FELADATKIÍRÁS

A feladatkiírást a tanszéki adminisztrációban lehet átvenni, és a leadott munkába eredeti, tanszéki pecséttel ellátott és a tanszékvezető által aláírt lapot kell belefűzni (ezen oldal HELYETT, ez az oldal csak útmutatás. Fénymásolat nem jó, ezért mindenki igényeljen megfelelő számú eredeti iratot az adminisztrációban).

Budapesti Műszaki- és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatika Tanszék

Csengődi Péter

MULTIMÉDIÁS ALKALMAZÁS FEJLESZTÉSE FELÜGYELT KÖRNYEZETBEN

KONZULENS

Albert István

BUDAPEST, 2006

HALLGATÓI NYILATKOZAT

Alulírott **Csengődi Péter**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök, stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Tudomásul veszem, hogy az elkészült diplomatervben található eredményeket a Budapesti Műszaki- és Gazdaságtudományi Egyetem, a feladatot kiíró egyetemi intézmény saját céljaira felhasználhatja.

Kelt: Budapest, 2006. 05. 18.

.....

Csengődi Péter

Tartalomjegyzék

Összefoglaló

Mivel a hardverek az utóbbi évtizedekben rohamos fejlődésen mentek át, sebességük és kapacitásuk megengedhetővé tették a szoftverek számára az erőforrások lazább kezelését, pazarlását. Nagyobb projekt fejlesztésében így nem az alkatrészek jelentenek korlátot, hanem a hibalehetőségek sokasága, az aprólékos munka, és az olvashatatlan forráskód.

A felügyelt platform nagy előrelépést jelentett fejlesztési idő lerövidítésében, egyaránt csökkentette a hibaforrásokat és egyszerűsítette a forráskód szintaxisát. Ezzel együtt a .Net Framework rengeteg olyan programozási feladatra ajánl fel előre megírt megoldást, amelyek a legtöbb alkalmazás fejlesztése során előfordulnak.

Mindezzel együtt az ipari alkalmazások többsége C++ nyelvben íródott, és felügyelt kódban csak kiegészítő programokat, szerkesztőket implementálnak. A legfőbb indokként azt hozzák fel, hogy a .Net Platform mögött elhelyezkedő programozási technológia több erőforrást és processzoridőt igényel, mint a C++ programok natív kódú futása, ezért abban nem lehet elérni az ipar által megkövetelt teljesítményt.

A munkámat egy újabb példának szántam annak bizonyítására, hogy a felügyelt technológia jelenlegi állása szerint igenis alkalmas valós idejű multimédiás alkalmazás fejlesztésére. A program egy belső nézetű játék, amelyben egy labirintust kell végigjárnunk az adott szint teljesítéséhez. Fontosabb témák megfelelő labirintus véletlenszerű generálása és kezelése, dinamikus vetett árnyék megjelenítése, valamint többjátékos üzemmód hálózaton keresztül.

Abstract

In the last decades hardware components went through an apace development, their speed and capacity allowed software to use resources and processor time more sloppy and wasteful. Hardware became no more the bound of the development of huge projects; numbers of fault sources, minute work and unreadable source code are obstructive instead.

Managed platform became a big step forward to reducing the development time by making the number of fault sources less and easing down the syntax of the source code. Moreover .Net Framework offers previously implemented solutions for a lot of programming difficulties occurring in almost every application development.

However, industry uses mainly applications written in C++ language, while managed code remained on the level of editors, tools and utilities. Engineers explain this phenomenon with the fact, that the technology used by the .Net Platform needs more resources and processor time, than the execution of native C++ code, which still means the trademark of performance.

This work is meant to be another proof for the appropriateness of the managed technology for developing real-time multimedia applications, according to its present state. This program is a first person game, where the player has to find an exit out of a maze to accomplish the actual level. Main topics are generating and handling of a random-built mazes, interactive drop shadows, and multiplayer mode using network communication.

