SZAKDOLGOZAT

Kovács Zsolt

Debrecen
2010

Debreceni Egyetem
Informatikai Kar

Mobiltelefonos játékfejlesztés

Témavezető:
Bátfai Norbert
egyetemi tanársegéd

Készítette:
Kovács Zsolt
programtervező informatikus

Debrecen
2010

Tartalomjegyzék

1. Bevezetés

A mobiltelefonok hozzátartoznak a mindennapjainkhoz. Hazánkban is szinte mindenki rendelkezik legalább eggyel, a fiataltól az idős korosztályig. Magyarországon 2010 februárjában a 100 főre eső aktív mobiltelefon előfizetések száma 118,7 volt[1]. Ezek az előfizetések a lakosság 93,5 %-a közt oszlanak el[2]. A 65–74 évesek kivételével (74%) minden korcsoport (16-64 évesek) mobilhasználati aránya meghaladja a 90%-ot. A mobilhasználók alsó korhatára azonban valójában ennél jóval alacsonyabb, Finnországban például a gyerekek tipikusan 7 éves korukban kapják meg első mobiltelefonjukat[3].

A mobiltelefonokkal közel egy időben jelentek meg a rájuk készült játékok. Az első mobiltelefonos játék 1997-ben a Nokia 6110 készülékbe épített Snake volt. A következő mérföldkövet a WAP játékok jelentették. Az igazi áttörést azonban a Java ME technológia hozta 2002-ben. A riválisait leküzdve, hamar meghódította a piacot, 2008-ban az eladott mobiltelefonok 80%-a támogatta a Java ME valamelyik verzióját[4].

Maguk az eszközök is jelentős fejlődésen mentek keresztül, mára már jelentős számítási kapacitással és multimédiás képességekkel rendelkeznek. A legtöbb mobiltelefon képes internetezésre, zene lejátszására, képek/videók megjelenítésére, beépített kamerával utóbbiak rögzítésére. A játékok számára a legfőbb korlátot az eszközök jellegéből adódó kis képernyő, és a korlátozott adatbeviteli lehetőségek jelentik.

A mobil-játékok tipikus felhasználói az úgynevezett alkalmi játékosok[5], akik csak időnként, kedvtelésből vagy unaloműzésből fordulnak a játékokhoz. Az alkalmi játékosok 74%-át a nők adják, tipikusan a 30-45 éves korosztályból, a nemek és korosztályok eloszlása azonban játékstílusonként erősen változó. Az alkalmi játékosokra általánosan jellemző hogy:

- Nem szeretik az erőszakos játékokat
- A könnyen tanulható, intuitív irányítással rendelkező játékokat kedvelik
- A rövid intervallumokban is játszható játékokat preferálják

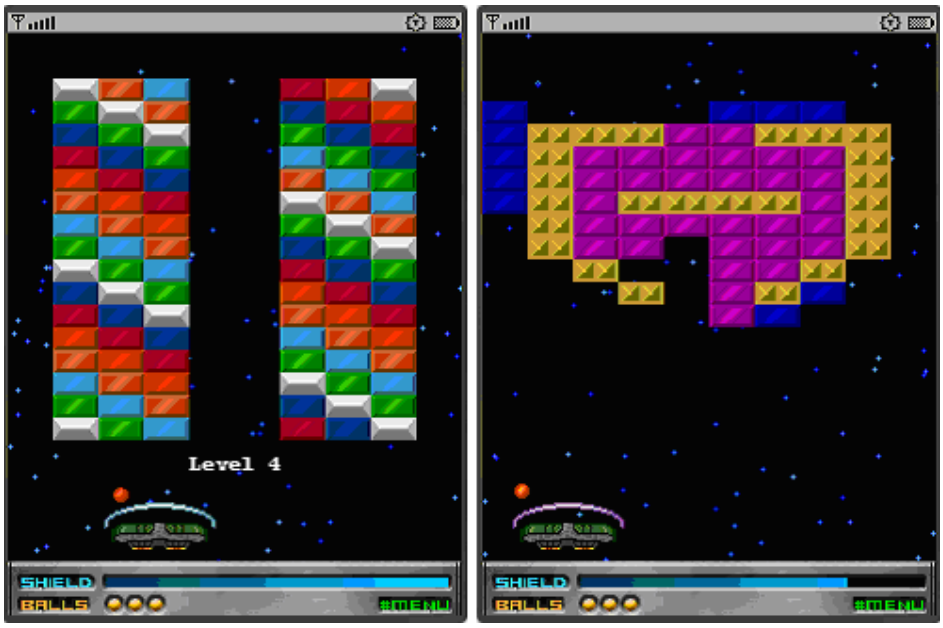
Ebből kifolyólag a mobil játékokkal szemben támasztott elvárások[5][10] nagyban eltérnek a „komoly” számítógépes játékokétól, azonban ez szerencsésen párosul a mobil eszközök korlátozásaival, mivel ezek az egyszerű játékok nem igényelnek bonyolult irányítást vagy számításokat.

Dolgozatom célja egy ilyen egyszerű, szórakoztató játék elkészítése, az alkalmazott technológiák, módszerek bemutatása.

1.1 A játék rövid bemutatása

A játék Arkanoid stílusú, azaz a játékos egy űrhajót mozgat a képernyő alján balra/jobbra, felette téglák találhatóak, amelyeket a pattogó labda segítségével össze kell törni. A játékos feladata, hogy az űrhajó mozgatásával meggátolja, hogy a labda a képernyő alján kiessen, visszapattintsa azt úgy, hogy minél több téglát eltaláljon. Amennyiben a labda mégis eléri a képernyő alját, a játékos elveszít egy életet. Ha minden élet elfogyott, a játék sikertelenül ér véget. Ha az összes tégla összetört, az adott pályát megnyertük, a játék a következő pályán folytatódik.

A labda visszapattintásához az űrhajó pajzsát használjuk, amely ívelt, így precíz célzás lehetséges, azonban korlátozott energiával rendelkezik. Időnként egy tégla összetörésekor egy bónusz jön létre, melyet az űrhajóval érintve az adott típusú bónusszal járó előnyhöz jutunk, mint például extra élet vagy pajzs energia.



Két kép a játékból

1.2 A Java technológia

A Java technológia 1991-ben indult útjára, és mára az informatika egyik meghatározó szereplőjévé vált. Megalkotásakor a cél egy egyszerű, platform-független, objektum-orientált, biztonságos nyelv elkészítése volt, mai sikerét ezen, és számos további jó tulajdonságának köszönheti.

A platform-függetlenség miatt a Java eredetileg egy interpreteres nyelv. Ez azt jelenti, hogy a program nem natív, az adott processzor által értelmezett kódra lesz lefordítva, hanem egy közbülső, bájtkódnak nevezett, a Java virtuális gép (JVM) által értelmezett kódra alakítjuk. A virtuális gép réteget képez a Java program és az adott gép platformja közt, egy absztrakt számítógép, a valódiakhoz hasonlóan rendelkezik utasításkészlettel, kezeli a memóriát, vezérli a végrehajtást, garantálja a biztonságot. Egy program futásakor a JVM értelmezi a bájtkódot, és végrehajtja a megfelelő utasításokat. A megoldás hátránya a lassúsága, amely a Java-ellenes előítéletek alapját képezte. A lassúság kiküszöbölésére a just-in-time (JIT) fordítási technikát alkalmazzák, melynek lényege hogy a bájtkódot futás közben natív kódra fordítja, ezáltal jelentősen növelve a program sebességét.

A különböző felhasználási területekhez Java Platformokat hoztak létre, melyek a nyelv bővített vagy épp szűkített változatai. A szerverek és vállalati alkalmazások igényeit a Java EE platform elégíti ki, az erre írt programok keretrendszerek segítségével készülnek, tipikusan szervereken futnak. A személyi számítógépeket a Java SE fedi le, ezek a programok a felhasználó gépén futnak, többnyire saját ablakban, jellemzően Java EE programok vastagklienseként, vagy önálló alkalmazásként. A korlátozott kapacitású illetve kisebb hordozható eszközök - mint például a mobiltelefonok - platformja a Java ME, a következő fejezetben ezzel ismerkedünk meg részletesebben. Ezek a Java legfőbb alkalmazási területei, azonban számos további helyen is alkalmazzák, ilyen például a Lego Mindstorm robotok. Többek között erről is lehet olvasni [11]-ben, melynek társszerzője vagyok.

2. Java Micro Edition (Java ME)

A platform eredetileg abból a célból készült, hogy a kis méretű, korlátozott kapacitású eszközökön is lehessen Java programokat futtatni. A Java ME platform valójában technológiák és specifikációk egy halmaza, melyeket összekombinálva kapunk egy teljes Java futtató-környezetet, amely jól igazodik az adott eszköz képességeihez. Ezzel a moduláris technológiával különböző eszközök széles skálája fedhető le.

A platform virtuális gépe a Kilobyte Virtual Machine (KVM) amely nevét a néhány kilobyte-os méretéről kapta, modulárisan bővíthető, és akár egy 16 bites 16MHz órajelű processzoron és 196 KB memórián is fut.

A specifikációk Java Specification Request (JSR)-ek formájában jönnek létre, a Java Community Process (JCP) eredményeként. A JCP egy eljárás mely folyamán az érdekelt felek döntenek a platform jövőjéről.

2.1 Konfigurációk, profilok, opcionális csomagok

A futtatási környezet alapját a konfiguráció adja, amely meghatározza az elérhető nyelvi könyvtárakat, JVM funkciókat. A profil valójában API-k együttese, melyek eszközök egy szűk halmazát támogatja az adott konfiguráció keretein belül. Az opcionális csomagok olyan funkcionalitásokat adnak melyek nem részei a profilnak, de az adott eszköz támogat. Ezek kombinációja egy olyan futtatási-környezetet eredményez, amely az adott eszközre optimalizált, annak képességeit jól kihasználja.

2.1.1 CDC (Connected Device Configuration)

A CDC a nagyobb teljesítményű, hálózati kapcsolattal rendelkező eszközöket (például kommunikátorok, PDA-k) célzó konfiguráció. Egy tipikus CDC-t támogató eszközben 32 bites mikroprocesszor, legalább 2MB RAM, 2.5 MB ROM található

A konfiguráció a Java SE könyvtárain alapszik, de a mobil eszközök korlátozásait is figyelembe veszi. A konfigurációhoz három profil létezik:

- Foundation Profile (JSR 219)

  Minimális profil, alapvető szolgáltatásokat tartalmaz, mint például hálózatkezelés és I/O, nem tartalmaz grafikus, illetve GUI eszközöket

- Personal Basis Profile (JSR 217)

  Tartalmazza a teljes Foundation Profile-t, emellett pehelysúlyú komponenseket és Xlet eszközrendszert is tartalmaz

- Personal Profile (JSR 216)

  Tartalmazza a teljes Personal Basis Profile-t, illetve teljes AWT és korlátozott bean-támogatást is

Az elérhető opcionális csomagok:

- RMI

  A Java SE RMI könyvtár egy szűkebb változata, távoli metódushívást megvalósító eszközrendszer

- JDBC

  A JDBC 3.0 API szűkített változata, adatbázisok kezelését segítő eszközrendszer

2.1.2 CLDC (Connected Limited Device Configuration)

A CLCD a korlátozott kapacitású eszközöket (például mobiltelefonok, személyhívók) célzó konfiguráció. Egy CLCD-t támogató eszközben 16 vagy 32 bites, legalább 16MHz órajelű processzor, legalább 192 KB memória és 160 KB nem felejtő memória van. Jellemzően alacsony fogyasztású, sokszor elemmel/akkumlátorral működő eszközök, korlátozott sávszélességű vezetékes vagy vezeték nélküli hálózati kapcsolattal.

