

Debreceni Egyetem
Informatika Kar

Java játékfejlesztés webes felületen

Témavezető:
Dr. habil. Boda István
tanszékvezető egyetemi docens

Készítette:
Kosina Zoltán
informatikus-könyvtáros BSc.

Debrecen
2009

Tartalomjegyzék

Bevezetés.....	3
A Java programozási nyelv.....	6
A számítógépes játékok története.....	8
Virtuális világok.....	11
A látványvilág megtervezése.....	12
Main.java.....	14
Utveszto.java.....	31
Szereplo.java.....	37
MozgoSzereplo.java.....	39
SzilardMozgoSzereplo.java.....	42
Cel.java.....	46
Celkereso.java.....	47
Falnezo.java.....	49
Main.html.....	50
Összefoglalás.....	51
Irodalomjegyzék.....	53
Függelék.....	55

Bevezetés

Rohanó világunk egyik legfontosabb kérdése, hogy hogyan is vagyunk képesek kikapcsolódni. A mindennapi munka mellett – mint azt mind nagyon jól tudjuk – az embernek szüksége van pihenésre. Ez a pihenés pedig nem korlátozódhat csupán a passzív pihenésre, azaz az alvásra, szükség van valamilyen aktív pihenési tevékenységre is. Az egyik legnépszerűbb aktív kikapcsolódási forma pedig maga a játék.

Az emberek az idők kezdete óta játszanak, hogy színesebbé tegyék mindennapjaikat, játszanak, hogy szórakoztassák gyerekeiket, vagy magukat, játszanak, hogy kicsit félretehessék gondjaikat, hogy kicsit felszabaduljanak problémáik alól. A rádió és a televízió is gyakran közvetít játékokat, hogy a passzív pihenést választókat is szórakoztassák, ám a számítógép megjelenésével ezek igencsak háttérbe szorultak. A számítógép ugyanis egy teljesen új kikapcsolódási formát hozott magával, a filmszerű élményeket nyújtó, mégis a végtelenségig interaktív számítógépes játékokat. És sorakozhatott akárhány játék a gyerekek polcain, az íróasztalon mégis százszor annyi elfért egyetlen gépbe sűrítve. Megannyi színes, hangos, kalandos mesevilág, melynek főhőse maga a gyerek lehetett, a játékos. A számítógépes játékok jöttek, láttak, és akit csak lehetett meggyőztek arról, hogy jobbak, mint a kézzel fogható, „hagyományos” játékok.

Amióta ugyanis létezik a számítógép, számítógépes játékok is léteznek. Kezdetben persze még nagy jóindulattal is csupán kis felhasználói körben élvezhető megoldások születtek, ám az egyre szebb megjelenítésű monitorok, és az egyre nagyobb teljesítményű gépek révén a számítógépes játékok villámgyors fejlődésnek indultak. Mára pedig odáig jutott ez a fejlődés, hogy a legtöbb játék már már önálló művészi alkotásnak is felfogható.

A legsikeresebb játékokon ma már többen dolgoznak, mint a legtöbb nagy költségvetésű filmen, a kiadók és a játékcégek ugyanis nem sajnálják a pénzt arra, hogy készülő játékuk minden részletét a terület legtapasztaltabb, legtehetségesebb embere készítse. A játékok képi világát festőművészi képességekkel felvértezett grafikusok készítik, a hangokat híres színészek mondják fel (Grath 2004 : 22. p.) a zenét pedig – melyeket híres komponisták alkottak – igazi nagy zenekarok adják elő.¹ Ez utóbbiban ráadásul még mi magyarok is az

¹ Echoes of War <http://www.echoes-of-war.com/>

élen járunk annak ellenére, hogy erről szokás szerint ismét csak kevesen tudnak. Az egyik nagy sikerű játék, a Hitman zenéjét például a Magyar Rádió és Televízió Szimfonikus Zenekara játszotta fel, mely külföldön Budapest Symphony Orchestra néven vált elismertté. (Mazur 2002 : 64. p.)

Ez az úgynevezett számítógépes játékipar pedig nem is létezhetne, ha az emberek nem igényelnék a játékokat. Így ha valaki manapság programozásra adja a fejét, nem kell attól tartania, hogy elítélik, ha nem komoly kutatásokban vesz részt, hanem a nem kevésbé komoly játékfejlesztésben. Egy webprogramozó számára pedig maga a játékkészítés is a játékot jelenthet, szó szerint játszva tanulást, hiszen egy-egy webes környezetben futtatható játék elkészítése után a komolyabb webes alkalmazások elkészítése sem jelent már kihívást. A mesterséges intelligencia leprogramozása pedig olyan tapasztalatokat adhat, mely még túl is szárnyalja a webes alkalmazásokkal szemben támasztott igényeket.

Ezért döntöttem hát úgy, mint kezdő webprogramozó, hogy egy webes környezetben futtatható számítógépes játékot készítek. S mivel tanulmányaim során a Java nyelvvel ismerkedtem meg behatóbban, kézen fekvő volt, hogy az említett játékot Java appletként fogom megvalósítani. A dolgozatom témája tehát adott volt, ám mivel egy számítógépes játék készítését tűztem ki célomul, a játék műfaját is ki kellett választanom.

Mivel a számítógépes játékok mesterséges intelligenciájával már korábban is foglalkoztam, és jó néhány PC játékhoz készítettem kisebb programrészleteket a játék beépített szerkesztőjével, egyértelmű volt számomra, hogy az általam készített útkeresési algoritmusok egyikét alkalmazó játékot készítek majd. Azt is be kellett látnom viszont, hogy komolyabb, sőt a jelen kor elvárásainak megfelelő grafikus felület készítéséhez nincsenek meg a szükséges képességeim, tapasztalataim, ezért egy olyan játékot kell majd készítenem, mely leginkább a számítógépes játékok múltját idézi majd.

A választásom végül egy úgynevezett első személyű nézőpontból irányítható labirintusos játékra esett. Már a dolgozat elkészítése előtt is ismertem több ilyen játékot, ám arról csak a kutatásaim során értesültem, hogy a legelső számítógépes játékok egyike is épp egy ehhez hasonló játék volt.² Így elhatároztam, hogy dolgozatom nem csupán a Java nyelv lehetőségeit szemléltető munka lesz, hanem egyfajta tisztelgés is a mára több, mint negyven éves múltra visszatekintő számítógépes játékok előtt.

² Videojátékok története <http://oli76.ingyenweb.hu/keret.cgi?/e04.htm>

Dolgozatom készítésénél az úgynevezett Dungeon Master³ típusú játékok általam is tapasztalt eszközeit használtam, ám bizonyos elemeket más jellegű játékokból is kölcsönöztem. Közben pedig megpróbáltam alkalmazkodni a kor elvárásaihoz is, és felhasználtam az összes általam ismert számítógépes játékkal kapcsolatos ismeretemet, hogy nem csupán reprezentatív, de szórakoztató játékot is készítsek.

A programok elkészítéséhez a Java SE Development Kit 6 Update 10-es verzióját használtam, és a NetBeans IDE 6.5-ös verzióját. A játék grafikus elemeinek megszerkesztéséhez a GIMP 2.6.5-ös verzióját használtam, a befejező kép Windows Vista alap háttérképei közül való, a zenék a Hexen: Beyond Heretic című játékból származnak, melyet 1999-ben nyílt forráskódúvá tettek⁴, a hangok megfelelő formátumúvá alakítását a Free MP3 WMA WAV Converter című programmal végeztem el, a kisebb hangformázásokat pedig a WavePad című szoftverrel. Végül pedig a dolgozat szövegét az OpenOffice.org 3.0.1 című programmal készítettem.

Ezen ingyenes programok felhasználásával, legjobb tudomásom szerint törekedtem arra, hogy az általam kitűzött célokat megvalósítsam, és a szakdolgozatom hasznos részét képezze a számítógépes játékokkal kapcsolatos tudományos munkák - véleményem szerint - méltatlanul szűk körének.

³ Dungeon Master (video game) [http://en.wikipedia.org/wiki/Dungeon_Master_\(computer_game\)](http://en.wikipedia.org/wiki/Dungeon_Master_(computer_game))

⁴ Hexen: Beyond Heretic <http://en.wikipedia.org/wiki/Hexen>

A Java programozási nyelv

A Java a Sun MycroSystems⁵ által kifejlesztett programozási nyelv, mely először a '90-es évek elején jelent meg, ám mind a mai napig fejlesztik.

„James Gosling és társai a C++ nyelvet egyszerűsítették le a kívánt cél érdekében, hiszen unixos környezetből jöttek, és addig C++-ban dolgoztak. Az új nyelvnek eredetileg az Oak (tölgy) nevet adták, mint mondják, azért, mert a Sunnál Gosling ablaka előtt egy gyönyörű szép tölgyfa állt. Később azonban a Sun emberei felfedezték, hogy Oak programnyelv már létezik. És miközben azon töprengtek, mi is legyen az új név, nagy élvezettel fogyasztották gőzölgő kávéjukat. E finom, történelmi kávé a Java nevet viselte, utalva ezzel származási helyére...” (Angster Erzsébet 2003 : 172. p.)

A Java egy objektum-orientált programozási nyelv, ami a programozási stílusra, és a nyelv struktúrájára utal. Nem az elvégzett feladatokra helyezi a hangsúlyt, mint az eljárás-orientált nyelvek, hanem a később is hasznosítható programrészekre, az objektumokra. A feladatok ugyanis állandóan változnak, ám azok az objektumok, amiket a feladatok megoldásához használunk, már kevésbé. Így egy objektum-orientált alkalmazás sokkal könnyebben fejleszthető, és továbbfejleszthető a tetszőlegesen lecserélhető objektumoknak hála, mint egy eljárás-orientált. Az eljárás-orientált alkalmazások lényege ugyanis a minél egyszerűbb, minél rövidebb, minél frappánsabb forráskód, mely az adott feladat elvégzésére a leoptimálisabb. Ám ha a feladat módosul, néha az egész kódot befolyásoló változtatásokat kell eszközölnünk, mely nagyban megnehezíti az alkalmazás fejlesztését. (Angster Erzsébet 2003 : 59-60. p.)

Az objektum-orientált programozási nyelvek kifejlesztésének célja az volt, hogy a nagy fejlesztési projekteket könnyebben lehessen kezelni, így csökkentve az elhibázott projektek számát. Az objektum-orientált nyelvek révén ugyanis egy-egy programot nyugodtan szét lehet osztani több programozó között, az objektumokra való utaláskor ugyanis nem kell ismernie mindenkinek a többiek programrészletét, elég csak azt tudniuk, mire lesz jó az, majd ha elkészül. A nyelv fejlesztésekor az is fontos szempont volt, hogy az elkészült programokat az operációs rendszertől függetlenül lehessen futtatni. A Java többé-kevésbé meg is felel ennek

⁵ Sun MycroSystems <http://hu.sun.com/>

az elvárásnak, mely szintén pozitívum. Természetesen szinte lehetetlen lenne olyan nyelvet írni, mely minden operációs rendszeren ugyanúgy működik, és az elkészült alkalmazás ugyanúgy le is fut, így a Java nyelvnek is szüksége volt egy közegre, melyre épülve már valóban teljesen univerzális lehet.

Ez a közeg a Java legfontosabb része, a Java Virtuális Gép (Java Virtual Machine – JVM). (Vég Csaba 2007 : 4. p.) Ezek a Java Virtuális Gépek fordítják le a forráskódot gépi kódra. A Java Virtuális Gép tehát minden gépen más, viszont használatának következménye minden esetben ugyanaz. Ez teszi lehetővé azt, hogy a forráskódot csak egyszer kelljen megírni, azt ugyanis minden gép ugyanúgy fogja lefuttatni a Java Virtuális Gépnek köszönhetően. Tehát ha megírunk egy Java programot egy Windows operációs rendszerrel rendelkező személyi számítógépen, az ugyanúgy fog majd lefutni egy mobiltelefonon, melynek szintén része a Java Virtuális Gép. Ez természetesen jelentős költségcsökkenést eredményez, hiszen nem kell a programot annyiszor megírni, ahány gépen csak alkalmazni akarjuk.

A Java nyelv fejlesztésekor fontos szempont volt továbbá a hálózati programozás elősegítése a megfelelő kódokkal és könyvtárakkal, és legalább ennyire koncentráltak arra is, hogy a távoli gépeken is képes legyen biztonságosan futni. (Daniel J. Berg, J. Steven Fritzinger 1999 : 3. p.)

A Java ezen önmaga elé kitűzött céloknak az évek során egyre inkább megfelelt, ezáltal a világ egyik legnépszerűbb programozási nyelvává válva. Az élet számos területén használták, és használják is mind a mai napig a Java alkalmazásokat, még a könyvtárakban is. Legismertebb területe a platformfüggetlenségéből adódóan mégis a mobiltelefonoké lett, melyeken leggyakrabban a mobiltelefonos játékok elindításakor láthatjuk viszont a Java jellegzetes csészéjét.

A számítógépes játékok története

„A játék nem komoly, nem 'vére megy', a nem igazság tartozik hozzá, ez valami jó, van értelme, hozzájárul az emberi léthez. Sokan értelmetlennek, nyereséggel nem járónak tekintik. Ezt szebben megfogalmazva: Öncélúnak nevezhetjük. Része a színlelés (például maga a szerepjáték), lehet tétje, valamint minden játék mutat valamilyen szabálykört, célja a boldoggá tétel is. Valójában még a legöncélúbbnak, legcéltalanabbnak tartott gyermekjátéknak is van haszna és célja, amint Jean Piaget és Susanna Millar rámutattak. Célja az exploráció, a kísérletezés és a felfedezés és ezáltal ismeretek és képességek megszerzése. Az emberiség számos nagy felfedezését a játéknak köszönheti.”⁶

A játékok tehát kétségtelenül jelentős szerepet töltenek be az emberek életében, éppen ezért célszerű a Java nyelvet egy olyan programon keresztül bemutatni, ami amellet, hogy felettebb reprezentatív, még szórakoztató is. A számítógép viszont - mint tudjuk - szinte határtalan lehetőségeket biztosít a játékfejlesztés szempontjából, egy-egy játék megalkotásának ugyanis manapság többnyire csak a képzeletünk szabhat határt. Anélkül viszont nehéz bármit is megtervezni, hogy egy minimális szinten ne ismerkednénk meg a játékok történelmével, és a lehetőségeinkkel.