1 Bevezető

A játékszoftverek fejlesztése és forgalmazása talán a legsikeresebb és legbizonytalanabb szoftveripari ágazat. Ezt a kettősséget annak köszönheti, hogy nagy a felhasználók köre, hiszen termékei nem konkrét megrendelők, vagy valamely szűk kör igényeit elégítik ki, hanem az átlagember érdeklődését próbálja felkelteni, ugyanakkor a nagy versengés és az egyre növekedő elvárások miatt újabb és újabb technikákat kell elsajátítanunk és felhasználnunk a fejlesztés során. A tág piacnak köszönhetően elmondható, hogy a grafikus megjelenítést tekintve ez a terület vált úttörővé, a legtöbb fejlesztés itt történik.

Továbbá ami miatt ezt az ágazaton különösen érdekesnek találom, az a tény, hogy a grafika szépsége nem határozza meg a játék sikerességét. Rengeteg régi szoftver maradt örökérvényűen a játékosok emlékeiben, míg más csúcs technológiát használó alkalmazások hamar feledésbe merültek. A fő szempont, hogy a játék menete mennyire ötletes, újszerű, kihívásokkal teli

Éppen ezért lenne fontos, hogy egy játékszoftver fejlesztése során minél könnyebben, gyorsabban túl lehetne esni a megjelenítő és vezérlő egységek fejlesztésén, hogy minél hamarabb felszínre kerüljenek azok az elemek, amelyek a játék lényegi részeit alkotják, illetve olyan személyek tudják megmutatni ötleteiket, akik nem jártasak eléggé a szoftverfejlesztésben.

Erre a problémára nyújt megoldást a .Net keretrendszer és a hozzá tartozó fejlesztői eszközök. Az objektum orientált alapok növelik a forráskódok olvashatóságát, érthetőségét, a felügyelt nyelvek csökkentik a hibalehetőségeket, és a keretrendszer osztálykönyvtárai használata felgyorsítja a fejlesztést. A DirectX felügyelt interfészének megjelenésével ezt a szemléletet a grafikus megjelenítés megírások is felhasználhatjuk.

A diplomatervem egy saját játékszoftver megírásán keresztül mutatja be a fentebb vázolt konfiguráció lehetőségeit. Szó esik néhány olyan technikai elemről, amellyel a grafikus megjelenítést javíthatjuk, továbbá más olyan elemekről, amellyel a játékunk hátterét lehet emelni, valamint hálózati játék lehetőségét biztosíthatjuk. Ezen kívül tárgyalásra kerül kiegészítő alkalmazások, eszközök fejlesztése is.

A szoftver használatát bemutató felhasználói dokumentáció és a fejlesztői környezet rövid jellemzése után a negyedik fejezet mutatja be az elkészült fő alkalmazás alaparchitektúráját. Itt különös figyelmet fordítottam a megfelelő tagoltságra, legfőképpen a Model-View-Controller hármas kialakítására. Ennek legfontosabb feladata, hogy az egyes egységek a többi modul minimális megváltoztatásával cserélhetővé válnak, aminek másodlagos következménye, hogy a DirectX specifikus, bonyolult szintaktika egyetlen egységre korlátozódik.

Az ötödik fejezet néhány olyan megoldást vázol fel, amellyel véletlenszerűsíthetjük a játék egyes szintjeit, ezzel mindig új kihívás elé állítva a felhasználót. A labirintusgenerálás mellett helyet kapott az is, hogyan lehet egy térképből megfelelő kinézettel és fizikai megtestesüléssel rendelkező teret automatikusan kialakítani.

A következő részben írtam le néhány grafikai szépítőelemet, majd egy egyszerű ütközésdetektáló algoritmus ismertetése következik.

A nyolcadik fejezet az inkrementális képszintézisben használható vetett árnyék algoritmusokról ad rövid összefoglalást, majd a Shadow Volume (árnyéktest) technikát ismertetem részletesen, kitérve annak megvalósítási nehézségeire is.

A kilencedik és tizedik fejezet az objektum orientált szemlélet és a .Net keretrendszer lehetőségeit mutatja be állapot- és adatkezelés terén, bemutatva az alkalmazásban használt korszerű megoldásokat.