Az aktuális verzió az 1.1-es, amely az eredeti specifikációt többek között lebegőpontos aritmetikával és gyenge referenciával bővítette, lefelé kompatibilis az 1.0-s szabvánnyal.

A konfiguráció egyetlen profilja a MIDP.

2.1.3 MIDP (Mobile Information Device Profile)

2009 decemberében jelent meg a 3.0-ás specifikáció, amely számos újítást és fejlesztést fog hozni, többek között fejlett eseménykezelést, MIDletek közti kommunikációt, LIBleteket, új multimédia API-t, fejlett RMS-t, továbbfejlesztett grafikus API-t és több kijelző támogatását. Fontos újítás hogy a MIDP 3.0 a CDC konfiguráción is támogatott lesz, azonban most még egyetlen eszköz sem támogatja a profilt. Jelenleg a MIDP 2.1-es profil az elterjedt, a játék is ehhez készült, a továbbiakban erről lesz részletesebben szó.

A profilhoz számos opcionális csomag létezik, melyek többek között 3D, Bluetooth, multimédia, SMS támogatást nyújtanak.

2.2 MIDP API

A Mobile Information Device Profile (MIDP) a CLDC-vel együtt alkotja a teljes Java futtatókörnyezetet a mobiltelefonok számára. Az elérhető opcionális csomagok az adott eszköztől függnek, típusonként igen eltérő lehet.

2.2.1 MIDletek

A MIDP környezethez készült alkalmazásokat MIDleteknek nevezzük. A MIDleteket az Application Management Software (AMS) kezeli, a MIDletek telepítése, futtatása, törlése az AMS-en keresztül történik.

A MIDleteket egy szabványos JAR (Java Archive Resource) fájlba csomagolják, amely valójában egy ZIP fájl, amely tartalmazza a bájtkódot, és az erőforrásokat (képek, hangok, adatok …stb.). Ha egy JAR-ban több MIDlet is van, akkor *MIDlet Suite*-ról beszélünk, ezek általában hasonló vagy együttműködő MIDletek, amelyek osztoznak az erőforrásokon. Egy MIDlet csak a saját suite-jában lévő erőforrásokat éri el, más suite-ok erőforrásait nem.

A JAR tartalmaz egy manifest fájlt, amely a futtatáshoz szükséges, illetve egyéb információkat tartalmaz a MIDletről. Hasonló információkat tartalmaz a JAR-hoz mellékelt JAD (Java Application Descriptor) fájl, amely alapján az AMS el tudja dönteni, hogy az adott eszközön futtatható-e az alkalmazás, még az előtt hogy a JAR-t letöltené, mivel a JAD fájl a JAR-al ellentétben kisméretű.

Minden MIDlet tartalmaz egy `javax.microedition.midlet`-ből származó osztályt. Ez vezérli az alkalmazás működését, az alkalmazás indításakor az AMS létrehozza ennek az osztálynak egy példányát.

Egy MIDletnek három állapota lehet: felfüggesztett, aktív, leállított. Indításkor felfüggesztett állapotban van a MIDlet, majd az AMS a `MIDlet`.*startApp()* metódus hívásával aktívvá teszi, ekkor kezdődik el ténylegesen az alkalmazás futása, erőforrásokat foglal le, és elkezdi a működését.

Futás közben külső események, mint például bejövő telefonhívás hatására az AMS meghívhatja a `MIDlet`.*pauseApp()* metódust, ekkor a MIDlet felfüggesztett állapotba kerül. Ekkor a programnak minden tevékenységet fel kell függeszteni, és bár a program futása csak

ideiglenesen szakad meg, de a startApp-ban lefoglalt erőforrásokat ilyenkor célszerű felszabadítani. A program folytatásához ismét a midlet.startApp fog meghívódni. A MIDlet felfüggesztett és aktív állapot közt akárhányszor válthat, akár önként is, a `MIDlet.`*notifyPaused()* hívásával. A felfüggesztett MIDlet kérheti az AMS-től az aktiválást a `MIDlet.`*resumeRequest()* hívásával.

A program leállításához az AMS a `MIDlet.`*destroyApp(…)* metódust hívja, ekkor a program elmenti a szükséges adatokat, és felszabadítja a lefoglalt erőforrásokat. A MIDlet leállított állapotba kerül, innen már más állapotba nem kerülhet. A program önként is leállhat, ezt a `MIDlet.`*notifyDestroyed()* metódus hívásával teheti meg, ekkor a *destroyApp(…)* metódus már nem hívódik meg.

2.2.2 LCDUI

A kisméretű kijelző és a korlátozott erőforrások miatt az AWT és Swing API-k alkalmatlanok a mobiltelefonok felhasználói felületének megvalósítására. További probléma hogy az egyes eszközök eltérően közelítik meg a felhasználói felület kérdését, így az LCDUI egy absztrakt eszközrendszert ad a probléma megoldására, mely futáskor az adott eszközhöz illő felületet eredményez majd.

2.2.2.1 Alapok

Az API központi fogalma a képernyő. Az alkalmazás futása során képernyőről képernyőre halad, egyszerre mindig csak egy látható. A kijelzőt (`Display`) a `MIDlet.`*getDisplay()* hívásával érhetjük el, majd a `Display.`*setCurrent(Displayable d)* metódussal jelenítjük meg az adott képernyőt a kijelzőn. A `Displayable` osztály abstract, leszármazottjai a `Screen` és a `Canvas`. A `Canvas` alacsony szintű támogatást nyújt, a játékok ezt terjesztik ki az egyedi tartalom implementálásához.

A `Screen` leszármazottjai magas szintű eszközrendszert nyújtanak a felhasználói felület implementálására:

- Figyelmeztetés (`Alert`) – egyszerű szöveges üzenetet ír ki
- `Form` – összetett tartalom megjelenítését valósítja meg
- Lista (`List`) – elemek egy listájából választhatunk egy vagy több elemet
- Szövegdoboz (`TextBox`) – többsoros szöveg szerkesztésére alkalmazható

2.2.2.2 Form-ok

Bonyolultabb képernyők megalkotására alkalmazható. A `Form` létrehozásakor üres, az *append* metódussal adhatunk hozzá `Item` leszármazottakat, ezek elhelyezéséről a képernyőn a `Form` dönt(általában egy egyszerű függőleges lista):

- `StringItem` – nem interaktív szöveg
- `ImageItem` – nem interaktív kép
- `TextField` – egysoros szöveg bevitelét teszi lehetővé
- `ChoiceGroup` – alternatívák közüli választást valósítja meg
- `DateField` – dátum kiválasztó
- `Gauge` – csúszka/állapotsáv

2.2.2.3 Események, parancsok

Míg a `Canvas` közvetlen hozzáférést biztosít a billentyűzethez, addig a `Screen` csak eseményeken és parancsokon keresztül értesül a felhasználó tevékenységéről.

Az események az egyes elemek (`Item`) állapotának megváltozásáról tájékoztatnak. Ehhez regisztrálni kell egy `ItemStateListener`-t implementáló példányt az adott elemen.

A parancsok az adott képernyőhöz rendelt eseményekről értesítenek. Maguk a parancsok általában a kijelző alján szöveges formában jelennek meg. Az események/parancsok feldolgozásához regisztrálni kell egy `CommandListener`-t implementáló példányt az adott `Displayable` példányon. Az egyetlen kivétel az `Alert`, melyhez nem adható parancs.

2.2.3 Perzisztencia

Adatok hosszú távú tárolására kínál lehetőséget a Record Management System (RMS). Az így tárolt adatok az alkalmazásunk befejezése, a telefon kikapcsolása után is megmaradnak. Az adatok a MIDlet suite-hoz kapcsolt, névvel rendelkező tárolókban (`RecordStore`) helyezkednek el. Több tároló is létezhet, ezek a suite minden MIDlet-jéből elérhetőek, de más suite-okból nem. A suite törlésével az adatok is törlődnek. A tárolóban az adataink bájttömbként tárolódnak, elérni őket a rekordazonosítóval tudjuk, ami egy sorszám. A rendelkezésre álló tárterület néhány tíz kilobájt és több megabájt között van eszköztől függően.

2.2.4 Multimédia

A MIDP profil tartalmazza a Mobile Media API (MMAPI) opcionális csomag egy egyszerűsített változatát. Ez audiofájlok lejátszását és rögzítését teszi lehetővé, míg a teljes csomag videofájlok kezelésére is alkalmas. További funkcionalitást ad az Advanced Multimedia Supplements (JSR 234) opcionális csomag.

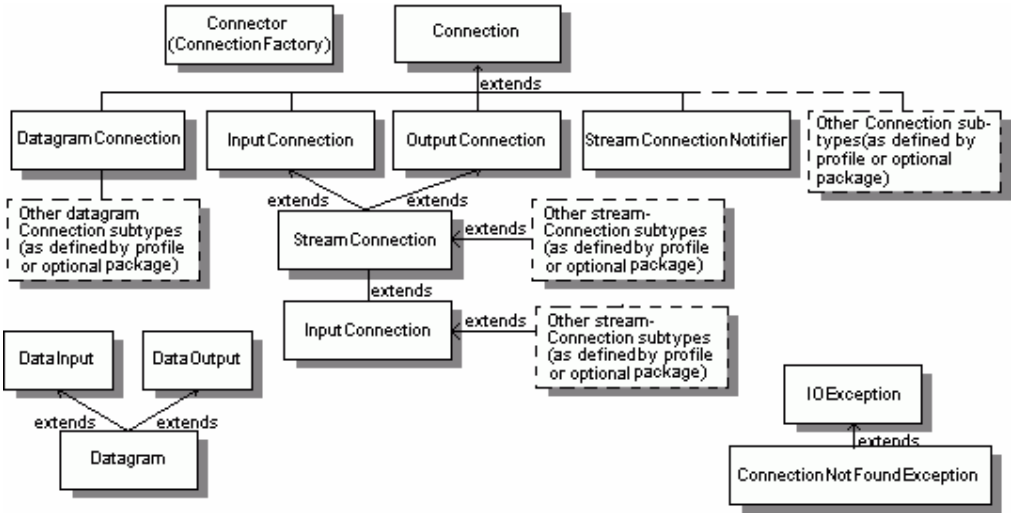
Az API középpontjában a `Player` interfész áll, ez vezérli a lejátszást. Az egyes `Player` implementációk egy adott típusú adatfolyamot képeznek le hanggá. `Player` példányt a `Manager`-en keresztül kérhetünk. A `Player` működését befolyásolhatjuk a `Control` segítségével, pl. hangerő, hangszín módosítása.

A `Player`-nek 5 állapota van: UNREALIZED, REALIZED, PREFETCHED, STARTED, és CLOSED. Létrehozás után UNREALIZED állapotban van, a *realize()* metódus hívásakor inicializálja az erőforrás lefoglalásához szükséges adatokat, és REALIZED állapotba kerül. A *prefetch()* metódus hívásakor a `Player` lefoglalja a szükséges erőforrásokat, inicializálódik, és PREFETCHED állapotra vált. A *start* metódus hívásakor STARTED állapotra vált, és megkezdi a lejátszást. Az adatfolyam végén visszavált PREFETCHED állapotba. A *close()* metódus hívásakor felszabadulnak az erőforrások és a `Player` CLOSED állapotba vált, ezután már nem használható.

A támogatott formátumok eszközfüggőek, a profil csak a tone generálást követeli meg. A JSR 185 kötelezővé teszi a wav és MIDI támogatását is. A támogatott formátumok listáját futás közben lekérhetjük a `Manager`.*getSupportedContentTypes(…)* metódussal.

