



Universitatea
Transilvania
din Brașov
FACULTATEA DE MATEMATICĂ
ȘI INFORMATICĂ

Programul de studii:
Informatică Aplicată

Lucrare de licență Aplicație android de divertisment

Autor: **Alexandru Dogaru**
Coordonator științific: **Conf.dr. Deaconu Adrian-Marius**

Brașov, 2023



Cuprins

1. Introducere	2
1.1. Prezentarea temei alese	3
1.2. Studiu comparativ cu aplicațiile existente	4
1.3. Structura lucrării	4
1.4. Enumerarea părților originale	5
2. Tehnologii folosite	5
2.1. Limbajul Java	5
2.2. Platforma Android	8
2.3. Android Studio	8
2.4. GIMP	9
2.5. Firebase	9
3. Descrierea aplicației	11
3.1. Bucla de joc și Clasa Game	11
3.2. Generarea mapei	14
3.3. Elementele grafice	15
3.3.1. Sprite și Spritesheet	15
3.3.2. Tile	17
3.3.3. Tilemap	18
3.4. Clasa Bomb și clasa Explosion	20
3.5. Clasa Player	23
3.5.1. OfflinePlayer	33
3.5.2. OnlinePlayer	37
3.5.3. OnlineEnemy	40
3.5.4. OfflineEnemy	41
3.6. Mod Singleplayer	49
3.7. Mod Multiplayer	52
3.8. Firebase	54
3.9. Activitățile	56
4. Utilizarea aplicației	58
5. Concluzii și perspective de dezvoltare	63
5.1. Concluzii	63
5.2. Perspective de dezvoltare	63
5.3. Bibliografie	64



1. Introducere

Generațiile actuale au privilegiul de a fi contemporane cu o perioadă de maximă dezvoltare și inovație în multe domenii, dar în special, în domeniul informaticii și particular în cel al jocurilor pe dispozitive.

Primele jocuri de pe telefon au apărut în jurul anului 1998, odată cu lansarea primelor telefoane mobile cu ecran color. Aceste jocuri erau foarte simple, de obicei constau din puzzle-uri sau jocuri de memorie și erau distribuite prin intermediul operatorilor de telefonie mobilă. Cu toate acestea, aceste jocuri au început să devină din ce în ce mai populare în rândul utilizatorilor de telefoane mobile, datorită ușurinței cu care puteau fi descărcate și instalate.

În anii următori, tehnologia a evoluat și au apărut telefoane mobile cu procesoare puternice și ecrane mai mari, care au permis dezvoltarea unor jocuri mai complexe și grafic mai bine realizate. Acest lucru a dus la apariția unor jocuri de acțiune, jocuri de aventură și chiar jocuri de sport.

Cu timpul, au apărut și jocuri cu mai multe nivele și caracteristici complexe, precum și jocuri multiplayer. În prezent, telefoanele mobile sunt utilizate pentru a juca jocuri din ce în ce mai sofisticate, cum ar fi jocuri cu grafică 3D, jocuri cu mai multe personaje și jocuri cu mai multe moduri de joc.

Primele jocuri de pe telefon au evoluat de la jocuri simple la jocuri complexe și grafic sofisticate, oferind utilizatorilor o varietate de opțiuni de divertisment și o experiență de joc mai completă. Acest lucru a dus la creșterea popularității jocurilor pe telefon și a contribuit la dezvoltarea industriei jocurilor mobile.

În plus, apariția jocurilor pe telefon a permis accesul la aceste jocuri pentru o audiență mult mai largă, inclusiv pentru cei care nu aveau acces la consolă sau PC-uri de joc. Acest lucru a dus la o creștere a numărului de jucători și la o mai mare diversitate a jocurilor disponibile.

Cu toate acestea, jocurile mobile au și o serie de provocări, cum ar fi problemele legate de protecția datelor și a confidențialității, precum și problemele legate de dependență și sănătate mentală. În consecință, este important ca utilizatorii să fie conștienți de aceste probleme și să ia măsurile necesare pentru a se proteja și a se bucura în siguranță de jocurile mobile.



Figura 1.1: Evoluția jocurilor mobile



1.1. Prezentarea temei alese

Tema aleasă are ca punct de plecare jocul „Bomberman”. Istoria jocului Bomberman începe în anul 1985, când jocul a fost lansat pentru prima dată pe platforma Nintendo Entertainment System (NES). Jocul a devenit rapid popular în Japonia și a fost apoi lansat în Statele Unite și Europa.

În timpul anilor '80 și începutul anilor 90, Bomberman a fost lansat pe o varietate de platforme, inclusiv pe Super Nintendo Entertainment System (SNES), Sega Genesis, și altele. Acestea au inclus și versiuni multiplayer, permițându-le jucătorilor să joace împreună. În anii 1990 și 2000, jocul Bomberman a continuat să fie lansat pe o varietate de platforme, inclusiv pe PlayStation, Nintendo 64 și Game Boy Advance. De asemenea, au fost lansate și versiuni online, permițându-le jucătorilor să joace împreună prin intermediul internetului.

În prezent, jocul Bomberman este încă popular și este disponibil pe o varietate de platforme, inclusiv pe consola Nintendo Switch și pe dispozitive mobile. De asemenea, sunt disponibile și versiuni online, permițându-le jucătorilor să joace împreună prin intermediul internetului. În ciuda faptului că au trecut mai bine de trei decenii de la lansarea sa inițială, jocul Bomberman continuă să fie apreciat de fani și să fie considerat ca unul dintre clasicii jocurilor video.

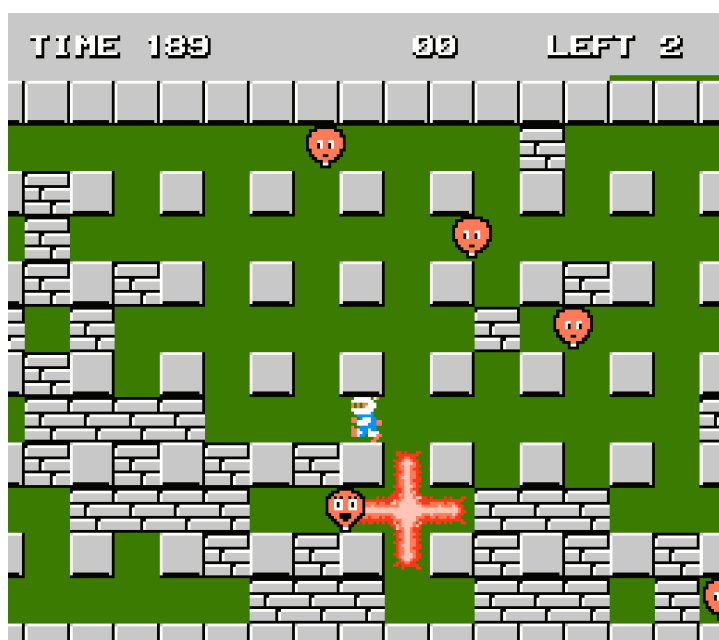


Figura 1.2: Jocul Bomberman

Mecanica jocului Bomberman se bazează pe utilizarea de bombe pentru a distruge obstacole și pentru a elimina inamicii. Jucătorul controlează un personaj numit Bomberman, care poate pune bombe pe terenul de joc și se poate mișca în jurul lor. După o perioadă de timp, bombele explodează, distrugând obstacolele și eliminând inamicii din apropiere. Jucătorul poate utiliza aceste explozii pentru a accesa zonele ascunse ale terenului de joc sau pentru a elimina inamici care în mod normal ar fi invincibili. Jocul are, de obicei, mai multe niveluri, fiecare cu un set de inamici și obiective diferite. Jucătorul trebuie să treacă prin niveluri și să elimine toți inamicii pentru a trece la următorul nivel.

În unele versiuni ale jocului, jucătorul poate colecta power-up-uri care îi oferă abilități suplimentare, cum ar fi capacitatea de a pune mai multe bombe simultan sau



de a face exploziile mai mari. În versiunile multiplayer, jucătorii pot juca împotriva altor jucători, încercând să se elimine unul pe celălalt prin utilizarea bombei. Acest tip de joc poate fi jucat atât local, cât și online.

În general, mecanica jocului Bomberman se bazează pe strategie și gândire rapidă, necesitând ca jucătorii să planifice și să reacționeze rapid la situațiile din joc pentru a reuși să treacă prin niveluri și să elimine inamicii.

Lucrarea are ca temă „Aplicație android de divertisment” și presupune realizarea unui joc care folosește mecanica jocului Bomberman cu scopul de a elimina inamicii. Aceștia sunt agenți inteligenți bazați pe decizii luate în funcție de anumite reguli în modul Singleplayer, sau alte persoane în modul Multiplayer. Câștigătorul este caracterul rămas ultimul în joc.

1.2. Studiu comparativ cu aplicațiile existente

Una dintre cele mai cunoscute aplicații originale Bomberman este „Bomberman Classic”, care este disponibil pentru diferite platforme, inclusiv iOS și Android. Aceasta oferă o experiență de joc asemănătoare cu versiunea originală a jocului, cu nivele de joc clasice și mecanici de joc tradiționale.

O altă aplicație originală Bomberman este „Super Bomberman R”, care este disponibil pentru Nintendo Switch. Acesta oferă un gameplay modernizat, cu grafică 3D și opțiuni de joc multiplayer.

În afara aplicațiilor originale, există și alte jocuri similare care au fost inspirate sau au fost dezvoltate în jurul conceptului de Bomberman, cum ar fi „Play with Fire” sau „Block Fortress War”. Acestea au mecanici de joc similare, dar pot avea caracteristici unice cum ar fi grafica sau modalitățile de joc diferite.

Aplicația realizată respectă mecanica principală de joc și astfel se pot plasa bombe care explodează pentru a elibera spațiu și a elimina inamici. Spațiile eliberate pot să conțină power-up-uri, iar toți inamicii sunt alte avatare cu aceleași caracteristici și abilități ca cel controlat de utilizator. Există un singur tip arenă de joc, dar se poate modifica dimensiunea acesteia și numărul de cutii prezente în aceasta care pot fi eliberate. Inamicii sunt controlați de alți utilizatori în Multiplayer, sau de un agent inteligent în Singleplayer.

1.3. Structura lucrării

În capitolul 1 este prezentată tema aleasă și sursa de inspirație, un studiu comparativ ale acestora și metoda diferită de implementare a aplicației.

În capitolul 2 sunt prezentate tehnologiile folosite în vederea realizării aplicației și atingerii obiectivelor propuse. Tehnologiile precizate sunt Limbajul Java, sistemul de operare Android, mediul de dezvoltare Android studio, platforma Firebase și programul open-source GIMP.

În capitolul 3 sunt descrise și explicate majoritatea funcționalităților aplicației și modul în care acestea au fost implementate.

În capitolul 4 este prezentat modul de folosire al aplicației, fiind explicate metodele prin care se creează jocuri în cele două moduri diferite.

În capitolul 5 sunt precizate concluziile precum și posibilitatea de dezvoltare a aplicației în viitor.



1.4. Enumerarea părților originale

Crearea unui joc online tradițional implică utilizarea unui server central pentru a gestiona logica jocului și comunicarea între jucători. Jucătorii se conectează la server prin intermediul unei aplicații sau prin browser, iar acțiunile lor sunt trimise către server, care le procesează și le distribuie celorlalți jucători conectați. Serverul poate fi programat utilizând diferite limbaje de programare, cum ar fi C++, Java sau Python.

Acest model de joc necesită un server dedicat pentru a gestiona numărul mare de conexiuni simultan, precum și resursele necesare pentru a procesa logica jocului. De asemenea, acesta necesită un sistem de autentificare pentru a se asigura că doar jucătorii autorizați se pot conecta la joc.

Spre deosebire de modul acesta de a crea jocuri online, aplicația realizată nu folosește un așa-zis server în modul Multiplayer, ci o bază de date actualizată și citită în timp real, iar acțiunile sunt gestionate de fiecare dispozitiv și transmise către baza de date. S-a folosit o bază de date Firebase Realtime Database pentru a administra informațiile și controlul acestora de către utilizatori.

În modul Singleplayer, inamicii caută pătrate care nu au explozie dacă sunt aproape de acestea, iar dacă sunt în siguranță, ei caută puncte de interes pentru a elibera spațiu folosind bombe sau pentru a elimina inamici.

2. Tehnologii folosite

2.1. Limbajul Java

Java este un limbaj de programare orientat pe obiecte, creat de James Gosling și dezvoltat de Sun Microsystems în 1995. Este utilizat pentru a dezvolta aplicații software pentru diverse platforme, cum ar fi Windows, MacOS, și sisteme de operare mobile, precum Android. Java este popular pentru aplicațiile de afaceri, jocuri și aplicații web, datorită posibilității sale de a rula pe orice sistem prin intermediul aplicației Java Virtual Machine (JVM). Java este, de asemenea, utilizat în inteligența artificială și înțelegerea automată, prin intermediul bibliotecilor și framework-urilor cum ar fi Weka, Mallet și TensorFlow.

Principiile limbajului Java sunt:

- **Simplitate** - Java a fost proiectat să fie un limbaj simplu de învățat și de utilizat, cu un set restrâns de cuvinte-cheie și o sintaxă clară.
- **Orientare pe obiecte** - Java este un limbaj orientat pe obiecte, care permite programatorilor să definească clase și obiecte care încapsulează comportamentul și starea unui program.
- **Independența de platformă** - Java este compilat într-un cod intermediar numit „bytecode”, care poate fi executat de orice sistem care are o mașină virtuală Java (JVM) instalată. Aceasta permite ca Java să fie utilizat pe o varietate de platforme, fără a fi nevoie de recompilare.



- **Securitate** - Java a fost proiectat cu securitatea în minte, cu caracteristici precum verificarea tipurilor la runtime, gestionarea memoriei automată și restricțiile de acces la fișiere și alte resurse.
- **Scalabilitate** - Java permite programatorilor să dezvolte aplicații care pot fi extinse și optimizate pentru a gestiona volum mare de date și trafic de utilizatori.
- **Flexibilitate** - Java oferă o gamă largă de biblioteci și framework-uri care permit programatorilor să dezvolte aplicații pentru diverse scopuri, cum ar fi aplicații web, jocuri, inteligența artificială și înțelegerea automată.

Java folosește un colector de gunoi automat pentru a gestiona memoria în ciclul de viață al obiectului. Programatorul determină când sunt create obiectele, iar runtime-ul Java este responsabil pentru recuperarea memoriei uneori obiectele nu mai sunt utilizate. Odată ce nu mai există referințe la un obiect, memoria inaccesibilă devine eligibilă pentru a fi eliberată automat de către colectorul de gunoi. Ceva similar cu o pierdere de memorie poate încă să apară dacă codul programatorului ține o referință la un obiect care nu mai este necesar, în mod obișnuit atunci când obiectele care nu mai sunt necesare sunt stocate în containere care sunt încă în utilizare. Dacă sunt apelate metode pentru un obiect inexistent, o excepție de pointer nul este aruncată. Una dintre ideile din spatele modelului de gestiune automată a memoriei Java este că programatorii pot fi scutiți de sarcina de a efectua gestiune manuală a memoriei.

Colectarea gunoiului poate avea loc oricând. Ideal, aceasta va avea loc atunci când un program este inactiv. Este garantat să fie declanșat dacă nu există suficientă memorie liberă pe heap pentru a alocă un nou obiect; acest lucru poate face ca un program să se blocheze momentan. Gestiunea explicită a memoriei nu este posibilă.

Java nu suportă aritmetică de pointer de tip C/C++, unde adresele de obiect pot fi manipulate aritmetic (de exemplu, prin adăugarea sau scăderea unui offset). Acest lucru permite colectorului de gunoi să relocheze obiectele referențiate și asigură siguranța tipurilor și securitatea. La fel ca în C++ și alte limbi orientate pe obiecte, variabilele tipurilor de date primitive Java sunt stocate direct în câmpuri (pentru obiecte) sau pe stivă (pentru metode), în loc să fie stocate pe heap, așa cum este adevărat în mod obișnuit pentru tipurile de date non-primitive. Aceasta a fost o decizie conștientă a designerilor Java din motive de performanță.

Rezolvând problema gestiunii memoriei nu eliberează programatorul de sarcina de a gestiona în mod corespunzător alte tipuri de resurse, cum ar fi conexiunile la rețea sau baza de date, gestionarea de fișiere, etc., mai ales în prezența excepțiilor.

Sintaxa limbajului Java este similară cu a altor limbaje de programare orientate pe obiecte, cum ar fi C++ sau C#. Ea include:

- Clase care conțin atributele și metodele unui obiect. Obiectele pot fi create din clase și pot fi utilizate pentru a apela metodele și a accesa atributele acestora.
- Variabile de mai multe tipuri de date, cum ar fi int, double, boolean, și String. Variabilele pot fi declarate cu aceste tipuri de date și pot fi utilizate pentru a stoca valori.
- Structuri de control cum ar fi if-else, for, și while, care permit programatorilor să controleze fluxul de execuție al unui program.
- Metode care pot fi apelate de alte părți ale programului. Metodele pot avea argumente și pot returna valori.
- Organizarea claselor în pachet pentru a oferi accesul controlat la clase și pentru a preveni conflictele de nume.



- Un mecanism de gestionare a excepțiilor pentru a gestiona erorile care pot apărea în timpul execuției unui program. Programatorii pot utiliza structuri cum ar fi try-catch pentru a gestiona excepțiile.
- Modificatorii de acces cum ar fi public, private, protected, care permit controlul accesului la clase, metode și attribute.
- Definirea de interfețe, care specifică metodele care trebuie să fie implementate de o clasă.
- Posibilitatea claselor să moștenească attribute și metode de la alte clase, prin intermediul mecanismului de moștenire.

Acestea sunt doar câteva dintre caracteristicile sintaxei limbajului Java. Există multe altele, cum ar fi tipurile generice, modificatorii de static și final, și șirurile de caractere formate.

Mașina Virtuală Java (JVM) este un software care permite rularea codului compilat Java într-o formă intermediară numit bytecode, indiferent de platforma hardware sau sistemul de operare. JVM interpretează acest bytecode, alocă memorie și gestionează alte resurse pentru a permite codului Java să ruleze în mod eficient.

JVM este responsabilă pentru gestionarea memoriei în timpul executării codului Java. Acest lucru se realizează prin intermediul unui colector de gunoi care alocă și eliberează automat memoria pentru obiectele create în cod. Aceasta înlătură responsabilitatea programatorului de a gestiona manual memoria, prevenind astfel pierderile de memorie și alte erori comune asociate cu gestiunea manuală a memoriei.

JVM și Java Class Libraries (JCL) formează împreună Platforma Java. JCL este o colecție de clase și interfețe care oferă funcționalitate de bază pentru aplicațiile Java, cum ar fi gestionarea fișierelor, conexiunile la rețea, gestionarea bazelor de date și multe altele.

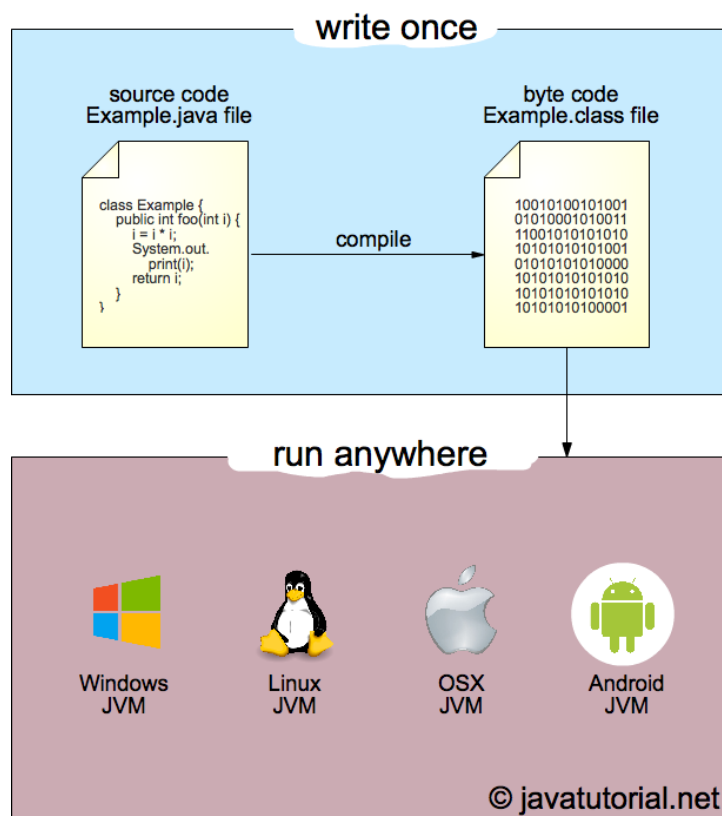


Figura 2.1: Mașina virtuală Java



2.2. Platforma Android

Android este o platformă software pentru dispozitive mobile, bazată pe sistemul de operare Linux. A fost dezvoltată inițial de către compania Android Inc., care a fost achiziționată de către Google în 2005. Începând cu 2008, Google a lansat prima versiune a sistemului de operare Android, iar acum este utilizat pe miliarde de dispozitive în întreaga lume, cum ar fi telefoane inteligente, tablete sau televizoare inteligente.

SDK sau Software Development Kit, este un set de instrumente și biblioteci care permit dezvoltatorilor să scrie codul pentru aplicații. SDK-ul include tot ce este necesar pentru a dezvolta aplicații, cum ar fi documentație, instrumente de compilare, instrumente de depanare și biblioteci. SDK-urile sunt disponibile pentru o varietate de platforme software, cum ar fi Android, iOS, Windows sau MacOS.

În cazul platformei Android, SDK-ul include instrumente precum Android Studio, care este un mediu de dezvoltare integrat pentru Android, și un instrument de depanare utilizat pentru a conecta dispozitivele Android la calculatorul dezvoltatorului. SDK-ul include, de asemenea, biblioteci Java și alte resurse care permit dezvoltatorilor să acceseze funcționalități ale sistemului de operare Android, cum ar fi GPS, camera, sau conexiunea la internet.

