da si pojednostavimo daljnji rad.

6.1.7. Sustav adaptivnog učitavanja razine

Da bismo učinili učitavanje razine fleksibilnije, kôd učitavanja razine prebacit ćemo u odvojenu klasu, te uz nju napraviti još nekoliko pomoćnih enumeracija, klasa i metoda.

Počet ćemo dodavanjem enumeracije ItemType kojom ćemo na jednom mjestu definirati sve vrste objekata koje možemo imati u igri:

```
enum ItemType {
  door,
  trophy,
  bonusPoints,
  deadly,
  map,
  collision,
  enemy,
  player,
  playerBullet,
  enemyBullet }
```

Sada možemo napraviti pomoćne klase koje ćemo koristiti. Počet ćemo s apstraktnom klasom `RenderHelper` koja će biti baza za ostale klase. Ona će sadržavati inicijalne x/y koordinate objekta u prostoru, definiciju tipa objekta i zastavice `isAnimated` koja govori je li objekt animiran ili statične slike. U kôdu to izgleda ovako:

```
abstract class RenderHelper {
  final double x;
  final double y;
  final ItemType type;
  final bool isAnimated;

  RenderHelper(this.x, this.y, this.type, this.isAnimated);
}
```

Nakon bazne apstraktne klase možemo napraviti četiri konkretne klase koje koristimo kao pomoć pri iscrtavanju razine i upravljanju objektima.

`SpriteRenderHelper` je pomoćni objekt za iscrtavanje objekata koji su pomični, ali nemaju animiranu sliku (kao npr. vrata, dijamanti i sl.) Uz bazne varijable sadrži i varijablu `sprite` tipa `SpriteComponent` koja predstavlja objekt s grafikom i koju ćemo moći poslije iscrtavati:

```
class SpriteRenderHelper extends RenderHelper {
  final SpriteComponent sprite;
  SpriteRenderHelper(this.sprite, x, y, type) : super(x, y, type, false);
}
```

`AnimCompRenderHelper` je pomoćni objekt za iscrtavanje objekata koji su pomični i imaju animiranu sliku (kao npr. igrač) Uz bazne varijable sadrži i varijablu `comp` tipa `AnimationComponent` koja je namijenjena lakšem upravljanju objekata sa *spritesheet* animacijama:

```
class AnimCompRenderHelper extends RenderHelper {
  final AnimationComponent comp;
  AnimCompRenderHelper(this.comp, x, y, type) : super(x, y, type, true);
}
```

`CollisionRenderHelper` je pomoćni objekt za upravljanje pravokutnikom kolizije. Uz bazne varijable sadrži visinu i širinu pravokutnika kolizije:

```
class CollisionRenderHelper extends RenderHelper {
  final double width;
  final double height;
  CollisionRenderHelper(this.width, this.height, double x, double y)
      : super(x, y, ItemType.collision, false);
}
```

Zadnja klasa je `MapRenderHelper` je pomoćni objekt za iscrtavanje *sloja pločica* razine, u suštini omotač (eng. *wrapper*) za postojeći kôd za učitavanje koji smo koristili:

```
class MapRenderHelper extends RenderHelper {
  final TiledComponent tiledMap;
  MapRenderHelper(this.tiledMap) : super(0.0, 0.0, ItemType.map, false);
```

```
}
```

Nakon deklariranja pomoćnih klasa za učitavanje i iscrtavanje objekata možemo napraviti klasu `GameLevel` koja služi za prikaz razine. Sastoji se od varijable koja predstavlja putanju do *.tmx* datoteke razine i redni broj razine. Uz to će imati asinkronu metodu `getItemsToRender` koja priprema sve objekte koje treba isrtavati i vraća ih u obliku liste koju možemo proslijediti našoj glavnoj igri da ih iscrta. U toj metodi bit će sadržan sav kôd za učitavanje razine koji smo dosad koristili, proširen dodatnim tipovima objekata:

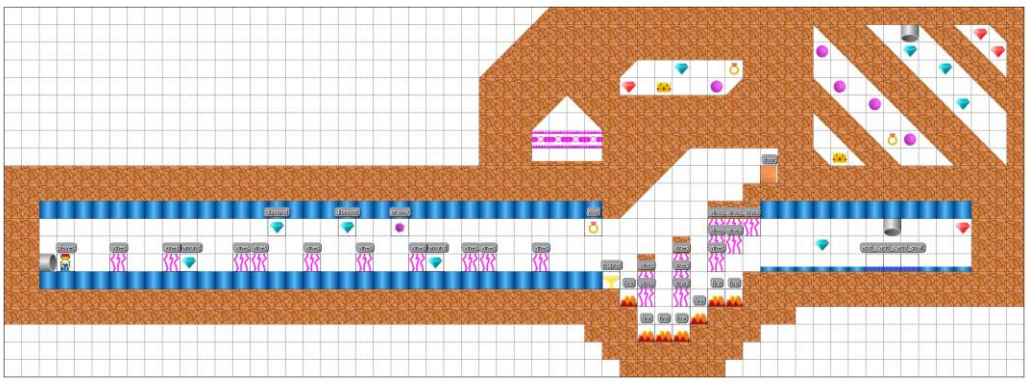
```
class GameLevel {
  final int number;
  final String filename;
  GameLevel({@required this.number, @required this.filename});

  Future<List<RenderHelper>> getItemsToRender() async {
    List<RenderHelper> itemsToRender = [];

    // Sloj pločica
    TiledComponent _map = TiledComponent(filename,
                          Size(TILE_SIZE, TILE_SIZE));
    itemsToRender.add(MapRenderHelper(_map));

    // Sloj objekata
    final ObjectGroup objGroup = await _map.getObjectGroupFromLayer("objects");

    for (TmxObject obj in objGroup.tmxObjects) {
      Offset offset;
      ItemType type;
      bool isAnimated = false;

      switch (obj.name) {
        case "player":
          offset = Offset(1, 8);
          type = ItemType.player;
          isAnimated = true;
```

```
  break;
case "enemy":
  offset = Offset(5, 8);
  type = ItemType.enemy;
  isAnimated = false;
  break;
case "door":
  offset = Offset(5, 3);
  type = ItemType.door;
  isAnimated = false;
  break;
case "diamond":
  offset = Offset(7, 1);
  type = ItemType.bonusPoints;
  isAnimated = false;
  break;
case "ring":
  offset = Offset(5, 6);
  type = ItemType.bonusPoints;
  isAnimated = false;
  break;
case "vines":
  offset = Offset(1, 6);
  isAnimated = true;
  type = ItemType.deadly;
  break;
case "fire":
  offset = Offset(4, 2);
  isAnimated = true;
  type = ItemType.deadly;
  break;
case "acid_small":
  offset = Offset(1, 1);
```

```
    isAnimated = true;
    type = ItemType.deadly;
    break;
  case "trophy":
    offset = Offset(1, 3);
    isAnimated = true;
    type = ItemType.trophy;
    break;
  case "collision":
    type = ItemType.collision;
    break;
}

if (offset == null)
     offset = Offset(0, 0);

// Kolizije
if (type == ItemType.collision) {
  itemsToRender.add(CollisionRenderHelper(obj.width,
          obj.height, obj.x, obj.y));
}

// Animirane komponente
else if (isAnimated) {
       AnimationComponent comp = AnimationComponent(
    32,
    32,
    Animation.sequenced(
      'DangerousDaveRedux.png',
      type == ItemType.trophy || type == ItemType.player ? 4 : 3,
      textureX: (offset.dx - 1) * TILE_SIZE,
      textureY: (offset.dy - 1) * TILE_SIZE,
      textureWidth: TILE_SIZE,
```

```
            textureHeight: TILE_SIZE,
          ),
        );

        itemsToRender.add(AnimCompRenderHelper(comp, obj.x,
                    obj.y - TILE_SIZE, type));
      } else {
        Sprite _sprite = Sprite(
          'DangerousDaveRedux.png',
          x: (offset.dx - 1) * TILE_SIZE,
          y: (offset.dy - 1) * TILE_SIZE,
          width: TILE_SIZE,
          height: TILE_SIZE,
        );
        SpriteComponent sprite =
            SpriteComponent.fromSprite(TILE_SIZE, TILE_SIZE, _sprite);

        while (!sprite.loaded())
          await Future.delayed(Duration(milliseconds: 50));
        itemsToRender.add(SpriteRenderHelper(sprite, obj.x,
           obj.y - TILE_SIZE, type));
      }
    }

    return itemsToRender;
  }
}
```

Sada kada imamo sustav adaptivnog učitavanja razine možemo početi integrirati sve komponente u jednu smislenu cjelinu. Također, vrijeme je da napravimo jednu kompleksniju razinu s više elemenata i ona izgleda ovako:



Slika 29: Kompleksnija razina (vlastita izrada)

6.1.8. Integracija u igru

Možemo početi integrirati elemente igrača i sustava adaptivnog učitavanja razine u jednu cjelinu. U početnoj klasi gdje smo započeli implementaciju igre maknut ćemo sav kôd za učitavanje razina i uz konstantu `TILE_SIZE` dodati još nekoliko varijabli:

- `lives` – broj preostalih života
- `points` – broj prikupljenih bodova
- `player` – referenca na objekt igrača
- `levels` – popis s objektima `GameLevel` koji opisuju razine

Također, počet ćemo ubacivati naše dvije razine kao objekte tipa `GameLevel`. Da bismo omogućili promjenu razina morat ćemo nakon svake odigrane razine ponovo iscrtati *widget* s igrom. To ćemo najlakše napraviti tako da *widget* s igrom *zamotamo*, odnosno povežemo s *widgetom* koji može mijenjati stanje. Biblioteka *hooks* nam to na lak način omogućava. Unutar `build` metode tog *widgeta* možemo instancirati objekt stanja koristeći funkciju `useState`, nakon čega ćemo taj objekt poslati našoj igri:

```
class DangerousDaveWidget extends HookWidget {
  @override
  Widget build(BuildContext context) {
    final level = useState(0);
    final _DangerousDave game = _DangerousDave(level);
    return Scaffold(body: game.widget);
  }
}
```

Ne smijemo zaboraviti promijeniti klasu naše igre da može prihvatiti taj dodatni parametar, deklarirajući varijablu i dodavanjem konstruktora.

