SZAKDOLGOZAT

Szentpéteri Zsolt

Debrecen
2010

Debreceni Egyetem
Informatikai Kar

Játékfejlesztés Java környezetben

Témavezető:
Espák Miklós
Egyetemi tanársegéd

Készítette:
Szentpéteri Zsolt
Programtervező Informatikus

Debrecen
2010

Tartalomjegyzék

Bevezetés

A játékfejlesztés napjainkban eléggé elterjedt iparággá nőtte ki magát. A játékszoftvert fejlesztő cégek nap mint nap dobnak piacra újabbnál újabb játékokat. A fejlesztés sok apró összetevőből áll, és nagyfokú összehangoltságot igényel. Nagy szerepe van annak, hogy ismerjük a piaci helyzetet, és tudjuk, mi az, amivel szívesen játszana az általunk meghatározott célközönség. Ez ahhoz szükséges, hogy eladható, és egyben a célközönség számára érdekes játékot fejlesszünk. Vannak kisebb úgynevezett garázsprojektek, melyeket kis létszámú társaságok szoktak fejleszteni, vannak középméretű, és egészen nagy játékfejlesztő cégek is, melyek többnyire nagy volumenű, akár több évig fejlesztendő játékszoftvereket is fejleszthetnek. Többféle platformra történhet a fejlesztés, így például napjainkban a legelterjedtebbek a PC - re, IPhone-ra, PSP-re, stb… történő fejlesztések. Az Internet egyre nagyobb fokú elterjedésével tömegesen találhatóak a piacon a webes játékok, melyek lehetnek Flash, vagy akár PHP-ben írt böngésző alapú játékok. Egészen a kisgyermekektől kezdve az idősebbekig, bárki megtalálhatja a számára legmegfelelőbbet, az Interneten való szörföléssel.

Az Invictus Games Kft., egy Debrecenben székelő játékfejlesztő cég, mely cégnél én is dolgozom, s itt fejlesztették ki IPhone-ra a Fly Control nevű játékot. Az IPhone egyike a manapság legkedveltebb játékfejlesztő platformnak, mivel ha valaki úgy dönt, hogy ilyen játékot szeretne fejleszteni, annak az eladások alapján juttatásokat nyújt az Apple, s így akár bárki, aki regisztrált felhasználó, könnyen bevételre tehet szert. Az Apple, naponta számos új játékot dob a piacra, melynek jelentős részét játékfejlesztő csoportok, cégek fejlesztik. Minden egyes letöltés után a bevétel bizonyos százalékát átutalja a fejlesztőknek az Apple.

A Fly Control nevű játékban, az érintőképernyő segítségével kell irányítanunk a folyamatosan megjelenő legyeket anélkül, hogy összeütköznének. Ennek az ötletnek a PC-n való változatát valósítottam meg Java környezetben, néhány olyan dologgal kiegészítve, amelyet hiányoltam, vagy úgy gondoltam, hogy feldobná a játékélményt.

Jelen szakdolgozatomban ennek a játéknak a fejlesztését szeretném bemutatni. Olvasható fejlesztői kézikönyv, melyben megismerhetjük, miként is tudunk barangolni játékon belül, és

látható lesz miként lettek kialakítva a fontosabb osztályok. A fejlesztés során fontos szempontnak tartottam az újrafelhasználhatóságot. Ezáltal a lokalizáció, a naplózás, az alaposztályok, és egyéb osztályok is úgy lettek kialakítva, hogy később, ha más alkalmazást szeretnék fejleszteni, akkor könnyen felhasználhatóak legyenek.

1

1 Felhasználói kézikönyv

1.1 Bevezetés

Üdvözöljük a Fly Control nevű játékban! A következő néhány oldalon a játék felhasználói kézikönyvét olvashatja, s ezekkel az információkkal közelebb juthat a játékban használatos fogalmak megismeréséhez, és a játékban való tájékozódáshoz.

1.2 A Fly Control játék lényege

A játékban folyamatosan megjelenő objektumokat kell irányítanunk az egér segítségével, egy meghatározott célpont irányába. Mivel egyszerre akár több ilyen objektum is megjelenhet, cél az, hogy ezek nem ütközhetnek egymással. Ha mindezt figyelembe véve sikeresen eljuttattuk a célba az objektumokat, akkor pontot kapunk érte. Cél a minél több pont gyűjtése.

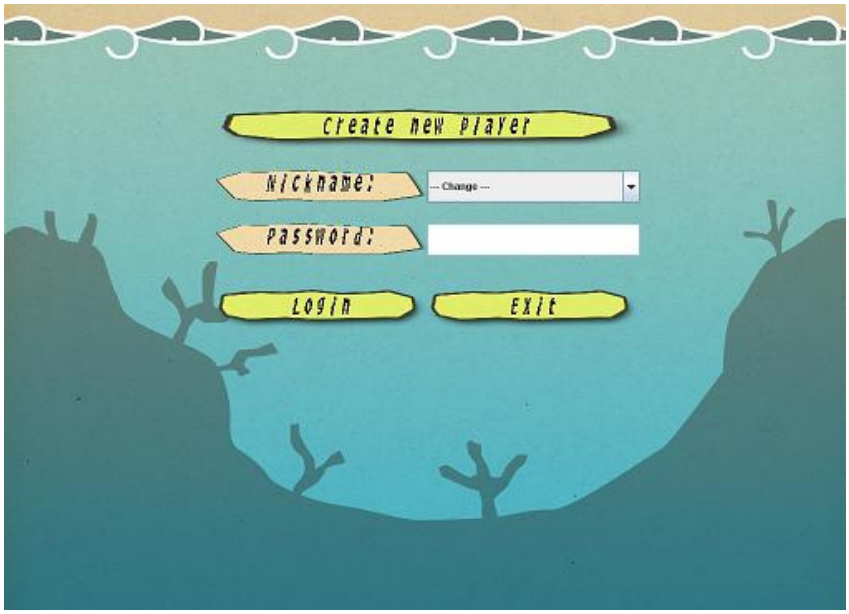
1.3 Regisztráció

A játékban való részvételhez szükséges egy gyors és egyszerű regisztráció, ahol mindössze egy felhasználónevet, és egy jelszót kell megadnia. A jelszó, és a felhasználónév sem tartalmazhatja a „#” speciális karaktert! A sikeres regisztrációhoz meg kell adnia mindkét adatot.

1.4 Kilépés gomb

A bejelentkező képernyőn található kilépés gomb, a játék ablakának bezárását, s ezáltal a játékból való kilépést szolgálja.

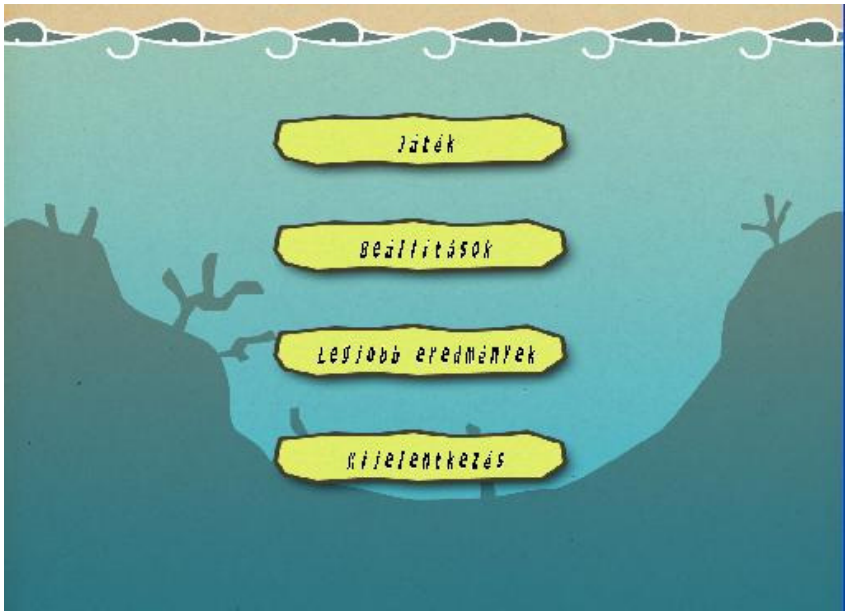
1.5 Bejelentkezés



A bejelentkező képernyő

A bejelentkezéshez egy korábban végrehajtott regisztráció szükséges. Az akkor megadott felhasználónévvel, és jelszóval sikeresen beléphet a rendszerbe. A belépés után a főmenüben találja magát a felhasználó, ahol a következő lehetőségek közül választhat:

- Játék
- Beállítások
- Legjobb pontok
- Kijelentkezés



A főmenü

1.6 Lokalizáció

A játékban lehet nyelvet választani. Ezzel a funkcióval a megadott nyelvek közül kiválaszthatja a felhasználó a számára legmegfelelőbb lokalizációt.

1.7 Szintek

A felhasználó húsz választható szinten játszhat. A szintek különböző nehézségeket jelentenek. Az első szinten a legkönnyebb játszani, míg a huszadikon a legnehezebb. Egy szint fél percig tart, ami annyit jelent, hogy az objektumok fél percig jelennek meg folyamatosan, megadott időközönként, s ezalatt kell elkerülni az ütközést. Amennyiben ütközés történt, s a játékosnak van még élete, akkor a szint újraindul, s ismét fél percig tart. A játékos dönthet arról, hogy melyik szinttel szeretné kezdeni a következő játékot.

Ha úgy gondolja, hogy ki szeretné próbálni milyen is a játék magasabb szinteken, a „Beállítások” menüpontban változtathatja meg a kezdőszintet.

1.8 Pályák

A pályák adják magát a játékteret. Több pálya létezik, és a felhasználó választhatja ki, hogy éppen melyiken szeretne játszani. A pályának van egy kitüntetett pontja, amely az irányítható objektumok célpontját szolgálják.

1.9 Beállítások

A beállítások menüpont a főmenüből érhető el. Itt állítható be a felhasználó által választott lokalizáció, valamint a szint, amellyel a következő játékot indítani szeretné, és a pálya is, ahol játszani szeretne. A „Mentés” opcióval mentheti el a változásokat, míg a „Vissza” gombbal visszakerül a főmenübe.

1.10 Legjobb pontok

A legjobb pontok menüpont alatt láthatja a regisztrált játékosok közül a húsz legjobb pontot elérők névsorát, és azt, hogy mikor érték el ezt a pontot.

1.11 Kijelentkezés

A kijelentkezés gombra kattintva a játékos visszakerül a bejelentkező képernyőre, és újra beléphet, vagy választhat másik felhasználónévvel való belépést.

1.12 Játékmező

A játékmező a játékos által a „Beállítások” menüpontban beállított pályára juttat ki. Megtalálható rajta az irányítható objektumok célpontja, ahova el kell vezérelnünk őket. A játékmezőn jelennek meg a pályához tartozó irányítható és nem irányítható objektumok is.