Számítógépes játéktörténelemmel kapcsolatos cikket elég sokat találhatunk az interneten, melyek természetesen számtalan részletben különböznek egymástól a téma nem éppen tudományos mivoltának köszönhetően. A mára már legendássá vált legelső játékokról azonban mindenhol szó esik.

„Az egyetemeken kutató mérnök kifejlesztették az első számítógépes játékokat, illetve demonstrációkat abból a célból, hogy a nyílt napokon, bemutatókon valami érdekes dolgot tudjanak mutatni a látogatóknak. 1958-ban elkészítették a Tennis For Two nevű játékgépet.

Később egy kutató labor IBM nagygépein tűnt fel a Bouncing Ball nevű játék, mely végülis csak egy demonstráció volt, amiben egy labda pattogott a képernyőn. A programnak sikere volt, a látogatók órákig bámulták ahogy a labda pattog a képernyőn.

6 Játék (pszichológia) [http://hu.wikipedia.org/wiki/Játék_\(pszichológia\)](http://hu.wikipedia.org/wiki/Játék_(pszichológia))

Szintén demonstrációs program volt a Mouse in the Maze nevű program, mely valamennyire interaktív volt, a 'játékos' egy labirintust tervezhetett, sajtokat rakhatott le, melyet egy gép által irányított stilizált egér összeszedett.”⁷

A legelső valódi videojáték Ralph Baer 1968-as Chase Game című, tévékészülékre csatlakoztatható játéka volt, melyben két pont kergette egymást, két-két tekergethető gomb segítségével.⁷ Baer-t hamarosan leszerződtette a Magnavox, 1972-ben pedig kiadták a játék finomított verzióját is tartalmazó Odyssey című játékgyűjteményt, melyhez játékonként egy-egy fóliát is adtak. Ezeken a fóliákon volt a játékok háttere, melyek élvezhetőbbé tették a grafikaiilag még igen elmaradott játékokat. Szerencsére az akkori programozók sem elégedtek meg ennyivel, és mindent megtettek, hogy mind a grafika, mind a hardver terén egyre kiemelkedőbbet alkossanak. A játékipar pedig a kétdimenziós ábrázolási módból hamarosan kilépett a térbe. A John Carmack által fejlesztett Doom, és elődje a Wolfenstein 3D óriási mérföldkövet jelentett a játékvilágban, ám megjelenésük nem volt előzmény nélküli. A Doom-típusú játékok ugyanis csak abban voltak újítóak, hogy a 3D-s nézetben szabadon foroghatott a játékos, ám már előtte is voltak olyan játékok, melyek fix képekkel, és ezekre épülő programháttérrel azt az illúziót keltették, hogy a játékos valóban részese a játék világának, és a főhős szemén keresztül szemlélheti azt.

Az első igazán szépen kivitelezett, és nagy népszerűsége szert tett 3D megjelenítést utánozó játék a Dungeon Master⁸ volt, melynek elődjeként talán a csak 8 színből építkező Dungeons Of Daggorath⁹ című játékot tekinthetjük. A számítógépes- és egyéb újkori játéktörténelem során a dungeon szó olyan mély tartalmat kapott, melyet magyar nyelvre lefordítani mind a mai napig nem sikerült senkinek. A dungeon leginkább úgy írható körül, mint egy labirintus, melyet a játékos kalandozóként deríthet fel, számtalan életveszélyes csapdát, és ellenséget átvészelve, agyafűrt fejtörőket megoldva, és a továbbjutást megakadályozó ajtók kulcsait keresve¹⁰. A dungeonok igazi légköre akkor teremődött meg, mikor először megjelentek a monitorokon a sötétségbe vesző végtelen folyosók. A kezdetben csak és kizárólag a szürke kőfalak, a kövezett padló, és a lapos, jellegtelen plafon díszleteiből gazdálkodó dungeonok nem várt szintű, magával ragadó, szorongással teljes hangulatot

7 Videojátékok története <http://oli76.ingyenweb.hu/keret.cgi?/e04.htm>

8 Dungeon Master (video game) [http://en.wikipedia.org/wiki/Dungeon_Master_\(computer_game\)](http://en.wikipedia.org/wiki/Dungeon_Master_(computer_game))

9 Dungeons Of Daggorath http://en.wikipedia.org/wiki/Dungeons_of_Daggorath

10 RPG.hu http://rpg.hu/szotar/main.php?do_this=list_by_letter&letter=D

hordoztak magukban. A játékosok tudták, hogy csak egy játékkal állnak szemben, érzékeik mégis megcsalták őket, és agyuk egy részével elhitették, hogy mélyen a föld alatt, dohos levegőjű, zárt labirintusrendszerekben tapogatózva keresik a kijáratot, miközben mindenhol ismeretlen veszélyek leselkednek rájuk. (1. ábra)

Virtuális világok

Korábban a játékok minél többet próbáltak mutatni a játékos számára a virtuális térből, ám a Dungeon Master típusú játékok megjelenésével horrorba illő környezetet teremtettek azzal, hogy bár sokat látott a játékos, mégis csak a töredékét a veszélyekkel kecsegtető játéktérnek. Az addigi játékokban mindig a képernyő középpontja környékén volt a játékos avatarja, amire vigyáznia kellett, így akárhonnan érkezett a veszély, még volt ideje felkészülni ellene. Ám onnantól kezdve, hogy a játékos nem „kívülről látta magát”, úgynevezett harmadik személyű nézőpontból, megszűnt a felsőbbrendű „mindent látóság” érzése, és a játékos visszakényszerült a valódi első személyű nézőpontba, köszöntve a valóság egyik legkellemetlenebb velejáróját, azt, hogy hátul nincs szeme. Ettől kezdve a játékosok megtapasztalhatták – legalábbis virtuálisan – hogy milyen az, ha hátba támadják őket, átélhették a félelmet, az ijedtséget, a kiszolgáltatottság érzését.

A játékok innentől kezdve csakúgy, mint nem sokkal előttük a filmek, sajátos módszerrel segítettek az embereknek bizonyos szinten leküzdeni a félelmeiket. Később ezt az orvostudomány is felfedezte, és kísérleti kezeléseik részévé tette a számítógépes játékokat. A grafika fejlődésével olyan valós illúziót voltak képesek kelteni, melyek révén klausztrofóbiások, vagy agorafóbiások is képesek voltak a gyógyulás útjára lépni. A bezártságtól kórosan iszonyodó embereket olyan virtuális térbe helyezték, ahol szűk folyosók, és apró szobák várták őket, és hosszas játék után beleélve magukat a főhősük helyébe, megtanulták virtuálisan leküzdeni félelmeiket. Utána pedig, a valóságba visszakerülve sokkal jobban viselték a hasonló helyzeteket. Hasonlóan ehhez, a nagy terektől, és magasságtól rettegőket hasonló virtuális terekbe helyezték, és miközben megnyugvasként hatott rájuk, hogy bármikor kimenekülhetnek, lassan megbarátkoztak a környezettel. Betegségük pedig ez által enyhült.¹¹

¹¹ Gyógyítás virtuális valóságban http://www.magyarhirlap.hu/Archivum_cikk.php?cikk=90985&archiv=1&next=0

A látványvilág megtervezése

A Java programozási nyelv alapjait elsajátítva ugyan még igen korlátozottak a lehetőségeink, ám viszonylag egyszerűen, és nem több év, no meg egy egész játékfejlesztői gárda segítségével készíthetünk egy ahhoz hasonló játékot, mint a fentebb említett Dungeon Master, és hasonmásai.

A folyosó, mely a játék központi eleme az ábrázoló geometria végtelen távoli pontjának fogalmára alapozva négy, a végtelenben találkozó egyenesből áll csupán, mely legtöbbször a képernyő középpontján áthaladó, egymást metsző, két egyenes. Mivel ez a leggyakoribb kép, amit lát a játékos, ezért erre helyeződnek majd fel az azt módosító képrészletek, melyek az elágazásokat jelenítik majd meg, esetleg a zsákutcát a folyosó végén. (2. ábra)

Ahhoz, hogy egy elágazást megjelenítsünk, így nem is kell egy teljesen új képet betöltenünk. Elég, ha az alapképünkre kirajzoljuk az elágazás képét. Így ha a játékos egy elágazáshoz ér, elég csupán az elágazás képrészletét kirajzolnunk az alapként szolgáló folyosó képére, nem kell újrarajzoltatnunk mindent a számítógéppel. (3. ábra)

Kereszteződés esetén sem bonyolódik a feladat, csupán az előző kis kép tükörképét kell kirajzolnunk az adott helyre, és már el is hitettük a játékos, hogy egy kereszteződéshez ért.

A zsákutca megjelenítéséhez szintén elég egyetlen kicsi módosítást végrehajtanunk az alapképen. Elég, ha eltakarjuk a végtelenbe vezető folyosó képét egy egyszerű falat ábrázoló téglalappal. Ennek megjelenítése szintén nem igényel túl sok erőforrást, az eredmény mégis egyértelmű. A játékos rögtön tudni fogja, hogy innen már csak visszafelé vezet az út. (4. ábra)

Az eddigiek kombinálásával pedig egy egyszerű kanyart is megjeleníthetünk, ezzel tulajdonképpen végére is érve a lehetőségek vizsgálatának. Minden további kép ugyanis, ami a játékos elé tárulhat a labirintusban való bókászás közben, ebből a három kis képből felépíthető. (5. ábra) Egy kis matematikát alapul véve megállapíthatjuk ugyanis, hogy mindössze ezt a 3 kis képet variálva meg tudunk jeleníteni $2*2*2=8$ variációt. Ez pedig megegyezik azon lehetőségek számával, mellyel a játékos találkozhat a labirintusban. Ahhoz tehát, hogy a játék játszható legyen, már ennyi is elég.

Ám nem kell túl kritikusnak lennünk magunkkal szemben, hogy megállapítsuk, akármilyen izgalmas is a labirintusunk, ha csupán 8 kép váltakozásáról szól a játék, hamar

unalomba fulladhat. Ráadásul, ha az elágazásokat olyan távol jelenítjük meg egymástól, hogy a képen a végtelenbe veszve már nem is látszanak, még a tájékozódást is élvezhetetlenül megnehezítjük.

Éppen ezért célszerű továbbfejlesztenünk a megjelenítést úgy, hogy a játékos ne csak azt a helyet lássa – ne csak azt az elágazást – amelyikben éppen van, hanem azt is, hogy ha tovább halad előre, hova jut. Magyarán a programnak nem csak azt kell innentől vizsgálnia, hogy jelenleg merre haladhat tovább a játékos, hanem azt is, hogy ha egyenesen haladna tovább, addig, amíg el nem éri a folyosó végét, hol kanyarodhat még el.

Ha a program már ezt is képes vizsgálni, nem kell mást csinálnunk csak ugyanazt a képkirajzoló algoritmust használnunk, amit korábban, csak mindig egy fokkal kisebb változatban. (6. ábra)

Az elinduláshoz elegendő grafikus felület terve tehát már kész. Látható, hogy annak idején is hasonlóan egyszerű dolga volt a programozóknak, mint most nekünk. Annyi különbséggel, hogy nekik még ki is kellett találni ezt a módszert, számunkra pedig már készen van tálalva, csak fel kell használnunk.

Main.java

Ha már objektum-orientált programozásról van szó, egyértelmű, hogy készülő játékunk nem állhat egyetlen programból. Ehelyett egy főprogramot, azaz főosztályt, és az általa meghívott segédosztályokat fogjuk elkészíteni. A főosztályunk neve legyen egyszerűen csak Main.java, mely a játék csomagban fog elhelyezkedni és amellet, hogy importálja majd a Java szabványos csomagjait – az awt, az awt.event, és az applet csomagokat – egy általunk készített csomagot is importálni fog, annak minden osztályával. Ez az általunk meghatározott másik csomag a szereplo nevű csomag lesz, melyet a későbbiekben még bővebben is fogunk tárgyalni.

```
package jatek;  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
import szereplo.*;  
public class Main extends Applet implements KeyListener {
```

Mivel webes környezetben futtatható játékot szeretnénk készíteni, a Main.java osztályt az Applet osztály leszármazottjává kell tennünk. Az öröklődés szabályainak köszönhetően ettől kezdve már meghívhatjuk az Applet osztály mezőit és metódusait, a fentebb importált osztályoknak hála pedig még több lehetőségünk nyílik egy igen kellemes applet elkészítésére. A KeyListener osztály implementálására, azaz megvalósítására pedig azért lesz szükségünk, mert billentyűzettel irányítható játékot szeretnénk készíteni. Tény, hogy az egérrel kapcsolatos eseményeket is le tudnánk kezelni, ám ha szem előtt tartjuk a Java platformfüggetlenségi elveit, számítanunk kell arra is, hogy készülő játékunkat minimális átalakítások után olyan gépeken szeretnénk majd futtatni, ahol esetleg még egér sincs (pl.: mobiltelefon), ezért programunkat a lehető legfrappánsabban, a lehető legkevesebb billentyű használatát igénylő módon készítjük el, az egér teljes száműzésével.

