

Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozáselmélet és
Szoftvertechnológiai Tanszék

Bombás játék

dr. Gregorics Tibor
egyetemi docens

Börcsök Máté
programtervező informatikus BSc

Budapest, 2014

Tartalomjegyzék

I. Bevezetés	1
1. Témaválasztás	2
2. A játék rövid ismertetése	3
II. Felhasználói dokumentáció	5
3. A program használata	6
3.1. Menü	6
3.2. Csapatválasztás	7
3.3. Játékosok	8
3.4. Szörnyek	8
3.5. Klasszikus mód	9
3.6. Csapat mód	10
3.7. Vadászat	11
3.8. Történet mód	12
3.8.1. Első pálya	12
3.8.2. Második pálya	13
3.8.3. Harmadik pálya	14
3.8.4. Negyedik pálya	15
3.9. Eredménytábla	16
3.10. Szerver létrehozása	17
3.11. Csatlakozás	17
3.12. Irányítás	18
4. A program ismertetése	19
4.1. A pálya elemei	19
4.2. Bónuszok	19
4.3. Függőségek	21
4.4. Fejlesztőkörnyezetek telepítése	21

5. A megoldott problémák rövid ismertetése	23
III. Fejlesztői dokumentáció	24
6. Követelményspecifikáció	25
6.1. Keretrendszer	25
6.2. Menü	27
6.3. Csapatválasztó	27
6.4. Játéklogika	27
6.5. Játéktábla	28
6.6. Hálózat	30
7. Tervezés	31
7.1. A játéklogika	31
7.1.1. Menü és csapatválasztás	32
7.1.2. Játékmódok kezelése	32
7.1.3. InputOutput modul	33
7.2. Játéktábla	34
7.2.1. Terep	34
7.2.2. Játékos	35
7.2.3. Szörnyek	35
7.2.4. Bónuszok	36
7.2.5. Bombák, lángnyelvek	36
7.2.6. A játéktábla	37
7.3. Hálózat	37
7.3.1. Hálózat kezelése	37
7.3.2. Hálózati terv, első verzió	38
7.3.3. Hálózati terv, második verzió	39
7.3.4. A hálózati protokoll	40
7.3.5. A hálózat és a játéktábla	41
7.4. Megjelenítés, bemenetek	41
7.4.1. A főprogram és az időzítés	42
7.4.2. Megjelenítés	42
8. Megvalósítás	45
8.1. Implementációs döntések	45
8.1.1. Matrix osztály és a pálya	45
8.1.2. Szövegek kezelése	46
8.2. Felhasznált szoftverek	47

8.3.	Az implementációhoz használt technológiák	47
8.3.1.	OpenGL	47
8.3.2.	SDL	48
8.3.3.	Képek betöltése	49
8.3.4.	Bemenetek kezelése	50
8.4.	Ütközésvizsgálat	50
8.5.	Tanulságos hibák	51
8.5.1.	Időzítés	51
8.5.2.	Képernyőkép készítés	51
8.5.3.	Hálózati rész és a játéktábla integrálása	52
8.5.4.	Kerekítési hiba	52
9.	Tesztelés	53
9.1.	Végfelhasználói tesztelés	55
9.2.	Követelményspecifikáció szerinti tesztelés	56
9.3.	Ismert hibák	56
9.3.1.	Kapcsolat megszakad	56
9.3.2.	Számítógép lefagyása	56
10.	Bővítési lehetőségek, a jövő	57
10.1.	Fejlesztési ötletek	57
10.1.1.	Mesterséges intelligencia	57
10.1.2.	Igazi 3D megjelenítés	57
10.1.3.	Hangok	58
10.1.4.	Játékos közösség kiépítése	58
10.1.5.	Újabb játékmódok	58
10.1.6.	Többnyelvűség	58
10.1.7.	Pontszámítás	58
11.	Nyílt forráskód	59
Irodalomjegyzék		60

I. rész

Bevezetés

1. fejezet

Témaválasztás

A dolgozat fő témája a platformfüggetlen technológiák ismertetése a *C++* programozási nyelv használatával. Szerettem volna teljesen az alapoktól kiindulva egy játékot implementálni, ami látványos és komoly technikai háttérrel is rendelkezik.

A céлом az volt, hogy a kész termék egy szórakoztató játék legyen, amivel szívesen játszanak az emberek. Mióta programozással foglalkozom, mindig is fejlesztgettem kisebb játékokat (akár beadandó formájában), de egyik sem lett olyan minőségű vagy méretű, amit másoknak is megmutatnék, esetleg többször elővenném, mert jó vele játszani. Ezen szerettem volna változtatni. A szakdolgozat adta elvárások megkövetelik a nagy méretű feladatot és minőségi kivitelezést is. Egy kihívásként tekintettem az egészre, amiből sokat tanulhatok. Szerettem volna új technológiákat megismerni, amiket nem tanítottak az egyetemen. Ez sok kutatómunkával jár.

Személy szerint én jobban kedvelem az olyan játékokat, amik közösségek, egy számítógép előtt többen játszhatják egyszerre. Sajnos erre egyre kevesebb példát láttni a mai játékokban.

Volt egy játék (a következő fejezetben részletezem), amivel régen sokat játszottam, de egy idő után monotonná vált a játékmene. Sok újítás jutott eszembe, amivel fel lehetne dobni, meg lehetne változtatni az unalmas részeket. Ezek az ötletek addig érlelődtek, amíg úgy döntöttem, ezt szeretném megvalósítani szakdolgozatként. Ez egy kihívást is jelent, hiszen van már egy kész termék, annál mindenki jobbat, szórakoztatóból szerettem volna készíteni. Ez nem egyszerű feladat, hiszen az elődjét annak idején valószínűleg több erőforrással fejlesztették, de a modern eszközök segítségével nem éreztem lehetetlennek.

2. fejezet

A játék rövid ismertetése

Ebben a játékban több játékos verseng egy véletlenszerűen generált pályán, amin a cél az ellenfelek felrobbantása. Az alapötlet egy 1991-ben készített játékból származik, a Dyna Blaster-ból, más néven Bomberman. Igazi közösségi játék, többen is játszhatják egy képernyő előtt.

A játékban egy négyzetekre felosztott pályán járkálhatunk, közben bombákat rakhattunk le. A cél az ellenfelek felrobbantása. Ebben a pályán lévő szörnyek és falak próbálják megakadályozni a játékost. A bombák négy irányba robbannak: fel, le, jobbra, balra. Kezdéskor a játékosnak kettő nagyságú tüze és egy bombája van. Ezeket növelni lehet a pályán felszedhető bónuszokkal, amiket falak felrobbantásával találhatunk. A falak mögött továbbá „vírusok” is vannak, amik különféle furcsaságokkal színesítik a játékot és a játékosra vannak kihatással. Például felgyorsul, lelassul, folyamatosan bombát rak. Tehát lehet jó és rossz is ennek a hatása, de ezt előre nem lehet tudni.

A bombák miután leraktuk, 2 másodperc után robbannak. Azonban ha egy lángnyelv egy másik bombától eléri, akkor azonnal robbannak. Emiatt láncreakciók alakulhatnak ki és ezt kihasználva könnyen meg lehet lejni az ellenfeleket. A játékosok és a szörnyek nem tudnak átmenni a bombákon, a szörnyek ilyenkor visszafordulnak, ha tudnak.

A játékmenet nagyon gyors, a győzelemhez nagy koncentráció szükséges. Attól, hogy ilyen kevés eszközünk vannak a játékban - lényegében csak bombákat tudunk lerakni és szaladni - rengeteg stratégiát alkalmazhatunk. Be lehet zárni a másikat, szövetségek alakulhatnak ki, ki lehet használni egy már lerakott bombát és annak segítségével váratlanul hamarabb aktiválni egy másikat. A játék elején fontos a gyorsaság, minél több bónuszt tudunk felvenni, annál jobbak az esélyeink.

Az irányításához billentyűzeten kívül a program felismer kontrollereket is. (joystick, gamepad, stb.)

Több mód is van a játékban, amit a következő fejezetben fogok részletezni. Az eredeti játékban csak egyfélé mód létezik, ami ebben is megtalálható klasszikus

mód néven. Egy idő után ez monotonná válhat. Például ha egy játékos a kör elején kiesik, megvárhatja amíg a többi játékos végez a körrrel. Ez akár öt percig is tarthat, ami nagyon sok idő. Továbbá, ha minden falat felrobbantottak már és csak ketten maradtak, szinte döntetlen helyzet áll elő azonos képességű játékosokkal. Ezen úgy próbáltam meg változtatni, hogy játék közben is épülnek falak, a játékosok pedig egy idő után újraélednek. Így megszűnt a várakozási idő és pörgősebb lett a játék.



2.1. ábra. Az eredeti játék



2.2. ábra. A Bombás játék

II. rész

Felhasználói dokumentáció

3. fejezet

A program használata

3.1. Menü

A program elindítása után először a menüvel találkozunk, ahol ki kell választani a játékmódot. A menüben egérrel nem navigálhatunk, csak a billentyűzettel vagy a kontrollerekkel. A fel/le nyilakkal lehet lépkedni, a tűz gombbal pedig kiválasztani egy menüpontot. (bővebben az irányítás részben van szó a bemenetekről)



3.1. ábra. A menü

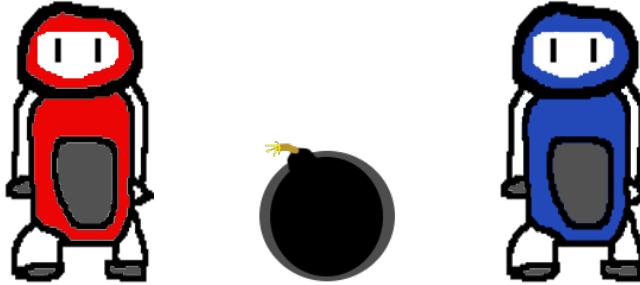
3.2. Csapatválasztás

Miután kiválasztottunk egy játékmódot a menüben, a csapatválasztás jelenik meg. A játékosok itt választhatják ki, melyik kontrollert használják és melyik csapatban szeretnének lenni. Ezt az adott kontrolleren a balra, illetve jobbra nyíllal lehetik meg, amivel a piros és kék csapat közül választhatnak. Amennyiben mégsem szeretnének játszani, a lefele gombbal lehet a választást visszavonni.



3.2. ábra. A csapatválasztás

3.3. Játékosok



3.3. ábra. A játékosok álló helyzetben, és köztük egy bomba.

A játékosok lényegében az úton tudnak menni és bombákat lerakni. A bombák felrobbantják a falakat, út van mögöttük. A betonon nem tudnak átmenni.

Meghatározott kezdőhellyel rendelkeznek, minden esetben ugyan ott fognak újraéledni. Éledés után két másodpercig sérthetetlenek, se szörnyek, se bombák nem árthatnak nekik. Erre azért van szükség, hogy a játékos ne halhasson meg azonnal a kezdésnél, például ha egy szörny éppen arra ment. Amíg a sérthetetlenség tart, a játékos sárgás színű.

Egy vírus hatása tíz másodpercig tart, ilyenkor a játékos zöldes színűvé változik. (bővebben a 4.2 fejezetben)

3.4. Szörnyek



3.4. ábra. A két szörnytípus

A szörnyek véletlenszerűen mozognak a pályán. Ha játékossal találkoznak, a játékos meghal.

A barna csak úton tud mozogni, a kék viszont képes átmenni a falakon. Ez sok bonyodalmat okozhat a játékosoknak.

A sebességük megegyezik a játékos „lassú” sebességével, tehát nem mozognak gyorsan, de kiszámíthatatlanok.

3.5. Klasszikus mód

A klasszikus mód körökre osztott, és csak egy játékos maradhat életben. A győztes pontot kap. Döntetlen esetén, ha mindenki meghal, senki nem kap pontot. Új kör kezdődik egészen addig, míg valaki el nem ér öt győzelmet.

A játékosok nem élednek újra, amennyiben valaki hamarabb kiesik, meg kell várnia a játék végét.

A pályán négy szörny van, mind a két típusból kettő-kettő. Nem élednek újra a körök folyamán.



3.5. ábra. A klasszikus mód

3.6. Csapat mód

Csapat módban a játékosok csapatokat alkotnak, és folyamatosan megy a játék. Ha egy játékest felrobbantanak, az újraéled a csapat kezdőhelyén, és folytatja a játékot, az ellenél pedig pontot kap.

Mivel elég gyorsan elfogyna a fal, a bónuszok és a szörnyek így ezek folyamatosan, véletlenszerű helyeken termelődnek újra. Természetesen nem azonnal, mielőtt valami változás történne a pályán animáció jelzi azt, így a játékosok felkészülhetnek rá. Enélkül monoton lenne a játék, csak kergetőznének a játékosok a pályán. Ez a mód dinamikusabb, mert nem kell megvárni a kör elejét a vesztes játékosnak. Klasszikus módban ez több perc is lehet.

Amelyik csapat hamarabb összegyűjt öt pontot, az nyeri a kört. Összesen öt győzelemre van szükség a játék végéhez.



3.6. ábra. A csapat mód

3.7. Vadászat

Vadászat módban a játékosoknak nem egymással, hanem a szörnyek ellen közösen kell harcolniuk. A szörnyek folyamatosan jelentkeznek meg a játék folyamán, eközben falak épülnek a pályán nehezítésként. A játék figyelembe veszi, hogy mennyi és milyen képességű játékos van, próbál annyi szörnyet generálni a pályára, hogy minden pont elég legyen, de ne is legyen túl sok.



3.7. ábra. A vadászat mód, épülő falakkal és éppen éledő szörnnyel (jobb alsó kék szörny alatt)

3.8. Történet mód

A történet módban előre elkészített pályákon kell végighaladni. Ez a mód öt pályából áll, amelyek folyamatosan egyre nehezednek.

3.8.1. Első pálya

Ezt a pályát ismertetőnek szántam, sok falon kell átjutni és két szörnyet kell megölni, de ezen kívül nem veselkedik nagy veszély a játékosra, saját magán kívül. A falak mögött sok bónusz található, meg lehet ismerkedni a tulajdonságaikkal. A jobb felső sarokban pedig minden két szörnytípusból látható egy. Látni lehet, hogy a kék színű átmegy a falakon, míg a barna nem.



3.8. ábra. Az első pálya

3.8.2. Második pálya

A második pályán már sok szörny van, minél gyorsabban át kell jutni rajtuk. Nem található fal a pályán, így bónuszokat sem tudunk szerezni. Ez megnehezíti a dolgunkat, mivel csak két kocka hosszúságú lángnyelv és egy bomba áll rendelkezésünkre.