A játéktér

1.13 Az irányítható objektumok

Ezek azok az objektumok, amelyeket az egér segítségével irányíthatunk. Az egér bal gombját lent tartva jelölhetjük ki az utat, ahol azt szeretnénk, hogy mozogjon. Véletlenszerűen jelennek a játéktér bármely oldalán, s mindaddig véletlenszerűen mozognak, míg irányt nem mutatunk nekik. Ha kijelöltük számukra az irányt, s a kijelölés végpontja nem egyezik meg a célponttal, akkor a végpont elérése után ismét véletlenszerűen mozognak.

1.14 Nem irányítható objektumok

A nem irányítható objektumok is ugyanolyan tulajdonsággal rendelkeznek, mint az irányíthatóak, de ezek annyiba speciálisak, hogy nem tudjuk őket irányítani. Magasabb szinteken jelennek meg először, céljuk megnehezíteni az irányítható objektumok irányítását.

1.15 Irányítás

Az irányítást tehát az egér bal gombjának lenyomásával, és annak lent tartásával tehetjük meg. Amennyiben olyan területre irányítjuk az egeret, amely a játéktér területén kívülre esik, az útkijelölés befejeződik, amikor elhagyja az egér a játékteret.

1.16 A játékos „élete”

A játékos, amint kikerül a játékmezőre, onnantól számítva öt élettel rendelkezik. Ez azt jelenti, hogy ötször ütközhet másik objektummal. Tehát ha az egyik irányítható objektuma ütközik egy másik akár irányítható, akár nem irányítható objektummal, akkor veszít egyet az életéből. Amint elfogy az élete, azaz nulla lesz ez az érték, akkor a játéka befejeződik, és az addig szerzett pontjai alapján határozódik meg, hogy bekerül-e a legjobb pontú játékosok közé.

1.17 Ütközésvizsgálat

Az ütközésvizsgálat alapján, ha két objektum elég közel ér egymáshoz, akkor azt ütközésnek vesszük, s ha maradt még a játékosnak élete, akkor továbbjátszhat ugyanezen a szinten, életének eggyel való csökkentésével.

1.18 Ajándékok

Az ajándékok célja, hogy a játékos több pontot tudjon szerezni. Egy ajándék a játéktér valamely véletlenszerű pontján megjelenő objektum, különböző pontértékekkel. 10 másodpercig látszik egy ajándék. A megszerzéséért az irányítható objektumot kell úgy irányítani, hogy az ütközésvizsgálat alapján „ütközzön” az ajándékkal, s amennyiben ez sikerül megkapja az érte járó jutalompontot. A megjelenéstől számítva időarányosan csökken a szerezhető jutalompont értéke. Tehát minél hamarabb sikerül megszerezni az ajándékot, annál több jutalompont kapható érte. Hatféle ajándéktípus található meg a játékban, mindegyik más-más alap pontértékkel rendelkezik.

1.19 Értékelő képernyők

Az értékelő képernyők a szintek végén felbukkanó képernyők. Háromféle ilyen képernyőt láthatunk, az egyik, amikor ütközés történt, a másik, amikor sikeresen teljesítettük a szintet, és

a harmadik, amikor elfogyott az élete a játékosnak, és nincs több élete, ebben az esetben kapja a végleges értékelő képernyőt.

1.20 A log fájl elküldése

Ha a játék során hibát észlel a játékos, akkor a hiba javítása érdekében lehetősége van arra, hogy a játék log fájlját elküldje a fejlesztőnek. Ehhez annyit kell tennie, hogy a log.properties fájlban a naplózás szintjét beállítani magasabb szintre. Négyféle szint létezik, a NONE, ami az alapbeállítás, INFO, DEBUG és az ERROR. Ezek egyre magasabb szintű logot eredményeznek, tehát ha a legmagasabbat állítja be a felhasználó, akkor több eséllyel található benne értékes információ, mintha az INFO-t állítja be. Ha megfelelő a beállítás, akkor létrejön egy log könyvtárban egy új log fájl, s ezt kell elküldeni a fejlesztőnek a hiba felderítésének érdekében.

2 Alap funkciók

2.1 Logger

2.1.1 IDE - Integrated Development Environment[2]

Jelentése: Integrált fejlesztői környezet. Haszna abban rejlik, hogy a számítógép-programozást jelentősen megkönnyítse. Sokkal gyorsabban lehet vele alkalmazásokat fejleszteni, mivel a legtöbb IDE már ott segíti a programozót, ahol tudja. Az IDE-k általában tartalmaznak egy szövegszerkesztőt a program forráskódjának szerkesztésére, egy fordítóprogramot vagy értelmezőt, valamint nyomkövetési, grafikusfelület-szerkesztési és verziókezelési lehetőségeket sok egyéb mellett. Ilyen például a NetBeans IDE is, aminek segítségével a szakdolgozatom készült.

2.1.2 Naplózás

A programozásban a szoftver fejlesztése során nagy jelentősége van az úgynevezett naplózásnak és a program belövésének. Ez a kettő nem keverendő össze, inkább nagyon is jól működnek együtt. Amennyiben nincs meg a megfelelő fejlesztő környezet (IDE), amivel a

[2] Integrált fejlesztői környezet - http://hu.wikipedia.org/wiki/Integr%C3%A1lt_fejleszt%C5%91i_k%C3%B6rnyezet

belövést hatékonyan el tudnánk végezni, akkor jöhet hasznosan a naplózás. A fejlesztő elhelyezhet a forráskódban `log` kiírásokat, ahol akár az aktuális változók értékeit is kiírattathatja egy erre a célra létrehozott `log` fájlba. Lehetnek azonban olyan kiírások is, ahol egyszerűen csak arra vagyunk kíváncsiak, hogy az a bizonyos kódrészlet egyáltalán lefutott e. A programozóra van bízva, milyen információkat szeretne tudni, és hogy mikor mit naplóz ki. Ha a naplózás szintje, megfelelően van beállítva, az alkalmazás futtatása során készül egy "`log fájl`", ami tehát sok hasznos információt tartalmazhat a fejlesztőnek, az alkalmazás fejlesztése, tesztelése és az alkalmazás üzembe helyezése után is.

A tesztelések végeztével, az alkalmazás üzembe helyezése után a legtöbb esetben a felhasználó kezébe kerül az alkalmazás, de még bármikor előfordulhatnak olyan hibák, amikre a fejlesztő nem gondolt és a tesztelés során sem derül fény rá. Ebben az esetben is hasznos lehet a `log` fájl, mivel a felhasználó a hiba jelenésekor csatolhatja a létrejött `log` fájlt, s ekkor a fejlesztő sokkal könnyebben kideríthetí, mi is okozhatta a hibát, ezáltal hamarabb is tudja javítani azt.

A naplózáshoz létezhet egy úgynevezett konfigurációs fájl, ahol megadhatjuk a naplózás tulajdonságait. A konfigurációs fájl lehet `XML`, vagy `Property` (tulajdonság-érték párok) fájl formátumú.

Fontos a naplózás során, hogy nem csak egyszerű információkat szeretnénk kiíratni, mert abból a későbbiekben nem biztos, hogy pontosan meg tudjuk állapítani mi is történt valójában. Fontos a `log`-ban lévő információ pontosságához, hogy legyen hozzárendelve időpont. Ez alapján tudjuk megmondani, hogy az adott kód mikor futott le pontosan. Fontos lehet még, hogy a különböző komponensek különbözőképpen jelenjenek meg, mint ahogy az is, hogy lehessen a naplózást bővíteni, vagy szűkíteni, abban az esetben, ha éppen több, vagy kevesebb információra van szükségünk. Ez alapján lettek a naplózásnak szintek kialakítva. A szintek sorrendben következnek egymás után, és minél nagyobb szintet állítunk be a naplózásnak, annál több információt tartalmaz majd a `log` fájlunk.

2.1.3 A Logger osztály

A `Fly Control` alkalmazásban készítettem egy saját `Logger` osztályt, amiben megtalálhatóak a naplózás alapjai. A `Logger` osztálynak egyetlen, `Singleton` objektuma létezik.

2.1.3.1 A konstruktor

A konstruktor paraméter nélküli, de itt állítjuk be a `setParameters` metódussal a `Logger` paramétereit, s hozzuk létre magát a `log` fájlt, ha szükséges.

2.1.3.2 Főbb metódusok

- `setParameters`
- `createOutputFile`
- `appendFile`
- `log`

2.1.3.3 A setParameters metódus

Ezzel a metódussal állítjuk be a konfigurációs fájl alapján a naplózáshoz szükséges paramétereket.

2.1.3.4 A konfigurációs fájl

A konfigurációs fájl egy `Property fájl`, melyben a következő tulajdonság-érték párok találhatóak:

```
log-level: NONE
screen: ON
directory: "log"
```

2.1.3.5 A log-level tulajdonság

A `log-level` tulajdonságban adhatjuk meg a naplózás szintjét. A `Logger` osztályban 4 szint lett definiálva. Ezek a szintek: `NONE, INFO, DEBUG, ERROR`. A legalacsonyabb a `NONE` míg a legmagasabb szint az `ERROR` szint. Ha a `NONE` értéket adjuk ennek a tulajdonságnak, akkor nem mentődik egyetlen naplóüzenet sem. Ha pedig az `ERROR` értéket kapja ez a tulajdonság, akkor minden egyes naplóüzenet kimentésre kerül.

2.1.3.6 A screen tulajdonság

A `screen` tulajdonság fejlesztői szinten érdekes tulajdonság, mivel ez az összes `log` kiírást kiírja a standard kimenetre is.

2.1.3.7 A directory tulajdonság

A `direcory` tulajdonságnak lehet megadni, hogy a gyökérkönyvtáron belül melyik könyvtárba mentse a `log` fájlokat. Ha nem létezik ez a könyvtár, akkor automatikusan létrejön. A későbbiekben itt találhatja a felhasználó a régebbi `log` fájlokat is.

2.1.3.8 A createOutputFile metódus

A metódus lényege, hogy elkészítse a lemezmeghajtónkon a `log` fájlt. Amennyiben a naplózás szintje `NONE` értékre van beállítva, abban az esetben nem is hozzuk létre. A fájl neve egy jól kitalált módon az aktuális időhöz köthető, s a következő névvel jön létre: log-YYMMDDHHMMSS.txt, ahol Y az évet, M a hónapot, D a napot, H az órát, az újabb M a percet és az S a másodpercet jelenti. Mindezt persze az aktuális idő alapján.

2.1.3.9 Az appendFile metódus

A szerepe, hogy a paraméterül kapott `String`-et kiírja a log fájlba.