Főosztályunk első soraiban annak mezőit deklaráljuk, melyeket a későbbiekben többször is felhasználunk majd. Programunk érthetősége, és saját munkánk megkönnyítése érdekében bizonyos konstansokat is bevezetünk, és olyan névvel illetjük őket, melyekre

később hivatkozva egyértelműen meg tudjuk majd állapítani, hogy az egyes számokhoz milyen fogalmakat rendeltünk. Az irányokhoz, a menüpontokhoz, valamint a játék főbb „helyszíneihez” azért rendelünk szimplán számokat, hogy azokkal később könnyedén tudjunk műveleteket elvégezni. Így, ha mondjuk a játékos a menüben a „le gomb” megnyomásával a következő menüpontot akarja kijelölni, nem kell tudnunk, hogy épp melyik menüpont van kijelölve, csupán növelnünk kell eggyel az épp kijelölt menüpont sorszámát. A későbbiekben viszont mégsem kell egy rejtett jelentésű számra hivatkoznunk, hiszen minden egyes számhoz hozzá van rendelve a jelentése, a funkciója.

```
public static final int BALRA=0, FELFELE=1, JOBBRA=2, LEFELE=3, BAL=37,
FEL=38, JOBB=39, LE=40, BELEPES=0, TERKEP=1, ZENE=2, MERET=3, MENU=0,
LENT=1, JATEK=2, FENT=3;

public static int jelenlegiKisterkep,jelenlegiMenupont,jelenlegiMeret,hely,szint,nagyitas;
public static int[] meret=new int[3];
public static boolean zeneVan;
public static Image menu,folyoso,zene,csend,lent,fent;
public static Image[] menupont=new Image[4],kisterkep=new Image[3],kepmeret=new
Image[3],lejaro=new Image[5],balkanyar=new Image[5],jobbanyar=new
Image[5],fal=new Image[5],feljaro=new Image[5],kijarat=new Image[5];
public static AudioClip jelenlegiZene,jatekzene,menuzene,szintlepes,vege;
public static Utveszto utveszto;
```

Egy applet betöltődésekor az első metódus, ami végrehajtódik, az `init()`. Ezt a metódust felülírva érhetjük el, hogy rögtön kezdje el betölteni a böngésző az általunk később használni kívánt képeket, hangokat, és az egyes mezők kezdőértékeit. Ismét egyszerűsítve feladatunk, a játék során felhasználni kívánt képeket és hangokat, ahol lehet sorszámozzuk, így elegendő egy-egy ciklusban megadnunk a betöltési utasításokat egyetlen változó sorszámmal.

```
public void init() {
    /* képek betöltése */
    csend=getImage(getCodeBase(),"csend.PNG");
    fent=getImage(getCodeBase(),"fent.PNG");
    folyoso=getImage(getCodeBase(),"folyoso.PNG");
    menu=getImage(getCodeBase(),"menu.PNG");
```

```

lent=getImage(getCodeBase(),"lent.PNG");
zene=getImage(getCodeBase(),"zene.PNG");
for(int i=0;i<3;i++) {
    kepmeret[i]=getImage(getCodeBase(),"kepmeret"+i+".PNG");
    kisterkep[i]=getImage(getCodeBase(),"kisterkep"+i+".PNG");
}
for(int i=0;i<4;i++) {
    menupont[i]=getImage(getCodeBase(),"menupont"+i+".PNG");
}
for(int i=0;i<5;i++) {
    balkanyar[i]=getImage(getCodeBase(),"balkanyar"+i+".PNG");
    fal[i]=getImage(getCodeBase(),"fal"+i+".PNG");
    jobbkanyar[i]=getImage(getCodeBase(),"jobbanyar"+i+".PNG");
    feljaro[i]=getImage(getCodeBase(),"feljaro"+i+".PNG");
    kijarat[i]=getImage(getCodeBase(),"kijarat"+i+".PNG");
    lejaro[i]=getImage(getCodeBase(),"lejaro"+i+".PNG");
}
/* zenék betöltése */
jatekzene=getAudioClip(getCodeBase(),"zene"+(int)(Math.random()*19)+".wav");
menuzene=getAudioClip(getCodeBase(),"menu.wav");
szintlepes=getAudioClip(getCodeBase(),"szintlepes.wav");
vege=getAudioClip(getCodeBase(),"vege.wav");
/* kezdőértékkadás */
meret[0]=400;
meret[1]=640;
meret[2]=800;
hely=MENU;
szint=3;
nagyitas=160/(szint*2+3);
zeneVan=true;
jelenlegiMenupont=0;

```

```

        jelenlegiMeret=2;
        jelenlegiKisterkep=0;
        jelenlegiZene=menuzene;
        jelenlegiZene.loop();
        this.addKeyListener(this);
    }
    public void stop() {
        jelenlegiZene.stop();
    }

```

A mezők kezdőértékeinek meghatározásakor megadjuk, hogy a játékban mekkora képernyőméreteket lehessen beállítani, hogy a játék a menüvel kezdődjön, valamint hogy az első útvesztőben 3*3 csomópont legyen. Beállítjuk továbbá, hogy az aktuális útvesztő kis térképe a bal felső sarokban maximum 160 képpont legyen, és mindig az útvesztő nagyságával arányosan csökkenjen a folyosórészletek megjelenítésének nagysága. Ez utóbbival azt érjük el, hogy bármekkora is legyen az aktuális útvesztő, a térképe mindig megközelítőleg ugyanakkora lesz, így nem fogja soha kitakarni a játéktér zavaróan nagy részét. Ezt követően megadjuk, hogy a játék átlátható jellemzőinek melyek legyenek a kezdőértékei, végül elindítjuk a zenét, és elkezdjük figyelteni a billentyűlenyomásokat. A stop() metódust csak a biztonság kedvéért írjuk felül. A legtöbb böngésző ugyanis külön utasítás nélkül is leállítja a zenét, ha elnavigálunk az oldalról, de gondolva az erre nem képes böngészőkre, nem árt, ha ezt is lekezeljük.

Az applet betöltődése után először a paint(Graphics g) metódusra lesz szükségünk. Ez fogja majd megjeleníteni a háttérben végrehajtódó programjaink eredményét. Ám mivel játékunk több jól elkülöníthető részre fog tagolódni, ezért magát a paint(Graphics g) metódust is tagolnunk kell majd, az egyes részleteknek megfelelően.

Mivel a játékos először a menüvel találkozik, majd az átvezető képpel, aztán az útvesztővel, végül pedig a győzelem képével, ezért a megjelenítést is ebben a sorrendben tárgyaljuk majd.

```

    public void paint(Graphics g) {
        setBackground(Color.BLACK);
        switch(hely) {

```

```

case MENU:
    /* menü kirajzolása */
    g.drawImage(menu, (800-meret[jelenlegiMeret])/2, 0, meret[jelenlegiMeret],
3*meret[jelenlegiMeret]/4, this);
    g.drawImage(menupont[jelenlegiMenupont], (800-meret[jelenlegiMeret])/2, 0,
meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
    g.drawImage(kisterkep[jelenlegiKisterkep], (800-meret[jelenlegiMeret])/2, 0,
meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
    g.drawImage(kepmeret[jelenlegiMeret], (800-meret[jelenlegiMeret])/2, 0,
meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
    if(zeneVan) g.drawImage(zene, (800-meret[jelenlegiMeret])/2, 0,
meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this); else g.drawImage(csend, (800-
meret[jelenlegiMeret])/2, 0, meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
    break;

```

A különböző helyszínek kirajzolásáért felelős programrészletek elkülönítésére a többirányú elágazások bevezetése a kézen fekvő megoldás. A menühöz rendelt konstans szám nevesítésének köszönhetően pedig magában a kódban sem annak kell megjelennie, hogy a 0. helyszínen vagyunk, a gép ugyanis már jól tudja, hogy az a MENU nevű helyszínnel egyenlő. Az egyes képeket az egész játék során úgy rajzoljuk ki, hogy mind a méretük, mind a kirajzolásuk helye változtatható legyen a menüből. S mivel a képeket csoportosítottuk, és sorszámoztuk, így ismét elegendő csupán a jelenlegi sorszámokra hivatkoznunk, bármik is legyenek azok.

A képek kirajolásában egy igen frappáns trükköt is alkalmazhatunk. Az alapelv minden esetben az lesz, hogy egy statikus, teljes képernyőt betöltő képre illesztjük majd rá a változó képrészleteket. A menü háttere például mindig ugyanaz marad, csak az egyes gombok megjelenítéséért felelős képrészletek változnak majd mindig. Ha a böngészők, vagy a Java esetleg nem kezelné az átlátszó képek fogalmát, akkor ezeket a változó képrészleteket úgy kellene kirajzoltatnunk, hogy előtte megkeressük a kép leendő helyének koordinátáit. Ez a menüben még nem is lenne olyan nagy feladat, – bár az, hogy a képernyő mérete változtatható, még ezt is megnehezítené – ám az útvesztő egyes elágazásainak kirajzoltatása már kellemetlen mértékben megnehezedne számunkra. Szerencsénkre azonban gif-ekkel, sőt

png-kkel is dolgozhatunk, így egész egyszerűen elkészíthetjük a kis képek teljes méretű változatait, melyeken a szükségtelen részeket átlátszóvá tehetjük. Ezeket a több helyen átlátszó képeket aztán egymásra illeszthetjük, és nem is kell bajlódni a pozicionálással, ugyanis minden kis képrészlet helyét már a kép megszerkesztésekor meghatározhatjuk. (7. ábra)

case LENT:

```
g.drawImage(lent, (800-meret[jelenlegiMeret])/2, 0, meret[jelenlegiMeret],  
3*meret[jelenlegiMeret]/4, this);  
break;
```

Mivel az átvezető képnek nincsen egyetlen változó része sem, ezért a kirajzolására is elég egyetlen utasítás. Maga a játék képeinek kirajzolása viszont összetettsége miatt további tagolást is igényel. Külön tárgyaljuk tehát a folyosók 3D-hatású megjelenítését, és a 2D-s felülnézeti térkép kirajzolását.

case JATEK:

```
/* folyosó kirajzolása */  
g.drawImage(folyoso, (800-meret[jelenlegiMeret])/2, 0, meret[jelenlegiMeret],  
3*meret[jelenlegiMeret]/4, this);  
Falnezo latas=new Falnezo(utveszto.jatekos);  
if(jelenlegiKisterkep==1) utveszto.terkep[latas.getX()]  
[latas.getY()]=Utveszto.LATOTT;
```

Először a folyosó képét rajzoljuk ki, és erre helyezzük majd rá az elágazások, és a folyosó végét jelző fal, valamint a szint végét jelző, felfelé haladó létra képét.

Annak megállapítására, hogy a játékos éppen mit láthat, egy külön objektumot hozunk létre, a Falnezo típusú latast. Ennek az objektumnak az osztálya egyébként a szereplo csomagban található. Mint ahogy arra a későbbiekben is ki fogunk térni, a Falnezo osztály konstruktora egy olyan függvény, melynek paramétere szintén a szereplo csomag egyik osztálya, a SzilardMozgoSzereplo osztály. Emiatt lehet az Utveszto típusú utveszto objektum SzilardMozgoSzereplo típusú jatekos objektumát paraméterként megadni a Falnezo osztály konstruktorában. Maga a konstruktorfüggvény így tulajdonképpen mindössze létrehozza a jatekos egy azonos pozícióval és iránnyal rendelkező másolatát, ami azonban már nem SzilardMozgoSzereplo, hanem Falnezo típusú lesz. Rögtön utána pedig felderítetté teszi a

latas, azaz jelenleg a játékos pozícióját, hogy ha a játékos felderítendő térképpel játszik, akkor ezt a rész is lássa kezdettől fogva. A játékoshoz hasonlóan az utveszto is a terkep objektuma, a LATOTT pedig a konstansa.

```

while(latas.eloreSzabad(utveszto.terkep) && latas.getLepes()<5) {
    if(latas.balraSzabad(utveszto.terkep)) {
        g.drawImage(balkanyar[latas.getLepes()], (800-meret[jelenlegiMeret])/2, 0,
meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
        if(jelenlegiKisterkep==1)
utveszto.terkep[latas.baloldal(utveszto.terkep).getX()]
[latas.baloldal(utveszto.terkep).getY()]=Utveszto.LATOTT;
    }
    if(latas.jobbraSzabad(utveszto.terkep)) {
        g.drawImage(jobbkanyar[latas.getLepes()], (800-meret[jelenlegiMeret])/2, 0,
meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
        if(jelenlegiKisterkep==1)
utveszto.terkep[latas.jobboldal(utveszto.terkep).getX()]
[latas.jobboldal(utveszto.terkep).getY()]=Utveszto.LATOTT;
    }
    latas.elore(utveszto.terkep);
    if(jelenlegiKisterkep==1) utveszto.terkep[latas.getX()]
[latas.getY()]=Utveszto.LATOTT;
}

