2D IGRA RAZVIJENA U FLUTTER OKRUŽENJU

Šabić, Antonio

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijske tehnologije (Informacijska tehnologija)

ANTONIO ŠABIĆ

Z A V R Š N I R A D

2D IGRA RAZVIJENA U FLUTTER OKRUŽENJU

Split, rujan 2023.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijske tehnologije (Informacijska tehnologija)

Predmet: Objektno programiranje

Z A V R Š N I R A D

Kandidat: Antonio Šabić

Naslov rada: 2D Igra razvijena u Flutter okruženju

Mentor: Ljiljana Despalatović, viši predavač

Matični broj: 0177045517

Split, rujan 2023.

Sadržaj

Sažetak **4**

1. Uvod **5**

2. Opis i pravila igre **6**

3. Arhitektura aplikacije **9**

4. Implementacija komponenti **12**

4.1. Koordinator 12

4.2. Komunikacija među igračima 13

4.3. Flame Game 15

4.4. Zvuk 16

4.5. Komponente 18

4.5.1. PositionComponent 19

4.5.2. FlameGame 20

4.5.3. Movable 20

4.5.4. Character 22

4.5.5. Shooter 23

4.5.6. Komponente kolizije 24

4.5.7. SpriteComponent 27

4.5.8. PartsManager 27

4.5.9. Događaji pritiska tipki i dodira 28

4.5.10. Brojač vremena 30

4.5.11. Postavke 31

5. Zaključak 33

6. Literatura 34

Sažetak

Ovaj završni rad obuhvaća opis izrade 2D igre napisane u Flutter okruženju. U radu su definirana pravila igre, arhitektura izrađene aplikacije, pojedine cjeline koje čine logiku aplikacije i ostale postavke projekta. Također je opisana i svaka korištena komponenta Flame biblioteke na kojoj se zasniva aplikacija. Kroz rad se projekt opisuje od općeg vida aplikacije prema konkretnim elementima koji je čine. To je napravljeno u svrhu da čitatelju bude što jasnije kako je izvedena i da se prikaže što jasnije svrha samog razvoja te aplikacije.

Ključne riječi: Bloc, Flame, Flutter.

Summary

2D Flutter game

This final paper is written to describe the 2D game built in the Flutter framework. Things that are being discussed in this paper are rules of the game, application architecture, various parts that make the whole logic of the game and the rest of the project settings. Also every used component of the Flame library is being described on which the application is based on. Throughout the final paper we describe everything from an abstract point of view to a more concrete one. The reason for that is to bring the reader closer to the point of this project and to understand parts of this project as best as possible.

Keywords: Bloc, Flame, Flutter.

1. Uvod

Igra Space Arena je 2D igra za dvoje igrača koji se nalaze na istoj lokalnoj mreži, kojima je cilj u igri uništiti neprijatelja u svemiru u određenom vremenskom periodu. Igra je napisana u jeziku Dart koristeći Flutter okruženje i biblioteku Flame koja je popularna za razvoj igara u Flutter okruženju. Flame okruženje olakšava razvojnim programerima da što lakše riješe problematiku igara, odnosno ima ugrađene osnovne mehanizme koji pomažu u konstrukciji arhitekture igre kao što su petlja igre (engl. *game loop*), lako ugrađivanje izgleda i kretnji likova (*sprites*) i sl., dok je Flutter zadužen za izgradnju korisničkog sučelja i rješavanje sistemskih problema. Igra podržava Windows, Android, iOS, macOS i Linux platformu što korisniku olakšava pristup igranju uz manje prilagodbe za svaku od platformi koje ću detaljnije opisati kasnije.

Flutter okruženje je odabrano zbog jednostavnosti korištenja i pouzdanosti u razvoju aplikacije. Flutter okruženje je u svijetu razvoja aplikacija od 2017. godine što ga čini prilično mladim okruženjem, ali nipošto se ne smatra lošim odabirom za razvoj aplikacija jer ima veliku podršku od razvojnih programera treće strane koji rade izvrsne biblioteke dostupne za korištenje svima. Izvrsnost i prednost Flutter okruženja je u tome što omogućava stvaranje biblioteka koje apstrahiraju ponašanje koje je implementirano zasebno na svakoj od platformi te razvojni programeri koji koriste takve biblioteke ne znaju (barem ne trebaju znati) kako su točno takva rješenja implementirana na svakoj od platformi zasebno. Flame je samo jedna od takvih biblioteka koja je korištena u ovom projektu koju treba izdvojiti jer postaje sve popularnija i kvalitetnija s vremenom.

2. Opis i pravila igre

Radnja igre je smještena u svemiru u nepoznatom vremenu bez definiranog razloga. Igraju dva stvarna igrača u stvarnom vremenu jedan protiv drugog (engl. *PvP*) te je cilj svakome od igrača uništiti neprijatelja (njegov matični brod). Svaki od igrača ima na raspolaganju dva broda, jedan pokretniji borbeni avion te matični brod koji se može modularno nadograđivati za razne realizacije taktika u igri. Kada se borbeni avion uništi, nakon nekog određenog vremena brod se opet stvara na svojoj početnoj poziciji.

Matični brod se nadograđuje s tri različita dijela: oružjem, štitom te pogonom. Svaki od dijela ima svoju cijenu te se može ugraditi na matični brod ili na neki već ugrađeni dio samo ako je zadovoljen uvjet da igrač već ima određen broj resursa potrebnih za izgradnju istoga. Svaki od dijelova zauzima jednak kvadratni prostor kao i ostale jedinice/brodovi. Igrač stječe resurse pucanjem u resursne stijene koje se periodično stvaraju nasumično u prostoru igre i koje imaju određenu količinu resursa koje daju.

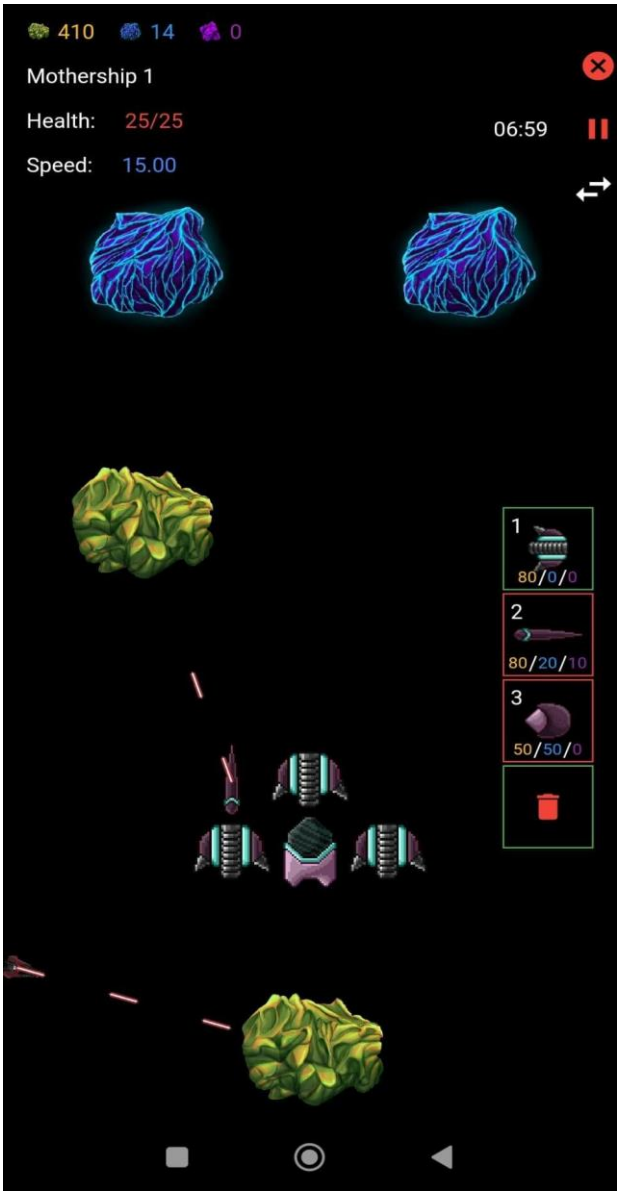
Postoje tri vrste resursa: zlato, kristal i plazma. Zlato je najviše dostupno i služi u određenim mjerama za izgradnju svih dijelova. Plazma je rjeđi resurs i potreban je za izgradnju oružja i pogona u manjoj mjeri, iako se i dobiva u manjoj mjeri od zlata pucajući u resursnu stijenu plazme. Kristal je potreban samo za izgradnju oružja, što čini kristal najtraženijim resursom uz činjenicu da se u prostoru igre može nalaziti istovremeno samo jedna resursna stijena kristala, odnosno, ne stvara se nova stijena kristala ako nije uništena do kraja zadnja stvorena. Stijena kristala ima puno manje resursa od ostala dva resursa.