2.2.5 Hálózatok

A hálózati kommunikációt egy általános keretrendszer, a Generic Connection Framework (GCF) valósítja meg. Az általános kapcsolattípust a `Connection` interfész írja le, ezt terjeszti ki többi interfész, melyek a hierarchiában lefele haladva egyre specializáltabbak. Kapcsolatok létrehozása a `Connector` osztályon keresztül történik URL segítségével.



A GCF osztály-hierarchia

Négyféle kapcsolattípust támogat a MIDP profil, ezek közül azonban csak a Http implementációja kötelező, a többi opcionális:

- Http
- Datagram (UDP)
- Socket (TCP)
- Comm (soros port)

A keretrendszer erőssége hogy tetszőlegesen bővíthető. Példa erre a BlueTooth támogatás (JSR 82). Az API sikerességét mutatja, hogy a CDC konfigurációban és a Java SE platformon is megjelent.

2.2.6 Game API

A MIDP 2.0-ás verzióban jelent meg, célja a játékfejlesztés megkönnyítése, szabványos eszközrendszer biztosítása a gyakori feladatokhoz[8].
Csomagja a javax.microedition.lcdui.game amely 5 osztályt tartalmaz, ezek:

- `GameCanvas`: Double-Buffering technikát megvalósító vászon. Ennek segítségével egyenletesen mozgó animációkat jeleníthetünk meg. Lényege hogy a *getGraphics()* metódussal lekért grafikai objektumra rajzolás egy bufferbe történik, majd a *flushGraphics()* hatására a teljes buffer átmásolódik a kijelzőre. Az osztály másik fontos feladat a billentyűzet kezelése.
- `Layer`: absztrakt osztály, a gyakorlatban a leszármazottjait használjuk, ezek a `Sprite` és a `TiledLayer`.
- `Sprite`: egy grafikus elemet megvalósító osztály. Legfontosabb funkciója az animációk támogatása. A konstruktorában megadott képet képes képkockákra felbontani. A képkockák sorfolytonosan kell hogy elhelyezkedjenek a képen, akár több sor és/vagy oszlopban. Ez alapján a képkockák egy 0-tól induló sorszámot kapnak, ezzel hivatkozhatunk rájuk az animáció megadásakor. Emellett támogatja a referencia pixel körüli forgatást és tükrözést, illetve más `Sprite`-okkal való ütközés detektálását, akár az átlátszó pixelek figyelembevételével is.
- `TiledLayer`: csempézett réteget megvalósító osztály. A konstruktorban megkapott képet a `Sprite`-hoz hasonlóan fel tudja darabolni, azonban a képkockák sorszáma itt egytől indul, és a réteg felépítésére használja fel őket. Úgy lehet elképzelni, mint képek egy táblázatát. Azt hogy hol melyik képkocka jelenjen meg, egy integer tömb adja meg, amely a képkockák sorszámait tartalmazza. Ahol 0 van, ott üres marad a réteg. Van lehetőség az egyes csempék animálására is.
- `LayerManger`: `Layer` példányok együttes kezelésére szolgáló osztály. A *paint()* metódusának meghívásakor kirajzolja az összes hozzárendelt `Layert`, nem kell külön-külön kirajzolni azokat. A kirajzolási sorrendet alapesetben az elemek hozzáadásának sorrendje határozza meg, de van lehetőség tetszőleges helyre is beszúrni a kirajzolási sorban.

3. A játék részletes elemzése

3.1 Tervezés

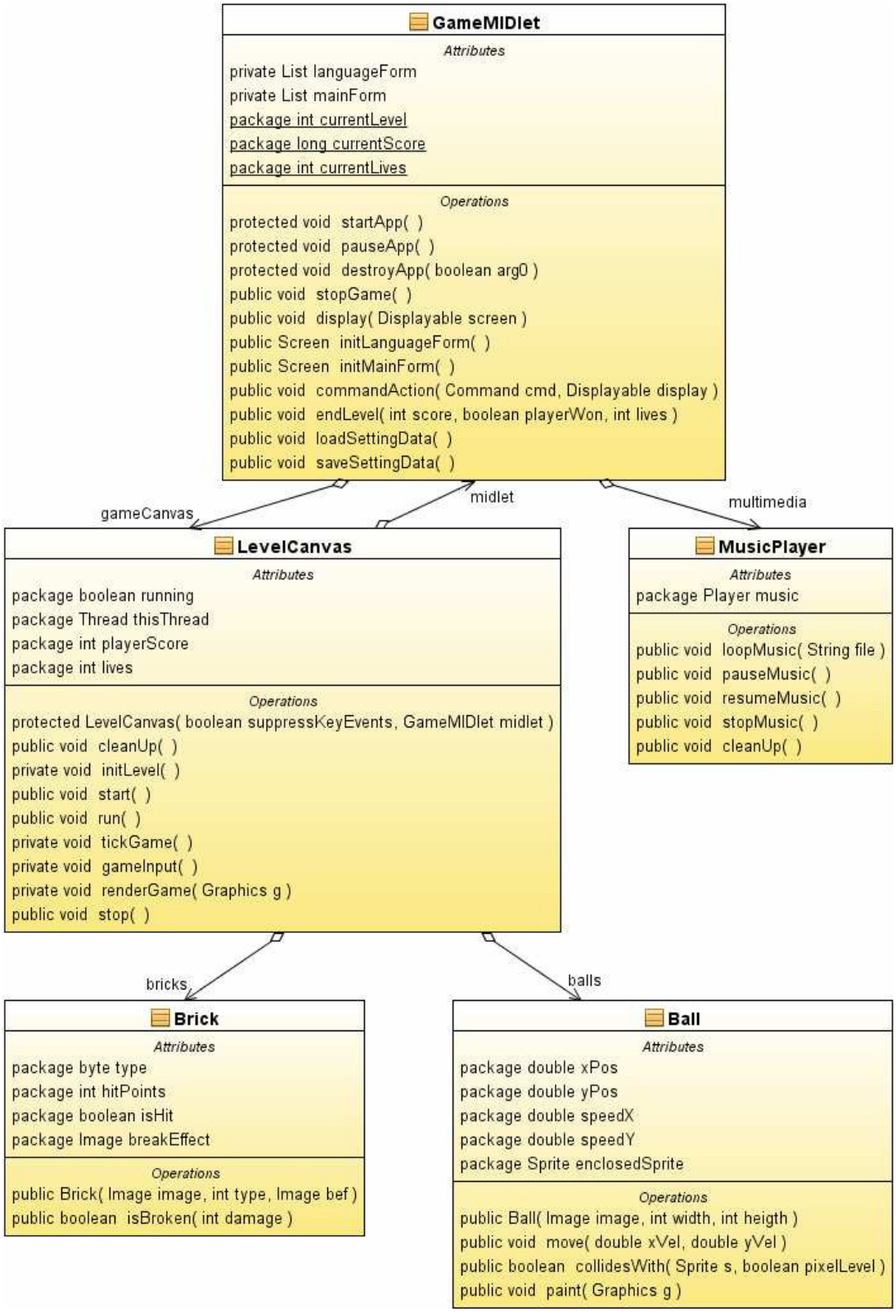
A szoftverfejlesztés első lépése a tervezés. Először meg kell fogalmazni a követelményeket, hogy mit várunk el a programtól. Ez jellemzően egy szöveges dokumentum, Java ME játékok esetén ez általában igen rövid, a játékok egyszerűségéből adódóan.

A követelmények felállítása után meg kell vizsgálni azok megvalósíthatóságát. A mobiltelefonok képességei igen szűkösek, a túl bonyolult és számításigényes funkciók, még ha elméletben szórakoztatónak is tűntek, mit sem érnek, ha a lassúságukkal játszhatatlanná teszik a játékot, ezért törekedni kell az egyszerűségre. A megvalósíthatóság mellett arra is oda kell figyelni, hogy az egyes funkciók és azok összhatása pozitív hatással van-e a játékmenetre. Ezt nem mindig lehet előre tudni, így előfordulhat, hogy fejlesztés közben derül ki, hogy a játék egyes elemei túl unalmasak vagy frusztrálóak, ekkor ezeket újra kell tervezni, vagy el kell távolítani.

A következő lépése a fejlesztésnek az architektúra megtervezése, amely meghatározza a program osztályait és azok kölcsönhatását. Ezt jelentősen megkönnyíti az UML diagrammok használata, melyek segítségével könnyen és gyorsan el lehet készíteni az architektúra tervét. A Java ME játékok esetén ennek alapja kötött, mivel a MIDlet API szükségessé teszi egy `MIDlet`-ből származó osztály használatát. Az egyedi, interaktív tartalom implementálásához a `Canvas` vagy `GameCanvas` osztályok egyikét kell mindenképp származtatni, így a program alapjául szolgáló két osztály rögzített. Ezek köré kell építeni az architektúránkat, lehetőleg minimális számú osztályból.

A tervezést követően jön az implementálás, amely tipikusan az evolúciós modellt követi. Ennek lényege, hogy elkészítünk egy prototípust, amely az alapvető funkciókat tartalmazza csak, majd ezt bővítjük az egyes verziókban, míg végül eljutunk a kész programhoz. A köztes verziók egy-egy funkcióval bővítik a megelőző verziót, így azok külön-külön kipróbálhatóak, ha sikertelennek bizonyulnak, a verzió eldobásával törölhetőek. [7]-ben részletesen olvashatunk a prototípusok használatáról a játékfejlesztésben.

A fejlesztés folyamán célszerű SVN eszközöket alkalmazni, amelyek megkönnyítik a verziókezelést. Az evolúciós modell miatt ez különösen indokolt is, mivel a sok verzió közt könnyű elveszni.



A főbb osztályokat tartalmazó osztály-diagramm

3.2 Menürendszer

A játék elindításakor először a menüvel szembesül a felhasználó. A kezdőképernyőn a támogatott nyelvek közül kell választani. A nyelv kiválasztása után megjelenik a főmenü, amely egy egyszerű lista, az alábbi menüpontokat tartalmazza:

- Folytatás

  Ha a játék közben térünk vissza a főmenübe, akkor azt folytathatjuk, egyébként a program legutóbbi futtatásakor félbehagyott pályát kezdhetjük újra.