2.1.3.10 A log metódus

A `Logger` osztály statikus `log` metódusával írathatunk ki adatokat a program aktuális állapotáról a `log` fájlba. Ez a metódus kétféle paraméterezéssel hívható. Az egyik esetben meg kell adnunk a naplózás szintjét, és azt a `String`-et, amit ki szeretnénk íratni. Több adat kiíratása, a tagok összekonkatenálásával tehető meg. A `log` metódus másik fajtája, amikor csak egy `String`-et adunk meg paraméterként. Ekkor ugyanaz a kód fut le, mint az első esetben, annyi különbséggel, hogy a naplózás szintje a `log-level` tulajdonságban beállított érték lesz. Egy ilyen `log` metódus meghívásával állítódik össze a `log` fájl egy rekordja.

2.1.3.11 A log fájl kinézete

A `log` fájl rekordokból épül fel. Egy rekordban szerepel a naplózás szintje, a játék indulásától eltelt idő, és maga a naplózandó szöveg.

Példa egy ilyen rekordra:

```
[INFO] 00:00:01 Ez egy naplózott szöveg
```

2.2 Lokalizáció

A lokalizáció nem más, mint a szoftver lefordítása egy, vagy több nyelvre. Ezt azonban többféle módon értelmezhetjük. Jelentheti a szoftverben használatos összes szöveg, vagy kép, de akár a stílus honosítását is.
Ha egy szoftver olyan felhasználói rétegnek készül, akik között lehetnek eltérő nyelvűek, akkor érdemes fontolóra venni, hogy kiszolgáljuk-e a több nyelvű közönséget. Amennyiben amellett döntünk, hogy szükséges a szoftver több nyelven való publikálása, akkor azt többféle módon tehetjük meg. Ennek egyik fajtája, hogy magában a programkódban alakítjuk ki hozzá a megfelelő struktúrát. Ekkor elég egy szöveges nyelvi fájlt szerkeszteni, s máris több nyelvűvé tehetjük szoftverünket. Ez a nyelvi fájl lehet egy egyszerű `text` fájl, vagy `XML` is.

A fent említett módszernek hátránya lehet, ha az alkalmazásban vannak olyan képek, amelyeken valamilyen szöveg jelenik meg, mivel ezek szintén tartalmaznak fordítandó szövegeket. Ekkor magát a képet kell lefordíttatnunk, s a programkódot is fel kell készítenünk az ilyen eset kezelésére.

2.2.1 A lokalizációs fájl

A `Fly Control` nevű játékban a lokalizációs fájl egy `XML`, melyben a különböző szövegek egy-egy azonosítóval vannak ellátva. Ezzel az azonosítóval hivatkozunk a programkódban a megadott szövegekre. Amennyiben új nyelvre akarjuk lefordítani a játék szövegét, létre kell hoznunk egy új, megfelelően formázott, s minden paraméterét jól beállított lokalizációs fájlt, s bemásolni a `locale` könyvtárba. Ezt bárki megteheti figyelve arra, hogy mindent megfelelően állítson be, mert csak ekkor fogja azt az eredményt kapni, amit elvár. Így akár a világ bármely nyelvére lefordítható a játék.

2.2.1.1 Részlet a magyar lokalizációs fájlból:

```
<?xml version='1.0' encoding='UTF-8'?>
<Datas id='2' langName='Magyar'>
     <Data id='1' str='Játék'/>
     <Data id='2' str='Beállítások'/>
     <Data id='3' str='Főmenü'/>
     <Data id='4' str='Kijelentkezés'/>
     ...
</Datas>
```

A fent található lokalizációs fájlrészletből látható, hogy a fájl, az `XML` szerkesztés szabályainak megfelelően van kialakítva. A fájl UTF-8 karakterkódolásban van. Az UTF-8 kódolás egyszerre támogatja a magyar, az angol, görög, kínai, és sok más egyéb különleges karakterek megjelenítését, és így minden nyelv egységes kódolást tud használni. A `Datas` gyökérelemnek két attribútumot kell megadnunk. Ezek közül az első az `id,` mely maga a nyelvi fájl azonosítója. Ha új nyelvi fájlt akarunk létrehozni, akkor ezt mindenképpen olyan értékre kell beállítanunk, ami még nem szerepel egy másik lokalizációs fájlban sem. Ez alapján történik ugyanis a programkódban minden egyes hivatkozás az aktuális nyelvre, így

nem érdemes változtatnunk sem. A másik attribútum a `langName`, amivel a lokalizációs fájl nevét határozhatjuk meg. Ezzel a névvel fog szerepelni minden egyes nyelvre való hivatkozáskor, az alkalmazáson belül. Látható, hogy a `Datas` elemnek vannak gyermek elemei. Ezek a `Data` elemek. A `Data` elemmel lehet létrehozni a lokalizált `String`-eket. Két attribútumot kell megadnunk, melyből az első szintén egy `id`, amely a lokalizált `String` azonosítója lesz. A másik attribútuma pedig az `str`, ami értelemszerűen maga a lokalizált szöveg. Az `id`-knek egyedinek kell lenniük, mivel ezekkel hivatkozunk a programkódban a lokalizált szövegekre. Minden szövegnek jól meghatározott `id`-je van, így nem cserélhetjük őket össze. Amennyiben olyan szöveget akarunk létrehozni, ahol paraméter szerepel a szövegben, azt is megtehetjük. Ha egész számot szeretnénk szerepeltetni a szövegben, a paraméter helyére %d-t kell írnunk, s így a megfelelő érték fog majd a paraméter helyére kerülni. Értelemszerűen, ha mondjuk szöveget szeretnénk kiíratni, akkor %s-et kell írnunk, egyébként pedig a programozásban megszokott jelöléseket használjuk.

2.2.2 A Loc osztály

A `Loc` osztály a lokalizációs fájl beolvasására, és a nyelvek kezelésére létrehozott osztály. A fájl feldolgozásához szükséges volt egy `XML` beolvasó metódus megírása. Az XML-ek beolvasására egy egész csomagot hoztam létre, melynek neve `fly.xml` lett. Az ebben a csomagban szereplő `MyXmlParser` osztály segítségével, mely egy `org.xml.sax.helpers.DefaultHandler`[3] osztálynak a leszármazottja, valósítottam meg az XML elemzését. Mivel az XML-ben a lokalizált szövegekhez tartozik egy jól definiált, és egyedi azonosító, ezért a tárolásukra egy kulcs-érték párt tartalmazó `HashMap` objektumot használtam. A kulcs egy Integer objektum, míg az érték egy `String` objektum. Neve: `locMap`. Ahhoz, hogy az új nyelvekre való lokalizáció egyszerű legyen, nem a programkódban kellett meghatározni a nyelvek azonosítóját, hanem be kellett építeni magába a lokalizációs XML dokumentumba. Ez a `Datas` elem `id` attribútuma. A programban viszont tudnunk kell, mely nyelvek léteznek. A nyelvek egy `java.util.ArrayList`-ben vannak tárolva, melyben `Language` típusú objektumok

[3] DefaultHandler osztály - http://java.sun.com/j2se/1.4.2/docs/api/org/xml/sax/helpers/DefaultHandler.html

szerepelnek. Ennek neve: `langs`. Írni kellett tehát egy olyan metódust, ami feltölti ezt a `langs ArrayList`-et, amiben ugyebár az alkalmazásban használt összes nyelvet tároljuk. Ez a metódus végignézi a `locale` könyvtárban található összes fájlt, s ha mindent rendben talált az aktuális `XML` olvasása során, akkor létrehoz egy `Language` objektumot, s beleteszi a `langs ArrayList`-be. A `loadAllLocFile` metódussal kaphatjuk meg a várt eredményt, s ebben a metódusban van meghívva egy másik metódus, a `makeLangByXML`, ami a fent említett eljárást reprezentálja.

2.2.2.1 Főbb metódusok

- `loadAllLocFile`
- `makeLangByXML`
- `loadLocFile`
- `ls`

2.2.2.2 A loadAllLocFile metódus

```
public static void loadAllLocFile()
{
    File f = new File(localeDir);
    if(f.isDirectory())
    {
        File [] files = f.listFiles();
        if(files != null)
        {
            for(int i = 0; i < files.length; i++)
                if(files[i] != null)
                {
                    String src = localeDir;
                    if(src != null && src.compareTo("") != 0)
src += "\\";
                    Language l =
makeLangByXML(src+""+files[i].getName());
                    if(l != null)
                    {
                        if(langs == null)
langs = new ArrayList<Language>();
                        langs.add(l);
                    }
                }
        }
    }
}
```

2.2.2.3 A makeLangByXML metódus:

```
public static Language makeLangByXML(String src)
{
    try
    {
    MyXmlParser parser = new MyXmlParser();
    ArrayList<MyElement> v = parser.parse(src);
    if(v != null)
        for(int i = 0; i < v.size(); i++)
        {
            MyElement element = v.get(i);
            if(element.getQname() != null &&
element.getQname().compareTo("Datas") == 0)
            {
                MyAttributes atts = element.getAttributes();
                try{
                    int id = new Integer(atts.getValue("id"));
                    String value = atts.getValue("langName");
                    return new Language(id, src, value);
                }
                catch(Exception e){
                    Logger.log("Loc - makeLangByXML - 1 - " +
e.getMessage());
                }
            }
        }
    }
    catch(Exception e){
        Logger.log("Loc - makeLangByXML - 2 - " + e.getMessage());
    }
    return null;
}
```

Miután tudjuk, hogy mely nyelveket használhatjuk, a felhasználó a játékba való belépésekor az általa korábban kiválasztott nyelvet kell betöltenünk. Ha még nem volt korábban kiválasztott nyelve, mert mondjuk először lép be egy új játékos, akkor a `getDefaultLocalization` metódus által meghatározott nyelvet töltjük be. Ez minden bizonnyal az angol lesz. Egy nyelv betöltése valójában a `locMap` változó feltöltése adatokkal. Ennek betöltésére a `loadLocFile` metódus szolgál.