Na početku igre, igrači su smješteni u prostoru na unaprijed definiran način. Prvi igrač (*host*) je smješten na dnu po sredini prostora sa svoja oba broda u blizini resursne stijene zlata. Drugi igrač je smješten simetrično prvom igraču u odnosu na sredinu (horizontalnu). U sredini su smještene dvije resursne stijene plazme jednako udaljene od oba igrača.

Na sučelju je igraču vidljivo u gornjem desnom kutu preostalo vrijeme igre i znak za pauziranje/pokretanje igre. Na gornjem lijevom dijelu sučelja je igraču vidljivo stanje resursa koje je prikupio te podaci o trenutno odabranom brodu (ime, brzina i šteta boda). S desne strane se nalaze slike dijelova broda obrubljene obojenim kvadratima koji opisuju dostupnost

kupovine pojedinog dijela (zeleno ako je dostupno, crveno ako nije). Također na kraju stupca u kojem se nalaze ti dijelovi stoji oznaka za izbrisati pojedini dio svog matičnog broda.

Odabirom na znak, te slijedno odabirom na dio broda koji se želi uništiti, uništava se dio broda i svi njegovi nasljednici, odnosno svi dijelovi broda koji su sagrađeni na njemu ili na dio broda koji je sagrađen na njemu što se može vidjeti kao ponuđena opcija u igri na slici 1. Isto se događa ako neprijateljski brod uništi neki od dijelova matičnog broda.



Slika 1: Dijelovi matičnog broda

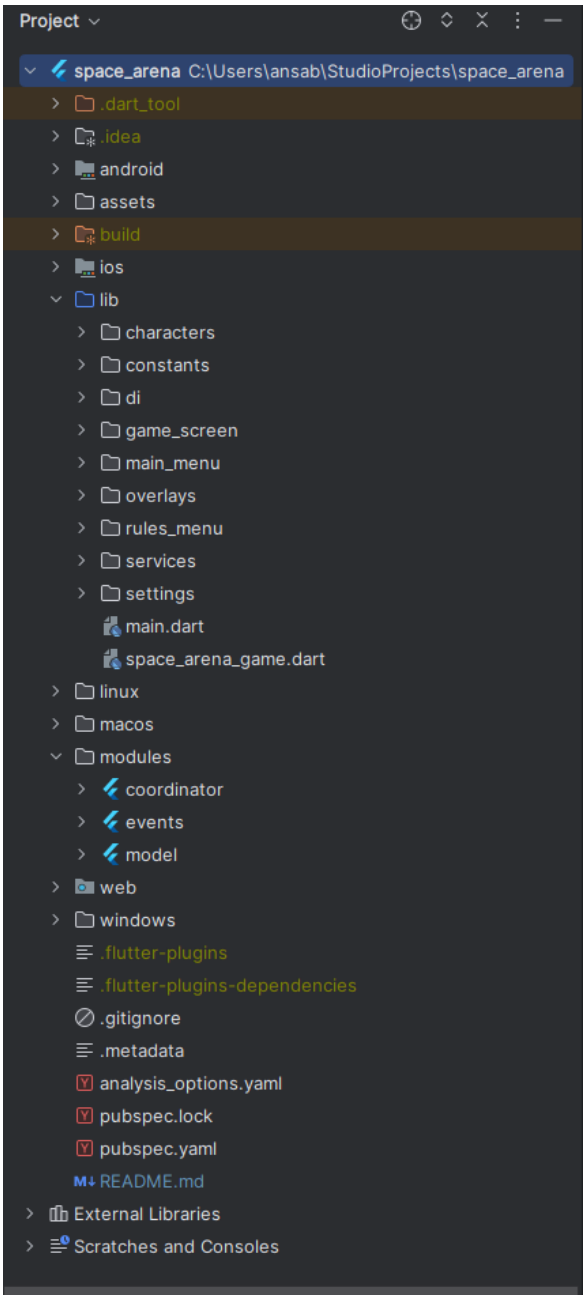
Kao i brodovi, svaki od dijelova matičnog broda ima određen vijek trajanja definiran količinom projektila koje može primiti od neprijateljske strane. Štitovi su najdugotrajniji te se koriste u svrhu zaštite matičnog broda. Pucanje se postiže na način da se jedinicom koja je sposobna ispaljivati projektile dođe u određeni razmak od neprijateljske jedinice te se odvija automatski ispaljivanjem projektila u definiranom vremenskom odmaku. Svaki projektil pogotkom u neprijateljsku jedinicu čini štetu veličine jedne jedinice, stoga npr. ako neka jedinica ima trajnost od 25 jedinica, treba primiti 25 neprijateljskih projektila da bi bila uništena. Nije moguće postići uništavanje vlastitih jedinica projektilima (engl. *friendly fire*) već projektili prolaze kroz prijateljsku jedinicu kao da nije uopće tu.

Jedinice se pokreću desnim klikom na određeno mjesto u prostoru te se time odabrana jedinica usmjerava prema toj točki određenom brzinom. Zakretanje jedinice se automatski izvršava. Izgradnjom dijelova na matični brod raste okretna inercija broda što ga čini manje osjetljivim na okret. Okretnost se smanjuje proporcionalno udaljenosti najudaljenijeg izgrađenog dijela od matičnog broda, stoga je poželjnije graditi dijelove gomilajući ih bliže matičnom brodu.

Svi dijelovi se mogu graditi susjedno već izgrađenom dijelu broda osim pogona koji se mora izgraditi isključivo na donjem dijelu u odnosu na usmjerenost broda. Npr. pogon ne možemo staviti na nos matičnog broda. Igraču je dozvoljeno graditi dijelove isključivo u gornjem, desnom, donjem i lijevom smjeru od odabrane jedinice osim ako to nije dozvoljeno za slučaj izgradnje pogona. Nikako ne može graditi npr. u gornjem - desnom smjeru.

3. Arhitektura aplikacije

Flutter okruženje inicijalno organizira kôd na način da se cijeli projekt nalazi u jednom direktoriju koji obuhvaća razne datoteke i direktorije specifične za postavku projekta. Na slici 2 je prikazan izgled projekta u vidu organizacije kôda.



Slika 2: Organizacija kôda projekta

Ovaj projekt je izrađen pomoću Android Studio alata koji omogućava lakše stvaranje i manipuliranje cijelim projektom. U Android Studio alatu, omogućeno je instaliranje dodataka koji nude bolju podršku za razvijanje Flutter aplikacije. Neki od njih koji su bitni za razvoj su: Flutter, Dart i Bloc dodaci bez kojih bi razvoj aplikacije bio gotovo nemoguć. Dodaci nam nude predloške koji su osnova za rad u navedenom okruženju. Flutter dodatak nam omogućava stvaranje Flutter projekta na način da stvori početne potrebne datoteke koje definiraju Flutter projekt.

Stvaranjem Flutter projekta alat kreira u glavnom (engl. *root)* direktoriju direktorije koji su specifični za pojedinu platformu (Android, iOS, macOS, Windows i Linux) u kojima je izvorni kôd od svake od pojedinih platformi koji je potreban za izgradnju (engl. *build*) pojedinih verzija aplikacije. Osim navedenih direktorija, imamo i `lib` direktorij koji se u pravilu koristi za glavni dio produkcijskog kôda. Na glavnom nivou projekta također imamo i `pubspec.yaml` datoteku u kojoj definiramo sve postavke projekta kao što su ime projekta, verzija Flutter i Dart okruženja koje se koristi, verzija aplikacije, ovisnosti na druge lokalne/vanjske biblioteke, postavke priloženih datoteka (engl. *assets*) i drugo.