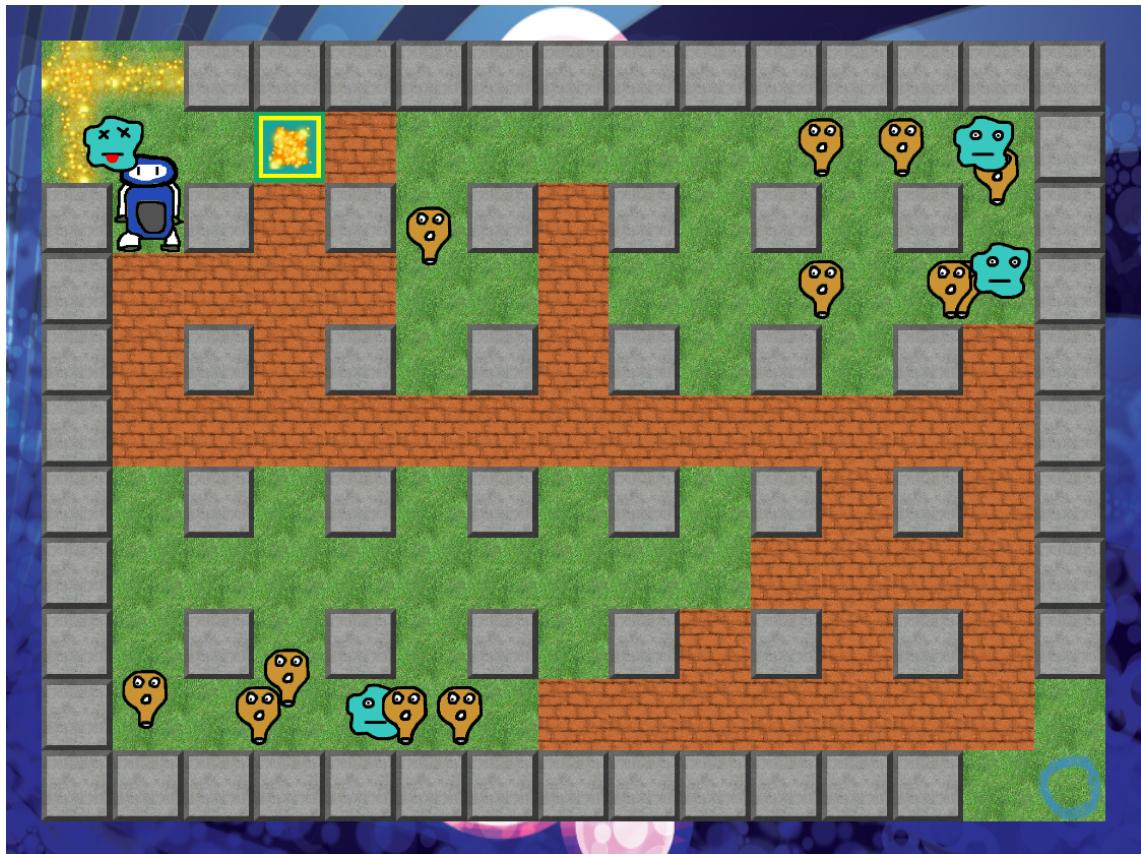
A legjobb stratégia csak végigrohanni a pályán a szörnyek között, akik a cél közelében könnyen fel tudnak gyűlni.



3.9. ábra. A második pálya

3.8.3. Harmadik pálya

Ezen a pályán még több szörny van és megjelennek a falon átmenő kékek is. Miközben ki akarunk törni a kezdőhelyről, gyakran átjönnek megzavarva minket a bónuszok szerzésében. De utána sem vagyunk biztonságban, meg kell küzdeni a barna szörnyekkel is.



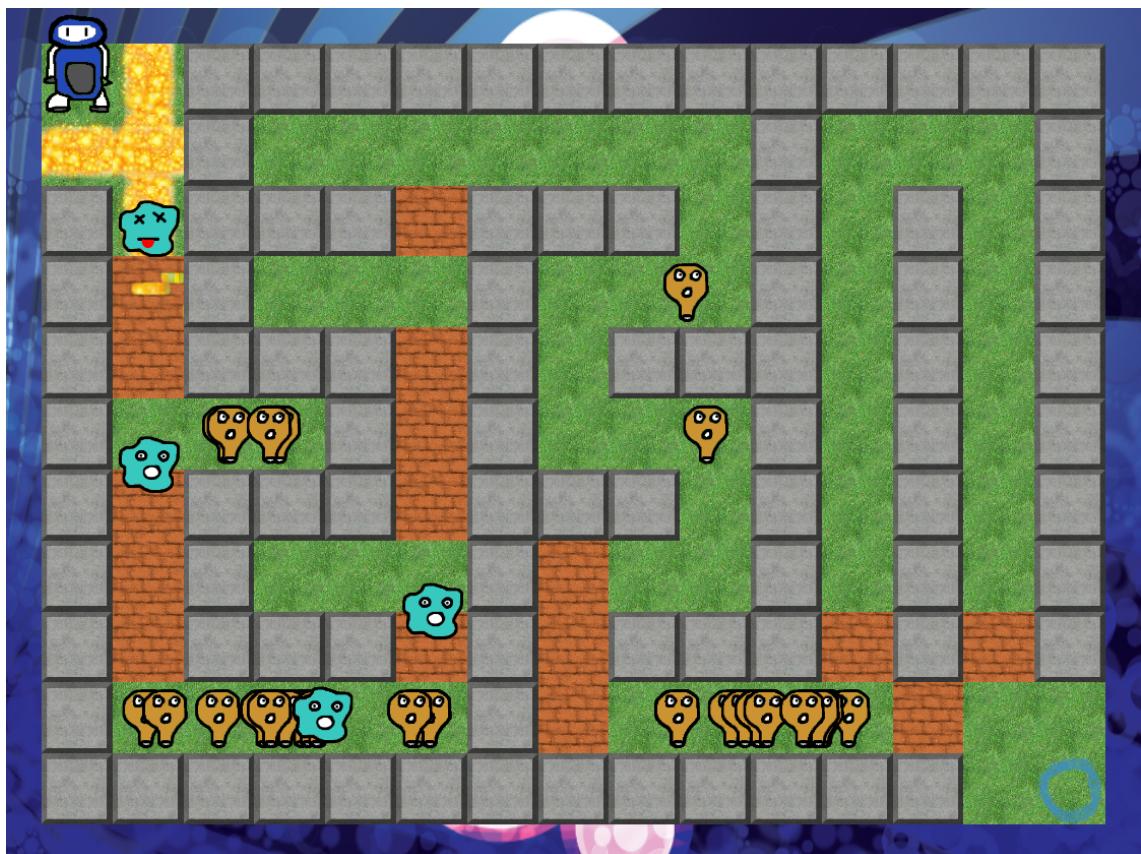
3.10. ábra. A harmadik pálya

3.8.4. Negyedik pálya

A legnehezebb pálya. Nagyon szűk a mozgástér és rengeteg a szörny. Elég nehéz végigmenni rajta, szerencsére is szükség van hozzá.

A pályán található sarkok az elrejtőzést szolgálják a saját bombák elől. Figyelni kell a kék szörnyeket is, a bombáink lerakását az ő mozgásukat figyelve kell időzíteni.

Vírusokat felvenni erősen nem ajánlott, mivel egy „Beragadt!” vírus szinte biztosan eldönti a játék sorsát. (a vírusokról a 4.2 fejezetben bővebben is van szó)



3.11. ábra. A negyedik pálya

Az ötödik pálya már csak levezetés, miután a játékos beért a célba, vége a játéknak.

Külön pontszámítás nincs, de részletes statisztika látható a történtekről.

3.9. Eredménytábla

Az eredménytábla minden kör végén megjelenik, részletes információkkal szolgálva a játékosoknak. Tartalmazza a győztes körök számát, a megölt játékosok számát, hányszor halt meg, és hány szörnyet ölt meg.

Az utolsó három érték történet és klasszikus módban nem nullázódik, viszont a csapat és vadászat módban igen, mivel ott ezet a számok döntik el a győztes állást.

A körök között a tábla hét másodperc után magától eltűnik. Ezt helyi játék esetén felgyorsíthatjuk, ha valamelyik játékos megnyomja a tűz gombját. Hálózati játéknál ezt nem engedtem meg, mivel van aki kevesebb és van aki több ideig szeretné nézni.

	Győzelem	Megölt	Meghalt	Szörny
	0	0	1	0
	1	0	0	0
	0	0	1	0
	0	0	1	0
	0	0	1	0

Nyomd meg a tűz gombot a következő körhöz

3.12. ábra. Az eredménytábla

3.10. Szerver létrehozása

Szerver létrehozásánál választanunk kell egy játékmódot, majd várakozunk a távoli játékosokra. A történet mód nem választható ilyenkor.

Miután mindenki csatlakozott és csapatot választott, a tűz gombbal indíthatjuk a játékot és minden ugyan úgy történik, mintha csak egy gép előtt játszanának. Kinézetre is szinte ugyan úgy néz ki.

3.11. Csatlakozás

Ha egy szerverhez akarunk csatlakozni, ismernünk kell az IP címét. Miután azt megadtuk, az enter megnyomásával csatlakozhatunk. Ha sikeres volt, láthatjuk milyen játékmódban van a szerver és választhatunk csapatot. A játékot a szerver indítja el.

A szerver IP címét Windowson a következőképpen kaphatjuk meg leggyorsabban:

1. Windows gomb + R megnyomásával előjön a futtatás ablak.
2. Írjuk be, hogy „cmd”, majd nyomjuk meg az entert.
3. A konzolablakba írjuk be, hogy „ipconfig”, majd enter.
4. Keressük meg a 192.168.*.* alakú címet, ez a helyi címünk.

Ha interneten és nem helyi hálózaton szeretnénk játszani, a címet könnyen megkaphatjuk, ha rákeresünk a „what is my ip” kifejezésre.

Ha a számítógépünk router mögött van, valószínűleg szükséges lesz a port átirányítás beállítása, ha szervert szeretnénk indítani.

A port: 1324 (TCP)

3.12. Irányítás

Az irányítás történhet billentyűzetről, gamepadről, joystickról. Miután kiválasztottuk a módot, a következő menüponton választhatunk a bemenetek közül.

- Billentyűzenen a más játékokban is megszokott nyilakkal, valamint a WASD billentyűkkel irányíthatunk két játékost. Bombát a jobb oldali CTRL és a TAB billentyűvel rakhatnak.
- Gamepad vagy Joystick segítségével még négy játékos játszhat egyszerre egy számítógépnél. Kontrollerek esetén nem állítottam külön feltételeket a gombokra. A „fő” irányítója felel a mozgásért, bombát pedig bármelyik gomb megnyomásával rakhatunk. Rengetegféle kontroller van a piacon, és mindegyiknek teljesen más a beállítása, a gombjainak az elhelyezkedése. Kezelem az analóg kontrollereket is. Az eredeti játék az analóg karokat is úgy vette, mintha gombok lennének. Mivel ezeket nem erre terveztek, irányíthatatlan lett a játékos. Az én verziómban viszont nagyon jól, pontosan használható.

Az irányítás elég trükkös, ha az ember egy fal mellett megy és közben a fal felé is próbál menni, akkor lelassul egy kicsit. Ez a lehetőség véletlen került a programba, de annyira megtetszett, hogy benne hagytam. Így ha elég ügyes a játékos, gyorsabban tud menni, mint a többiek.

4. fejezet

A program ismertetése

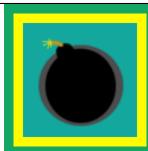
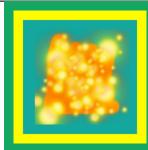
4.1. A pálya elemei

A pálya négyzet alakú építőelemekből áll. Lehet fal, beton, út vagy győzelmi mező. Ez utóbbi csak történet módban található meg. A pályaelemeken való átjárhatóságot a következő táblázattal mutatom be:

Kép	Neve	Átjárható	Felrobbantható
	Téglafal	Nem	Igen
	Út	Igen	Nem
	Beton	Nem	Nem

4.2. Bónuszok

A falak mögött különböző bónuszok találhatóak véletlenszerűen. Ezek felsorolva:

Kép	Neve	Hatása
	Bomba	Plusz egy bombát rakhat le a játékos
	Tűz	Meghosszabbítja a játékos bombáinak lángnyelvét minden irányban egy négyzetnyivel
	Vírus	<p>A vírus 10 másodpercig hat a játékosra, az alábbiak közül véletlenszerűen aktiválódik valamelyik. A játékos ilyenkor zöldes színűre változik. A vírusok felsorolva:</p> <ul style="list-style-type: none"> • A játékos felgyorsul nagyjából kétszeres sebességre • A játékos lelassul, a szörnyek sebességére • A tűz gomb beragad és a játékos folyamatosan rakja a bombákat • A játékos nem tud bombát raktani • Végtelen számú bombát tud raktani • Végtelen hosszúságú tűz • Azon a helyen pillanatokon belül egy szörny fog megjelenni (animáció kíséretében)

4.3. Függőségek

A program „Cross-platform”, tehát minden olyan számítógépen, amire van C++ fordító és linker, el lehet indítani. Viszont vannak szoftveres függőségei:

- Legalább OpenGL 1.4 verzió. Ezt használom a kirajzoláshoz. A függvénykönyvtárat 3D megjelenítésre terveztek, én mégis ezt használtam, mivel sokkal gyorsabb, mint a szoftveres kirajzolás. A mai modern számítógépekben van 3D gyorsítás, ami a processzor helyett a videokártyát terheli. Ez a különbség még ilyen kevés rajzolást igénylő programknál is meglátszik a teljesítményben.
- SDL 1.2 verzió. Ezt használom a bemenetek, ablakok kezelésére, idő lekérdezésére, hangok lejátszására. A könyvtár LGPL [5] licenc alatt áll, így nem lehet statikusan linkelni korlátok nélkül.

A megfelelő működéshez úgy tapasztaltam, szüksége van egy viszonylag modern videokártyára is. Hogy az OpenGL megfelelően működjön, a videokártyának szüksége van az operációs rendszerhez tartozó driver (meghajtó) program működésére.

Az SDL függvénykönyvtárat mellékeltem a Windowsos verzióban. A manapság elérhető Linux disztribúciókban alapból megtalálható, ezért ide nem mellékeltem. Amennyiben ez még sincs így, külön kell telepíteni az operációs rendszerre.