2.2.2.4 A loadLocFile metódus

```
public static void loadLocFile(Language lang)
{
    try
    {
        locMap.clear();
        MyXmlParser parser = new MyXmlParser();
        if(Gamelogic.player != null && Gamelogic.player.getOptions()
!= null)
            lang = Gamelogic.player.getOptions().getSelectedLang();
        else if(lang == null)
            lang = getDefaultLocalization();
        ArrayList<MyElement> v = parser.parse(lang.getSrc());
        if(v != null)
            for(int i = 0; i < v.size(); i++)
            {
                MyElement element = v.get(i);
                if(element.getQname() != null &&
element.getQname().compareTo("Data") == 0)
                {
                    MyAttributes atts = element.getAttributes();
                    try{
                        int id = new Integer(atts.getValue("id"));
                        String value = atts.getValue("str");
                        Logger.log("Loc: id = " + id + " value = " +
value);
                        locMap.put(id, value);
                    }
                    catch(Exception e){
                        Logger.log("Loc - loadLocFile - 1 - " +
e.getMessage());
                    }
                }
            }
        Logger.log("Lokalizácio OK");
    }
    catch(Exception e){
        Logger.log("Loc - loadLocFile - 2 - " + e.getMessage());
    }
}
```

2.2.2.5 A lokalizált szövegek kezelése

Ha egy olyan szöveget akarunk elhelyezni a programkódban, amit lokalizálni szeretnénk, akkor azt a következő módon tehetjük meg:

```
Loc.ls("1$|Play");
```

Látható, hogy ha egy szöveget lokalizálni akarunk, akkor azt úgy tehetjük meg, hogy a `Loc` osztály `ls` metódusának egy jól definiált módon adunk meg egy `String`-et. Ez a jól definiáltság a következőt jelenti. Három részből áll a paraméterül adott `String`. Az első rész a szöveg azonosítója, majd következik a „$l” elválasztó jel, s ezután maga a lokalizálandó szöveg. Ennek a három résznek a konkatenációja adja a paraméterül adható `String`-et.

2.2.2.6 Az ls metódus

```
public static String ls(String str)
{
    int index = str.indexOf("$|");
    if(index == -1) return str;
    String keyStr = str.substring(0, index);
    String valueStr = str.substring(index + 2);
    try{
        Integer i = new Integer(keyStr);
        if(locMap.containsKey(i))
            return locMap.get(i);
        return valueStr;
    }
    catch( Exception e)
    {
        Logger.log("Loc - ls - " + e.getMessage());
        return valueStr;
    }
}
```

2.2.2.7 Az ls metódus működése

Az `ls` metódus tehát a `Loc` osztály egy statikus metódusa, s egy `String` objektumot vár paraméterül.

```
public static String ls(String str)
```

Megkeressük a „$l” elválasztó jel indexét:

```
int index = str.indexOf("$|");
```

Amennyiben nem találtuk meg az elválasztó jelet, visszatérünk a paraméterül kapott szöveggel. Az elválasztó jel sikertelenségét az index lokális változó -1 értéke mutatja.

```
if(index == -1) return str;
```

Ezek után szét kell bontanunk a szövegünket, megkeresvén a lokalizált szöveg azonosítóját, és magát a lokalizálandó szöveget.

```
String keyStr = str.substring(0, index);
String valueStr = str.substring(index + 2);
```

Mivel az azonosítónk jelenleg még csak szövegesen létezik, ezért át kell alakítanunk `Integer` típusúvá. Ezt egy `try-catch`-ben tesszük meg, mivel ha nem megfelelő az azonosítónk, akkor `NumberFormatException`-t kaphatunk. Ezután megkeressük a `locMap` változóban a `containsKey` metódussal, hogy tartalmazza-e a változó az ezzel az azonosítóval ellátott elemet. Amennyiben megtalálta, úgy visszaadja a metódus az ehhez az azonosítóhoz tartozó lokalizált szöveget, de ha nem találja meg, akkor visszaadja azt a szöveget, ami a „$l” elválasztó jel után található. Ha valamilyen kivétel dobódna, ami lehet `NumberFormatException`, vagy akár `NullPointerException` is (a `locMap` lehet „NULL”), akkor is a „$l” elválasztó jel utáni szövegrésszel tér vissza az `ls` metódus.

```
try{
     Integer i = new Integer(keyStr);
     if(locMap.containsKey(i))
         return locMap.get(i);
     return valueStr;
}
catch( Exception e)
{
Logger.log("Loc - ls - " + e.getMessage());
return valueStr;
}
```

2.3 Alapkomponensek[1]

A `javax.swing` csomagját használva kialakítottam egy saját csomagot, ennek neve a `basecomponents`, melyben az összes általam használt `javax.swing` csomagbeli osztályt felüldefiniáltam. Ez azért kellett, mert szükségem volt a `paintComponent` metódusok felüldefiniálására, és a felbontás váltás miatt a `setBounds`, és `setSize` metódusok átalakítására. Ezek helyett létre lett hozva egy új `setMyBounds`, és `setMySize` metódus, amely meghívja az ősosztálybeli megfelelőeket, a `Config` osztályban definiált konstansok alapján, amelyek a felbontást (a képernyő szélességét, és hosszúságát) tárolják.

2.3.1 A BaseFrame osztály

Egy `BaseFrame` objektum a játék egyetlen `Frame`-jének alapja. A `JFrame` objektumból származik, ezáltal minden tulajdonságával rendelkezik. A `Gamelogic` osztály `init` metódusában hozzuk létre, s a `Gamelogic` `baseFrame` változójában tároljuk.

2.3.1.1 A konstruktor

Paraméter nélküli, de a fontosabb tulajdonságokat itt állítjuk be. A méretét, a láthatóságát, és azt, hogy nem méretezhető a `Frame`.

2.3.2 A BaseWindow osztály

Ez az osztály szolgál a játékban használatos ablakok kezelésére. Ha egy új ablakot akarunk létrehozni, ezt az osztályt kell felüldefiniálni. Maga is egy `JPanel` objektum, ezáltal rendelkezik minden `JPanel` tulajdonsággal is.

[1] Bátfai Norbert: Programozó Páternoszter, Debrecen, 2005, 147.oldal

2.3.2.1 A konstruktor

Paraméter nélküli, de a fontosabb tulajdonságokat itt állítjuk be. A mérete a `Config` osztályban definiált maximum értékekre áll be, kap egy háttérszínt, ami azért fontos, mert ha nem adunk meg a `setImage` metódussal háttérképet, akkor ez jól látszik, hogy pótolnunk kell. Kap egy `null layout`-ot, majd meghívódik az `initializeComponents` metódus.

2.3.2.2 Főbb metódusok

- `setTitle`
- `setImage`
- `initializeComponents`
- `destroy`
- `openPopup`
- `closePopup`
- `animIn`

2.3.2.3 A setTitle metódus

Ezzel tudjuk beállítani a `Gamelogic` osztály `baseFrame` objektumának címét.

2.3.2.4 A setImage metódus

A célja, hogy a `BaseWindow` objektum háttérképét beállítsa. Kétféleképpen tudjuk hívni ezt a metódust. Az egyik, amikor magát az Image objektumot kapja paraméterül, a másik pedig, amikor az Image objektum elérési útvonalát. Ez a háttérkép minden `repaint` metódushíváskor a `paintComponent`-ben hívott `drawBackground` metódusban rajzolódik ki.

2.3.2.5 Az initializeComponents metódus

Minden egyes `BaseWindow` objektum leszármazottja felüldefiniálja ezt a metódust. Ebben adhatjuk hozzá az ablakon megjelenítendő komponenseket.

2.3.2.6 A destroy metódus

Az ablakon használt összes gyermek objektumot megszünteti. Általában ablakváltáskor van rá szükség.

2.3.2.7 Az openPopup metódus

Lehetőség van felugró ablakok kezelésére a játékban, s ezt is `BaseWindow` objektummal lehet megtenni. Amennyiben van aktuális ablakunk, s erre akarunk feldobni `popup`-ot, akkor a technikai oldal a következő. Kigyűjtjük az aktuális ablak összes komponensét, mert ugyebár ezeket le kell tiltanunk, hogy ne legyenek kattinthatóak, míg egy feldobott ablak létezik. Végigmegyünk egy `for` ciklussal ezeken az objektumokon, s megvizsgáljuk, hogy egyáltalán engedélyezett-e ez az objektum, s ha igen, akkor kigyűjtjük egy `actWindowComps` vektorba, és letiltjuk. Majd a `popup`-unkat hozzáadjuk az aktuális ablakunk komponenseihez.

2.3.2.8 A closePopup metódus

Ez a metódus az `openPopup` metódussal létrehozott `popup` bezárására szolgál. Megvizsgálja, hogy van e komponens `actWindowComps` vektorban, s ha talált, akkor végignézi egy `for` ciklussal, majd engedélyezi mindet. Ugye ezt megteheti, mivel csak olyat raktunk bele ebbe a vektorba, ami előtte engedélyezett volt.

2.3.2.9 Az animIn metódus

Létrehozhatunk tehát felugró ablakokat a játékban, de ha azt szeretnénk, hogy ezek animálva jelenjenek meg, arra szolgál az `animIn` metódus. Négyféle animációt támogat a rendszer, méghozzá a négy irányból történő beúszást. A metódusnak három paramétert kell megadnunk, az első az irány, hogy honnan ússzon be, a másik kettő pedig a startpozíció két `x` és `y` értéke. A startpozíciót mindig úgy határozzuk meg, hogy hol az a hely, ahol még éppen nem látszik a felugró ablak. Van két konstans, amik az animáció tulajdonságát határozzák meg: `ANIMPIXEL`, `ANIMMILLISEC`. Az `ANIMPIXEL` határozza meg, hogy hány pixellel toljuk el az ablakot, amint letelt `ANIMMILLISEC` idő. Ha letelt ez az idő kiszámoljuk az új pozíciót

az iránynak megfelelően, s újrarajzoltatjuk a `repaint` metódussal. Mindezt addig csináljuk, míg az ablakunk el nem éri az általunk korábban meghatározott pozíciót. Ezzel kész is egy beúszó ablak.

2.4 Adatfájlok kezelése

A játékban használt adatok tárolására adatfájlokat kellett létrehozni. Ilyen adatok lehetnek a felhasználók, a beállítások vagy akár a legjobb pontszámok adatai. Az adatfájlok kimentése, és beolvasása során szükséges volt az adatok egyfajta titkosítására, hogy bárki ne láthassa, és ne szerkeszthesse ezeket. Ha nem lennének titkosítva, akkor könnyen láthatnánk akár mások jelszavát, s ez visszaélésekre is adhatna alapot. Az adatfájlok bináris fájlként lettek létrehozva, melyekben egész számokat tárolunk, s az adatok egyfajta Caesar titkosítást használva vannak tárolva. Merül fel a kérdés, hogy a karakterek titkosítása hogyan zajlik. Erre lett létrehozva a játékban használt Caesar titkosítás.

2.4.1 Az alkalmazásban használt Caesar titkosítás[5]

Karakterek kódolása esetén minden kódolandó karakterhez hozzárendeljük a karakterkódját, és létrehozunk két kulcsot, melyet két egész szám reprezentál. Először a karakterkódot megszorozzuk az első kulccsal, majd hozzáadjuk a második kulcsot, ez lesz az új karakterkód, s ezt kell tárolni az adatfájlban.

Dekódolás esetén ennek a módszernek az ellenkezőjét használjuk. Ez alapján, ha beolvasunk egy egész számot, s tudjuk hogy ez egy karakter eltolt kódja, akkor annak az eredeti értékét úgy kaphatjuk meg, hogy először kivonjuk a számból a második kulcs értékét, majd elosztjuk az első kulcs értékével, s meg is kaptuk az eredeti karakter kódját.