Osim postavki koje Flutter dodatak stvori sam, u projekt je dodan direktorij `modules` u koji su smješteni svi moduli aplikacije. Moduli su u vidu Flutter projekta lokalni paketi (engl. *packages*) koji služe kao ovisnosti na glavni projekt te su kao takvi postavljeni u spomenutoj `pubspec.yaml` datoteci. Svaki od paketa (modula) imaju izgled kao i cijeli projekt, odnosno imaju svoj `lib` direktorij i `pubspec.yaml` datoteku. Ove dvije značajke definiraju općenito Dart paket koji šire može biti pretvoren u Flutter projekt postavljajući u `pubspec.yaml` datoteci ovisnost na Flutter okruženje i njegovu verziju (postoje stabilne i eksperimentalne verzije). U ovom projektu se koristila stabilna verzija Flutter okruženja.

Korištenjem više modula, koji su ovisnosti cijelog projekta, zadovoljio se *multi-module* oblik aplikacije. *Multi-module* arhitektura nam omogućava smisleno odvajanje kôda u zasebne jedinice koje su idealno neovisne jedna o drugoj. Povod razdvajanja produkcijskog kôda na više modula je zadovoljavanje uvjeta slojevitosti aplikacije u pogledu toka podataka (engl. *dataflow*). Tok podataka se definira na način da je usmjeren iz smjera izvora podataka s vanjskih/lokalnih izvora (model) kao što su podaci koji dolaze s mreže, sistemski podaci ili su

recimo spremljeni u lokalnu bazu, prema odredištu korisničkog sučelja. Korisničko sučelje se smatra kao najgornji sloj, a model kao najniži sloj.

Tok podataka se također može promatrati na način da podaci dolaze s vrha, odnosno da podatke dobivamo korisničkim unosom na korisničko sučelje. Moduli se mogu raditi na način da odvajaju slojeve aplikacije, odvajaju posebne značajke aplikacije ili čak kombinacijom to dvoje. Ova aplikacija koristi kombinaciju to dvoje na način da je više usmjerena prema odvajanju po značajkama zato što je aplikacija uvelike isprepletena ovisnostima koje se protežu uzduž cijelog toka podataka pa i nije bilo potrebe razdvajati aplikaciju po slojevima već po značajkama koje stvarno i jesu neovisni elementi.

U slučaju ove aplikacije najveći dio prometa podataka u najnižem sloju se izvršava preko mreže. U aplikaciji je izdvojen pogled klijenta u mreži i poslužitelja u mreži, odnosno koordinatora u mreži. Za usklađivanje podataka prema klijentima je zaslužan koordinator pa je time kao posebna značajka izdvojen kao zaseban modul. Druga dva modula služe isključivo za definiciju modela (ne u vidu sloja) koji oblikuju podatke koji se šalju preko toka podataka. Ta dva modula su `model` i `events`.

Ostale funkcionalnosti u kôdu su smještene direktno u `lib` direktoriju odvojene u posebne direktorije za bolju organizaciju kôda. Većinom je kôd u tom direktoriju podijeljen na jedinice korisničkog sučelja pa se može reći da se taj segment kôda podudara s prezentacijskim slojem koji sadržava u sebi kôd koji opisuje korisničko sučelje i *business* logiku koja kontrolira stanja vezana za isto sučelje te kao takva je smještena na istoj razini kao i kôd za samo sučelje. Sama komponenta igre je također smještena u ovom direktoriju jer Flame okruženje nudi nerazdvojnu vezu između komponenti korisničkog sučelja i logike koja okružuje te komponente. Najveći dio logike je smješten upravo u ovom dijelu kôda koji je razmotren nešto kasnije u radu.

Ovime je opisan generalni izgled kôda ove aplikacije odnosno njena arhitektura. U radu je opisana svaka od ključnih komponenti ove aplikacije od dna prema vrhu u vidu toka podataka.

4. Implementacija komponenti

4.1. Koordinator

Koordinator (u kôdu definiran kao `coordinator`) se koristi za usklađivanje podataka među klijentima. Apstraktno gledajući, komunikacija među klijentima je uspostavljena preko koordinatora koji ima dvije veze, prema svakom klijentu po jedna, koji je zadužen za prijenos poruka s jednog klijenta na drugi.

Komunikacija se odvija na način da klijent za događaj (engl. *event*), za koji treba obavijestiti sve u mreži, šalje informaciju koordinatoru koji isti taj *event* šalje prema obojici klijenata istovremeno. Ovime se uspostavlja komunikacija na način da svaki od klijenata vodi brigu o stanju igre reagirajući na događaje koje zaprimi od koordinatora umjesto da koordinator sprema kompletno stanje igre te takvo odašilje periodično svim klijentima. Ovakav odabrani pristup komunikacije se naziva *event-driven* pristupom dok je njegova opreka *dedicated server*.

Dedicated server idealno vrši točniji prikaz stvarnosti, odnosno, bolje koordinira igru među igračima, ali dodatno opterećuje fizički poslužitelj koji je zadužen za izvršavanje te funkcije zato što neovisno o ponašanju korisnika, mora slati stalno veću količinu podataka (sve podatke koji definiraju stanje igre) u puno manjem vremenu što bi mogao biti veliki zahtjev za poslužitelja. Npr. ako se želi postići brzina osvježavanja od 120 Hz, što bi diskutabilno učinilo igru jako dobrom u performansama, trebali bi se podaci slati svako 8.33 ms svakome od klijenata.

S druge strane, *event-driven* poslužitelj se koristi za prosljeđivanje događaja koji nose u sebi puno manje podataka (npr. podaci da se neka jedinica pomakla na novu točku prenosi podatke samo o novoj poziciji, x i y koordinatu te informaciju o kojoj je jedinici to riječ) i kao takav se prosljeđuje svim klijentima koji taktom igre sami koordiniraju novi položaj. Isprva se ovaj pristup može učiniti prilično nepouzdanim, ali promatrajući ovakve postavke na lokalnoj mreži, pristup je odličan za točnost prijenosa podataka. U slučaju da igra dopušta komunikaciju na mrežama koje su veće od lokalnih (internet), situacija bi bila vjerojatno znatno drugačija jer

bi vrijeme slanja poruke bilo znatno veće pa bi se potencijalno koristio *dedicated server* umjesto ovakvog.

Glavni razlog odabira ovog pristupa, osim što sama limitacija odabira mreže zadovoljava taj izbor, je taj što funkciju poslužitelja može izvršavati gotovo svaki računalni uređaj koji ima mogućnost spajanja na internet i ne nužno izvrsne performanse računalne snage. U ovu kategoriju spadaju svi danas komercijalni mobiteli, kompjuteri ili poslužitelji koji su svima lako dostupni. Zato se isti taj poslužitelj može pokrenuti i na samom mobitelu koji se koristi i kao klijent. Ovime, potrebna su samo dva uređaja za ostvarivanje ovakve komunikacije gdje je jedan uređaj ujedno i *host* i klijent dok je drugi uređaj samo klijent.

Iako je odabrani pristup odličan i u vidu integriteta podataka i u performansama nije idealan za neke od posebnih slučajeva koji se dešavaju prilikom izvođenja igre. Ako neki od igrača izgubi konekciju ne možemo sa sigurnošću dovesti igru u sinkronizirano stanje na oba klijenta. U tom slučaju moramo ponovno izvršiti konekciju na poslužitelja s obe strane, ali proslijediti kompletno stanje igre. Također možemo dodatno periodično slati kompletno stanje igre s jednog klijenta (sa klijenta koji je ujedno i *host*) čime dodatno zadovoljavamo sinkronizaciju, a nismo zasitili poslužitelja.

U najčešćim slučajevima ova dodatna sinkronizacija nije ni potrebna, ali ju je zgodno imati u raznim slučajevima zasićenja mreže, pada performansi usljed izvršavanja neke druge aplikacije koja znatno opterećuje resurse uređaja i sl.

4.2. Komunikacija među igračima

Za realizaciju opisane konekcije unutar lokalne mreže korištena su dva protokola, *udp* i *tcp* odvojeno. *Udp* je korišten za prvotnu uspostavu konekcije prilikom traženja klijenata međusobno. Prvi igrač odabirom *multiplayer* izbora dolazi do opcije *host* i *join*. *Host* opcijom započinje periodično slanje *udp* poruke svake sekunde s određenog priključka. Poruka prenosi informaciju o *ip* adresi odašiljatelja odnosno *ip* adresu uređaja na kojem je korisnik odabrao opciju *host*. Ovo je *broadcast* poruka koja se odašilje svima na mreži.