Java este limbajul de programare cel mai utilizat pentru a scrie aplicații pentru platforma Android. Acest lucru se datorează faptului că atât sistemul de operare Android, cât și SDK-ul sunt bazate pe Java.

Bibliotecile de interfață utilizator (UI) sunt o parte esențială a platformei Android și sunt utilizate pentru a crea interfețe grafice atractive și intuitiv pentru utilizatori. Acestea oferă dezvoltatorilor un set de instrumente și componente pentru a crea elemente de interfață precum butoane, căsuțe de text, liste, slide-uri și multe altele.

Accesul la magazinul de aplicații este esențial pentru utilizatorii dispozitivelor Android, deoarece acest lucru le permite să descarce și să instaleze aplicații. Magazinul de aplicații oficial pentru Android este Google Play Store, care este disponibil pe majoritatea dispozitivelor cu sistem de operare Android. Utilizatorii pot accesa Google Play Store prin intermediul aplicației preinstalate sau prin accesarea site-ului web.

2.3. Android Studio

Android Studio este un mediu de dezvoltare integrat (IDE) creat de Google pentru dezvoltarea aplicațiilor pentru platforma Android. Acest software gratuit și open-source oferă o serie de instrumente și caracteristici pentru dezvoltatorii de aplicații Android, cum ar fi un editor de cod puternic, instrumente de depanare, un designer de interfață grafică, un emulator și suport pentru biblioteci și instrumente diferite. Acestea permit dezvoltatorilor să scrie, să editeze, să testeze codul lor și să creeze interfețe utilizator atractive și intuitive pentru utilizatorii finali.

Android Studio este mediul de dezvoltare ales în realizarea aplicației.



Figura 2.2: Android Studio Logo



2.4. GIMP

GIMP (GNU Image Manipulation Program) este un program gratuit și open-source pentru editarea și manipularea de imagini. Acesta este un instrument puternic pentru editarea de imagini care poate fi utilizat pentru a realiza o varietate de sarcini, cum ar fi retușarea fotografiilor, crearea de grafice pentru web, ilustrații și altele.

Are o interfață utilizator similară cu alte programe de editare de imagini profesionale precum Adobe Photoshop, dar este disponibil gratuit și poate fi utilizat atât pe sisteme de operare Windows, MacOS, cât și Linux.

Oferă o serie de instrumente pentru editarea de bază, cum ar fi tăiere, selectare, ștergere și altele, precum și instrumente avansate precum filtre de imagine, efecte de distorsiune, stratificare și altele.

De asemenea, GIMP include suport pentru multiple formate de imagine, cum ar fi JPEG, PNG, TIFF, BMP și altele, precum și suport pentru plugin-uri care pot fi utilizate pentru a adăuga funcționalitate suplimentară.

Este un program popular utilizat de fotografi, artiști și graficieni care doresc să editeze și să manipuleze imagini și este considerat ca fiind una dintre cele mai bune alternative gratuite pentru Adobe Photoshop.

A fost folosit pentru a crea imaginile folosite în aplicație.



Figura 2.3: GIMP Logo

2.5. Firebase

Firebase este o platformă de dezvoltare mobilă și web dezvoltat de Google. Acesta oferă o serie de servicii cloud-based pentru a ajuta dezvoltatorii să construiască și să ruleze aplicații web și mobile cu mai puțin efort.

Una dintre caracteristicile principale ale Firebase este baza de date real-time. Aceasta permite dezvoltatorilor să creeze aplicații care pot accesa și actualiza date în timp real, fără a fi nevoie să gestioneze propriile servere.

Firebase oferă suport pentru o varietate de funcționalități care pot fi utilizate pentru a ajuta la dezvoltarea și întreținerea aplicațiilor web și mobile, cum ar fi:

- O varietate de opțiuni de autentificare, cum ar fi autentificarea prin intermediul conturilor de utilizator populare, cum ar fi Google, Facebook, Twitter, sau prin e-mail și parolă.
- Stocarea de fișiere, cum ar fi imagini și videoclipuri, care pot fi accesate de către utilizatorii aplicației.



- Trimiterea de notificări către utilizatorii aplicației, atât pe dispozitivele mobile, cât și pe web.
- Stocarea de date, cum ar fi informații despre utilizatori sau setări ale aplicației.
- Analiza utilizării aplicației, cum ar fi numărul de utilizatori activi.

Firestore oferă două tipuri de baze de date:

Baza de date Realtime - este o bază de date NoSQL care permite stocarea și sincronizarea datelor în timp real. Aceasta poate fi accesată prin intermediul unei API REST sau prin intermediul unui SDK pentru diferite limbaje de programare, cum ar fi Java, JavaScript sau Swift. Aceasta poate fi utilizată pentru a crea aplicații cu funcționalități de colaborare în timp real, cum ar fi jocuri sau aplicații de chat.

Firestore: Aceasta este o bază de date document-orientată care permite stocarea și sincronizarea datelor în timp real. Aceasta poate fi accesată similar cu baza de date Realtime. Este o bază de date orientată pe document, care este mai ușor de utilizat pentru aplicații care au nevoie de un model de date mai flexibil.

Ambele baze de date sunt oferite ca servicii cloud-based, fără a fi nevoie să se configureze sau să se gestioneze propriul server de baze de date.

Există câteva diferențe cheie între baza de date Real-time și Firestore:

- Firestore este mai bună în ceea ce privește scalabilitatea, deoarece poate gestiona mai multe cereri de citire și scriere în același timp.
- Firestore este de obicei mai rapid decât Realtime Database, deoarece permite întrebări mai sofisticate și indexarea automată a documentelor.
- Firestore este mai scump decât Realtime Database în ceea ce privește utilizarea în producție.
- Ambele servicii oferă suport pentru utilizarea offline, dar Realtime Database oferă o experiență mai bună în acest sens, deoarece permite sincronizarea datelor în mod mai eficient.

În realizarea aplicației a fost folosită o bază de date Realtime datorită faptului că s-a dorit o structură a datelor simplă și rapidă.

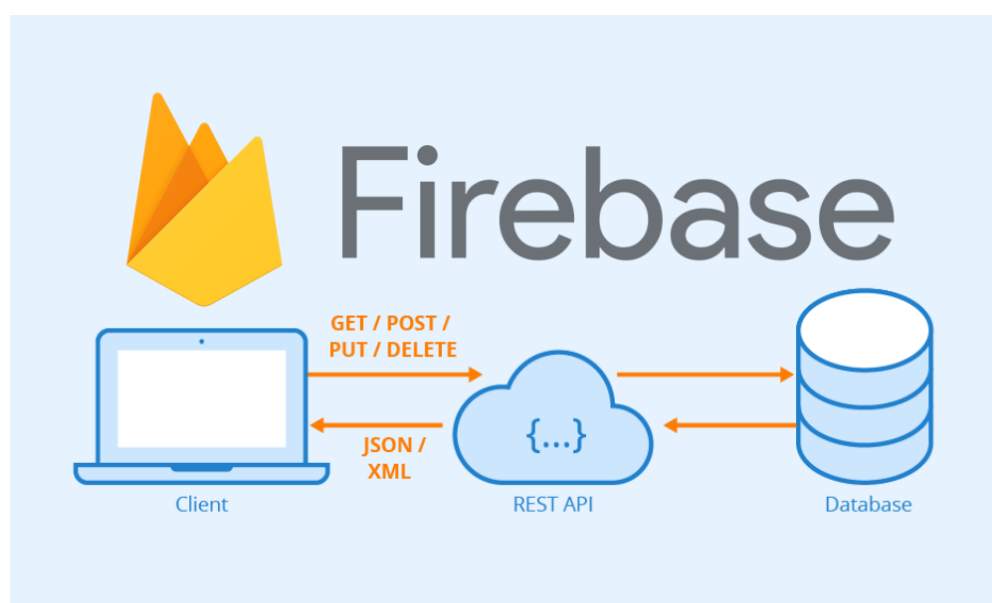


Figura 2.4: Conexiune Firebase



3. Descrierea aplicației

3.1. Bucla de joc și Clasa Game

Bucla de joc este mecanismul prin care progresează jocul. Această secvență rulează încontinuu atât timp cât jocul încă se desfășoară. În clasa Game, se interceptează comenzile date de utilizator, se gestionează majoritatea obiectelor jocului, unde se creează, de unde se actualizează obiectele prin apelarea funcției obiectului respectiv de update, și în final, se desenează starea actuală pe ecran. Bucla va folosi apeluri la metode din clasa Game pentru a gestiona secvențele menționate anterior de un număr precis pe secundă.

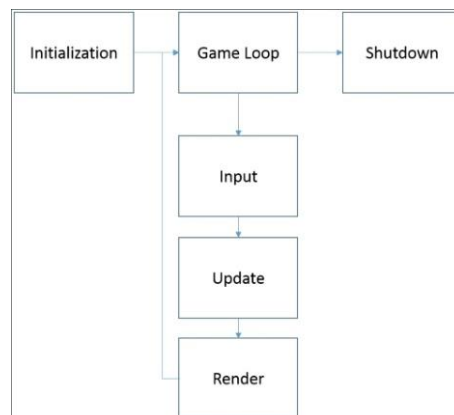


Figura 3.1: Structura jocurilor

Clasa GameLoop extinde clasa Thread, astfel sincronizând actualizările obiectelor cu desenarea lor în fiecare cadru. Se declară constanta MAX_UPS ca fiind 30.0, ea fiind cea care va limita numărul de actualizări ale obiectelor în fiecare ciclu. Funcția startLoop este apelată la crearea suprafeței de desinare, în care se setează variabila isRunning ca fiind true și apelând funcția start pentru a porni firul de execuție, iar funcția stopLoop este apelată se pune pauză la bucla de joc, unde se setează isRunning ca fiind false. Se suprascrie funcția run din clasa Thread. Apoi se declară variabile locale pentru timp și numărul de cicluri și se creează bucla de joc care va rula atât timp cât variabila isRunning mai sus expusă este true. În blocul try se încearcă actualizarea obiectelor din game prin apelarea funcției update, să se deseneze pe ecran cu ajutorul canvas, și se numără de câte ori au fost actualizate obiectele, se prinde orice excepție, și în final, se verifică dacă variabila canvas a fost alocată în try, și dacă da, se încearcă să se trimită pixeli desenați în canvas mai departe către desinare și se numără de câte ori s-a desenat pe ecran.

```
while (isRunning) {
    try {
        canvas = surfaceHolder.lockCanvas();
        synchronized (surfaceHolder) {
            game.update();
            updateCount++;

            game.draw(canvas);
        }
    } catch (IllegalArgumentException e) {...} finally {
        if (canvas != null) {
            try {
                surfaceHolder.unlockCanvasAndPost(canvas);
                frameCount++;
            } catch (Exception e) {...}
        }
    }
}
```

Figura 3.2: Bucla de Joc



După aceea se calculează cât a durat acest ciclu, și în cazul în care a durat mai puțin decât s-a dorit, se așteaptă timpul necesar pentru a trece la următorul ciclu în perioadă constantă de timp de fiecare dată. În cazul în care a durat mai mult acest ciclu, se vor realiza mai multe actualizări fără a mai și desena în canvas, astfel având un număr constant de actualizări pe secundă ale obiectelor. În final, se actualizează variabilele `averageUPS` și `averageFPS`, care vor reține numărul mediu de actualizări pe secundă și respectiv numărul mediu de cadre desenate pe secundă, acestea putând fi folosite pentru a monitoriza performanțele jocului.

Clasa `Game` este cea în care se vor gestiona majoritatea interacțiunilor dintre obiectele de joc. Ea este o clasă abstractă deoarece va fi extinsă în clasele `SingleplayerGame` și `MultiplayerGame`, care au comportament diferit pentru cele două moduri de joc, și însăși clasa `Game` este extinsă din clasa `SurfaceView` și implementează interfața `SurfaceHolder.Callback`.

În constructorul clasei, se ia `surfaceHolder`-ul din superclasa părinte `SurfaceView` și i se adaugă drept interfață callback `game`-ul, se iau dimensiunile ecranului ca pixeli ce vor fi folosiți pentru a găsi dimensiunile pe care trebuie să le aibă obiectele desenate pe ecran, se inițializează bucla de joc, se inițializează `spriteSheet`-ul, el fiind clasa în care se despart toate imaginile individuale din `png`-ul cu texturile obiectelor de pe ecran, se inițializează mapa în funcție de parametri primiți ca parametri în constructor, ei fiind lățimea, lungimea și șansa de a genera o cutie în orice spațiu liber, se inițializează joystickul care preia direcția și viteza cu care se va deplasa jucătorul și se inițializează și butonul care va pune o bombă pe ecran la apăsarea sa de către utilizator, la inițializarea acestora două se trimite ca parametru poziția unde să fie desenate pe ecran și ce dimensiuni să aibă, ele fiind calculate ca fiind un raport în funcție de dimensiunea ecranului memorat anterior.

Se suprascrive metoda `onTouchEvent` pentru a putea trimite apăsările ecranului de către utilizator și a fi prelucrate în mișcările avatarului său. Mai întâi se ia codul acțiunii evenimentului realizat, prin realizarea operației de „și” pe biți cu acțiunea și constanta `MotionEvent.ACTION_MASK`, umând să se extragă numărul degetului asociat evenimentului, ceea ce se realizează prin o altă operație „și” pe biți între acțiune și constanta `MotionEvent.ACTION_POINTER_INDEX_MASK`, iar rezultatul va fi deplasat la dreapta cu constanta `MotionEvent.ACTION_POINTER_INDEX_SHIFT`. Cu aceste calcule, acum se pot folosi indexul și codul acțiunii să se afle poziția atingerii ecranului și alte informații necesare gestionării multiplelor atingeri ale ecranului.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    int action = event.getAction();
    int actionCode = action & MotionEvent.ACTION_MASK;
    int pointerIdIndex = (action & MotionEvent.ACTION_POINTER_INDEX_MASK)
        >> MotionEvent.ACTION_POINTER_INDEX_SHIFT;
    int pointerId = event.getPointerId(pointerIdIndex);
    float x = event.getX(pointerIdIndex);
    float y = event.getY(pointerIdIndex);
    int pointerCount = event.getPointerCount();
```

Figura 3.3: Identificarea atingerilor ecranului



Deoarece utilizatorul va atinge 2 puncte pe ecran în același timp, trebuie gestionate ambele acțiuni. Se va realiza acest fapt prin memorarea indexului când este apăsat butonul și când joystickul. Se verifică dacă este apăsat fiecare prin apelul funcției `isPressed(x, y)` implementate în ambele clase pentru buton și joystick, ceea ce va fi realizat la evenimentul de apăsare cu cel puțin un deget și la mișcarea lor. La acțiunea de ridicare a cel puțin unui deget, la anularea acțiunii sau în afara interfeței, atunci se resetează indecșii, direcția joystickului și informația de apăsare a obiectelor.

```
switch (actionCode) {
    case MotionEvent.ACTION_DOWN:
    case MotionEvent.ACTION_POINTER_DOWN:
    case MotionEvent.ACTION_MOVE:
        if (joystick.isPressed(x, y)) {
            joystickPointerId = pointerId;
            joystick.setIsPressed(true);
        }
        if (button.isPressed(x, y)) {
            buttonPointerId = pointerId;
            button.setIsPressed(true);
        }
        break;
    case MotionEvent.ACTION_UP:
    case MotionEvent.ACTION_POINTER_UP:
    case MotionEvent.ACTION_CANCEL:
    case MotionEvent.ACTION_OUTSIDE:
        if (joystickPointerId == pointerId) {
            joystick.setIsPressed(false);
            joystick.resetActuator();
            joystickPointerId = -1;
        } else if (buttonPointerId == pointerId) {
            button.setIsPressed(false);
            buttonPointerId = -1;
        }
        break;
}
```

Figura 3.4: Gestionarea atingerilor ecranului

În final, se verifică pentru fiecare atingere dacă este indexul degetului de pe joystick pentru a muta cercul care ne arată direcția în acesta.

Alte metode importante care se suprascriu sunt cele de `draw`, `update` și `surfaceCreated`.

În `surfaceCreated` verifică dacă `gameLoop`-ul a fost terminat, caz în care se adaugă din nou `callback` la `surfaceHolder` instanța actuală a jocului și se creează un nou `gameLoop`, urmat de pornirea buclei de joc.

În `draw` se apelează funcțiile de `draw` ale tuturor obiectelor desenate pe ecran precum mapa de joc, joystickul și butonul, și avatarul utilizatorului atât timp cât el încă mai are viață.

În `update` se actualizează obiectele jocului: joystickul, butonul și listele cu bombe și explozii. Acestea sunt 2 liste care memorează toate bombele și exploziile



pentru a putea fi parcurse rapid și a fi actualizate. După parcurgerea fiecărei liste și actualizarea obiectelor, se obține o listă de bombe și una de explozii care în urma actualizării trebuiesc șterse. În update se apelează și funcția pentru a verifica finalul jocului.

```
public void update() {
    if (playerCountChanged) {
        handleGameEnded();
    }

    List<Bomb> bombRemoveList = new ArrayList<>();
    for (int idx = 0; idx < bombList.size(); idx++) {
        bombList.get(idx).update(bombRemoveList);
    }
    bombList.removeAll(bombRemoveList);

    List<Explosion> explosionRemoveList = new ArrayList<>();
    for (int idx = 0; idx < explosionList.size(); idx++) {
        explosionList.get(idx).update(explosionRemoveList);
    }
    explosionList.removeAll(explosionRemoveList);

    joystick.update();
    button.update();
}
```

Figura 3.5: Metoda update din clasa Game

3.2. Generarea mapei

Clasa responsabilă pentru crearea tabelii cu fiecare tip de pătrat din mapa de joc este MapLayout. Mapa este formată din mai multe tipuri de pătrate: perete din cărămizi care nu pot fi sparte cu explozii, cutie¹ care poate fi spartă și drum. Pătratul cu drum este singurul care poate fi străbătut de către un avatar.

În constructorul acesteia sunt parametri care denotă lățimea și lungimea mapei și aproximarea numărului de cutii pe hartă. În continuare se inițializează un tablou bidimensional cu tipul de date int în care se vor memora tipurile de pătrate din mapa astfel: 1 este drum, 2 este perete și 3 este cutie.

Două bucle for sunt folosite pentru a crea o margine a mapei cu pereți pentru a restricționa spațiul de joc. Prima buclă parcurge lungimea mapei și atribuie pătratele de sus și de jos ca fiind pereți și apoi în a doua parcurge lățimea în spațiile rămase. Urmează două bucle for care parcurg pe rând fiecare pătrat. Dacă acesta are ca și coordonate pe linie și coloană numere pare, atunci acel spațiu va fi alt perete pentru a crea coridoare multiple de mers. Dacă nu sunt pare, se generează un număr de tip float și este comparat cu șansa de a pune o cutie, dacă numărul generat este mai mic decât șansa, atunci pătratul va fi o cutie. În final, dacă nu a fost pus un perete sau o

¹ Cutie – un obiect pe mapă care poate fi spart cu o bombă explodând lângă aceasta și are șansa ca în locul ei să apară un obiect care oferă avantaj pe parcursul jocului celui care trece peste el



cutie, atunci se pune drum. După ce s-a completat întreaga tabelă, se ia fiecare colț din interiorul marginii de pereți și se pune ca fiind drum el și un pătrat lateral (cu excepția marginii). Acest lucru se realizează ca toți jucătorii să aibă un spațiu de unde să înceapă jocul unde pot să se deplaseze și să pună bombe fără ca ei să fie răniți.

După ce se generează tabloul bidimensional de înt-uri, va fi convertit într-o matrice de obiecte Tile.

3.3. Elementele grafice

De la tipul de pătrate ale mapei, trebuie afișate pe ecran. Pentru aceasta s-a realizat un spritesheet.

3.3.1.Sprite și Spritesheet





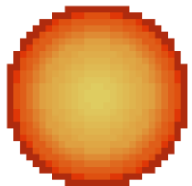

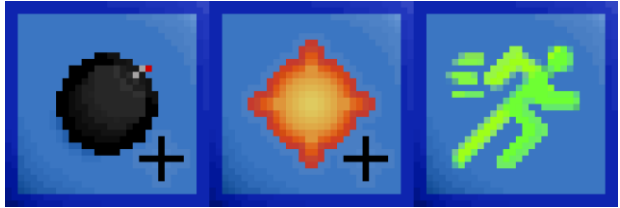
Un spritesheet este o imagine compusă dintr-un set de imagini mici care pot fi utilizate în programe de animație sau jocuri video pentru a crea o animație fluentă. Spritesheet-urile sunt de obicei utilizate pentru a economisi spațiu și pentru a îmbunătăți performanța, deoarece încărcarea unei singure imagini mai mari poate fi mai rapidă decât încărcarea mai multor imagini mai mici individual. Un spritesheet poate conține imagini pentru diferite personaje sau elemente din joc, cum ar fi monștri, personaje jucător, obstacole sau medii. Fiecare imagine din spritesheet poate conține una sau mai multe frame-uri de animație, care pot fi afișate într-o secvență rapidă pentru a crea o animație fluidă. Spritesheets pot fi create folosind diferite programe de design grafic, cum ar fi Adobe Photoshop sau GIMP. Spritesheet-ul realizat este format din 25 de sprite-uri² aranjate câte 5 pe linie și 5 pe coloana, fiecare având 32 pe 32 pixeli



Figura 3.6: Spritesheet-ul aplicației

² Sprite - imagine mică care poate fi afișată pe un ecran de calculator sau pe un dispozitiv mobil. În general, sprite-urile sunt utilizate în programe de animație și jocuri video pentru a crea personaje, obiecte și fundaluri.



	Drum(Walk) – pe acest pătrat se pot deplasa caracterele, pot să pună bombe și rămân cu explozie deasupra pentru un timp după declașarea bombeii.
	Perete(Wall) – pătrat care nu poate fi distrus de explozii și nu poate fi străbătut, funcționează ca o barieră
	Cutie(Crate) – similar cu peretele, dar aceasta poate fi distrusă de explozii; există o probabilitate ca în urma sa să rămână un obiect care le crește abilitățile jucătorilor
	Bombă – este pus deasupra drumurilor de jucători, după 2.5 secunde de la apariție explodează și distruge cutii
	Explozie – sunt lăsate pe drumurile care au fost prinse în raza bombeii după detonare și dispar după 0.8 secunde de la apariție, dacă un caracter merge pe acest spațiu, acesta pierde o viață
	Avatar – controlat de utilizatori sau de agentul inteligent bazat pe decizii luate în funcție de reguli, prima imagine este starea staționară, iar la deplasare se alternează cele 3 imagini pentru a crea iluzia de mișcare; există 4 caractere diferite pentru a fi diferențiate în timpul jocului
	Power ups – pătrate care dacă sunt străbătute de un caracter i se îmbunătățesc abilitățile; primul îi crește numărul de bombe care pot fi puse de el pe ecran, al doilea crește raza de acțiune a bombeii, ultimul îi crește viteza maximă

Prelucrarea imaginii de baza are loc în clasa SpriteSheet unde se încarcă din resurse colecția într-un bitmap³. Se află coordonatele unui sprite dorit prin înmulțirea liniei și coloanei primite ca parametri cu dimensiunea sa respectivă în bitmap. S-au

³ Bitmap - o imagine stocată într-un format de fișier care conține informații despre fiecare pixel individual din imagine



creat funcții care apelează metoda anterioară pentru a obține obiecte sprite fiecărei imagini în afară de avatare. Avatarele sunt formate din vectori de sprite-uri, fiecare vector are atâtea elemente câte imagini are fiecare avatar, astfel se formează patru tablouri de trei sprite-uri.

Clasa Sprite memorează spriteSheet-ul și pătratul unde se află elementul în bitmap. Clasa oferă trei funcții draw cu care desenează pe ecran imaginea. Primesc ca parametru canvasul și pătratul unde sunt desenate în acesta, iar opțional, un obiect paint și o variabilă float care reprezintă unghiul de rotație cu care pătratul să fie desenat. Metodele folosesc apeluri la funcția clasei Canvas drawbitmap pentru a desena porțiunea din bitmap. În cazul în care se dorește folosirea unghiului, se va roti canvasul cu acesta în punctul reprezentând centrul patratului unde se va desena, se apelează funcția drawBitmap și se revine la poziția inițială a canvasului.