Unutar klase deklarirat ćemo još jednu varijablu – `playerOffset` – kojom ćemo spremati x/y koordinatu početne pozicije igrača koju ćemo poslije koristiti.

Također, da imamo lak način za praćenje svih objekata u igri, deklarirat ćemo nekoliko lista i objekata koji su direktno vezani za objekte unutar razine:

- `deadlies` – popis objekata koji trenutačno usmrte igrača
- `collectibles` – popis objekata koje možemo pokupiti i koji nam daju bodove
- `collisions` – popis pravokutnika kolizije
- `enemies` – popis protivnika
- `playerBullet` – referenti objekt za metak igrača
- `enemyBullets` – popis projektila koje protivnici pucaju
- `trophy` – referenca na kalež/ćup koji treba pokupiti da bi se pojavila vrata
- `door` – referenca na objekt vrata kojim dovršavamo razinu

Sada možemo napraviti asinkronu metodu `initGame` koja će učitavati sve objekte i postavljati scenu za igru, kao i davati funkcionalnosti objektima. Prvo dohvaćamo podatke o trenutnoj razini, nakon čega inicijaliziramo varijable i objekte na prazne vrijednosti (prazno polje za polja i `null` za objekte) i zatim dohvaćamo objekte za iscrtavanje na zaslon. Sâmo iscrtavanje delegirat ćemo metodi `renderItems`. U kôdu to izgleda ovako:

```
void initGame() async {
  GameLevel gameLevel = levels[level.value];


  collectibles = [];
  deadlies = [];
  collisions = [];
  enemies = [];
  playerBullet = null;
  enemyBullets = [];


  List<RenderHelper> items = await gameLevel.getItemsToRender();
```

```
    renderItems(items);
  }
```

Metoda `renderItems` služit će za iscrtavanje objekata na zaslon. Na početku ona će ukoniti igračeve i metke protivnika ako su već bili iscrtani (npr. u slučaju da igrač izgubi život pa se vrati na početak razine). Nakon toga prolazimo kroz svaki element i ovisno o njegovom tipu drukčije radimo.

Ako je objekt tima `CollisionRenderHelper` iz njega izvučemo koordinate, širinu, visinu i napravimo pravokutnik, koji zatim dodamo u popis objekata kolizije:

```
if (item is CollisionRenderHelper) {
  Rect _tempRect = Rect.fromLTWH(item.x, item.y, item.width, item.height);
  collisions.add(_tempRect);
  continue;
}
```

Ako je u pitanju igrač, odnosno objekt tipa `AnimCompRenderHelper` i gdje je `type` vrijednosti `ItemType.player`, prvo postavimo vrijednost varijable `playerOffset` na x/y koordinate igrača, zatim instanciramo objekt klase `Player` i pridružujemo ga globalnoj varijabli `player`, nakon čega ga dodajemo na scenu:

```
if (item is AnimCompRenderHelper && item.type == ItemType.player) {
        playerOffset = Offset(item.x, item.y);
        player = Player(this, collisions, item.comp.animation)
          ..x = item.x
          ..y = item.y
          ..width = TILE_SIZE
          ..height = TILE_SIZE;
        add(player);
        continue;
}
```

Ako su objekti tipa vrata (`ItemType.door`) ili neprijatelji (`ItemType.enemy`) za sada samo pridružujemo objekt globalnoj varijabli, odnosno dodajemo ga u listu:

```
if (item.type == ItemType.door) {
        door = item;
        continue;
```

```
      }
if (item.type == ItemType.enemy) {
    enemies.add(item);
    continue;
}
```

Ako je objekt tipa `MapRenderHelper` dodamo ga na scenu odmah, a ako je objekt tipa `SpriteRenderHelper` ili `AnimCompRenderHelper` dodajemo ga na scenu usput mu pridružujući x/y koordinate zapisane u *Helper* objektu:

```
if (item is SpriteRenderHelper) add(item.sprite..x = item.x..y = item.y);
else if (item is AnimCompRenderHelper)
   add(item.comp..x = item.x..y = item.y);
else if (item is MapRenderHelper) add(item.tiledMap);
```

Ako su u pitanju objekti koji mogu trenutačno usmrtiti igrača (`ItemType.deadly`), koje skupljamo da bismo dobili bodove (`ItemType.bonusPoints`) ili trofej (`ItemType.trophy`) dodajemo ih u pripadajuće liste, odnosno pridružujemo ih pripadajućim varijablama:

```
if (item.type == ItemType.deadly) deadlies.add(Offset(item.x, item.y));
if (item.type == ItemType.bonusPoints) collectibles.add(item);
if (item.type == ItemType.trophy) trophy = item;
```

Nakon dodavanja objekata moramo provjeriti postoji li trofej unutar razine, ako ne postoji (kao npr. kod razine *Hodnik*), moramo iscrtati vrata da bi igrač mogao završiti razinu:

```
if (trophy == null && door != null) {
      add(door.sprite..x = door.x..y = door.y);
}
```

Na kraju na zaslon iscrtavamo protivnike:

```
for (SpriteRenderHelper enemy in enemies) {
      add(enemy.sprite
        ..x = enemy.x
        ..y = enemy.y + TILE_SIZE * 1.5
        ..anchor = Anchor.center
      );
}
```

6.1.9. Dodavanje funkcionalnosti igri

Sada kada imamo sve potrebne objekte iscrtane i reference na njih spremljene u odgovarajuće varijable, odnosno polja, možemo početi dodavati funkcionalnost igri. To ćemo činiti nadjačavanjem metode `update`.

Prvo što moramo napraviti je provjeriti preklapa li se objekt igrača s objektom protivnika ili objektom koji trenutačno ubija igrača. Ako se to dogodi pozvat ćemo metodu `die` koja će predstavljati smrt igrača (zasada je prazna). Da bi izbjegli dupliciranje kôda napisat ćemo metodu koju možemo koristiti za provjeru preklapanja dva objekta. Ona će raditi slično kao i metoda koju smo napisali za kolizije, samo ovdje provjerava razmak između dvije točke a ne točke i pravokutnika. Uz to ćemo napisati i metodu koja uspoređuje razmak između igrača i objekta, a u suštini poziva prvu metodu i kao drugi objekt uzima koordinate igrača. Obje funkcije imaju opcionalni parametar `sensitivity`, koji možemo koristiti ako provjeravamo koliziju između objekata koji nisu veličine pune pločice, kao npr. projektili i metci. Funkcije u kôdu izgledaju ovako:

```
bool isItemOverlapping(Offset item1, Offset item2,
    {double sensitivity = TILE_SIZE * 0.9}) {
  double distX = (item1.dx - item2.dx).abs();
  double distY = (item1.dy - item2.dy).abs();

  return (distX < sensitivity && distY < sensitivity);
}

bool isPlayerOverlapping(Offset item, {double sensitivity = TILE_SIZE * 0.9}) {
  return isItemOverlapping(item, Offset(player.x, player.y));
}
```

Sada te metode možemo iskoristiti unutar metode `update`:

```
for (RenderHelper item in enemies) {
      if (item is SpriteRenderHelper &&
          isPlayerOverlapping(Offset(item.sprite.x, item.sprite.y))) {
        die();
      }
      if (item is AnimCompRenderHelper &&
```

```
          isPlayerOverlapping(Offset(item.comp.x, item.comp.y))) {
        die();
      }
    }
    for (Offset item in deadlies) {
      if (isPlayerOverlapping(item)) { die(); }
    }
```

Sljedeće na redu je dodavanje funkcionalnosti objektima koje možemo prikupiti. Ovo je prilično jednostavno: provjerimo preklapa li se igrač i željeni objekt, označimo objekt za uklanjanje sa scene u sljedećem pozivu funkcije `render` te dodamo igračevim bodovima određeni broj bodova:

```
for (RenderHelper item in collectibles.toList()) {
      if (isPlayerOverlapping(Offset(item.x, item.y))) {
        if (item is SpriteRenderHelper) markToRemove(item.sprite);
        if (item is AnimCompRenderHelper) markToRemove(item.comp);
        collectibles.remove(item);
        points += 500;
      }
    }
```

Nakon toga možemo dodati funkcionalnost skupljanja trofeja. Ako se igrač i trofej preklapaju označimo trofej za uklanjanje, njegovu varijablu postavimo na `null` da znamo da je pokupljen i postavimo vrata za dodavanje u sljedećem ciklusu igre:

```
if (trophy != null && door != null &&
    isPlayerOverlapping(Offset(trophy.x, trophy.y))) {
      markToRemove(trophy.comp);
      trophy = null;


      addLater(door.sprite..x = door.x..y = door.y);
}
```

Funkcionalnost završavanja razine i pobjede igre je sljedeća na redu. Prvo trebamo provjeriti je li zastavica levelDone vrijednosti false, da vrata i trophy nisu null i da se igrač preklapa s objektom vrata. Ako da, postavimo zastavicu levelDone na true, dodamo 1000 bodova igraču i nakon toga provjeravamo je li ovo bila zadnja razina igre. Ako je, prikazujemo

zaslon pobjede, ako ne povećavamo broj razine za 1 što će reaktivno pokrenuti sljedeću razinu:

```
if (!levelDone && door != null && trophy == null &&
        isPlayerOverlapping(Offset(door.x, door.y))) {
      levelDone = true;
      points += 1000;
      if (level.value == levels.length - 1) {
        addWidgetOverlay(
          'winScreen',
          VictoryScreen(
            onPressed: () {
              removeWidgetOverlay('winScreen');
              quitGame();
            },
          ),
        );
      } else {
        level.value++;
      }
    }
```

Zadnje što moramo isprogrmirati je funkcionalnost metaka igrača i metaka/projektila protivnika. Ako se protivnikov metak/projektil poklopi s igračem poziva se funkcija `die`, koja je vrlo jednostavna, iscrtava čestice na mjestu igrača, smanjuje broj života, a ako je on 0 prikazuje zaslon izgubljene igre i vraća bodove i živote na početne vrijednosti:

```
void die() {
    drawExplosionParticles(Offset(player.x, player.y));
    player.x = (playerOffset.dx);
    player.y = (playerOffset.dy);

    lives--;
    if (lives == 0) {
      points = 0;
      lives = 3;
```

```
      addWidgetOverlay(
        'deathScreen',
        drawDeathScreen(),
      );
    }
  }
```