  Ha nincs folytatható játék, akkor a menüpont nem jelenik meg.
- Új játék

  Új játékot indíthatunk, a pályaválasztó képernyőt jeleníti meg.
- Beállítások

  A beállítások képernyőt jeleníti meg, ahol a zenét kapcsolhatjuk be/ki illetve beállíthatjuk annak hangerejét.
- Dicsőségtáblázat

  A játék során elért rekord-pontszámok táblázatát jeleníti meg.
- Segítség

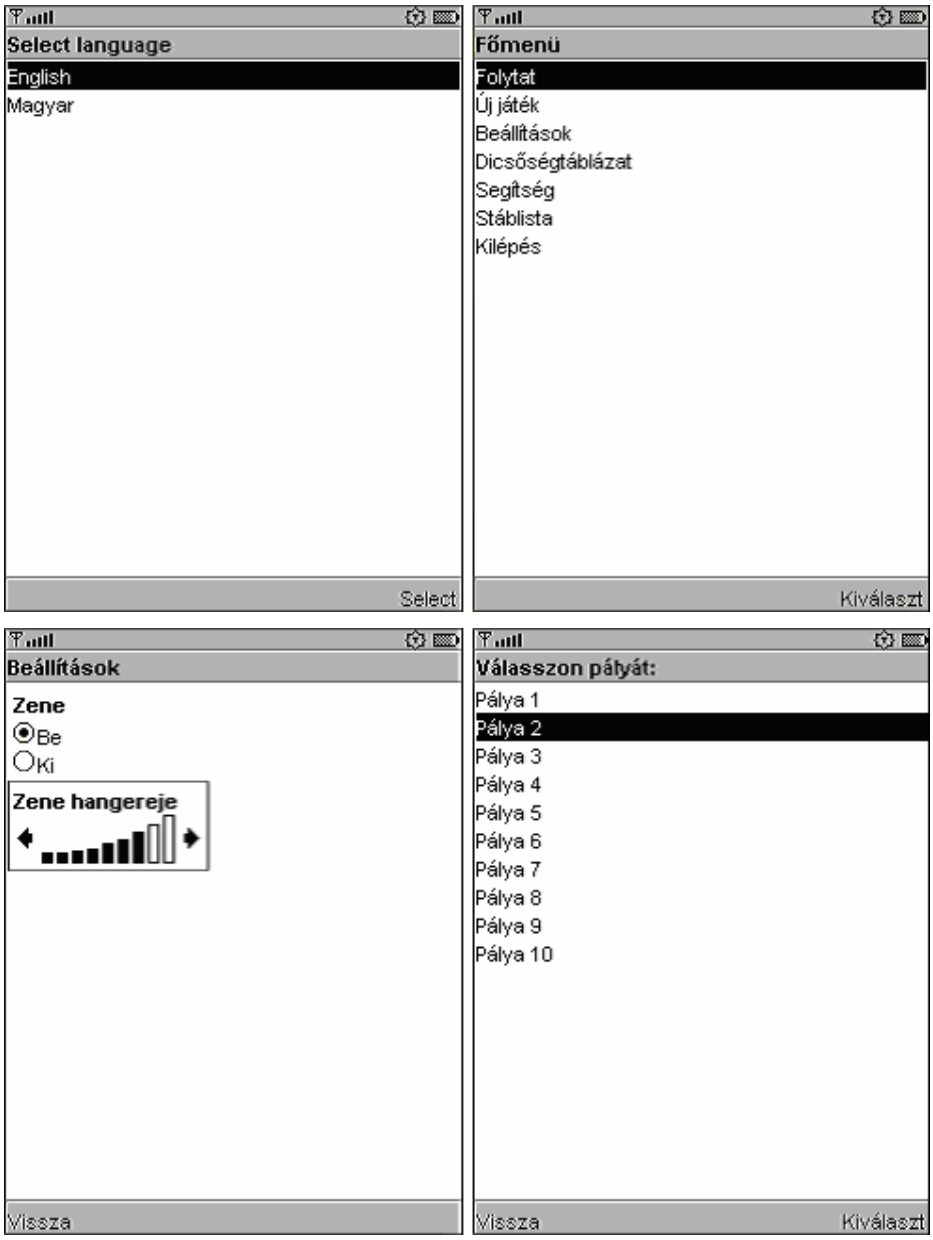
  Instrukciókat jelenít meg a játékról és az irányításról.
- Stáblista

  A játék és a felhasznált erőforrások készítőinek neveit jeleníti meg.
- Kilépés

  Kilép a programból.

A pályaválasztó képernyő felsorolja a játszható pályákat, ezek közül kell kiválasztani egyet, ekkor elindul a játék a kiválasztott pályán. Új játék indításakor a korábbi mentett állapot felülíródik. Játék közben a menügomb megnyomásával visszatérhetünk a főmenübe, ekkor a játék szünetel, a folytatás menüpont kiválasztásával visszatérhetünk a játékhoz, ami a legutóbbi állapotból folytatódik.

A játék végeztével amennyiben rekord-pontszámot ért el a játékos, a megjelenő képernyőn felvihetjük az új rekordot a dicsőségtáblázatba. Egy szövegdobozba nevet vagy tetszőleges szöveget írhatunk, alatta megjelenik az elért pontszám és a helyezés. A mentés parancs hatására az eredmény elmentődik, majd megjelenik a dicsőségtáblázat.



Néhány képernyő a menürendszerből

3.3 Játékmenet

A játékban négyféle objektum van: játékos űrhajója, labdák, téglák, bónuszok. A játékos űrhajója a képernyő alján helyezkedik el, vízszintesen mozgatható a képernyő szélei között. Az űrhajó rendelkezik egy pajzzsal, ami képes visszapattintani a mozgó labdát, azonban korlátozott mennyiségű energiával rendelkezik, amely minden visszapattintással csökken. Az energia elfogytával a pajzs megszűnik. A pajzs ívelt, a visszapattintott labda új

iránya attól függ, hogy hol érte a pajzsot. Ennek köszönhetően viszonylag pontosan lehet célozni a labdával. A labda a játék kezdetén elkapott állapotban van, ez azt jelenti, hogy a labda önállóan nem mozog, azonban a játékos űrhajója hurcolja maga után, így már az első lövés is célozható. Az ellövés után a labda önállóan mozog az aktuális irányba, a képernyő széleit, a pajzsot, illetve téglákat érintve visszapattan, megváltoztatva az irányát. A képernyő alját elérő labdát elveszíti a játékos, és a labdakészletből egy új elkapott állapotú labda jelenik meg a hajó felett. A labdakészlet kiürülése és az utolsó labda elvesztése után a játéknak vége, a játékos veszít, majd visszakerül a főmenübe, vagy ha új rekordot ért el, akkor felviheti az új pontszámot a dicsőségtáblázatba.

A labda, a pajzs, és a hajó

A téglákról visszapattanó labda megsebzi azokat, típustól függően bizonyos mennyiségű sérülés után a tégla összetörik, majd eltűnik. A téglák elhelyezkedése az adott pályától függ, különböző alakzatokat vesznek fel, emellett a pálya az egyes téglák színét is meghatározza, azonban ez csupán esztétikai jelentőségű. A színekkel ellentétben a tégla megjelenése meghatározza a típusát, ettől függően az adott tégla egy, kettő, vagy három sebzésponttal rendelkezik, ennek megfelelő számban kell az adott téglát a labdával eltalálni, hogy eltűnjön. A negyedik téglatípus elpusztíthatatlan, a labda nem sebzi azt. A téglák típustól függő pontszámot érnek, amely eltűnésükkor hozzáadódik a játékos aktuális pontszámához. A hozzáadott értéket befolyásolja a Combo-számláló. Ennek az értéke kezdetben 1, és minden egyes eltüntetett tégla megnöveli az értékét eggyel. A tégla értéke megszorzódik a számlálóval, és végül az így kapott érték adódik hozzá az aktuális pontszámhoz. Ha a labda visszapattan a pajzsról vagy eléri a képernyő alját, akkor a számláló értéke visszaállítódik 1-re. Amikor az összes összetörhető téglát eltüntette a játékos, akkor a pályát megnyeri, az adott pályán elért pontszám kijelzése után betöltődik a következő pálya.

A különböző téglatípusok

Egy tégla eltűnésekor 15% esély van rá, hogy megjelenik egy véletlen típusú bónusz a tégla helyén, majd ez konstans sebességgel mozog a képernyő alja felé. Ha a játékos hajójának sikerül megérinteni, akkor a típustól függő hatás aktivizálódik. A korlátozott ideig tartó bónuszok hatása a labda elvesztésével megszűnik. A bónusz típusa a megjelenéséből látható, egy betű jelzi milyen fajtájú:

- B: megnöveli a labdakészletet eggyel.
- C: a pajzzsal ütközéskor visszapattanás helyett a labda elkapott állapotba kerül. Korlátozott ideig érvényes csak a hatása, ezidő alatt a pajzs színe megváltozik.
- S: ha az ütközésének hatására a tégla eltűnik, a labda nem pattan vissza, hanem eredeti irányába halad tovább, ezt a labda színének megváltozása jelzi. Csak korlátozott ideig érvényes.
- M: hatására a labda 4 új labdává oszlik szét, melyek átlós irányban egymástól eltávolodva haladnak.
- E: megnöveli a pajzs energiáját.

Előfordulhat, hogy egyszerre több bónusz is megjelenik, ezek hatása kombinálódhat, kivéve az M bónusz esetét, amely eltörli az aktív S és C bónuszok hatását, és amíg egynél több labda van addig ezek felvétele nem váltja ki a hozzájuk tartozó hatást, továbbá az M bónusz nem halmozható, tehát többszöri felvétele nem növeli 4 fölé a labdák számát.

Képek a játékból

Képek a játékból

3.4 Irányítás

A jelenleg elterjedt mobil eszközök háromféle módszert kínálnak a játékok irányításának megvalósítására. Ezek közül a játék kettőt implementál, mivel a harmadik nem túl elterjedt és az emulátoron tesztelése igen körülményes.

Az első a hagyományosnak számító billentyűz7et. A MIDP API két módszert is biztosít a billentyűzet kezelésére:

- getKeyStates():

  bitmaszkokat alkalmaz az egyes billentyűkhöz rendelt virtuális gombok állapotának követésére, a MIDP 2.1-es verzió vezette be.

```
int keyStates = getKeyStates();
if((keyStates & LEFT_PRESSED) != 0) {
     //lenyomott balra-gomb kezelése …
}else{
     //felengedett balra-gomb kezelése …
}
```

- keyPressed(int keyCode) / keyReleased(int keyCode):
  hagyományos eseménykezelés alapú megoldás, egy gomb megnyomása/elengedése eredményeképp lefut a megfelelő metódus, amely paraméteréül megkapja a lenyomott gomb kódját.

```
public final void keyPressed(int keyCode) {
    super.keyPressed(keyCode);
    if (keyCode = = KEY_STAR) {
        running = false;
    }
    if (keyCode = = KEY_POUND
         || keyCode == KeyCodeAdapter.SOFT_KEY_LEFT
         || keyCode == KeyCodeAdapter.SOFT_KEY_RIGHT)
    {
         //kilépés a menübe…
    }
}
```

A két módszer nagy hiányossága hogy közvetlenül nem támogatják a menügombokat, arra hivatkozva, hogy azok nincsenek minden eszközön jelen, azonban manapság ez ritka. A második módszer segítségével a probléma bár körülményesen, de megkerülhető. További probléma, hogy egységes szabvány hiányában a különböző gyártók eltérő kóddal látják el a menügombokat, illetve egyes készülékekben a gombok nem is váltanak ki eseményt, így lenyomásuk egyáltalán nem érzékelhető a program számára. A megoldást egy `KeyAdapter` osztály nyújtja, amely megpróbálja felismerni a készülék típusát, és annak megfelelően beállítani az aktuális, menügombokhoz rendelt kódokat.

A játék billentyűzettel történő irányítása a következőképpen lett implementálva: az űrhajó mozgatására a 4/6 gombok illetve a navigációs gomb bal/jobb iránya lett rendelve, a labda ellövésére az 5 illetve OK gombok használhatóak. A főmenübe visszamenni a menügombokkal lehet.

A második inputkezelési módszer az érintőképernyő. A MIDP API három metódus segítségével kezeli az eseményeket:

A kijelző megérintésekor a *pointerPressed(…)* metódus hívódik meg, paraméterei az érintés koordinátái. A kijelző elengedésekor a *pointerReleased(…)* metódus fut le, paraméterei az elengedés előtti utolsó koordináták. A mutatópálca vagy ujjunk mozgatásakor a *pointerDragged(…)* metódus hívódik meg, a mozgás folyamán sokszor, paraméterei az új koordináták. A MIDP API által használt koordináta rendszer eltér a Descartes-féle

rendszertől, az origó a kijelző bal-felső sarka, innen a jobb-alsó sarok felé haladva mindkét tengelyen nőnek az értékek.