Drugi igrač u međuvremenu ulazi također u *multiplayer* način igre i odabire *join* pa time počinje slušati *udp* poruke koje se propagiraju unutar mreže. U trenutku kada dohvati *udp* poruku, dobije informaciju o *ip* adresi na koju se treba povezati te odgovori s *broadcast* porukom koja sadržava informaciju o svojoj *ip* adresi.

Kada *host* zaprimi *udp* poruku koja sadržava *ip* adresu drugog igrača, prekida *udp* poslužitelja koji je periodično odašiljao *udp* poruke te pokreće *tcp* poslužitelja koji stvara dvije veze koju svaku pojedinačno definiraju izvorišna i odredišna *ip* adresa, te izvorišni i odredišni *port*.

Jedna *tcp* veza je između *tcp* poslužitelja i samog sebe (odredišna i izvorišna *ip* adresa je ista) dok je druga *tcp* veza između *tcp* poslužitelja i drugog klijenta. Kada poslužitelj dobije informaciju o uspješnom uspostavljanju obe veze, šalje događaj na koji klijenti reagiraju započinjanjem igre. Važno je napomenuti da su sve priložne datoteke potrebne za *render* učitane već pri samom pokretanju igre, prije uspostavi konekcija, pa se inicijalizacija igre izvrši u istom vremenu, odnosno ne treba dodatno, potencijalno dugo, učitavanje datoteka.

Svaki klijent obavlja obradu događaja koji se šalju sa strane poslužitelja, a `ClientConnection` je klasa koja je zadužena za to. U klasi izvršavamo slušanje poruka koje dobivamo od poslužitelja te vršimo de-serijalizaciju podataka koje dobiju u binarnom zapisu. De-serijaliziramo podatke na način da prvo binarne podatke pretvorimo u *utf8* zapis, odnosno jedan od formata teksta koji je korišten danas kao standard u komunikaciji tekstom.

Ovakva vrsta komunikacije je asinkrona, odnosno možemo dobiti događaj u bilo kojem trenutku u odnosu na drugi događaj, pa čak i u istom (klijent prima dvije poruke kao jednu spojenu), iako nikad nisu ispresjecani podaci jer je svaki događaj zapakiran u *tcp* format. Zato moramo voditi računa o tome da moguće imamo više dospjelih poruka odjednom (najčešće osim jedne budu dvije poruke) na način da stvorimo posrednika (engl. *buffer*) koji vrši čitanje podataka dok se ne isprazni. Cilj je presjeći dolazne podatke na način da odredimo početak i kraj dospjelih podataka pa zato stavljamo početnu i završnu oznaku u poruci. Na početak poruke se postavi naziv događaja koji se šalje dok se na kraj te poruke postavi oznaka *end*.

Ostatak poruke koji se nalazi između ove dvije oznake sadrži tijelo poruke koje je napisano u *json* formatu. *Json* format je najčešći današnji standard prijenosa poruke u obliku objekta koji sadrži određene atribute koji imaju svoj naziv i vrijednost (koje mogu biti brojčane, tekstualne, nizovi ili pak drugi *json* objekti) pa ih je kao takve lako prepoznati i de-serijalizirati današnjim modernim programskim jezicima i okruženjima. Ovime dobivamo potpunu informaciju o dolaznim porukama pa ih naknadno obrađujemo po prirodi dolaznog događaja.

Događaji koje možemo dobiti su događaji o početku igre, kretnji određenog lika, pucanju letjelice ili oružja matičnog broda, kreiranju dijela broda, stvaranju resursnih stijena, pauziranju igre, registriranju klijenta, ponovnom pokretanju igre i o kompletnom stanju. Svi ostali događaji u igri se mogu stvarati i obrađivati lokalno poput dobivanja resursa pucajući u resursne stijene, kupovima određenih dijelova, promjena odabrane letjelice i sl.

Svi navedeni događaji i kôd koji je zadužen za uspostavu konekcije i provođenje iste možemo smatrati slojem modela aplikacije. Nadalje, opisuje se glavna *business* logika koja definira kako se sama aplikacija ponaša prema korisniku obrađujući dobivene događaje i kako manipulira podatke koje dobiva od strane korisnika i naknadno ih šalje sloju modela, odnosno odašilje događaj na već spomenuti način.

4.3. Flame Game

Već prije spomenuto, Flame okruženje je korišteno za implementiranje rješenja svega vezano za logiku igre. Flame okruženje je u stvari biblioteka koja je stvorena u Flutter okruženju u svrhu jednostavnog stvaranja igre u *multi-platform* stilu. Ovim se vodi računa da postoje implementacije za sve platforme koje su već prije spomenute na način da razvojni programer i korisnik ne znaju za razliku o platformi (*platform-agnostic*), barem u idealnom slučaju dok je u stvarnosti potrebno nekad promijeniti i prilagoditi izgled ili kôd da podjednako i pravedno funkcionira na svakoj od platformi. Flame je idealan kandidat za ovo jer je to biblioteka koja je bazirana baš na toj problematici.

Flame kao biblioteka nudi brojne značajke za stvaranje jednostavne pa i kompliciranije 2D igre. Trenutno Flame ima limit na razvoj samo 2D igara, ali to nije bio problem za razvoj ove igre jer je i bila zamišljena u 2D obliku.

Biblioteka je stvarana da se modularno koristi po potrebi, odnosno, nudi značajke koje mogu biti uključene ili isključene u ovisnosti o potrebama igre. Iako ne podržava implementaciju *multiplayer* stila igranja unutar samog Flame okruženja, podržava ugradnju vanjskih Flutter implementacija koje rješavaju taj problem. U slučaju ove igre, korišten je nativni Flutter kôd za uspostavu *utp* i *tcp* konekcija, stoga nije bilo potrebe za nikakvim vezivanjem na vanjske biblioteke.

Odabran je *tcp* način konekcije zato što se cjelokupna komunikacija izvršava u sferi lokalne mreže, o čemu se ranije pričalo, te nije potreban nijedan apstraktniji protokol za komunikaciju i pakiranje podataka. U internet sferi bi se eventualno koristio *http, websocket* ili nešto treće što nam u slučaju ove aplikacije nije potreba.

Flame nam nudi kreiranje karaktera koji su definirani kao složeni objekti raznih atributa potrebne za izvedbu igre, njihovih vizualnih prikaza (*sprites*) i promjena istih u vremenu, odnosno animacija, izvođenje zvuka u igri (koje se bazira na popularnoj biblioteci audioplayers koja se koristi inače u Flutter okruženju za složenije izvođenje zvuka), jednostavnu implementaciju fizike unutar igre i ostalo.

U nastavku je pobliže opisana svaka od komponenti i ostale značajke koje ova biblioteka nudi i korištena je u ovoj aplikaciji.

4.4. Zvuk

Zvuk je u igri izveden pomoću već spomenutog okruženja i koristi se u tek nekoliko mjesta. Zvuk se u igri pojavljuje inicijalno otvaranjem glavnog izbornika u igri te se izvršava u petlji zauvijek sve dok igrač ne odluči započeti igru i ne uđe u nju kada se pokreće drugi zvuk igre koji se također izvršava zauvijek u petlji dok se sama igra ne završi. Tokom igre, mogu se čuti i zvukovi pucanja iz oružja na matičnom brodu ili s borbenog broda. Svaki novi projektil dodaje novi zvuk ispaljivanja projektila te se time ne prekida pozadinska muzika igre. Osim

ovog zvuka pucanja, dodan je i zvuk eksplozije koji se događa kada se uništi jedan od brodova ili neki resursni kamen. Ovaj zvuk eksplozije se također dodaje poviše svih zvukova te ne prekida nijedan koji je u trenutku izvršavanja.