[5] Caesar titkosítás - http://www.inf.unideb.hu/~bujdoso/kurzusok/halogyak/kisea/Titkositas_modszerei_dia.ppt

2.4.2 A játékban használt adatfájlok:

- Regisztrált felhasználók adatfájlja
- Legjobb pontszámok adatfájlja
- Játékosok beállításai

2.4.3 A regisztrációs fájl

Az alkalmazásban lehetőség van újabb és újabb felhasználók regisztrálására. Ehhez meg kell adni egy felhasználónevet, és egy jelszót. Erre egyrészt azért van szükség, mivel rangsor készül a játékosok által elért pontokból, s tudnunk kell a felhasználók neveit, másrészt a jelszó azért szükséges, hogy a felhasználók ne férhessenek hozzá egymás adataihoz. A felhasználók nyilvántartása egy bináris fájlban történik (`users.dat`). Egy regisztráció sikeres, ha a következő feltételek teljesülnek:

- Kitöltötte a felhasználónév mezőt
- Kitöltötte a jelszó mezőt
- Még nincs ilyen nevű felhasználó
- A felhasználónév és a jelszó nem tartalmaz „#” karaktert

Látható, hogy a felhasználónév, és a jelszó mező nem tartalmazhat egy speciális karaktert, még hozzá a „#”-et. Ennek technikai okai vannak:
Az adatfájlban megadott sorrendben vannak elhelyezve az adatok. Legelőször az eddig legutoljára regisztrált játékos azonosítója szerepel, mivel ebből tudjuk kinyerni azt az információt, hogy melyik legyen a legközelebb regisztrált játékos azonosítója. Ezután vannak felsorolva az eddig regisztrált felhasználók, a következő adatsorrendben:

- A játékos azonosítója
- A játékos felhasználóneve
- A játékos jelszava

A játékos felhasználóneve és a jelszava is karakterterekből áll, így ezeket titkosítani kell. Erre használjuk a Caesar titkosítást. Ha azonban szimplán egymás után írnánk a felhasználónév és a jelszó kódolt karaktereit, akkor nem tudnánk kideríteni, hogy hol is közöttük a határ. Erre a célra szolgál a speciális karakter a „#”. Ezt a speciális karaktert hozzáfűzzük a felhasználónévhez és a jelszóhoz is, s beolvasáskor tudjuk, ha elértük ezt a karaktert, akkor odáig tart az aktuális szöveg. Természetesen a speciális karakter is kódolva jelenik meg az adatfájlban, és ezért nem szerepelhet a felhasználónév és a jelszó karakterei között sem.

2.4.4 A legjobb pontszámok fájl

A legjobb pontszámok szintén adatfájlban tárolódnak. Ennek neve `highScores.dat`. Egy rekord ebben a fájlban a következőképpen épül fel:

- A játékos azonosítója
- A pálya azonosítója, amelyen ezt a legjobb pontot elérte
- Az időpont millisecond-ben tárolva.
- Az elért pontszám

Ebben az adatfájlban a 20 legjobb eredményt tároljuk, pontszámok alapján rendezve. Itt nincs szükség karakterek titkosítására, mivel csak egész számokat tárolunk. A highScores.dat-ban a legelső adat itt is megkülönböztetett, mivel ebben tároljuk a játék éppen aktuális verziószámát. Erre azért van szükség, mert ha új kérés, vagy áttervezés alapján megváltozik a legjobb pontszámok adatszerkezete, akkor a verziószám alapján akár régebbi adatokat is be tudunk olvastatni a rendszerrel.

2.4.5 A beállítások fájl

A beállításokat szintén adatfájlban tároljuk. Az előzőektől eltérően azonban itt minden egyes felhasználó adata külön fájlba kerül. Ha regisztrál egy játékos, automatikusan létrejön számára egy `options` fájl. A fájl neve a játékos felhasználónevéhez hozzáfűzött `_option.dat` fájlnév lesz. Így az `abc` felhasználó adatfájlja `abc_option.dat` lesz.

Egy option fájlban a következő adatokat tároljuk sorrendben:

- Lokalizációs azonosító
- Aktuálisan kiválasztott pálya azonosítója
- A kezdő szint azonosítóját

Az option fájlok beolvasása és kiírása során felmerülhetnek különböző problémák. Ilyen lehet az, hogy nem létezik már az a lokalizációs fájl, aminek az azonosítója ki lett mentve. Vagy másik ilyen lehetséges probléma, hogy maga a pálya nem létezik már. Ezek kiküszöbölésére biztonságosan lett megírva a programkód, ilyenkor mindig egy létező adatot tölt be a megfelelő helyre.

Az adatfájlok tehát teljesen a játék által generálódnak. Ha nem létezik egy, akkor az automatikusan létrejön, ha hivatkozás van rá. Persze ehhez feltétlenül szükséges, hogy az aktuális könyvtárba írási joga legyen a felhasználónak!

3

3 A játék főbb funkciói

3.1 Pályák

A játék szerves elemei a pályák. Egy pálya határozza meg azt a környezetet, amit a játékos a játékmezőre lépve tapasztal. A látvány több részből tevődik össze:

- A háttérkép
- Az irányítható objektumok képei
- A nem irányítható objektumok képei

3.1.1 A pályát leíró XML

Ahhoz, hogy egy pályát létre tudjuk hozni, szükséges egy adatfájl, amiben megtalálható az összes forrásra való hivatkozás, ami egy pálya létrehozásához szükséges. Ez az adatfájl lesz a `track.xml`. Minden egyes pályához külön leíró `XML` tartozik. A létrehozáshoz szükséges adatok a következők:

- A pálya azonosítója
- A háttérkép elérési útvonala
- A háttérképen található célpont középpontjának koordinátái
- A pálya neve

- Az irányítható, és a nem irányítható objektumok képeinek elérési útvonala

3.1.1.1 Példa pálya XML-re:

```
<?xml version="1.0"?>
<Track id='1' bgSrc='tracks\\track1\\bgimg.jpg' targetX='250'
targetY='430' trackName='track01'
     flyAnimMoveableDir='tracks\\track1\\flymoveable'
flyAnimNotMoveableDir='tracks\\track1\\flynotmoveable'>
</Track>
```

A célpont koordinátáit a 800 x 600-as felbontás szerint kell megadnunk, s a rendszer méretezi magának az aktuális felbontás alapján. A példa `XML` alapján látható, hogy egy pálya határozza meg azt is, hogy milyen irányítható, és nem irányítható objektumok szerepeljenek rajta. Ez azért fontos, hogy lehessen többféle környezetű pályát létrehozni. Gondolok itt arra, hogy egy szobához a benne repkedő bogarak illenek, míg az akváriumhoz a halak.

3.1.2 Új pálya létrehozása

Látszik, hogy a pályák kialakítása úgy történt meg, hogy mindent adatfájlból tölt be. Ez azért jó, mivel könnyedén hozhatunk létre új pályákat. Ehhez a következőket kell tennünk. Szükségünk van egy új azonosítóra, mindenképpen olyanra, ami még nem létezik. Ha véletlenül olyan azonosítót adnánk meg, ami már létezik, akkor nem fog megjelenni a játék pályái között, és a `log` fájlban megtalálható lesz egy figyelmeztetés is, hogy miért is nem jelenik meg a pálya. Ezentúl szükségünk van egy új háttérképre, s azon egy célpontra, amit koordinátákkal adunk meg. Kitaláljuk a pályánk nevét, ami bármilyen fantázianév lehet. Végezetül létre kell hoznunk az irányítható, és a nem irányítható objektumok animációjához szükséges textúrákat. Ha ezekkel megvagyunk, akkor hozzunk létre egy könyvtárat úgy, hogy a neve `track[ID]` legyen, és az `[ID]` helyére a pálya azonosítója kerüljön. Ha megvagyunk, másoljuk bele a pálya háttérképét és nevezzük át `bgImg`.jpg-re. Persze a kiterjesztés fontos, hogy megfelelő legyen. Szükségünk van még két könyvtárra a `track[ID]` könyvtáron belül. Ezek neve: `flymoveable` és `flynotmoveable`. Ide kell bemásolnunk az irányítható és nem irányítható objektumok animációihoz szükséges képeket.

Ezeknek a képeknek a nevei bármi lehet. Már csak egy dolgot kell tennünk, létre kell hoznunk a pályaleíró XML-ünket. Az XML id attribútuma legyen a pálya azonosítója, a bgSrc attribútum a pálya háttérképének elérési útvonala, a targetX a célpont x koordinátája, a targetY a célpont y koordinátája, a trackName attribútum a pálya neve, a flyAnimMoveableDir attribútum az irányítható objektumok könyvtárának elérési útvonala, míg a flyAnimNotMoveableDir a nem irányítható objektumok elérési útvonala. Ha létrehoztuk a leíró XML-ünkent, bemásoljuk a track[ID] könyvtárba s ezzel kész is van egy új pálya. Ha ezt a könyvtárt bemásoljuk a tracks könyvtárba, és mindent a leírtaknak megfelelően csináltunk a pálya meg fog jelenni a játékban.

Ezzel a módszerrel azt próbáltam elérni, hogy könnyen bővíthető legyen új pályákkal a játék, és ha kereskedelmi forgalomba kerülne ez a játék, akkor bárki tudjon magának új pályát létrehozni. Sőt, ha a játéknak lenne külön weboldala, akkor ott folyamatosan lehetne feltölteni újabb és újabb pályákat.

3.2 Szintek

A játékmenet során, a játékos szinteken keresztül juthat előre. 20 szintet különböztetünk meg, s mindegyik szinthez tartozik egy azonosító. Ezek rendre 1-20-ig egész számok. A kezdőszint az 1-es azonosítóval rendelkező. A szintek azt jelentik, hogy a játék egyre jobban nehezedik, ahogy egy játékos egy új szintre jut.

Szintek kezelésére a Level osztály szolgál. Létrehozásuk a Gamelogic osztály init metódusában történik meg a Level osztály statikus createLevels metódusának meghívásával.

3.2.1 A Level osztály

3.2.1.1 A konstruktor

Egy Level objektum konstruktorának az alábbi paramétereket kell megadnunk:

- Azonosító
- Az irányítható bogarak mennyi időközönként jelenjenek meg (sec)
- A nem irányítható bogarak mennyi időközönként jelenjenek meg (sec)
- Az irányítható bogarak sebessége
- A nem irányítható bogarak sebessége

Ezekkel a paraméterekkel már elég eltérő és változatos szinteket lehet létrehozni.