A játék irányítása az érintőképernyővel a következőképp történik: a főmenübe visszatéréshez a képernyő jobb-alsó sarkában található menü ikonra kell „kattintani”. A labda ellövéséhez meg kell érinteni a képernyőt, majd elengedni. Letartott állapotban a mutató húzásával mozgathatjuk a hajót, amely követi a mutató vízszintes pozícióját.

```
protected void pointerDragged(int x, int y) {
        if (pointerPressed) {
            pointerX = x;
        }
}

protected void pointerPressed(int x, int y) {
        //menü gombra „kattintás”
        if (x > 194 && y > GAMEAREAHEIGTH + 17) {
            //kilépés a menübe
            …
        } else {
            pointerPressed = true;
            clicked = true;
        }
}

protected void pointerReleased(int x, int y) {
        pointerPressed = false;
```

A harmadik input-módszer a gyorsulásérzékelők és giroszkópok használata, melyek a telefon térbeli elmozdulását illetve elfordulását mérik. Ezek segítségével is lehetne irányítani a játékot, azonban ez itt nincs implementálva, mert ezek a funkciók csak a felsőkategóriás készülékekben érhetők el, és e funkciók emulálása körülményes, mivel az SDK-ba épített emulátor nem kezeli őket.

3.5 Felépítés

Bár a Java objektum-orientált nyelv, a Java ME játékok sokszor mégis funkcionális megközelítést alkalmaznak, hogy csökkentsék a méretet és növeljék a sebességet. Az általam készített játék is alkalmazza ezeket a praktikákat. A program forráskódja összesen 12 osztályból áll, ezek közül a fontosabbakat vizsgáljuk most meg részletesen.

3.5.1 GameMIDlet.java

Az egyik legfontosabb osztály a `GameMIDlet`, amely a `MIDlet` osztály leszármazottja, a program kiindulópontja. Implementálja a `CommandListener` interfészt ami az LCDUI-val készült menürendszer működtetéséhez szükséges. Számos példány-adattagot definiál, ezek főleg a GUI megvalósításához szükségesek, továbbá statikus adattagokat is definiál, melyek a beállításokat tárolják futás közben, illetve egyéb funkciókat megvalósító osztályok példányai.

```
public class GameMIDlet extends MIDlet implements CommandListener
{
    //GUI
    List languageForm, mainForm, levelSelectForm,
        loadForm, inGameMenu;
    Form settingsForm, helpForm, creditsForm,
        highScoreForm, newHighScoreForm;
    ChoiceGroup musicChoice;
    Gauge musicVolumeGauge;
    TextField highScoreName;
    StringItem highScoreValue, highScorePosition;
    //commands
    Command comSelect, comBack, highScoreSave;
    //highscores
    Score[] scores;
    //Util
    static MusicPlayer multimedia;
    //options
    boolean paused = false;
    static boolean musicOn;
    static byte musicVolume = 100;
    static int currentLevel;
    static long currentScore;
    static int currentLives;
    //game screen
    static LevelCanvas gameCanvas;
```

Ahogy azt már korábban tárgyaltuk, a `MIDlet` életciklusa felfüggesztett állapotban indul, majd meghívódik a *startApp()* metódus. Ugyanez a metódus fut le, amikor a játék külső esemény hatására felfüggesztett állapotba került (a *pauseApp()* metódus lefutása után), majd az esemény után folytatódhat a játék. A két eset elválasztását oldja meg a paused változó. Első híváskor az értéke hamis, ekkor a metódus példányosítja a zenelejátszást megvalósító `MusicPlayer` osztályt, majd megjeleníti a nyelvkiválasztó képernyőt. További hívásakor a metódus folytatja a zene lejátszását (a zenelejátszó ellenőrzi, hogy engedélyezve van-e).

```
protected void startApp() {
    if (paused) {
        multimedia.resumeMusic();
    } else {
        multimedia = new MusicPlayer();
        display(initLanguageForm());
    }
    paused = false;
}

protected void pauseApp() {
    paused = true;
    multimedia.pauseMusic();
}

public void display(Displayable screen) {
    Display.getDisplay(this).setCurrent(display);
}
```

A program kétféleképpen állhat le, vagy a főmenü kilépés gombjával, vagy az AMS hívására, mindkét esetben a *destroyApp()* metódus hívódik meg, a két esetet együtt kezeli. Először, amennyiben létezik, leállítja a futó játékvásznat, majd a zenét, elmenti a beállításokat és egyéb perzisztens adatokat, végül jelzi az AMS-nek hogy a program leállt (az utóbbira csak a menüből kilépéskor van szükség, de a másik esetben sem okoz problémát).

```
protected void destroyApp(boolean arg0) {
    stopGame();
    saveSettingData();
    notifyDestroyed();
}

public void stopGame() {
    if (gameCanvas != null) {
        gameCanvas.stop();
        gameCanvas.cleanUp();
        gameCanvas = null;
    }
    GameMIDlet.multimedia.cleanUp();
}
```

A korábban már bemutatott menürendszer is itt van implementálva az LCDUI API segítségével. Az egyes képernyők a megfelelő *init___Form()* metódusokkal jönnek létre. Megjelenítésükről a *display(*`Displayable` *screen)* metódus gondoskodik.

```
public Screen initLevelSelectForm() {
    if (levelSelectForm == null) {
        levelSelectForm = new List(
            Strings.get(Strings.STR_SELECT_LEVEL)
            ,List.IMPLICIT);
        for (int i = 1; i <= Levels.levels.length; ++i) {
            levelSelectForm.append(
                Strings.get(Strings.STR_LEVEL) + i ,null);
        }//kivalasztas parancs hozzáadása
        comSelect =  new Command(
            Strings.get(Strings.STR_SELECT),
            Command.ITEM, 1);
        levelSelectForm.setSelectCommand(comSelect);
        levelSelectForm.addCommand(initBackCommand());
        //listener hozzáadása a form-hoz
        levelSelectForm.setCommandListener(this);
    }
    return levelSelectForm;
}
```

A létrehozott képernyők működtetését a *commandAction(…)* metódus végzi, amely a képernyőkhöz adott parancsok aktivizálódásának hatására fut le. Paraméteréül megkapja az aktuális képernyőt és parancsot.

```
public void commandAction(Command cmd, Displayable display){
    …
    } else if (display.equals(levelSelectForm)) {
        if (cmd = = comBack) {
            display(initMainForm());
        } else {
            int selected = levelSelectForm.getSelectedIndex();
            currentLevel = selected;
            currentScore = 0;
            currentLives = 3;
            stopGame();
            openGame();
        }
    } else if (display.equals(highScoreForm)) {
        if (cmd == comBack) {
            display(initMainForm());
        }
    } else if …
```

Először megvizsgálja hogy melyik volt az aktuális képernyő, majd ettől függően a különböző parancsok esetén elvégzi a megfelelő műveleteket.

A program a beállítások és a dicsőségtáblázat tárolását az RMS-en keresztül végzi. Induláskor megnyitja az „options” nevű `RecordStore`-t, majd ha van benne tárolt adat kiolvassa azokat. Az adatokat az RMS-től egy bájttömb formájában kapjuk meg, amit egy `ByteArrayInputStream` inputjaként kell felhasználni, majd erre ráépíteni egy `DataInputStream`-et, melynek segítségével már tetszőleges egyszerű típusos adatokat olvashatunk szekvenciálisan.

```
public void loadSettingData() {
    try {
        RecordStore options =
            RecordStore.openRecordStore("options", true);
        // ha a recordstore nem üres
        if (options.getNumRecords() != 0) {
            loadOptions(options.getRecord(1));
        }
        options.closeRecordStore();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

public void loadOptions(byte[] data) throws IOException {
    ByteArrayInputStream bais = new ByteArrayInputStream(data);
    DataInputStream dis = new DataInputStream(bais);
    try {
        musicOn = dis.readBoolean();
        musicVolume = dis.readByte();
        currentScore = dis.readLong();
        currentLevel = dis.readInt();
        currentLives = dis.readInt();
        //discsöségtáblázat beolvasása
        for (int i = 0; i < scores.length; i++) {
            long value = dis.readLong();
            String name = dis.readUTF();
            if (name == null) {
                name = Strings.get(Strings.STR_EMPTY);
            }
            Date date = new Date(dis.readLong());
            scores[i] = new Score(value, name, date);
        }
        dis.close();
    } finally {
        dis.close();
    }
}
```

Kilépés előtt ezek az adatok elmentésre kerülnek. A mentés módja nagyon hasonló a beolvasáséhoz, anyi különbséggel hogy Input helyett Output-kezelő osztályokat használunk, az adatokat pedig nem olvassuk hanem írjuk.

```
public void saveSettingData() {
    try {
        RecordStore options =
            RecordStore.openRecordStore("options", true);
        byte[] data = getOptionsBytes();
        if (options.getNumRecords() != 0)
        // ha a recordstore nem üres
        {
            options.setRecord(1, data, 0, data.length);
        } else {
            options.addRecord(data, 0, data.length);
        }
        options.closeRecordStore();
    } catch (Exception ex) {  ex.printStackTrace();    }
}

public byte[] getOptionsBytes() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);
    byte[] output;
    try {
        dos.writeBoolean(musicOn);
        dos.writeByte(musicVolume);
        dos.writeLong(currentScore);
        dos.writeInt(currentLevel);
        dos.writeInt(currentLives);
        for (int i = 0; i < scores.length; i++) {
            dos.writeLong(scores[i].value);
            if (scores[i].name != null) {
                dos.writeUTF(scores[i].name);
            } else {
                dos.writeUTF(Strings.get(Strings.STR_EMPTY));
            }
            dos.writeLong(scores[i].when.getTime());
        }
        dos.flush();
    } finally {
        output = baos.toByteArray();
        dos.close();
        baos.close();
    }
    return output;
}
```

Az egyes pályák befejeztével az *endLevel(int score, boolean playerWon, int lives)* metódus dönti el hogyan tovább. Ha van még hátra új pálya akkor a pontszám elmentése után betöltődik a következő pálya. Egyébként megjelenik egy szöveges üzenet, melynek szövege attől függ hogy a játékos nyert-e vagy vesztett. Ha a végső pontszám elég magas, az üzenet után megjelenő képernyőn elmenthetjük, ekkor bekerül a dicsőség-táblázatba.

3.5.2 LevelCanvas.java

Magát az interaktív játékot a `LevelCanvas` osztály valósítja meg. A `GameCanvas` leszármazottja, mivel kihasználja a Game API nyújtotta lehetőségeket. Implementálja a `Runnable` interfészt, mert a játékot működtető kód külön szálban fut. Számos statikus és példányváltozót definiál melyek többek között definiálják az animációkat, tárolják az aktuális állapotot, hivatkoznak a felhasznált képekre, `Sprite`-okra. A konstruktor beállítja a teljes képernyős módot, azaz a felső állapotsáv, és az alsó parancssáv eltűnik, a teljes kijelző a játék rendelkezésére áll.

Működését a *star()t* metódus hívásakor kezdi meg, először kirajzol a képernyőre egy üzenetet, ezzel jelezve a felhasználónak hogy elkezdett betöltődni, meghívja a *initLevel()* metódust, ami betölti a szükséges képeket, és nullázza az állapotokat nyilvántartó változókat. Ezután példányosítja majd elindítja a működtető-szálat. A szál kódja a *run()* metódusban található, melyben egy while ciklus végzi a műveleteket.