Zvukovi su pokrenuti određenim događajima koji su već spomenuti te neki od njih kao što su izvođenje pozadinske glazbe se moraju manualno pokrenuti pri izvršenju određenog dijela kôda dok agregirajuće zvukove poput pucanja i eksplozije pokrećemo prilikom izvođenja nekih od metoda koje su ugrađene u neke od komponenti Flame okruženja. Jedna od njih je `onRemove` metoda koja definira uništenje određene komponente pa ćemo za komponente za čije uništenje želimo pokrenuti zvuk eksplozije pokrenuti taj zvuk pozivanjem određene funkcije na mjestu pozivanja te metode. Primjer koda za izvršavanje zvuka je prikazan u ispisu 1.

```
class Player {
 static final List<AudioPlayer> backgroundStack = [];
 static Future<void> playBackgroundIdle() async {
   for (var element in backgroundStack) {
     element.stop();
   }
   final player = await
FlameAudio.loopLongAudio("space_background.mp3");
   backgroundStack.add(player);
 }

 static Future<void> playBackgroundGame() async {
   for (var element in backgroundStack) {
     element.stop();
   }
   final player = await
FlameAudio.loopLongAudio("space_background_1.mp3");
   backgroundStack.add(player);
 }

 static Future<void> playLaser() async {
   FlameAudio.play("laser.mp3",volume: 0.2);
 }

 static Future<void> playMineExplosion() async {
   FlameAudio.play("mine_explosion.mp3",volume: 0.6);
 }
 static Future<void> playShipExplosion() async {
   FlameAudio.play("ship_explosion.mp3",volume: 0.6);
```

```
  }
}
```

Ispis 1: *Audio player*

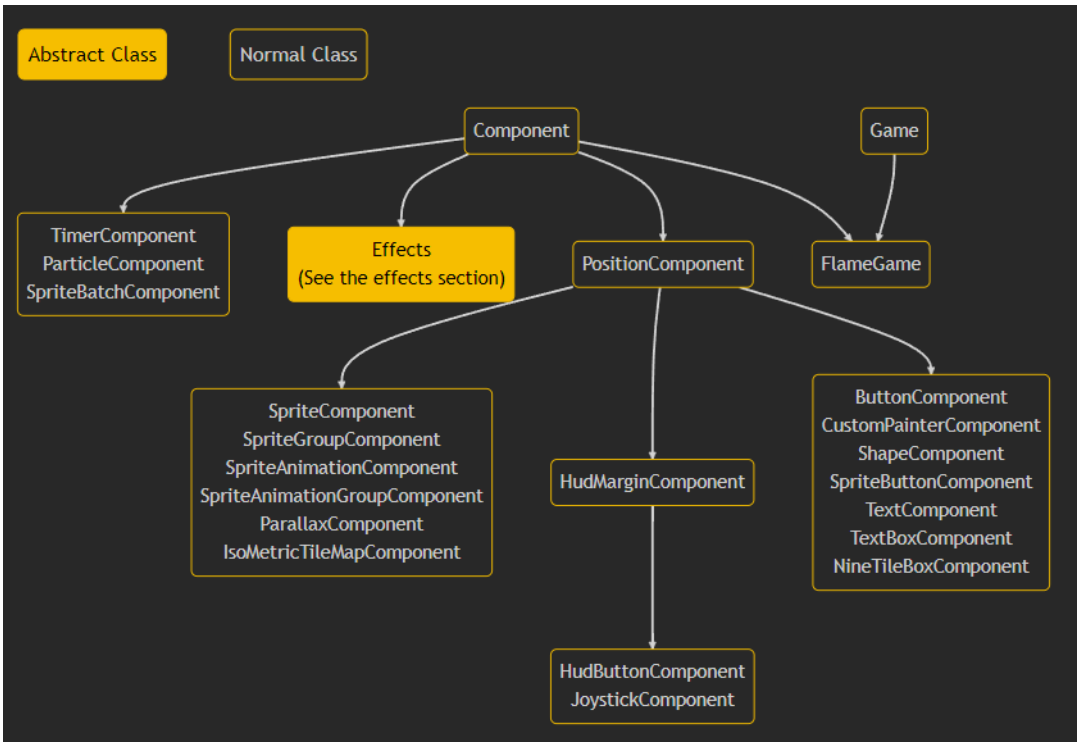
4.5. Komponente

Flame koristi brojne komponente u obliku Dart klasa koje definiraju kako se pojedini element iscrtavaju i ponašaju u igri.

Glavni element na kojem se Flame zasniva je `Component` klasa koja definira osnovne atribute koja odgovara svakom liku kojeg vidimo na korisničkom sučelju. `Component` klasa definira životni ciklus takvog lika, ondnosno ima ugrađene metode `onLoad()` i `onDestroy()` kojima definira početak i kraj života te komponente. Također nudi kontrolu nad svojim nasljednicima (engl. *children*) i svojim uzlaznicima (engl. *ascendants*) na način da postoji referenca na onu komponentu čiji je ona nasljednik te daje mogućnost dodavanja komponenti niže u stablo nasljeđivanja, odnosno omogućava daljnje nasljeđivanje drugim komponentama.

U ovoj aplikaciji je korišten ovaj princip nasljeđivanja u primjeru matičnog broda sa svojim dijelovima. Svaki dio je nasljednik matičnog broda kao i onih dijelova na koje se nastavilo nasljeđe. Npr. na matični brod se izgradio štit na koji se dodatno izgradilo oružje, tako imamo vezu među njima na način da je oružje nasljednik štita, a štit je nasljednik matičnog broda.

Komponenta nudi još mnogo kontrole u vezi svog životnog ciklusa, ali te značajke nisu korištene u ovoj aplikaciji, stoga je zanemarena.



Slika 3: Stablo Flame komponenti

Od priloženih komponenti koje su prikazane na slici 3, u aplikaciji su se koristile samo `FlameGame/Game` i `PositionComponent/SpriteComponent`. Iako se nude i druge zgodne komponente kojima se mogu riješiti pojedini problemi u igri poput vremenskog brojača, korištena su prilagođena rješenja koja nude veću jednostavnost i bolju kontrolu.

4.5.1. PositionComponent

Najviše se od navedenih koristila `PositionComponent` komponenta koja osim navedenih karakteristika obične komponente nudi i informacije o poziciji i veličini komponente. Ove karakteristike konkretnije opisuju klasu obične komponente te postaju ovime opipljivije. Komponenta pozicije osim same pozicije na ekranu ili igri nudi i informacije o veličini i kutu te komponente. Veličina i kut se dodatno povezuju informacijom o sidrištu (engl. *anchor*) koje definira oko koje točke s obzirom na veličinu te komponente se ona okreće.

4.5.2. FlameGame

Još jedna bitna komponenta za spomenuti koja služi kao izvorišna točka cijele igre je već navedena `FlameGame`. Nudi kontrolu prilikom pokretanja igre, učitavanja svih likova te njihove promjene u vremenu. Najbitnija je zadnje spomenuta značajka koja kontrolira cijelo kretanje i ostala zbivanja komponenti unutar igre. Sve komponente koje su ugrađene u igri se ažuriraju nakon pozivanja metode ažuriranja igre koja se poziva pri obradi svakog okvira (engl. *farme*) što je najčešće svako 16.6 ms, a odgovara brzini osvježavanja 60 Hz.

Ažuriranje se prenosi preko svih elementa igre koje se nalaze u njoj te je nastavno ažuriranje dodatno definirano u svakoj od komponenti posebno ako je potrebno. Svakoj komponenti koja je dodana kao dijete u igru, ne treba se manualno ažurirati stanje već Flame to odrađuje za nas, međutim promjena pozicije se manualno odrađuje u ovoj igri zbog bolje kontrole.