```
public void draw(Canvas canvas, Rect displayRect) {
    draw(canvas, displayRect, paint: null);
}

public void draw(Canvas canvas, Rect displayRect, Paint paint) {
    canvas.drawBitmap(spriteSheet.getBitmap(), spriteRect, displayRect, paint);
}

public void draw(Canvas canvas, Rect displayRect, float rotationAngle, Paint paint) {
    canvas.save();
    canvas.rotate(rotationAngle, displayRect.exactCenterX(), displayRect.exactCenterY());
    canvas.drawBitmap(spriteSheet.getBitmap(), spriteRect, displayRect, paint);
    canvas.restore();
}
```

Figura 3.7: Metodele draw din clasa Sprite

3.3.2. Tile

Este o clasă abstractă care este implemetată de toate pătratele care produc mapa de joc. Conține definiția unei enumerații LayoutType care expune fiecare tip de obiecte care formează harta. Există o constantă layoutType în fiecare tile pentru a putea fi ușor de recunoscut subclasa la apelurile metodelor din clasa lor părinte și mai există și un obiect din clasa Rect⁴ care memorează coordonatele și dimensiunea tile-lului în imaginea desenată.

Metoda statică getTile din această clasă primește ca parametri un index de tip int care este convertit la tipurile pătratelor din enumerație, spriteSheet-ul și pătratul unde se află în mapă și returnează un nou obiect care este unul din fiecare tip de pătrat care formează harta. Obiectele fac parte din clasele care extind clasa Tile și acestea sunt:

- WalkTile, WallTile și CrateTile care reprezintă pătratele de drum, perete și respectiv cutie. Aceste trei clase au o implementare asemănătoare între ele, memorează sprite-ul lor obținut din spriteSheet-ul primit ca parametru în constructor și acesta este desenat la apelarea funcției suprascrise de draw în care se folosește funcția de draw a obiectului sprite;

⁴ Rect – Clasă provenită din pachetul android.graphics pentru gestionarea dreptunghiurilor



- SpeedPowerUpTile, BombCountPowerUpTile și ExplosionRangePowerUpTile care reprezintă pătratele care conțin posibilitatea de a îmbunătăți caracterele prin creșterea vitezei, creșterea numărului de bombe pe care le poate folosi și respectiv creșterea razei de acțiune a exploziei bombei. Aceste trei clase au o implementare asemănătoare între ele, spre deosebire de cele trei tipuri expuse anterior, acestea memorează și sprite-ul drum, iar la metoda suprascrisă de draw se desenează sprite-ul drum mai întâi, urmat de apelul metodei draw pentru sprite-ul power up-ului său, dar desenarea acestuia se realizează cu un obiect paint care micșorează opacitatea sa. Se folosește această metodă pentru a se vedea o porțiune din pătratul de drum pentru ca jucătorul să realizeze că acel pătrat poate fi străbătut.
- BombTile și ExplosionTile care reprezintă pătratele de drum dar au o bombă sau explozie în ele. Implementarea este asemănătoare cu cele 3 anterioare prin faptul că folosesc două sprite-uri în realizarea imaginii finale din mapă prin desenarea pătratului de drum urmat de cel de bombă, respectiv explozie. Datorită desenării a celor două în această ordine, cel de al doilea sprite este desenat deasupra acoperind o porțiune din imaginea drumului, ceea ce arată utilizatorului că pătratul are deasupra sa acele obiecte. Spre deosebire de tile-urile anterioare, aici memorăm și obiectul bomb respectiv explosion, acestea fiind obiectele logice din pătrate.

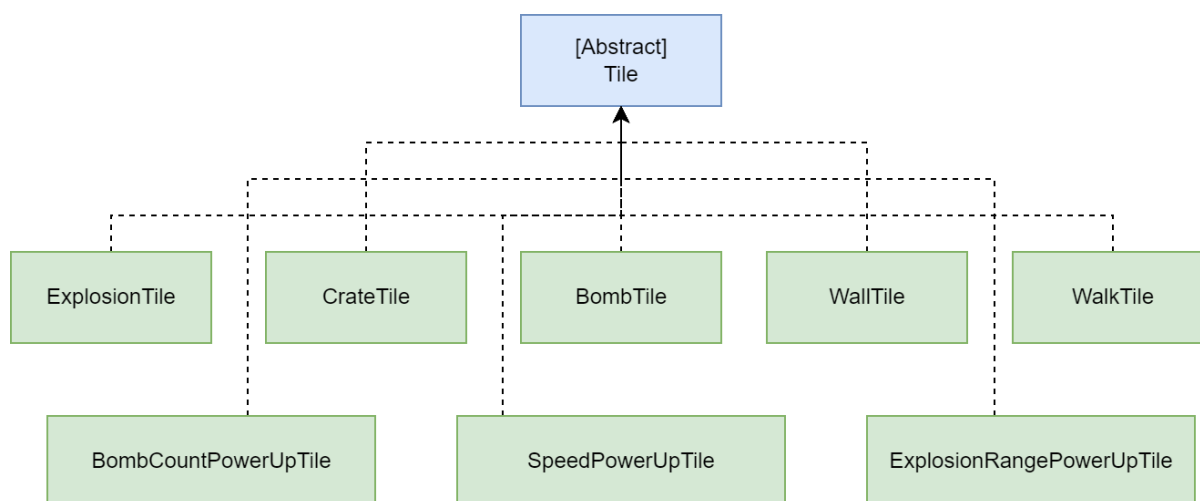


Figura 3.8: Diagrama moștenirilor clasei Tile

3.3.3. Tilemap

Este clasa responsabilă de crearea imaginii de afișat pe ecran reprezentând jocul. Clasa conține un tablou bidimensional de tile-uri care este mapa de joc și un bitmap care este de fapt imaginea desenată.

Pentru a desena pe ecran mapa jocului, se calculează ce dimensiune trebuie să aibă pe ecran fiecare pătrat în parte, astfel se împarte înălțimea ecranului la numărul de linii ale hărții, iar lățimea se împarte la numărul de coloane. Se ia un procentaj din lățime pentru a lăsa o margine laterală când se desenează. Dintre aceste două dimensiuni calculate pentru înălțime și lățime se alege cea cu valoare mai mică, astfel mapa finală o să aibă o dimensiune optimă fiind vizibilă în totalitate.



```
private void setSpriteSizeOnScreen() {
    int height = Utils.screenHeight / numberOfRowTiles;
    int width = (int) (Utils.screenWidth * 0.9 / numberOfColumnTiles);

    spriteSizeOnScreen = Math.min(height, width);
}
```

Figura 3.9: Determinarea dimensiunii unui sprite pe ecran

În constructor avem acest calcul al dimensiunii sprite-urilor, inițializăm un mapLayout care ne oferă baza hărții, creăm un obiect mapRect al clasei Rect folosind coordonatele x și y ale centrului de ecran și jumătăți din lungimea și lățimea mapei pentru a determina poziția mapei în mijlocul ecranului în înălțime și în lățime. După determinarea acestei poziții memorăm în clasa ajutoare Utils decalajele laterale ale mapei.

```
int mapHeightOffset = (spriteSizeOnScreen * numberOfRowTiles) >> 1;
int mapWidthOffset = (spriteSizeOnScreen * numberOfColumnTiles) >> 1;
int screenCenterX = Utils.getScreenCenterX();
int screenCenterY = Utils.getScreenCenterY();
mapRect = new Rect(
    left: screenCenterX - mapWidthOffset,
    top: screenCenterY - mapHeightOffset,
    right: screenCenterX + mapWidthOffset,
    bottom: screenCenterY + mapHeightOffset);
Utils.mapOffsetX = mapRect.left;
Utils.mapOffsetY = mapRect.top;
```

Figura 3.10: Determinarea poziției mapei de joc pe ecran

În final, folosind mapLayoutul, se creează matricea tilemap cu aceleași dimensiuni ca mapLayout. Luând pe rând fiecare element al matricei, se alocă în acestea un nou tile creat prin funcția getTile din clasa Tile, metodă care returnează unul din cele opt tipuri diferite de obiecte moștenite din Tile în prin valoarea venită din mapLayout. După crearea tuturor elementelor tilemap-ului, se creează un bitmap de dimensiunea mapei noastre. Se folosește un canvas care are ca suprafață de desenare acest bitmap pentru a fi completat, parcurgând fiecare element din tilemap și apelând metoda de draw al tile-ului în canvasul anterior.

Pentru a modifica mapa după crearea inițială, se apelează funcția changeTile care după crearea unui nou tile în locul specificat, va redesena acel pătrat în bitmap folosind canvasul.

Funcția de draw a clasei este apelată în partea de draw clasei Game. Prin apelarea acesteia, se desenează bitmap-ul hărții existent. Deoarece nu este nevoie de o desenare în permanență al fiecărui tile, putem avea doar un bitmap care reprezintă toată harta în schimb, iar acest bitmap este modificat doar când este nevoie, cum ar fi la folosirea unei bombe de jucător, fapt care ajută la minimalizarea resurselor folosite ale dispozitivului pe care rulează.



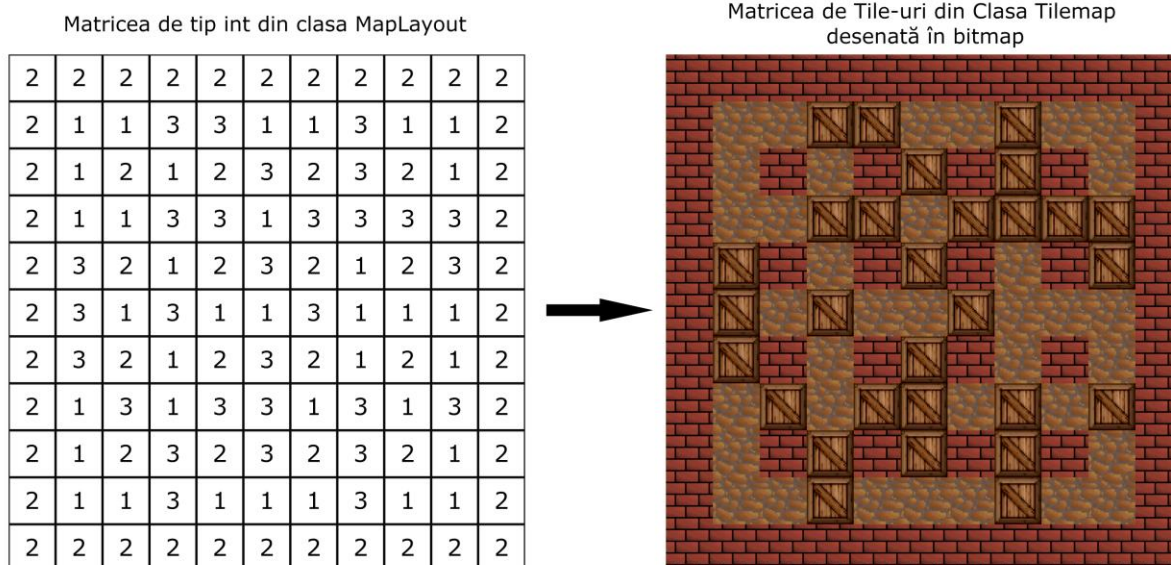


Figura 3.11: Procesul de creare a mapei de joc

3.4. Clasa Bomb și clasa Explosion

Sunt cele două clase care gestionează evoluția jocului. Bomb este obiectul logic care este lăsat de un caracter pentru a elimina adversari sau a sparge cutii, iar Explosion este obiectul rămas după ce o bombă a explodat în urma sa.

Clasa Bomb are drept câmpuri o constantă statică BOMB_TILE_LAYOUT_ID care memorează indexul din enumerarea tipurilor de pătrate din clasa Tile pentru a putea fi convertit la apelul funcției changeTile din clasa Tilemap, identificatorul caracterului cărui îi aparține bomba, raza de acțiune, linia și coloana tile-ului unde a fost plasată în mapa de joc, lista explosionList folosită în clasa Game pentru a actualiza exploziile și tilemap-ul.

În constructorul acestei clase se primește raza de acțiune a bombei care diferă în funcție de caracter, linia și coloana, identificatorul avatarului, lista cu explozii și tilemap-ul. Acesta este constructorul folosit pentru a adăuga o nouă bombă în tilemap la poziția specificată în parametrii. În acesta se inițializează câmpul clasei updatesBeforeExplosion care ia valoarea constantei MAX_UPS înmulțit cu 2.5, astfel rezultatul obținut reprezintă numărul de actualizări ale jocului până la declanșarea bombei, rezultând că acesta durează 2.5 secunde. În final, se schimbă prin funcția changeTile, tile-ul aflat anterior la acea poziție cu un BombTile, și se setează în acel tile bomba fiind obiectul actual prin folosirea cuvântului cheie this.

Funcția de update oferită de clasa Bomb are ca parametru o listă cu obiecte Bomb, listă în care se vor adăuga obiectele care trebuie eliminate în urma actualizării. Prima acțiune realizată este de a reduce valoarea variabilei updatesBeforeExplosion cu unu, reprezentând înaintarea în cicluri de joc drept timp. În cazul în care variabila a devenit mai mică decât unu, se declanșează bomba. Când o bombă explodează, se va răspândi sub formă de plus în cele patru direcții, având epicentrul unde fusese amplasată și se va schimba tipul tile-ului din bombTile într-un explosionTile. Pe fiecare direcție, explozia se răspândește cu distanța maximă range, care reprezintă raza de acțiune a bombei salvată în câmpul obiectului. În timpul răspândirii exploziei, se verifică ce tile este în calea lui. Dacă se întâlnește:



- tile de tip perete, atunci răspândirea, în continuare, pe acea direcție este oprită
- tile de cutie, atunci în locul cutiei se va pune tile de explozie și se adaugă explozia din el în câmpul explosionList, iar răspândirea exploziei se oprește după aceasta
- pătrate de drum sau power ups, atunci în locul lor se pun tile-uri de explozie și se adaugă explozia din el în câmpul explosionList, iar răspândirea exploziei se continuă, astfel power up-urile dispar dacă au fost atinse de explozie
- tile de explozie, se resetează numărul până la dispariția exploziei din acel pătrat la valoarea inițială de la crearea acesteia
- pătrat cu bombă, atunci se va apela funcția de a declanșa și acea bombă, astfel se pot crea reacții în lanț a tuturor bombelor care sunt în rază una față de cealaltă și care au cale liberă între ele

```
private void explodeLength(List<Bomb> bombRemoveList, int idxRow, int idxColumn) {
    for (int idx = 1; idx < range; idx++) {
        switch (tilemap.getTilemap()[row + idxRow * idx][column + idxColumn * idx].getLayoutType()) {
            case BOMB:
                ((BombTile) tilemap.getTilemap()[row + idxRow * idx][column + idxColumn * idx])
                    .getBomb().triggerExplosion(bombRemoveList);
                break;
            case EXPLOSION:
                ((ExplosionTile) tilemap.getTilemap()[row + idxRow * idx][column + idxColumn * idx])
                    .getExplosion().setUpdatesBeforeDisappear(Explosion.EXPLOSION_DURATION);
                break;
            case CRATE:
                explosionList.add(new Explosion( row: row + idxRow * idx,
                    column: column + idxColumn * idx, tilemap));
                return;
            case WALK:
            case BOMB_POWER_UP:
            case RANGE_POWER_UP:
            case SPEED_POWER_UP:
                explosionList.add(new Explosion( row: row + idxRow * idx,
                    column: column + idxColumn * idx, tilemap));
                break;
            case WALL:
            default:
                return;
        }
    }
}
```

Figura 3.12: Metoda explodeLength din clasa Bomb

Clasa Explosion are drept câmpuri o constantă statică EXPLOSION_TILE_LAYOUT_ID care memorează indexul din enumerarea tipurilor de pătrate din clasa Tile pentru a putea fi convertit la apelul funcției changeTile din clasa Tilemap, linia și coloana tile-ului unde se află în mapa de joc și tilemap-ul.

Constructorul are ca parametrii linia și coloana poziției tile-ului în mapă, lista tuturor exploziilor parcurse pentru a actualiza elementele și tilemap-ul. Acest constructor gestionează și dacă în urma unei explozii care a avut loc pe o fostă cutie, va apărea un Power up. Variabila tileBehindExplosion va memora tipul de tile care va rămâne în urma unei explozii, iar inițial valoarea ei va fi



MapLayout.WALK_TILE_LAYOUT_ID reprezentând indexul unui pătrat drum. Variabila `updatesBeforeDisappear` se ia valoarea constantei `MAX_UPS` înmulțit cu 0.8, astfel valoarea obținută reprezintă numărul de actualizări ale jocului până la dispariția exploziei, rezultând că acesta durează 0.8 secunde. Urmează o condiție pentru a identifica dacă tipul de pătrat care se află în acest moment în mapa de joc este cutie, caz în care se generează aleator un număr de tip float, dacă acesta este mai mic ca 0.3f, înseamnă ca acel pătrat o sa fie un power up. Care dintre cele trei disponibile se va folosi este tot generat aleator prin apelul funcției `getPowerUpBehindExplosion` și i se va aloca variabilei `tileBehindExplosion`. Se ia un număr de tip int având minimul zero și maximul un număr care este divizibil cu trei, pe acesta se folosește operatorul modulo cu trei, rezultatul fiind de la zero la doi inclusiv, și se adună șase, pentru ca numărul generat să fie între șase și opt, care sunt indecșii tile-urilor de power ups în enumerația tipurilor de tile. După determinarea pătratului ce va rămâne în urma exploziei actuale, se apelează funcția de `changeTile` pe `tilemap` pentru a schimba pătratul într-un obiect `ExplosionTile`, iar în final, în noul tile, se setează explozia actuală ca fiind cea din tile prin folosirea cuvântului cheie `this` în clasă.