3.2.1.2 Főbb metódusok

- `createLevels`
- `getLevel`

3.2.1.3 A createLevels metódus

```
public static void createLevels()
    {
        levels = new ArrayList<Level>();
        levels.add(new Level(1, 5, 1000, 60, 0));
        levels.add(new Level(2, 5, 1000, 70, 0));
        levels.add(new Level(3, 4, 1000, 80, 0));
        levels.add(new Level(4, 4, 1000, 90, 0));
        levels.add(new Level(5, 4, 1000, 100, 0));
        levels.add(new Level(6, 4, 20, 110, 40));
        levels.add(new Level(7, 4, 12, 120, 40));
        levels.add(new Level(8, 4, 9, 80, 40));
        levels.add(new Level(9, 4, 9, 90, 50));
        levels.add(new Level(10, 4, 9, 100, 80));
        levels.add(new Level(11, 4, 8, 90, 70));
        levels.add(new Level(12, 4, 8, 100, 70));
        levels.add(new Level(13, 4, 5, 100, 60));
        levels.add(new Level(14, 4, 5, 110, 70));
        levels.add(new Level(15, 3, 5, 120, 50));
        levels.add(new Level(16, 3, 4, 110, 60));
        levels.add(new Level(17, 3, 4, 120, 70));
        levels.add(new Level(18, 3, 3, 100, 70));
        levels.add(new Level(19, 3, 3, 110, 90));
        levels.add(new Level(20, 2, 2, 100, 90));
    }
```

Látható, hogy különböző paraméterezéssel egyre nehezebb szintek hozhatóak létre.

3.2.1.4 A getLevel metódus

A `Level` osztály statikus változója a `levels ArrayList`, mely `Level` objektumokat tartalmaz. Ha egy szintre akarunk hivatkozni, akkor meg kell hívnunk a `getLevel` metódust, mely ebből a `levels` vektorból keresi ki azonosító alapján a hivatkozandó szintet.

3.2.2 Magasabb szintek

Ha egy játékos úgy dönt, hogy nem az első szinttel szeretné kezdeni a játékot, hanem ki szeretné próbálni milyen is a játék magasabb szinteken, akkor azt megteheti a beállítások menüben. Ekkor a játékos saját beállításai megváltoznak, és beállítódik a kezdőszintje az általa kiválasztottra. Ha újraindítja a játékot, akkor is ezzel a szinttel fog kezdeni mindaddig, míg át nem állítja egy újabb értékre.

3.3 Pontozás

A pontozás több részből tevődik össze. Részben a játékmezőn célba juttatott objektumokért lehet pontot szerezni, másrészt a játékmezőn folyamatosan megjelenő ajándékokból (Gift) is kaphat plusz pontot a játékos.

A célba juttatott objektumokért járó pont a szintek növekedésével egyre nő. Egy objektum célba juttatásáért kapott pont az alábbi képlet alapján számolódik:

```
Pont = adott szint azonosítója * egy objektumért kapható alappont
```

Az egy objektumért kapható alappont 100 pont. Így például egy harmadik szinten célba juttatott irányítható objektumért 3 * 100 pont szerezhető. A játékos pontja a `FlyManager` osztályban számolódik. A `FlyManager` osztály `completedFly` változója számolja, hogy

hány darab `Fly` objektumot sikerült már célba juttatnunk. A szerzett pont ez alapján, az előző képlet megszorozva a `completedFly` változóval.

3.3.1 A Giftek

A Giftek más néven ajándékok, azt a célt szolgálják, hogy a játékos extra pontokat szerezhessen a játék során. Egy ajándék a játékmező valamely teljesen véletlenszerű pontján jelenhet meg. Az, hogy milyen időközönként, a következő képlet alapján számolódik. Van egy minimális idő, aminek mindenképpen el kell telnie, és ezen felül, hozzáadódik egy véletlenszerűen kiválasztott érték a [0, 2] egész számokat tartalmazó intervallumból. Ez tehát azt jelenti, hogy 5, 6 vagy 7 másodpercenként jelennek meg új ajándékok a játékmezőn. Egy gift 10 másodpercig látszik, és utána eltűnik. Ha ezalatt a játékos az egyik irányítható bogarat úgy irányítja, hogy az ütközésvizsgálat alapján ütközik a gifttel, akkor az ajándékért járó pontérték hozzáíródik a szerzett pontjaihoz.

3.3.2 A Gift osztály

3.3.2.1 A konstruktor

A konstroktor paramétere:

- Az ajándék típusa (int).

A konstruktorban a paraméterben kapott típus alapján beállítódik a maximálisan szerezhető jutalompont, és az ajándék háttérképe is.

3.3.2.2 Főbb metódusok

- `init`
- `setImage`
- `destroy`
- `run`
- `getModifiedRewardPoint`
- `setCompleted`

- `getRandomGift`

3.3.2.3 Az init metódus

Ez a metódus szolgál a `Gift` objektum megjelenítésére. Kap egy véletlenszerű pozíciót a `Gamelogic` osztály `rnd` változóját használva, beállítódik a mérete, és természetesen kirajzolódik a játékmezőn. Fontos még, hogy ebben a metódusban indítjuk el a `run` metódushoz szükséges szálat is.

3.3.2.4 A setImage metódus

Ezzel a metódussal adjuk meg az ajándék típusa alapján, hogy melyik kép legyen az ajándék háttérképe.

3.3.2.5 A destroy metódus

A `running` változó, ami alapján a `run` metódus fut, ebben fog hamis állapotba kerülni. Miután kinulloztuk az `img` változót is, ami a `gift` hátterét adta, elég egy újra színezés s már meg is szűnt a `gift` objektumunk.

3.3.2.6 A run metódus

A `Gift` osztály implementálja a `Runnable` interfészt, méghozzá azért, hogy egy `Gift` objektumnak meg tudjuk mondani mennyi a hátra maradt ideje. Ha a játékosnak nem sikerült felvennie ezt az ajándékot, akkor meghívódik a `destroy` metódus.

3.3.2.7 A getModifiedRewardPoint metódus

Egy ajándékért járó pontmennyiség időarányosan számolódik. Ha a játékos megszerzi az ajándékot, akkor az alábbi képlet alapján számolódik az érte kapható jutalompont:

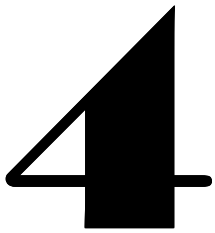
```
Jutalompont = hátralévő idő / maximum idő * maximális jutalompont
```

3.3.2.8 A setCompleted metódus

Ez a metódus több feladatot is ellát. Részben beállítja a `Gift` objektum `completed` változóját igaz értékre, ebből tudjuk, hogy ezt az ajándékot már teljesítette a játékos. Másrészt a `getModifiedRewardPoint` metódus által meghatározott jutalompontot hozzáírja a `FlyManager` ajándékokért kapható jutalom pontjaihoz. Ezután a `destroy` metódussal megszünteti az objektumot.

3.3.2.9 A getRandomGift metódus

Hatféle ajándék létezik, és ha létre akarunk hozni egy új ajándékot, akkor azt a `Gift` osztály statikus `getRandomGift` metódusával tehetjük meg. Ez visszatér egy új teljesen véletlenszerű `Gift` objektummal. Az ajándékokért kapható maximális pont a típus alapján számolódik. Ez azt jelenti, hogy egy alapérték, jelen esetben a 100, alapján a típustól függően, a típus * alapérték lesz a maximálisan megszerezhető jutalompont.



4 A játék főbb osztályai

4.1 A Gamelogic osztály

A `Gamelogic` osztály egy olyan speciális osztály, ahol össze vannak gyűjtve azok a metódusok, és változók, amelyeket vagy nem tudunk egyetlen konkrét objektumhoz sem kötni, vagy teljesen alap feladatokat ellátó metódusról, változóról van szó. Ezért ezek statikusak, s bármely osztály által használhatóak.

4.1.1 Főbb metódusok

- `init`
- `changeScreen`
- `login`
- `logout`

4.1.2 Az init metódus

A neve alapján látszik, hogy ez inicializálja az összes indításhoz szükséges adattagot. Megtalálható benne például a játékosok beolvasása, a játékban található szintek létrehozása, a pályák betöltése, a lokalizációs fájlok betöltése, és az egyik legfontosabb metódus a képernyőváltás (`changeScreen`), mely létrehozza a legelső, bejelentkező képernyőt. A

`Gamelogic` osztály egy statikus változója a `baseFrame` változó, ami itt az `init` metódusban példányosul, s ez lesz a játék alapframe-je. Egy másik fontos változó az `actualWindow`, amely pedig megmondja, hogy melyik képernyő éppen az aktuális képernyő.

4.1.3 A changeScreen metódus

Paraméterként a metódus egy `BaseWindow` objektumot kap, mely az új képernyő lesz. A metódus lényege, hogy a régi (éppen aktuális) képernyő adatait eltüntesse, és az új adatait létrehozza, és kirajzolja. Ehhez az kell, hogy a `baseFrame` objektum összes komponensét meg kell szüntetni, és az új képernyő adatai alapján feltölteni. Itt kap értéket az `actualWindow` változó is, méghozzá a paraméterben kapott képernyő lesz az `actualWindow` új értéke. Amennyiben ugyanolyan ablakot akarunk betölteni, mint amilyen az `actualWindow`, akkor az egész metódus törzse nem fog lefutni, mivel ez nem engedélyezett.

4.1.4 A login és a logout metódus

A `Gamelogic` osztályban lett létrehozva ez a két metódus, melyekkel értelemszerűen a bejelentkezést, és a kijelentkezést tudjuk végrehajtani. A `login` metódusban kap a Gamelogic osztály `player` változója értéket, méghozzá a paraméterül kapott `Player` objektumot. Ezen kívül elvégzi a bejelentkezéshez szükséges fontos teendőket, mint például a játékos beállításainak betöltését, a lokalizációs betöltést, itt hozza létre a ranglistákat, és elvezényel bennünket a főmenübe.
A `logout` metódus pedig megszünteti a `player` objektumot, törli az aktuális lokalizációt, és visszavezényel a bejelentkező képernyőre.

4.2 A Fly osztály

A `Fly` osztály valósítja meg magát a mozgó objektumot a játéktéren. Ezzel az osztállyal hozhatunk létre általunk mozgatható, és nem mozgatható objektumokat. A `Fly` osztály a

`JPanel` osztályt bővíti, és az aktuális ablak tartalmához adjuk hozzá, ha létrehozunk egy új `Fly` objektumot.

4.2.1 A konstruktor

A konstruktor paraméter nélküli. Első fontos teendője, hogy az animációhoz szükséges szálat elindítsa. Itt határozódik meg ezen kívül a `Fly` objektum kezdőpozíciója, és a véletlenszerű irányítás miatt az első célpont pozíció. A konstruktor állítja be a `Fly` objektum méretét is, melynek értékei két `private` változóban tárolódnak (`width_, height_`).

4.2.2 Főbb metódusok:

- `setSpeed`
- `setControllable`
- `equalsPosition`
- `getRandomStartPos`
- `updatePosition`

4.2.3 A setSpeed metódus

Ezzel a metódussal tudjuk beállítani a `Fly` objektum sebességét.