4.5.3. Movable

`Movable` klasa (prikazana u ispisu 2) je prilagođena klasa koja služi za definiranje komponenti kojima je definirana kretnja tokom igre. U ovu skupinu npr. ne ulaze resursne stijene koje se periodično stvaraju tokom igre jer nemaju definiranu kretnju, odnosno stoje uvijek na istom mjestu. U kôdu niže vidimo da se `update` metoda prilagođava za `Movable` komponentu, to jest, manualno mijenjamo kut i poziciju takvih komponenti.

```
@override
void update(double dt) {
 updateAngle(dt);
 updatePosition(dt);
 super.update(dt);
}
```

Ispis 2: Ažuriranje `Movable` komponente

Kut se mogao promijeniti koristeći Flame mehanizam ali ne paše intuitivnosti ove igre. Od letjelice se očekuje da se brže okreće što je kut od smjera kretanja do smjera prema odredištu veći, a sporije što je isti manji. Kôd koji kontrolira takvo okretanje je prikazan u ispisu 3.

```
void updateAngle(double dt) {
  ///Native rotation is not intuitive for this game so it is
implemented in a custom way
 if (destination != null) {
   final dir = Vector2(1, 0)..rotate(angle);
   final diff = destination! - position;
   final dot = dir.x * diff.x + dir.y * diff.y;
   final det = dir.x * diff.y - dir.y * diff.x;
   final ang = atan2(det, dot);
   if (ang.abs() >= 0.01) {
     angle += rotationSpeed * dt * ang;
   }
 }
}
```

Ispis 3: Ažuriranje kuta `Movable` komponente

Ovime je zadovoljena promjena kuta da je manja što se tijelo u pokretu približava smjeru prema odredištu uz dodatni *rotationSpeed* parametar koji linearno mijenja brzinu rotacije. U ispisu 4 je prikazan način na koji se zadovoljava promjena pozicije kretajuće komponente. Cijela kretnja se bazira na tome da se pozicija mijenja kada je definirana destinacija (koju dobijemo reagiranjem na događaj kretanja s poslužitelja ako odgovara toj komponenti) te pomoću dodatnog parametra brzine te komponente mijenjamo njenu trenutnu poziciju ako ne izlazi van ekrana igre. Također u ovom dijelu mijenjamo stanja komponente koja su zaslužna za izgled ili animaciju komponente, pa tako razlikujemo kada je komponenta u kretnji, u mirnom stanju ili oštećena.

```
void updatePosition(double dt) {
 if (destination == null) {
   return;
 }
 final diff = destination! - position;
 final newPosition = position + diff.normalized() * speed * dt;
 if (newPosition.x > 0 &&
```

```
      newPosition.y > 0 &&
      newPosition.x < Constants.worldSizeX &&
      newPosition.y < Constants.worldSizeY) {
    position = newPosition;
  }
  if (current != MovableState.damaged) {
    current = MovableState.moving;
  }
  if (diff.length < Constants.proximityDistance) {
    destination = null;
    if (current != MovableState.damaged) {
      current = MovableState.idle;
    }
  }
}
```

Ispis 4: Ažuriranje pozicije `Movable` komponente

4.5.4. Character

Iduća komponenta koja je napravljena prilagodbom u svrhu apstrahiranja svih komponenti koje su svjesne događaja s poslužitelja je `Character` koja je prikazana u ispisu 5.

```
/// Generalized mixin which represents server-aware in-game
character
///
/// [characterId] should be used only to handle server-aware
characters such as motherships, fighters and mines
mixin Character implements TeamCharacter {
 int get characterId =>
getIt<CharacterManager>().getCharacterId(character: this);
}

mixin TeamCharacter on PositionComponent {
 abstract Team team;
 abstract String name;
 abstract bool picked;
}
```

Ispis 5: `Character i TeamCharacter` *mixin*

Mixin se općenito u Dart jeziku koristi kao proširenje na klasu gdje se definiraju metode koje se mogu, ali ne trebaju pozvati u klasama koje koriste taj *mixin*. Kasnije u kôdu često treba informacija o identifikaciji neke komponente, ali na način da je jednoznačno određeno na nivou poslužitelja, a ne samo na lokalnom nivou. Također se koristi i `TeamCharacter` *mixin* koji pobliže definira komponentu kojem timu pripada (postoji tim 1, tim 2 i neutralni tim), imenom i činjenicom je li odabrana. Oba *mixin* elementa su definirana samo na komponentama pozicije.

4.5.5. Shooter

Dodatno na takvu apstrakciju karaktera (prije spomenut `Character`) je definiran `Shooter` *mixin* koji bolje opisuje ponašanje svih komponenti koje imaju sposobnost ispaljivanja projektila i prikazan je u ispisu 6.

```
mixin Shooter on Character {
 DateTime lastShot = DateTime.now();
 abstract int damage;

 Vector2? _target() {
   final candidate = getIt<CharacterManager>()
       .characters
       .where((element) => element.team == Team.neutral || (team !=
element.team))
       .reduce((a, b) =>
           a.absolutePosition.distanceTo(absolutePosition) <
b.absolutePosition.distanceTo(absolutePosition) ? a : b);
   if (candidate.absolutePosition.distanceTo(absolutePosition) <=
Constants.shootingDistance) {
     return candidate.absolutePosition - absolutePosition;
   }
   return null;
 }

 @override
 void update(double dt) {
   super.update(dt);
   final tar = _target();
   final now = DateTime.now();
   if (tar != null && now.difference(lastShot).inMilliseconds >
Constants.shotPeriodMillis) {
     getIt<ClientConnection>().addEvent(ShootEvent(
```

```
          team: team,
          startX: absolutePosition.x,
          startY: absolutePosition.y,
          dirX: tar.x,
          dirY: tar.y,
          damage: damage));
      lastShot = now;
    }
  }
}
```

Ispis 6: `Shooter` *mixin*

`Shooter` *mixin* nudi način na koji definira metu u svojoj blizini koja je definirana kao konstanta, što bi značilo da svaka komponenta koja je sposobna ispaljivati projektile puca u prvu metu u određenom radijusu od sebe i to periodično također određeno konstantom. Konstante brzine ispaljivanja projektila i radijusa za mete su jednake za sve komponente koje ispaljuju projektile. Jedine komponente koje mogu ispaljivati projektile su borbeni brod i oružje koje se može ugraditi na matični brod kao njegov dio. U kôdu također vidimo da se ispaljivanje uslijed pronađene mete vrši na način da se pošalje događaj ispaljivanja projektila prema poslužitelju. Na takav događaj odgovara odgovarajuća meta na način da primi štetu.

4.5.6. Komponente kolizije

Komponenta koju je pogodio projektil zaprima štetu pozivajući metode koje nudi Flame *mixin* `CollisionCallbacks`. Iz ovog *mixin* elementa se koristi metoda koja se izvršava prilikom kolizije dvaju tijela. Tako npr. `Bullet` klasa, koja pobliže definira projektil, proširuje taj *mixin* pa i koristiti spomenutu metodu kao u ispisu 7.

```
@override
void onCollision(Set<Vector2> intersectionPoints, PositionComponent
other) {
 super.onCollision(intersectionPoints, other);
 if (other is TeamCharacter && other.team != team && other is!
Bullet) {
   getIt<SpaceArenaGame>().remove(this);
   if (other is HasHealth) {
```

```
      (other as HasHealth).currentHealth--;
      if ((other as HasHealth).currentHealth <= 0) {
        if (getIt<CharacterManager>().characters.contains(other)) {
          getIt<CharacterManager>().add(RemoveCharacter(character:
other as Character));
        } else if (other.parent != null) {
          other.removeFromParent();
        }
      }
    }
  }
}
```

Ispis 7: Metoda kolizije za projektil

Ovu metodu kolizije koristi i druga pogođena komponenta koja je pokretna i koja je definirana drugačije na način koji je prikazan u ispisu 8.

```
@override
Future<void> onCollision(Set<Vector2> intersectionPoints,
PositionComponent other) async {
 super.onCollision(intersectionPoints, other);
 if (other is Bullet && other.team != team) {
   if (thisPlayer()) {
     getIt<SpaceArenaGame>().camera.shake(intensity: 3);
   }
   if (current != MovableState.damaged) {
     current = MovableState.damaged;
     animation?.onComplete = () {
       current = MovableState.idle;
       animations?[MovableState.damaged]?.reset();
     };
   }
 }
}
```

Ispis 8: Metoda kolizije za `Movable` klasu

I kao zadnje, metodu kolizije ćemo koristiti i za resursne stijene jer i one nude drugačije ponašanje pri koliziji za razliku od ostalih. Takvo ponašanje je opisano u ispisu 9.

```
@override
```

```
Future<void> onCollision(Set<Vector2> intersectionPoints,
PositionComponent other) async {
 super.onCollision(intersectionPoints, other);
 if (other is Bullet && other.team ==
(getIt<CharacterManager>().pickedCharacter)?.team) {
   getIt<BankBloc>().add(AddValue(mineType: mineType));
 }
}
```

Ispis 9: Metoda kolizije za komponente resursne stijene

Usporedbom navedenih metoda kolizije vidimo da projektil brine o šteti lika kojeg je pogodio te u slučaju maksimalne štete lika, izbacuje ga iz igre, odnosno uništava. Pokretne komponente se brinu samo o efektima prilikom kolizije s metkom jer je već šteta obrađena od strane projektila pa se shodno tome trese kamera ako je riječ o igračevoj komponenti te promijeni stanje koje definira kako lik trenutno izgleda pa u slučaju štete promijeni oblik u animaciju ranjenog lika. Resursne stijene iz kolizije samo dodaju potrebne resurse određenom igraču koji je zaslužan za gađanje te resursne stijene.