```

Miután létrehoztuk a latast, egy ciklus segítségével elindítjuk őt előre, hogy felderítse, mit láthat maga a játékos. Csak addig küldjük, míg el nem éri a folyosó végét. Ha viszont őt lépésen belül nem éri el, a folyosó vége már úgyis túl messze van ahhoz, hogy kirajzoltassuk, így tovább már nem is küldjük. Minden egyes lépés előtt a latas meghívva saját metódusait ellenőrzi, hogy a baloldalán és a jobboldalán van-e elágazás, s ha van, nem csupán kirajzolja a megfelelő képet, de még a felderítendő térkép számára is megjelöli azt látottként. A kirajzolandó képek sorszáma, azaz a rajtuk lévő képrészlet mérete attól függ, hogy hány lépés távolságra van az elágazás, vagy épp a folyosó vége a játékostól, tehát hogy a latas nevű objektum hány lépést tett meg azóta, hogy a játékos másolataként létrejött.

```

        if(latas.getLepes()<5) {
            if(latas.balraSzabad(utveszto.terkep)) {
                g.drawImage(balkanyar[latas.getLepes()], (800-meret[jelenlegiMeret])/2, 0,
meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
                if(jelenlegiKisterkep==1)
utveszto.terkep[latas.baloldal(utveszto.terkep).getX()]
[latas.baloldal(utveszto.terkep).getY()]=Utveszto.LATOTT;
            }
            if(latas.jobbraSzabad(utveszto.terkep)) {
                g.drawImage(jobbkanyar[latas.getLepes()], (800-meret[jelenlegiMeret])/2, 0,
meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
                if(jelenlegiKisterkep==1)
utveszto.terkep[latas.jobboldal(utveszto.terkep).getX()]
[latas.jobboldal(utveszto.terkep).getY()]=Utveszto.LATOTT;
            }
            g.drawImage(fal[latas.getLepes()], (800-meret[jelenlegiMeret])/2, 0,
meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
            if(latas.erint(utveszto.start)) g.drawImage(lejaro[latas.getLepes()], (800-
meret[jelenlegiMeret])/2, 0, meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
            if(latas.erint(utveszto.cel))
                if(szint==13) g.drawImage(kijarat[latas.getLepes()], (800-
meret[jelenlegiMeret])/2, 0, meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
                else g.drawImage(feljaro[latas.getLepes()], (800-meret[jelenlegiMeret])/2, 0,
meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
            if(latas.balraSzabad(utveszto.terkep)) g.drawImage(balkanyar[latas.getLepes()],
(800-meret[jelenlegiMeret])/2, 0, meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
            if(latas.jobbraSzabad(utveszto.terkep))
g.drawImage(jobbkanyar[latas.getLepes()], (800-meret[jelenlegiMeret])/2, 0,
meret[jelenlegiMeret], 3*meret[jelenlegiMeret]/4, this);
        }

```

Miután kirajzoltattuk az elágazások képeit a számítógéppel, a folyosó végének kirajzolása következik. Egy folyosó négyféleképpen érhet véget, s ezért mind a négy esetet le kell kezeltetnünk. Ha a folyosó vége egyszerűen csak zsákutcába torkollik, nem kell mást tennünk, mint kirajzoltatnunk a falat a megfelelő méretben. Ha azonban egy felfelé, vagy egy lefelé haladó létrát kell látnia a játékosnak, azt is külön ki kell rajzolnunk. Ráadásul, ha az utolsó szintet is elérte a játékos, még a szabadba vezető létra képe is más lesz. (8. ábra) Itt jegyzendő meg az is, hogy a cikluson belül csak a valódi elágazásokat rajzoltattuk ki, a folyosó végén lévő kanyarokat nem. Ezért a ciklusban lévő, oldalakat figyelő programrészletet meg kell ismételtnünk. Ezeket az eseteket – mint láthattuk – egyszerű feltételes elágazásokkal le lehet kezelni, a játékos pedig már csak a látványos eredményben gyönyörködhet majd, a folyosó és a jó néhány ráillesztett kis kép kombinációjában.

A felülnézeti térkép kirajzolásához már nem előre elkészített képeket használunk majd, hanem a Java beépített rajzoló osztályait.

```
/* térkép kirajzolása */
if(jelenlegiKisterkep!=2) {
    for(int i=0; i<utveszto.meret; i++) for(int j=0; j<utveszto.meret; j++) {
        if(jelenlegiKisterkep==0) {
            switch (utveszto.terkep[i][j]) {
                case Utveszto.FAL: g.setColor(Color.BLACK); break;
                case Utveszto.KERET: g.setColor(Color.DARK_GRAY); break;
                case Utveszto.FOLYOSO: g.setColor(Color.GRAY); break;
                case Utveszto.START: g.setColor(Color.RED); break;
                case Utveszto.CEL: g.setColor(Color.GREEN); break;
            }
        }
        if(jelenlegiKisterkep==1) {
            if(utveszto.terkep[i][j]==Utveszto.LATOTT) g.setColor(Color.GRAY);
            else g.setColor(Color.BLACK);
        }
        g.fillRect(i*nagyitas, j*nagyitas, nagyitas, nagyitas);
    }
}
```

Térképet természetesen csak akkor rajzolunk ki, ha a játékos nem állította be külön, hogy ő térkép nélkül akar játszani. Mivel az utveszto objektumban már szerepel az, hogy a 2 dimenziós, egész számokat tartalmazó tömbként megjelenő térkép egyes részein folyosó, vagy fal van, egyszerűen csak végigfuttatunk a tömbön két egymásba ágyazott ciklust, mely minden egyes térképrészleten ellenőrzi, hogy folyosórészt, vagy falat kell-e kirajzolnia. Ha a játékos eleve felderített térképet szeretne, külön színnel jelöljük a kezdőpontot, és a végpontot is. Ha viszont maga szeretné felderíteni a térképet, elég, ha csak a látott részeket rajzoltatjuk ki. A térkép minden részletére egy kis teli négyzetet fogunk majd kirajzolni, a feltételek csupán azt határozzák meg, hogy milyen színnel.

A játékos kirajzolására már nem négyzetet használunk, hanem egy kis háromszöget, melynek koordinátáit külön tömbökben tároljuk. A háromszög csúcsa jelöli majd, hogy a játékos épp merre néz. Ez mind a négy irány esetén ugyanabban a pontban lesz, csupán a háromszög másik két csúcsának koordinátái fognak változni.

```
/* játékos kirajzolása a térképre */  
g.setColor(Color.YELLOW);  
int[] xek=new int[3];  
int[] yok=new int[3];  
xek[0]=utveszto.jatekos.getX()*nagyitas+nagyitas/2;  
yok[0]=utveszto.jatekos.getY()*nagyitas+nagyitas/2;  
switch(utveszto.jatekos.getIrany()) {  
    case BALRA:  
        xek[1]=(utveszto.jatekos.getX()+1)*nagyitas;  
        xek[2]=(utveszto.jatekos.getX()+1)*nagyitas;  
        yok[1]=utveszto.jatekos.getY()*nagyitas;  
        yok[2]=(utveszto.jatekos.getY()+1)*nagyitas;  
        break;  
    case FELFELE:  
        xek[1]=utveszto.jatekos.getX()*nagyitas;  
        xek[2]=(utveszto.jatekos.getX()+1)*nagyitas;  
        yok[1]=(utveszto.jatekos.getY()+1)*nagyitas;  
        yok[2]=(utveszto.jatekos.getY()+1)*nagyitas;
```

```

        break;
    case JOBBRA:
        xek[1]=utveszto.jatekos.getX()*nagyitas;
        xek[2]=utveszto.jatekos.getX()*nagyitas;
        yok[1]=utveszto.jatekos.getY()*nagyitas;
        yok[2]=(utveszto.jatekos.getY()+1)*nagyitas;
        break;
    case LEFELE:
        xek[1]=utveszto.jatekos.getX()*nagyitas;
        xek[2]=(utveszto.jatekos.getX()+1)*nagyitas;
        yok[1]=utveszto.jatekos.getY()*nagyitas;
        yok[2]=utveszto.jatekos.getY()*nagyitas;
        break;
    }
    g.fillPolygon(xek, yok, 3);
}
break;

```

Az első három helyszín kirajzolása ezzel elintéztünk is tekinthető, csupán az utolsó helyszín maradt még hátra, a győzelmi kép betöltése. Ennek kirajzolása ugyanolyan egyszerű, mint az átvezető képé, hiszen ezen sem változik semmi.

```

    case FENT:
        g.drawImage(fent, (800-meret[jelenlegiMeret])/2, 0, meret[jelenlegiMeret],
3*meret[jelenlegiMeret]/4, this);
        break;
    }
}

```

A `paint(Graphics g)` metódust követően a billentyűkezelés következik, melyet érthető okokból szintén helyszínenként tárgyalunk. A `KeyListener` interfész metódusai közül csak a `keyPressed(KeyEvent e)` metódust használjuk majd, ám mivel implementáljuk magát az interfészt, ezért a másik két metódusát is meg kell írunk, még ha üresen hagyjuk, akkor is.

```

public void keyTyped(KeyEvent e) {}
public void keyPressed(KeyEvent e) {
    switch(hely) {
        case MENU:
            switch (e.getKeyCode()) {
                case BAL: switch(jelenlegiMenupont) {
                    case TERKEP: jelenlegiKisterkep=(jelenlegiKisterkep+2)%3; break;
                    case ZENE: if(zeneVan) {
                        zeneVan=false;
                        jelenlegiZene.stop();
                    }
                    else {
                        zeneVan=true;
                        jelenlegiZene.loop();
                    } break;
                case MERET:
                    jelenlegiMeret=(jelenlegiMeret+2)%3;
                    break;
            } break;
    }
}

```

A billentyűlenyomásokkal kapcsolatos szerteágazó lehetőségek lekezelésének egyik módja egy háromszorosan egymásba ágyazott switch-case többirányú elágazás. Ezek közül a külső helyszínenként csoportosít, a középső billentyűnként, a belső pedig menüpontonként.

Az első esetben a menüben lenyomott bal gomb esetén fellépő eseményeket írjuk meg attól függően, hogy a játékos épp melyik menüpontot jelölte ki. Ha a térképbeállításokon áll, bal gomb hatására az aktuális beállítás előtti lehetőségre állítjuk át a térképet. S mivel a játékosnak lehetővé tesszük, hogy az 0. sorszámú beállítás esetén bal gombot lenyomva a 2. sorszámúra válthasson, ezért a műveletet hármassal maradékos osztással egészítjük ki. A maradékos osztás azonban a -1-et nem alakítja át 2-vé, ezért nem kivonunk egyet az aktuális sorszámból, hanem hozzáadunk kettőt. Így már minden esetben helyesen váltja majd át a beállítást a program. A ZENE menüpont esetén már egyszerűbb a dolgunk, csupán

ellentettjére kell változtatnunk a jelenlegi beállítást. A megjelenítés méretével kapcsolatos beállítások esetében pedig ugyanúgy járunk el, mint a térképénél.

A fel nyíl megnyomása esetén egész egyszerűen csak csökkentjük a kiválasztott menüpont sorszámát. Az elv ugyanaz, mint a térkép, vagy a képméret esetén.

```
case FEL: jelenlegiMenupont=(jelenlegiMenupont+3)%4; break;
```

A jobb gombhoz rendelt események épp ellentettjei a bal gombhoz rendelteknek, annyi különbséggel, hogy ha a játékos a belépés menüpontra nyom egy jobb gombot, elindul a játék, azaz változik a helyszín, és a zene, valamint létrejön az utveszto nevű objektum. A biztonság kedvéért egy kivételkezelést is elhelyezhetünk a programban, ami újrapróbálkozik az utvesztő létrehozásával, ha a rengeteg véletlen tényező miatt első alkalommal nem jön létre az objektum.

```
case JOBB: switch(jelenlegiMenupont) {  
case BELEPES: {  
hely=LENT;  
if(zeneVan) {  
    jelenlegiZene.stop();  
    jelenlegiZene=jatekzene;  
    jelenlegiZene.loop();  
}  
try { utveszto=new Utveszto(jelenlegiKisterkep,szint); } catch (Exception ex)  
{ utveszto=new Utveszto(jelenlegiKisterkep,szint); }  
} break;  
case TERKEP: jelenlegiKisterkep=(jelenlegiKisterkep+1)%3; break;  
case ZENE: if(zeneVan) {  
    zeneVan=false;  
    jelenlegiZene.stop();  
}  
else {  
    zeneVan=true;  
    jelenlegiZene.loop();  
} break;
```

```

        case MERET:
            jelenlegiMeret=(jelenlegiMeret+1)%3;
            break;
    } break;

```

A le nyíl megnyomásának kezelése pedig természetesen a fel nyíl lenyomása által kiváltott események ellentettjeihez fűződik majd.

```

        case LE: jelenlegiMenupont=(jelenlegiMenupont+1)%4; break;
    }
    break;

```

Mivel a játék irányítását a négy iránybillentyű használatára szűkítettük, az első helyszínen eseménykezelését ezzel be is fejeztük. A második helyszínről már jóval kevesebb eseményt kötünk majd, és eleve csak két iránygombot veszünk majd figyelembe ezzel is rávezetve a játékost a játék irányításának módjára.

```

case LENT:
    switch (e.getKeyCode()) {
        case BAL:
            hely=MENU;
            if(zeneVan) {
                jelenlegiZene.stop();
                jelenlegiZene=menuzene;
                jelenlegiZene.loop();
            }
            break;
        case FEL: hely=JATEK; break;
    }
    break;

```

A bal gomb lenyomásakor egyszerűen visszaküldjük a játékost a menübe, a fel gomb megnyomása esetén pedig betöltjük neki magát a játékot. Mivel a játékzenét már az átvezető képernyőn elkezdjük bejátszani, ezért a menübe visszatéréskor gondoskodnunk kell a zene visszaállításáról is. Más teendőnk azonban itt nem akad.

case JATEK:

```
switch (e.getKeyCode()) {  
    case BAL: utveszto.jatekos.balra(); break;  
    case FEL: utveszto.jatekos.elore(utveszto.terkep); break;  
    case JOBB: utveszto.jatekos.jobbra(); break;  
    case LE: utveszto.jatekos.hatra(utveszto.terkep); break;  
}
```

A játék közbeni billentyűkezelés igen bonyolult lenne, ha nem alkalmaznánk segédosztályokat. Így viszont egész egyszerűen megmondhatjuk a játékost jelképező objektumnak, hogy az adott billentyű lenyomásakor hogyan reagáljon. Az objektum saját metódusaiba pedig már bele vannak építve a különböző lehetőségeket lekezelő programrészek, így azokkal már nem is kell törődnünk.