4.4. Fejlesztőkörnyezetek telepítése

Először a Linuxos környezet telepítését fogom leírni, mivel az az egyszerűbb. Ubuntu Linux 12.04 alatt fejlesztettem, ami az aptitude nevű csomagkezelővel rendelkezik. Így nem kell lefordítani a forrásfájlokat, elég a megfelelő csomagokat telepíteni. Ezt a legegyszerűbben a következő parancssal tehetjük meg, amit a terminálba írunk:

```
sudo apt-get install libsdl-devel libsdl-net-devel
```

Ezek a fejlesztői (developer) csomagok, automatikusan települnek a futáshoz szükséges (runtime) csomagok is. Ezek birtokában kiadhatjuk a „make” parancsot a program főkönyvtárában, ami lefordítja a játékot.

Windows alatt több fejlesztőkörnyezetet is választhatunk. Én a Cygwin-t választottam, ami egy Unix-szerű terminált és környezetet biztosít. Ez alá telepítettem a megfelelő programokat.

Először is a <http://cygwin.com/install.html> oldalról letöltöttem a setup.exe-t, ami az internetről leszedi a szükséges csomagokat. A telepítés folyamán felajánlja bizonyos csomagok telepítését, ahol ki lehet választani mi szükséges nekünk. Itt ki

kellett választani a gcc, gcc-c++, make csomagokat. Hogy az SDL megfelelően működjön, a következő linket is meg kell látogatni: <http://www.libsdl.org/extras/win32/cygwin/>, innen le kell szedni az opengl-devel tömörített fájlt. Ennek tartalmát a /usr/include/w32api/ mappába kell kitömöríteni (gl.h, glu.h).

```
./configure && make && make install
```

5. fejezet

A megoldott problémák rövid ismertetése

A játék nem egy klasszikus programozási feladat, de elég összetett. Objektum-orientált programozást használtam, így könnyen részekre lehet bontani, és átláthatóbb is. A főbb részek felsorolva, röviden a megoldáshoz felhasznált módszereket ismertetve:

- Megjelenítés. A megjelenítést alacsony szinten implementáltam, csak az OpenGL könyvtárat és egy png fájlokat betöltő osztályt használtam hozzá. Ez még nem elég egy játék megjelenítéséhez. Szükség volt szöveg kiírására, animáció lejátszására és arra, hogy egy képben több képet tároljak (a helyfoglalás miatt). Szintén kezelní kell az ablakot, amit a felhasználó át is méretezhet, ekkor a játékeret is át kell méretezni.
- Játékmódok kezelése, menü.
- Játéktábla kezelése. Ez talán a legbonyolultabb része a programnak. Nyilvántartja a táblán lévő összes objektumot. Ez a rész felel a pozíciók frissítéséért és az ütközésvizsgálatért is. Emiatt szükség van az idő pontos nyilvántartására is, figyelni kell, mennyi idő telt el a képkockák között, hogy megfelelően, egyenletes sebességgel mozogjanak a játékosok. Az ütközésvizsgálat pedig annyiból különleges, hogy a játékosok nagyon keskeny helyeken járkálnak. Így nem elég egy egyszerű vizsgálat, hogy ütközik-e vagy sem, mert akkor használhatatlan lenne a játék.
- Hálózat kezelése.

Ezeket a témaikat tovább részleteztem a következő fejezetben, a fejlesztői dokumentációban.

III. rész

Fejlesztői dokumentáció

6. fejezet

Követelményspecifikáció

A feladat nehezen specifikálható, ám nem lehet elhanyagolni, tudni kell mi a kitűzött cél. Nagyvonalúan annyi volt a követelmény, hogy hasonló funkcionálitást érjek el, mint a Dyna Blaster, kiegészítve az új ötleteimmel és hálózat kezeléssel.

A következő alfejezetekben funkcionális és nem funkcionális követelményeket fogok megfogalmazni a program különböző részeivel kapcsolatban.

A funkcionális követelményeket táblázatban írom le, ahol bizonyos eseteket specifikálok azok előfeltételével, tevékenységével és eredményével. A táblázat eset és előfeltétel mezőibe inkább kulcsszavakat írtam az átláthatóság miatt, mintsem részletes esetleírásokat.

A nem funkcionális követelményekben minőségi elvárásokat fogalmazok meg feltételek mellett, amik szintén szükségesek a működéshez.

A végfelhasználói tesztek során ezeket az eseteket használtam fel.

6.1. Keretrendszer

A keretrendszer minden további funkcionálitásnak az alapja, lényegében a megjelenítést és időzítést tartalmazza.

Odafigyeltem arra, hogy semmilyen funkcióban ne kötődjön a játékhoz, ezért teljesen független tőle.

Nem funkcionális követelmények:

- Platformfüggetlenség, működjön minél több rendszeren, amelyek teljesítik a minimális rendszerkövetelményeket. Minimális követelmény: Linux, Windows.

Eset	Előfeltétel	Tevékenység	Eredmény
Ablak létrehozása		Amikor a program elindul, hozzon létre egy ablakot és környezetet a grafika megjelenítéséhez.	Egy ablak megjelenik és a rajzolás is működik rajta.
Grafika betöltése		Tölts be a rendelkezésre álló képeket és animációkat.	A grafikus memóriában elvannak tárolva a képek, bevannak töltve az egyéb fájlok.
Megjelenítés	Grafika betöltve	A rendszer legyen képes kirajzolni az ablakra a betöltött képeket.	A program képes rajzolni az ablakra.
Hiányzó kép		Kezelje a hiányzó képeket, ne szálljon el amennyiben hiányzik egy kép.	Hibaüzenet a konzolon, a grafika nem jelenik meg.
Bemenetek	Az ablak létre van hozva	Kezelje a billentyűzetet és kontrollereket. (joystick, gamepad)	Elérhetők a bemenetek, külön-külön vagy egy közös interfészen keresztül is.
Időzítés	Az ablak létre van hozva	Az újrarajzolást végezze el a keretrendszer és legyen lehetőség lekérdezni az előző képkockához képest eltelt időt.	Újrarajzoláskor megjelennek a kirajzolt grafikák és megtudhatjuk az eltelt időt.
Többszörös képek	Megjelenítés	Legyen lehetőség egy képnak csak egy részét kirajzolni.	Felgyorsul a betöltés, könnyebben szerkeszthetők és kezelhetőek a képek.
Szövegek	Többszörös képek	A szövegek kirajzolását a többszörös képek alapján tudja kezelní, helyettesítse be a megfelelő képszeleteket.	Lehet írni az ablakra szöveget.
Animáció	Megjelenítés	Legyen lehetőség animációkat automatikusan kirajzolni anélkül, hogy minden képkockánál mi számoljuk ki, melyik kép következik.	Megjeleníthetünk egyszerűen animációkat.

6.2. Menü

A program elindítása után, miután a keretrendszer betöltötte a grafikákat, a menünek kell megjelenni.

Eset	Előfeltétel	Tevékenység	Eredmény
Menüpontok	Grafika betöltve	Jelenjenek meg a választható menüpontok.	A felhasználó látja, mi közül lehet választani.
Navigálás	Menüpontok	A fel/le nyilakkal és a tűz gombbal bármelyik bemenetről lépkedni lehessen a menüben.	A felhasználó tud választani.
Választás	Navigálás	Miután a felhasználó kiválasztotta a kívánt menüpontot, lépj tovább a program.	A menü eltűnik és továbbadja a vezérlést.

6.3. Csapatválasztó

A csapatválasztóban határozható meg a játékosok által, melyik vezérlővel szeretnének lenni és melyik csapatban.

Eset	Előfeltétel	Tevékenység	Eredmény
Navigálás	Menüpontok	A balra/jobbra nyilakkal Lehessen váltogatni a csapatok között.	A felhasználó tud választani.
Visszavonás	Menüpontok	A le gomb esetén legyen inaktív az adott játékos.	A felhasználó vissza tudja vonni a választását.
Választás	Navigálás	Amikor valaki megnyomja a tűz gombot, lépj tovább a program.	A csapatválasztó eltűnik és továbbadja a vezérlést.

6.4. Játéklogika

Ez ad keretet a játéknak, a játéktáblát figyeli, megállítja, illetve elindítja amikor szükséges. A játékmódokat külön-külön nem részleteztem, mivel sok esetet generálna, de nem szabad eltekinteni ettől. Majdnem minden mód másképp működik, más paramétereket kell figyelni a táblán, ez a legkritikusabb része a modulnak.

Eset	Előfeltétel	Tevékenység	Eredmény
Kezdet	Választás a csapatválasztóban	A játék állítsa be megfelelően a játékosok kezdőpozícióját, kezelje a pályát, hozza létre a kezdetben szükséges állapotokat.	Módtól függően betöltődik vagy generálódik egy pálya, létrejönnek szörnyek. A játékosok elhelyezkednek rajta a kezdőpozícióban és elindul a játék.
Figyelő	Kezdet	A játék figyeli a pályán történteket és megállítja, amennyiben győztes vagy döntetlen állás van.	A játék nem lesz végtelen, a kör vége kapja meg a vezérlést.
Kör vége	Figyelő	Egy kör végén meg kell jelezni az eredményeket a játékosoknak. Ez pár másodperc után eltűnik, de ha valaki megnyomja a tűz gombot, hamarabb indul a következő kör.	A játékosok minden kör végén tájékozódnak az aktuális állásról.
Pontszám figyelő	Kör vége	Amennyiben az egyik játékos elér egy bizonyos ponthatárt, ő lesz a győztes és visszalépünk a menübe.	Vége a játéknak, a vezérlést továbbadja a menünek.

6.5. Játéktábla

A játéktáblán történik a legtöbb esemény és a legbonyolultabb megjelenítés. Itt már minden adott, kik játszanak és milyen módban.

Nem funkcionális követelmény, hogy az ütközésvizsgálat „jó” legyen. Ezt nehéz lenne lespecifikálni. Amikor egy játékost irányítanak az az elvárt érzés, hogy könnyen irányítható és nem akad el semmiben. Viszont az is elvárás, hogy ne menjen olyan mezőre, ami tiltott.

Eset	Előfeltétel	Tevékenység	Eredmény
Kezdet	Kezdet a Játéklogika által kért változtatások végrehajtódnak.	A játéklogika által kért változtatások végrehajtódnak.	Indulhat az első képkocka.
Képkocka	Kezdet	Amíg nincs vége a körnek, addig újra és újra végrehajtódik ez az eset. Frissítjük a pozíciókat, majd kirajzoljuk az objektumokat.	A következő események történnek: <ol style="list-style-type: none"> Játékosok és szörnyek mozgatása Ütközésvizsgálat Bombák felrobbanása, amennyiben szükséges Hálózati események (itt nem részletezem) Kirajzolás
Játékos mozgatás	Képkocka és a játékos él	A játékosokat a bemenetekről vagy hálózatról vezérelve mozgatjuk.	Megváltozik a pozíciójuk.
Szörny mozgatás	Játékos mozgatás	A szörnyek egyenletesen mozognak, ha akadályba ütköznek, irányt váltanak, de néha ezt véletlenszerűen is megteszik.	Szörnyek mászkálnak a pályán.
Ütközés-vizsgálat I.	Mozgatás	Megnézzük, hogy a játékosok nekimentek-e egy szörnynek vagy lángnyelvnek.	Amennyiben igen, meghalnak.
Ütközés-vizsgálat II.	Ütközés-vizsgálat I.	Megnézzük, hogy a játékosok nekimentek-e valamilyen akadálynak	Amennyiben igen, visszaléptetjük.
Bomba-vizsgálat	Ütközés-vizsgálat II.	Megvizsgáljuk a bombák időzítőjét, amennyiben lejárt, felrobbantjuk.	A robbanás hatására lángnyelvek keletkeznek. Ezek láncreakció szerűen aktiválhatnak más bombákat is.
Kirajzolás	Bomba-vizsgálat	Kirajzoljuk az objektumokat a képernyőre.	Láthatóvá válik az aktuális állapot.

6.6. Hálózat

A hálózat kezelése.

Eset	Előfeltétel	Tevékenység	Eredmény
Szerver indul		A program elkezd figyelni egy kiválasztott portot és várja az érkező játékosokat.	Megjelennek az eddig csatlakozott játékosok.
Kliens indul	IP cím adott	A program megpróbál kapcsolódni az adott címre.	Ha sikerült, választani lehet csapatot.
Csapat-választás	Kapcsolat létrejött	A váltások szinkronban vannak a hálózaton.	Lásd: 6.3 fejezet
Játék indítása	Csapat-választás	A kliensek értesítést kapnak a szervertől, hogy a játék elindult.	A játék minden a hálózaton lévő példányon elindul.
Kliens küld	Játék indítása	A kliens elküldi a saját pozíóját, amennyiben az változott.	A szerver fogadja.
Szerver küld	Játék indítása	A szerver elküldi az összes frissítést minden kliensnek, ami történt, kivéve amit az aktuális klienstől kapott.	A kliensek fogadják.
Kliens fogad	Játék indítása		

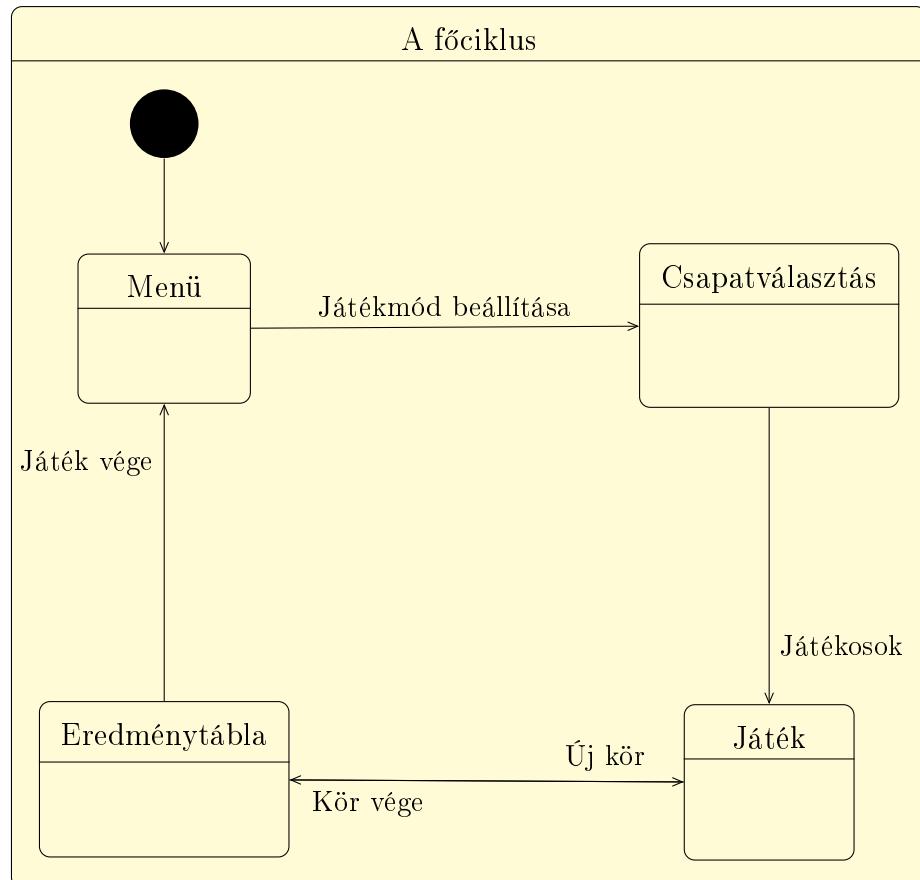
7. fejezet

Tervezés

7.1. A játéklogika

A tervezésnél, mivel elég bonyolult feladatról van szó, nagyobb részekre bontottam a programot. Először egy folyamatábrát készítettem, ami nagyvonalúan a program futását mutatja be.