Sada možemo napisati kôd za igračev metak, ako on postoji (tj. nije `null`) pomičemo ga udesno za određeni broj piksela. Nakon toga kreiramo zastavicu `removed` koja će nam trebati u nastavku. Sada prolazimo kroz popis svih neprijatelja i provjeravamo je li metak u koliziji s neprijateljem. Ako je, uklanjamo metak i protivnika sa zaslona, objekt protivnika brišemo iz liste neprijatelja, te postavljamo varijablu `playerBullet` na `null` i zastavicu `removed` na `true`. Na kraju iscrtavamo čestice na mjestu protivnika za vizualni efekt. Sada provjeravamo stanje zastavice `removed`. Ako je vrijednosti `true`, uklanjamo sve metke tog protivnika sa zaslona. Ako je vrijednost `false` i metak je prešao određenu udaljenost uklanjamo ga sa scene i omogućavamo igraču pucanje novog metka postavljanjem varijable `playerBullet` na `null`:

```
if (playerBullet != null) {
      playerBullet.sprite.x += 1 + t + 3;
      bool removed = false;

      for (SpriteRenderHelper enemy in enemies.toList()) {
        if (isItemOverlapping(Offset(enemy.sprite.x, enemy.sprite.y),
            Offset(playerBullet.sprite.x, playerBullet.sprite.y),
            sensitivity: TILE_SIZE * 0.5)) {
          markToRemove(playerBullet.sprite);
          markToRemove(enemy.sprite);
          enemies.remove(enemy);
          playerBullet = null;
          removed = true;

          drawExplosionParticles(Offset(enemy.sprite.x, enemy.sprite.y));
        }
```

```
    }

    if (removed) {
      for (SpriteRenderHelper bullet in enemyBullets.toList()) {
        markToRemove(bullet.sprite);
      }
      enemyBullets.clear();
    }

    if (!removed && playerBullet.sprite.x > 1200) {
      markToRemove(playerBullet.sprite);
      playerBullet = null;
    }
  }
```

Na redu je dodavanje funkcionalnosti metku protivnika. Prvo pomičemo metak ulijevo za određeni pomak, nakon toga provjeravamo preklapa li se metak s igračem. Ako se preklapa, pozovemo metodu `die` te uklanjamo objekt metka sa zaslona i iz liste metaka protivnika. Na kraju provjeravamo ako je metak došao do početka zaslona, ako je uklanjamo ga:

```
for (SpriteRenderHelper blt in enemyBullets.toList()) {
    blt.sprite.x -= 2 + t;

    if (isPlayerOverlapping(
      Offset(blt.sprite.x, blt.sprite.y), sensitivity: 16)) {
      die();
      markToRemove(blt.sprite);
      enemyBullets.remove(blt);
    }

    if (blt.sprite.x < TILE_SIZE) {
      markToRemove(blt.sprite);
      enemyBullets.remove(blt);
    }
  }
```

Zadnje što moramo implementirati u metodi `update` je funkcionalnost pucanja neprijatelja. Da igra bude zanimljivija neprijatelj počinje pucati tek u određenom rasponu, koji je namješten da previše ne iznenadi igrača, ali da i igri da određeni element zahtjevnosti. Prvo neprijatelju dajemo nasumičnu rotaciju, kao i u originalnoj igri. Zatim izračunamo udaljenost igrača i protivnika. Ako je manja od određenog broja (plus dodatna nasumična udaljenost) i protivnik nije ispucao više od 2 metka protivnik ispucava metak nakon čega se dodaje vremensko kašnjenje da bi se spriječilo prebrzo pucanje:

```
for (SpriteRenderHelper enemy in enemies) {
      enemy.sprite.anchor = Anchor.bottomCenter;
      enemy.sprite.angle += t * 2;

      double distanceFromPlayer = (player.x - enemy.sprite.x).abs();

      if (distanceFromPlayer <= 300 + Random().nextInt(100) &&
          enemyBullets.length <= 2) {
        fireEnemyBullet(enemy);
        Future.delayed(Duration(seconds: (0.5 *
            Random().nextInt(4)).toInt()));
      }
    }
```

Metoda `fireEnemyBullet` je prilično jednostavna. Kao argument prima objekt neprijatelja. Nakon toga generira objekt tipa `Sprite` i njemu pripadnu komponentu, kao i nasumičnu y koordinatu te dodaje tu komponentu u listu metaka neprijatelja:

```
void fireEnemyBullet(SpriteRenderHelper enemy) {
  Sprite _sprite = Sprite(
    'DangerousDaveRedux.png',
    x: 7 * TILE_SIZE,
    y: 0 * TILE_SIZE,
    width: TILE_SIZE,
    height: TILE_SIZE,
  );
  SpriteComponent sprite =
      SpriteComponent.fromSprite(TILE_SIZE, TILE_SIZE, _sprite);
```

```
  int rand = Random().nextInt(TILE_SIZE.toInt() * 2);
  add(sprite
    ..x = enemy.sprite.x + TILE_SIZE / 2
    ..y = enemy.sprite.y - TILE_SIZE + rand);

  enemyBullets.add(SpriteRenderHelper(sprite,
      enemy.sprite.x + TILE_SIZE / 2,
      enemy.sprite.y - TILE_SIZE + rand, ItemType.enemyBullet));
}
```

Trebat će nam i metoda za ispucavanjem igračevog metka, koja radi na isti način kao i ona za protivnikov metak:

```
void fireBullet() {
  if (playerBullet != null) return;
  Sprite _sprite = Sprite(
    'DangerousDaveRedux.png',
    x: 7 * TILE_SIZE,
    y: 1 * TILE_SIZE,
    width: TILE_SIZE,
    height: TILE_SIZE,
  );
  SpriteComponent sprite =
      SpriteComponent.fromSprite(TILE_SIZE, TILE_SIZE, _sprite);

  add(sprite
    ..x = player.x + TILE_SIZE / 2
    ..y = player.y);

  playerBullet = SpriteRenderHelper(sprite, player.x + TILE_SIZE / 2,
      player.y + TILE_SIZE / 2, ItemType.playerBullet);
}
```

6.1.10. Dodatne funkcionalnosti

Igra je sad po funkcionalnosti kompletna, ali treba dodati još nekoliko značajki da bi izgledala više kao igra. Dodat ćemo korisničko sučelje s kontrolama i mogućnost upravljanja tipkovnicom.

Flame omogućava iscrtavanje bilo kojeg *widgeta* preko igre koristeći `addWidgetOverlay` metodu. Moramo joj proslijediti šifru kojom možemo poslije ukloniti dio sučelja i sâm *widget*. Poziv ćemo napraviti dvaput jer želimo sučelje na gornoj i donjoj strani zaslona, pa će nam biti zgodno to izdvojiti u metodu `addOverlays`, koju možemo dalje zvati po potrebi:

```
void addOverlays() {
  addWidgetOverlay(
    'top',
    Positioned(
      top: 20,
      right: 20,
      left: 20,
      child: Row(
        children: [
          ControllerButton(
            icon: FluentSystemIcons.ic_fluent_pause_regular,
            onTap: () => addWidgetOverlay(
              'pauseMenu',
              PauseMenu(
                quitAction: () {
                  removeWidgetOverlay('pauseMenu');
                  if (quitGame != null) quitGame();
                },
                unpauseAction: () {
                  removeWidgetOverlay('pauseMenu');
                },
```

```
            ),
          ),
        ),
        Spacer(),
        AcrylicIconTile(
          icon: lives > 0
              ? FluentSystemIcons.ic_fluent_heart_filled
              : FluentSystemIcons.ic_fluent_heart_regular,
          backColor: Colors.red,
          iconColor: Colors.white,
        ),
        SizedBox(width: 8),
        AcrylicIconTile(
          icon: lives > 1
              ? FluentSystemIcons.ic_fluent_heart_filled
              : FluentSystemIcons.ic_fluent_heart_regular,
          backColor: Colors.red,
          iconColor: Colors.white,
        ),
        SizedBox(width: 8),
        AcrylicIconTile(
          icon: lives > 2
              ? FluentSystemIcons.ic_fluent_heart_filled
              : FluentSystemIcons.ic_fluent_heart_regular,
          backColor: Colors.red,
          iconColor: Colors.white,
        ),
      ],
    ),
  ),
);
```

```
addWidgetOverlay(
  'bottom',
  Positioned(
    right: 20,
    bottom: 20,
    left: 20,
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      crossAxisAlignment: CrossAxisAlignment.end,
      children: [
        Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            Text(
              '$points',
              style: TextStyle(fontSize: 18),
            ),
            SizedBox(
              height: 8,
            ),
            ControllerButton(
              icon: FluentSystemIcons.ic_fluent_target_regular,
              onTap: () => fireBullet(),
            ),
          ],
        ),
        Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Row(
              children: [
                ControllerButton(
                  icon: FluentSystemIcons.ic_fluent_arrow_up_left_regular,
```

```
      onTapUp: (_) => player.stopMovingX(),
      onTapDown: (_) {
        player.jump();
        player.moveLeft();
      },
    ),
    SizedBox(width: 8),
    ControllerButton(
      icon: FluentSystemIcons.ic_fluent_arrow_up_regular,
      onTap: () => player.jump(),
    ),
    SizedBox(width: 8),
    ControllerButton(
      icon: FluentSystemIcons.ic_fluent_arrow_up_right_regular,
      onTapUp: (_) => player.stopMovingX(),
      onTapDown: (_) {
        player.jump();
        player.moveRight();
      },
    ),
  ],
),
SizedBox(height: 8),
Row(
  children: [
    ControllerButton(
      icon: FluentSystemIcons.ic_fluent_arrow_left_regular,
      onTapUp: (_) => player.stopMovingX(),
      onTapDown: (_) => player.moveLeft(),
    ),
    SizedBox(width: 8),
    SizedBox(height: 48, width: 48),
    SizedBox(width: 8),
```

```
                ControllerButton(
                  icon: FluentSystemIcons.ic_fluent_arrow_right_regular,
                  onTapUp: (_) => player.stopMovingX(),
                  onTapDown: (_) => player.moveRight(),
                ),
              ],
            ),
          ],
        ),
      ],
    ),
  );
}
```

To u sučelju izgleda ovako:

Slika 30: Korisničko sučelje igre s kontrolama (vlastita izrada)

Upravljanje tipkovnicom implementirat ćemo dodavanjem *mixina* `KeyboardEvents` glavnoj klasi igre. Također treba nadjačati onKeyEvent događaj koji nam daje informacije o pritisnutoj tipki, te o tome je li tipka pritisnuta (keyDown) ili otpušetna (keyUp):

```
@override
  void onKeyEvent(RawKeyEvent e) {
    final bool isKeyDown = e is RawKeyDownEvent;

    switch (e.data.logicalKey.debugName) {
      case "Arrow Up":
        player.jump();
        break;
      case "Arrow Left":
        if (isKeyDown)
          player.moveLeft();
        else
          player.stopMovingX();
        break;
      case "Arrow Right":
        if (isKeyDown)
          player.moveRight();
        else
          player.stopMovingX();
        break;
      case "Control Left":
      case "Control Right":
        fireBullet();
        break;
    }
  }
}
```

Dodano je još nekoliko sitnih funkcionalnosti koje neće biti opisane u sklopu ovog rada jer nisu ključne za igru. Time smo dovršili implementaciju našeg klona igre *Dangerous Dave*.