Amit azonban még mindenképp ellenőriznünk kell, hogy az esetleges helyváltoztatás után a játékos elért-e egy olyan helyet, melyhez mi külön eseményeket szeretnénk rendelni. A játékos objektumának beépített metódusai ebben is a segítségünkre van.

```
if(utveszto.jatekos.erint(utveszto.cel)) {  
    if(szint==13) {  
        hely=FENT;  
        szint=3;  
        nagyitas=160/(szint*2+3);  
        if(zeneVan) {  
            jelenlegiZene.stop();  
            jelenlegiZene=vege;  
            jelenlegiZene.loop();  
        }  
    }  
}
```

Először is ellenőrizzük, hogy a játékos elérte-e a célt, és ha igen, akkor az a kijárat-e, vagy csak szimplán egy következő szintre juttató létra. Ha a kijáratra lelt rá a játékos a tíz szint teljesítése után, jutalmul betöltjük neki a győzelmi helyszínt, és zenét, valamint beállítjuk a következő játék alapbeállításait.

```

else {
    szint++;
    nagyitas=160/(szint*2+3);
    try { utveszto=new Utveszto(jelenlegiKisterkep,szint); } catch (Exception ex)
{ utveszto=new Utveszto(jelenlegiKisterkep,szint); }
    if(zeneVan) szintlepes.play();
}
}

```

Ha azonban csupán egy alsóbb szint feljárójára lett rá, növeljük az aktuális szint sorszámát, a térkép kirajzolásának paramétereit, majd létrehozuk az útvesztő egy új példányát, elővigyázatosságból kivételkezeléssel is kiegészítve. Adalékként pedig bejátszunk egy kis dallamot, hogy valamiképpen mégiscsak honoráljuk a játékos teljesítményét.

Előfordulhat, hogy a játékosnak nem felelnek meg az adott beállítások, és szeretne visszatérni a menübe, hogy megváltoztassa azokat. Nyilván megvan az a lehetősége, hogy újra betöltse az appletet tartalmazó oldalt, így a játékot is, de nem árt, ha biztosítunk egy alternatív módszert is. A legegyszerűbb megoldás az lenne, ha a `esc` billentyű lenyomása esetén a játékos rögtön visszatérne a menübe, ám ha már egyszer fáradoztunk az átvezető kép elkészítésével, célszerűbb egy a játékba jobban illeszkedő megoldást keresni. Már csak azért is, hogy a „billentyűminimalizáló” törekvéseink ne vesszenek kárba.

Játékunk környezete úgy készült el, hogy a játékos a menüt is a játék szerves részének érezze. Gyakorlatilag ugyanis a játékosnak ki kell másznia a menüt tartalmazó kis „gödörből” a játék elején. Így kézenfekvő tehát, hogy a menübe visszatérés a kezdőponton megjelenített létrán való lemászással történjen. A menübe visszatérés ezzel ugyanolyan egyszerű lesz, – legalábbis programozás szempontjából – mint a következő szintre való átlépés.

```

if(utveszto.jatekos.erint(utveszto.start)) {
    hely=MENU;
    szint=3;
    nagyitas=160/(szint*2+3);
    if(zeneVan) {
        jelenlegiZene.stop();
        jelenlegiZene=menuzene;
    }
}

```

```

        jelenlegiZene.loop();
    }
}
break;

```

Különbség csupán a helyszínben lesz, és a bejátszandó zenében.

Az utolsó helyszín lekezelése még az átvezető képnél is egyszerűbb, ugyanis bármelyik gombot is nyomja majd meg a játékos, mindenképp visszatér majd a menübe.

```

case FENT:
    hely=MENU;
    if(zeneVan) {
        jelenlegiZene.stop();
        jelenlegiZene=menuzene;
        jelenlegiZene.loop();
    }
    break;

```

A billentyűkhöz fűződő eseményeket ezzel az egész játékra vonatkozóan meghatároztuk, már csak a megjelenítő kép újrarajzolása van hátra, mely a megváltozott viszonyok miatt elengedhetetlen.

```

    }
    repaint();
}
public void keyReleased(KeyEvent e) {}
}

```

Az üres keyReleased(KeyEvent e) metódus megírásával pedig végére is értünk a Main.java osztálynak, melynek metódusai jó főnökként gondoskodnak arról, hogy a játékunk megfelelően működjön. Ám segédosztályok nélkül még ez az osztály sem ér semmit, így következzen is a játék csomag másik létfontossága osztálya, az Utveszto.java.

Utveszto.java

Az Utveszto.java a Main.java osztállyal ellentétben már egy önálló osztály lesz. Nem szükséges ugyanis, hogy leszármazottja legyen az Applet-, vagy bármelyik más osztálynak. Helye, mint már említettük a jatek csomagban van, és importálja a szereplo csomag osztályait, hogy azokkal dolgozni tudjon.

```
package jatek;
```

```
import szereplo.*;
```

```
public class Utveszto {
```

A főosztályhoz hasonlóan itt is nevesítünk néhány konstanst, melyeket később kívánunk majd felhasználni. Ebben az esetben a konstansok az útvesztő „tervjazának” részleteit írják majd le, azt, hogy az egyes helyeket a programnak hogyan kell kezelnie.

```
public static final int FAL=0, KERET=1, FOLYOSO=2, LATOTT=3, START=4, CEL=5;
```

```
public int meret;
```

```
public int[][] terkep=new int[100][100];
```

```
public Szereplo start;
```

```
public Cel cel;
```

```
public SzilardMozgoSzereplo jatekos;
```

A mezők deklarálásánál kiemelendő a terkep nevű kétdimenziós tömb, mely maga a már fentebb is említett „tervjaz”, játékunk gerince. Szintén fontos a három szereplo csomagbeli objektum, a start, a cel, és a jatekos, melyek játékunk legjelentősebb szereplői lesznek.

```
public Utveszto(int szint) {
```

```
meret=szint*2+3;
```

```
jatekos=new SzilardMozgoSzereplo(szint);
```

Az Utveszto osztály legfontosabb metódusa maga a konstruktor, melynek paramétere az útvesztő méretét állítja be. A paraméterben megadott szám azt mutatja, hogy az útvesztőben hány csomópont van, azaz hány olyan hely, ahol esetleg elágazás lehet. Mivel minden második lépés egy ilyen helyre vezet, ezért a játéktér méretét úgy kapjuk meg, hogy az adott számot megkétszerezzük és levonunk belőle egyet. Mivel azonban az útvesztőépítő algoritmus alapja az útvesztő szélén található „keret”, ezért mindkét oldalon plusz 2, tehát

összesen 4 helyet még biztosítanunk kell.

A méret beállítása után a játékost jelképező objektum létrehozása következik, melynek egyik konstruktora a véletlenszerű elhelyezést is lehetővé teszi. Ezt használjuk most ki.

```
/* keretezés */  
for(int i=0; i<meret; i++) for(int j=0; j<meret; j++) terkep[i][j]=KERET;  
for(int i=1; i<meret-1; i++) for(int j=1; j<meret-1; j++) terkep[i][j]=FAL;
```

A keretező programrészlettel kezdődik maga az útvesztő elkészítése. Az első ciklus beállítja a térkép határán található mezők értékeit, a második metódus pedig fallal tölti fel a belső részt.

A játékost tehát kezdetben gyakorlatilag befalazzuk.

Útvesztőnket úgy tervezzük meg, hogy minden egyes változata egy zsákutcából kezdődjön, és abban is érjen véget. Ehhez azonban biztosítanunk kell azt, hogy a folyosóépítő algoritmus véletlenül sem készít többirányú elágazást a játékos kezdőpontjában.

```
/* kiindulás */  
terkep[jatekos.getX()][jatekos.getY()]=START;  
start=new Szereplo(jatekos);  
MozgoSzereplo indito=new MozgoSzereplo(jatekos);  
indito.elore(terkep);  
terkep[indito.getX()][indito.getY()]=FOLYOSO;  
indito.elore(terkep);  
terkep[indito.getX()][indito.getY()]=FOLYOSO;  
jatekos.elore(terkep);
```

A legegyszerűbb megoldás a kiindulási zsákutca megszerkesztésére, ha – miután a játékos pozícióját megjelöljük START-ként, és elkészítjük annak objektumát – létrehozunk egy segédobjektumot, mely folyosót váj a játékos előtt lévő két falrészbe. Miután pedig a játékos útja már ki van jelölve, annak csak egyet kell előre lépnie, hogy mögötte a kiindulási ponthoz rendelt események ne zavarják meg a játék kezdetét. Maga az útvesztő építése pedig már a játékos előtti folyosórészről fog indulni.

Ahhoz viszont, hogy ez a módszer hibátlanul működjön, a jatekos konstruktorában a véletlenszerű elhelyezéskor nem szabad engednünk, hogy az objektum a játéktér szélén jöjjön létre. Ha ugyanis ott jelenik meg a játékos, és épp a keret felé néz, már az első lépése leviszi a

játéktérrel, és ezzel meghiúsul az egész útvesztőépítés. A probléma megoldásáról mégsem ezen a részen kell gondoskodnunk, elegendő, ha a játékos konstruktorában figyelünk oda erre.

Az útvesztőépítéssel kapcsolatban több elvárásunk is akad. Egyrészt, hogy önmagába visszatérő folyosók ne legyenek. Másrészt, hogy még így is minden lehetséges helyen legyen folyosó. Harmadrészt, hogy a cél a lehető legmesszebb legyen a kiindulási ponttól. Negyedrészt, hogy minden példányának más legyen a térképe.

Elvárásainknak úgy felelhet meg leginkább a labirintusunk, ha a konstruktorfüggvény minden alkalommal véletlenszerűen építi azt meg. Az építés módszerének meghatározásához elég csupán azt megfigyelnünk, hogy mi hogyan oldanánk meg ezt a feladatot a való életben.

Első lépésként húznánk egy rövidke vonalat a lapunk egy találmányra kiválasztott részére, mely az első folyosót jelképezné. Utána ebből a vonalkából kiindulva húznánk egy másikat egy véletlenszerű irányba. A következő lépésként pedig már nem csak az irányról döntenénk véletlenszerűen, hanem az új folyosórész kiindulási pontjáról is, mely az eddigi folyosórendszer bármelyik csomópontjában lehet. Ezt az eljárást addig ismételgetnénk, míg már nem maradna szabad rész a lapunkon, közben pedig végig figyelnénk arra, hogy stilizált útvesztőnknek egyetlen folyosója se haladjon le a lapról.

A módszer számítógépes megvalósítása sem sokkal bonyolultabb.

```
/* útvesztőépítés */
```

```
Szereplo[] folytathatoFolyosoreszek=new Szereplo[szint*szint];
```

```
folytathatoFolyosoreszek[0]=new Szereplo(indito);
```

```
int folytathatoFolyosoreszekSzama=1;
```

```
while(folytathatoFolyosoreszekSzama>0) {
```

```
    int vizsgaltFolyoso=(int)(Math.random()*folytathatoFolyosoreszekSzama);
```

```
    boolean folytathato=false;
```

```
    int lehetslegesIranyokSzama=0;
```

```
    int[] lehetslegesIranyok=new int[4];
```

Első lépésként létrehozuk a folytatható folyosórészek tömbjét, melybe mindig azokat a folyosórészeket fogjuk tölteni, melyek a négy irány közül valamelyik irányba még folytathatóak egy újabb folyosórészlettel. Mivel ezen folyosórészek maximális száma az útvesztőben található csomópontok számával egyenlő, elég, ha egy „szint*szint”-es méretű tömböt hozunk létre. A folytatható folyosórészek tömbjének kezdetben csak egyetlen eleme

van, a kiindulási pont, melyet a játékos helyzete alapján az indító nevű objektum határoz meg. Azt pedig, hogy a tömbbe eddig hány elemet töltöttünk, egy külön változóban tároljuk.

Az útvesztő építését addig folytatjuk, míg van folytatható folyosórész. Ha már nincs több, az azt jelenti, hogy az összes lehetséges helyet kitöltöttük már folyosóval, célunk tehát elértük.

A folyosóépítési algoritmus azzal kezdődik, hogy a számítógép véletlenszerűen választ egyet a folytatható folyosórészek közül, ezért van szükségünk a folytatható folyosórészek számára. Ezt követően ellenőriznünk kell, hogy az adott folyosórész folytatható-e még, és ha igen, akkor hány irányba folytatható, és melyek ezek az irányok.

```
/* lehetséges irányok meghatározása */  
Szereplo vizsgaltFolyosoresz=new  
Szereplo(folytathatoFolyosoreszek[vizsgaltFolyoso]);  
if(vizsgaltFolyosoresz.balSzomszedFal(terkep)) {  
    folytathato=true;  
    lehetsegesIranyok[lehetsegesIranyokSzama++]=Main.BALRA;  
}  
if(vizsgaltFolyosoresz.felsoSzomszedFal(terkep)) {  
    folytathato=true;  
    lehetsegesIranyok[lehetsegesIranyokSzama++]=Main.FELFELE;  
}  
if(vizsgaltFolyosoresz.jobbSzomszedFal(terkep)) {  
    folytathato=true;  
    lehetsegesIranyok[lehetsegesIranyokSzama++]=Main.JOBBRA;  
}  
if(vizsgaltFolyosoresz.alsoSzomszedFal(terkep)) {  
    folytathato=true;  
    lehetsegesIranyok[lehetsegesIranyokSzama++]=Main.LEFELE;  
}
```

A vizsgálatban a Szereplo osztály beépített metódusai vannak segítségünkre. Bármelyik irányba is folytatható a vizsgált folyosórész, a logikai változó értéke attól kezdődően igaz

lesz. Ha azonban a négy feltétel közül egyik sem teljesül, a változó értéke hamis marad, a folyosóról pedig megállapítható lesz, hogy nem folytatható.

```

if(folytathato==false) {
    /* a vizsgált folyosó kivonása a folytatható folyosók közül */
    folytathatoFolyosoreszek[vizsgaltFolyoso]=folytathatoFolyosoreszek[folytathatoFolyosoreszekSzama-1];
    folytathatoFolyosoreszekSzama--;
}

```

A nem folytatható folyosórészeket természetesen rögtön el kell távolítani a tömbből, mely úgy történik, hogy a tömb utolsó elemét a vizsgált folyosó helyére tesszük, majd csökkentjük a folytatható folyosók számát.