```
public Explosion(int row, int column, Tilemap tilemap) {
    this.row = row;
    this.column = column;
    this.tilemap = tilemap;

    tileBehindExplosion = MapLayout.WALK_TILE_LAYOUT_ID;

    updatesBeforeDisappear = EXPLOSION_DURATION;

    if (tilemap.getTilemap()[row][column].getLayoutType() == Tile.LayoutType.CRATE) {
        if (Utils.generator.nextFloat() < .3) {
            tileBehindExplosion = getPowerUpBehindExplosion();
        }
    }
    tilemap.changeTile(row, column, EXPLOSION_TILE_LAYOUT_ID);

    ((ExplosionTile) tilemap.getTilemap()[row][column]).setExplosion(this);
}

private int getPowerUpBehindExplosion() {
    return Utils.generator.nextInt( bound: 90) % 3 + 6;
}
```

Figura 3.13: Constructorul clasei *Explosion*

Funcția de update a clasei `Explosion` primește ca parametru o listă de explozii care vor fi eliminate în funcția de update a clasei `Game`. Prima etapă este de a reduce o unitate din variabila `updatesBeforeDisappear` pentru a reduce durata pe ecran a exploziei, iar dacă în urma reducerii variabila este mai mică decât unu, înseamnă ca nu mai este nevoie de tile-ul de explozie. Astfel schimbăm pătratul folosind funcția de `tilemap.changeTile` având ca parametrii linia și coloana actualei explozii și variabila `tileBehindExplosion` care are memorat ori id-ul pentru un `WalkTile`, sau, în cazul în care a fost cutie și în urma ei a lăsat un power up, indexul pentru aceasta. În final, se adaugă obiectul `Explosion` actual la lista de explozii eliminate.



3.5. Clasa Player

Este o clasă abstractă care implementează principalele acțiuni realizate de toate caracterele din joc, fie caracterul jucătorului ori inamicul.

Are o multitudine de câmpuri pentru ca acesta să poată interacționa cu majoritatea obiectelor de joc. Reține tilemap-ul, id-ul caracterului său, coordonate x și y pentru poziționarea în mapă, direcția de deplasare, viteza de deplasare, un obiect Rect pentru a fi mai ușor de analizat interacțiunea cu obiectele, un obiect din clasa Animator, folosit pentru a anima caracterul în timpul deplasării împreună cu obiectul PlayerState și variabila rotationAngle care determină cu câte grade trebuie întors caracterul în timp ce se deplasează, lista de bombe și lista de explozii, un obiect StatsBar care arată deasupra avatarului câte vieți mai are, și variabile caracteristice avatarului, cum sunt numărul de vieți rămase, câte bombe poate să folosească în același timp pe mapă și ce rază au aceste bombe, informații transmise când folosește o bombă în constructorul clasei Bomb și câte power up-uri de viteză a adunat, care îi vor influența viteza maximă finală.

Constructorul primește ca parametri tilemap-ul, lista de bombe și lista de explozii, numărul liniei și numărul coloanei de început a caracterului care vor fi folosite pentru a determina obiectul Rect al clasei, animatorul său, numărul de bombe, raza lor, câte puteri de viteză are și numărul de vieți.

În acesta se determină variabila defaultMaxSpeed care reprezintă viteza de deplasare fără speedUp⁵-uri prin apelarea metodei getPlayerDefaultMaxSpeed implementată în clasa ajutoare Utils. Viteza reprezintă numărul de pixeli străbătuți pe ciclu de joc de caracter, calculat folosind de două ori variabila spriteSizeOnScreen împărțit la constanta MAX_UPS din clasa GameLoop, adică de două ori numărul de pixeli a unui pătrat împărțit la numărul de actualizări pe secundă reprezentând o viteză de două pătrate pe secundă. Variabila playerRect care reține hitbox⁶-ul caracterului se creează prin găsirea tile-lului din hartă care se află la linia și coloana primite ca parametrii, din care se citește obiectul Rect al acestuia care îi reține poziția în mapă, de unde se creează un nou obiect Rect prin folosirea constructorului clasei care primește ca parametru un alt Rect pentru a fi copiat. Se creează un obiect Paint numit invincibilityPaint căruia i se setează opacitatea cu funcția setAlpha la o valoare relativ mică, acest obiect fiind folosit pentru a desena caracterul oferindu-i o vizibilitate redusă, stadiu în care intră după ce este atins de explozie și primește o perioadă scurtă de invincibilitate în care nu poate să i se mai scadă numărul de vieți. Se creează și o listă care conține toate tipurile de power-up uri pentru a fi iterate mai târziu. Se creează un obiect PlayerState care este folosit pentru urmărirea stării motorii a caracterului și un obiect StatusBar.

Clasa PlayerState este cea care determină în ce stare de deplasare se află caracterul în funcție de viteza acestuia. Conține definiția enumerației State care are trei elemente: NOT_MOVING, STARTED_MOVING și IS_MOVING, un obiect de tipul State și player de care aparține. În constructor, valoarea obiectului state se setează ca fiind NOT_MOVING. Conține o metodă update care este apelată în funcția de update a claselor care moștenesc clasa Player. În acesta se află un bloc switch pe variabila state a clasei și tratează diferitele cazuri ale enumerației:

⁵ SpeedUp – power up de viteză

⁶ Hitbox – zona de unde începe interacțiunea cu alte obiecte de joc



- dacă valoarea este NOT_MOVING, se verifică printr-o condiție dacă viteza caracterului fie pe coordonata x fie pe coordonata y este diferită de zero, atunci variabila state se va schimba să fie de tipul STARTED_MOVING
- dacă valoarea este STARTED_MOVING, se verifică prin condiție dacă viteza în continuare pe coordonate este diferită de zero, caz în care state se va schimba în IS_MOVING
- dacă valoarea este IS_MOVING, se verifică prin condiție dacă vitezele pe coordonate sunt amândouă zero, însemnând că avatarul s-a oprit, caz în care valoarea variabilei state se schimba în NOT_MOVING din nou

```

public void update() {
    switch (state) {
        case NOT_MOVING:
            if (player.getVelocityX() != 0 || player.getVelocityY() != 0) {
                state = State.STARTED_MOVING;
            }
            break;
        case STARTED_MOVING:
            if (player.getVelocityX() != 0 || player.getVelocityY() != 0) {
                state = State.IS_MOVING;
            }
            break;
        case IS_MOVING:
            if (player.getVelocityX() == 0 && player.getVelocityY() == 0) {
                state = State.NOT_MOVING;
            }
            break;
        default:
            break;
    }
}

```

Figura 3.14: Metoda update din clasa PlayerState

Clasa Animator este cea care creează animația caracterelor prin alterarea diferitelor imagini ale avatarului. Conține o constantă MAX_UPDATES_BEFORE_NEXT_MOVE_FRAME care este determinată din constanta MAX_UPS din GameLoop, determinând numărul de actualizări ale buclei de joc până când trebuie să schimbe imaginea din animație timp de 0.1 secunde. Reține și o listă de sprite-uri care conține sprite-urile caracterului. Variabila updatesBeforeNextMoveFrame este cea care ia inițial valoarea constantei care denotă câte cadre durează pe ecran imaginea până la schimbarea ei. În constructor primește numai tabloul de imagini ale caracterului.

Funcția de draw primește ca parametri canvasul în care desenează, player-ul căreia îi aparține și un obiect Paint. Din obiectul player se extrage Rect-ul acestuia de pe mapă și este folosit pentru a crea un nou obiect, urmând ca el să fie deplasat cu valorile mapOffsetX și mapOffsetY deoarece pătratul din player este cel cu valori în funcție de mapa de joc, dar acesta nefiind desenată din colțul din dreapta sus ci în mijloc, și mapa are un decalaj. Prin mutarea pătratului cu același decalaj pe care îl are harta, desenul se va suprapune cu poziția sa pe ecran. Urmează un corp switch pe valoarea variabilei State din playerState-ul caracterului.

- Dacă este stare NOT_MOVING, atunci se va apela funcția de draw pe sprite-ul din lista cu imaginile caracterului la poziția reprezentând imaginea statică a acestuia.



- Dacă este stare `STARTED_MOVING`, atunci variabila `updatesBeforeNextMoveFrame` care numără cadrele rămase cât apare imaginea curentă pe ecran își ia valoarea inițială dată de `MAX_UPDATES_BEFORE_NEXT_MOVE_FRAME`, apoi este apelată metoda de `draw` pe sprite-ul din lista cu imaginile caracterului la poziția `idxMovingFrame`, câmp din clasa `Animator` care are valoarea inițială unu.
- Dacă este starea `IS_MOVING`, atunci mai întâi se scade o unitate din variabila `updatesBeforeNextMoveFrame`, iar dacă pe urmă este mai mică ca unu, înseamnă că trebuie schimbată imaginea desenată pentru avatar. În acest caz, se resetează valoarea variabilei care numără cadrele până la următoarea schimbare de cadre și se apelează funcția `toggleIdxMovingFrame` a clasei. Există trei diferite imagini pentru fiecare caracter care pot fi desenate pe ecran, prima din listă este cea folosită și pentru cazul în care avatarul stă pe loc, dar ea este folosită și ca o trecere între celelalte două imagini, dintre care una este imaginea pentru pasul stâng și cealaltă pentru pasul drept. În aceasta metodă există un bloc `switch` pe variabila `idxMovingFrame`, astfel, dacă ea are valoarea zero, atunci se verifică dacă câmpul de tip boolean al clasei `oddTimeAction` are valoarea de adevăr, caz în care i se va schimba în valoarea `false` și `idxMovingFrame` ia valoarea doi, iar în caz contrar, `oddTimeAction` se va schimba în valoarea `true`, iar `idxMovingFrame` ia valoarea unu. `Switch`-ul mai are două cazuri, dacă are valoarea unu sau doi, cazuri în care `idxMovingFrame` ia valoarea zero, fiind imaginea de trecere între celelalte două imagini.

În fiecare caz, apelul funcției de `draw` din clasa `sprite` este cea care primește ca parametri în plus față de `canvas`ul și `pătratul` unde să fie desenat, mai primește și unghiul de rotație și obiectul `paint`. Rotația fiind determinată de direcția de mers a caracterului și de rotația sa în `spriteSheet`, iar `paint`-ul este cel care are opacitatea scăzută în cazul perioadei de invincibilitate sau `null` în mod normal.

```
public void draw(Canvas canvas, Player player, Paint paint) {
    Rect drawnRect = new Rect(player.getPlayerRect());
    drawnRect.offset(Utils.mapOffsetX, Utils.mapOffsetY);

    switch (player.getPlayerState().getState()) {
        case NOT_MOVING:
            playerSpriteArray[ID_NOT_MOVING_FRAME].draw(canvas, drawnRect,
                player.getRotationAngle(), paint);
            break;
        case STARTED_MOVING:
            updatesBeforeNextMoveFrame = MAX_UPDATES_BEFORE_NEXT_MOVE_FRAME;
            playerSpriteArray[idxMovingFrame].draw(canvas, drawnRect,
                player.getRotationAngle(), paint);
            break;
        case IS_MOVING:
            updatesBeforeNextMoveFrame--;
            if (updatesBeforeNextMoveFrame < 1) {
                updatesBeforeNextMoveFrame = MAX_UPDATES_BEFORE_NEXT_MOVE_FRAME;
                toggleIdxMovingFrame();
            }
            playerSpriteArray[idxMovingFrame].draw(canvas, drawnRect,
                player.getRotationAngle(), paint);
            break;
        default:
            break;
    }
}
```

Figura 3.15: Metoda `draw` din clasa `Animator`



Clasa Player are funcția sa de update abstractă aceasta fiind implemetată în clasele care extind superclasa. În metoda de draw, se apelează funcția de draw a obiectului statsBar, urmat de cea a obiectului animator, obiectul Paint trimis la funcție este usedPaint, căreia i se schimbă valoare în funcție de situație.

Clasa StatsBar este cea care afișează deasupra tuturor caracterelor câte vieți mai au rămas. Ea are două constante care reprezintă dimensiunea imaginii în pixeli în înălțime și lățime, imaginea fiind o pictograma a unei inimi pentru a fi ușor de înțeles pentru utilizator. Mai conține un obiect Rect care reprezintă pătratul sursă al imaginii folosit pentru desenarea într-un bitmap, două obiecte Paint, unul pentru culoarea imaginii desenului final, iar celălalt pentru scrisul desenului și mai conține obiectul player de care aparține.

În constructor se inițializează srcRect care are dimensiunea imaginii pentru inima desenată, se creează borderPaint, culoarea cu care se va desena chenarul pentru informația cu numărul de vieți, se generează un bitmap care conține imaginea cu inimă pentru a fi ușor de desenat și se creează un Paint specific pentru a desena numărul de vieți. Această variabilă numită textPaint va avea culoare albă, cu scris bold, iar dimensiunea scrisului va avea dimensiunea pătratului caracterului înmulțit cu valoarea 0.66f, valoarea care în raport cu înălțimea avatarului va fi optimă pentru încărarea în spațiul ales.

Funcția clasei de draw va desena fiecare dintre obiectele conținute în desenul final. Mai întâi se desenează un chenar care va conține și inima și numărul de vieți pentru a avea o vizibilitate mai bună. I se va crea un obiect Rect numit borderRect care va fi chenarul poziționat în imaginea finală de joc. La crearea obiectului, se vor folosi dimensiunile pătratului caracterului, astfel se va crea cu același dimensiuni ca și avatarul, dar coordonata laturii inferioare a pătratului va fi cu jumătate din înălțimea pătratului de origine mai mică formând un dreptunghi. După ce i se creează forma, se apelează funcția offset de pe obiectul borderRect având ca parametri decalajul mapei față de ecran, iar în final, se va desena pătratul calculat cu culoarea determinată anterior folosind funcția drawRect de pe obiectul canvas venit ca paramentru. Urmează generearea pătratului unde va determina poziția imaginii cu inimă. Acesta are ca bază dimensiunile obiectului borderRect, doar că latura dreaptă se oprește la jumătatea lățimii pătratului de chenar, astfel se va desena imaginea în jumătate din chenar folosind metoda de drawBitmap. În final, se folosește funcția drawText a canvasului în spațiul rămas în chenar folosind obiectul Paint pentru text. Se va scrie numărul de vieți din obiectul Player de care aparține.

```
public void draw(Canvas canvas) {
    // Draw border
    Rect playerRect = player.getPlayerRect();
    Rect borderRect = new Rect(
        playerRect.left,
        playerRect.top,
        playerRect.right,
        playerRect.bottom - playerRect.height() / 2);
    borderRect.offset(Utils.mapOffsetX, dy, Utils.mapOffsetY - borderRect.height() / 2);
    canvas.drawRect(borderRect, borderPaint);

    // Draw heart
    Rect heartRect = new Rect(
        borderRect.left,
        borderRect.top,
        borderRect.right - borderRect.width() / 2,
        borderRect.bottom);
    canvas.drawBitmap(heartBitmap, srcRect, heartRect, paint);

    // Draw number
    canvas.drawText(Integer.toString(player.getLivesCount()),
        heartRect.right, heartRect.bottom, textPaint);
}
```

Figura 3.16: Metoda draw din clasa StatsBar



Clasa Player conține o multitudine de metode, acestea fiind folosite în subclasele acesteia în principal pentru actualizarea obiectului. Acestea sunt:

- **initRectInTiles** - Determină în ce pătrat din mapă se încadrează fiecare latură din pătratul caracterului. Astfel vor exista patru variabile care reprezintă în ce pătrat aparțin. Valorile se determină prin obținerea pozițiilor laturilor pătratului caracterului, i se aplică marja de calcul, și se împarte poziția în pixeli la dimensiunea unui pătrat, astfel se obțin pătratele. Marja de calcul numită safe reprezintă o zonă unde caracterul poate să suprapună pătratul său cu alte obiecte fără ca acestea să înregistreze interacțiunea, variabila fiind un raport dintre dimensiunea pătratelor mapei și este atribuită în constructor.
- **handleDeath** - Această metodă verifică dacă în momentul curent, caracterul suprapune pătratul său cu un tile care conține o explozie, astfel scăzându-i numărul de vieți. Mai întâi se verifică dacă variabila time este mai decât unu, reprezentând timpul de invincibilitate a caracterului după ce a atins o explozie. În cazul în care valoarea este mai mică decât unu, înseamnă că nu este invincibil în momentul actual, așa că se urmăresc spațiile atinse de el. Urmează să fie verificate dacă în oricare dintre cele patru pătrate ale hărții atinse de avatar există vreo explozie, caz în care se scade numărul de vieți al caracterului cu unu, se atribuie constanta INVINCIBILITY_TIME variabilei time pentru a număra perioada în care nu se va mai urmări atingerea de explozii, și se setează în variabila usedPaint obiectul invincibilityPaint folosit pentru a desena starea în care este invincibil. Pentru cazul în care timpul este mai mare ca zero, atunci se scade o unitate din variabila time, se aplică operatorul modulo pe time, iar dacă acea valoare este zero, obiectul usedPaint își ia valoarea null, sau valoarea invincibilityPaint dacă valoarea este doi.

```
protected void handleDeath() {
    if (time < 1) {
        if (tileIsLayoutType(bottom, left, Tile.LayoutType.EXPLOSION) ||
            tileIsLayoutType(bottom, right, Tile.LayoutType.EXPLOSION) ||
            tileIsLayoutType(top, left, Tile.LayoutType.EXPLOSION) ||
            tileIsLayoutType(top, right, Tile.LayoutType.EXPLOSION)) {
            livesCount--;
            time = INVINCIBILITY_TIME;
            usedPaint = invincibilityPaint;
        }
        return;
    }
    time--;
    int aux = time % 4;
    switch (aux) {
        case 0:
            usedPaint = null;
            break;
        case 2:
            usedPaint = invincibilityPaint;
            break;
    }
}
```

Figura 3.17: Metoda handleDeath din clasa Player



- `handlePowerUpCollision` – Această metodă parcurge lista `powerUpsLayoutTypes` care conține diferitele `PowerUp`-uri, iar pe fiecare interacție, verifică dacă oricare din cele patru colțuri ale `hitbox`-ului avatarului intersectează un pătrat al mapei care conține unul dintre obiectele din listă. În acest caz, se folosește metoda `usePowerUp` care primește ca parametru tipul de pătrat pentru a fi folosit. După apelarea funcției, se folosește metoda `changeTile` pentru a schimba `tile`-ul care a fost folosit într-un `tile` de drum.

```
protected void handlePowerUpCollision() {
    for (Tile.LayoutType layoutType : powerUpsLayoutTypes) {
        if (tileIsLayoutType(bottom, left, layoutType)) {
            usePowerUp(layoutType);
            tilemap.changeTile(bottom, left, MapLayout.WALK_TILE_LAYOUT_ID);
        } else if (tileIsLayoutType(bottom, right, layoutType)) {
            usePowerUp(layoutType);
            tilemap.changeTile(bottom, right, MapLayout.WALK_TILE_LAYOUT_ID);
        } else if (tileIsLayoutType(top, left, layoutType)) {
            usePowerUp(layoutType);
            tilemap.changeTile(top, left, MapLayout.WALK_TILE_LAYOUT_ID);
        } else if (tileIsLayoutType(top, right, layoutType)) {
            usePowerUp(layoutType);
            tilemap.changeTile(top, right, MapLayout.WALK_TILE_LAYOUT_ID);
        }
    }
}
```

Figura 3.18: Metoda `handlePowerUpCollision` din clasa `Player`

- `getMaxSpeed` - Această metodă returnează viteza maximă pe care o poate avea un caracter. Viteza este obținută prin înmulțirea variabilei `defaultMaxSpeed` având valoarea inițială cu unu adunat cu numărul de abilități de viteză avute și cu o constantă de tip `float` subunitar `INCREASE_IN_SPEED_BY_POWER_UP` reprezentând cu cât crește viteza inițială cu fiecare putere.

```
protected double getMaxSpeed() {
    return defaultMaxSpeed * (1 + INCREASE_IN_SPEED_BY_POWER_UP * speedUps);
}
```

Figura 3.19: Metoda `getMaxSpeed` din clasa `Player`

- `tileIsLayoutType` - Metodă apelată pentru a determina dacă la poziția primită este de tipul dat ca argument, returnând rezultatul verificării.
- `useBomb` - La apelul funcției se încearcă lăsarea unei bombe în pătratul aflat în centru avatarului. Se determină `tile`-ul respectiv prin apelarea metodelor `getPlayerRow` și `getPlayerColumn` din clasa ajutoare `Utils`. Acestea folosesc centru pătratului caracterului, iar valoarea aceea este împărțită la dimensiunea



pătratului pentru a determina linia și coloana. Se numără câte bombe are obiectul Player în tilemap în momentul actual. Acesta se realizează în metoda maxBombCountReached prin iterarea listei cu bombe și numărând câte din acestea au același playerId ca avatarul actual. Valoarea de adevăr obținută prin compararea numărului obținut cu numărul maxim de bombe pe care le poate folosi este returnată. Dacă valoarea venită este adevărată, se întrerupe execuția metodei useBomb pentru a limita numărul de bombe. Se determină poziția caracterului în hartă, iar dacă poziția respectivă este un tile de tip drum, atunci se va crea o nouă bombă și va fi adăugată în lista cu bombe.

```
protected void useBomb() {
    if (maxBombCountReached()) {
        return;
    }

    int rowIdx = Utils.getPlayerRow( p: this);
    int columnIdx = Utils.getPlayerColumn( p: this);

    if (tileIsLayoutType(rowIdx, columnIdx, Tile.LayoutType.WALK)) {
        bombList.add(new Bomb(bombRange, rowIdx, columnIdx, playerId, explosionList, tilemap));
    }
}
```

Figura 3.20: Metoda useBomb din clasa Player

- getOrientation - Calculează distanța parcursă în ultima actualizare a obiectului prin calcularea distanței dintre două puncte, acestea fiind punctul de origine și un punct format din valorile velocityX și velocityY. Se determină distanța parcursă pe x și y prin împărțirea vitezelor la distanța determinată anterior.

```
protected void getOrientation() {
    double distance = Utils.getDistanceBetweenPoints( p1x: 0, p1y: 0, velocityX, velocityY);
    directionX = velocityX / distance;
    directionY = velocityY / distance;
}
```

Figura 3.21: Metoda getOrientation din clasa Player

- getAngle - Metoda returnează o valoare care reprezintă unghiul cu cât se vor roti imaginile din obiectul animator în funcția de draw. Folosește metoda atan2 din clasa predefinită Math cu valorile calculate anterior pentru directionY și directionX pentru a determina unghiul în radiani format de acestea în planul cartezian. Apoi valoarea este transformată în grade folosind metoda toDegrees. În final, se scad nouăzeci de grade reprezentând rotația avută în imaginea de origine din spriteSheet. Deoarece direcțiile sunt obținute din cele două viteze, pentru o singură viteză și cealaltă zero, va exista o distanță și valoarea zero, iar valoarea obținută din atan2 va fi un multiplu de nouăzeci. Acesta este comportamentul dorit deoarece un avatar poate să se deplaseze doar într-una din cele patru direcții simultan.

```
protected int getAngle() {
    return (int) Math.toDegrees(Math.atan2(directionY, directionX)) - 90;
}
```

Figura 3.22: Metoda getAngle din clasa Player



- **movePlayer** - Această metodă modifică valorile câmpurilor `positionX` și `positionY` din clasă. Se schimbă prin adăugarea vitezelor pe cele două direcții. Aceste valori sunt folosite în mai multe funcții pentru a calcula deplasările avatarului. După determinarea lor, se va deplasa poziția obiectului `playerRect` la poziția determinată de valorile calculate anterior.
- **detectCollisions** - Această metodă este apelată în funcția de update a clasei după ce se obțin vitezele caracterului rezultate din mișcarile transmise de utilizator. Funcția va identifica dacă se poate deplasa în direcția dorită și va modifica vitezele în funcție de situații. Mai întâi se verifică dacă există viteză pe oricare dintre axa x sau y, iar dacă nu, se întrerupe execuția metodei, deoarece nu are ce coliziuni să verifice. Se continuă creând un nou obiect `Rect` numit `newRect` care este format cu obiectul `playerRect`, iar acesta este deplasat folosind metoda `offsetTo` la poziția unde ar trebui mutat caracterul în funcție de viteze și pozițiile anterioare. Urmează să se identifice în ce direcție se deplasează în funcție de care dintre viteze nu este zero și dacă este pozitivă sau negativă. În funcție de direcție, se apelează o metodă care calculează interacțiunea în acea direcție.