6.2. Tetris

Ova je igra napisana u „čistom“ Flutteru, bez korištenja biblioteke *Flame*.

Opis implementacije ići će kroz sljedeće korake:

- Izrada podatkovnog modela za rad s podatcima
- Definiranje blokova
- Implementacija igre
- Korisničko sučelje i upravljanje igrom

6.2.1. Izrada podatkovnog modela za rad s podatcima

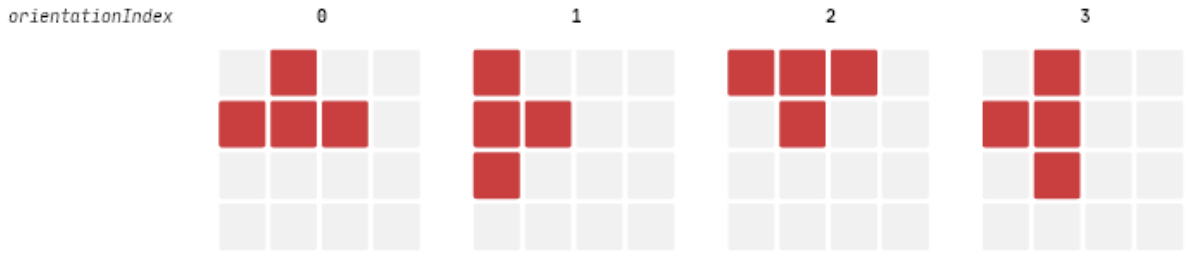
Prije opisa implementacija moramo definirate osnovne elemente koji će nam trebati – *blokove* i *atome*. Blok je osnovni element igre, a dolazi u više oblika. Sastavljen je od više *atoma*. Može se kretati u četiri smjera (gore, dolje, lijevo, desno) te rotirati u dva (u smjeru kazaljke na satu, u smjeru obrnutom od kazaljke na satu).

Slika 31: Osnovni elementi igre Tetris (vlastita izrada)

Atom ima vrlo jednostavnu implementaciju – x/y koordinate i boja:

```
class Atom {
  int x, y;
  Color color;

  Atom(int x, y, {Color color = Colors.transparent})
      : this.x = x,
        this.y = y,
        this.color = color;
}
```

Slijedi implementacija bloka. Prvo svojstvo je definicija mogućih orijentacija bloka. To je lista koja u sebi ima liste atoma (`List<List<Atom>>`). Taj oblik nam je pogodan jer lako i konzistentno možemo zapisati sve blokove koje imamo. Sljedeće od varijabli su x/y koordinate bloka i trenutni indeks rotacije. On nam govori koja je trenutna rotacija određenog bloka.



Slika 32: Rotacije bloka i orientationIndex (vlastita izrada)

Od metoda implementirat ćemo *gettere* za visinu, širinu, popis atoma i boju bloka, kao i *setter* za boju bloka. Definirat ćemo i enumeraciju `BlockMove` sa svim mogućim smjerovima pomicanja bloka. Nju ćemo koristiti u metodi `move` koja će raditi pomicanje bloka. Klasa `Block` i enumeracija `BlockMove` u kôdu izgledaju ovako:

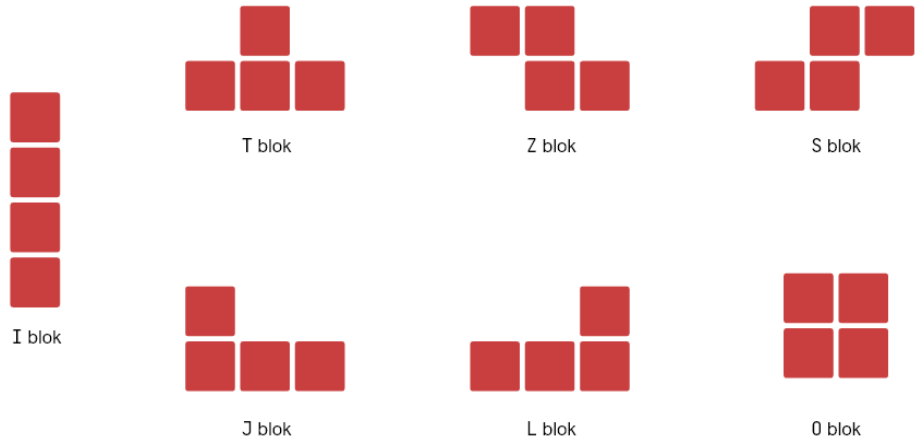
```
enum BlockMove { up, down, left, right, rot_right, rot_left }
class Block {
  List<List<Atom>> orientations = List<List<Atom>>();
  int x, y, orientationIndex;

  Block(this.orientations, Color color, this.orientationIndex) {
    x = 3;
    y = -height;
    this.color = color;
  }

  set color(Color color) {
    for (var orientation in orientations)
      for (var atom in orientation)
          atom.color = color;
  }
  Color get color => orientations[0][0].color;
  List<Atom> get atoms => orientations[orientationIndex];
```

```
  int get width => atoms.max((a, b) => a.x > b.x ? a.x : b.x).x + 1;
  int get height => atoms.max((a, b) => a.y > b.y ? a.y : b.y).y + 1;

  void move(BlockMove move) {
    if (move == BlockMove.up) y--;
    if (move == BlockMove.down) y++;
    if (move == BlockMove.left) x--;
    if (move == BlockMove.right) x++;
    if (move == BlockMove.rot_right)
      orientationIndex = ++orientationIndex % 4;
    if (move == BlockMove.rot_left)
      orientationIndex = (orientationIndex + 3) % 4;
  }
}
```

Sada imamo osnovne elemente za građenje blokova pa ih sada možemo iskoristiti za definiranje svih mogućih blokova i njihovih stanja.

6.2.2. Definiranje blokova

Blokove ćemo definirati proširivanjem klase `Block` i pozivanjem njenog konstruktora. U Tetrisu imamo sedam glavnih blokova:



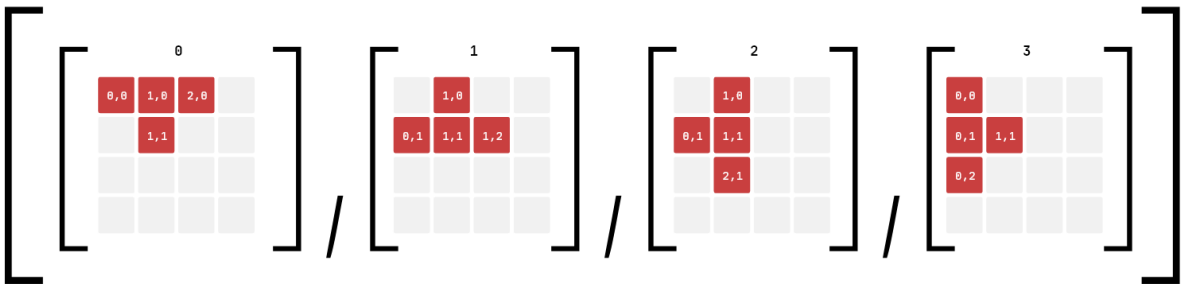
Slika 33: Tetris blokovi (vlastita izrada prema [24])

Zapis u kodu jednog bloka izgleda ovako:

```
class TBlock extends Block {
```

```
  TBlock(int orientationIndex)
      : super([
          [Atom(0, 0), Atom(1, 0), Atom(2, 0), Atom(1, 1)],
          [Atom(1, 0), Atom(0, 1), Atom(1, 1), Atom(1, 2)],
          [Atom(1, 0), Atom(0, 1), Atom(1, 1), Atom(2, 1)],
          [Atom(0, 0), Atom(0, 1), Atom(1, 1), Atom(0, 2)],
        ], Colors.blueAccent[400], orientationIndex);
}
```

Radi lakše vizualizacije isti blok kôda u grafičkom formatu izgleda ovako:



Slika 34: Grafički prikaz zapisa *T* bloka i svih njegovih orijentacija (vlastita izrada)

6.2.3. Implementacija igre

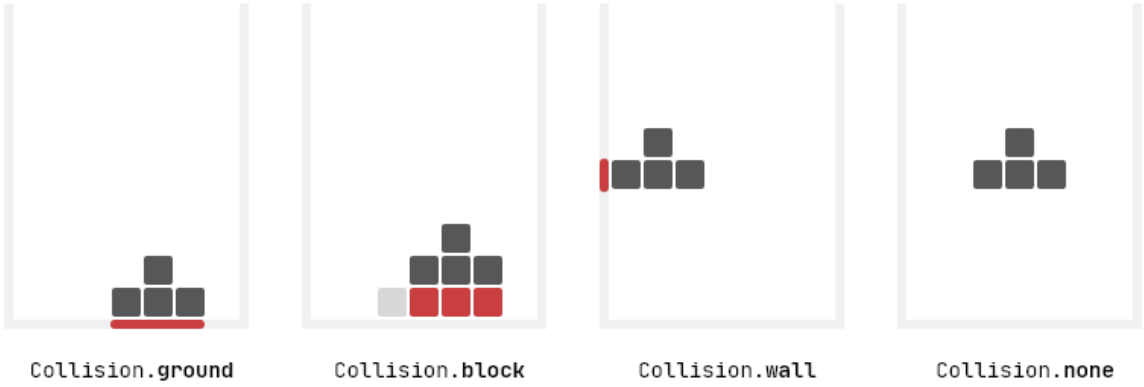
Prvo ćemo definirati nekoliko konstanti:

- `blocksX` – horizontalna veličina ploče za igru (u broju atoma)
- `blocksY` – vertikalna veličina ploče za igru (u broju atoma)
- `refreshRate` – broj milisekundi između osvježavanja ploče za igru

Definirat ćemo i enumeraciju `Collision` koja definira sva moguća stanja kolizije blokova:

```
enum Collision { ground, block, wall, none }
```

Te kolizije prikazane su na slici ispod:



Slika 35: Moguće vrste kolizija (vlastita izrada)

Za implementaciju igre dovoljan nam je jedan `StatefulWidget`, odnosno *widget* koji ima ugrađeno *stanje* (skup varijabli i odgovarajućih vrijednosti u nekom trenutku, podaci s kojima trenutno radimo).