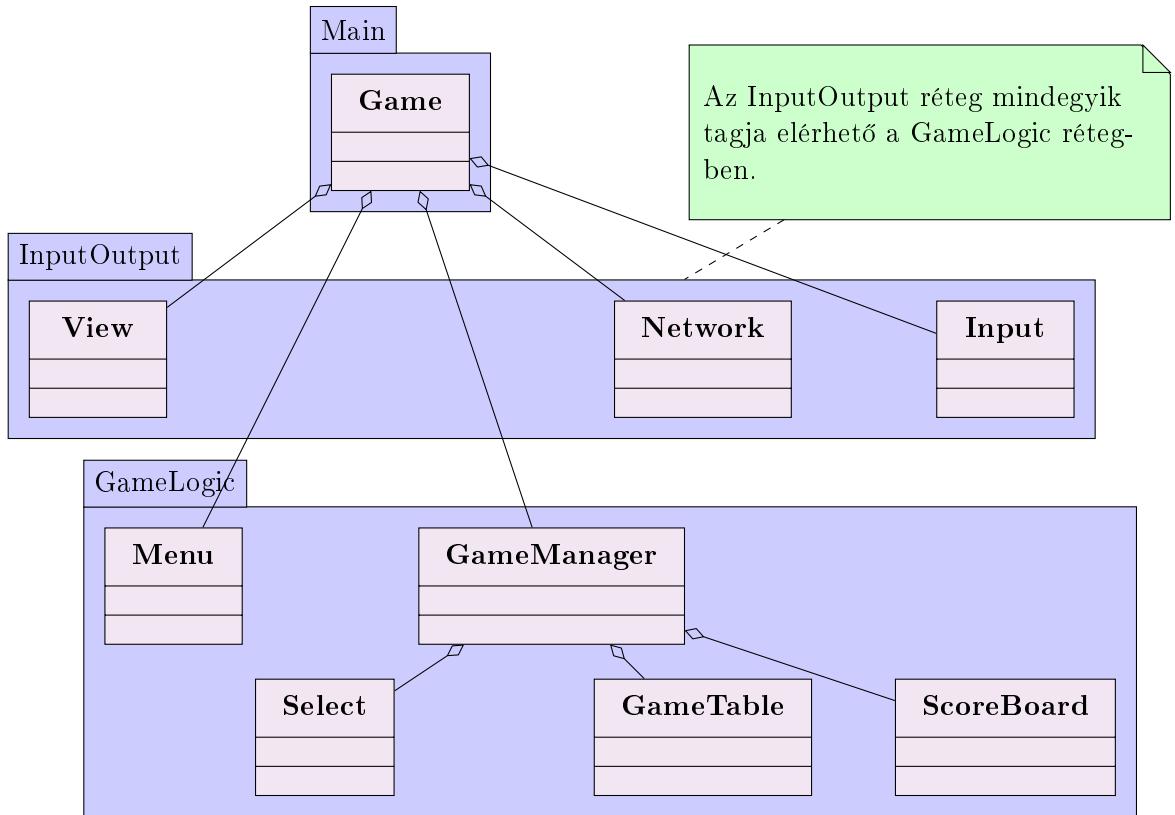
Az egész program főosztálya a Game osztály, amiben a főciklus is található.



7.1. ábra. A főciklus.

Ezeket a részeket a tervezésnél külön egységeként kezeltem és a következőkben fogom részletezni. Ezen kívül vannak más feladatok is, amik a háttérben történnek.

Magát a tervet nem egy lépésben készítettem el, hanem több iterációban. Nem részleteztem az iterációkat a dolgozatomban. Például a játékos mozgását először nem terveztem meg, hogy fogom pontosan megoldani. Miután egy „álló” ember képes volt a pályán mozogni, megterveztem, hogyan tudna abba az irányba nézni, amerre mennie kell és mozgatni a lábait.



7.2. ábra. A játék fontosabb osztályai

7.1.1. Menü és csapatválasztás

A játék indításakor a menü jelenik meg. Ez egy elég egyszerű osztály felépítésre, ezért nem is részletezem. Kimenete: Játékmód, játék helye

Ezután a csapatválasztás következik, ennek kimenete a játékosok beállításai. Ezután létrejöhet a GameTable.

7.1.2. Játékmódok kezelése

A játékmódok kezelését a GameManager osztály végzi. `/gamemanager.h/` Ez az osztály hívja meg a GameTable inicializáló utasításait, a menüben kiválasztott

utasításoknak megfelelően. Miután elindult a játék, minden képkocka után megnézi, történt-e valami a játék számára fontos dolog. Lényegében ez az osztály adja meg a keretet a programnak. Mindegyik játékmódhoz más paraméterű játéktérre van szükség, komoly logika van a háttérben.

Ahogy már említettem, szükséges bemenete a kiválasztott játékmód és a játékosok beállításai.

Ezután létrejön a GameTable, amin elindul a játék. Közben a GameManager figyeli a történeteket, amennyiben győztes kondíció van, megállítja a játékot, és megjeleníteni a ScoreBoard osztályt. Ha szükséges újabb kört indít, vagy visszatér a menübe, és minden kezdődhet előlről.

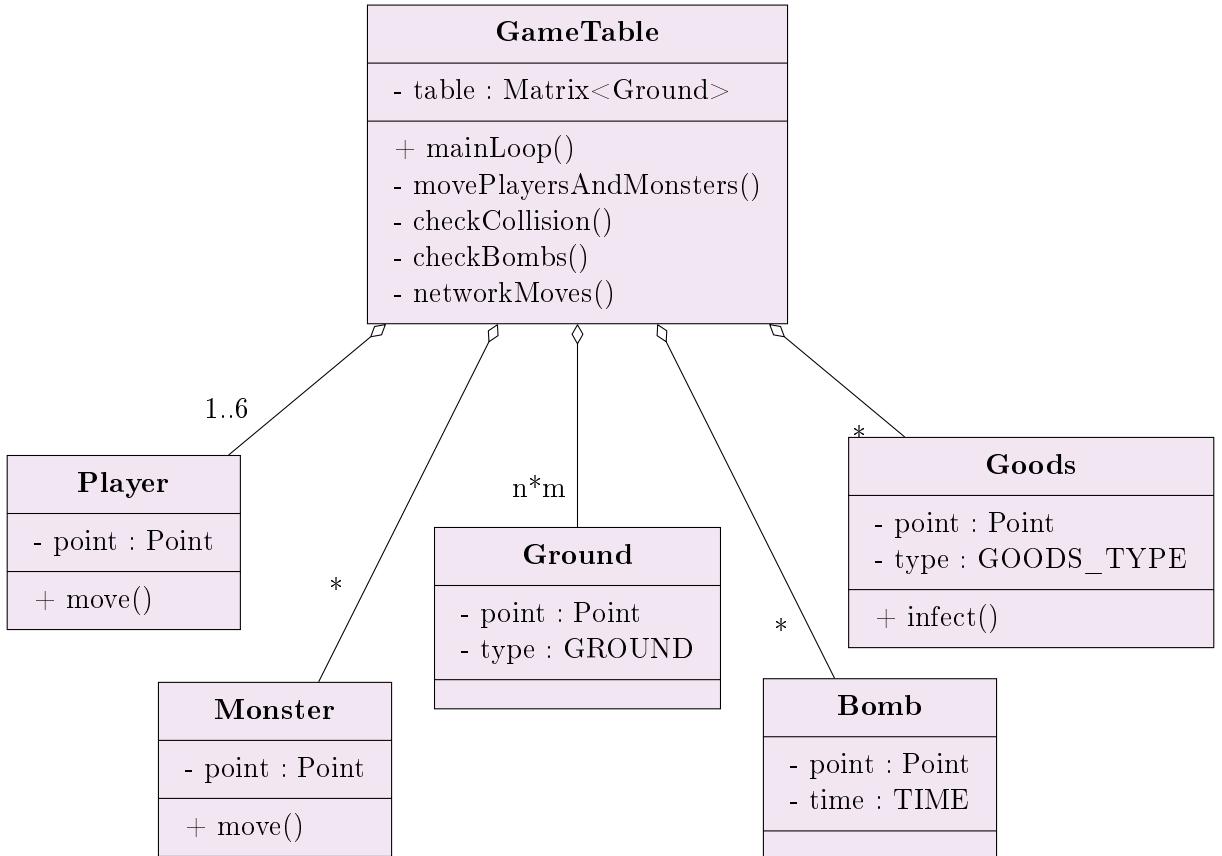
7.1.3. InputOutput modul

Így neveztem el a View, Input és Network osztályokat közösen. Ezek lényegében a program perifériái. Indításkor jönnek létre egy példányban, és a program futása végéig megmaradnak. Azok az osztályok, amelyeknek szüksége van rá, egy referenciát kapnak róla, és úgy használják.

7.2. Játéktábla

Bemenetként kapja a játékosok számát, a bemeneteket (amivel a játékosokat vezéreljük) hozzákapcsoljuk és elindulhat a játék. Az összes többi modulhoz kapcsolódik, használja a megjelenítő réteget, bemeneteket és a hálózatot is. A játéklogika réteg pedig létrehozza és vizsgálja, mi történik a táblán.

Rengeteg információt tartalmaz, csak néhány információt helyeztem az UML osztálydiagramjára. Tovább fogom részletezni a következő fejezetekben.



7.3. ábra. A főciklus.

7.2.1. Terep

Az alsó megjelenítendő réteg a terep. Ezen járhatnak a játékosok és erre rakhatják a bombákat.

A terep lényegében négy állapotú lehet, út, fal, beton, cél mező. Ezt tárolom egy $n \times m$ mátrixban. A játékosok ezen járkálhatnak, és módosíthatják bombák lerakásával.

7.2.2. Játékos

A játékosok a Player osztályban vannak definiálva. Sok információt kell róluk eltárolni. Felsorolva:

- Mi az azonosítója.
- Ki vezérli őket (Control osztályra mutató pointer).
- Hol vannak és hol voltak az előző képkockán, utóbbira az ütközésvizsgálat és az animáció miatt van szükség.
- Sebességük, bombák száma, tüzük nagysága.
- Élnek-e még.
- Győztes körök száma, hányszor robbant fel, hány szörnyet és játékost robbantott fel.
- Mikor fertőződött meg melyik vírussal, ez mikor fog lejárni, ekkor visszaállítani a tulajdonságokat.
- Melyik színű „bábuval” van, melyik animációs fázisban tart, mikor kell a következőbe lépnie, amennyiben megáll, visszaváltani az alapállapotra.
- Hálózati vezérléshez a megfelelő függvények

Kezelní kell a játékosok mozgását is. Van az osztálynak egy move függvénye, ami lekérdezi a bemenet kezeléséért felelős osztályból, ami hozzá van rendelve, hogy a játékos milyen parancsokat adott neki. Ezt a képkockák között eltelt idő függvényében teszi, tehát ugyanolyan sebességű lesz egy játékos, bármilyen számítógépen játsszák. Ezzel az osztállyal el tudjuk fedni, hogy a játékos milyen periférián keresztül van irányítva, billentyűzet, joystick vagy távoli gép.

Továbbá a setX és setY függvény sem egy egyszerű értékkadásból áll. Ennek is figyelembe kell venni, hogy a pozíció, amit az ütközésvizsgálattól kap, a szabályok és a játékos sebessége alapján mennyire helyes. Itt figyelembe kell venni az irányt, a két képkocka között eltelt időt, az előző és a jelenlegi képkockán lévő helyzetet és persze az új pontot, ahol lennie kéne a játékosnak. Majd megvizsgáljuk, hogy az új pont túllépi-e a kiszámított új pozíciót, eszerint állítjuk be az új pozíciót.

7.2.3. Szörnyek

A pályán a szörnyek véletlenszerűen mozognak. Egy szörnyről tudni kell:

- Milyen típusú, melyik animációt kell hozzárendelni.

- Hol van jelenleg, hol volt előzőleg (hogy tudjuk, mikor vált mezőt), melyik irányba tart éppen.
- Átmegy-e a falon
- Melyik animációs fázisban van, mikor kell váltania.
- Mivel véletlenszerűen mozognak, ezt is ebben az osztályban kell elvégezni, és a hálózati játék miatt minden játékosat értesíteni kell róla.

A típustól függően változhat a sebessége, vagy átmehet a falakon.

7.2.4. Bónuszok

A falak felrobbantásával bónuszok jelenhetnek meg a játéktáblán. Ezeknek csak a típusa és helye számít. Típus szerint lehet bomba növelő, tűz nagyság növelő, vagy vírus. Ezek alapján a játékoson meg tudja határozni, végre tudja hajtani a szükséges módosításokat.

7.2.5. Bombák, lángnyelvek

A játéktáblára a játékosok bombákat rakhatsanak le. Egy bombáról ismerni kell:

- Az elhelyezkedésének helyét.
- Ki rakta le, ez a pontszámítás miatt érdekes.
- Mikor kell felrobbannia.
- Mekkora a mérete.