```
protected void detectCollisions() {
    if (velocityX == 0 && velocityY == 0) {
        return;
    }

    Rect newRect = new Rect(playerRect);
    newRect.offsetTo((int) (positionX + velocityX), (int) (positionY + velocityY));

    if (velocityX != 0) {
        if (velocityX < 0) {
            goesLeft(newRect);
        } else {
            goesRight(newRect);
        }
    } else if (velocityY < 0) {
        goesUp(newRect);
    } else {
        goesDown(newRect);
    }
}
```

Figura 3.23: Metoda `detectCollisions` din clasa `Player`

- **velocityChanging** - Metodă care primește ca parametru o linie și coloană din `tilemap`. Se urmărește dacă tipul de pătrat la acele coordonate este unul care ar influența viteza caracterului. Acestea sunt tile-urile de drum, cutie și bombă, în celelalte cazuri, viteza nu este influențată de acestea.



```
protected boolean velocityChanging(int row, int column) {
    switch (tilemap.getTilemap()[row][column].getLayoutType()) {
        case WALL:
        case CRATE:
        case BOMB:
            return true;
        default:
            return false;
    }
}
```

Figura 3.24: Metoda velocityChanging din clasa Player

- goesDown, goesUp, goesRight, goesLeft - Aceste funcții au aceleași principii de funcționare, ele primesc ca parametru obiectul clasei Rect numit newRect care este poziția unde va fi mutat caracterul în metoda de update în funcție de vitezele actuale din variabilele velocityX și velocityY. Ele sunt apelate pentru deplasarea într-una din cele patru direcții posibile și aleasă una dintre ele. Metodele analizează cu ce se va intersecta pătratul caracterului la actualizare și va modifica vitezele astfel ca el să rămână încadrat în drumurile din mapa de joc, și va ajuta utilizatorul pentru a lua viraje în mapă chiar dacă el nu se află la o poziție exactă optimă pentru viraj.
- La începutul oricăreia dintre ele se identifică indicii pătratelor din mapă care pot fi implicate în coliziune. Astfel, pentru deplasarea în sus sau în jos, se identifică care sunt indicii coloanelor în care se află latura stângă și latura dreaptă a obiectului newRect, iar linia este determinată în funcție de direcția de deplasare, astfel, pentru deplasarea în sus se calculează indicele liniei care conține latura de sus a pătratului caracterului, respectiv latura de jos pentru direcție descendentă. Se folosește același principiu pentru deplasarea în laterale, obținându-se doi indici pentru linii și un indice pentru coloana pe care se află newRect. Urmează apelarea metodei velocityChanging pentru a determina dacă se intersectează caracterul cu pătrate care nu pot fi străbătute de acesta. Se vor determina două valori de adevăr, pentru coliziunile laterale pentru deplasarea verticală sau coliziunile laturii de jos și sus pentru deplasarea pe orizontală. În funcție de valorile de adevăr obținute, există patru cazuri diferite:
 - Dacă ambele valori sunt false, asta înseamnă că nu trebuie modificate vitezele deoarece pătratul caracterului nu se intersectează cu nimic în următoarea actualizare.
 - Dacă ambele valori sunt adevărate, înseamnă că se va intersecta la ambele laturi, cea ce va produce oprirea sa. Dar va trebui să se oprească la limita dintre pătrate, din acest motiv se va căuta o valoare pentru viteză mai mică decât cea inițială ca atunci când se modifică poziția caracterului, acesta să nu producă o coliziune. Calculăm distanța până în pătratul cu care se produce interacțiunea, iar dacă aceea este mai mică decât viteza curentă, atunci o vom folosi pe aceea. Altfel, se folosește viteza inițială. Cazul în care distanța care trebuie străbătută este mai mică ca viteza inițială este aceea în care utilizatorul folosește o bombă. Bomba o să fie pusă pe pătratul pe care se află, iar caracterul trebuie să poată să se deplaseze deasupra acesteia, atât timp cât avatarul are bomba sub acesta, dar el nu poate să treacă peste aceasta dacă



s-a îndepărtat. Dacă nu ar fi folosită viteza cu modulul mai mic, viteza rezultată ar putea fi mai mare ca viteza maximă a caracterului.

- Ultimele două cazuri sunt cele în care valorile produse sunt una afirmativă și una negativă, iar acestea sunt luate independent. Pentru acest caz, se implementează o mecanică de joc, astfel, dacă jucătorul merge spre un obiect prin care nu poate să treacă și are în lateral aproape de poziția inițială un pătrat pe care poate să meargă, atunci în loc să stea pe loc deoarece nu are unde să se deplaseze în direcția în care se deplasează, el se va deplasa cu o viteză mai mică decât viteza inițială în direcția care îl aduce mai aproape de pătratul pe care poate să meargă. Această mecanică este utilă în timpul jocului pentru a crea o fluiditate de mișcare pentru utilizatori, fiind mai ușor să ia un viraj. Această mecanică este implementată schimbând viteza velocityX în viteza velocityY (sau viceversa) dar înmulțită cu constanta SPEED_MINIMIZING subunitară, astfel se produce o viteză pe cealaltă direcție mai mică decât cea inițială ca o penalitate pentru că nu s-a poziționat optim la viraj, iar viteza inițială va deveni zero. În continuare, se va calcula viteza finală similar ca pentru intersectarea cu două obiecte.

```
protected void goesUp(Rect newRect) {
    // Calculate the column and row indices of the given Rect
    int newRow = newRect.top / spriteSizeOnScreen;
    int newLeftColumn = newRect.left / spriteSizeOnScreen;
    int newRightColumn = (newRect.right - 1) / spriteSizeOnScreen;

    // Check if the left or right of the player's Rect is colliding with the given Rect
    boolean leftCollision = velocityChanging(newRow, newLeftColumn);
    boolean rightCollision = velocityChanging(newRow, newRightColumn);

    // Update velocity based on collision status
    if (leftCollision) {
        if (rightCollision) {
            // Both the left and right of the player's Rect are colliding with the given Rect
            // Calculate the minimum distance needed to avoid the collision
            int minVelocity = -playerRect.top % spriteSizeOnScreen;
            velocityY = Math.max(velocityY, minVelocity);
        } else {
            // Only left column is occupied by an obstacle, set velocityX and velocityY to 0
            velocityX = -velocityY * SPEED_MINIMIZING;
            velocityY = 0;
            int minVelocity = playerRect.left % spriteSizeOnScreen;
            velocityX = Math.min(velocityX, minVelocity);
        }
    } else if (rightCollision) {
        // Only right column is occupied by an obstacle, set velocityX and velocityY to 0
        velocityX = velocityY * SPEED_MINIMIZING;
        velocityY = 0;
        int minVelocity = -playerRect.right % spriteSizeOnScreen;
        velocityX = Math.max(velocityX, minVelocity);
    }
}
```

Figura 3.25: Metoda goesUp din clasa Player



Clasa Player se află la baza a patru subclase, acestea sunt cele pentru jucător și pentru inamici.

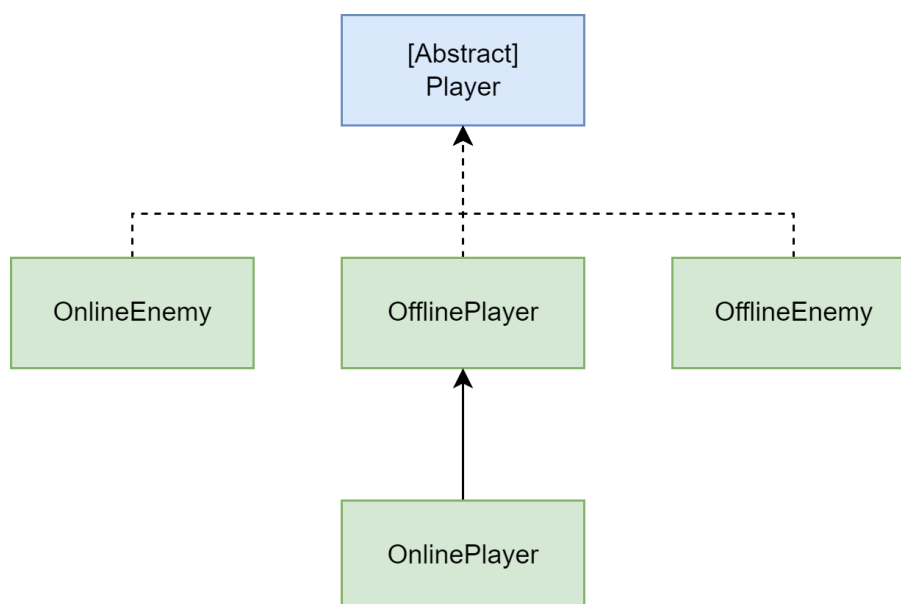


Figura 3.26: Diagrama moștenirilor clasei Player

3.5.1. OfflinePlayer

Este o clasă extinsă din clasa abstractă player. Această clasă este cea folosită în tipul de joc Singleplayer pentru utilizator. Spre deosebire de clasa de bază, există în plus obiectele Joystick și Button. Ele sunt cele care preiau comenzile de la utilizator și le transmit mai departe.

Clasa Joystick este un obiect care apare pe ecran și în funcție de unde este apăsător și ținut apăsător, transmite direcția selectată de deplasare a caracterului. Este format din două cercuri având aceeași origine dar dimensiuni și culori diferite.

În constructor se rețin pozițiile cercurilor și razele lor, și se inițializează două obiecte Paint având culori diferite, acestea fiind cele folosite pentru a desena obiectele pe ecran.

Metoda clasei de draw va desena cercurile pe ecran folosind metoda obiectului canvas primit ca parametru pentru desenarea de cercuri drawCircle, care primește ca parametri coordonatele punctului de origine, raza cercului și un obiect Paint. Mai întâi se desenează cercul mai mare, urmat de cercul mai mic, care va fi desenat deasupra.

Metoda de update a clasei este folosită pentru a schimba locația cercului interior mai mic vizibil la desenare. Acest lucru se realizează prin modificarea coordonatelor cercului interior. Se determină poziția începând de la centru cercului exterior care nu își modifică poziția, la care se adună valorile câmpurilor actuatorX respectiv actuatorY înmulțite cu raza cercului interior. Cele două câmpuri au valori subunitare, astfel se obține o poziție a cercului interior care are în cel mai îndepărtat caz originea pe lungimea cercului exterior.



```

public void update() { updateInnerCirclePosition(); }

private void updateInnerCirclePosition() {
    innerCircleCenterPosX = (int) (outerCircleCenterPosX + actuatorX * outerCircleRadius);
    innerCircleCenterPosY = (int) (outerCircleCenterPosY + actuatorY * outerCircleRadius);
}

```

Figura 3.27: Metoda update din clasa Joystick

Funcția isPressed primește ca parametru două valori double reprezentând coordonatele unui punct pe ecran. Se determină distanța de la punctul primit ca parametru și originea cercului exterior. Se returnează valoarea de adevăr dacă distanța este mai mică ca raza cercului exterior, reprezentând faptul că punctul este conținut în cercul respectiv sau nu.

```

public boolean isPressed(double touchPositionX, double touchPositionY) {
    double joystickCenterToTouchDistance = Utils.getDistanceBetweenPoints(
        outerCircleCenterPosX,
        outerCircleCenterPosY,
        touchPositionX,
        touchPositionY
    );

    return joystickCenterToTouchDistance < outerCircleRadius;
}

```

Figura 3.28: Metoda isPressed din clasa Joystick

Există o metodă resetActuator care setează valorile actuatorX și actuatorY la zero reprezentând poziția când nu există interacțiune cu utilizatorul, și metoda setActuator care prin parametrii primește coordonatele unui punct pe ecran în funcție de care să se calculeze valorile actuatorului. Se calculează deltaX și deltaY reprezentând diferența coordonatelor punctului primit ca parametru și cu punctul de origine a cercului exterior și se calculează deltaDistance care reprezintă distanța dintre cele două puncte. Dacă această distanță este mai mică decât raza cercului exterior, înseamnă că punctul primit ca argument, se află în interiorul razei cercului. Pentru acest caz, valorile actuatorX și actuatorY vor fi obținute din împărțirea valorilor delta la raza cercului exterior, astfel obținându-se valori subunitare pentru variabilele menționate. Pentru cazul în care distanța dintre puncte este mai mare ca raza cercului, atunci câmpurile sunt calculate împărțind delta-urile la deltaDistance. Datorită acestei abordări, dacă punctul primit la argument este în interiorul cercului, atunci valorile obținute adunate vor fi mai mici ca o unitate, dar în caz opus, acestea vor fi întotdeauna unu.



```

public void setActuator(double touchPositionX, double touchPositionY) {
    double deltaX = touchPositionX - outerCircleCenterPosX;
    double deltaY = touchPositionY - outerCircleCenterPosY;
    double deltaDistance = Utils.getDistanceBetweenPoints( p1x: 0, p1y: 0, deltaX, deltaY);

    if (deltaDistance < outerCircleRadius) {
        actuatorX = deltaX / outerCircleRadius;
        actuatorY = deltaY / outerCircleRadius;
    } else {
        actuatorX = deltaX / deltaDistance;
        actuatorY = deltaY / deltaDistance;
    }
}
}

```

Figura 3.29: Metoda setActuator din clasa Joystick

Clasa Button este un obiect care apare pe ecran și transmite dacă este apăsat pentru a fi folosit la punerea de bombe. Este format din-un cerc care are culori diferite dacă este apăsat sau nu.

În constructor se reține poziția cercului și raza sa, și se inițializează două obiecte Paint având culori diferite, acestea fiind cele folosite pentru a desena obiectul pe ecran în funcție de starea sa.

Metoda de update a clasei modifică culoarea obiectul din variabila usedPaint, ea devenind culoare diferită dacă este apăsat ecranul în interiorul cercului.

```

public void update() { updateColor(); }

private void updateColor() {
    usedPaint = isPressed ? pressedCirclePaint : circlePaint;
}

```

Figura 3.30: Metoda update din clasa Button

Conține o metodă isPressed care este implementată ca cea din clasa Joystick, care returnează valoarea de adevăr când un punct primit ca argument se află în interiorul cercului.

Metoda de draw desenează cercul în funcție de obiectul Paint care se află în variabila usedPaint folosind metoda drawCircle a obiectului canvas.

Aceste două obiecte au folosite metodele în clasa Game în metoda onTouchEvent pentru a gestiona evenimentele de atingere a ecranului în spațiul în care sunt desenate pe ecran, unde se setează câmpurile actuatoX și actuatoY pentru Joystick și câmpul isPressed pentru Button.

Clasa OfflinePlayer are o metodă selectDirectionFromActuator care setează viteza în funcție de actuatorul din clasa Joystick. Inițial, valorile vitezelor se setează ca zero, pentru ambele direcții. Pentru direcția care are actuatorul cu valoarea modulului mai mare, aceea este direcția căreia i se va schimba viteza, viteza fiind calculată prin înmulțirea vitezei maxime a caracterului provenite din metoda getMaxSpeed a clasei Player, cu valoarea actuatorului pentru direcția respectivă.



Deoarece valoarea actuatorului este subunitară, se poate obține o viteză care este un procentaj din viteza maximă a caracterului, iar dacă utilizatorul nu ține apăsat joystick-ul pe marginea cercului exterior, pentru un control precis al vitezei.

```
protected void selectDirectionFromActuator() {
    // Get the joystick orientation
    double actuatorX = joystick.getActuatorX();
    double actuatorY = joystick.getActuatorY();

    velocityX = 0;
    velocityY = 0;

    // Select direction by actuator value
    if (Math.abs(actuatorX) > Math.abs(actuatorY)) {
        velocityX = actuatorX * getMaxSpeed();
    } else {
        velocityY = actuatorY * getMaxSpeed();
    }
}
```

Figura 3.31: Metoda `selectDirectionFromActuator` din clasa `OfflinePlayer`

Clasa `OfflinePlayer` oferă implementarea metodei `update` care este declarată abstractă în superclasa `Player`. Aceasta gestionează toate acțiunile întreprinse de către obiectul `OfflinePlayer` folosind metode implementate în superclasă și clasa actuală. Prima acțiune verifică dacă obiectul are câmpul `livesCount` mai mic ca unu, însemnând că avatarul nu mai are vieți, caz în care se oprește execuția metodei, pentru a nu influența jocul dacă avatarul nu mai face parte din acesta. Se apelează metoda `initRectInTiles`, pentru a determina în care pătrate de mapă se află caracterul în această instanță. Se apelează metoda `selectDirectionFromActuator` pentru a lua viteza inițială a caracterului, iar apoi se apelează `detectCollisions` pentru a modifica viteza în cea finală în funcție de ce pătrate intersectează. Pentru cazul în care există o viteză nenulă, se apelează `movePlayer` pentru a deplasa player-ul cu viteza determinată anterior, metoda `getOrientation` pentru a obține direcția de deplasare folosită apoi în apelul metodei `getAngle` pentru a seta unghiul reprezentând rotirea în mapă a avatarului folosită în etapa de desen a obiectului. Urmează să se actualizeze obiectul `playerState` prin funcția `update` pentru a

```
@Override
public void update() {
    if (livesCount < 1) {
        return;
    }
    initRectInTiles();

    selectDirectionFromActuator();

    detectCollisions();

    if (velocityX != 0 || velocityY != 0) {
        movePlayer();

        getOrientation();
        // Update player orientation
        rotationAngle = getAngle();
    }

    // Update player state for animation
    playerState.update();

    // Use bomb
    if (button.getIsPressed()) {
        useBomb();
    }

    // Player death handler
    handleDeath();

    // Player picks power up handler
    handlePowerUpCollision();
}
```

Figura 3.32: Metoda `update` din clasa `OfflinePlayer`



schimba starea de deplasare pentru animație. Se verifică dacă este apăsat butonul pentru bombă, caz în care se apelează metoda de useBomb care folosește bombă dacă este posibil. Urmează să se apeleze metoda handleDeath pentru a gestiona obiectul pentru cazul în care a pierdut o viață și este în perioada caracterului de invincibilitate, iar în final, se apelează handlePowerUpCollision pentru a gestiona coliziunea cu tile-uri de PowerUp.

3.5.2. OnlinePlayer

Este o clasă care extinde OfflinePlayer și este folosită pentru jocurile de tip Multiplayer. Deoarece jucătorul trebuie să folosească obiectele de Joystick și Button folosite în clasa OfflinePlayer pentru a controla caracterul și în jocurile online, se extinde din aceasta în loc de clasa abstractă Player. Are ca deosebire de clasa OfflinePlayer faptul că schimbările caracteristicilor avatarului trebuie transmise în obiectul clasei PlayerData care este un câmp al său. Aceasta clasă este folosită pentru a crea un obiect cu conexiune la baza de date Firebase pentru a transmite informațiile caracterului în timp curent la aceasta. Conține câmpuri pentru poziția sa în mapă, unghiul de rotație, numărul de vieți, câte bombe poate folosi, raza acestora, cât timp de invincibilitate mai are, de cât timp ține apăsat butonul pentru folosirea bombei, numele caracterului de care aparține și starea de mișcare actuală. Acestea sunt publice pentru a fi vizibile la conectarea cu baza de date. Clasa conține un constructor fără parametrii și unul care primește toate câmpurile ca argumente pentru inițializarea obiectului.

Constructorul pentru OnlinePlayer apelează constructorul clasei părinte pentru inițializarea obiectelor, creează obiectul playerData cu informațiile sale, reține referința către baza de date unde este stocat jocul actual și transmite informațiile din playerData la acea referință la un câmp care folosește playerId-ul caracterului. Referința reprezintă calea către obiectul din baza de date, iar aceasta este determinată de codul primit ca parametru, cod care este folosit doar de utilizatorii participând la jocul actual.

```
playerData = new PlayerData(getRelativePoxX(),
    getRelativePoxY(),
    rotationAngle,
    livesCount,
    bombRange,
    bombsNumber,
    bombUsed: 0,
    invincibilityTime: 0,
    bundle.getString(PERSON_NAME),
    PlayerState.State.NOT_MOVING.toString());

reference = FirebaseDatabase.getInstance().getReference(bundle.getString(CODE));
reference.child(playerId).setValue(playerData);
```

Figura 3.33: Inițializarea obiectelor în constructorul clasei OnlinePlayer

Metoda de update din clasa părinte este suprascrisă. Este similară cu cea din origine, dar aceasta modifică și valorile din playerData în etapele metodei. Câmpurile care se modifică sunt cele pentru unghiul caracterului, starea de mișcare și timpul cât



a apăsător butonul de bombă, iar la final se modifică valorile din baza de date prin rescrierea obiectului în aceasta.

```
@Override
public void update() {
    if (livesCount < 1) {
        return;
    }
    initRectInTiles();
    selectDirectionFromActuator();
    detectCollisions();

    if (velocityX != 0 || velocityY != 0) {
        movePlayer();

        getOrientation();

        // Update player orientation
        rotationAngle = getAngle();
        playerData.rotationData = rotationAngle;
    }

    // Update player state for animation
    playerState.update();
    playerData.movingState = playerState.getState().toString();

    // Use bomb
    if (button.getIsPressed()) {
        useBomb();
        playerData.bombUsed = ++timeUsingBomb;
    } else {
        playerData.bombUsed = 0;
    }

    // Player death handler
    handleDeath();

    // Player picks power up handler
    handlePowerUpCollision();

    reference.child(playerId).setValue(playerData);
}
```

Figura 3.34: Metoda update din clasa OnlinePlayer



Metoda `usePowerUp` din `Player` este suprascrisă pentru a modifica și obiectele câmpurilor din `playerData` la apel. În aceasta se apelează metoda din clasa părinte și se folosește un block switch pentru a selecta ce valoare să modifice.