Unutar *stanja widgeta* definirat ćemo nekoliko varijabli koje ćemo koristiti:

- `isPlaying` – je li igra u tijeku
- `duration` – frekvencija osvježavanja igrače ploče
- `boardKey` – *ključ*, odnosno referenca na objekt
- `tileSize` – veličina pločice
- `block` – podaci o sljedećem nadolazećem bloku
- `timer` – brojač proteklog vremena
- `action` – trenutni pomak (BlockMove) igrača
- `oldAtoms` – lista atoma blokova koji nisu cijeli
- `score` – broj bodova
- `isGameOver` – zastavica za kraj igre

Sada možemo metodom `startGame` definirati postavljanje igrače ploče. Inicijalizirati ćemo gore navedene varijable i postaviti *timer*:

```
void startGame() {
  isPlaying = true;
  isGameOver = false;
  score = 0;
  oldAtoms = List<Atom>();
```

```
    RenderBox renderBoxGame = _boardKey.currentContext.findRenderObject();
    tileSize = renderBoxGame.size.width / blocksX;
    block = getNewBlock();
    timer = Timer.periodic(duration, onPlay);
  }
```

Definirat ćemo i metodu za prekid igre `endGame`. Ona će poništiti *timer* i postaviti zastavicu `isPlaying` na `false`. Metodom `setState` dat ćemo signal Flutteru da se interno stanje *widgeta* promijenilo i da osvježi prikaz istog:

```
void endGame() {
  timer.cancel();

  setState(() {
    isPlaying = false;
  });
}
```

Metodom `updateScore` provjerit ćemo postoji li popunjeni red. Ako postoji, dodat ćemo ga u polje `oldAtoms` i igraću dodati bodova. Ako se kombinira više punih redova zaredom, povećava se *combo* što dovodi do više osvojenih bodova. To u kôdu izgleda ovako:

```
void updateScore() {
  var combo = 1;
  Map<int, int> rows = {};
  List<int> rowsToRemove = [];

  if (oldAtoms != null) {
    for (Atom element in oldAtoms) {
      rows.update(
        element.y,
        (value) => ++value,
        ifAbsent: () => 1,
      );
    }
```

```
  }

  rows.forEach((rowNum, count) {
    if (count == blocksX) {
      score += combo++;

      rowsToRemove.add(rowNum);
    }
  });

  if (rowsToRemove.isNotEmpty) {
    removeRows(rowsToRemove);
  }
}
```

Definicija pomoćne funkcije za brisanje redova je sljedeća. Prvo sortiramo polje redova koje treba ukloniti da bi krenuli odozgo prema dolje. Nakon toga prolazimo kroz redove koje treba obrisati i ostale blokove pomičemo dolje za jednu jedinicu. To u kôdu izgleda ovako:

```
void removeRows(List<int> rowsToRemove) {
  rowsToRemove.sort();
  for (var rowNum in rowsToRemove) {
    oldAtoms.removeWhere((oldAtom) => oldAtom.y == rowNum);

    for (Atom oldAtom in oldAtoms) {
      if (oldAtom.y < rowNum) {
        ++oldAtom.y;
      }
    }
  }
}
```

Nakon toga definirat ćemo nekoliko pomoćnih funkcija za provjeru sudara blokova sa zemljom, blokom iznad ili stranama igrače ploče:

```
bool checkAtBottom() => block.y + block.height == blocksY;
```

```
bool checkAboveBlock() {
  for (Atom oldAtom in oldAtoms) {
    for (var atom in block.atoms) {
      var x = block.x + atom.x;
      var y = block.y + atom.y;

      if (x == oldAtom.x && y + 1 == oldAtom.y) return true;
    }
  }
  return false;
}

bool checkOnEdge(BlockMove action) =>
    (action == BlockMove.left && block.x <= 0 ||
        action == BlockMove.right && block.x + block.width >= blocksX);
```

Slijede metode za iscrtavanje blokova na zaslon. Prva metoda – `getPositionedSquareContainer` vratit će grafički prikaz jednog atoma na odgovarajućoj poziciji. Metoda `drawBlocks` iscrtava sve blokove na igraćoj ploči, atom po atom.

```
Widget getPositionedSquareContainer(Color color, int x, y) {
  return Positioned(
    left: x * tileSize + 1,
    top: y * tileSize + 1,
    child: Container(
      width: tileSize - 2,
      height: tileSize - 2,
      decoration: BoxDecoration(
        borderRadius: BorderRadius.circular(2),
        color: color,
      ),
    ),
  );
}
```

```
Widget drawBlocks() {
  if (block == null) return null;
  List<Positioned> atoms = [];

  // Trenutni blok
  for (Atom atom in block.atoms) {
    atoms.add(getPositionedSquareContainer(
        atom.color, atom.x + block.x, atom.y + block.y));
  }

  // Atomi iz liste oldAtoms
  if (oldAtoms != null) {
    for (Atom element in oldAtoms) {
      atoms.add(
        getPositionedSquareContainer(element.color, element.x, element.y),
      );
    }
  }
  return Stack(children: atoms);
}
```

Sljedeća je metoda za dohvaćanje novog nasumičnog bloka:

```
Block getNewBlock() {
  int blockType = Random().nextInt(7);
  int orientationIndex = Random().nextInt(4);

  if (blockType == 0) return IBlock(orientationIndex);
  if (blockType == 1) return JBlock(orientationIndex);
  if (blockType == 2) return LBlock(orientationIndex);
  if (blockType == 3) return OBlock(orientationIndex);
  if (blockType == 4) return TBlock(orientationIndex);
  if (blockType == 5) return SBlock(orientationIndex);
  if (blockType == 6) return ZBlock(orientationIndex);
```

```
  return null;
}
```

Posljednja metoda koju ćemo implementirati je `onPlay`, koja izvršava jedan ciklus igre. Prvo ćemo postaviti inicijalni status kolizije na ništa (`Collision.none`). Nakon toga u `setState` bloku provjeravamo ako je radnja (`action`) različita od `null` i nismo na rubu igrače ploče, pomičemo blok u određenu stranu. Ako dođe do sudara blokova pomičemo ih u suprotnu stranu. Nakon toga provjeravamo kolizije s donje i gornje strane bloka i postavljamo varijablu `status`, odnosno pomičemo blok. Sljedeće detektiramo jesmo li udarili u blok koji je iznad a došao je s vrha igrače ploče, tada znamo da je igra završila. Ako nije, ažuriramo bodove.

```
void onPlay(Timer timer) {
  Collision status = Collision.none;

  setState(() {
    if (action != null && !checkOnEdge(action)) {
      block.move(action);
    }

    // Suprotna radnja u slučaju kolizije
    for (Atom oldAtom in oldAtoms) {
      for (Atom atom in block.atoms) {
        final x = block.x + atom.x;
        final y = block.y + atom.y;
        if (x == oldAtom.x && y == oldAtom.y) {
          if (action == BlockMove.left) block.move(BlockMove.right);
          if (action == BlockMove.right) block.move(BlockMove.left);
          if (action == BlockMove.rot_right)
             block.move(BlockMove.rot_left);
          if (action == BlockMove.rot_left)
             block.move(BlockMove.rot_right);
        }
      }
    }
```

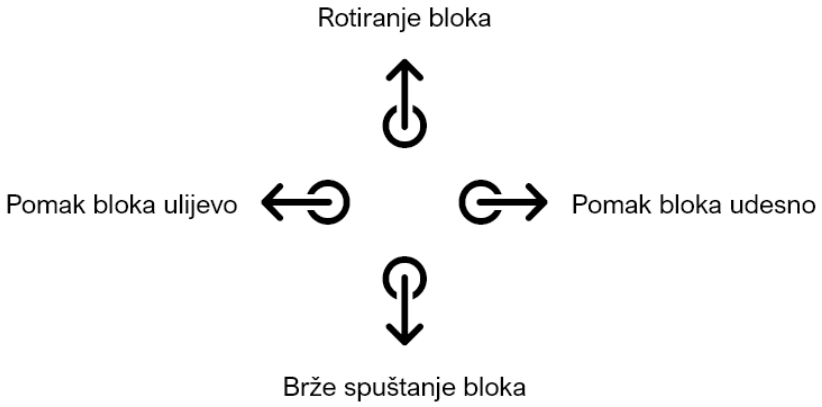
```
    if (!checkAtBottom()) {
      if (!checkAboveBlock()) {
        block.move(BlockMove.down);
      } else {
        status = Collision.block;
      }
    } else {
      status = Collision.ground;
    }

    if (status == Collision.block && block.y < 0) {
      isGameOver = true;
      endGame();
    } else if (status == Collision.ground || status == Collision.block) {

      for (Atom atom in block.atoms) {
        atom.x += block.x;
        atom.y += block.y;
        oldAtoms.add(atom);
      }

      block = getNewBlock();
    }

    action = null;
    updateScore();
  });
}
```

6.2.4. Korisničko sučelje i upravljanje igrom

Igra nudi nekoliko načina upravljanja:

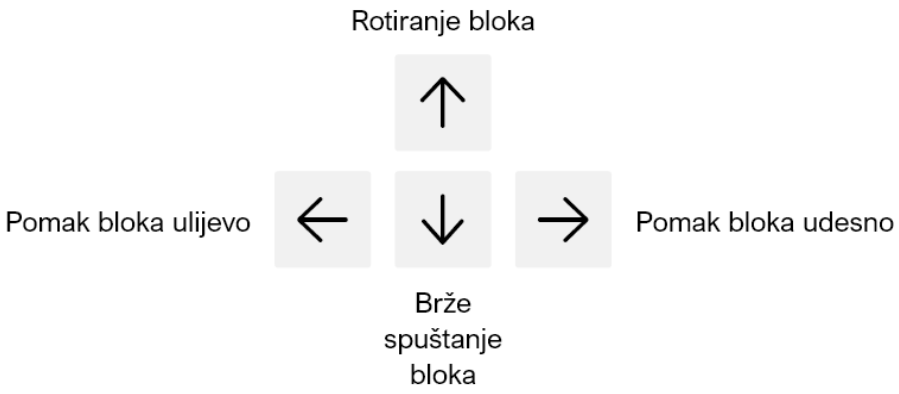
- Geste na dodirnom zaslonu
- Tipkovnica
- Zaslonske kontrole

Detekcija gesti riješena je korištenjem paketa `SimpleGestureDetector`, koji nudi metode `onHorizontalSwipe` i `onVerticalSwipe`. Unutar svake možemo pročitati smjer geste (gore/dolje, odnosno lijevo/desno) i sukladno tome pokrenuti potrebnu akciju. U nastavku je vidljiva shema kontrola za upravljanje gestama.