Amikor a játékos lerak egy bombát, csökkenteni kell a bombáinak számát. Amikor felrobban, növeljük egyel. A robbanás a játéktáblán jelenik meg, és lángnyelveket generál. Erre azért van szükség, mert a robbanás nem csak egy pillanatig tart, pár tizedmásodpercig a robbanás után is van hatása. Ennek az osztálynak is tároljuk a helyét, hogy mikor jár le, és az animáció állapotát, ami abból különleges, hogy a lángnyelv vége másmilyen, lekerekített animációt kapott. A lángnyelveknek is tudnia kell, melyik játékos indította, különben nem tudnánk számolni a pontokat. Egy robbanás láncreakciót indíthat be, ha egy lángnyelv elér egy másik bombát, akkor az attól függetlenül, hogy mikor fog lejárni (aktiválódni), azonnal felrobban. Az eredeti tulajdonosa is felül lesz definiálva, az számít, kinek a bombája indította el a folyamatot. A bomba felrobbanásakor a bombExplode függvény hívódik meg. Ha a bomba azért robbant, mert lejárt, négy irányba, ha egy másik bomba aktiválta, három irányba (ahonnan beérkezett a lángnyelv, abba az irányba nem) hívja meg a

flame függvényt. Ez egy rekurzív függvény, melynek paraméterei: x, y, méret, honnan, ki. Irány szerint az x vagy y változót egyel növelte vagy csökkentve, a méretet egyel csökkentve, a honnan és ki paramétert megőrizve hívjuk meg újra a függvényt. Megvizsgálja, nem ütközött-e falba vagy betonba a láng. Ha falba ütközik, akkor azt felrobbantja, és megáll a rekurzió. Ha betonba, akkor is megáll, de azt nem robbantja fel. Ha bónuszba ütközik, azt is megsemmisíti, és megáll. A lángoló fal és bónusz nincs semmilyen hatással a játékosra.

7.2.6. A játéktábla

A játéktábla és elemeinek a kirajzolása már egyszerű, mivel minden objektumról tudjuk, hol helyezkedik el és milyen képet kell kirajzolni róla, animáció esetén is. Utóbbi a state függvény segítségével tudhatjuk meg. Mivel a View osztály jól kezeli a paramétereket, ennek a függvénynek csak egy string a visszatérési értéke, ahol a kép nevén kívül a transzformációkat és egyéb paramétereket is megadhatunk a # karakter után. A játékosok, szörnyek, bombák és lángnyelvek animációját kézzel, a saját osztályain belül kezelem, mivel ezek nagyon gyakran változnak. Az animáció kirajzolást különböző egyéb, a játékot közvetlenül nem érintő grafikára használom, például szörny születése, játékos vagy szörny halála, képernyő elsötétítése.

7.3. Hálózat

7.3.1. Hálózat kezelése

A hálózat kezeléséhez végül az `SDL_net` függvénykönyvtárat választottam. Eredetileg BSD socketeket akartam használni, de semmi sem garantálja, hogy minden rendszeren ugyanúgy működjön. Főleg a Little/Big-Endian processzorok közötti különbség okozott volna gondot. Ezen kívül így nem kellett törődnöm a töredék csomagokkal, amik a TCP protokollból adódóan érkezhetnének és megnehezítenék a feldolgozást.

Az hálózati módban nem írtam külön protokollt a játékosok megtalálásához, ezt túl bonyolultnak találtam. A csatlakozásnál a program egy IP címet kér, amit a kliensnek ismernie kell.

Ha a játékos csatlakozik egy szerverhez, először az előszobába (lobby) kerül, ahol a szerver tulajdonosa beállítja a játékmódokat, a játékos pedig a karakterét, esetleg csapatot választ.

Választanom kellett az UDP és a TCP protokollok között.

TCP esetén a csomagok a küldés sorrendjében fognak megérkezni, garantáltan megérkeznek, a kapcsolat megszakadása észlelhető és különféle hibafelismerő algoritmusokat alkalmaz minden csomagra.

UDP esetén az előzőek közül egyik sem igaz! Még kapcsolatot se kell felépíteni a távoli számítógéppel. Ám ezeknek a lehetőségeknek a hiánya előnyükre is lehet, ha belegondolunk.

7.3.2. Hálózati terv, első verzió

Ez a terv nem valósult meg, de a hálózati protokoll kialakulásának a része volt. Nem volt tapasztalatom még hálózati programozásban, főleg nem valós idejű, folyamatosan érkező adatok kezelésében.

Tegyük fel, hogy egy csomag valamilyen hálózati probléma miatt nagyon lassan érkezik meg. Ekkor a TCP protokoll szerint a csomagoknak sorrendben kell megérkezni, így megvárja ezt a csomagot, hiába érkezne meg utána a többi. A játék szempontjából nem jelent problémát, ha egy csomag kimerül, legrosszabb esetben is egy pár pixellel nagyobbat lép a karakter. Emiatt választottam az UDP protokollt. Az SDL_net lehetőséget ad arra, hogy az UDP módban is létrehozzon egy kapcsolatot (virtuálisan), de lehetőség van a klasszikus kapcsolat nélküli adattovábbításra is. Ezzel számomra megkönnyítette az átláthatóságot. Mivel UDP módban nincs biztosítva a csomagok megérkezésének a sorrendje, ezért a kritikus adatok, mint például szövegek küldése, bombák lerakása, írnom kellett egy egyszerű hibafelismerő és sorba rendező algoritmust UDP-re. Használhattam volna párhuzamosan minden kapcsolódási módot, ám ezzel csak tovább bonyolítottam volna az eddig is bonyolult rendszert.

Egy két ágból álló protokollt terveztem. Az egyik ág a pozíciófrissítésekre használható, az ilyen csomagok első bitje nullával kezdődik. Utána következik 7 biten keresztül az objektum azonosítója, amit frissíteni fogunk, majd az X és Y tengelyen elfoglalt helye, a következő formátumban. Mivel teljesen ki akarom használni a csomagok által biztosított keresztmetszetet, vettem a pálya teljes méretét, és arányosan nézve 0 – 255-ig néztem a koordinátákat. Mivel csak három bájtöt töltöttem ki, így bevezettem egy számlálót is, hogy ha egy csomag „eltévedne”, akkor ne frissítsen egy már nem aktuális állapotra a program. Tehát egy csomag így néz ki:

```
ID = 0xFF & id;  
C = ++prevC;  
X = posX / maxX * 256;  
Y = posY / maxY * 256;
```

A második ág, amikor fontos üzenetet küldünk. Ez lehet a csatlakozáshoz szükséges információ, szöveges üzenet a többi játékosnak, játékmenettel kapcsolatos állapotváltozások, bomba lerakása, szörnyek irányváltása. Az ilyen üzenetek első bitje egy, utána egy számláló van. Itt már fontosabb a szerepe, mivel nem engedhetjük

meg a csomagok összekeveredését. Hogy jelezni tudjuk egy csomag hiányát, minden csomagról nyugtát kell küldeni (ACK) a szervernek, aminek hiányában újra kell küldeni a csomagot.

Hosszas mérlegelés és tesztelés után arra jutottam, TCP protokollt fogok használni az egyszerűség és biztonság miatt. Ezzel a torlódást nehéz kiküszöbölni nagy hálózati forgalom esetén, de nagyban leegyszerűsödik a hálózat kezelése és biztonságosabbá is válik. A modellt úgy terveztem meg, hogy ne okozzon gondot a távoli és helyi játékosok megjelenítése. Elég csak a Control osztályt másképp meghívni a hálózati osztályból. Ha UDP protokollt használtam volna, nagyon nehéz lett volna kezelni az elveszett csomagok által okozott inkonzisztenciát. Emellett sorba is kell rendezni a csomagokat, de legalább figyelmen kívül kell hagyni azokat, melyek késve érkeztek, van már frissebb információ helyettük.

A legnagyobb probléma az, hogy lehetetlen minden játékos képernyőjén pontosan ugyan azt mutatni. Pár milliszekundum különbség a leggyorsabb hálózat esetén is lesz. Ezért azt a döntést hoztam, hogy minden játékosnak van igaza, ő dönt a beérkezett információk alapján.

7.3.3. Hálózati terv, második verzió

Az eredeti tervtől szinte teljesen eltértem. Rájöttem, hogy ezt így nagyon nehéz lenne megvalósítani, ezért úgy döntöttem fix méretű TCP csomagokat fogok csak küldeni a hálózaton. Ez nagyon jó döntésnek bizonyult, mivel még a feladatot ennyire leegyszerűsítve is sok problémával találkoztam. A csomagok nagyon kis méretűek maradtak, így nem volt túl valószínű a csomagok feltorlódása még viszonylag gyenge hálózati kapcsolat esetén sem.

Kihasználtam, hogy viszonylag kis bonyolultságú adatokat kell átküldeni. Négy tulajdonságot kell eljuttatni szinte minden esetben:

- Mi történt
- Kivel
- Hol, (x, y) koordináta

Ezeket az információkat el lehet tárolni két 1 bájtos és két 2 bájtos változóban. Ezzel nagyon kis méretű csomagokat hozhatunk létre, nagyon kicsi sávszélességre lesz szükség.

A legjobb megoldás az lett volna, ha egy TCP és egy UDP kapcsolatot is létrehozok, majd a „fontos” csomagokat TCP-n küldöm, a kevésbé fontosakat pedig UDP-n. Kevésbé fontosnak egyedül a mozgást ítélem meg, mivel ez a leggyakoribb üzenet és nem feltétlen az a fontos, hogy minden pixelre pontosan jelenjen meg az ellenfél.

Viszont így két kapcsolatot kell létrehozni, sokkal bonyolultabb kezelní és több a hibalehetőség. Például, mi a teendő, ha csak az egyiket sikerült létrehozni, vagy játék közben valamelyik megszakad.

Úgy döntöttem, ha lassú lesz az implementáció, akkor elgondolkodom a harmadik változaton, mivel az alapja mindenkinek ugyan az, csak a csomagok elküldési módja más.

7.3.4. A hálózati protokoll

Minden csomag azonos méretű, hat bájtos. Ez van, ahol veszteségekkel jelentkezik, de a csomagok nagy része a szörnyek és játékosok mozgása miatt keletkezik. Nem éreztem szükségesnek a további optimalizálást.

	Esemény	Azonosító	x	y
1	Mozgás	Játékos mozgás	x	y
2	Szörny mozgás	Szörny ID + irány	x	y
3	Bomba lerakás	Ki rakta le	x	y
4	Vírus lerakva	Típusa	x	y
5	Vírus fertőzés	Kit + mivel	x	y
6	Játékos meghal	Ki	x	y
7	Szörny születik	Új szörny ID	x	y
8	Fal épül	-	x	y
9	Pontszám megjelenítése	Játékos ID	Pontszám	-
10	Játékmód	Mód azonosító	-	-
11	Játékosok száma	n	-	-
12	Játék indul	1	-	-
13	Játékos ID beállítása	ID	-	-
14	Játékos csapata	Játékos ID	1/2	-
15	Fal azonnali építése	-	x	y
16	Eredménytábla frissítése	Játékos ID	Típus	Pontszám
17	Játékos újraélesztése	Játékos ID	-	-
Méret:	1 B	1 B	2 B	2 B

Az első csoport a játéktáblán történő, kizárolag játék közbeni állapotokat, a második csoport a játék hátterében történő eseményeket tartalmazza.

Amennyiben valamilyen nagy méretű üzenettel kéne bővíteni a protokollt, nagyobb módosítás nélkül is meg lehetne azt oldani. Több csomagot küldhetünk egymás után, majd egy záró csomag vélegesítené az üzenetet.

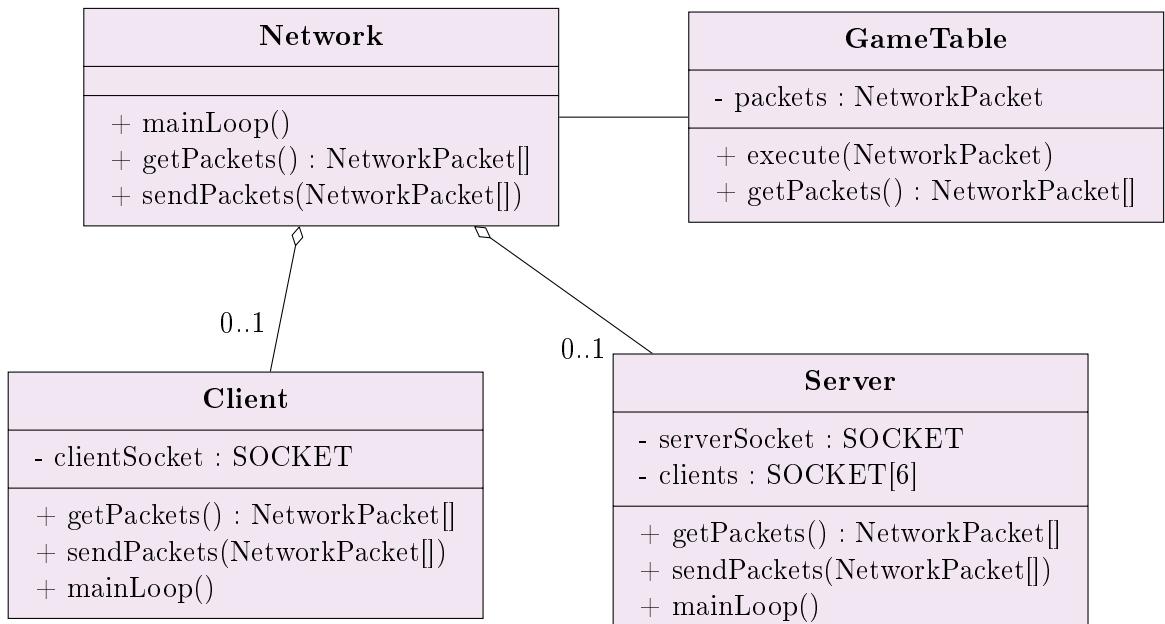
Ahol két tulajdonság is található (2 és 5-ös típusú csomag), ott egy adatba egy függvény segítségével kettőt csomagolunk. Így nagyban lecsökken a felhasználható számok mennyisége, de van ahol ez megengedhető.

7.3.5. A hálózat és a játéktábla

Az üzenetek feldolgozását egy külön osztály végzi. Két szerepkör van, szerver és kliens, amit a Network osztály fog egybe. Ez azért is fontos, mert valaki nem lehet egyszerre szerver is meg kliens is, ezt szerettem volna korlátozni.

Fontos azt is figyelembe venni, hogy nem lehet bizonyos döntéseket a kliensekre bízni a késés miatt, amit a csomagok érkezési ideje jelent.