Mixin `HasHealth` se dodaje svakoj komponenti koja se uništi za određenu količinu štete koju primi koja je definirana u istom *mixin* elementu. Prikazan je u ispisu 10.

```
mixin HasHealth {
 int get maxHealth;

 abstract int currentHealth;
}
```

Ispis 10: *Mixin* `HasHealth`

Kolizija se može desiti jedino među komponentama koje imaju definirane tzv. *hitbox* prostore. *Hitbox* opisuje prostor komponente koji prilikom presjecanja drugog *hitbox* prostora uzrokuje koliziju i to već samim dodirivanjem te dvije površine. Sve komponente u ovoj igri su otprilike podjednake veličine u horizontalnom i vertikalnom smjeru pa se *hitbox* definirao kao simetričan dvodimenzionalan lik. Dva idealna kandidata za takav uvjet su kvadrat i krug, ali je odabran krug od ta dva jer zauzima površinu koja bolje odgovara samim crtežima likova. Jedina

iznimka je projektil koji je definiran *hitbox* pravokutnikom zbog svog specifičnog oblika (tanka laserska zraka, puno duži u jednom smjeru od drugog okomitog).

Zaključno, u vidu kolizija, komponente se mogu smatrati kao krugovi koji se nalaze u prostoru koji vrše koliziju s drugim komponentama prilikom presjecanja ili prilikom pogotka tankog pravokutnog projektila.

4.5.7. SpriteComponent

`SpriteComponent` je dodatna komponenta koja ima veliku važnost u ovoj igri. Ovom komponentom se određuje vizualni izgled komponente. Izgled je definiran jednom ili više slika koje se uzastopno izmjenjuju da dočaraju igraču kretnju komponente. Tako se npr. borbenom brodu vidi animacija pogona prilikom kretanja te se zaustavi kada i brod stane. Prilikom samog pokretanja igre (prije povezivanja s poslužiteljem) vrši se učitavanje svih mogućih *sprite* elemenata kojima je opisan oblik komponente jer za to treba najviše vremena pa kasnije pokretanje igre prilikom povezivanja s poslužiteljem čini bržim. Svi *sprite* elementi su upravljani pomoćnom klasom `SpriteManager` koja brine o učitavanju svih *sprite* elemenata koje se kasnije pridijele svakoj komponenti zasebno. Primjer učitavanja nekog *sprite* elementa je prikazan u ispisu 11.

```
///Bullet
bulletSprite = await
getIt<SpaceArenaGame>().loadSprite("bomber/Charge_1.png");
```

Ispis 11: Učitavanje slike lasera

4.5.8. PartsManager

`PartsManager` je klasa koja upravlja postavljanjem svih dijelova matičnog broda. Klasa vrši svoju funkciju tako da mapira svaki postavljeni dio u dvodimenzionalni prostor s

cjelobrojnim koordinatama gdje je na poziciji (0,0) postavljen inicijalno matični brod. Ako dio postavimo neposredno naprijed od broda, spomenuta klasa mapira taj dio u dvodimenzionalnom prostoru na točku koordinate (0, 1), odnosno povećava y vrijednost za 1 od one vrijednosti koja je pridodijeljena onom dijelu s kojeg vežemo novi dio.

Ako se postavlja dio u suprotnom smjeru priča je analogna samo što se smanjuje y vrijednost za 1 umjesto da se povećava. Za okomitu orijentaciju priča je analogna s tim da je desni smjer odabran kao pozitivan, a vrijednost koju mijenjamo odgovara x koordinati pa ako dodamo dio na desnu stranu od broda, dijelu je pridodijeljena točka koordinate (1,0). Ovime bilježimo i pratimo mogućnosti postavljanja tih dijelova kontrolirajući međusobnu susjednost ili moguće preklapanje pa se ovime onemogućuje igraču da izgradi neki dio koji se preklapa s već postojećim dijelom.

Ova klasa se također koristi i za izračun brzine matičnog broda s obzirom na najudaljeniji izgrađeni dio na tom brodu (gledamo najveću euklidsku udaljenost između broda i svih dijelova pojedino te nju uzimamo kao faktor za izračun brzine okretanja matičnog broda).

4.5.9. Događaji pritiska tipki i dodira

Flame također nudi korištenje metode koja se izvodi kada se pritisne određena tipka na tipkovnici, ili metoda koja se pokrene prilikom lijevog ili desnog klika miša. *Mixin* elementi koji se koriste za dodavanje tih metoda Flame igri kao komponenti su `TapDetector`, `SecondaryTapDetector` i `KeyboardEvents`. Implementacija rješenja je navedena u ispisu 12.

```
@override
KeyEventResult onKeyEvent(RawKeyEvent event, Set<LogicalKeyboardKey>
keysPressed) {
 if (keysPressed.contains(LogicalKeyboardKey.tab)) {
   if (_characterManager.characters.where((element) => element.team
== _characterManager.team).length == 1) {
     return super.onKeyEvent(event, keysPressed);
   }
   final character = (_characterManager.pickedCharacter ==
_characterManager.fighter
```

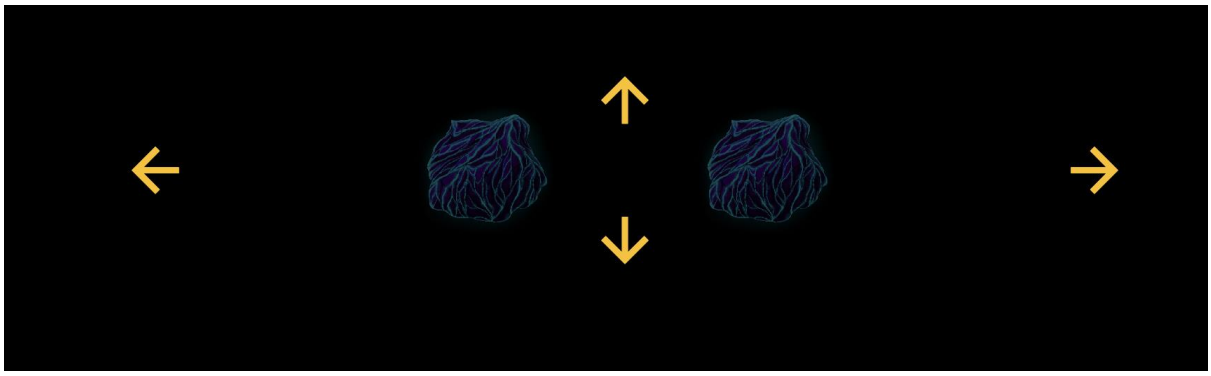
```
        ? _characterManager.mothership
        : _characterManager.fighter) as Character?;
    if (character != null) {
      _characterManager.add(PickCharacter(character: character));
    }
  } else if (keysPressed.contains(LogicalKeyboardKey.digit1)) {
    final bankState = getIt<BankBloc>().state;
    final part = PartType.values.firstWhere((element) => element.key
== 1);
    final valid =
        part.price.validate(other: Price(gold: bankState.gold,
crystal: bankState.crystal, plasma: bankState.plasma));
    if (valid) {
      getIt<OverlayCubit>().placePart(type: part);
    }
  } else if (keysPressed.contains(LogicalKeyboardKey.digit2)) {
    final bankState = getIt<BankBloc>().state;
    final part = PartType.values.firstWhere((element) => element.key
== 2);
    final valid =
        part.price.validate(other: Price(gold: bankState.gold,
crystal: bankState.crystal, plasma: bankState.plasma));
    if (valid) {
      getIt<OverlayCubit>().placePart(type: part);
    }
  } else if (keysPressed.contains(LogicalKeyboardKey.digit3)) {
    final bankState = getIt<BankBloc>().state;
    final part = PartType.values.firstWhere((element) => element.key
== 3);
    final valid =
        part.price.validate(other: Price(gold: bankState.gold,
crystal: bankState.crystal, plasma: bankState.plasma));
    if (valid) {
      getIt<OverlayCubit>().placePart(type: part);
    }
  }
  return super.onKeyEvent(event, keysPressed);
}
```

Ispis 12: Metoda pritiska neke od tipki

U kôdu se može vidjeti da igra reagira na pritisak tipke *tab*, tipke broja 1, broja 2 i broja 3. Prilikom pritiska na tipku *tab* mijenjamo odabranog igrača ako u igri postoji trenutno više od jedne komponente koja se može odabrati. Pritiskom na tipku 1 odabire se štit, tipkom 2 se odabire oružje, a tipkom 3 pogon kao dio koji se želi postaviti na matični brod. Ako postoji

dovoljno resursa, nastavlja se postavljanje dijela na način da se odabire prvo dio matičnog broda ili matični brod na kojem se želi postaviti taj dio. Ako je igrač uspješno odabrao dio klikom na njega, dobiva upit na koju stranu od smjera gibanja matičnog broda želi postaviti taj dio.