Slika 36: Shema kontrola za upravljanje gestama (vlastita izrada)

Upravljanje tipkovnicom implementirano je preko *Flutter*ovog `RawKeyboardListenera`, koji daje mogućnost slušanja metode koja ima informacije o pritisnutom gumbu. Kontrola shema upravljanja tipkovnicom prikazana je na slici ispod:



Slika 37: Shema kontrola za upravljanje tipkovnicom (vlastita izrada)

Upravljanje zaslonskim kontrolama napravljeno je u trenutku izrade korisničkog sučelja. Ispod ploče za igru postoje gumbi za pomicanje blokova, rotiranje i početak/pauzu/nastavak igre.

Korisničko sučelje izrađeno je kombinacijom nekoliko *widgeta* postavljenih u stupac:

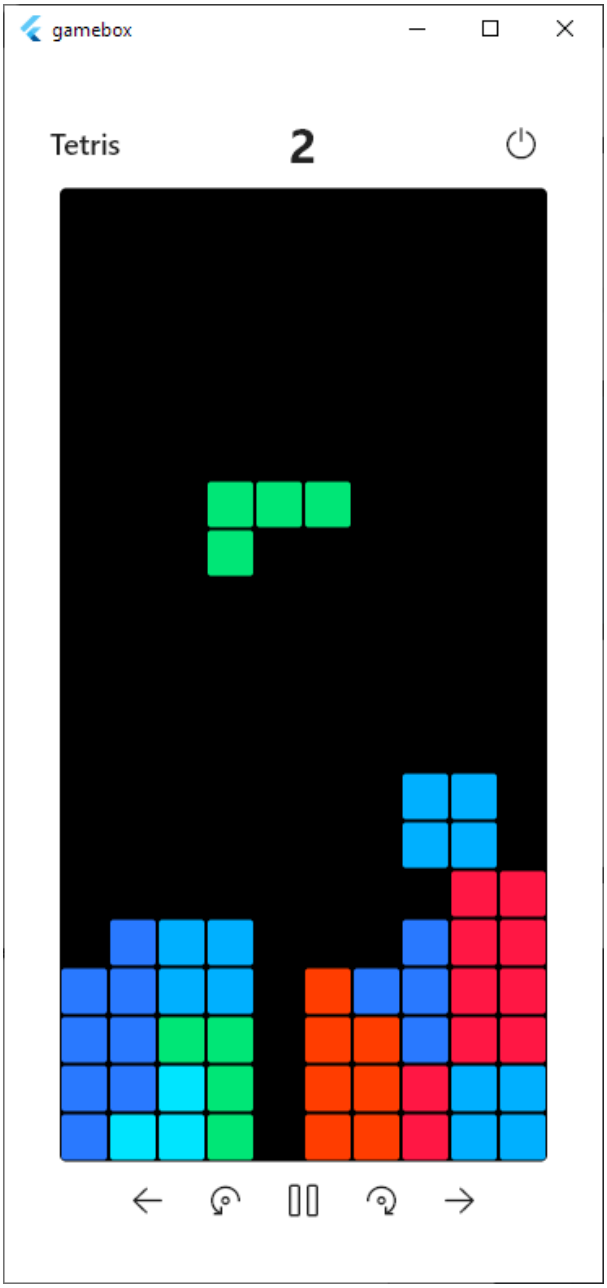
```
@override
Widget build(BuildContext context) {
return Theme(
  data: ThemeData.light(),
  child: RawKeyboardListener(
    focusNode: f,
    onKey: (RawKeyEvent r) {
      if (r.logicalKey == LogicalKeyboardKey.arrowLeft) {
        setState(() => action = BlockMove.left);
      }
      if (r.logicalKey == LogicalKeyboardKey.arrowRight) {
        setState(() => action = BlockMove.right);
      }
      if (r.logicalKey == LogicalKeyboardKey.arrowUp) {
        setState(() => action = BlockMove.rot_right);
      }
      if (r.logicalKey == LogicalKeyboardKey.arrowDown) {
        setState(() => action = BlockMove.down);
      }
    },
    autofocus: true,
    child: Scaffold(
      backgroundColor: Colors.white,
      body: SimpleGestureDetector(
        onHorizontalSwipe: (direction) {
          if (direction == SwipeDirection.left) {
            setState(() => action = BlockMove.left);
          }
```

```
  if (direction == SwipeDirection.right) {
    setState(() => action = BlockMove.right);
  }
},
onVerticalSwipe: (direction) {
  if (direction == SwipeDirection.up) {
    setState(() => action = BlockMove.rot_right);
  }
  if (direction == SwipeDirection.down) {
    setState(() => action = BlockMove.down);
  }
},
child: Column(
  mainAxisAlignment: MainAxisAlignment.center,
  children: [
    Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: [
        Text('Tetris'),
        Text('${score ?? 0}'),
        FlatButton(
          onPressed: () => Navigator.of(context).pushReplacement(
            MaterialPageRoute(
              builder: (context) => HomeScreen(),
            ),
          ),
          child: Text('Izlaz'),
        ),
      ],
    ),

    Container(
```

```
  key: boardKey,
  decoration: BoxDecoration(
    borderRadius: BorderRadius.circular(4),
    color: Colors.black,
  ),
  width: 300,
  height: 600,
  child: drawBlocks(),
),
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: [
    IconButton(
      icon:
          Icon(FluentSystemIcons.ic_fluent_arrow_left_regular),
      onPressed: !didStart
          ? null
          : () => setState(() => action = BlockMove.left),
    ),
    IconButton(
      icon: Icon(
        FluentSystemIcons
            .ic_fluent_arrow_rotate_counterclockwise_regular,
      ),
      onPressed: !didStart
          ? null
          : () => setState(() => action = BlockMove.rot_left),
    ),
    if (!didStart)
      IconButton(
        icon: Icon(
          FluentSystemIcons.ic_fluent_play_regular,
          color: Colors.blue.shade700,
```

```
            ),
            onPressed: () => startGame(),
          )
        else
          IconButton(
            icon: Icon(
              isPlaying ?? true
                  ? FluentSystemIcons.ic_fluent_pause_regular
                  : FluentSystemIcons.ic_fluent_play_regular,
            ),
            onPressed: () => isPlaying ?? true
                ? setState(() => isPlaying = false)
                : setState(() => isPlaying = true),
          ),
        IconButton(
          icon: Icon(FluentSystemIcons
              .ic_fluent_arrow_rotate_clockwise_regular),
          onPressed: !didStart
              ? null
              : () => setState(() => action = BlockMove.rot_right),
        ),
        IconButton(
          icon:
            Icon(FluentSystemIcons.ic_fluent_arrow_right_regular),
          onPressed: !didStart
              ? null
              : () => setState(() => action = BlockMove.right),
        ),
      ],
    ),
  ],
),),),),);
```

Korisničko sučelje sa zaslonskim kontrolama vidljivo je na slici ispod:



Slika 38: Korisničko sučelje igre Tetris (vlastita izrada)

6.3. Minesweeper

Kao i Tetris, i ova je igra napisana bez korištenja *Flame* biblioteke.

Opis implementacije ići će kroz sljedeće korake:

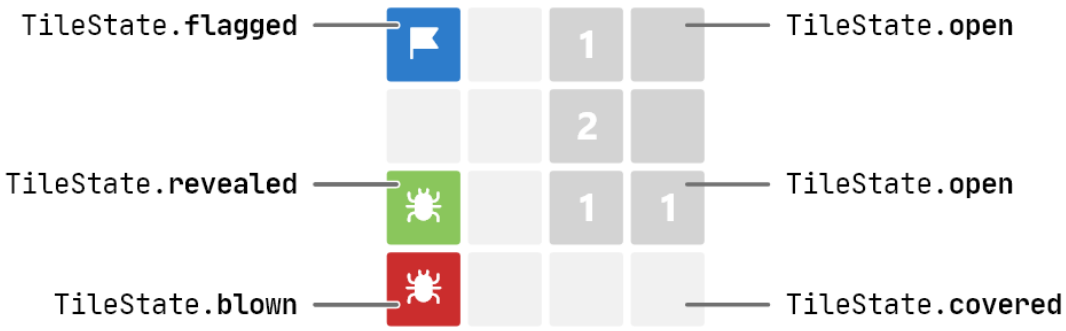
- Izrada podatkovnog modela za spremanje podataka ploče

- Implementacija glavnih funkcionalnosti igre
    - Postavljanje ploče
    - Otkrivanje polja
- Implementacija korisničkog sučelja
- Dodatne funkcionalnosti

6.3.1. Izrada podatkovnog modela za rad s podatcima

Budući da se ova igra oslanja na polje, mrežu dijelova, logičan izbor je odabrati dvodimenzionalno polje kao bazu zapisa podataka. Radit ćemo s dvije liste. Prva lista (`mines`) sadrži informacije o lokaciji mina na ploči za igru. Druga lista (`uiState`) sadrži informacije o stanju svakog polja na ploči za igru. Postoji pet mogućih stanja polja:

- `covered` – zadano stanje
- `blown` – kada igrač klikne na minu i izgubi igru
- `open` – otvoreno polje (s brojevima ili bez)
- `flagged` – polje sa zastavicom (gdje igrač sumnja da je mina)
- `revealed` – uspješno otkrivena mina (prikazuje se nakon pobjede razine)



Slika 39: Moguća stanja polja (vlastita izrada)

6.3.2. Implementacija igre

Kao i kod Tetrisa koristit ćemo `StatefulWidget` jer trebamo imati *stanje* koje se može mijenjati u toku igre. Uz gore navedene liste od varijabli će nam trebati:

- `rows` – broj redaka na ploči za igru
- `cols` – broj stupaca na ploči za igru

- `numOfMines` – broj mina na ploči za igru
- `alive` – je li igrač živ
- `wonGame` – zastavica za pobjedu igre
- `minesFound` – broj trenutno pronađenih mina
- `timer` i `stopwatch` – za praćenje proteklog vremena

Prva metoda koju ćemo implementirati je `resetBoard`. Ona služi za postavljanje, odnosno ponovno postavljanje ploče za igru. Prvo postavljamo gore navedene varijable na inicijalne vrijednosti, resetiramo tajmer i štopericu te generiramo mine na nasumičnim mjestima i zapišemo ih u listu `mines`. To u kôdu izgleda ovako:

```
void resetBoard() {
  alive = true;
  wonGame = false;
  minesFound = 0;
  stopwatch.stop();
  stopwatch.reset();

  timer?.cancel();
  timer = Timer.periodic(Duration(seconds: 1), (_) => setState(() {}));

  uiState = List<List<TileState>>.generate(
    rows,
    (_) => List<TileState>.filled(cols, TileState.covered),
  );

  mines = List<List<bool>>.generate(
    rows,
    (_) => List<bool>.filled(cols, false),
  );

  Random random = Random();
  int remainingMines = numOfMines;
```

```
  while (remainingMines > 0) {
    int pos = random.nextInt(rows * cols);

    int row = pos ~/ rows;
    int col = pos % cols;

    if (!mines[row][col]) {
      mines[row][col] = true;
      remainingMines--;
    }
  }
}
```

Sljedeća na redu je metoda za generiranje *widgeta* ploče za igru s pripadnim funkcionalnostima. Oni će nam služiti za iscrtavanje ploče poslije. Na početku postavljamo zastavicu `hasCoveredCell` na `false`. Ona će nam poslije reći postoji li polje stanja *pokriveno*.

Nakon toga inicijaliziramo polje u koje ćemo spremati redove ploče. Nakon toga kroz dvije ugniježđene `for` petlje prolazimo kroz ploču za igru i ovisno o polju odabiremo pravilno polje za iscrtati i u određenim slučajevima dodajemo funkcionalnost na dodir/klik ili *long press*/klik i držanje. U kôdu je to implementirano ovako:

```
Widget buildBoard() {
  bool hasCoveredCell = false;

  List<Row> boardRow = <Row>[];
  for (int y = 0; y < rows; y++) {
    List<Widget> rowChildren = <Widget>[];

    for (int x = 0; x < cols; x++) {
      TileState state = uiState[y][x];

      int count = mineCount(x, y);

      if (!alive) {
        if (state != TileState.blown) {
```

```
    state = mines[y][x] ? TileState.revealed : state;
  }
}

if (state == TileState.covered || state == TileState.flagged) {
  rowChildren.add(GestureDetector(
    onLongPress: () {
      HapticFeedback.heavyImpact();
      flag(x, y);
    },
    onTap: () {
      HapticFeedback.selectionClick();
      if (state == TileState.covered) {
        probe(x, y);
      }
    },
    child: CoveredMineTile(
      flagged: state == TileState.flagged,
      revealed: state == TileState.flagged && wonGame,
      posX: x,
      posY: y,
    ),
  ));
  if (state == TileState.covered) {
    hasCoveredCell = true;
  }
} else {
  rowChildren.add(
    OpenMineTile(
      state: state,
      count: count,
    ),
  );
```

```
      }
    }

    boardRow.add(
      Row(
        children: rowChildren,
        mainAxisAlignment: MainAxisAlignment.center,
        key: ValueKey<int>(y),
      ),
    );
  }
  if (!hasCoveredCell) {
    if (minesFound == numOfMines && alive) {
      wonGame = true;
      stopwatch.stop();
    }
  }

  return Column(children: boardRow);
}
```

Sljedeće što je potrebno implementirati su pomoćne funkcije:

- `onBoard` – provjerava je li polje unutar ploče za igru
- `bombs` – provjerava postoji li `true` vrijednost u listi `mines` za određenu koordinatu
- `mineCount` – broj mina na ploči za igru
- `probe` – nakon klika na polje bez zastavice
- `open` – otvaranje praznog, *netaknutnog* polja
- `flag` – postavljanje zastavice na polje

Implementacije ovih funkcija prikazane su u nastavku:

```
bool onBoard(int x, y) => x >= 0 && x < cols && y >= 0 && y < rows;

int bombs(int x, y) => onBoard(x, y) && mines[y][x] ? 1 : 0;
```

```
int mineCount(int x, y) {
  int count = 0;

  count += bombs(x - 1, y);
  count += bombs(x + 1, y);
  count += bombs(x, y - 1);
  count += bombs(x, y + 1);
  count += bombs(x - 1, y - 1);
  count += bombs(x + 1, y + 1);
  count += bombs(x + 1, y - 1);
  count += bombs(x - 1, y + 1);

  return count;
}

void probe(int x, y) {
  if (!alive) return;

  if (uiState[y][x] == TileState.flagged) return;

  setState(() {
    if (mines[y][x]) {
      uiState[y][x] = TileState.blown;
      alive = false;
      timer.cancel();
    } else {
      open(x, y);
      if (!stopwatch.isRunning) stopwatch.start();
    }
  });
}
```

```
void open(int x, y) {
  if (!inBoard(x, y)) return;

  if (mineCount(x, y) == 0 && uiState[y][x] == TileState.flagged)
    --minesFound;

  if (uiState[y][x] == TileState.open) return;
  uiState[y][x] = TileState.open;

  if (mineCount(x, y) > 0) return;

  open(x - 1, y);
  open(x + 1, y);
  open(x, y - 1);
  open(x, y + 1);
  open(x - 1, y - 1);
  open(x + 1, y + 1);
  open(x + 1, y - 1);
  open(x - 1, y + 1);
}

void flag(int x, y) {
  if (!alive) return;

  setState(() {
    if (uiState[y][x] == TileState.flagged) {
      uiState[y][x] = TileState.covered;
      --minesFound;
    } else {
      uiState[y][x] = TileState.flagged;
      ++minesFound;
    }
  });
```

```
}
}
```

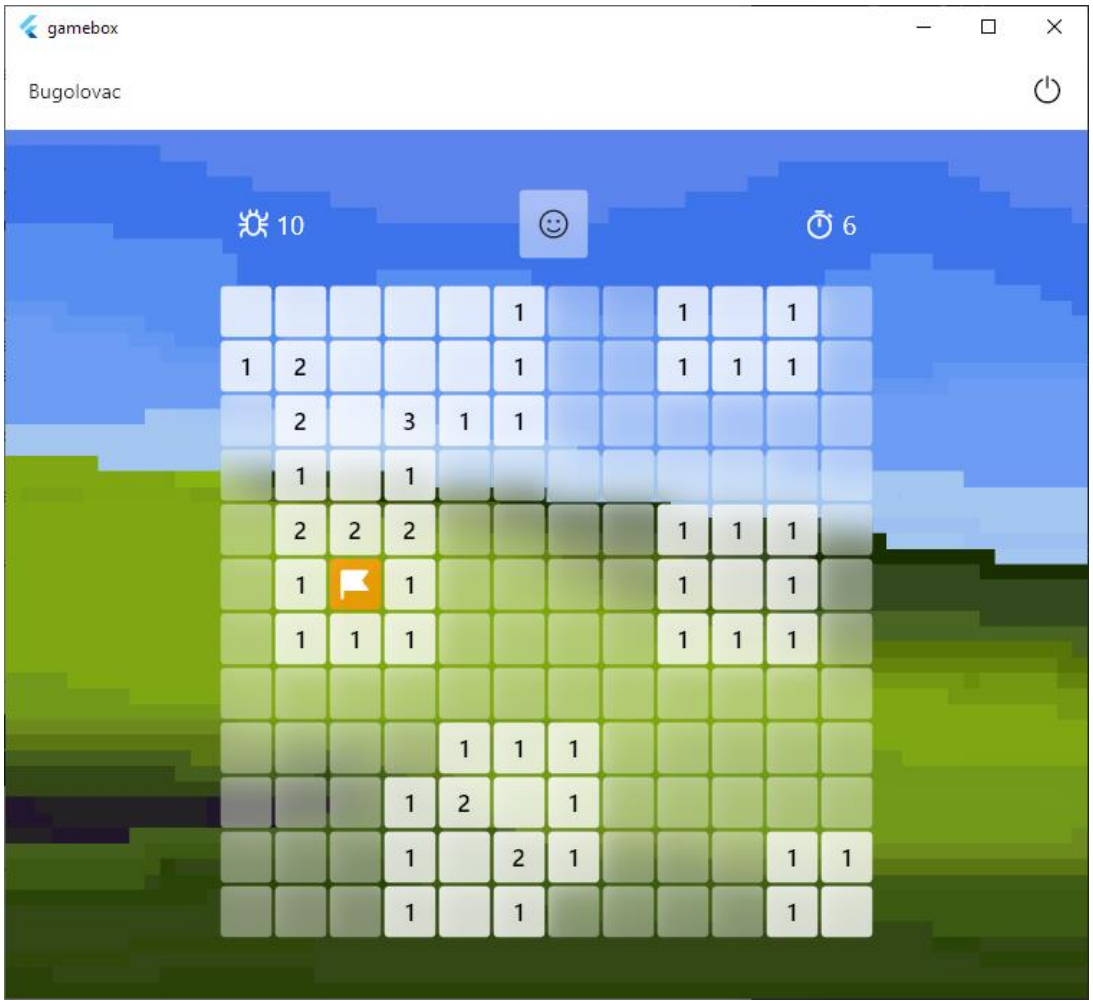
Sve što preostaje je napraviti korisničko sučelje i naša igra je u potpunosti implementirana. Sučelje se sastoji od pozadine s pozadinskom slikom iznad koje je *stupac* (`Column`) u kojem su informacije o igri (broj mina, proteklo vrijeme u sekunda) te gumb za restiranje igre). Ispod toga je ploča za igru. To u kôdu izgleda ovako:

```
@override
Widget build(BuildContext context) {
  int timeElapsed = stopwatch.elapsedMilliseconds ~/ 1000;
  return Scaffold(
    resizeToAvoidBottomInset: false,
    appBar: AppBar(
      textTheme: ThemeData.light().textTheme,
      backgroundColor: Colors.white,
      elevation: 0,
      title: Text('Bugolovac'),
      actions: [
        IconButton(
          onPressed: () => Navigator.of(context).pushReplacement(
            MaterialPageRoute(
              builder: (context) => HomeScreen(),
            ),
          ),
          icon: Icon(
            FluentSystemIcons.ic_fluent_power_regular,
            color: Colors.black,
          ),
        ),
        SizedBox(width: 8),
      ],
    ),
    body: Container(
      decoration: BoxDecoration(
```

```
  image: DecorationImage(
    fit: BoxFit.cover,
    image: NetworkImage(
        '( url )'),
  ),
),
child: Column(
  mainAxisAlignment: MainAxisAlignment.center,
  children: [
    SizedBox(
      width: tileSize * cols,
      child: Row(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        children: [
          Row(
            children: [
              Icon(FluentSystemIcons.ic_fluent_bug_report_regular),
              SizedBox(width: 4),
              Text(
                '$numOfMines',
                style: TextStyle(fontSize: 18),
              )
            ],
          ),
          ControllerButton(
            bgColor: wonGame
                ? Colors.lightGreen[200]
                : alive ? Colors.white : Colors.red[200],
            icon: alive
                ? (wonGame
                    ? FluentSystemIcons.ic_fluent_trophy_regular
                    : FluentSystemIcons.ic_fluent_emoji_regular)
                : FluentSystemIcons.ic_fluent_emoji_sad_regular,
```

```
                onTap: () => resetBoard(),
              ),
              Row(
                children: [
                  Icon(FluentSystemIcons.ic_fluent_timer_regular),
                  SizedBox(width: 4),
                  Text(
                    '$timeElapsed',
                    style: TextStyle(fontSize: 18),
                  )
                ],
              ),
            ],
          ),
        ),
        SizedBox(height: tileSize / 2),
        buildBoard(),
      ],
    ),
  ),
 );
}
```