Az ábra nem teljesen pontos, a lényeget mutatja. A Network és a GameTable között nincs közvetlen kapcsolat, a GameManageren megy át minden adat. Erre azért volt szükség, mert nem csak a GameTable használja a hálózatot.



7.4. ábra. A hálózat osztálydiagramja

7.4. Megjelenítés, bemenetek

Ennél a modulnál nem tekinthettem el az implementációtól, mivel a platformfüggetlenség érdekében egy külső függvénykönyvtárat akartam használni. Több alternatíva közül végül az SDL-t választottam, mivel egyszerű és minden tartalmaz, amire nekem szükségem volt.

7.4.1. A főprogram és az időzítés

A Game osztályban van a főciklus (main loop), ami terminálásig folyamatosan fut. Ez a ciklus tartalmaz késleltetést is, mivel fölösleges kihasználni minden erőforrást, elég ~ 60 képkockát kirajzolni másodpercenként. Ezt a *frameBegin*, *frameEnd*, *frameDelay* függvények biztosítják a következőképpen.

A *frameBegin* eltárolja az előző időt a *prev* változóban, majd lekérdezi a jelenlegi időt a *now* változóba.

A *frameDelay* kiszámítja, hogy mennyi idő telt el a két képkocka kirajzolása között. Erre azért van szükség, mert ez az idő változhat, az objektumok mozgatásánál pedig ezt figyelembe kell venni.

A *frameEnd* függvény kiszámítja mennyi időt kell várni, tehát a *now*-hoz hozzáadja a *TICK_INTERVAL* változót, ezt pedig eltároljuk a *next_time*-ban. Az *end* változóba eltároljuk a jelenlegi időt. Amennyiben *end* kisebb, mint *next_time*, a programot késleltetjük (elaltatjuk) *next_time - end* milliszekundummal. Ezt az *SDL_Delay* függvényen keresztül tehetjük meg, de a kompatibilitás, és jövőbeli fejlesztések miatt egy külön header fájlon keresztül.

7.4.2. Megjelenítés

A megjelenítésért felelős osztály a program indulásakor létrejön egy példányban és a futás végéig megmarad.

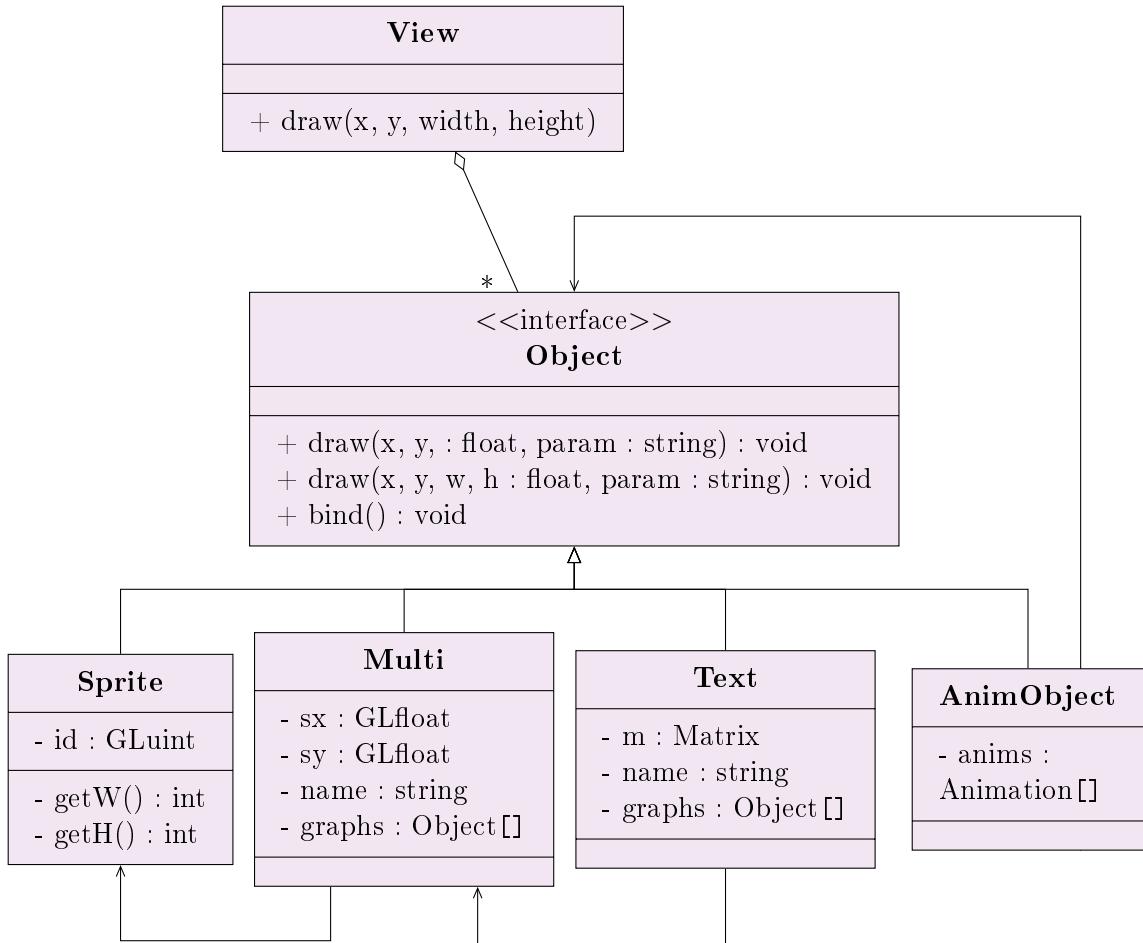
Hogy bármilyen megjelenjen a képernyőn, először létre kell hoznia egy ablakot, benne az OpenGL környezettel. Ezt az SDL segítségével valósítja meg. Az ablak lehet teljes képernyős, vagy átméretezhető. Ablak átméretezés esetén az osztály automatikusan átméretezi az OpenGL tartalmat is. Az ablak méretétől függetlenül rajzoláskor úgy kezelem a képernyőt, mintha 640×480 méretű lenne, lebegőpontos számokkal. Ez megkönnyítette a többi osztályban a programozást, hiszen nem kell a képernyő tényleges méretével foglalkozni, azt megoldja a megjelenítésért felelős rutin.

Ezek után használhatjuk a betöltő *load*, és kirajzoló *draw* függvényeket. Mivel elég sok fájl van, amit be kell tölteni, készítettem egy *loadPackage* nevű függvényt, ami betölti az összes fájlt, ami fel van sorolva egy szöveges fájlban. A fájlok különböző típusúak, mind az Object osztályból származik.

A kirajzolható objektumok típusai:

- Sprite osztály, kiterjesztése: png

A *Sprite* osztály egy OpenGL textúrát tölt be PNG fájlból. A betöltés a konstruktor meghívásakor történik, ahol meg kell adni a fájlnevet is. A fájl betöltését a LodePNG végezi, ami egy *vector<unsigned char>* konténerbe tölti a pixelek adatait. Ezt a *glGenTextures*, *glBindTexture* és *glTexImage2D* függvényekkel



7.5. ábra. Az Object osztályból származtatás UML-diagramja.

betöltjük a videomemóriába. Innentől kezdve elég az azonosítóját megjegyezni, a glBindTexture függvény meghívása után rajzolhatunk vele. A rajzoláshoz ki kell számolni a textúra és vertex koordinátákat, ezeket két tömbbe helyezni, majd a glDrawArrays függvény meghívásával rajzolhatjuk ki. A forráskódban megtalálható megjegyzésben a régi glBegin(GL_QUADS) módszer is, de ez a leglassabb mód a kirajzolásra, így nincs benne a végleges kódban.

Amikor töröljük az objektumot, a destruktur felszabadítja a textúrát a videomemóriából. Kirajzolni a *draw* függvénnyel lehet, amit többféleképpen paraméterezhetünk. Meg kell adni az $x \in [0, 640]$, $y \in [0, 480]$ paramétereket, melyek a kép helyét adják meg a képernyőn. Opcionálisan megadhatjuk a *w*, *h* szélesség, magasság értékét. Ha nem adunk meg semmit, akkor a betöltött kép méretét fogja használni. Végül megadhatjuk az *std::string param* értékét, amivel transzformációkat hajthatunk végre. (lásd 3.2. ábra) Ezek a transzformációk működnek a többi objektumon is, mivel lényegében minden egyik ezt az osztályt használja. A transzformációkat kombinálhatjuk egymással, ekkor egymás után hajtódnak végre. A param paraméterbe adhatjuk meg őket.

h	Vízszintes tükrözés
v	Függőleges tükrözés
l	Forgatás balra
r	Forgatás jobbra

7.6. ábra. Transzformációk

- Multi, kiterjesztése: mlt

A *Multi* osztály egy egyszerű fájlt tölt be, ami tartalmazza egy *Sprite* nevét, és hogy hány részre van feldarabolva vízszintesen és függőlegesen. Kirajzoláskor a *draw* függvény *std::string param* paraméterében adjuk meg, melyik részletet szeretnénk kirajzolni a képből, két egymás után következő számmal vesszővel vagy szóközzel elválasztva. Ezt a formátumot *stringstream* segítségével könnyen előállíthatjuk, és az öröklődésben sem okoz gondot, hogy különbözőek a paraméterek.

- Text, kiterjesztése: txh

A *Text* osztály felel a szövegek kiírásáért. Ezzel a téma körrel részletesebben is foglalkozok a következő fejezetben. Betölti a betűkhöz tartozó adatokat, ami egy karaktereket tartalmazó mátrix képként (*Multi* osztályt használva), és egy másik mátrix UTF-8 szövegként reprezentálva. A *draw* függvény *std::string param* paraméterében lévő szöveget írja ki a karakterek mátrixban való megkeresésével és kirajzolásával.

- AnimObject, kiterjesztése: anm

Az *AnimObject* osztálytalál időzített statikus animációkat rajzolhatunk ki. Mivel az animáció nem csak egy képkockáig él, így a *View* osztálynak el kell tárolnia külön is. Egy *.anm* fájl *Sprite*-ként betöltött képek nevéből, és millisekundumban megadott idők sorozatából áll, ami a késleltetést jelenti a következő képhez képest. A kirajzolásához először a már ismert *draw* függvényt használjuk ugyan olyan módon, ám ettől még nem fog megjelenni a képernyőn. Meg kell hívni a *View* osztály *drawAnim* függvényét, ez automatikusan kirajzolja a még futó animációkat, kiszámolja minden az aktuális képkockát és törli a már lejárt példányokat. Elég könnyen kezelhető, ám nem alkalmas minden feladatra. Nem akartam összekeverni a játékteret a megjelenítéssel, így ezt a fajta kirajzolást csak mellékhatás nélküli megjelenítésre használtam, ami nem befolyásolja a pályán történő dolgokat.

Hogy milyen típusú az objektum, betöltéskor a kiterjesztés alapján dől el. Egy *Object** típusú elemeket tároló vector-ban tároljuk őket.

8. fejezet

Megvalósítás

A programkódot teljes egészében dokumentáltam DoxyGen használatával. Megtekinthető a forráskód könyvtárán belül a doxygen könyvtárban.

8.1. Implementációs döntések

8.1.1. Matrix osztály és a pálya

A pálya elemeinek tárolásához és a szövegek kiírásához betűk eltárolására mátrixra volt szükségem. Úgy döntöttem, egy template osztályt hozok létre.

Az elemek elérésén kívül definiáltam a ki- és bemenet operátort, keresés és számlálás függvény. Ezek sokat segítenek a használatában. Ezeknek a megírása során vált világossá, mennyire egyszerű lesz előre elkészített pályákat beolvasni fájlból a jó felépítés miatt.

Eredetileg csak annyit terveztem a történet módba, hogy a pálya minden véletlenszerűen lesz legenerálva és a táblát vezérlő osztály határozza meg a paramétereit. De miután megírtam a Ground és Matrix osztályokat beolvasó operátorokkal együtt, rájöttem, hogy három sor lenne egy pályabetöltő függvény megírása. A pályákat szövegszerkesztővel lehet megszerkeszteni, a következő formátumban:

```
<egy négyzet mérete> <pálya eltolása, x> <pálya eltolása, y>
<pálya magassága, m> <pálya szélessége, n>
for i = 1..m
    for j = 1..n
        < ' .' : Út | 'f' : Fal | 'x' : Beton, 'W' : Győztes mező >
<következő pálya>
while !eof
    <szörny típus> <x> <y>
```

Opcionálisan lehetünk bele szóközöket, sőremeléseket, például a kirajzolandó sorok végére.

8.1.2. Szövegek kezelése

Szövegek kezelésére, például a menüben, szerverek, játékosok listájánál, számok kiírására a következő lehetőségeket találtam:

1. minden szöveghez külön képet készíteni.
2. Az operációs rendszer saját függvényeit használni.
3. A *FreeType* függvénykönyvtárat használni, esetleg az *SDL_ttf*-el kiegészítve az egyszerűbb használat érdekében.
4. Mátrixot készíteni a fontosabb betűkből, elmenteni képként és szövegként is, majd egy ezeket kezelő osztállyal minden kirajzolni a megfelelőt.

Az első lehetőség nem működik dinamikus tartalmakkal, emellett nagyon sok helyet foglalhat. A második jó ötletnek tűnik, de mivel én platformfüggetlen kódot szeretnék, nagy munka lenne minden operációs rendszeren megtalálni, hogy is érhetjük el ezt a funkcióját. Az *SDL* sajnos erre nem képes. Erre kínál megoldást a *FreeType*, amit az *SDL_ttf* is felhasznál, így a használata sem okozott volna gondot. Viszont ez sem tökéles módszer, mivel a szövegeket nem szabad állandóan kiszámoltatni, el kell tárolni a memoriában képként. Végül a negyedik lehetőség mellett döntöttem és érdekes kihívás volt leprogramozni. Emellett nem növelte a függvények számát sem és kisebb maradt a méret. Ellene szól, hogy csak azokat a betűket tudja kirajzolni, amiket előre kirajzoltattunk. Össze lehetne kombinálni az előző lehetőségekkel, hogy minden betűt ismerjen, mégis nagyon kevés memóriát foglalna.

Az implementációhoz először írni kellett egy Unicode [8] karaktereket kezelő osztályt */utf8.h/*. Az UTF-8 karakterek nem feltétlenül férnek el egy byte-on. A [0, 127] tartományon lévő karakterek megegyeznek az ASCII kódtábla karaktereivel. Ha ezen a tartományon felüli karaktert szeretnénk reprezentálni, akkor azt két (vagy több) byte-on kell eltárolni.