Upit se sastoji od četiri strelice po sredini ekrana gdje je svaka postavljena na poziciju i u smjeru kojeg predstavlja i ako je omogućena gradnja u tom smjeru, prikazana je žutom bojom, a ako ne, prikazana je sivom bojom (prikazano na slici 4). Ako igrač nadalje odabere neku od omogućenih strana, dio se uspješno izgradi na tom mjestu u odnosu na odabrani dio i upit se zatvara. Cijela opisana logika se odvija koristeći `Bloc` klasu koja je dio *business* logike koja okružuje dio korisničkog sučelja koji predstavlja segment za odabiranje gradnje dijelova matičnog broda i brisanje tih dijelova.



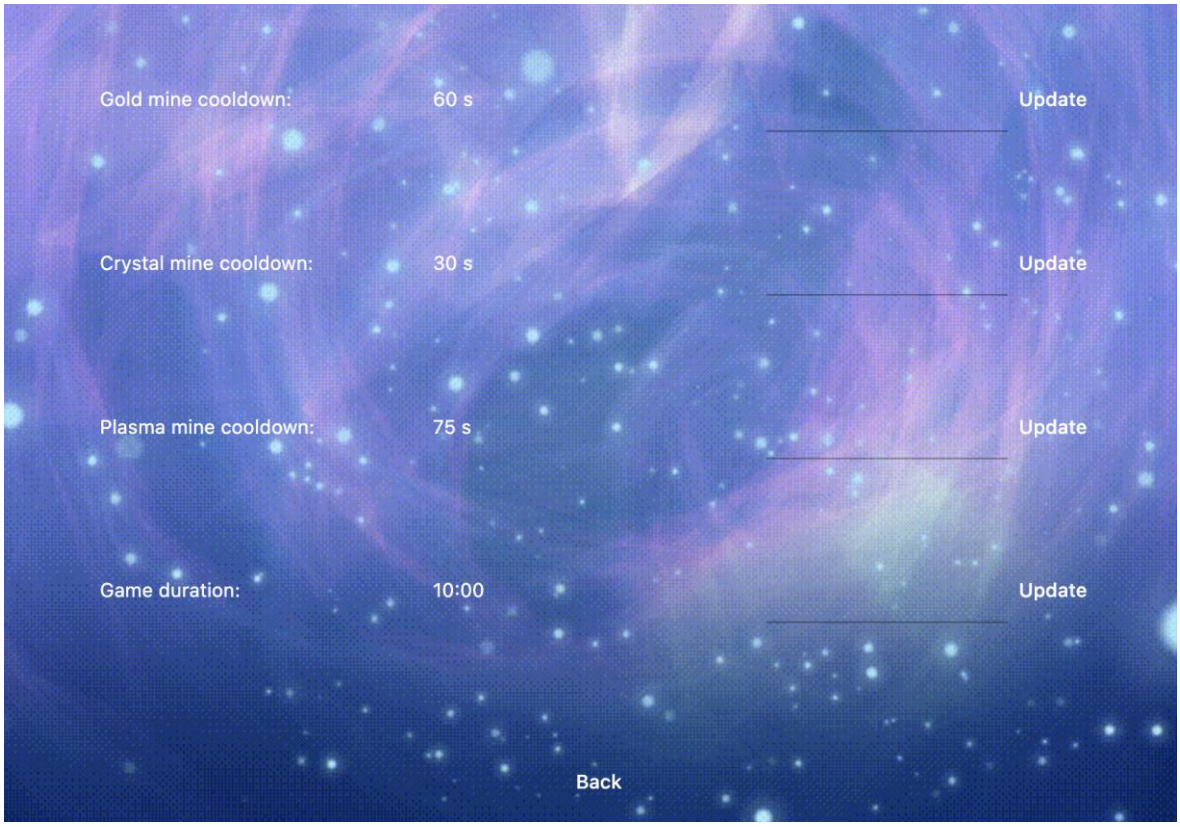
Slika 4: Strelice odabira gradnje dijelova broda

4.5.10. Brojač vremena

U igri je napravljen prilagođen brojač vremena koji odbrojava od početka igre sve do kraja. Odbrojava 10 minuta, što je početno postavljena vrijednost ili vrijeme koje se može postaviti u postavkama igre gdje možemo doći odabirom iz glavnog izbornika. Brojač je izveden u obliku `Bloc` klase koja ima svoje definirane događaje za odbrojavanje, pauziranje i ponovno pokretanje brojača. Pauziranje i ponovno pokretanje se izvodi pritiskom na sliku u gornjem desnom kutu na ekranu i dostupno je na korištenje oba igrača. Pritiskom na tu sliku šalje se događaj za pauziranje ili ponovno pokretanje igre koji primaju oba igrača preko odašiljanja istog od strane poslužitelja te reagiraju na taj događaj.

4.5.11. Postavke

U igri je omogućena promjena nekih od postavki igre kao što su vrijeme ponovnog stvaranja pojedine resursne stijene i ukupno dopušteno vrijeme igre. Svaka od postavki se može promijeniti unosom broja sekundi koje predstavljaju novu vrijednost. Dopuštene promjene postavki u igri su prikazane na slici 5.



Slika 5: Meni postavki

Ove vrijednosti su osnovne komponente koje mijenjaju dinamiku igre te se trebaju mijenjati s opreznošću da ne dođe do prekomjernog stvaranja resursa ili da igra ne traje besmisleno kratko ili dugo. Ostale bitne vrijednosti kao što su brzina pucanja projektila, šteta projektila, trajnost brodova i dijelova broda i druge bitne vrijednosti u igri nisu prilagodljive tako da igra zadrži svoj karakterističnu izgled i ponašanje.

5. Zaključak

Flame i Flutter okruženja su očito dobri alati za razvoj aplikacija kojima se može doći do željenog cilja. Može se vidjeti da su jako dobri za razvoj raznih 2D igara (nažalost limitacija je trenutno da ne možemo lagano razvijati u spomenutim okruženjima 3D igre) uz neke prednosti i mane. Što se tiče samog Flutter okruženja može se vidjeti da je odličan alat za dodavanje dodatnih komponenti korisničkog sučelja kao što su u ovoj igri razni izbornici ili statusne trake i slično. Valja napomenuti da trenutno Flame ne nudi svoja rješenja

implementacije konekcije već se sama moraju implementirati koristeći Flutter kao u ovoj aplikaciji ili slično što može otežati i usporiti razvoj aplikacije, iako za jednostavnije slučajeve konekcije i ne stvara veliki problem.

Sve u svemu, ova okruženja su prilično zrela za ozbiljniji razvoj igara te se može preporučiti svakome na korištenje za istu svrhu.

6. Literatura

[1] https://pub.dev (posjećeno 25.8.2023.)

[2] https://docs.flame-engine.org/latest/ (posjećeno 25.8.2023.)

[3] *Clean architecture: Robert C. Martin* (posjećeno 25.8.2023.)

[4] https://bloclibrary.dev/#/ (posjećeno 25.8.2023.)

[5] https://dev.to/mjablecnik/flutter-modular-architecture-2f15 (posjećeno 25.8.2023.)

[6] https://api.flutter.dev/flutter/dart-io/RawDatagramSocket-class.html (posjećeno 25.8.2023.)

[7] https://www.techtarget.com/searchnetworking/definition/TCP (posjećeno 25.8.2023.)