4.2.4 A setControllable metódus

Ez a metódus szolgál annak a beállítására, hogy az adott `Fly` objektum irányítható-e vagy sem.

4.2.5 Az equalsPosition metódus

Paraméterül egy másik `Fly` objektumot kap, s célja eldönteni, hogy a paraméterül kapott `Fly` objektum, és az aktuális `Fly` példány ütközik-e. A `crashRadius` változóban tárolt érték alapján adva van egy sugár, amely alapján az ütközés vizsgálódik. Amennyiben a két `Fly`

objektum középpontjának távolsága kisebb, mint ennek a sugárnak a kétszerese, abban az esetben ütközés történt, s a metódus `true` értékkel tér vissza. Egyébként a metódus visszatérési értéke `false`.

4.2.6 A getRandomStartPos metódus

A `Fly` objektum a képernyő négy oldalán jelenhet meg először, ezért szükséges volt egy olyan metódus megírása, ami egy olyan pozícióval tér vissza, amely a képernyő egy véletlenszerűen kiválasztott oldalának a koordinátáit határozza meg. Erre szolgál a `getRandomStartPos` metódus. Technikai megvalósítása, hogy randomizál két `boolean` típusú változót, s ennek az összes lehetséges kombinációját véve megkapjuk, hogy éppen melyik oldalt jelenjen meg a `Fly` objektum. Innen már csak a megfelelő `x`, vagy `y` koordinátát kell generálnunk, s meg is kaptuk a keresett koordinátát.

4.2.7 Az updatePosition metódus

Ez a metódus szolgál a `Fly` objektumok irányítására, és a véletlenszerű mozgatására, s erről az „Irányítás és ütközésvizsgálat” fejezetben részletesen is olvashatunk.

4.2.8 A Fly objektumok animációja

Az irányítható és nem irányítható objektum megjelenítése animációval történik. Ez azt jelenti, hogy ha megjelenik egy ilyen objektum a játéktéren, akkor az olyan hatást kelt, mintha ténylegesen mozogna. Ehhez az szükséges, hogy meg kellett rajzolni az összes olyan animációs képet, amelyeket meg szeretnénk jeleníteni az animáció során.

A `Fly` objektumok animációja a `Fly` osztályban történik, a `Runnable` interfész `run` metódusának segítségével. A konstruktorban indítjuk el a szükséges szálat, hogy megadott időközönként tudjuk újabb és újabb képeket megjeleníteni, azt a hatást keltve, mintha ténylegesen mozogna.

4.2.8.1 Az animáció technikai oldala

A `Fly` osztály `imgs` vektora tárolja azokat a képeket, amelyek az animáció során megjelenhetnek. Ez a `setImages` metódusban kap értéket, amely a pálya létrehozásakor hívódik. Erre azért van szükség, mivel a pályához szorosan kötődik, hogy milyen `Fly` jelenjen meg rajta. Beolvasása során a pályához tartozó megfelelő könyvtárból felolvassa a képeket, és ezt egy vektorban, az `imgs` vektorban tárolja. Ezáltal akárhány animációs kép elhelyezhető ebben a könyvtárban, mivel az összes képet fel fogja tudni olvasni, s elhelyezni ebben a vektorban. A `Fly` osztály `actualImg` változója tárolja, hogy melyik is az aktuálisan kiválasztott animációs kép, s ezt fogja megjeleníteni. Ennek értékadása a `run` metódusban történik. Háromszáz millisec-enként történik váltás, ennyi időre sleep-eltetjük a szálat, s mindig az `imgs` vektorban következő animációs kép lesz az `actualImg` változó értéke. Miután új értéket kapott az `actualImg` változó, újrarajzoltatjuk a `repaint` metódussal, s innentől ezt a képet fogja megjeleníteni a következő képváltásig. Ha elérte az utolsó elemet a vektorban, akkor ismét az első elemtől kezdi a képek olvasását. Ezzel elértük, hogy animáljon a `Fly` objektumunk.

4.3 A FlyManager osztály

A `FlyManager` osztály adja a játékmenet alapját. Egyszerre csak egy `FlyManager` példány létezhet, ezért `Singleton` objektumként lett létrehozva. Egy ilyen objektumnak akkor szabad létrejönnie, ha új szintre lép a játékos, tehát ez az osztály menedzseli az adott szinten történt változásokat.

4.3.1 Főbb metódusok

- `init`
- `deinit`
- `run`
- `addOneFly`
- `destroyFlys`

- updateFlys
- addRandomGifts, removeGifts, checkGifts
- iniEndGameWindows

4.3.2 Az init metódus

Az init metódus szolgál a játéktér kialakítására, ezért ennek a metódusának meg kell adnunk egy GameWindow példányt, mely a játékmező komponenseit fogja tartalmazni. Szintén az init-ben állíthatjuk be az irányítható és a nem irányítható Fly objektumok megjelenésének időközét. Ezt két változóban tároljuk: flyGapTime és notControlableflyGapTime. Fontos szerepe még, hogy a Runnable interfész miatt szükséges szál elindítása is ebben a metódusban történik.

4.3.3 A deInit metódus

Ha egy szintnek eljutottunk a végére, akár teljesítette a játékos akár nem, meghívódik a deInit metódus, mely tulajdonképpen alapértékekre állítja a FlyManager változóit.

4.3.4 A run metódus

A FlyManager osztály implementálja a Runnable interfészt, melyben többek között figyeli az ütközéseket, új Fly objektumokat helyez el, ha szükséges, figyeli az ajándékozást, és még egyéb dolgokat. A run metódus felelős tehát a FlyManager osztály folyamatos update-jéért. Ötféle idő tárolásra lett létrehozva öt long típusú változó, melyekben kezdetben a System osztály currentTimeMillis metódusának értékei, azaz az éppen aktuális időnek a millisecond-ben megadott értékei tárolódnak. Ezek a váltózók a startTime, lastFlyTime, lastnotMoveableFlyTime, actualTime és a nextGiftTime. A startTime változó értéke nem változik, szerepe abban rejlik, hogy tudjuk, mennyi idő van még hátra az aktuális szintből. Az aktuális időt az actualTime változóban tároljuk, és ha ebből kivonjuk a startTime változó értékét, máris tudjuk mennyi idő telt el a szint kezdete óta. A lastFlyTime változó szolgál arra, hogy tudjuk, mikor adtunk hozzá legutóbb Fly

objektumot. Ha a `flyGapTime` változóban tárolt idő eltelt a `lastFlyTime` óta, akkor kell hozzáadnunk egy újabb `Fly` objektumot az `addOneFly` metódus `true` paraméterrel való meghívásával. Ugyanez a helyzet a nem irányítható `Fly` objektumok hozzáadásánál, csak itt a `lastnotMoveableFlyTime` változóban tároljuk a legutóbb hozzáadott nem irányítható `Fly` objektum idejét, és az eltelt időnek `notControlableflyGapTime`-nek kell lennie. Az ajándékozásra vonatkozó idő a `nextGiftTime` változóban tárolódik, s ha ez megegyezik az aktuális idővel, akkor hozzáadunk egy véletlenszerű ajándékot az `addRandomGift` metódussal. A `run` metódusban kezeljük ezen kívül a `Fly` objektumok, és az ajándékok ütközésvizsgálatát. Ha két `Fly` objektum ütközik, amelyikből értelemszerűen az egyik irányítható, akkor a játékos veszít egyet az életéből, s erre kap egy figyelmeztető ablakot is.

4.3.5 Az addOneFly metódus

Az `addOneFly` metódussal adhatunk újabb `Fly` objektumot a játéktérhez. A `FlyManager` `flys` változójában tároljuk azokat a `Fly` objektumokat, amelyeket hozzáadunk, így a későbbiekben könnyen tudunk rájuk hivatkozni. A `flys` változó egy `ArrayList`, melyben `Fly` objektumok szerepelnek.

4.3.6 A destroyFlys metódus

A `destroyFlys` metódussal tudjuk megszüntetni a korábban már `addOneFly` metódussal létrehozott `Fly` objektumokat. Ez eltünteti játékmezőről is az összes addig hozzáadott `Fly` objektumot.

4.3.7 Az updateFlys metódus

Az `updateFlys` metódussal tudjuk a már korábba létrehozott `Fly` objektumokat update-elni, azaz a vizsgálni a pozíciójukat. Ezt a `Fly` objektum `updatePosition` metódusával tehetjük meg.

Tudunk most már létrehozni, update-elni, és megszüntetni is `Fly` objektumot, már csak az ütközésvizsgálat hiányzik. Az ütközések vizsgálatára a `checkFlyCrash` metódus szolgál.

4.3.8 Ajándékok kezelése

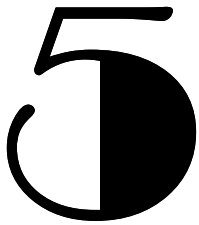
Az ajándékozás kezelése is a `FlyManager` osztályban történik. A giftek tárolására a `gifts ArrayList` szolgál, melyben `Gift` típusú objektumokat tárolunk. Az `addRandomGift` metódussal hozhatunk létre egy véletlenszerű ajándékot, és elhelyezzük a `gifts` vektorban, a `removeGifts` metódussal tudjuk megszüntetni az összes hozzáadott ajándékot, és megszüntetni a `gifts` vektor elemeit. A `checkGifts` metódus szolgál az ajándékokkal való ütközésvizsgálatra. Ha üközés történik, akkor meghívódnak a megfelelő metódusok, kiszámolódik az ajándékért kapott jutalompont, és levesszük az ajándékot a játékmezőről.

4.3.9 Az initEndGameWindows metódus

Ha lejárt a szintre adott idő, akkor az `initEndGameWindows` metódus fut le. Kétféle módon érhet véget szabályosan az aktuális szint. Egyik eset, amikor még van következő szint, másik, amikor már nincs. Ha van még következő szint, akkor a `LevelCompletedGUI` ablak dobódik fel, ha nincs, akkor a `ResultsGUI` ablak. Ezeken az ablakokon tudatjuk a játékossal, hogy milyen pontokat is szerzett a játékban. Mindezeket szintén a `FlyManager` osztály lekérdező metódusai alapján tudhatjuk. Tudjuk azt, hogy mennyi pontja van összesen, és azt is, hogy az aktuális fordulóban mennyi pontot szerzett.

4.3.10 A FlyManager újrainicializálása

Ha létezik újabb szint, s tovább játszunk, akkor hívjuk a `FlyManager` osztály `deInit` metódusát, átváltjuk a `Gamelogic actualLevel` változóját a következő szintre, létrehozzuk az új játékmezőt, s újrainicializáljuk a `FlyManager` osztály `Singleton`-ját, és kezdődhet minden elölről.