A ciklusfeltétel a running változótón alapszik, értéke mindaddig true amíg a játék fut, ez akkor is igaz ha külső esemény miatt szüneteltetve van. A ciklusmag első lépésben elmenti az aktuális időt, ennek célját később részletesen kifejtem. Ezután megvizsgálja hogy a játék szüneteltetve van-e, ezt a midlet.paused logikai változó jelzi. Ha ennek értéke igaz akkor a szál egy másodpercig felfüggeszti a futását, egyébként lépteti a játékot. Ehhez a kilépési feltétel megvizsgálása után három metódust hív meg, először a *gameInput()* metódus fut le amely kezeli az inputot, ezután a *tickGame()* metódust hívja meg amely lépteti a játékot, végül a *renderGame()* metódus fut le amely kirajzolja az aktuális állapotot a képernyőre. Ezek lefutása után ismét feljegyzi az aktuális időt. A kezdeti és az imént lemért idő különbsége megadja hogy mennyi idő telt el a játék léptetése alatt, majd ha ez egy előre meghatározott lépésköz mennyiség alatt van akkor a szál a lépésközig hátralévő időre felfüggeszti működését. Ez azt a célt szolgálja hogy a különböző teljesítményű telefonokon is pontosan ugyanolyan sebességgel fusson a játék. A ciklus befejeztével, ha az nem megszakítás miatt ért

véget, kiíródik a pályán elért pontszám illetve egy szöveges üzenet attól függően hogy a játékos nyert vagy vesztett. Legvégül meghívja a `GameMIDlet` *endLevel()* metódusát.

```
public final void run() {
    Graphics g = getGraphics();
    long start;
    long end;
    int duration;
    int timeStep = 1000 / DESIRED_FRAMERATE;
    while (running) {
        try {
            start = System.currentTimeMillis();
            if (!midlet.paused) {
                ++tickCount;
                //nyerési feltétel
                if (destroyedBrickCount == DestroyableBrickCount)
                {
                    break;
                }
                gameInput();
                tickGame();
                renderGame(g);
                } else {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException ie) {
                    }
                }
                end = System.currentTimeMillis();
                duration = (int) (end - start);
                if (duration < timeStep) {
                    try {
                        Thread.sleep(timeStep - duration);
                    } catch (InterruptedException ie) {
                    }
                }
            } catch (Throwable t) {
                System.out.println("Tick: " + tickCount);
                t.printStackTrace();
            }
       }
       …
```

A *gameInput()* metódus kezeli az irányítást, amelyről már korábban volt szó. A *tickGame()* metódus lépteti a játékot, ez valósítja meg a játék logikáját. Egyszerre akár több labda is lehet a képernyőn, ezért ezek egy `Vector`-ban tárolódnak. A *tickGame()* metódus egy ciklussal végigmegy a labdák vektorán, melyben lépteti azok animációját, majd egy belső

ciklus indul, melynek célja hogy a gyorsan mozgó labdák ne „ugorjanak át” egy téglát sem. Ezt úgy valósítja meg, hogy a labdák mozgását felbontja több apró mozdulatra, melyek során legfeljebb 1 pixelnyit mozdul el a labda, ezáltal a labdák mozgása realisztikus lesz. Az ütközéseket vizsgáló algoritmusok hatékonyságának köszönhetően az ezzel járó számítás-többlet nem lassítja jelentősen a játék sebességét.

```
private final void tickGame() {
    for (int j = 0; j < balls.size(); ++j) {
        Ball ball = (Ball) balls.elementAt(j);
        ball.nextFrame();
        for (int i = 0; i < BALLSPEED; ++i) {
            ball.move(ball.speedX / BALLSPEED,
                ball.speedY/BALLSPEED);
        …
```

A belső ciklus ezután megvizsgálja a képernyő szélével való ütközéseket, amelyhez csupán a labda pozícióját kell megvizsgálni. Ütközés esetén pedig pusztán a labda haladási irányának megfelelő kopmonensét kell –1-el megszorozni, illetve a labdát újrapozícionálni hogy ne maradjon ütköző állapotban. A képernyő aljával való ütközés speciális eset, mivel ekkor a játékos elveszti a labdát, és ez akár a játék elvesztését is jelentheti. Emellett ilyenkor a bónuszok hatásai is megszűnnek.

A belső ciklus a következő lépésben a téglákkal való ütközést vizsgálja. A labda és téglák relatív mérete alapján elméletben legfeljebb 4 téglával érintkezhet a labda, így csak ezeket kell megvizsgálni. A téglák egy kétdimenziós tömbben tárolódnak, a négy téglából a bal felső indexeit úgy kapjuk meg hogy a labda x és y koordinátáit osztjuk a téglák szélességével/magasságával. A többi tégla az egyes indexek 1-el megnövelésével érhető el.

```
int col = ball.getX() / BRICKWIDTH;
int row = ball.getY() / BRICKHEIGTH;
if (row < bricks.length && col < bricks[row].length
    && bricks[row][col] != null
    && ball.collidesWith(bricks[row][col], true))
{
        brickHit(ball, bricks[row][col]);
 } else if …
```

A labdák és téglák a `Sprite` osztály leszármazottjai, így az ütközések tényleges vizsgálatát a Game API által implementált *collidesWith(Sprite s, boolean pixelLevel)* metódus végzi, amely figyelembe veszi az átlátszó pixeleket is.

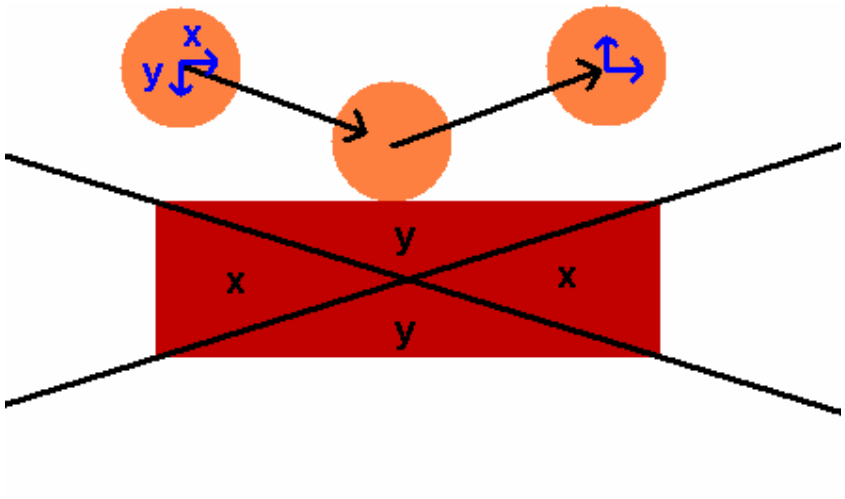
Ha a fenti feltétel teljesül valamelyik téglára a négy közül, akkor meghívódik a *brickHit(Ball ball,Brick b)* metódus, az ütköző labdával és téglával a paramétereiben, melynek feladata az ütközés feloldása, és az egyéb szükséges műveletek elvégzése.

```
private void brickHit(Ball ball, Brick b) {
    boolean broken = false;
    if (b.type != 3 && b.isBroken(1)) {
        broken = true;
        bricks[b.getY()/BRICKHEIGTH][b.getX()/BRICKWIDTH] = null;
        makeBreakEffect(b);
        destroyedBrickCount++;
        setComboCount(comboCount + 1);
        playerScore += comboCount * (b.type + 1) * 10;
        if (superChargeCountDown <= 0 && balls.size() == 1
            && Util.getRandom(100) <= 15)
        {
            PickUp p = new PickUp(pickupImg, Util.getRandom(5));
            pickUps.addElement(p);
            p.setPosition(b.getX(), b.getY());
        }
    } else {
        if (b.type != 3) {
            b.isHit = true;
        }
    }
    if (superChargeCountDown <= 0
        || (superChargeCountDown > 0 && ! broken)) {
        bounceBallOffBrick(ball, b);
    }
}
```

A broken változó átmenetileg tárolja hogy a tégla összetört-e. A tégla típusának vizsgálata után a `Brick`.*isBroken(int damage)* metódus hívásával megállapítjuk hogy a tégla az ütközés hatására összetört-e. Ha igen, akkor kitöröljük a téglákat nyilvántartó tömbből, létrehozunk egy törés-animációt a *makeBreakEffect()* metódussal, megnöveljük az összetört téglákat nyilvántartó számlálót és a Combo-számlálót, majd hozzáadjuk a tégla értékét a pontszámhoz. Ezután, ha a generált véletlen szám beleesik a 15%-os tartományba, létrehozunk egy véletlen típusú bónuszt. Ha a tégla nem tört össze, beállítjuk az isHit változót true-ra ami kirajzoláskor felvillantja a téglát, jelezve hogy eltaláltuk. Végül, ha az S bónusz

nem aktív, akkor a labdát visszapattintjuk a tégláról a *bounceBallOffBrick(Ball ball, Brick b)* hívásával.

A labda mozgása két komponensre, x és y irányú lendületre bontva tárolódik. A téglával ütközéskor általában a labda a téglának valamelyik oldalát érinti, ekkor a visszapattanáshoz az x/y lendületek egyikét a –1-szeresére kell változtatni. Speciális eset, amikor a sarkot érinti, ekkor mindkettőt meg kell változtatni. Azt hogy melyik a megváltoztatandó komponens, az alábbi ábra szemlélteti:



A tér felosztása a tégla átlóival

A teret a tégla átlóival négy részre osztjuk, a két komponenshez két-két negyed tartozik, ezeket a vörös téglába írt x/y-ok jelzik. A megfelelő komponenst megváltoztatjuk attól függően, hogy ütközéskor a labda melyik negyedben tartózkodik. Azt hogy melyik az aktuális negyed, a *bounceBallOffBrick* metódus az alábbi módon számolja ki:

Először kiszámolja a tégla és a labda középpontjainak relatív koordinátáit.

```
public void bounceBallOffBrick(Ball ball, Brick b) {
    double x, y;
    x = b.getX() + b.getWidth() / 2
        - ball.getX() - ball.getWidth() / 2;
    y = b.getY() + b.getHeight() / 2 - ball.getY()
        - ball.getHeight() / 2;
```

Ha a téglák négyzet alakúak lennének, egyszerűen a két érték viszonyából tudnánk melyik irányhoz tartozó negyedben vagyunk. Ha például abszolút értékben az x nagyobb, mint az y, akkor az x irányhoz tartozó két negyed egyikében van a labda. Az hogy ezek közül melyik, az előjel alapján eldönthető. Mivel a téglák téglalap alakúak, az x/y értékek

összehasonlítása előtt el kell őket osztani a téglák szélességével/magasságával, ezzel visszavezetve a problémát a négyzetes esetre. Az aktuális negyedtől függően a megfelelő lendület-komponens negálása mellett a labda pozícióját is meg kell kissé változtatni, hogy az ütköző állapot megszűnjön. Ennek egyrészt technikai oka van, másrész úgy lehetne szemléltetni, hogy a visszapattanó labda nem hatol bele a téglába, csupán érinti annak felületét.

```
if (Math.abs(x / BRICKWIDTH) <= Math.abs(y / BRICKHEIGTH)) {//y
    //újrapozícionálás
    if (y > 0 ) {
        ball.setPosition(ball.getX(), b.getY()-ball.getHeight());
    } else if (y < 0 ){
        ball.setPosition(ball.getX(), b.getY() + b.getHeight());
    }
    ball.speedY = -ball.speedY;
}
if (Math.abs(x / BRICKWIDTH) >= Math.abs(y / BRICKHEIGTH)) {//x
    if (x > 0 ) {
        ball.setPosition(b.getX()-ball.getWidth(), ball.getY());
    } else if (x < 0 ) {
        ball.setPosition(b.getX() + b.getWidth(), ball.getY());
    }
    ball.speedX = -ball.speedX;
}
```

Ezután, visszatérve a *tickGame()* metódus belső ciklusához, a program megvizsgálja, ütközik-e a labda a játékos pajzsával. Ha a C bónusz aktív, akkor a labda sebessége nullázódik, y koordinátája pedig átállítódik, hogy a hajó felett, nem ütköző állapotban legyen, egyébként visszapattan. A labda új irányát a labda és a hajó középpontjai által meghatározott szög adja. Ebből a szögből a szinusz és koszinusz függvények segítségével kiszámítható az x és y irányú lendület-komponens.

```
int newHeading = Util.getFacingAngle(player.getX()
    + playerShip.getWidth() / 2
    , playerShip.getY() + playerShip.getHeight() / 2
    , ball.getX() + ball.getWidth() / 2
    , ball.getY() + ball.getHeight() / 2);
ball.speedX = Math.cos(Math.toRadians(newHeading)) * BALLSPEED;
ball.speedY = -Math.sin(Math.toRadians(newHeading)) * BALLSPEED;
```

Ezzel a belső és külső ciklus lezárul, a labdák kezelése megtörtént. A pajzs animálása után a *tickGame()* metódus a bónuszok kezelésével foglalkozik. Egy ciklus végigmegy a bónuszokat tároló vektoron, törzsében először megnézi hogy a vizsgált bónusz elérte-e a

képernyő alját, ekkor azt törli, majd továbblépteti a ciklust. Egyébként megvizsgálja, hogy a játékos hajója ütközik-e a bónusszal. Ütközés esetén aktivizálja a típustól függő hatást, figyelembe véve az esetleges kizáró feltételeket. Végül csökkenti a korlátozott ideig tartó bónuszok számlálóit, azok nullára csökkenése esetén megszünteti azok hatását.