A hálózati kommunikációról és a többszereplős játék kialakításáról a tizenegyedik fejezetben írtam, ahol ismertetem az alkalmazás hálózati architektúráját, kommunikációs protokollját, valamint az adatforgalmat mind új játék létrehozásakor és ahhoz való csatlakozáskor, mind játék közben.

Végül a tizenkettedik és tizenharmadik fejezetben írtam le azon kiegészítő eszközök felépítését és működését, amelyekkel a játék kinézetén, élvezhetőségén lehet javítani.

2 Felhasználói dokumentáció

2.1 Telepítés, indítás

A játék egyszerű kitömörítéssel, felmásolással is telepíthető, mindazonáltal a futtathatóságnak a következő követelményei vannak:

Microsoft .Net Platform 1.1
DirectX 9.0c (SDK: summer version)

A végfelhasználó szempontjából a szoftvernek két fontos egysége van:

Superstition: Maga a játék. A *Release\Superstition.exe* állományt kell elindítani a futtatáshoz.

GameServer: Központi szerver program hálózati futáshoz. A futtatandó állomány a *GameServer\GameServer.exe*.

2.2 Egyszemélyes játék

2.2.1 Indulás

A játék indításakor megjelenik egy menü, ahol kiválaszthatjuk a megfelelő beállításokat, indítási paramétereket. A fő menünek öt eleme van: **Játék indítása *[Game]***, **Hálózati funkciók *[Network Play]***, **Beállítások *[Options]***, **Visszatérés a játékhoz *[Resume Game]*** (csak játék közben), **Kilépés *[Quit]***. A menü kezelése a *„fel”* és *„le”*, valamint az *„enter”* billentyűkkel történik. Az utóbbi két menüpont különösebb magyarázatot nem igényel.

2.2.2 Beállítások

Elsősorban a tesztelés érdekében jött létre ez a menüpont. Segítségével a hatásokat ki- és bekapcsolhatjuk, valamint a grafikai modellt tehetjük részletezettebbé, vagy kevésbé részletessé a gyorsítás érdekében. Ez utóbbit a **Detail** menüpont átállításával tudjuk hangolni, amelynek hatása csak játék indítása előtt érvényes, ezért játék közben nem állítható át. Ha **Dynamic Shadows** pont be van kapcsolva, akkor a program vetett árnyékokat is készít. Mivel maga a játéktér egyszerűbb elemekből áll, amiknek könnyen (és gyorsan) lehet árnyékot készíteni, a bonyolultabb (mesh) alakok csak akkor rendelkeznek árnyékkal, ha a **Mesh Shadows** menüpont is be van kapcsolva.

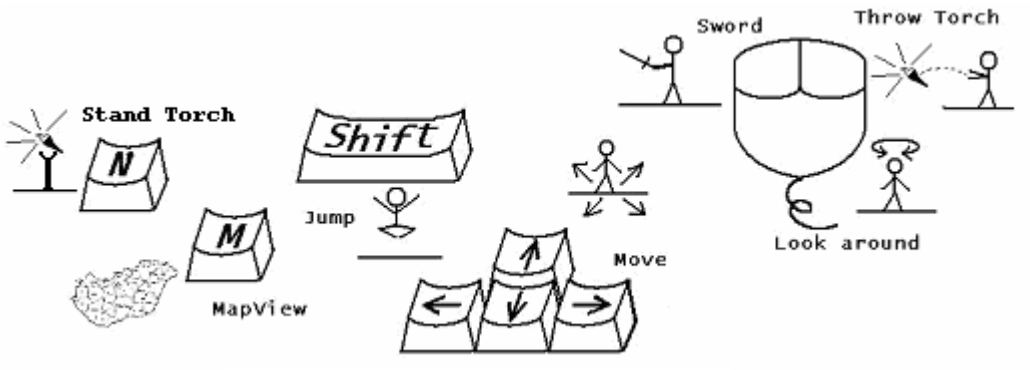
2.2.3 Játék indítása

Új játékot egyszemélyes módban a **Game** menüpont **Load Map** menüpontjával lehet. Ezzel kiválaszthatunk egy előre elkészített *xml* fájlt a *maps* könyvtárból. Ezek az állományok segédinformációt tárolnak a véletlenszerű generáláshoz, ami segítségével egy pálya létrejön. Ezért ugyanannak az *xml* fájlnak a kiválasztásával előállhatnak különböző szerkezetű, de nagy vonalakban hasonló pályák.