Sučelje igre prikazano je na slici ispod:



Slika 40: Korisničko sučelje igre Minewseeper (vlastita izrada)

7. Zaključak

Želja za istraživanjem novog razvojnog okvira uz mogućnost usputne izrade igara bio je glavni poticaj za odabir ove teme diplomskog rada. Prije odabira teme pratio sam razvoj *Fluttera* i malo ga koristio, ali ovo mi je bila prilika napraviti nešto *ozbiljnije* i isprobati nove značajke istog. Proces izrade je bio zahtjevan u trenutcima, tu je već dosta velika *Flutter* zajednica iznimno pomogla.

Biblioteka *Flame* pokazala se odličnom za igre poput *Dangerous Dave*a. U proces izrade igre unosi onoliko puno ili malo kompleksnosti mi želimo, s mogućnosti dodavanja naprednijih značajki. Kada se to kombinira s *Flutter*ovim *stateful hot reload*om (mogućnosti ažuriranja sučelja i kôda aplikacije bez ponovnog pokretanja iste) dobivamo platformu koja omogućava relativno brz razvoj igara.

Naravno, nije sve savršeno. Na primjer, kompletnija implementacija za učitavanem *Tiled* razina bila bi odlična, ali prilično sam zadovoljan s vlastitim sustavom adaptivnog učitavanja razina. Također, dio upustava za *Flame* još nije napisan, pa sam za neke stvari morao listati desetke stranica *Google* rezultata. Najbitnije je da sam na kraju uspio naći rješenje.

Dart se pokazao kao odličan programski jezik. Uzima dobre elemente iz C#-a, JavaScripta, Jave i sličnih jezika i kombinira ih u jezik u kojem se može uživati dok se piše kôd. Kada se to kombinira s odličnim proširenjima za *Visual Studio Code* i *IntelliJ IDEA*-u, dobiva se odlična platforma za razvoj aplikacija na svim platformama.

Mogućnost izvršavanja istog kôda na više platformi činila se kao nemoguća, ali vidjevši to svojim očima bio sam jako sretan da bez ikakvih prepravki sve radi kako treba. Naravno, *web* podrška je trenutno u *beti*, a *desktop* podrška u *alpha-i*, pa tu i tamo nešto zašteka na par trenutaka, to će se s vremenom sve ispleglati. Jedini „veći“ *bug* koji sam pronašao je nemogućnost korištenje *Google Fonts* proširenja u *desktop* aplikaciji zbog paketa za učitavanje datoteka koji nedostaje.

Sve u svemu, zadovoljan sam izborom teme, platforme i napravljenim rezultatom, preporučio bih svima da probaju *Flutter*, ako ne za igre, za aplikacije općenito.

Popis literature

[1] Google, »Flutter,« Google, 2020. [Mrežno]. Available: https://flutter.dev/. [Pokušaj pristupa 15 8 2020].

[2] Google, »Flutter logo,« [Mrežno]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/1/17/Google-flutter-logo.png/320px-Google-flutter-logo.png.

[3] Google, »Flutter Showcase,« 2020. [Mrežno]. Available: https://flutter.dev/showcase. [Pokušaj pristupa 19 8 2020].

[4] Google, »Dart,« 2020. [Mrežno]. Available: https://dart.dev/. [Pokušaj pristupa 20 8 2020].

[5] Google, »Dart logo,« 2020. [Mrežno]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/f/fe/Dart_programming_language_logo.svg/320px-Dart_programming_language_logo.svg.png. [Pokušaj pristupa 20 8 2020].

[6] Google, »Dart platforms,« 2020. [Mrežno]. Available: https://dart.dev/platforms. [Pokušaj pristupa 20 8 2020].

[7] Flame, »About Flame,« Flame, 2020. [Mrežno]. Available: https://flame-engine.org/docs/. [Pokušaj pristupa 22 8 2020].

[8] Flame, »Components,« Flame, 2019. [Mrežno]. Available: https://flame-engine.org/docs/components.md. [Pokušaj pristupa 21 8 2020].

[9] Tiled, »About Tiled,« 2019. [Mrežno]. Available: https://doc.mapeditor.org/en/stable/manual/introduction/#about-tiled. [Pokušaj pristupa 21 8 2020].

[10] Tiled, »Tiled logo,« 2019. [Mrežno]. Available: https://www.mapeditor.org/img/tiled-logo-white.png. [Pokušaj pristupa 21 8 2020].

[11] Tiled, »Working with layers,« 2019. [Mrežno]. Available: https://doc.mapeditor.org/en/stable/manual/layers/#tile-layer-introduction. [Pokušaj pristupa 21 8 2020].

[12] Tiled, »Editing tile layers,« 2019. [Mrežno]. Available: https://doc.mapeditor.org/en/stable/manual/editing-tile-layers/. [Pokušaj pristupa 21 8 2020].

[13] Tiled, »Editing tilesets,« 2019. [Mrežno]. Available: https://doc.mapeditor.org/en/stable/manual/editing-tilesets/. [Pokušaj pristupa 21 08 2020].

[14] J. Romero, »Dangerous Dave player spritesheet,« 12 8 2015. [Mrežno]. Available: http://smwstuff.net/skin=1020. [Pokušaj pristupa 4 8 2020].

[15] iD software, »Doom I slika,« [Mrežno]. Available: https://www.polygon.com/2019/7/26/8932238/doom-classic-2-3-bethesda-net-account-required-meme. [Pokušaj pristupa 22 8 2020].

[16] SoftDisk publishing, »Dangerous Dave slika,« [Mrežno]. Available: https://www.playdosgames.com/online/dangerous-dave/. [Pokušaj pristupa 22 8 2020].

[17] Spectrum Holobyte, »Tetris slika,« [Mrežno]. Available: https://www.abandonwaredos.com/abandonware-game.php?abandonware=Tetris&gid=1076. [Pokušaj pristupa 22 8 2020].

[18] V. Matthews, »The Many Different Types of Video Games & Their Subgenres,« 12 4 2018. [Mrežno]. Available: https://www.idtech.com/blog/different-types-of-video-game-genres. [Pokušaj pristupa 22 8 2020].

[19] Više autora, »Platform Game,« 20 8 2020. [Mrežno]. Available: https://tvtropes.org/pmwiki/pmwiki.php/Main/PlatformGame. [Pokušaj pristupa 22 8 2020].

[20] Više autora, »Video Game Genres,« 2020. [Mrežno]. Available: https://tvtropes.org/pmwiki/pmwiki.php/Main/VideoGameGenres. [Pokušaj pristupa 22 8 2020].

[21] P. Hanna, »Video Game Technologies,« 2016. [Mrežno]. Available: https://www.di.ubi.pt/~agomes/tjv/teoricas/01-genres.pdf. [Pokušaj pristupa 22 8 2020].

[22] Više autora, »Puzzle Game,« 6 4 2020. [Mrežno]. Available: https://tvtropes.org/pmwiki/pmwiki.php/Main/PuzzleGame. [Pokušaj pristupa 22 8 2020].

[23] A. Weigart, »Need a Game Idea? A List of Game Mechanics and a Random Mechanic Mixer.,« 30 7 2012. [Mrežno]. Available: https://inventwithpython.com/blog/2012/07/30/need-a-game-idea-a-list-of-game-mechanics-and-a-random-mechanic-mixer/. [Pokušaj pristupa 22 8 2020].

[24] Tetris wiki, »Tetromino,« [Mrežno]. Available: https://tetris.fandom.com/wiki/Tetromino. [Pokušaj pristupa 22 8 2020].

[25] Google, »Flutter FAQ,« Google, 2020. [Mrežno]. Available: https://flutter.dev/docs/resources/faq. [Pokušaj pristupa 19 8 2020].

[26] Tiled, »Export formats,« Tiled, 2019. [Mrežno]. Available: https://doc.mapeditor.org/en/stable/manual/export/. [Pokušaj pristupa 21 8 2020].

Popis slika