„Hasznos” bitek	Utolsó karakter	Byte 1	Byte 2
7	0x007F	0xxxxxx	
11	0x07FF	110xxxx	10xxxxxx

8.1. ábra. UTF-8 Karakterek bináris szinten

Az osztályom két byte-ig támogatja a karaktereket, de ebben az összes fontos karakter benne van a [128, 2047] intervallumon. A beolvásás és kiírás C++ stream-eken keresztül történik. A kirajzoláshoz a korábban ismertetett Multi osztályt használja.

Megkeresi a szöveges reprezentációban a koordinátákat, majd kirajzolja a megfelelő helyre a nagy kép abban a pozícióban lévő részét `/text.h`.

Megjegyzés: A C++11 szabvány már támogatja az Unicode karaktereket.

8.2. Felhasznált szoftverek

A dokumentációt *LaTeX* nyelven írtam, a *TexMaker* nevű szerkesztővel. Felhasználtam az *elteikthesis* [2] nevű kiegészítést, valamint a legfrissebb magyar szótagoló szótárt (*babel csomag*). Az UML ábrákat a *TikZ-UML* [9] nevű kiegészítéssel rajzoltattam ki.

A programot *C++* nyelven írtam, a kódöt pedig a *GNU-C++* fordítóval fordítottam le.

Több fejlesztői környezetet is használtam a program készítése során. Először a *Geany* nevű szövegszerkesztőt, és konzolból fordítottam. Később ez nem volt elég az igényeimnek, nehézkes volt kezelni a sok osztályt, ezért átváltottam a *CodeBlocks*-ra, amit az egyetemen is használtunk. Végül átváltottam a *NetBeans*-re, ami jó döntés volt, utólag belegondolva ebben kellett volna kezdeni is. Sokkal több a lehetőség benne, és jobb a felépítése. Átláthatóbbá teszi az egész rendszert. Nincs külön projektfájl hozzá, egy *Makefile*-t generál a fordításnál, így nem teszi magát a programot a fejlesztőkörnyezettől függővé. Ilyen méretű feladatnál mindenkor érdemes modern fejlesztőkörnyezetet használni, megkönyíti a program átláthatóságát.

A grafika elkészítéséhez a *GIMP*-et használtam. Nagyon jó nyílt forráskódú szoftver.

8.3. Az implementációhoz használt technológiák

8.3.1. OpenGL

Furcsának tűnhet, hogy egy kétdimenziós programhoz OpenGL-t használok, amit főleg három dimenzióra terveztek. Tervezéskor figyelembe vettet minden lehetőséget. Alapvetően kétféle rajzolás létezik a mai operációs rendszereken, hardveres és szoftveres. A szoftveres rajzolást általában egyszerűbb kivitelezni, de még egy ilyen egyszerű program esetében is performancia problémával találkozhatunk. Szoftveres rajzolás esetén az operációs rendszer dönti el, mikor rajzolja újra az ablakot. Így nem lehet biztosítani az ember számára folyamatosnak tűnő 60 képkocka / másodpercet. Ráadásul észrevehetően elég nagy CPU terheléssel jár.

Az OpenGL viszont a videokártyát használja a CPU helyett, a videomemóriában tárolja a képeket (textúrákat), így szinte egyáltalán nem terheli a CPU-t. A program így szinte elhanyagolható terheléssel, folyamatos 60 képkocka / másodperccel

tud működni. Csak háromdimenziós megjelenítésre van lehetőség, de ezen is könnyen meg lehet oldani sík megjelenítését, csak ki kell választani egy szeletet a térből, majd a virtuális kamerát fix módon „odaállítani”. Viszont ennek van egy hátránya is, nem tudjuk, legalábbis nem érdemes pixel pontosan kezelni a képernyőt. A megjelenítés egy lebegőpontos háromdimenziós koordináta-rendszer síkba való leképezésével történik. Ebből előnyt is kovácsolhatunk, mivel jól megválasztott képernyőkezelés esetén minden felbontáson jól fog megjelenni a program.

Egy nagyon fontos tulajdonság, hogy a videokártyák nagy része csak kettő hatvány dimenziójú képeket tud kezelni. A szélesség, magasság eltérhet, de csak 1, 2, 4, 16, 32, 64, 128, 256, ... pixel méretű lehet. Erre oda kell figyelni, de a képek nyújtása már hardveresen történik pillanatok alatt, ami CPU-val számolva nagyon lassú lenne. Ezen kívül a videokártyák arra is képesek, hogy egy képet több felbontásban tárolnak el és kirajzoláskor, a megjelenített mérettől függően a legközelebbit választják ki. Ezzel drasztikus sebességnövekedést lehet elérni és jobban is fog kinézni a program. A technika neve „mipmap”.

8.3.2. SDL

Az SDL (Simple DirectMedia Layer) [3] egy kiválóan dokumentált [4], egyszerűségre és kis méretre törekedő függvénykönyvtár. Multiplatform, rengeteg operációs rendszerre és architektúrára fordították le: Linux, *BSD, MacOS, Win32, BeOS. Ezeken kívül létezik Androidra is, és több, már elavult operációs rendszerre. LGPL licenc alatt áll, aminek a lényege, hogy zárt forráskódú programba nem lehet statikusan fordítani, de dinamikusan korlátok nélkül használható.

Mindent biztosít, amire a programomnak szüksége volt:

- Ablakkezelés. Csak egy ablakot tud megnyitni egyszerre. Szerencsére nekem elég is volt ennyi.
- Billentyűzet, egér kezelése eseményeken keresztül.
- Kontrollerek kezelése. Ezt azért vettem külön, mert sok hasonló függvénykönyvtár van az interneten, de amiket próbáltam, azok közül egyik se kezel ilyen könnyen és gond nélkül ezt a bemenetet.
- OpenGL környezet megnyitása. Az OpenGL ES (Embedded System) támogatása még fejlesztés alatt van, de van megoldás az integrálására. [7]
- CPU idő lekérdezése. Ha multiplatform fejlesztünk, fel kell készülni rá, hogy minden rendszer másképp működik. Ez különösen igaz a CPU idő kérdésére, a processzorok különböző mértékegységen mérhetnek, és más függvényeket kell használni. Az SDL használatával elég csak egyet.

- Hang lejátszása. Kiegészítések nélkül egyszerre egy hangot és csak WAV formátumban tudja lejátszani.
- Unicode karakterek bevitelle billentyűzetről.

Ezen kívül még tartalmazza a következő lehetőségeket:

- Párhuzamos programozás.
- Szoftveres rajzolás.
- CD-ROM kezelése.
- Kiegészítésekkel ezeken kívül még rengeteg minden. Például: Rich Text Format szövegek, True Type betűtípusok, hálózat, képek betöltése különféle formátumokból, több csatornás hangkezelés.

Az előbb leírtak az SDL 1.2 verzióra vonatkoznak, amit a programban is használtam. Már megjelent a 2.0 verzió, ami egy teljesen újragondolt függvénykönyvtár, nem is kompatibilis az elődjével, de ez nem is gond, mivel a mai igényekhez van igazítva. Hivatalosan is támogatja a mobil platformokat (Android, iOS) többek között az érintőképernyő esemény szintű kezelését. Az elődje is támogatta nem hivatalosan némelyik verziót, ám nem volt igazán hatékony a fejlesztés vele.

A licence is megváltozott, már nem GPL, hanem zlib licenc, ami lehetővé teszi zárt forráskód mellett is a statikus linkelését és lényegében bármilyen felhasználását.

8.3.3. Képek betöltése

Képek betöltéséhez az SDL biztosítja a BMP formátumot. Ez nem volt elegendő számomra, mivel nem kezeli az átlátszóságot és túl nagy a mérete. Az *SDL_image* képes erre, de mellékelni kell hozzá több függvénykönyvtárat is, és szintén nagy a mérete.

A *LodePNG* [6] nevű függvénykönyvtár jelentett erre megoldást. Nagyon kis méretű (~ 60 kB lefordítva Linuxon), emellett egy forrás és egy header fájlból áll minden össze. Használható C és C++ nyelven is, utóbbin rendes C++ stílusban, a vector és string konténereket alkalmazva, csak a kiterjesztését kell átirni. Használata nagyon egyszerű, és egyértelmű, néhány függvényt kell csak meghívni. Dokumentálva nincs túlzottan, de sok példa van a használatára a hivatalos weboldalán. Szabadon felhasználható bármilyen célra.

8.3.4. Bemenetek kezelése

A bemenetek az SDL függvénykönyvtáron keresztül érkeznek az Input osztályba. Az `SDL_PollEvent(event)` függvény olvassa be, amit utána elágazásokkal kezelhetünk. Itt tudjuk lekezelni, ha átméretezik, bezárják az ablakot, lenyomnak vagy felengednek egy billentyűt a billentyűzeten, vagy egy joystickon. A joystickoknak egyedi azonosítójuk van.

Hálózati játékos irányítása vagy a menüben lépkedés esetén kényelmes, ha az összes bemenettel irányítani lehet. Erre lehetőséget is adtam egy külön virtuális vezérlő alkalmazásával, ami az összes bemenetről érkező adatot összesíti.

A játéktábla szempontjából mindegy, hogy egy játékost hogyan irányítanak, billentyűzetről vagy joystickról. Ezért, hogy egyszerre tudjam kezelní őket, elkészítettem a Control osztályt, ami lényegében egy interfész a kettő között. Ezt használja fel a GameTable.

Sajnos a hálózatnál nem alkalmazhattam ezt, mert a csomagok nem rendszeresen jönnek és teljesen másképp kell kezelní ezt a problémát.

8.4. Ütközésvizsgálat

A játékosok ütközésvizsgálatának egy külön fejezetet szánok, mivel nagyon nehéz feladat volt. Az alap elgondolásom szerint a játékos mehet bármerre, majd egy ütközésvizsgálat algoritmus visszateszi a legközelebbi járható helyre, ha nem jó helyen van. De nem műköött az egyszerű elgondolás, miszerint ha a játékos falba ütközik, akkor nem mehet tovább és visszateszem oda, ahonnan jött. Ha ezt használtam volna, a játékos csak akkor tudott volna irányt váltni, ha pixel pontosan a kereszteződésben állt volna meg, és amennyiben bombát rakott volna le, azonnal elakad. Ez nyilvánvalóan nem engedhető meg, viszont a szörnyeknél kis módosítással kiváloan bevált.

Sok kísérletezés után végül arra jutottam, hogy elég négy irányba vizsgálni a falakat (fel, le, jobbra, balra) és közvetlen a játékos alatt. A középpont, amihez viszonyítva vizsgálunk mindig egészre kerekítve a legközelebbi négyzetet a játékoshoz. A bombákat viszont külön kellett kezelni. mindenkor figyelembe kell venni a vizsgálatnál, hogy be lehessen zárni a többi játékost. Az egyszerű elgondolás, hogy ha a játékos lerak egy bombát, az ne számítson bele az ütközésbe, amíg le nem lép róla. De ez az elgondolás is rossz. Ha figyelembe vesszük, ki rakta le a bombát, nem jó, mivel sokszor szinte egymáson mennek a játékosok, és ekkor a másik ok nélkül elakadna. Végül az lett a megoldás, hogy a bombától csak távolodni lehessen, a közepe felé menni ne. De még ez is kevés, mivel ha egy játékos több bombát is lerakhat, be tudja zárni saját magát két bomba közé. Erre a megoldás az volt, hogy az előző

pozícióján ne vizsgálja a bombákat, így csak lemenni tud róla, visszamenni nem. Ez így leírva nem tűnik bonyolultnak, de rengeteg sok kísérletezés van mögötte. Továbbá észrevettem, hogy néha „ugrálnak” a játékosok amikor egy fal sarkának ütköznek. Ilyenkor minden beállítottam a legközelebbi járható helyre a játékost, figyelembe véve a kereszteződést. De gyakran ez az elmozdulás nagyobb volt, mint amennyit lehetett volna az időzítési szabályok szerint. Ezért el kellett tárolni az előző pozícióját is és aszerint vizsgálni, mekkora sebességgel lehet a járható út felé.

Összességében elég jól sikerült az ütközésvizsgálat, ez volt talán a legérdekesebb probléma amit meg kellett oldani és nagyon sok kísérletezés kellett hozzá.

A játékosok és szörnyek mozgatása, az ütközésvizsgálat és a bombák kezelése is a játéktábla move függvényében kerül meghívásra, a megfelelő sorrendben.

A hálózat kezeléséhez a játéktáblának nincs köze, csak a helyi játékos szemszögéből. Róla tudni kell, elmozdult-e, mert ha nem, akkor fölösleges a pozícióját frissíteni. Tudni kell mikor rakott le bombát, és az mikor fog felrobbanni.

8.5. Tanulságos hibák

8.5.1. Időzítés

Egy nagyon ritkán előforduló, de annál komolyabb hibát kellett kijavítanom a programnak ebben az egyszerű részében. Rossz változót használtam a késleltetésnél, emiatt előfordulhatott az, hogy a késleltető függvény lévő kivonás negatív paraméterrel került meghívásra. Azonban ez egy előjel nélküli egész szám kell, hogy legyen, tehát a végeredmény túlcordult. Így a függvény lényegében 4 294 967 295 milliszekundumra (~ 50 nap) elaltatta a programot, bezárni is nehezen lehetett.

8.5.2. Képernyőkép készítés

Implementáltam egy képernyőkép készítő függvényt is, mivel a LodePNG képes kódolásra is és ki akartam próbálni. Ez egyáltalán nem szerepelt a tervezésben, csak a kíváncsiság miatt készült. Egy frame kirajzolása után meghívom a glGetPixel függvényt, majd átkódolom png fájlformátumba, utána lementettem screenshot_n+1 néven. Külön meg kell keresni, hogy hanyadik kép következik, hogy jó legyen az elnevezése. Sajnos ezzel így van egy probléma, az OpenGL fordítva értelmezi az ytengely koordinátáit, mint a png. Ezért meg kell fordítani a képet. A kódom első verziója:

```
for i = 1 .. y do
    for j = 1 .. x do
        swap( t[i*x + j], t[(y-i)*x + j] );
```

Az eredmény pixelre pontosan ugyan az volt, mint a megfordítás előtt. Nem értettem miért nem működik, minden megpróbáltam már, amikor rájöttem, hogy kétszer fordítom meg a képet. A ciklust csak $y/2$ -ig kell futtatni. Ez egy elég tanulságos hiba volt számomra. Egy kódrészlet, ami látszólag nem csinál semmit, általában rossz változót használunk vagy nem fut le, nem végzi el a változtatásokat, ezért volt nehéz megtalálni.