```
@Override
protected void usePowerUp(Tile.LayoutType layoutType) {
    super.usePowerUp(layoutType);
    switch (layoutType) {
        case RANGE_POWER_UP:
            playerData.bombRange++;
            break;
        case BOMB_POWER_UP:
            playerData.bombNumber++;
            break;
        default:
            break;
    }
}
```

Figura 3.35: Metoda `usePowerUp` din clasa `OnlinePlayer`

Metoda `handleDeath` este și ea suprascrisă, ea având ca diferență doar că la schimbarea valorilor în funcția implementată în `Player` pentru numărul de vieți și time, schimbă și valorile pentru `livesCount` și timpul de invincibilitate din obiectul `playerData`.

Ultima funcție suprascrisă este `movePlayer` folosită pentru a deplasa pătratul caracterului. Mai întâi se apelează metoda care a fost suprascrisă, iar apoi se schimbă valorile din `playerData` pentru poziția caracterului. Valorile din `playerData` sunt relative la mapă obținute prin împărțirea coordonatelor caracterului la dimensiunile respective ale mapei. Se folosesc valori relative pentru putea fi convertite apoi la adversar în timpul jocului pentru formatul și dimensiunea ecranului său.

```
@Override
protected void movePlayer() {
    super.movePlayer();

    playerData.posX = getRelativePoxX();
    playerData.posY = getRelativePoxY();
}

private double getRelativePoxX() {
    return positionX / tilemap.getMapRect().width();
}

private double getRelativePoxY() {
    return positionY / tilemap.getMapRect().height();
}
```

Figura 3.36: Metoda `movePlayer` din clasa `OnlinePlayer`



3.5.3. OnlineEnemy

Această clasă este folosită în jocurile de tip multiplayer, ea reprezentând acțiunile utilizatorilor inamici și extinde clasa abstractă Player.

Câmpurile suplimentare față de clasa părinte sunt un obiect PlayerData și un ValueEventListener. Valorile scrise în baza de date sunt citite folosind un listener și reținute în obiectul playerData la fiecare modificare, cea ce se execută în clasa MultiplayerGame care gestionează jocurile cu mai mulți utilizatori. Obiectul ValueEventListener este reținut pentru a putea fi ulterior eliminat pentru a nu rămâne ca un proces activ când nu mai este nevoie de citirea valorilor din baza de date.

Constructorul clasei apelează constructorul din superclasă și creează obiectul playerData.

Metoda de update este suprascrisă, ea fiind simplificată față de cea din Player deoarece informația a fost prelucrată anterior pe dispozitivul utilizatorului a cărei informație îi corespunde, iar obiectul actual este folosit de ceilalți utilizatori din joc pentru a arăta acea informație. Se mută pătratul playerRect al caracterului în funcție de coordonatele relative din playerData convertite pentru mapa de joc actuală prin înmulțirea acestora cu dimensiunile mapei.

```
private int getPosX() {  
    return (int) (playerData.posX * tilemap.getMapRect().width());  
}  
  
private int getPosY() {  
    return (int) (playerData.posY * tilemap.getMapRect().height());  
}
```

Figura 3.37: Metodele de convertire a pozițiilor relative la cele actuale din clasa OnlineEnemy

Câmpurilor playerState, rotationAngle, livesCount, bombsNumber și bombRange din clasă le sunt alocate valorile din playerData cu același nume. Pentru variabila bombUsed din playerData, se verifică dacă este mai mare ca zero, caz în care se apelează metoda useBomb pentru a încerca folosirea unei bombe. În final se apelează funcțiile handleDeath și handlePowerUpCollision care au fost suprascrise de către clasă.

Metoda handlePowerUpCollision suprascrisă, spre deosebire de cea din clasa Player, nu apelează metoda usePowerUp deoarece dacă avatarul trece peste un PowerUp, creșterea valorilor este efectuată pe dispozitivul respectiv, valorile modificate fiind transmise doar prin playerData. Astfel, este apelată metoda pentru a schimba pătratul peste care a trecut.

Funcția handleDeath este și ea simplificată, ea primind timpul ca invincibilitate prin playerData, trebuie doar gestionată valoarea din variabila usedPaint în funcție de acest timp. Pentru cazul în care timpul este zero, se va intra pe ramura de zero a blocului switch care va pune null în usedPaint, rezultând într-o reprezentare normală pe ecran.



```

@Override
protected void handleDeath() {
    int aux = playerData.invincibilityTime % 4;
    switch (aux) {
        case 0:
            usedPaint = null;
            break;
        case 2:
            usedPaint = invincibilityPaint;
            break;
    }
}

```

Figura 3.38: Metoda handleDeath din clasa OnlineEnemy

3.5.4. OfflineEnemy

Moștenește clasa abstractă Player și este folosită pentru a gestiona inamici din modul de joc Singleplayer. Aceștia sunt controlați de reguli stabilite pentru mișcările lor. Se caută mișcările în funcție de starea actuală a caracterului în mapă, dacă trebuie să se ferească de o explozie sau să folosească bombe în spații optime care au valoare mare.

Clasa conține definiția a două enumerații: SearchType care are trei valori, stepOnNoExplosion, stepOnNoCurrentExplosion și stepOnAllExplosion, aceasta reprezentând tipul de căutare pentru mișcarea pe care o va executa, și OriginDirection, enumerație care conține valori pentru cele patru direcții și o valoare NONE, folosite pentru a completa tabela care înregistrează drumurile posibile ale caracterului.

Conține trei constante, POWER_UP_SCORE, CRATE_SCORE și ENEMY_SCORE, valori folosite pentru a completa o matrice cu scorurile pătratelor accesibile de către caracter. O mapă care are ca și chei obiecte String, iar ca valori, obiecte Pair provenite din androidx.core.util. perechea este alcătuită din două valori Integer. Obiectul final mapă numit enemiesPos este primit ca parametru în constructor, iar acesta reține pozițiile tuturor caracterelor care conțin cel puțin o viață, cheia reprezintă playerId-ul lor, iar în pereche se rețin linia și coloana actuală în care se află acestea. Ca și câmpuri, există două tablouri bidimensionale de același număr de linii și coloane ca tilemap-ul, acestea sunt allExplosionMap care are obiecte de LayoutType care enumeră tipul de Tile-uri și este folosit pentru a reține o mapă în care toate bombele în momentul actual au fost deja explodate, folosită pentru căutarea spațiilor sigure, și directions care este alcătuit din elemente din enumerația OriginDirection. Cea din urmă tabelă este folosită pentru înregistrarea direcției de origine de unde a început mișcarea.

Constructorul apelează super constructorul și reține obiectul enemiesPos venit ca argument.

Metoda de update este implementată similar cu cea din clasa OfflinePlayer, doar că se folosește bombă și vitezele sunt determinate în funcția makeMoves a clasei.

În funcția makeMoves, se apelează metoda findDestination care returnează un obiect pair de Integer reținut în variabila endPos care reprezintă destinația finală calculată pentru caracter în momentul actual. Dacă valoarea returnată este nulă, atunci se oprește execuția metodei mai departe. În caz contrar, se apelează metoda getPath care primește ca parametru variabila endPos și returnează o listă de obiecte



OriginDirection numită path, reprezentând drumul către pătratul destinație. Urmează să se verifice dacă variabila endPosScore care este câmp în clasă are valoare mai mare ca zero, valoare care este atribuită în execuția metodei findDestination, și dacă lista path are elemente în aceasta, caz în care înseamnă că a ajuns la pătratul dorit pentru a folosi o bombă, ceea ce se execută prin apelarea metodei useBomb, urmând să se termine execuția metodei. În caz contrar, vitezele își presetează valoarea zero ca apoi în blocul switch să se schimbe una din acestea în funcție de ce direcție este trecută în ultimul obiect din lista path. În final, se folosește o reducere a vitezei finale aleator pentru a crea variație a mișcărilor, astfel, în timpul jocului, inamici să nu se suprapună și să rămână pe tot parcursul jocului.

```
private void makeMoves() {
    Pair<Integer, Integer> endPos = findDestination();
    if (endPos == null) {
        return;
    }
    ArrayList<OriginDirection> path = getPath(endPos);

    if (endPosScore > 0 && path.isEmpty()) {
        useBomb();
        return;
    }

    velocityX = 0;
    velocityY = 0;
    switch (path.get(path.size() - 1)) {
        case NONE:
            break;
        case LEFT:
            velocityX = -getMaxSpeed();
            break;
        case UP:
            velocityY = -getMaxSpeed();
            break;
        case RIGHT:
            velocityX = getMaxSpeed();
            break;
        case DOWN:
            velocityY = getMaxSpeed();
            break;
    }
    if (Utils.generator.nextFloat() < .3) {
        velocityX /= 2;
        velocityY /= 2;
    }
}
```

Figura 3.39: Metoda makeMove din clasa OfflineEnemy

Pentru a putea determina starea viitoare a jocului cu scopul de a evita exploziile următoare, se creează o mapă în care toate exploziile au fost detonate. Acest lucru se realizează în metoda findFutureExplosions care returnează o matrice de LayoutType. În aceasta se crează un nou tablou de dimensiunile tilemap-ului numit map, se apelează copyOldTilemap care primește map-ul ca parametru unde se va copia în acesta LayoutType-urile din tilemap și apoi se apelează triggerAllBombs cu



același parametru pentru a schimba mapa în cea cu toate bombele detonate și se returnează obiectul map.

În metoda `copyOldTilemap` care primește matricea de `LayoutType` ca parametru, parcurge toate tile-urile din tilemap și pentru fiecare din acestea se salvează din acestea `LayoutType`-ul primit prin apelul la metoda `getLayoutType` pe tile.

În metoda `triggerAllBombs` care primește matricea ca parametru, se parcurg toate obiectele din matricea venită ca parametru, iar dacă obiectul este bombă, atunci se apelează metoda `triggerBomb` din clasă și în spațiul cu bombă se înlocuiește cu o explozie. Metoda `triggerBomb` folosește același principiu ca metoda de detonare a bombei din clasa `Bomb`, primind ca parametru o bombă, se apelează funcția clasei `explodeLength` pentru a schimba în explozie în cele patru direcții. În `explodeLength`, verifică pătratele prinse în raza bombei și sunt schimbate în `LayoutType` de explozie.

```
private void triggerBomb(Bomb bomb, LayoutType[][] map) {
    for (int i = 0; i < 4; i++) {
        explodeLength(bomb, ROWS[i], COLUMNS[i], map);
    }
}

private void explodeLength(Bomb bomb, int idxRow, int idxColumn, LayoutType[][] map) {
    int row = bomb.getRow();
    int column = bomb.getColumn();
    int range = bomb.getRange();

    for (int idx = 1; idx < range; idx++) {
        switch (map[row + idxRow * idx][column + idxColumn * idx]) {
            case WALK:
            case BOMB_POWER_UP:
            case RANGE_POWER_UP:
            case SPEED_POWER_UP:
                map[row + idxRow * idx][column + idxColumn * idx] = LayoutType.EXPLOSION;
                break;
            case EXPLOSION:
            case BOMB:
                break;
            case CRATE:
            case WALL:
            default:
                return;
        }
    }
}
```

Figura 3.40: Metodele pentru detonarea bombelor din clasa `OfflineEnemy`

Funcția `findDestination` este cea care determină coordonatele pătratului la care se dorește să se ajungă, care sunt returnate sub forma unei perechi. Mai întâi se atribuie tabloului `allExplosionMap` valoarea returnată de metoda `findFutureExplosions`, reținând în acesta o mapă cu toate exploziile detonate. Se inițializează `endPosScore` cu valoarea `endPosScore` urmând să se modifice ulterior, câmpurile `playerRow` și `playerColumn` cu linia și coloana pătratului în care se află acum caracterul. Prin apelul la metoda `tileIsDangerous` se determină dacă pătratul pe



care se află caracterul este sigur, adică dacă acesta se află acum pe o explozie în tilemap sau dacă este pe o explozie în mapa cu toate bombele detonate. Cele două valori sunt reținute în variabilele `nowHasExpTile` și respectiv `allHasExpTile`, iar aceste valori sunt folosite pentru a determina ce tip de mișcare trebuie să execute caracterul. Astfel, dacă `nowHasExpTile` are valoarea de adevăr, atunci se returnează valoarea primită din apelul metodei `search` din clasă cu parametrul `SearchType.stepOnAllExplosion`, dacă `allHasExpTile` are valoarea de adevăr, atunci se returnează valoarea primită din `search` cu parametrul `SearchType.stepOnNoCurrentExplosion`, iar dacă amândouă valorile au fost zero, se apelează cu parametrul `SearchType.stepOnNoExplosion`.

```
private Pair<Integer, Integer> findDestination() {
    allExplosionMap = findFutureExplosions();
    endPosScore = 0;

    playerRow = getPlayerRow( p: this);
    playerColumn = getPlayerColumn( p: this);

    boolean allHasExpTile = tileIsDangerous(allExplosionMap[playerRow][playerColumn]);
    boolean nowHasExpTile = tileIsDangerous(tilemap.getTilemap()[playerRow][playerColumn]);

    if (nowHasExpTile) {
        //search in allExplosionMap first safe space
        return search(SearchType.stepOnAllExplosion);
    }
    if (allHasExpTile) {
        //search in allExplosionMap first safe space, but cant step on current danger
        return search(SearchType.stepOnNoCurrentExplosion);
    }
    //search in allExplosionMap for place to use bomb
    return search(SearchType.stepOnNoExplosion);
}
```

Figura 3.41: Metoda `findDestination` din clasa `OfflineEnemy`

Funcția `search` pregătește căutarea drumurilor posibil de efectuat de caracter. În aceasta se alocă o nouă matrice în câmpul `directions` de tip `OriginDirection` și de dimensiunile mapei, iar la poziția actuală a caracterului se setează valoarea `NONE`, deoarece aceasta este poziția de origine și nu provine din nici o direcție. Se creează un obiect coadă care conține perechi cu coordonate folosită pentru căutarea spațiilor de mers, și se adaugă coordonatele pătratului actual ale caracterului. Urmează un bloc `switch` în care, dacă valoarea primită ca parametru este `stepOnNoExplosion`, se returnează valoarea venită din apelul metodei `searchNoExplosion` cu parametrul `searchQueue`, coada anterior descrisă, iar dacă valoarea este una din celelalte două variante, se apelează metoda `searchExplosions` cu parametrul `searchExplosions` și variabila `searchType` pentru tipul de căutare. Cele două funcții folosesc un algoritm `Depth-first search`⁷ pentru a parcurge spațiile din mapă accesibile caracterului, folosind coada `searchQueue` pentru coada folosită în parcurgere și matricea `directions` pentru a nota tile-urile vizitate și a determina originea drumului.

⁷ Depth-first search – algoritm de parcurgere în adâncime pe un graf, arbore sau matrice (cazul de față)



```

private Pair<Integer, Integer> search(SearchType searchType) {
    int rows = tilemap.getNumberOfRowTiles();
    int columns = tilemap.getNumberOfColumnTiles();

    directions = new OriginDirection[rows][columns];
    directions[playerRow][playerColumn] = OriginDirection.NONE;

    Queue<Pair<Integer, Integer>> searchQueue = new LinkedList<>();

    searchQueue.add(new Pair<>(playerRow, playerColumn));

    switch (searchType) {
        case stepOnNoExplosion:
            return searchNoExplosion(searchQueue);
        case stepOnNoCurrentExplosion:
        case stepOnAllExplosion:
            return searchExplosions(searchQueue, searchType);
        default:
            throw new IllegalStateException("Unexpected value: " + searchType);
    }
}

```

Figura 3.42: Metoda search din clasa OfflineEnemy

Funcția searchExplosions este folosită pentru cazurile în care caracterul se află pe un pătrat care este sau care va deveni explozie, iar aceasta caută primul pătrat care nu este explozie pentru a fi în siguranță. Se aplică algoritmul de parcurgere în adâncime, astfel, se execută bucla while atât timp cât coada încă are elemente în ea, se extrage primul element din searchQueue și i se verifică cele patru laterale. Dacă pătratul lateral găsit are la coordonatele sale din directions valoarea nenulă, se trece la următoarea iterație a for-ului care verifică cele patru laterale deoarece pătratul a fost deja parcurs, în caz contrar, se folosește un switch pentru a determina ce tip de pătrat este cel parcurs acum. Dacă acesta este de tip CRATE, BOMB sau WALL, se trece la următorul pătrat deoarece acesta nu poate fi străbătut de caracter, dacă este EXPLOSION, atunci se verifică dacă tipul de căutare primit ca parametru este stepOnNoCurrentExplosion și dacă tile-ul verificat este explozie în tilemap, caz în care se continuă iterația prin oprirea switch-ului. Astfel, dacă avatarul a fost pe o explozie și este în perioada de invincibilitate, poate să treacă peste celelalte explozii actuale, dar dacă el nu a atins încă explozia, reprezentând tipul de căutare stepOnNoCurrentExplosion, atunci poate să treacă doar peste exploziile viitoare pentru a găsi un pătrat sigur. Dacă nu a fost suspendată execuția în switch în condiție, atunci se adaugă încă o pereche reprezentând pătratul actual și se determină valoarea din directions pentru pătratul actual prin apelul metodei getDirection. Funcția getDirection primește coordonatele pătratului adăugat acum și cel verificat anterior pentru a determina ce direcție să fie returnată prin valoarea diferenței lor.



```

private OriginDirection getDirection(int rowStart, int columnStart, int rowEnd, int columnEnd) {
    switch (rowStart - rowEnd) {
        case -1:
            return OriginDirection.DOWN;
        case 0:
            break;
        case 1:
            return OriginDirection.UP;
    }

    switch (columnStart - columnEnd) {
        case -1:
            return OriginDirection.RIGHT;
        case 1:
            return OriginDirection.LEFT;
    }

    return null;
}

```

Figura 3.43: Metoda `getDirection` din clasa `OfflineEnemy`

Iar în final, dacă valoarea pătratului actual este drum sau un PowerUp, se notează direcția în `directions` și se returnează perechea formată din coordonatele acestuia. Valoarea returnată este de fapt primul pătrat sigur întâlnit din care se determină drumul de deplasare.

```

private Pair<Integer, Integer> searchExplosions(Queue<Pair<Integer, Integer>> searchQueue,
                                                SearchType searchType) {
    while (!searchQueue.isEmpty()) {
        int row, column;
        Pair<Integer, Integer> pair = searchQueue.remove();

        for (int i = 0; i < 4; i++) {
            row = pair.first + ROWS[i];
            column = pair.second + COLUMNS[i];

            if (directions[row][column] != null) {
                continue;
            }

            switch (allExplosionMap[row][column]) {
                case WALK:
                case BOMB_POWER_UP:
                case RANGE_POWER_UP:
                case SPEED_POWER_UP:
                    directions[row][column] = getDirection(pair.first, pair.second, row, column);
                    return new Pair<>(row, column);
                case EXPLOSION:
                    if (searchType == SearchType.stepOnNoCurrentExplosion &&
                        tileIsDangerous(tilemap.getTilemap()[row][column])) {
                        break;
                    }
                    searchQueue.add(new Pair<>(row, column));
                    directions[row][column] = getDirection(pair.first, pair.second, row, column);
                    break;
                case CRATE:
                case BOMB:
                case WALL:
                default:
                    break;
            }
        }
    }
    return null;
}

```

Figura 3.44: Metoda `searchExplosions` din clasa `OfflineEnemy`



Funcția `searchNoExplosion` este cea folosită dacă se află în siguranță caracterul și caută un pătrat să folosească o bombă. Spre deosebire de `searchExplosions`, în această funcție se parcurg toate drumurile posibile și se determină destinația care are cel mai mult potențial. Se declară și se inițializează o matrice de dimensiunea mapei cu valori `int` numită `scores` care va conține valori în funcție de punctele de interes din acel pătrat, cum sunt `PowerUp`-urile, cutiile din jurul pătratului și celelalte caractere. Se iterează mapa de obiecte `enemiesPos` care conține pozițiile tuturor caracterelor, iar dacă obiectul curent nu este caracterul actual, atunci la poziția lui în tabloul `scores` se adaugă valoarea pentru inamici `ENEMY_SCORE`. Se declară o pereche numită `endPos` care va conține destinația finală a caracterului.

```
private Pair<Integer, Integer> searchNoExplosion(Queue<Pair<Integer, Integer>> searchQueue) {
    int rows = tilemap.getNumberOfRowTiles();
    int columns = tilemap.getNumberOfColumnTiles();
    int[][] scores = new int[rows][columns];

    int row, column;

    for (Map.Entry<String, Pair<Integer, Integer>> entry : enemiesPos.entrySet()) {
        row = entry.getValue().first;
        column = entry.getValue().second;
        if (entry.getKey().equals(playerId)) {
            continue;
        }
        scores[row][column] += ENEMY_SCORE;
    }
    Pair<Integer, Integer> endPos = null;
}
```

Figura 3.45: Crearea matricei `scores` în metoda `searchNoExplosion` din clasa `OfflineEnemy`

Urmează o buclă `while` similară cu cea din funcția `searchExplosions`, dar diferită în blocul `switch`. Dacă întâlnește pătrat `WALK`, atunci se continuă execuția prin adăugarea pătratului actual la coadă și este notat în `directions`, pentru `BOMB_POWER_UP`, `RANGE_POWER_UP` și `SPEED_POWER_UP`, este adăugat și notat ca mai sus, dar se și adaugă `POWER_UP_SCORE` la valoarea pătratului din `scores`, pentru `CRATE` se adaugă în `scores` `CRATE_SCORE` pentru pătratul anterior care are acces la cutie, iar pentru `EXPLOSION`, `BOMB` și `WALL`, se continuă execuția `for`-ului. După terminarea buclei `for`, se verifică dacă scorul obținut în pătratul în jurul căruia s-a căutat este mai mare decât cel din `endPosScore`, caz în care valoarea `i` se schimbă în aceasta și `endPos` ia valoarea pătratului respectiv. După terminarea execuției buclei `while`, se returnează valoarea `endPos` care a reținut perechea cu coordonatele pătratului cu cel mai mare scor, fiind determinată ca cea mai bună poziție de a folosi o bombă.