Egy elmentett játékot a **Load Game** menüponttal választhatunk. A menüpont kiválasztásakor lehetőségünk van egy a *saves* könyvtárban elhelyezkedő bináris állomány kijelölésére, amit betölteni szeretnénk. A **Game** almenüben játék közben lehetőségünk van egy pályát hasonlóképpen elmenteni.

2.2.4 Irányítás

Ha elindítottuk a játékot, a pályát a karakterünk szemszögéből láthatjuk. A következő ábra mutatja, hogy a karaktert miként lehet irányítani.



1. ábra: Az irányítás szemléltetése

Nyilak ..... Mozgás
Shift ..... Ugrás
M ..... Térkép nézet
N ..... Fáklya állítása
Egér Mozgatása ..... Karakter forgatása
Egér bal gomb ..... Kard kinyújtás (nincs különösebb hatása)
Egér jobb gomb ..... Fáklya eldobása/felvétele

2.3 Hálózati játék

2.3.1 Szerver

A hálózaton legelőször is szükségünk van egy szerverre. A **GameServer** programot kell betöltenünk, és elindítanunk. Elindítás előtt megadhatunk egy *port* számot, amivel a számítógép *IP-címével* együtt használva egyértelműen azonosíthatjuk a szervert. (Egy gépen azonos *port*-on csak egy szervet üzemeltethetünk.)

Egy adott szerver több játékot, úgy nevezett *host*-ot tud egyszerre futtatni. Egy adott *host* **Publish** művelettel lehet indítani, ami azt jelenti, hogy egy kliens saját, addigi egyszemélyes játékát meghirdeti a szerver felé. Ezután a *host*-ra **Join** művelettel lehet csatlakozni

Ahhoz, hogy a szerverhez hozzáférhessünk, először kell készíteni egy **konfigurációs állományt** a játék *servers* alkönyvtárába. A **konfigurációs fájl**-ok egyetlen címkét tartalmazó *xml* állományok, a gyökér címke attribútumként tartalmazza a szerver *nevét*, *címét*, a megnevezett *port*-ot, valamint egyéb információt.

2.3.2 Kliens

A játék főmenüjében található **Network Play** menüpont kiválasztásával a játék servers alkönyvtárába elhelyezett *konfigurációs állományok* listája jelenik meg. Egy ilyen fájl kiválasztásával jelölhetünk meg egy szervert. A program ezután feloldja a megnevezett szerver *címét*, és a feloldáskor kapott *IP-címekből* választhatunk.

Ez után információt kapunk arról, hogy a szerver host-jai közül melyek vannak elindítva, és melyek üresek. Az elindított *host* helyén **„Join Game**” felirat jelenik meg, valamint hogy mennyien csatlakoztak hozzá, és hogy mekkora a limit. Egy ilyen menüpontot kiválasztva csatlakozhatunk az adott *host*-on futó játékhoz.

Amennyiben mi még nem indítottunk el játékot, a többi *host* helyén a játék **„<empty>”** feliratot jelenít meg. Ha azonban éppen fut egy játék a kliensen, akkor **„Publish Game”** felirat jelenik meg. Ilyen menüpontot választva meghirdethetjük a játékunkat a kiválasztott szerveren, és várhatjuk mások csatlakozását.

3 Fejlesztői környezet

A szoftver fejlesztéséhez a .Net Framework 1.1 keretrendszert használó Microsoft Visual Studio .Net 2003. fejlesztői szoftvert használtam, a grafikus megjelenítést a DirectX 9.0c (summer version) végzi, amely rendelkezik egy, a Microsoft Visual Studio .Net 2003. belsejébe integrálható SDK komponenssel és mintakódokkal is.