Ha azonban a folyosórész folytathatónak bizonyul, folytatódik az eljárás.

```

else {
    /* a vizsgált folyosó folytatása az egyik lehetséges irányba */
    int valasztottIrany=lehetsegesIranyok[(int)
(Math.random()*lehetsegesIranyokSzama)];
    MozgoSzereplo folyosoEpito=new
MozgoSzereplo(vizsgaltFolyosoresz,valasztottIrany);
    folyosoEpito.elore(terkep);
    terkep[folyosoEpito.getX()][folyosoEpito.getY()]=FOLYOSO;
    folyosoEpito.elore(terkep);
    terkep[folyosoEpito.getX()][folyosoEpito.getY()]=FOLYOSO;
    folytathatoFolyosoreszek[folytathatoFolyosoreszekSzama]=folyosoEpito;
    folytathatoFolyosoreszekSzama++;
}
}

```

Az előzőleg megtalált lehetséges irányok közül ismét véletlenszerűen választ a számítógép, majd létrehoz egy folyosóépítő szereplőt, mely a választott irányba lépve kettőt folytatja a folyosót. Az újonnan létrehozott csomópontot pedig felveszi a folytatható folyosók közé. Az eljárás végeztével már csak a cél meghatározása van hátra, mely az elvárásainknak

köszönhetően nem lesz a legegyszerűbb feladat. Éppen ezért élünk azzal a lehetőséggel, hogy a célkeresést egy külön segédosztállyal és annak alosztályával végeztetjük majd el.

```
    /* cél meghatározása */  
    cel=new Cel(jatekos);  
    cel.elhelyez(terkep);  
    terkep[cel.getX()][cel.getY()]=CEL;  
}  
}
```

A cél meghatározása a mozgatható célobjektum létrehozásával kezdődik, melyre meghívva saját célkereső eljárását, már el is végezzük feladatunkat, és egyben az Utveszto.java osztály leírását is befejezzük.

Szereplo.java

A Szereplo.java a szereplo csomag alaposztálya. Minden további osztálynak ez az „ősosztálya”. Egyedül a jatek csomagot importálja, azt is csak azért, hogy használni tudja az Utveszto osztály konstansait. Két jellemzője a térképen elfoglalt helyének x és y koordinátája. Ezek a mezők a Java konvencióhoz híven privát elérésűek, de legalábbis védettek azért, hogy más osztályok ne tudják közvetlenül módosítani őket. A get prepozícióval ellátott függvények szolgálnak az értékek lekérésére, az értékek beállítására pedig szándékosan nem adjuk meg a lehetőséget, hogy az egyes objektumok helyét csak a konstruktorban lehessen meghatározni.

```
package szereplo;
import jatek.*;
public class Szereplo {
    protected int x, y;
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
```

Mivel bizonyos esetekben arra is szükség lehet, hogy ellenőrizzük, két objektum pozíciója azonos-e, létrehozhatunk egy metódust külön erre a célra. Ennek híján ugyanis az x és y koordinátákat külön kellene a felhasználáskor összehasonlítani.

```
public boolean erint(Szereplo s) {
    if(s.x==x && s.y==y) return true;
    return false;
}
```

Az útvesztő építésekor pedig a Szereplo típusú folyosórészek szomszédjait is vizsgálnunk kell, ezért létre kell hoznunk egy-egy függvényt mind a négy irány számára.

```
public boolean balSzomszedFal(int[][] terkep) {
    if(terkep[x-2][y]==Utveszto.FAL) return true;
}
```

```

        return false;
    }
    public boolean felsoSzomszedFal(int[][] terkep) {
        if(terkep[x][y-2]==Utveszto.FAL) return true;
        return false;
    }
    public boolean jobbSzomszedFal(int[][] terkep) {
        if(terkep[x+2][y]==Utveszto.FAL) return true;
        return false;
    }
    public boolean alsoSzomszedFal(int[][] terkep) {
        if(terkep[x][y+2]==Utveszto.FAL) return true;
        return false;
    }
}

```

Mivel a feltételekben csak azt vizsgáljuk, hogy az egyes szomszédos csomópontok típusa FAL-e, ezért ha az éppenséggel FOLYOSO, a függvény már hamis értékkel tér vissza, s így a folyosóépítő sem folytatja majd az útját abba az irányba, ezzel hurkokat létrehozva. A trükk az egészben, hogy mint azt már fentebb tárgyaltuk, a játéktér széle KERET típusú, melynek vizsgálatakor a függvények ismét csak hamis értékkel térnek vissza. Ez az, ami biztosítja, hogy a folyosók ne haladjanak ki a játéktérből.

```

    public Szereplo() {}
    public Szereplo(Szereplo s) {
        this(s.getX(),s.getY());
    }
}

```

A paraméter nélküli üres konstruktorra az öröklődés miatt lesz szükségünk, Szereplo paraméterű konstruktorra pedig többek között a folyosóvizsgálatoknál. Érdekesség, hogy az osztályunk leírása ezzel véget is ér, így némi hiányérzetet hagyva maga után a pusztán önmaguk másolataként létrehozható példányok miatt. Ám játékunk futásához ennél több mégsem kell, ugyanis akárhányszor Szereplo típusú objektumokat használunk, azok mind önmaguk, vagy leszármazottjuk másolataként jönnek létre.

MozgoSzereplo.java

A MozgoSzereplo.java a Szereplo.java leszármazott osztálya. Ebben is felhasználjuk az Utveszto.java konstansait, ezért importáljuk a jatek csomagot. Mivel a szülőosztály mezői protected típusúak voltak, ezért azokat ez az osztály, mint leszármazott osztály látja, ám más osztályok nem.

```
package szereplo;
import jatek.*;
public class MozgoSzereplo extends Szereplo{
    protected int irány, lepes;
    public int getIrány() {
        return irány;
    }
    public int getLepes() {
        return lepes;
    }
}
```

A megörökölt mezőkön és metódusokon kívül ez az osztály természetesen továbbiakkal is rendelkezik. Új jellemzőként köszönhetjük az irányt és a megtett lépések számát tartalmazó mezőket, új metódusaink pedig a mezők értékeinek lekérésén kívül a léptetéssel és a forgással kapcsolatosak.

```
public void bal(int[][] terkep) {
    x--;
    lepes++;
}
public void fel(int[][] terkep) {
    y--;
    lepes++;
}
public void jobb(int[][] terkep) {
    x++;
```

```

        lepes++;
    }
    public void le(int[][] terkep) {
        y++;
        lepes++;
    }

```

A lépések metódusait irányonként írjuk meg, és minden lépés után növeljük a lépések számát. A bemeneti paraméter bekérése azonban – akárhogy is vizsgáljuk a programot – feleslegesnek tűnhet, de csak addig, míg a következő leszármazott osztályt szemügyre nem vesszük. Magára a magyarázatra viszont csak akkor térünk ki.

```

    public void elore(int[][] terkep) {
        switch (irany) {
            case Main.BALRA: bal(terkep); break;
            case Main.FELFELE: fel(terkep); break;
            case Main.JOBBRA: jobb(terkep); break;
            case Main.LEFELE: le(terkep); break;
        }
    }
    public void hatra(int[][] terkep) {
        irány=(irány+2)%4;
        elore(terkep);
        irány=(irány+2)%4;
    }
    public void jobbra() {
        irány=++irány%4;
    }
    public void balra() {
        irány=(irány+3)%4;
    }
}

```

A játékos viszont az öt jelképező objektumot nem a térképhez, hanem saját irányához viszonyítva mozgatja majd. Ezért van szükség az előre-hátra mozgás, és a balra-jobbra forgás

lekezeléséhez. Az `elore(int[][] terkep)` metódus nem csinál mást, minthogy az irány alapján továbbadja a mozgási utasítást az illetékes metódusnak. A `hatra(int[][] terkep)` metódus pedig még nála is „lustább”. Egész egyszerűen hátrafordul, majd tesz egy lépést előre, végül ismét megfordul, így elérve a hátrálás eredményét. A forgató metódusok dolga a legegyszerűbb, nekik még paraméterre sincs szükségük, – később tárgyalandó okokból – csupán a már korábbi maradékos osztással kapcsolatos módszerrel a kívánt irány sorszáma módosítja az irány mező értékét.

```
public MozgoSzereplo() {}  
public MozgoSzereplo(MozgoSzereplo ms) {  
    super(ms);  
    irány=ms.getIrány();  
    lepes=0;  
}  
public MozgoSzereplo(Szereplo s,int irány) {  
    super(s);  
    this.írány=írány;  
    lepes=0;  
}  
}
```

A konstruktorok között itt is köszönhetjük az üres, és a másoló konstruktort, valamint egy új konstruktorfüggvényt, mely a másolókonstruktor egy szó szerint „irányadó” változata. Konstruktoraink, ahol lehet, hivatkoznak a szülőosztály konstruktorára ezzel beállítva az örökölt mezők értékét, majd saját mezőiket is beállítják az elvárt módon. A `MozgoSzereplo` típusú objektumoknak másra nincs is szükségük.

SzilardMozgoSzereplo.java

A SzilardMozgoSzereplo.java a MozgoSzereplo.java leszármazottja, és jó néhány részletben hasonlít is hozzá. Legfontosabb újítása, hogy tekintettel van a falakra, és nem lép az adott helyre, ha ott fal található. Ahhoz viszont, hogy tudja, hol található fal, és hol nem, szüksége van az adott térképre, mint bemenetei paraméterre. S mivel a mozgással kapcsolatos metódusai felülírják szülőosztályának hasonló metódusait, a szülőosztálytól megörökölt előre(int[][] terkep) metódus már nem a szülőosztályának falakkal nem törődő mozgási metódusait hívja majd meg, hanem a falaknak ütköző felülírt metódusokat. Íme a magyarázat arra is, hogy miért volt szükség felesleges paraméterekre a szülőosztályban a mozgási metódusoknál. Ha ugyanis azokat paraméterek nélkül adtuk volna meg, most nem tudnánk őket felülírni, és a megörökölt előre(int[][] terkep) metódus is csupán a megörökölt mozgási metódusokat hívta volna meg.

```
package szereplo;

import jatek.*;

public class SzilardMozgoSzereplo extends MozgoSzereplo {
    public void bal(int[][] terkep) {
        if(balSzabad(terkep)) super.bal(terkep);
    }
    public void fel(int[][] terkep) {
        if(felSzabad(terkep)) super.fel(terkep);
    }
    public void jobb(int[][] terkep) {
        if(jobbSzabad(terkep)) super.jobb(terkep);
    }
    public void le(int[][] terkep) {
        if(leSzabad(terkep)) super.le(terkep);
    }
}
```

A felülírt metódusok alapja tehát a járható utakat kereső metódusok, melyeknek bemeneti paramétere maga az Utveszto példányosításakor létrejövő terkep[][] tömb. Egy

egyszerű feltétel kiértékelésekor eldől, hogy az adott irányba haladhat-e az objektum, vagy sem, és ennek alapján már meg is határozható a visszatérési érték.

```
public boolean balSzabad(int[][] terkep) {  
    if(terkep[x-1][y]==Utveszto.FAL) return false;  
    return true;  
}  
public boolean felSzabad(int[][] terkep) {  
    if(terkep[x][y-1]==Utveszto.FAL) return false;  
    return true;  
}  
public boolean jobbSzabad(int[][] terkep) {  
    if(terkep[x+1][y]==Utveszto.FAL) return false;  
    return true;  
}  
public boolean leSzabad(int[][] terkep) {  
    if(terkep[x][y+1]==Utveszto.FAL) return false;  
    return true;  
}
```

A térkép irányai mellett itt is szükségünk lesz az objektum irányához viszonyított irányok lekezelésére, mely a szülőosztályhoz hasonlóan a „probléma” illetékesekhez történő továbbításán alapszik.

```
public boolean eloreSzabad(int[][] terkep) {  
    switch (irany) {  
        case 0: if (balSzabad(terkep)) return true; break;  
        case 1: if (felSzabad(terkep)) return true; break;  
        case 2: if (jobbSzabad(terkep)) return true; break;  
        case 3: if (leSzabad(terkep)) return true; break;  
    }  
    return false;  
}  
public boolean jobbraSzabad(int[][] terkep) {
```

```

        SzilardMozgoSzereplo nez=new SzilardMozgoSzereplo(x,y,irany);
        nez.jobbra();
        if (nez.eloreSzabad(terkep)) return true;
        return false;
    }

    public boolean balraSzabad(int[][] terkep) {
        SzilardMozgoSzereplo nez=new SzilardMozgoSzereplo(x,y,irany);
        nez.balra();
        if (nez.eloreSzabad(terkep)) return true;
        return false;
    }
}

```

A nez objektumok létrehozására azért van szükség, hogy a SzilardMozgoSzereplo osztály adott példányának ne kelljen elfordulnia balra, vagy jobbra ahhoz, hogy megállapítsa, haladhat-e abba az irányba. Ezért egyszerűen létrehoz egy-egy segédobjektumot, melyek még a forgolódás „fáradtságától” is megkímélik. Igazság szerint persze azért van erre a trükkre szükség, hogy egy egyszerű logikai értéket visszaadó lekérdező függvény ne módosítsa az objektum adott mezőjének értékét.