5 A Fly objektumok irányítása és az ütközésvizsgálat

5.1 A Fly objektumok irányítása

Az objektumok irányítása csak irányítható `Fly` objektumokra érvényes. Az irányítás az egér folyamatos lenyomásával tehető meg. Amíg nem választunk ki egy irányítható `Fly` objektumot sem, addig azok véletlenszerűen mozognak.

5.1.1 A véletlenszerű mozgás

Egy `Fly` objektum a játéktér valamely véletlenszerű oldalán jelenik meg. Erre szolgál a `getRandomStartPos` metódus, mely visszaadja a start pozíciót. A `Fly` objektum `tempTargetPos` változójában tárolódik egy pozíció, amely megmondja, hogy melyik az a pozíció, ahova a `Fly` objektumnak el kell jutnia. A konstruktorban határozódik meg az első `tempTargetPos` a `getRandomTargetPos` metódussal, s innentől az `updatePosition` metódusban határozódik meg a `Fly` objektum új pozíciója. Ha elérte a `tempTargetPos` változóban lévő célpontot, azaz a `Pos` osztály `equalsPosition` metódusa alapján megegyezik a pozíciójuk, akkor újraszámolódik a `getRandomTargetPos` metódussal a `tempTargetPos` változó, s kezdődik minden elölről.

5.1.2 Két pont közötti út meghatározása[4]

Mivel adott egy startpozíciónk, meg egy célpozíciónk, ezért szükségünk van a közöttük lévő pontokra, mivel ezeken át kell vezérelnünk a `Fly` objektumot. Két pontból egyenes egyenletét, s ezáltal a rajta lévő pontokat, az alábbi képlet alapján határoztam meg:

5.1.3 Az útvonal kijelölés

Az útvonal kijelölése a `GameWindow` osztályban történik. A `GameWindow` osztály a `BaseWindow` leszármazottja, így hozzáadható egy egérmozgást figyelő `MouseMotionAdapter` objektum, melynek a `mouseDragged` metódusában tudjuk figyelni melyek is azok a pontok, amelyeket az egér a mozgatás alatt érint. A `Fly` objektum kiválasztását az egérrel való rákattintással tehetjük meg. A `getFlyOnPos` metódus visszaadja azt a `Fly` objektumot, melyet az egér lenyomásakor érintünk. Ha nem talál ilyet, akkor `null` értékkel tér vissza. Amennyiben talál megfelelő `Fly` objektumot, akkor az aktuális pozíciót hozzáfűzi az erre a célra létrehozott `Fly` objektumbeli `controlPos` vektorhoz. Ebben tárolódnak tehát azok a pontok, amelyeken majd végig kell haladnia. Mivel a `mouseDragged` metódus elég gyakran hívódik, ezért csak akkor fűzünk hozzá új értéket, ha az, az előzőhöz képest minimum 20 pixel távolságra van. Amennyiben újra rákattintunk a `Fly` objektumra, onnantól ez `controlPos` vektor törlődik, s az új adatokkal töltődik fel.

5.1.4 Az útvonal bejárása

A `Fly` osztály `updatePosition` metódusában történik az útvonal bejárása. A `controlPos` vektorban található pozíciók jelentik a következő `tempTargetPos`-t, s az előző módszer alapján történik a pozícióváltás. Amint elfogynak a `controlPos`-ban lévő

[4] Koordináta geometria - http://hu.wikipedia.org/wiki/Koordin%C3%A1tageometria

pozíciók, és nem ért el a célhoz, akkor a `getRandomTargetPos` alapján új értéket kap a `tempTagretPos` változó, és újra véletlenszerűen mozog tovább a `Fly` objektum. Amennyiben a `Pos` osztály `equalsPosition` metódusa alapján egyezne a pálya célpontjával, akkor a `completed` változó igaz értékre áll, jelezve ezzel, hogy sikeresen célba juttattuk, s ezután az `updatePosition` metódus a törzsének már csak az első kiértékelő része fog lefutni, amely azt vizsgálja, hogy ez a `completed` változó hamis-e, s ha igen, akkor befejezteti futását.

5.2 Az ütközésvizsgálat

A játék egyik lényege, hogy a mozgó objektumok, nem ütközhetnek egymással, de ha mégis bekövetkezne, akkor bizonyos eseményeket le kell kezelnünk. Erre szolgál az ütközésvizsgálat. Minden `Fly` objektumnak van a játéktéren egy pozíciója. Erre a célra, hoztam létre a `Pos` osztályt.

5.2.1 A Pos osztály

5.2.1.1 A konstruktor

Két paramétert vár a konstruktor, az egyik az `x` koordináta, a másik az `y` koordináta a játéktéren.

5.2.1.2 Főbb metódusok

- `equals`
- `equalsAlmost`

5.2.1.3 Az equals metódus

Egy `Pos` objektumot vár paraméterül, s ezt hasonlítja össze az aktuális pozícióval. Visszatérési értéke `boolean` típusú. Teljes egyezést ad, azaz ha az `x` és `y` koordináták megegyeznek, csak akkor tér vissza igaz értékkel.

5.2.1.4 Az equalsAlmost metódus

Kétféle paraméterezéssel hívható metódus. Az egyik, amikor egy `Pos` objektum, és egy `int` típusú változó a paraméter. Ekkor szintén a paraméterben kapott `Pos` objektumhoz hasonlítja az aktuális változót, de nem pontos egyezést ad, hanem ha az aktuális pozícióhoz képest, a második paraméterben kapott (pixelben mért) távolságon belül van az új pont akkor tér vissza igaz értékkel a metódus. A másik paraméterezés esetében csak egy `Pos` objektumot kap paraméterül, de ez a metódus az előzőt hívja úgy, hogy az első paraméter ez a `Pos` objektum lesz, míg a távolság paraméter 10 lesz.

5.2.2 Az ütközésvizsgálat

5.2.2.1 A Fly osztály equalsPosition metódusa

Egy paramétert vár, amely egy `Fly` objektum, s az aktuális `Fly` objektumhoz vizsgálja ennek a pozícióját. Ehhez használja a `Pos` osztály `equalsAlmost` metódusát. A távolság a `Fly` osztály `crashRadius` változójában van tárolva. Igaz értékkel tér vissza, ha a két `Fly` objektum pozíciójára nézve az `equalsAlmost` igaz értéket ad, egyébként hamissal.

5.2.2.2 A FlyManager checkFlyCrash metódusa

A `Fly` objektumok ütközésvizsgálata ebben a metódusban történik. Ez a `FlyManager` `run` metódusában hívódik, így folyamatos vizsgálatot jelent. Végignézi az összes játéktéren lévő `Fly` objektumot, és megvizsgálja az `equalsPosition` alapján, hogy egyezik-e a pozíciójuk. Egyéb feltételek még, hogy az egyiknek mindenképpen irányíthatónak kell lennie, és egyiknek sem szabad célba juttatott objektumnak lennie. Ha ezek teljesülnek, akkor ütközés történt.

Összefoglalás

A játékfejlesztésről

A számítógépes játékfejlesztés egy olyan ága az iparnak, amelyben a programozókra nagyon sok feladat hárul. Mégis azt lehet mondani, hogy miközben kemény munka folyik, lehetőség nyílik szórakozásra az új fejlesztések kipróbálásával. A szakdolgozat témaköre kiválasztása után eldöntöttem, hogy egy Java játékot fogok fejleszteni. A „Szoftverfejlesztés Java környezetben” magában foglalja a játékszoftver fejlesztését is, így mivel munkahelyemen szintén ebben a környezetben mozgok, és szeretem is csinálni, ezért esett erre a választásom.

A Fly Control fejlesztése

A Fly Control nevű játék PC-re való megírása egy elég összetett folyamatot igényelt. A tervezési szakaszban sok olyan információ derült ki, amely a játék összképét előre irányba terelte. Így került bele például a lokalizáció is. A tervezés és megvalósítás során a legnagyobb fejtörést az okozta, miként lehetne az alkalmazást minél jobban felhasználó baráttá tenni. Az a funkció, hogy egy felhasználó akár saját pályát tudjon magának generálni, egy későbbi, már a fejlesztési idő alatti ötlet alapján került megvalósításra, így ehhez az implementáció elég nagy részét át kellett dolgozni. De összességében nagyon jó ötletnek bizonyult. Az egyetlen teendő amit szerettem volna még beletenni a játékba a felbontás váltás, melyet a beállítások menüpontban érhettük volna el. Ennek alapjai, már meg is találhatóak a programkódban, de a játék stabilitását rombolta volna, ha a végleges verzióba került volna, így idő hiányában ez sajnos kimaradt. A tervek alapján lehetett volna választani 800 x 600-as, 1024 x 768-as, ez egyébként az alap felbontás, és egyszerű módon bővítve, akár extrémebb felbontások közül is.

Sikerült azonban azt az elképzelést megvalósítani, hogy a későbbiek során, ha Java nyelven szeretnék újabb játékot fejleszteni, akkor van egy nagyon jó alap, amivel könnyen el tudom végezni az alábbi feladatokat, a kód újraírása nélkül:

- ablakváltások kezelése

- felugró ablakok létrehozása animációval
- lokalizálás
- naplózás
- XML fájlok beolvasása
- szövegeket titkosítani

Összességében tehát a tervezett funkciókat sikerült megvalósítani, és azok, akikkel teszteltettem a játékot, élvezettel játszottak vele.

Irodalomjegyzék

Könyvek

[1] Bátfai Norbert: Programozó Páternoszter, Debrecen, 2005

Internetes források

[2] Integrált fejlesztői környezet

http://hu.wikipedia.org/wiki/Integr%C3%A1lt_fejleszt%C5%91i_k%C3%B6rnyezet

[3] DefaultHandler osztály

http://java.sun.com/j2se/1.4.2/docs/api/org/xml/sax/helpers/DefaultHandler.html

[4] Koordináta geometria

http://hu.wikipedia.org/wiki/Koordin%C3%A1tageometria

[5] Caesar titkosítás

http://www.inf.unideb.hu/~bujdoso/kurzusok/halogyak/kisea/Titkositas_modszerei_dia.ppt

Függelék

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek, Espák Miklósnak szakdolgozatom elkészítésében nyújtott segítségéért és útmutató tanácsaiért.

Köszönettel tartozom szüleimnek, mivel lehetővé tették számomra azt, hogy egyetemre járhassak, és mindenben mellettem álltak, amikor szükségem volt rá.

Köszönettel tartozom munkatársaimnak a szakdolgozatomhoz való további ötleteikért, és tanácsaikért.

Hálás vagyok az összes barátomnak, akik mind szellemileg, mind lelkileg támogattak, és mellettem voltak az egyetemi éveim alatt.

Végül, de nem utolsó sorban köszönöm mindazoknak, akik bármilyen, akár legapróbb módon is segítették tanulmányaimat, és szakdolgozatom létrejöttét.