A program ezután kirajzolja az aktuális állapotot. A kijelzőre rajzoláshoz szükség van egy `Graphics` grafikus objektumra, melyet a *renderGame()* metódus paramétereként kap, ezt eredetileg a `GameCanvas`.*GetGraphics()* metódus hívásával kérte le a program. Ez valójában egy buffer, az erre történő rajzolás nem kerül ki a képernyőre mindaddig, amíg a *flushGraphics()* metódust meg nem hívjuk. Képek és szöveg kirajzoláshoz a grafikus objektum *drawImage(…)* illetve *drawString(…)* metódusait kell hívni. A `Sprite` példányok kirajzolásához azok *paint(Graphics g)* metódusát kell meghívni, melynek paraméterében át kell adni a grafikus objektumot, amelyre ki szeretnénk rajzolni.

A metódus először fekete színnel tölti ki a kijelzőt, majd kirajzolja a háttérhez tartozó véletlen elhelyezkedésű csillagokat. Ezt követően kirajzolja a játékos hajóját, a téglákat, bónuszokat, és vizuális effekteket. Legvégül kirajzolja a kijelző alján látható állapotsávot, amely az aktuális pajzs energiát és labdakészletet mutatja, majd meghívja a *flushGraphics()* metódust.

3.5.3 Egyéb osztályok

Az előbbi két oszály alkotja a program magját, azonban ezek mellett számos kisebb osztály is tartozik a programhoz, melyek szintén fontos szerepet töltenek be.

A Ball.java a labdák osztályát definiálja. A `Sprite` objektumok csak egészértékű mennyiséggel tudnak mozogni, ezzel korlátozzák a lehetséges irányok számát. Hogy a teljes 360 fokos skála elérhető legyen, az osztály példányszintű adattagként, double típussal definiálja az x/y pozícióját, és az x/y irányú lendületét, továbbá egy `Sprite` példányt is definiál. Az animációkkal, kirajzolással, ütközéssel kapcsolatos metódusokat delegálja a `Sprite` példánynak, a mozgással kapcsolatos metódusai azonban a saját adattagjaival dolgoznak, majd a metódusok végén a kerekítés után kapott értékkel frissítik a `Sprite` példány pozícióját.

A téglák a `Brick` osztály példányai, amely a `Sprite` leszármazottja, azt a téglák típusának és életpontjainak tárolásával, illetve a tégla megsebzését implementáló *isBroken(int damage)* metódussal egészíti ki. Ehhez hasonló a `PickUp` osztály amely a bónuszok

osztálya, a `Sprite` funkcionalitását csupán a bónusz típusának tárolásával, és a megjelenésének ettől függő beállításával egészíti ki.

A Levels.java az egyes pályák elrendezését definiálja. Kizárólag statikus adattagokat definiál, a pályákon a téglák elrendezését egy-egy kétdimenziós, 18x10-es byte tömbben tárolja. Az ebben szereplő értékek indexek, melyek az adott pályához tartozó, a téglák típusát és színét definiáló két tömb elemeire hivatkoznak. A későbbi bővíthetőség érdekében az egyes pályák elrendezését egy háromdimenziós tömb egyesíti, a típusok és színek tömbjeit pedig két kétdimenziós tömb egyesíti. A program később innen olvassa be a pályák számát, így egy új pálya hozzáadása nem igényli a többi osztály kódjának megváltoztatását. A módszer hátránya hogy így minden pálya végig a memóriában tárolódik, ami nagy méretű és mennyiségű pályák esetén könnyen a szabad memória elfogyásával járna. Jelen esetben azonban az elfoglalt terület minimális, így nem éri meg fájlban tárolni, mivel a fájlok megnyitása és beolvasása csak lassítaná a programot, miközben csak minimális memóriaterületet spórolnánk meg.

A `LinkedSpriteListElement` osztály `Sprite`-ok láncolt listáját valósítja meg. A `Vector` osztály, bár nagyon hasznos, nagyon lassú is. Csupán az elemek kiolvasásával jelentős mennyiségű idő megy el, így például ha a kirajzolandó objektumainkat egy `Vector`-ban tároljuk, kirajzoláskor az idő jelentős hányada csupán a `Vector` bejárásával megy el. Ha a `Vector` tartalmának mennyisége gyakran változik, szintén sok idő mehet el az elemeket tároló belső tömb újraméretezésére. A fölösleges időpazarlás elkerüléséhez így, amennyiben nincs szükség az elemek közvetlen, index alapján történő elérésére, célszerű az objektumokat egy láncolt listában tárolni, melynek mind a bejárása, mind a bővítése/törlése minimális időbe kerül.

3.5.4 Lokalizáció

A mobiltelefonok, és velük a játékok, az egész világon elterjedtek, a különböző országok eltérő módon formázzák a különböző mennyiségeket, mint például a idő, dátum, számok. A legfontosabb különbség azonban a beszélt nyelv. A felhasználók nagy része nem fogja megérteni az angol szöveget. A minél nagyobb felhasználótábor megszólítása érdekében be kell építeni a programba egy olyan mechanizmust, amely lehetővé teszi hogy a játékban látható szövegek egyszerűen, a program kódjának megváltoztatása nélkül, akár a kód újrafordítása nélkül is lefordítható legyen más nyelvekre.

Jelen esetben ezt a Strings.java valósítja meg. Az osztály statikus konstansokat és metódusokat definiál. A szövegek egy `Hashtable`-ben tárolódnak, az egyes szövegek azonosítói a korábban definiált statikus konstansok. A program indulásakor az elérhető nyelvek listája betöltődik a „languages” fájlból. Az itt szereplő sorokkal azonos nevű fájlban tárolódnak a lefordított szövegek. Miután a felhasználó kiválasztotta a nyelvet a listából, lefut a `Strings`.*load(String language)* metódus, amely paraméterében megkapja a kiválasztott nyelvet. Ekkor a metódus megnyitja az azonos nevű fájlt, majd soronként beolvassa és feldolgozza azt. Az egyes sorok a lefordított szövegeket <Azonosító>=<Szöveg> formában tartalmazzák. Az azonosítóknak meg kell egyezni a definiált konstansok egyikével, hogy a szöveget később vissza lehessen olvasni. A módszer hiányossága hogy újsor karaktereket nem lehet elhelyezni a szövegben, azonban a hosszú szöveg tördelése akkor is megoldandó problémát jelentene, ha támogatná az újsor karaktereket. Betöltés után a programból a lokalizált szövegeket a `Strings`.*get(String ID)* metódus adja vissza, ahol az ID az osztályban definiált konstansok egyike. Például:

```
mainForm.append(Strings.get(Strings.STR_NEW_GAME), null);
mainForm.append(Strings.get(Strings.STR_OPTIONS), null);
mainForm.append(Strings.get(Strings.STR_HIGHSCORES), null);
```

3.6 Grafika

A videojátékok igen fontos része a grafikai megjelenés, mivel egyrészt az első benyomás főleg a megjelenésen alapszik, másrészt később is sokat javíthat, vagy éppen ronthat a játékélményen a játék megjelenése. Maguk a felhasznált grafikus fájlok nagyban befolyásolják mind a futás közben felhasznált memória mennyiségét, mind a JAR fájl méretét. Az utóbbi jelentősen csökkenthető a képfájlok megfelelő tömörítésével és egyéb trükkökkel. Ezekre szükség is van, mivel a JSR 185 által meghatározott maximális JAR méret 64 kilobájt. Maguk az eszközök ugyan képesek lehetnek akár több megabájtos JAR fájlok futtatására is, azonban a programok letöltésére szolgáló OTA (Over-the-Air Provisioning) technológia még manapság is csak legfeljebb 300-400 kilobájtos programokat tud hatékonyan eljuttatni a felhasználóhoz. A platform általánosan támogatott képformátuma a PNG (Portable Network Graphics) amely a licenc-köteles GIF formátum szabadon felhasználható utódjaként jött létre. Ez egy veszteségmentes formátum, elődjéhez képes számos továbbfejlesztést is tartalmaz, mint például az alfa csatorna, gamma korrekció, kétdimenziós fokozatos kijelzés. A JSR 185

előírja a JPEG formátum támogatását is. Emellett az egyes eszközök további formátumokat is támogathatnak.

A játékban felhasznált grafikus fájlok mérete minimalizálva van a színek számának és mélységének csökkentésével, illetve maximális tömörítést biztosító algoritmusok használatával. A JAR méretének csökkentésére egy további trükköt is alkalmaztam:

A pályákon az adott típusú téglák megjelenése azonos, csupán színben térnek el, így fölösleges lenne a különböző színű téglákat külön letárolni, ehelyett egy szürke-sablont tárolok csupán, amely a 3.2 fejezetben már látható volt. A pályák betöltésekor a különböző színű téglák előállításához a `Util.`*GetColoredImage()* metódus először kiolvassa a képből az RGB adatokat egy tömbbe, amely a kép pixeleinek színértékét tartalmazza. Ezután felbontja a színező színt komponenseire, majd végigmegy a tömbön, és felülírja az értékeket azok módosított változatával, melyek a színező szín árnyalatai, attól függően, hogy a szürke-sablonban milyen érték szerepelt az adott helyen. A szürke-sablon fehér pixeljeinek helyén a színező szín fog szerepelni, míg a sötétebb pixelek a színező szín egy sötétített árnyalatára lesznek cserélve. A sötétítés egyenesen arányos az eredeti pixel sötétségével, így míg a fehér pixel nem sötétít, míg a fekete pixelek teljesen elsötétítik az eredeti színt, a feketétől alig megkülönböztethető árnyalatúvá. Legvégül visszaadja a tömbből készített immár színes képet.

```
public static Image getColoredImage(Image grayScaleImg,int color)
{
    int cData[] = new int[grayScaleImg.getWidth()
        * grayScaleImg.getHeight()];
    grayScaleImg.getRGB(cData, 0, grayScaleImg.getWidth(), 0, 0
        , grayScaleImg.getWidth()
        , grayScaleImg.getHeight());
    int red = ((color & 0x00ff0000) >> 16);
    int green = ((color & 0x0000ff00) >> 8);
    int blue = ((color & 0x000000ff) >> 0);
    for (int i = 0; i < cData.length; ++i) {
    int alpha = ((cData[i] & 0xff000000) >> 24);
    double darkenRatio = (cData[i] & 0x00ff0000)*1f /0x00ff0000;
    cData[i] = Util.getColor(alpha,
        Util.round(red * darkenRatio),
        Util.round(green * darkenRatio),
        Util.round(blue* darkenRatio));
    }
    Image newImage = Image.createRGBImage(cData,
        grayScaleImg.getWidth(),
        grayScaleImg.getHeight(), true);
    return newImage;
}
```

3.7 Hang/zene

A videojátékok egy másik fontos része a hang. A játékfejlesztők igen hamar felfedezték hogy a hang és zenei hatások igen hatékony eszközök a kívánt játékélmény megteremtésében. Példa erre az annak idején nagy sikereket elért Space Invaders, melyben az ellenséges űrhajók közeledtével egyre gyorsult a hang. A játékosok szívritmusát vizsgálva kiderült, hogy a hanggal együtt a szívritmus is felgyorsult, és ahogy az ellenfél egyre közelebb és közelebb ért, a játékosok pánikba estek. Hang nélkül azonban jelentősen gyengébb volt a reakció[9]. [6]-ban az elsődlegesen hangokon alapuló játékokat tanulmányozzák.