```

while (!searchQueue.isEmpty()) {
    Pair<Integer, Integer> pair = searchQueue.remove();

    for (int i = 0; i < 4; i++) {
        row = pair.first + ROWS[i];
        column = pair.second + COLUMNS[i];

        if (directions[row][column] != null) {
            continue;
        }

        switch (allExplosionMap[row][column]) {
            case WALK:
                searchQueue.add(new Pair<>(row, column));
                directions[row][column] = getDirection(pair.first, pair.second, row, column);
                break;
            case BOMB_POWER_UP:
            case RANGE_POWER_UP:
            case SPEED_POWER_UP:
                searchQueue.add(new Pair<>(row, column));
                directions[row][column] = getDirection(pair.first, pair.second, row, column);
                scores[row][column] += POWER_UP_SCORE;
                break;
            case CRATE:
                scores[pair.first][pair.second] += CRATE_SCORE;
                break;
            case EXPLOSION:
            case BOMB:
            case WALL:
            default:
                break;
        }
    }

    if (scores[pair.first][pair.second] > endPosScore) {
        endPosScore = scores[pair.first][pair.second];
        endPos = pair;
    }
}

return endPos;
}

```

Figura 3.46: Bucla while de parcurgere a mapei din metoda searchNoExplosion din clasa OfflineEnemy

Funcția getPath primește ca parametru o pereche reprezentând poziția finală la care dorește să ajungă caracterul, și returnează o listă cu direcțiile de la origine. Matricea directions are la coordonatele primite ca parametru direcția care determină originea, astfel urmând direcțiile, se determină traseul care trebuie străbătut pentru a ajunge la destinație. Aceasta se realizează în bucla while care este executată cât timp variabila direction care conține direcția din tabloul directions către origine are valoare diferită de NONE și este nenulă, se adaugă valoarea din directions în lista path care va conține drumul, într-un switch se determină cum se schimbă linia și coloana care determină pătratul din directions în funcție de valoarea din direction care urmează să fie schimbată pentru a determina poziția acutuală. La ieșirea din while, se returnează lista path care are ultimul element prima direcție care trebuie urmată de către caracter.




```

private ArrayList<OriginDirection> getPath(Pair<Integer, Integer> pos) {
    int row = pos.first;
    int column = pos.second;
    OriginDirection direction = directions[row][column];

    ArrayList<OriginDirection> path = new ArrayList<>();
    while (direction != null && direction != OriginDirection.NONE) {
        path.add(direction);
        switch (direction) {
            case LEFT:
                column++;
                break;
            case UP:
                row++;
                break;
            case RIGHT:
                column--;
                break;
            case DOWN:
                row--;
                break;
        }
        direction = directions[row][column];
    }

    return path;
}

```

Figura 3.47: Metoda getPath din clasa OfflineEnemy

3.6. Mod Singleplayer

Extinde clasa Game, iar aceasta este folosită pentru a gestiona un joc de tip Singleplayer, folosind pentru utilizator un obiect OfflinePlayer și obiecte OfflineEnemy pentru inamici.

Ca și câmpuri suplimentare față de clasa Game, conține două obiecte map numite enemies și enemiesPos având drept cheie tipul de date String, folosit pentru a memora playerId-ul player-ilor. Obiectul enemies are ca valori obiecte OfflineEnemy, iar aceasta este lista folosită pentru iterarea și gestionarea inamicilor din modul Singleplayer. Obiectul enemiesPos are ca valoare obiecte Pair formate din două valori Integer care reține linia și coloana tuturor player-ilor, obiect folosit de clasa OfflineEnemy pentru găsirea inamicilor pe mapă.

În constructor se apelează constructorul din superclasă, se atribuie câmpului player din clasa părinte un nou obiect OfflinePlayer, acesta fiind caracterul controlat de utilizator, inițializează obiectele mapă cu noi HashMap-uri⁸, și apelează metodele addEnemies și addEnemiesPos pentru a inițializa valorile din acestea.

Funcția addEnemies verifică succesiv câți inamici trebuie adăugați, astfel, dacă numărul este mai mare ca zero, înseamnă că trebuie adăugat cel puțin player-ul numărul doi, adică cel care folosește obiect animator pentru caracter roșu, se verifică dacă numărul este mai mare ca unu, ceea ce înseamnă că trebuie adăugat și caracterul de culoare galbenă, și valoarea este mai mare ca doi, atunci se adaugă și ultimul caracter, cel verde. Se apelează metoda createEnemy pentru a adăuga și pentru a

⁸ HashMap - permite stocarea de obiecte folosind identificatori



crea obiectele `OfflineEnemy` respective și adăugate în mapă. Metoda primește ca parametri coordonatele colțurilor hărții de joc deoarece acela este locul de unde începe deplasarea caracterelor, obiecte `Animator` pentru fiecare culoare și un string reprezentând id-ul caracterului.

```
private void addEnemies(Bundle bundle) {
    int count = bundle.getInt(ENEMY_COUNT);
    // add player2
    if (count > 0) {
        createEnemy( rowTile: tilemap.getNumberOfRowTiles() - 2,
                     columnTile: tilemap.getNumberOfColumnTiles() - 2,
                     new Animator(spriteSheet.getRedPlayerSpriteArray()),
                     Players.PLAYER2.toString());
    }

    // add player3
    if (count > 1) {
        createEnemy(PLAYER_START_ROW,
                     columnTile: tilemap.getNumberOfColumnTiles() - 2,
                     new Animator(spriteSheet.getYellowPlayerSpriteArray()),
                     Players.PLAYER3.toString());
    }

    // add player4
    if (count > 2) {
        createEnemy( rowTile: tilemap.getNumberOfRowTiles() - 2,
                     PLAYER_START_COLUMN,
                     new Animator(spriteSheet.getGreenPlayerSpriteArray()),
                     Players.PLAYER4.toString());
    }
}
```

Figura 3.48: Metoda `addEnemies` din clasa `SingleplayerGame`

Funcția `createEnemy` creează un nou obiect `OfflineEnemy` și îl adaugă în colecția `enemies` la cheia string primită ca parametru, iar în final, setează string-ul ca id pentru obiectul respectiv.

Funcția `addEnemiesPos` adaugă mai întâi la cheia caracterului numărul unei poziții din hartă a player-ului, acesta fiind utilizatorul, apoi sunt adăugate valorile inamicilor luând valorile prin iterarea colecției `enemies`. Valorile care sunt salvate la cheile respective sunt venite din apelul metodei `getPlayerPos` care primește un obiect `Player` și returnează o nouă pereche de coordonate ale pătratului în care se află prin apelul la metodele de identificare a liniei și coloanei pentru Player-i din clasa `Utils`.

```
private void addEnemiesPos() {
    enemiesPos.put(Players.PLAYER1.toString(), getPlayerPos(player));
    for (Map.Entry<String, OfflineEnemy> enemy : enemies.entrySet()) {
        enemiesPos.put(enemy.getKey(), getPlayerPos(enemy.getValue()));
    }
}

@NonNull
private Pair<Integer, Integer> getPlayerPos(Player p) {
    return new Pair<>(getPlayerRow(p), getPlayerColumn(p));
}
```

Figura 3.49: Metoda `addEnemiesPos` din clasa `SingleplayerGame`



Funcția `handleGameEnded` care este abstractă în clasa `Game` este implemetată pentru a verifica dacă jocul s-a încheiat. Aceasta se realizează prin verificarea numărului de vieți ale utilizatorului dacă este mai mare ca zero, caz în care dacă nu mai sunt inamici în colecția `enemies` pentru că au pierdut și au fost scoși din mapă, atunci înseamnă că utilizatorul a câștigat și se trimite pe ecran mesajul respectiv sub forma unui Toast⁹. Dacă utilizatorul nu mai are vieți, se verifică câte obiecte mai sunt în colecția `enemies`, dacă este liberă, înseamnă ca este o remiză deoarece toate caracterele rămase au pierdut în același timp, dacă este un singur obiect, înseamnă că acela a câștigat și se transmite un mesaj corespunzător.

```
@Override
public void handleGameEnded() {
    if (player.getLivesCount() > 0) {
        if (enemies.isEmpty()) {
            endgameMessage(WIN_END_MSG);
        }
    } else {
        switch (enemies.size()) {
            case 0:
                endgameMessage(TIE_END_MSG);
                break;
            case 1:
                String color = getColorString();
                endgameMessage(msg: color + " Won");
                break;
            default:
                break;
        }
    }
    playerCountChanged = false;
}
```

Figura 3.50: Metoda `handleGameEnded` din clasa `SingleplayerGame`

Funcția de `draw` a clasei este suprascrisă, în aceasta se apelează metoda din superclasă, iar apoi se iterează obiectele din colecția `enemies` și este apelată metoda de `draw` pentru aceștia.

Metoda de `update` este și ea suprascrisă, aceasta fiind necesară să gestioneze lista cu inamici. În aceasta se apelează metoda de `update` din clasa părinte. Dacă numărul de elemente din `enemies` este mai mare ca zero și caracterul utilizatorului mai are vieți, atunci se apelează funcția de `update` a câmpului `player` și se modifică valoarea din `enemiesPos` pentru poziția sa actuală. Dacă în urma actualizării a pierdut ultima viață, atunci se elimină acesta din colecția `enemiesPos`. Urmează să se parcurgă colecția `enemies` folosind un iterator. După ce se apelează funcția de `draw` pentru obiectele `OfflineEnemy` din `enemies`, se actualizează și pozițiile lor din colecție. În partea finală din actualizarea inamicilor se verifică dacă inamicul nu mai are vieți, caz în care acesta este eliminat din ambele colecții.

⁹ Toast - o notificare scurtă care apare pe ecran pentru a informa utilizatorul despre un eveniment sau o acțiune



```

@Override
public void update() {
    super.update();

    if (!enemies.isEmpty() && player.getLivesCount() > 0) {
        player.update();
        enemiesPos.put(player.getPlayerId(), getPlayerPos(player));

        if (player.getLivesCount() < 1) {
            playerCountChanged = true;
            enemiesPos.remove(player.getPlayerId());
        }
    }

    Iterator<Map.Entry<String, OfflineEnemy>> iterator = enemies.entrySet().iterator();
    while (iterator.hasNext()) {
        OfflineEnemy enemy = iterator.next().getValue();
        enemy.update();
        enemiesPos.put(enemy.getPlayerId(), getPlayerPos(enemy));
        if (enemy.getLivesCount() > 0) {
            continue;
        }
        //remove enemy
        playerCountChanged = true;
        enemiesPos.remove(enemy.getPlayerId());
        iterator.remove();
    }
}

```

Figura 3.51: Metoda update din clasa SingleplayerGame

3.7. Mod Multiplayer

Extinde clasa Game, iar aceasta este folosită pentru a gestiona un joc de tip Multiplayer, folosind pentru utilizator un obiect OnlinePlayer și obiecte OnlineEnemy pentru inamici. Clasa gestionează scrierea și citirea informațiilor în baza de date Firebase.

Aduce în plus drept câmpuri o listă de obiecte OnlineEnemy numită enemies folosită pentru a actualiza inamici și a-i desena, și o referință DatabaseReference folosită pentru a accesa informațiile din baza de date.

Constructorul îl invocă pe cel din superclasă, iar apoi continuă cu crearea elementelor. Spre deosebire de modul Singleplayer unde utilizatorul este întotdeauna caracterul numărul 1 de culoare roșie, în modul Multiplayer, caracterul este determinat de playerId-ul său, iar acesta ia valoare în funcție de ordinea de intrare în jocul actual, astfel, i se determină poziția sa în mapă și culoarea, iar acestea sunt folosite la crearea obiectului OnlinePlayer. Se continuă prin inițializarea listei enemies și formând referința din reference pentru a face conexiunea cu camera de joc din baza de date. În final, se execută metoda addEnemiesListeners care generează informațiile inamicilor.



Funcția `addEnemiesListeners` începe generearea obiectelor `OnlineEnemy`, în aceasta se creează o solicitare pentru a obține arborele din baza de date aflat la referința camerei sale de joc. În cazul în care a avut succes, se iterează cele patru id-uri ale caracterelor, ia dacă acesta nu este cel al player-ului și există în arbore, atunci se apelează `createEnemy` cu id-ul respectiv ca parametru.

```
private void addEnemiesListeners() {
    reference.get().addOnCompleteListener(task -> {
        if (!task.isSuccessful()) {
            Log.e(FIREBASE_TAG, RETRIEVE_DATA_ERROR, task.getException());
        } else {
            Log.d(FIREBASE_TAG, String.valueOf(task.getResult().getValue()));

            DataSnapshot dataSnapshot = task.getResult();

            ArrayList<String> enemiesIdList = new ArrayList<>(Arrays.asList(PLAYERS));
            enemiesIdList.remove(playerId);
            for (String id : enemiesIdList) {
                if (dataSnapshot.child(id).exists()) {
                    createEnemy(id);
                }
            }
        }
    });
}
```

Figura 3.52: Metoda `addEnemiesListeners` din clasa `MultiplayerGame`

În `createEnemy`, se alege ce culoare va avea caracterul în animator în funcție de valoarea primită ca argument și se creează obiectul clasei `OnlineEnemy`. Se apelează metoda `createListener` care primește ca argument obiectul creat pentru a face legătura obiectului cu cel din baza de date, iar apoi obiectul este adăugat în lista `enemies`.

În funcția `createListener`, se creează un listener (ascultător) atașat referinței bazei de date pentru a primi notificări dacă se produc modificări la datele de la referința respectivă. Se folosește metoda `addValueEventListener` din pachetul `Firebase` pentru a crea un nou `ValueEventListener` cu care, dacă se modifică datele din baza de date aflate la referința către inamicul respectiv, atunci obiectul `playerData` din `OnlineEnemy` o să primească noile valori, astfel, când este actualizat și desenat inamicul, acesta va folosi ultimele informații din baza de date din `playerData`.

```
private void createListener(OnlineEnemy enemy) {
    enemy.setListener(reference.child(enemy.getPlayerId()).
        addValueEventListener(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
                enemy.setPlayerData(dataSnapshot.getValue(PlayerData.class));
            }

            @Override
            public void onCancelled(@NonNull DatabaseError databaseError) {
                Log.w(CANCELLED_TAG, FAIL, databaseError.toException());
            }
        }));
}
```

Figura 3.53: Metoda `createListener` din clasa `MultiplayerGame`



Metoda de draw face suplimentar de cea din superclasă iterarea inamicilor din enemies și invocarea metodei lor de draw. Metoda de update este similară cu cea implementată în clasa Singleplayer, după apelarea metodei din superclasă, se actualizează caracterul utilizatorului dacă acesta mai are vieți și mai sunt inamici, iar apoi se parcurg inamici pentru a-i actualiza. Dacă în urma metodei de update nu mai au vieți, atunci se apelează metoda `removeEventListener` ascultătorul inamicului pentru a nu consuma resurse cu un obiect care nu mai are importanță și pe urmă este eliminat din lista enemies.

Funcția `handleGameEnded` verifică finalul de joc la fel ca metoda din `Singleplayer`, doar că atunci când s-a identificat finalul de joc, se apelează metodele `removeServer` și `removeListeners`. Funcția `removeListeners` iterează obiectele din enemies rămase și apelează metoda pentru a elimina listener-ul de pe aceștia, iar `removeServer` șterge tot arborele de la referință, astfel se eliberează informațiile camerei și se poate folosi același cod în viitor.

```
private void removeListeners() {
    for (OnlineEnemy e : enemies) {
        reference.removeEventListener(e.getListener());
    }
}

public void removeServer() {
    reference.removeValue();
}
```

Figura 3.54: Metodele `removeListeners` și `removeServer` din clasa `MultiplayerGame`

3.8. Firebase

Deoarece tipul de bază de date Firebase folosit este Realtime Database, datele sunt salvate ca arbori. Astfel, fiecare cameră de joc are informațiile salvate într-un arbore începând cu o rădăcină având valoarea codului ales de creatorul acestei camere.



Figura 3.55: Noduri pentru jocuri diferite din baza de date

Există două informații specifice unei camere, nodul `seed` care conține o valoare `long int` care este generată aleator de cel care a creat camera cu codul respectiv și va fi folosit de toți jucătorii pentru a crearea generatorului de numere aleatoare folosit de fiecare să își creeze harta de joc și diferitele acțiuni care folosesc generatorul, și nodul `gameState` care este folosit pentru a permite intrarea altor persoane în joc. Inițial, valoarea nodului este `waiting players`, stadiu în care, persoanele care au codul pot intra în camera de așteptare, dar nu pot să intre dacă sunt deja patru persoane care



au intrat deja. Nodul își schimbă valoarea atunci când cel care a creat camera deschide jocul, iar pe noua valoare nu pot intra alte persoane. Nodul camerei mai conține nodurile jucătorilor, aceștia pot să fie minim doi și maxim patru pentru a începe un joc. conexiunea cu programul cu nodul camerei din baza de date are loc prin intermediul clasei `ServerData` care conține aceste informații.



Figura 3.56: Noduri pentru jocuri diferite din baza de date

Când se începe jocul, se adaugă câmpurile pentru lățimea și lungimea mapei, și șansa să apară cutii în mapă, folosite la generarea hărții la fiecare utilizator.



Figura 3.57: Informațiile pentru un joc din baza de date

Nodul unui jucător conține informațiile din clasa `PlayerData`. La vizionarea lor în baza de date în timpul unui joc, sunt evidențiate câmpurile care au avut valorile modificate în ultima secundă, iar când aceștia rămân fără vieți sunt eliminați din nodul camerei.



Figura 3.58: Informațiile unui utilizator în timpul jocului din baza de date



3.9. Activitățile

Au fost create mai multe pentru a îndruma utilizatorul către cele două moduri de joc.

În activitatea care apare la deschiderea aplicației care folosește un TextView pentru a afișa numele aplicației, are un EditText folosit pentru a scrie numele utilizatorului, nume care va fi folosit în jocurile multiplayer pentru a se diferenția față de ceilalți utilizatori și două elemente Button la apăsarea cărora se intră în modul de joc Singleplayer sau respectiv Multiplayer. La apăsarea acestor butoane, numele care se află în momentul actual în câmpul pentru nume va fi reținut în memorie folosind suportul oferit în clasa PreferenceManager. La intrarea ulterioară în joc sau întoarcerea în această activitate, numele salvat va fi cel ce va apărea automat în EditText.

La apăsarea butonului pentru modul singleplayer, se deschide o activitate folosită pentru a crea un nou joc. Aici avem un Spinner folosit pentru a selecta câți inamici să existe, trei EditText-uri folosite pentru a scrie parametrii mapei de joc și două butoane, unul pentru a începe jocul și unul pentru întoarcerea la activitatea trecută. Când se începe jocul, informațiile care sunt notate în câmpurile pentru hartă sunt convertite la valori int pentru a le folosi în constructorul pentru crearea acesteia, iar dacă valorile din câmpuri nu au fost valide sau lăsate libere, se vor folosi valorile predefinite pentru un joc.

La intrarea în activitatea jocului singleplayer, se creează un nou generator pentru numere aleatoare și un nou obiect SingleplayerGame care primește un Bundle care conține informațiile despre mapa de joc, iar acest obiect creat este folosit de metoda setContentView pentru a-l seta ca fiind conținutul vizibil în activitate.

La apăsarea butonului pentru modul multiplayer din prima activitate, se deschide o nouă activitate care gestionează camera de joc pentru a fi accesibilă de către ceilalți utilizatori. La deschiderea acestei activități, se verifică dacă există un cont de Firebase creat pentru utilizator, cont folosit pentru a putea scrie în baza de date. Dacă nu există un cont creat, atunci se încearcă crearea unui nou cont anonim. Pentru reușire sau eșuare în crearea unui cont, vor fi folosite mesaje Toast pentru a anunța utilizatorul. După crearea inițială a contului, acesta va fi folosit pentru fiecare joc creat de acum înainte pentru a scrie și a citi date.