A SzilardMozgoSzereplo konstruktorainak egyike a már jól ismert másolófüggvény. A másik azonban végre megismerteti velünk játékunk alapját, a szereplo csomag objektumainak elsőszülöttjét, az egyetlen önálló objektumot, melynek az összes többi csupán másolata.

```

    public SzilardMozgoSzereplo(SzilardMozgoSzereplo sms) {
        super(sms);
    }

    public SzilardMozgoSzereplo(int szint) {
        x=(int)(Math.random()*(szint-2))*2+4;
        y=(int)(Math.random()*(szint-2))*2+4;
        irany=(int)(Math.random()*4);
        lepes=0;
    }
}

```

A szereplo csomag osztályainak egyetlen példánya, mely nem másolatként jön létre, a SzilardMozgoSzereplo típusú jatekos. Létrehozásakor mégsem adhatjuk meg x és y koordinátáit, valamint irányát, ugyanis a jatekos minden játék elején véletlenszerű helyen fog megjelenni. Mint azonban azt már fentebb is említettük, az sem mindegy, hogy ez a véletlen hely hol található a térképen. Ha ugyanis nem akarjuk, hogy a jatekos a térkép szélén jelenjen meg a legközelebbi keret irányába nézve, – ezzel arra készítetve a folyosóépítőt, hogy készítsen egy játéktérből kivezető folyosót – jobban be kell határolnunk a véletlen számok intervallumát. Ezt a forráskódban is jól látható képlet segítségével tehetjük meg.

Ez az a játék, ahol tényleg a játékos körül forog a világ. Ugyanis minden esemény a játékost jelképező objektumból indul ki, vagy a játékosból származó objektumok egyikéből. A világ magára a jatekosra épül, még a szülőosztályai is ennek az objektumnak a „kiszolgálására” jöttek létre. S mivel azokban már megtalálható a legtöbb „képesség”, melyet a jatekos használ, maga a SzilardMozgoSzereplo osztály leírása is a lehető legegyszerűbb.

Cel.java

A Cel.java a SzilardMozgoSzereplo.java osztály leszármazottja, mely annak metódusait két továbbival egészíti ki.

```
package szereplo;

public class Cel extends SzilardMozgoSzereplo {

    public void athelyez(MozgoSzereplo s) {
        x=s.x;
        y=s.y;
        lepes=s.lepes;
    }

    public void elhelyez(int[][] terkep) {
        athelyez(new Celkereso(this).celkereses(terkep,new Celkereso(this)));
    }
}
```

Az első egyszerűen áthelyezi az objektumot a kívánt helyre – eddig példátlan módon – megváltoztatva annak kezdeti tulajdonságait, a második pedig segítségül hívva majdani leszármazottját az utolsó, mindeddig teljesítetlen elvárásunknak való megfelelésre vállalkozik, megkeresi a kiindulási ponttól legtávolabbi pontot. Mindehhez pedig létrehoz két Celkereso objektumot, melyek beépített metódusuk segítségével visszaadják az általuk megtalált legtávolabbi pontot, ahová már a Cel objektuma is áthelyeződhet. A második Celkereso-re valójában csak azért van szükség, hogy a metódus meghívható legyen. Ám mivel az a kiindulási ponton van, biztosan az első célkereső pozíciója lesz a függvény visszatérési értéke, mivel az el is fog mozdulni onnan, a második pedig nem.

```
public Cel(SzilardMozgoSzereplo sms) {
    super(sms);
}
}
```

Osztályunk leírásából pedig már csak a szokásos másolókonstruktor hiányzik, mely szüleihez híven ismét azok konstruktorára hivatkozik.

Celkereso.java

A Celkereso.java a Cel osztály leszármazottja, a legtávolabbi pont megtalálásáért felelős osztály. Függvényének egyik paramétere az Utveszto osztály térképe, melyre a térképen való mozgásnál van szüksége, nehogy áthaladjon a falakon. Másik paramétere az éppen aktuális legtávolabbi pont, mellyel összehasonlítva az általa talált legtávolabbi pontot, visszatérési értéként megadja a kettő közül vett távolabbt.

package szereplo;

public class Celkereso extends Cel {

public Celkereso celkereses(int[][] terkep, Celkereso legtavolabbiPont) {

while(eloreSzabad(terkep)) {

elore(terkep);

if(balraSzabad(terkep)) {

Celkereso celkereso=new Celkereso(this);

celkereso.balra();

legtavolabbiPont.athelyez(celkereso.celkereses(terkep,legtavolabbiPont));

}

if(jobbraSzabad(terkep)) {

Celkereso celkereso=new Celkereso(this);

celkereso.jobbra();

legtavolabbiPont.athelyez(celkereso.celkereses(terkep,legtavolabbiPont));

}

}

if(legtavolabbiPont.lepes>lepes) return legtavolabbiPont;

return this;

}

Algoritmusának gerince egy ciklus, mely addig lépteti egyenesen előre a Celkereso objektumot, míg az el nem éri a falat. Közben pedig állandóan figyeli a két oldalát, és ha elágazást talál, abba az irányba is meghívja a célkereső függvényt. Megvárja, míg a függvény visszatér, és megjegyzi az abban az irányban talált legtávolabbi pontot, majd tovább halad. A

falat elérve ellenőrzi, hogy messzebb van-e már a kiindulási ponttól, mint az általa létrehozott objektumok által talált legtávolabbi pont, és ha igen, akkor saját pozícióját adja vissza az őt meghívó osztálynak, ha pedig nem, akkor az általa létrehozott objektumok által megtalált pontot. Ez az algoritmus csodás példája annak, milyen hasznosak is a rekurzív metódusok.

```
public Celkereso(SzilardMozgoSzereplo sms) {  
    super(sms);  
    lepes=sms.getLepes();  
    }  
}
```

A másolókonstruktor pedig csupán abban tér el ősei példájától, hogy a lemásolt objektum lépéseinek számát is átadja az újonnan létrehozott objektumnak, így az pontosan tudja milyen távol van az első másolat eredetijétől, azaz a jatekostól.

Falnezó.java

A Falnezó.java osztály a SzilárdMozgóSzereplő osztály leszármazottja, és létezésének célja mindössze annyi, hogy a felderítendő térkép kirajzolásakor ne csupán az adott folyosó jelenjen meg, hanem azoknak a folyosóknak az első kis részlete is, melyek abból elágaznak. Hiszen azokat a játékos is látja, még ha csak részben is.

package szereplo;

public class Falnezó extends SzilárdMozgóSzereplő {

public Szereplő baloldal(int[][] terkep) {

Falnezó baloldal=new Falnezó(this);

baloldal.balra();

baloldal.elore(terkep);

return baloldal;

}

public Szereplő jobboldal(int[][] terkep) {

Falnezó jobboldal=new Falnezó(this);

jobboldal.jobbra();

jobboldal.elore(terkep);

return jobboldal;

}

public Falnezó(SzilárdMozgóSzereplő sms) {

super(sms);

}

}

Két függvénye a SzilárdMozgóSzereplő oldalvizsgálóihoz hasonlóan létrehoz egy-egy segédobjektumot, melyek az adott irányba tesznek egy lépést, majd visszaadják saját pozíciójukat. Ha az osztály nem hozná létre ezeket a példányokat, magának kellene elmozdulnia a helyéről, és ez a felhasználását nagyban megkönnyítená.

Végül pedig megállapítható, hogy a Falnezó osztály konstruktorának sem kell újításokat bevezetnie ahhoz, hogy tökéletesen működjön.

Main.html

A Falnezo.java osztály elkészítésével be is fejeződött programozói munkánk, ám ahhoz, hogy játékunkkal játszassunk, még egy weblapot is készítenünk kell, mely appletünket futtatja, amint annak forráskódjait Java gépi kóddá fordítottuk a javac.exe segítségével.

Az APPLET tag lesz az, mely beépíti weblapunkba frissen elkészült játékunk, csupán meg kell adnunk az attribútumok értékeiként a megfelelő paramétereket. A codebase kulcsszó után adhatjuk meg, hogy hol található az appletünk alkotóelemei, és code után kell megadnunk azt, hogy pontosan hol található a futtatandó osztály. A width és height attribútumokkal pedig az appletünk méretét állíthatjuk be. (Angster Erzsébet 2003 : 330. p.)

```
<HTML>
<HEAD>
<TITLE>Útvesztő</TITLE>
</HEAD>
<BODY BGCOLOR="black" TEXT="yellow">
<p align="center"><APPLET codebase="classes" code="jatek/Main.class" width=800
height=600></APPLET></p>
<p align="center">A menüben a fel/le nyilakkal választhatsz a menüpontok közül, és a
jobbra/balra nyilakkal állíthatod át őket. Az útvesztőbe belépni jobbra nyíllal lehet.</p>
<p align="center">Az útvesztőben a fel/le nyilakkal mozoghatsz, és a jobbra/balra nyilakkal
foroghatsz. Ha az útvesztő mind a tíz szintjén megtalálod a létrát, mely a következő szintre
visz, szabad vagy. Ha viszont megpróbálsz leereszkedni egy korábbi szintre, az azt jelenti
feladtad, és visszakerülsz a menübe.</p>
<p align="center">Jó játékot!</p>
</BODY>
</HTML>
```

Amint a weblap is elkészült, már csak egy böngészőt kell találnunk, mely képes megnyitni az appletekkel kiegészített weblapokat, és máris elmerülhetünk a játékban. Nincs is más hátra, mint hogy itt is azt kívánjam, mint amit a weblap alján található ismertetőben:

Jó játékot!

Összefoglalás

Mindent összefoglalva megállapíthatjuk, hogy a Java programnyelv teljes mértékben alkalmas webes felületű játékok írására is, melyek elkészítése közben a nyelv jó néhány alapvető lehetőségével is megismerkedhetünk. A programozás közben szerzett tapasztalatok pedig nem csak későbbi webprogramozói munkásságunkat alapozhatják meg, hanem a további játékok elkészítésével kapcsolatos törekvéseinket is.

A játékprogram első sorainak megírásakor még nem is gondoltam, hogy egy ilyen összetett, és kihívásokkal teli feladatot vállaltam magamra. Aztán ahogy először viszont láttam az addig csak tervekben szereplő helyszíneket, úgy gondoltam képzelőerőmnek immár semmi sem szabhat gátat. Számtalan ötlet fogalmazódott meg bennem, melyek tovább színesíthették volna a játékot, ám ezekből mindössze fájóan keveset tudtam megvalósítani. Megismerkedhettem hát a programozó legnagyobb ellenségével, az időhiánnyal. Olyan sok cikkben olvastam már, hogy a játékfejlesztők még rengeteg mindent bele akartak építeni a játékukba, ám már nem jutott rá idejük, most pedig magam is megtapasztalhattam ezt. Végül pedig a játékcégekhez hasonlóan én is megvizsgáltam, mik azok a részletek, melyek elkészítésére még van időm, a többitől pedig fájó búcsút vettem.

A játékom így még időben elkészült, és remélem beváltotta a hozzá fűződő reményeket, igazi kikapcsolódást nyújtva az arra vágyók számára. Ha csak egyetlen ember is mondja azt magában, mikor feltűnik előtte a napfényes mező képe, hogy „na, ez jó volt”, nekem az már elég a sikerélményhez. Persze ettől függetlenül is remélem, hogy minél többen, minél többet fognak majd játszani vele.

Természetesen nem várom azt, hogy munkám olyan elismerésben részesüljön, mint azok a játékok, melyeken évekig dolgozik több tucatnyi szakember, ám ha egyszer megadatik a lehetőség, hogy egy ilyen csapat része legyek, biztosan élni fogok majd vele, hogy az elismerésekre én is méltó legyek. Persze, ha csak erre vágnék, bele sem kezdtem volna a játékfejlesztésbe. Ugyanis úgy programozni, hogy csak a jutalom lebeg a szemünk előtt, miközben nem is élvezzük magát a programozást, nem sok értelme van. Számomra ugyanis maga a játékfejlesztés jelenti az örömet. Programozás közben sokszor előfordult, hogy egy-egy ötlet megvalósításakor elégedetten nevettem fel, hogy milyen frappáns, és milyen

látványos kódot sikerült összehoznom, aztán mikor a játék kipróbálásakor is láttam, hogy a programrészlet tökéletesen működik, már tudtam, megérte vele dolgozni. Csak azt sajnálom, hogy munkám döntő hányada a játékos számára igazából rejtve marad, és csak az vigasztal, hogy egy-két komoly programozó is megismeri majd mind a játékot, mind a kódot, és látja majd, hogy mi is az, ami ezt az összetett gépezetet mozgatja.

A játék témájának kiválasztásával is elégedett vagyok. Azt hiszem az első önálló játékom számára jobb nem is akadhatott volna. Most viszont, hogy elkészültem vele, máris tolongnak az ötletek, hogy milyen játékot csináljak legközelebb. Talán ezt is továbbfejlesztsem valamilyen módon, vagy egy teljesen más jellegű játék készítésébe kezdek. Még nem tudom, de abban biztos vagyok, hogy nem leszek tétlen a jövőben, és gyakran fogok leülni a számítógép elé, hogy játékokat programozzak.