8.5.3. Hálózati rész és a játéktábla integrálása

Ezt a részt nem terveztem meg eléggyé és nehézkes volt megoldani a két modul integrációját. Ha újra kéne kezdenem a tervezést ez lenne az első rész, amit átgon-dolnék.

A legnagyobb problémát az okozta, hogy ugyan készítettem interfészt a beme-netek és a játéktábla közé, nem volt elégséges minden hálózati üzenet átadására.

Ettől függetlenül egyáltalán nem lett rossz a megoldás, amit írtam, de legközelebb másképp csinálnám.

Nem kéne külön kezelni ennyire, hogy távoli vagy helyi játékos van. Azt a hibát is elkövettem, hogy implementáláskor elkészülttem a helyi játékosok kezelésével és minden funkcióval, majd ezután kezdtem a hálózatnak. Jobb lett volna a két részt együtt írni, akkor hamarabb fény derült volna a terv hiányosságára és át lehetett volna gondolni jobban.

8.5.4. Kerekítési hiba

Egy érdekes hiba kizárálag akkor jött elő, ha Release módban fordítottam a prog-ramot. Amikor a bal felső sarokban a fal felé próbáltam menni, a játékos elkezdett „villogni”. Jobban megvizsgálva látszódott, hogy nem villog, hanem fel-le megy az animáció szerint.

A probléma a Way függvényben volt, ami az útirányt határozta meg. Ebben az előző és a jelenlegi koordinátát vizsgálom meg. Azt vártam volna, hogy egyenlőség lesz és STILL értékkel tér vissza. Ehelyett UP és DOWN értékek váltakoztak.

A megoldás egy epsilon érték bevezetése volt (0.01), és a különbséget vizsgáltam, hogy ennél kisebb-e.

9. fejezet

Tesztelés

Egy ilyen játékot tesztelni nem egyszerű, mivel nincsenek könnyen tesztelhető ki és bemeneti paraméterek. Elég nehézkes lenne automatikus teszteket írni, személy szerint én nem is látnám értelmét. Ez nem egy üzleti alkalmazás, nem is annyira kritikus a hibákra. Néha egy hiba még érdekesebbé is tehet egy játékot, például ha egy nehezen kivitelezhető billentyűkombinációval némi előnyt lehet szerezni, az sok játékosnak kihívást jelenthet. A legfontosabb az volt, hogy robusztus legyen a program. Minél többféle számítógépen és konfigurációval kipróbáltam.

- Ubuntu Linux 12.04
- Windows 7
- Windows Vista (nem a kész verziót)

Négyfél kontrollerrel volt lehetőségem kipróbálni az irányítást, mindenekkel megfelelően működött. Ez nem jelenti azt, hogy az összessel működni fog, de jó előjel.

A tesztelés közben arra derült fény, hogy egyes videokártyák vagy a meghajtó-programok hiánya akadályt jelent a megjelenítésben, tehát csak egy fekete képernyő jelenik meg, hibajelzés nélkül. Elég nehéz volt erre megoldást találni.

Játék tesztelésére a legjobb módszer, ha rábízzuk másokra. Megmutattam sok embernek a játékot és megkértem őket, játszanak vele. Többféle korosztályon is kipróbáltam, nagyon tanulságos dolgokra derült fény, amit mint fejlesztő, én sosem vettet volna észre, hiszen természetesen veszem. Például aki még nem játszott hasonló játékkal, nem triviális neki a játékosok mozgatása. Ezen kívül a bónuszok, amiket a pályára helyeztem, egyáltalán nem voltak beszédesek, nem is értették sokan mire jók. Emiatt döntöttem úgy, hogy létrehozok egy tanulópályát, ahol megmutatom az alapvető dolgokat, ami a játékhoz kell.

Ezen kívül sok hiba csak nagyon ritka esetekben jön elő. Ezeket egyedül elég bonyolult lenne tesztelni, hiszen a játék leginkább a többjátékos módra van kitalálva.

Végeztem performancia tesztet is a következőképpen, elindítottam teljes képernyős módban egy játékot, majd hagytam, hogy a szörnyek ellepjék a pályát. Egy átlagosnak mondható számítógépen a CPU teljesítménye 5% volt és 6 MB memóriát foglalt el. Általában az ilyen teszteknél nehéz az eredményt kiértékelni, de azt hiszem ez az eredmény magáért beszél.

Ezekben kívül bevezettem egy teszt paramétert, amivel lassú számítógépeket lehet szimulálni. Ezzel a módszerrel is elég sok hibát sikerült találnom. Például az ütközésvizsgálat nem volt felkészülve arra, ha egy képváltásnál egyszerre több mezőt is ugrik a játékos. Ez csak nagyon extrém esetben fordulhat elő, mégis előjöhet. Extrém gyors módban is kipróbáltam, de ez nem okozott gondot a programnak.

Az érdekesebb és tanulságos hibákat leírtam azokban a fejezetekben, ahol előfordultak.

9.1. Végfelhasználói tesztelés

Amikor mindenkel elkészültem, a dokumentációban levő képeket frissíteni kellett, mivel némileg megváltoztak az állapotok. Ez remek alkalom volt arra, hogy végigpróbáljam az összes funkciót.

Nem vezettem pontos tesztelési jegyzőkönyvet, de találtam néhány kellemetlen hibát.

- A program mellé bizonyos DLL-eket kell csomagolni Windows alatt, különben nem fog elindulni más számítógépeken.
- A menüben miután használtuk a Csatlakozás menüpontot, nem tudunk újra helyi játékot játszani, mert beragad. Mindig a csatlakozás dialógust hozta fel.
- A szövegek olvashatatlanok néhol. Ezt nem is olyan egyszerű észrevenni, hiszen a programozó tudja, minek kéne ott lenni, és az is van.
- Néhány része a programnak nem elég informatív, például a csapatválasztásnál nem volt információ a jobb/bal nyilak használatáról és az eredménytáblánál sem egyértelmű, hogy egy gombnyomással át lehet ugrani.
- Miután lejátszottuk az 5 kört, csak az Esc gombbal lehetett visszatérni a menübe hálózati módban.
- Amikor a kliens vagy a szerver az eredménytáblát nézte, át tudta ugrani azt. Ez inkonzisztens állapothoz vezetett köztük.
- Csapat és vadászat módban, hálózaton a falak elkezdtek épülni, de végül nem maradtak meg.
- A történet mód negyedik pályája szinte lehetetlen, könnyítettem rajta. De még így is nehéz.
- A hálózat megszakadása problémákat okozhat. Ugyan volt hibaüzenet, de néhol azonnal el is tűnt, így a felhasználó nem érzékelhette.
- Nem volt a programnak ikonja. Apróság, de „profibbnak” tűnik tőle a játék.
- A program indításakor megjelent a konzol is. (-mwindows linkelési paraméter kellett)

Amin én is meglepődtem, amikor először három kontrollert csatlakoztattam, hogy egyáltalán semmi problémát nem okozott, pedig a bemenetek kezelése egy bonyolultabb és sok kihatással járó rész.

9.2. Követelményspecifikáció szerinti tesztelés

Végül úgy döntöttem, nem szükséges a követelményspecifikáció szerint is végigtesztelni minden, hiszen a végfelhasználói teszt közben még mélyebben leteszteltem a programot. A követelmények nem annyira specifikusak, nem mennek annyira a részletekbe, mint mondjuk a felhasználói dokumentáció.

Ez lehet a követelményspecifikáció hiányossága is, de annak is megvolt a feladata a tervezés során. Nem feltétlen probléma, ha a specifikáció biztosít valamekkora szabadságot.

Összességében, miután egy ideje nem találtam már több hibát, úgy gondolom jól letesztelt programról van szó.

9.3. Ismert hibák

9.3.1. Kapcsolat megszakad

Amennyiben a szerver bontja a kapcsolatot, a kliens nem mindig érzékeli ezt hibaüzenet formájában.

Ilyenkor a szörnyeken lehet észrevenni, hogy nem váltanak irányt, majd egyszer csak kimennek a pályáról.

Leszögezném, hogy ezt más játékok is „timeout” formájában kezelik. Nem tartottam fontosnak a javítását, mivel a felhasználó lényegében értesül a hibáról és a program futása sem szakad meg.

9.3.2. Számítógép lefagyása

Ha a számítógép lefagy hosszabb időre (körülbelül több, mint két másodperc), a késleltetés számítás túl nagy lépésre kényszerítheti a játéktáblát. Ilyenkor előfordulhat, hogy egy szörny vagy a játékos átugrik pár mezőt.

A játékosok és szörnyek nem tudják elhagyni a pálya területét, de bekerülhetnek esetleg elzárt helyekre.

10. fejezet

Bővítési lehetőségek, a jövő

Talán minden programra igaz, sosem mondhatjuk azt, hogy elkészült teljesen. Úgy gondolom ezzel jobb is tisztába lenni és arra törekedni, hogy ami van, az jól működjön, hibamentesen.

Fontosnak tartom azt is, hogy ez ne csak egy diplomamunka legyen, ami a fiókban porosodik, hanem a feladatának végeztével is fejlődjön tovább, használják az emberek.

10.1. Fejlesztési ötletek

A következőkben a fejemben lévő fejlesztési ötleteimet fogom részletezni. Remélem, a jövőben minél több meg fog valósulni belőlük.

10.1.1. Mesterséges intelligencia

Ez a téma a kezdetek óta foglalkoztat. A rendszer is fel van rá készülve, hiszen egy MI játékos lényegében olyan lehetne, mint egy kliens, aki döntéseket hoz.

Tervek is születtek a működésére, de a megvalósításig nem jutottam el. Egy játékos három állapotú lehetne, támadó, menekülő és bónuszkereső. A pályán utakat keresne, megnézné, hogy a többi játékos hol tudja keresztezni az útját, milyen bombák robbanhatnak fel a környékén

10.1.2. Igazi 3D megjelenítés

Jelenleg ugyan lehetőség lenne háromdimenziós megjelenítésre is, én ezt nem használtam ki. Egyszerűbb volt képeket rajzolni és azokat megjeleníteni a síkban. A grafikán sokat javíthatna, ha legalább a pályát térben rajzolnám ki és fényekkel árnyékolnám.

10.1.3. Hangok

A hangok lejátszására is alkalmas lehetne a keretrendszer. Akár a View rétegbe is be lehetne integrálni, mivel ha egy olyan esemény történik, amihez hang is tarthatna, azt ki is rajzoljuk.

Nem éreztem nagy prioritásúnak ezt a dolgot, így végül nem implementáltam le és tartalmat sem kerestem hozzá.

10.1.4. Játékos közösségi kiépítése

Ahhoz, hogy egy játék sikeres legyen az kell, hogy használják.

Első lépés, hogy ingyenesen hozzáférhetővé teszem, forráskóddal együtt. A programkódot angolul írtam, így az esetleges hibák javítása könnyen menne.

Következő lépésként létre kéne hozni egy „mester szervet”, aminek segítségével a játékosok megtalálhatják egymást.

10.1.5. Újabb játékmódok

A játéktábla kis módosítással tudná kezeln a képernyő görgetését, fel van készítve rá, csak a sarkok eltakarását kéne megoldani. Elég lenne elő rajzolni a hátteret, az eltakarná a fölösleges részeket.

Ez rengeteg új lehetőséget adna. Ezek után nem lenne olyan kicsire korlátozva a pálya mérete, lehetne bonyolultabb pályákat is készíteni és bonyolultabb módokat.

Elsőként egy olyan módot valósítanék meg, amiben szintén csapatokra vannak osztva a játékosok, de nem egymást kell felrobbantani, hanem az ellenfél bázisára betörni, onnan pedig egy zászlót elhozni. Ezt a jelenlegi játékba is terveztem, de a pálya kis mérete miatt végül elvetettem.

10.1.6. Többnyelvűség

Erre mindenkor szükség lesz, mivel elég sok embert kizár az, hogy csak magyarul van a program. Elég lenne egy közös Singleton osztály (akkorcsak a Config), ahonnan a szavakat, mondatokat lehet lekérdezni. Az első olvasásnál töltöné be a .lang fájlt, akárcsak a loadPackage() eljárás.

10.1.7. Pontszámítás

Jelenleg a történet mód talán nem ad elég kihívást azzal, hogy nem méri a játékos teljesítményét. Egy egyszerű pontszámítással ezen is segíteni lehetne, ami figyelembe veszi az eltelt időt, megölt szörnyeket, és hogy hányszor halt meg a játékos.

11. fejezet

Nyílt forráskód

A program teljes egészében nyílt forráskódú és elérhető a következő címen:

<https://github.com/bmateusz-inc/bombas-jatek>

MIT licenc alatt tettek közzé, ami lényegében bármilyen felhasználást és módosítást engedélyez. Remélem lesz jövője, és lesz, aki játszik vele.

Irodalomjegyzék

- [1] Paul Martz, OpenGL röviden, *Kiskapu kiadó, 2007*
- [2] Majoros Dániel, elteikthesis class for LaTeX, 2010. *Letöltés ideje: 2012.09.17.*
<http://www.ctan.org/tex-archive/macros/latex/contrib/elteikthesis>
- [3] Az SDL hivatalos oldala *Letöltés ideje: 2012.09.18.*
<http://www.libsdl.org/index.php>
- [4] Az SDL dokumentációja *Letöltés ideje: 2012.09.18.*
http://www.libsdl.org/cgi/docwiki.cgi/SDL_API
- [5] GNU Lesser General Public License *Letöltés ideje: 2012.09.18.*
<http://www.gnu.org/licenses/lgpl.html>
- [6] A LodePNG hivatalos weboldala *Letöltés ideje: 2012.09.18.*
<http://lodev.org/lodepng/>
- [7] Combining OpenGL ES 1.1 and SDL *Letöltés ideje: 2012.09.18.*
[http://pandorawiki.org/Combining_OpenGL_ES_1.1_and SDL_to_create_a_window_on_the_Pandora](http://pandorawiki.org/Combining_OpenGL_ES_1.1_and	SDL_to_create_a_window_on_the_Pandora)
- [8] UTF-8 (UCS Transformation Format—8-bit) *Letöltés ideje: 2012.09.25.*
<http://en.wikipedia.org/wiki/UTF-8>
- [9] Maurice Diamantini, TikZ-UML for LaTeX *Letöltés ideje: 2012.09.27.*
<http://www.ensta-paristech.fr/~kielbasi/tikzuml>