Habár a mai digitális technikának köszönhetően a videojátékok jó minőségű hang és zenei hatásokat alkalmaznak, a mobiltelefonos játékok jelenleg a 90-es években használt technikákat alkalmazzák. Az általánosan támogatott formátum a MIDI, bár az egyes implementációk eltérő mintakészlettel rendelkezhetnek, így a végeredmény eltérő lehet az egyes eszközökön. Ennek ellenére a MIDI egy nagyszerű eszköz a háttérzene megvalósítására, mivel a fájlok mérete minimális. Sok készülék támogatja az MP3 formátumot is, azonban a zenefájlok több megabájtos mérete miatt a gyakorlatban nem alkalmazható a háttérzenéhez.

A hangok és zene lejátszása erőforrások tekintetében igen költséges feladat. Egy erőforrás-igényes játék esetén előfordulhat, hogy a zene elindításához már nincs elég memória. Számolni kell továbbá a hangok és zene lejátszásával járó többletterhelésre, így kevesebb processzoridő jut más feladatok elvégzésére. A hangeffektek esetén ez igen nagy problémát jelent, mivel azokat sok esetben gyakran le kellene játszani, ez a telefon túlterheléséhez vezethet, ami a játék teljesítményének romlásához, a játék futásának megszakításához, extrém esetben a telefon operációs rendszerének összeomlásához is vezethet. Egy lehetséges kompromisszum, ha csak olyan hangeffekteket alkalmazunk, amelyek csak ritkán hallhatóak, így elkerülve a túlterhelést. Másik lehetőség, ha egyáltalán nem alkalmazzuk őket. Ezt azért is célszerű megfontolni, mert nem minden eszköz képes egyszerre több hangfájl lejátszására. Háttérzene alkalmazása esetén a hangeffektek lejátszása egyes készülékeken nem várt viselkedéshez vezethet.

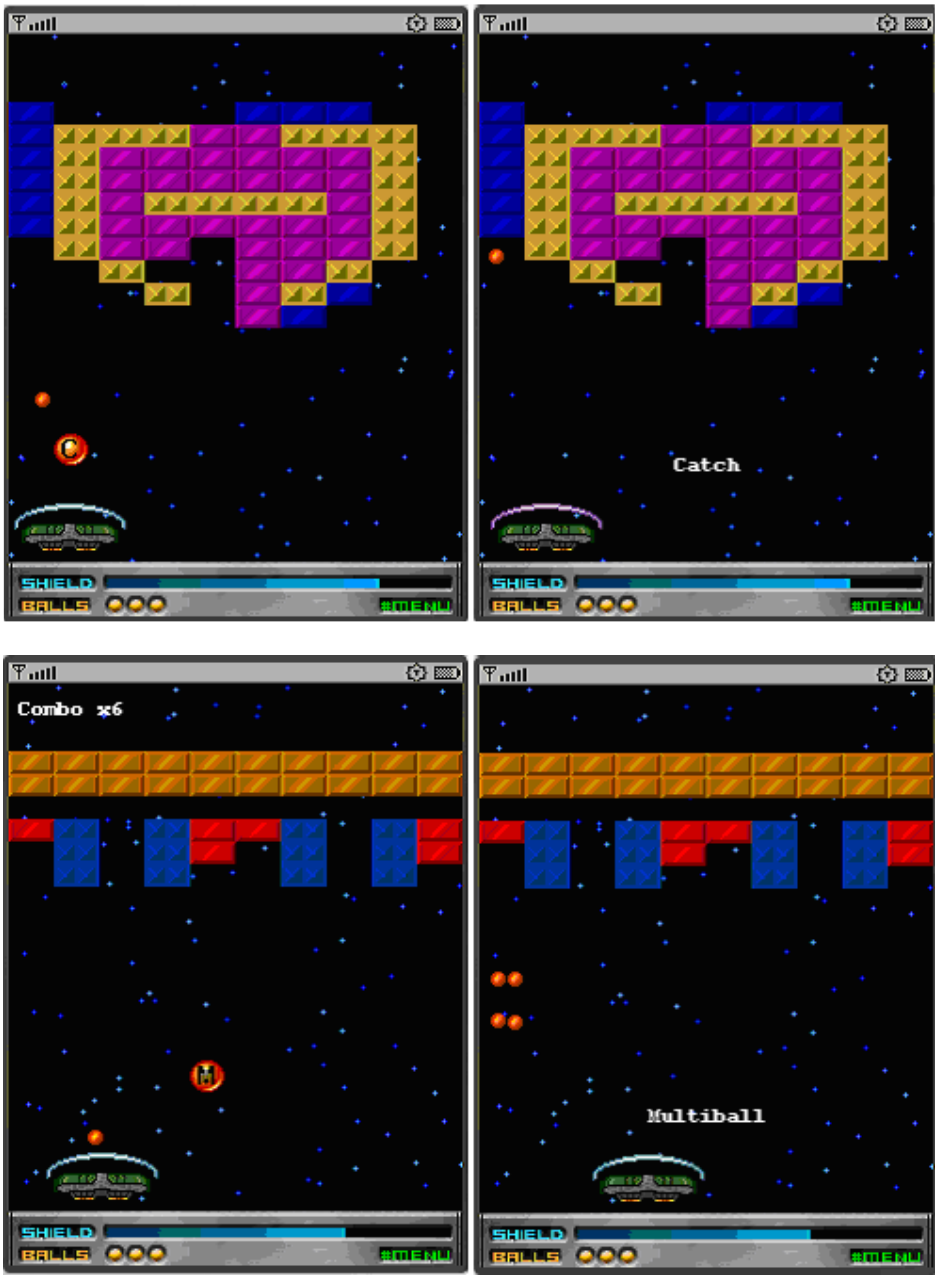
Az általam készített játékban MIDI formátumú háttérzene hallható, hangeffektek nincsenek, a fenti okok miatt.

4. Összefoglalás

Mint azt láthattuk, a Java ME platform számos korlátot és leküzdendő akadályt állít a játékfejlesztők elé, ám ugyanakkor sok helyen segíti is őket. Összességében a platform kihívásokkal és lehetőségekkel teli. A most megismert eszközök számtalan egyszerű vagy akár bonyolultabb játék elkészítését is lehetővé teszik, a platform azonban számos további lehetőséget is tartogat a fejlesztők számára, emellett folyamatosan fejlődik, ahogy maguk a mobil eszközök is. Jelenleg igen divatosak a 3D-s játékok, annak ellenére, hogy megjelenésük a 3D-s grafika hajnalát idézi. A jövőben azonban ez jelentősen javulni fog, ahogy a mobiltelefonok kapacitása nő.

Az elkészült játék fejlesztése során lehetőségem nyílt a platform mélyebb megismerésére, annak hiányainak és erősségeinek megtapasztalására. Fejlesztés közben számos ötlet felmerült, amelyek ugyan nem kerültek bele a játékba, azonban megvalósításuk érdekes kihívás lehet. Ilyen például a játékba épített pályaszerkesztő, kerek és egyéb alakú, illetve mozgó téglák, és ellenséges űrhajók hozzáadása a játékhoz.

Függelék



További képek a játékból

Irodalomjegyzék

- [1] Mobilhang gyorsjelentés 2010
  http://nhh.hu/dokumentum.php?cid=22728
- [2] A vállalkozások és a háztartások IKT-eszközökkel való ellátottsága és ezek használata, 2008, Központi Statisztikai Hivatal
  http://portal.ksh.hu/pls/ksh/docs/hun/xftp/idoszaki/ikt/ikt08.pdf
- [3] Elina Koivisto: Mobile Games 2010, 2007, Murdoch University
  http://research.nokia.com/files/NRC-TR-2007-011.pdf
- [4] Massimo Pezzini, Yefim V. Natis, Mark Driver, Nick Jones: Oracle's Acquisition of Sun Could Change Java's Course, 2009, Gartner
  http://www.gartner.com/resources/167600/167659/oracles_acquisition_of_sun_c_167659.pdf
- [5] IGDA 2008-2009 Casual Games White Paper
  http://archives.igda.org/casual/IGDA_Casual_Games_White_Paper_2008.pdf
- [6] Inger Ekman, Laura Ermi, Jussi Lahti, Jani Nummela, Petri Lankoski, Frans Mäyrä:
  Designing sound for a pervasive mobile game, 2005, ACM
- [7] Elina M. I., Riku Suomela, Jussi Holopainen : Using prototypes in early pervasive game development, 2008, ACM
- [8] Christopher Williams, Mark Burge: MIDP 2.0 changing the face of J2ME gaming, 2004, ACM
- [9] The Evolution of Video Game Music
  http://www.npr.org/templates/story/story.php?storyId=89565567
- [10] Annakaisa Kultima: Casual Game Design Values, 2009, ACM
- [11] Bátfai Norbert, Molnár Péter, Molnárné Nagy Mária, Rábai Bálint, Szitha Kristóf, Kovács Zsolt, Hudák László, Rák János:
  A Debreceni Fejlesztői Hálózat (beküldve), Híradástechnika, 2010
- Mark Callow, Paul Beardow, David Brittain:
  Big Games, Small Screens, 2007, ACM
- Martin J. Wells: J2ME Game Programming, 2004, Premier Press
- Ray Rischpater: Beginning Java™ ME Platform, 2008, Apress

- Carol Hamer: Creating Mobile Games: Using Java ME Platform to Put the Fun into Your Mobile Device and Cell Phone, 2007, Apress
- Bátfai, N.: Nehogy már a mobilod nyomkodjon Téged! DEENK, 2008., http://www.eurosmobil.hu/NehogyMar
- Java ME Technology - CDC http://java.sun.com/javame/technology/cdc/overview.jsp
- Connected Limited Device Configuration (CLDC); JSR 30, JSR 139 Overview http://java.sun.com/products/cldc/overview.html
- The Java Community Process http://www.jcp.org/en/procedures/overview
- What's New In MIDP 3.0 ? http://justanapplication.wordpress.com/2009/06/02/whats-new-in-midp-3-an-overview/
- The Generic Connection Framework http://developers.sun.com/mobility/midp/articles/genericframework/
- Over-the-Air Provisioning with the J2ME Wireless Toolkit http://developers.sun.com/mobility/midp/ttips/wtkota/
- Understanding JSR 185 http://developers.sun.com/mobility/midp/articles/jtwi/