Remélem továbbá, hogy dolgozatom írásának elején kitűzött célomat is teljesítettem, és sikerült méltó emléket állítanom a világ első számítógépes játékainak. A Mouse in the Maze, és az általam készített Útvesztő is talán nem a legösszetettebb játékok, ám céljuknak azt hiszem tökéletesen megfelelnek. Demonstrálják, hogy a számítógép a beprogramozása után képes megfelelően reagálni a felhasználók által teremtett helyzetekre, és olyan helyzeteket teremt válaszul, melyek még a programozók számára is újdonságként hatnak. Röviden, hogy a mesterséges intelligencia tökéletes kiegészítője lehet az emberi intelligenciának. Ha azonban éppenséggel nem világmegváltó kutatásokra használják, vitathatatlan, hogy kikapcsolódásnak sem utolsó dolog.

Végül pedig reményeim szerint igényes munkával bővíthetem a számítógépes játékokkal kapcsolatos irodalom szűk körét, és bizonyíthattam, hogy a számítógépes játékok is méltóak arra, hogy egy dolgozat róluk szóljon.

Dolgozatom zárásaként pedig hadd köszönjem meg tanárainknak, Dr. habil. Boda Istvánnak és Iszály György Barnának, hogy megismertették és megszerettették velem a Java programozási nyelvet, valamint Király Magdolnának, hogy ötleteivel és tanácsaival segített engem munkámban.

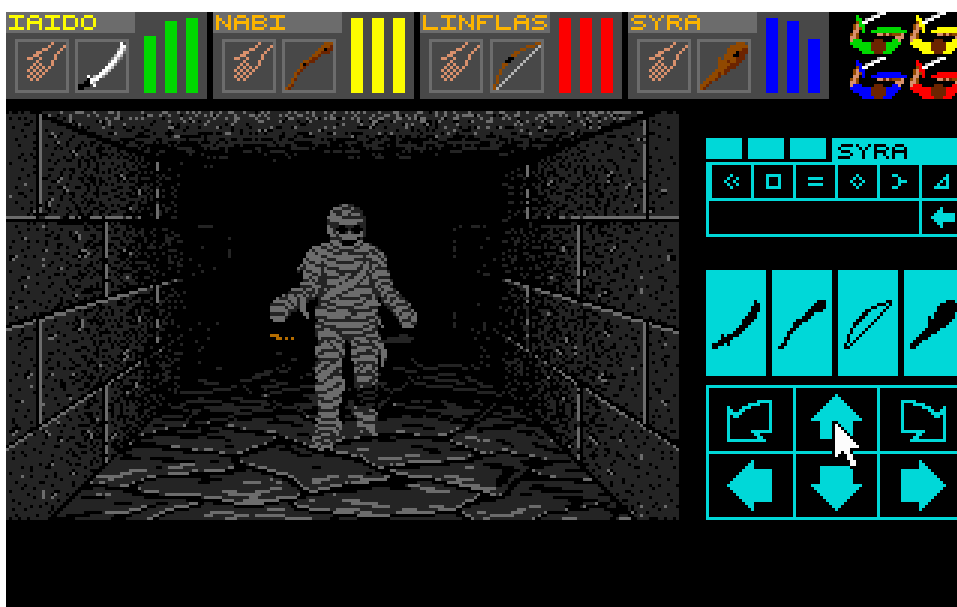
Irodalomjegyzék

1. Instant Java, Java EE, SOA / Vég Csaba. - Debrecen : Logos 2000, 2007.
2. Java felsőfokon / Daniel J. Berg, J. Steven Fritzinger. - Budapest : Kiskapu, cop. 1999
3. Java programozási nyelv : alapismeretek : nyitott rendszerű képzés - távoktatás -
oktatási segédlete : felsőoktatási tankönyv / Móricz Attila ; [közread. az] LSI
Oktatóközpont a Mikroelektronika Alkalmazásának Kultúrájáért Alapítvány. -
Budapest : LSI OMAK, 1997 [!2001] [Budapest] : Ligatura ; [Vác] : Naszályprint
4. Objektumorientált tervezés és programozás : Java / Angster Erzsébet. - 2. jav. kiad. -
[Budapest] : 4KÖR BT, 2003.
5. Grath: Driv3r. In: PC Guru, 2004. 3. sz. 20-23. p.
6. Júpi: Dungeon keeper. In: Pc-X, IV. évf. 9. sz. 1997. szeptember, 14-15. p.
7. Koronczi Gáspár: Diablo. In: 576 Kbyte, VIII. évf. 2. sz. (1997. február), 20-21. p.
8. Mazur: Hitman 2 [kettőspont] Silent Assassin. In: Gamestar, 2002. november,
64-67. p.
9. Mr Chaos: Hexen 2. In: Pc-X, IV. évf. 9. sz. 1997. szeptember, 8-9. p.
10. Pellus: Stonekeep. In: Pc-X, III. évf. 2. sz. (1996. február), 16. p.
11. Sam Joe: Hexen. In: Pc-X, III. évf. 2. sz. (1996. február), 13. p.
12. Shy DaCosta: Dungeon master 2. In: Pc-X, II. évf. 9. sz. (1995. szeptember), 10. p.
13. TJ: Dungeon keeper. In: 576 Kbyte, VIII. évf. 7-8. szám (1997. július-augusztus),
10-12. p.
14. Zong [és] Júpi: Diablo 2. In: Pc-X, 1998. április, 6-8. p.
15. Appletek aszt.inf.elte.hu/~kto/teaching/java/material/13_applet.pps
16. Dungeon Master (video game)
[http://en.wikipedia.org/wiki/Dungeon_Master_\(computer_game\)](http://en.wikipedia.org/wiki/Dungeon_Master_(computer_game))
17. Dungeon master [http://www.abandonia.com/files/games/585/Dungeon](http://www.abandonia.com/files/games/585/Dungeon_%20Master_3.png)
[%20Master_3.png](http://www.abandonia.com/files/games/585/Dungeon_%20Master_3.png)
18. Dungeons Of Daggorath http://en.wikipedia.org/wiki/Dungeons_of_Daggorath

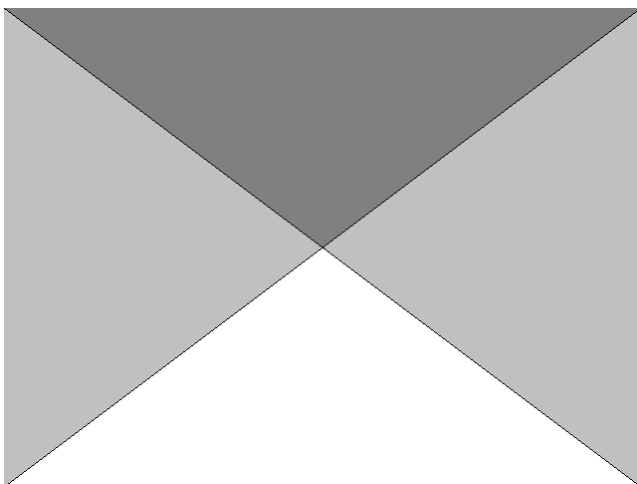
19. Echoes of War <http://www.echoes-of-war.com/>
20. Eseménykezelés http://home.sch.bme.hu/~gijo/8_ESEMENYEK/index.html
21. Free MP3 WMA WAV Converter <http://www.nbxsoft.com/mp3-converter.php>
22. GIMP <http://www.gimp.org/>
23. Gyógyítás virtuális valóságban http://www.magyarhirlap.hu/Archivum_cikk.php?cikk=90985&archiv=1&next=0
24. Hexen soundtrack <http://sycraft.org/content/audio/hexen.shtml>
25. Hexen: Beyond Heretic <http://en.wikipedia.org/wiki/Hexen>
26. Játék (pszichológia) [http://hu.wikipedia.org/wiki/Játék_\(pszichológia\)](http://hu.wikipedia.org/wiki/Játék_(pszichológia))
27. Java <http://java.sun.com/>
28. John Carmack http://hu.wikipedia.org/wiki/John_Carmack
29. NetBeans <http://www.netbeans.org/>
30. OpenOffice.org <http://www.sun.com/software/openoffice/index.jsp>
31. RPG.hu http://rpg.hu/szotar/main.php?do_this=list_by_letter&letter=D
32. Sun Microsystems <http://hu.sun.com/>
33. Videojátékok története <http://oli76.ingyenweb.hu/keret.cgi?/e04.htm>
34. WavePad <http://www.nch.com.au/wavepad/index.html>

Függelék

1. ábra: Dungeon Master¹²

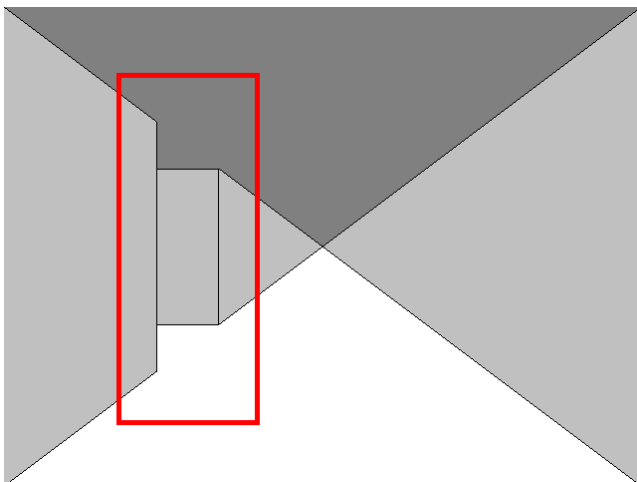


2. ábra: Folyosókép vázlat

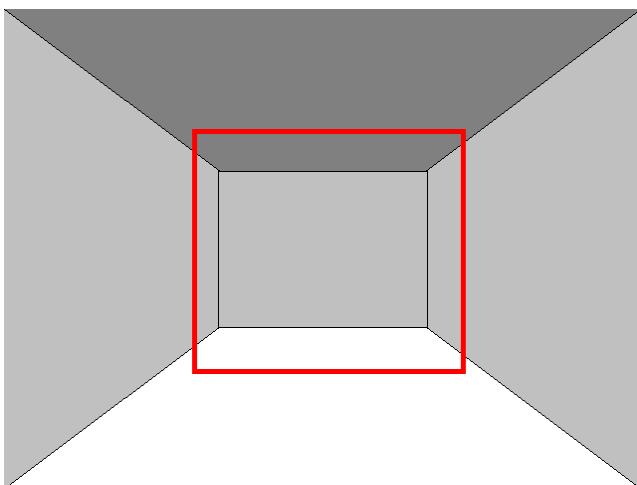


¹² Dungeon Master http://www.abandonia.com/files/games/585/Dungeon%20Master_3.png

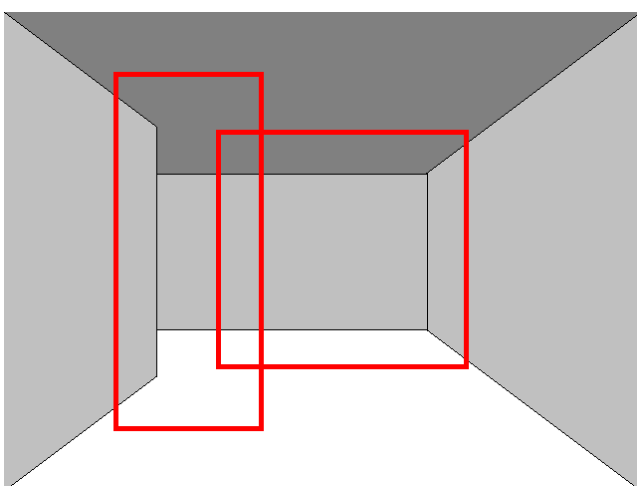
3. ábra: Elágazáskép vázlat



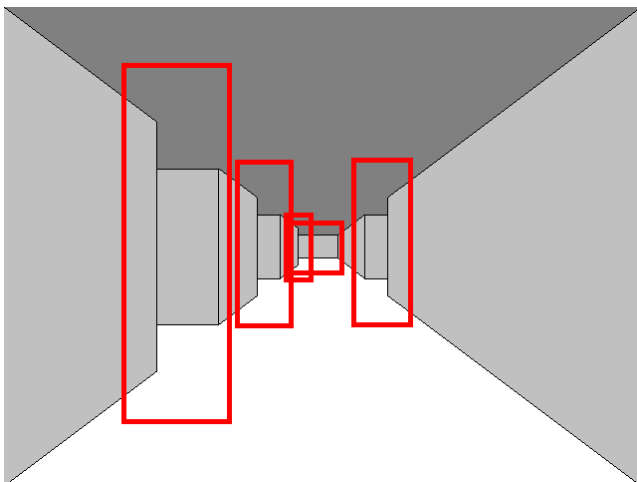
4. ábra: Zsákutcakép vázlat



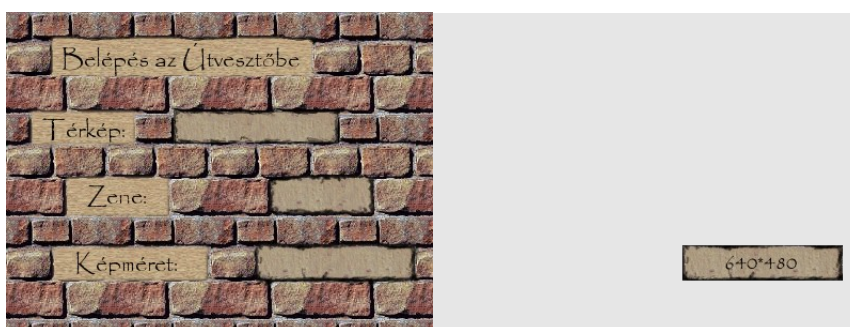
5. ábra: Kanyarodó folyosó vázlat



6. ábra: Végleges látványterv



7. ábra: Egymást átfedő képek módszere



8. ábra: A kész játék