```
private void signInAnonymously() {
    auth.signInAnonymously().addOnCompleteListener( activity: this, task -> {
        if (task.isSuccessful()) {
            // Sign in success
            Log.d(AUTH_TAG, SIGN_IN_SUCCESS);
            user = auth.getCurrentUser();
            Toast.makeText( context: MultiplayerCodeSetupActivity.this, SIGN_IN_SUCCESS,
                Toast.LENGTH_SHORT).show();
        } else {
            // If sign in fails
            Log.w(AUTH_TAG, SIGN_IN_FAIL, task.getException());
            Toast.makeText( context: MultiplayerCodeSetupActivity.this, SIGN_IN_FAIL,
                Toast.LENGTH_SHORT).show();
            super.onBackPressed();
        }
    });
}
```

Figura 3.58: Metoda signInAnonymously din activitatea MultiplayerCodeSetupActivity



Folosind un editText, se scrie un cod de cel puțin patru caractere care va fi folosit de toți utilizatorii participanți la joc. Există două butoane, unul pentru a accesa o cameră de joc deja creată, și unul pentru crearea acesteia. Când se creează un joc, se generează un nou obiect `ServerData` care va conține starea de joc (dacă încă așteaptă utilizatori sau au început), numărul folosit pentru generarea numerelor aleatoare și obiectul `playerData` care conține informațiile utilizatorului care a creat, acesta având întotdeauna id-ul `PLAYER1`. Informațiile acestea sunt scrise în baza de date la apăsarea butonului pentru creere. Când se încearcă intrarea într-o cameră deja existentă, se verifică dacă nu a fost atins numărul maxim de utilizatori, caz în care nu poate să aceseze, iar în caz opus, se citesc datele din baza de date, numărul pentru generatorul de numere este folosit pentru a crea obiectul respectiv și se identifică care din id-urile pentru jucători nu au fost ocupate. După ce se identifică, se creează un obiect `PlayerData` la id-ul selectat, iar acesta va fi folosit pentru a scrie informațiile caracterului utilizatorului.

Depinzând dacă utilizatorul a creat sau a accesat o cameră, se deschid activități diferite. Indiferent de acestea, amândouă au vizibil un obiect `ListView` folosit pentru a arăta utilizatorii din camera respectivă prin scrierea numelui jucătorului scris în prima activitate. Se folosește un ascultător pentru valorile nodului camerei astfel încât, când se modifică numărul de utilizatori, lista afișată se va modifica și ea. Se folosesc încă un listener pentru a urmări dacă valoarea din `gameState` din baza de date s-a schimbat, semnalând că a fost început jocul ceea ce determină deschiderea activității de joc și un listener pentru a urmări dacă nodul camerei încă există. Dacă nu mai există, înseamnă că cel care crease camera a închis-o, ceea ce determină întoarcerea la activitatea de setare a codului. În cazul activității celui care creează o cameră, aceasta conține aceleași câmpuri de setare a unei mape ca în singleplayer, lista anterior menționată, cele două butoane de începere și întoarcere și un buton suplimentar pentru a distribui codul camerei. Acest buton de împărțire poate fi folosit pentru a trimite la alți utilizatori codul pentru a intra în acest joc. La folosirea butonului de întoarcere, se șterg informațiile din baza de date pentru camera actuală, astfel putând fi creată o cameră nouă cu același cod. La apăsarea butonului de începere, se trimit parametrii pentru mapa de joc din `EditText`-uri la baza de date pentru a fi citite de ceilalți participanți și se schimbă valoarea din `gameState` pentru a determina schimbarea activității la cea de joc la toți cei care se află în activitatea celor care au accesat camera.

În activitatea de joc multiplayer, se citesc din baza de date informațiile despre dimensiunea mapei și șansa de apariție a cutiilor care sunt adăugate la bundle care este folosit pentru crearea obiectului de joc, iar acesta este setat ca fiind conținutul vizibil în activitate.



4. Utilizarea aplicației

Aplicația este realizată să ruleze cu telefonul în orientare orizontală, indiferent dacă are opțiunea de orientare automată pornită sau nu. La deschiderea aplicației, apare activitatea de selecție a modului de joc. Dacă este prima deschidere, câmpul care conține numele utilizatorului este liber. Acesta poate fi lăsat ca atare sau completat cu un nume care va fi folosit în jocurile cu mai multe persoane. După ce este completat și se intră în joc numele va rămâne salvat și se va afișa la intrarea ulterioară în aplicație. Numele rămas memorat poate fi schimbat ulterior, dacă se dorește.

Aici se selectează și modul de joc care este dorit apăsând pe unul din cele două butoane. Acestea duc către modul în care inamicii sunt inteligențe artificiale (Singleplayer) și respectiv modul de joc în care mai mulți utilizatori (Multiplayer) se conectează la același joc prin intermediul internetului, care trebuie să fie disponibil pe dispozitiv.



Figura 4.1: Meniul de deschidere

La intrarea în modul cu un singur jucător, utilizatorul are posibilitatea de a crea un joc în funcție de preferințele sale. Decide câți inamici să aibă, având opțiuni între unul, doi sau trei, și caracteristicile mapei de joc. Decide câte pătrate să aibă în înălțime și în lățime și cât de populată să fie de cutii de spart. La apăsarea butonului de începere a jocului, este creat un joc nou în funcție de valorile selectate. Dacă nu au fost modificate câmpurile, atunci se crează un joc cu mapa de dimensiune minimă și cu 50% cutii, dacă au fost completate, atunci se crează mapă după informațiile scrise. În caz de date invalide, se deschide un joc cu valori minime, iar dacă valorile trecute sunt mai mari decât cele posibile specificate, se folosesc dimensiunile maxime.





Figura 4.2: Opțiunile creării unui joc Singleplayer

La intrarea în modul cu mai mulți jucători, dacă este prima oară, utilizatorul este notificat de crearea unui cont. Acest cont va fi folosit pentru a identifica dispozitivul în jocurile online, fără a fi nevoie de intervenția utilizatorului.

Meniul care apare la intrarea în acest mod de joc oferă un câmp în care trebuie scris un cod. Codul este cel folosit pentru a aduce alt utilizator în joc. Există două opțiuni, cea de a te conecta la un joc folosind codul, sau de a crea un joc folosind un nou cod. Acesta trebuie să aibă minim patru caractere pentru a funcționa.

Când se încearcă crearea unui joc nou, dar codul este deja folosit, este înștiințat utilizatorul că nu poate folosi codul respectiv, iar dacă el încearcă să se alăture unui joc care deja a început, el este înștiințat că nu i se poate alătura din acest motiv.

Când a fost creat cu succes un joc cu un cod, utilizatorul intră în camera de așteptare. Aici are vizibili utilizatorii care s-au alăturat, are același câmpuri pentru a scrie preferințele sale pentru jocul pe care urmează să îl înceapă.

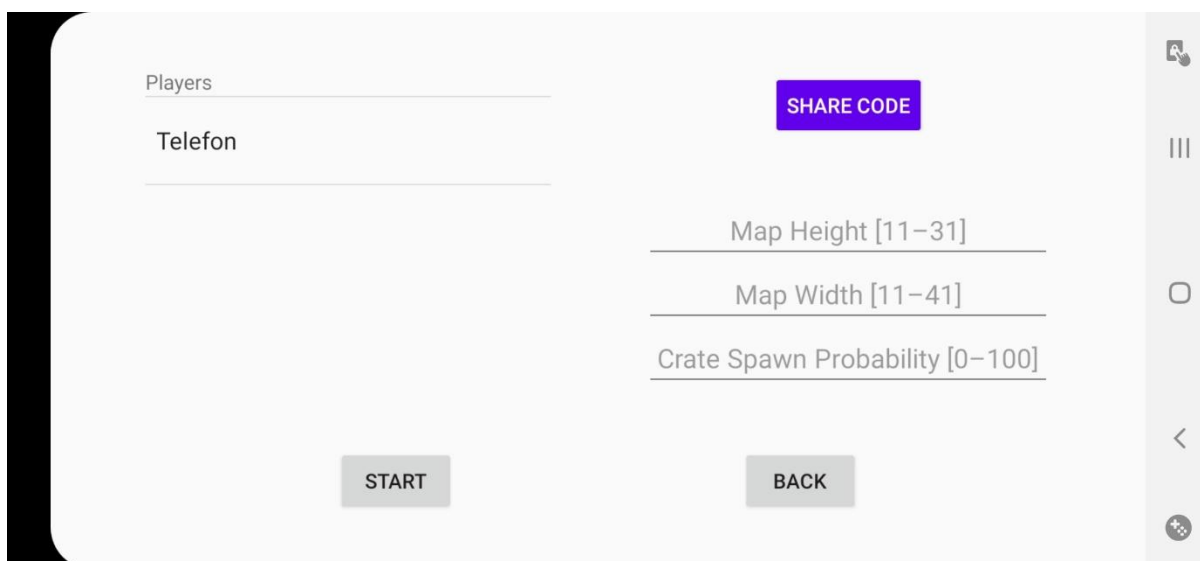


Figura 4.3: Opțiunile creării unui joc Multiplayer



Este un buton care poate fi folosit pentru a distribui codul prin intermediul altor aplicații pentru ca aceștia să se alăture jocului.

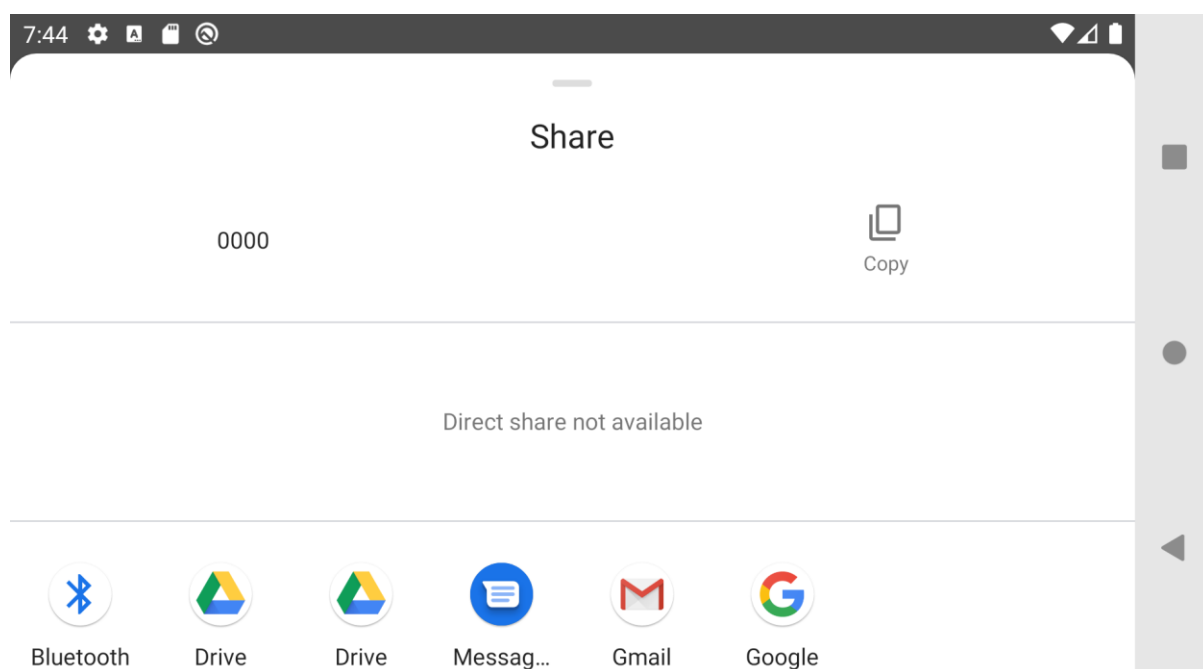


Figura 4.4: Opțiunile de distribuire a codului unui joc

Când sunt cel puțin doi utilizatori în cameră, creatorul camerei poate să folosească butonul pentru începerea jocului.

În cazul în care utilizatorul se alătură unui joc folosind un cod existent, el intră într-o cameră de așteptare care arată doar ceilalți utilizatori, jocul începând atunci când deținătorul codului decide să pornească jocul.

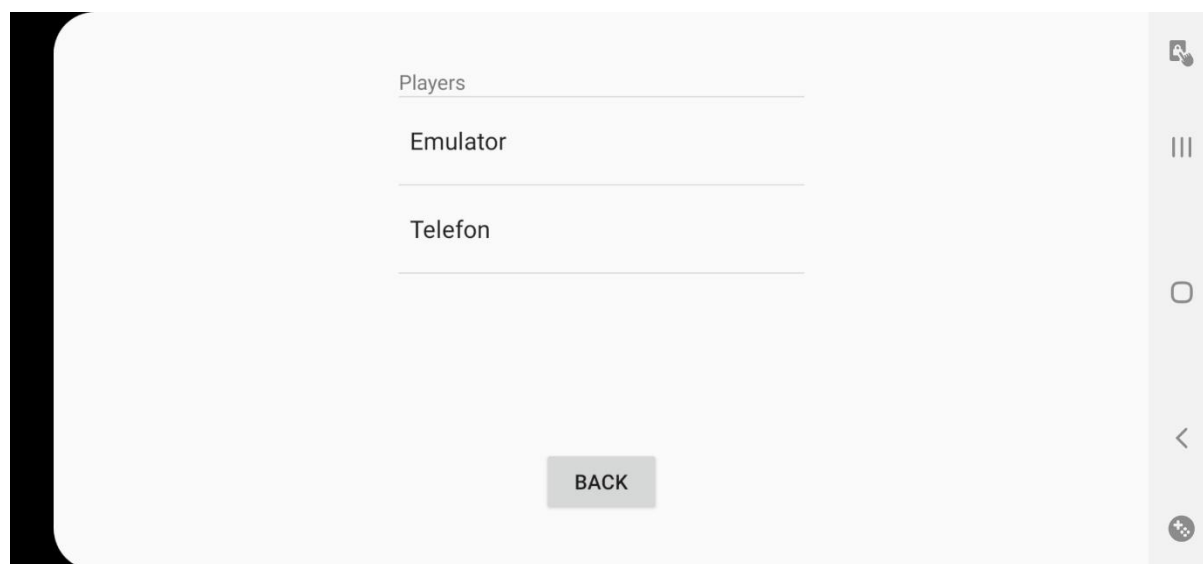


Figura 4.5: Activitatea de așteptare a pornirii jocului Multiplayer

Dacă cel care deținea codul decide să nu pornească jocul și se întoarce în meniu, ceilalți participanți sunt trimiși înapoi la partea de setare a unui cod de joc.



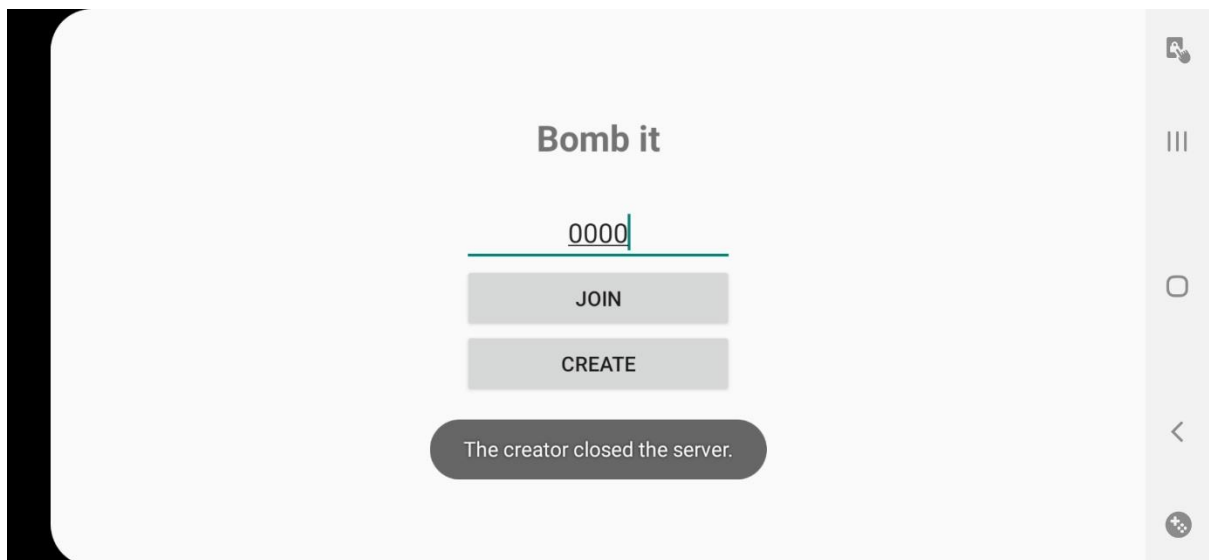


Figura 4.6: Jocul Multiplayer nu a început și a fost șters

Utilizatorul are două butoane folosite pentru a își controla caracterul. În partea stângă jos este o „manetă” folosită pentru a decide direcția și viteza de deplasare, dacă este apăsat înăuntrul cercului și deplasat spre exteriorul cercului, iar în partea dreaptă jos este un buton care la apăsare, se încearcă folosirea unei bombe în pătratul în care se află acum, iar aceasta va exploda, lăsând în urma ei o zonă care dacă este atinsă de un caracter, pierde o viață. Scopul este de a naviga pe harta de joc, să folosească bombe să spargă cutii pentru a elibera spațiu și a descoperi puteri care oferă avantaje caracterelor și să rămână ultimul în viață pe mapă.



Figura 4.7: Interfața jocului

Când mai rămâne doar un caracter în joc, deoarece ceilalți au rămas fără vieți, atunci este anunțat învingătorul. În cazul unui joc cu alți utilizatori, este scris și numele utilizatorului câștigător.

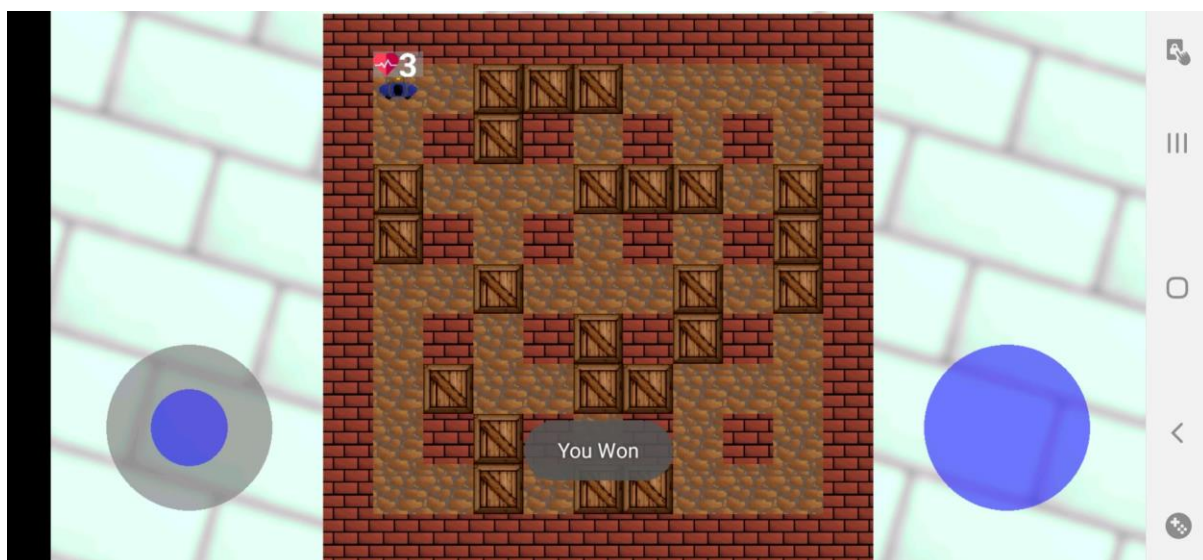


Figura 4.8: Utilizatorul a câștigat



Figura 4.9: Un inamic a câștigat



5. Concluzii, perspective de dezvoltare

5.1. Concluzii

Există o varietate de aplicații de divertisment disponibile pentru telefoanele mobile, cum ar fi jocuri, filme, muzică, podcast-uri, televiziune și reviste. Unele dintre cele mai populare aplicații de jocuri pentru telefoane mobile sunt Candy Crush, Angry Birds și Pokémon Go.

Jocurile simple de telefon sunt adesea cele mai apreciate datorită faptului că sunt ușor de înțeles și de jucat. Ele nu necesită multă abilitate sau cunoștințe pentru a fi înțelese și sunt adesea bazate pe concepte simple, cum ar fi puzzle-uri sau jocuri arcade. De asemenea, acestea sunt adesea accesibile pentru o varietate de vârste și niveluri de abilitate.

Aplicația realizată este un joc simplu, competitiv, ușor de înțeles, având un singur scop și un control intuitiv asupra caracterului. Aceasta a plecat de la jocul clasic „Bomberman”, care este popular și în prezent. Poate fi folosită pentru a crea competiții cu prieteni prin întremendiul modului Multiplayer de joc, sau de a îmbunătăți abilitățile de joc în modul Singleplayer.

5.2. Perspective de dezvoltare

Aplicația prezentată poate fi îmbunătățită prin implementarea unor opțiuni și variante de dezvoltare care oferă mai multe funcționalități și o experiență mai plăcută pentru utilizatori. Printre aceste posibilități de dezvoltare se numără:

- Crearea mai multor tipuri de mape de joc
- Adăugarea posibilității de a adauga prieteni și a îi invita la joc
- Crearea posibilității de a schimba culoarea avatarului tău
- Crearea un istoric a jocurilor terminate, vizibil utilizatorului
- Adăugarea mai multor tipuri de inamici pentru modul Singleplayer
- Antrenarea unui agent inteligent folosind metoda Reinforcement Learning¹⁰ pentru inamicii din modul Singleplayer
- Posibilitatea adăugării de inamici agenți inteligenți în jocurile Multiplayer
- Posibilitatea de a trimite mesaje în jocurile Multiplayer
- Adăugarea de sunete pentru joc
- Adăugarea mai multor tipuri de power-up-uri

¹⁰ Reinforcement Learning - metodă de învățare automată în care un agent inteligent învață prin experimentare și prin primirea unei recompense sau a unei penalități pentru acțiunile sale



5.3. Bibliografie

- Ken Arnold, James Gosling, David Holmes - THE Java™ Programming Language, Fourth Edition
- Peter Haggard - Practical Java: Programming Language Guide
- Said Hadjerrouit - Java as First Programming Language: A Critical Evaluation
- Timothy Budd - Understanding Object-oriented Programming with JAVA
- Joyce Farrell - Java™ Programming
- Awal Kharisma - What is Android?
- Robi Grgurina, Goran Brestovac and Tihana Galinac Grbac - Development Environment for Android Application Development: an Experience Report
- Donn Felker - Android Application Development For Dummies
- Ted Hagos - Android Studio
- Jason Ostrander - Android UI Fundamentals: Develop and Design
- Jerome DiMarzio - Beginning Android Programming with Android Studio
- Anirban Sarkar, Ayush Goyal, David Hicks, Debadrita Sarkar, Saikat Hazra - Android Application Development: A Brief Overview of Android Platforms and Evolution of Security Systems
- Sujit Kumar Mishra - Fundamentals of Android App Development: Android Development for Beginners to Learn Android Technology, SQLite, Firebase and Unity (English Edition)
- Chunnu Khawas, Pritam Shah - Application of Firebase in Android App Development-A Study
- Kumar S. Ashok - Mastering Firebase for Android Development
- Ashok Kumar S - Mastering Firebase for Android Development: Build real-time, scalable, and cloud-enabled Android apps with Firebase
- Bill Stonehem - Google Android Firebase: Learning the Basics
- Daniel Silber - Pixel Art for Game Developers
- Jonathan Cooper - Game Anim: Video Game Animation Explained
- Chris Solarski - Drawing Basics and Video Game Art: Classic to Cutting-Edge Art Techniques for Winning Video Game Design
- Ricardo Queirós - A survey on Game Backend Services
- Saiqa Aleem, Luiz Fernando Capretz & Faheem Ahmed - Game development software engineering process life cycle: a systematic review
- Mark J. P. Wolf - The Medium of the Video Game
- Scott Rogers - Level Up! The Guide to Great Video Game Design
- Ícaro Goulart, Aline Paes, Esteban Clua - Learning How to Play Bomberman with Deep Reinforcement and Imitation Learning
- <https://www.gamedesigning.org/learn/multiplayer/>
- <https://dev.to/ibmdeveloper/multiplayer-server-basics-ep-1-creating-a-multiplayer-game-server-5aed>
- <https://www.oreilly.com/library/view/opengl-game-development/9781783288199/ch01s02.html>