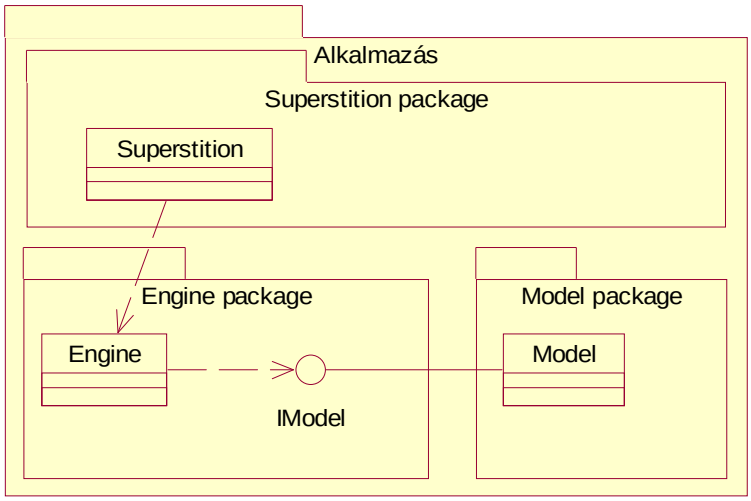
3.1 A .Net programozása

A legszélesebb körben használt nyelv a C#, amely szintaxisát tekintve a C++ utódja, azonban a működése (érték és referencia típusok, szemétgyűjtő algoritmus, stb.) sokkal inkább hasonlít a Java nyelvre. Fontos különbség azonban, hogy míg a Java nyelvet első sorban interpretált, platform független nyelvként tartják számon, a .Net környezetben a forráskódok először köztes nyelvre, végül gépi kódra fordulnak le, amely lehetővé teszi a Microsoft Windows platform specifikus tulajdonságainak kihasználását, amely a sebesség növelésére, optimalizációra ad lehetőséget.

3.2 DirectX

A DirectX 9.0 verziójának megjelenésével a grafikus megjelenítés kapott felügyelt alkalmazásfejlesztési felületet, aminek folyamán a programozás ezen a területen is sokkal egyszerűbbé vált: egy előre felépített és implementált osztály-hierarchiával biztosítják a forráskódok jobb olvashatóságát, az objektum orientált programozási módszerek felhasználását, továbbá kód-újrafelhasználás lehetővé tette, hogy gyakran használt algoritmusokat az alkalmazás fejlesztésének kezdetekor is készen kapjunk.

4 Az alkalmazás architektúrája



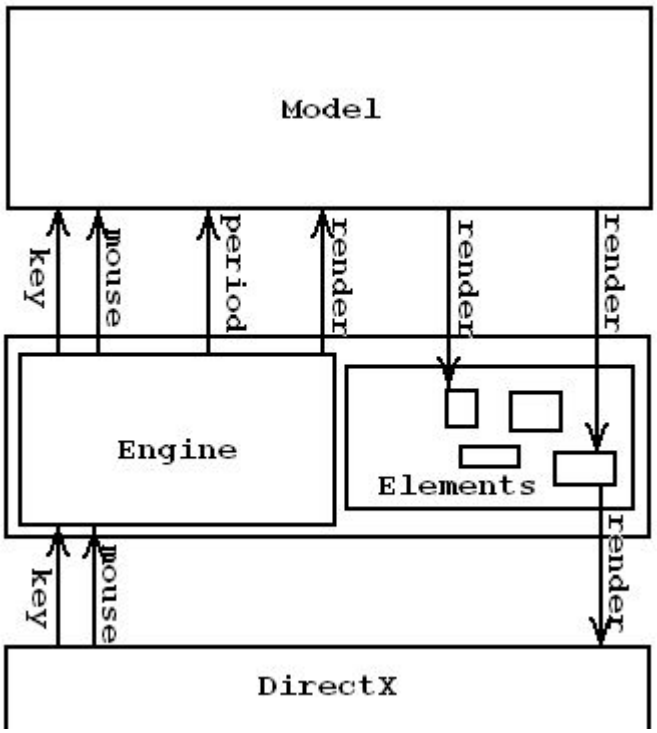
2. ábra: Az alkalmazás fő moduljai

Az alkalmazás alap architektúrája a Document-View programozási minta alapján készült el. Magát az alkalmazás inicializálásáért felelős a Superstition osztály, a funkcionalitások közül ezért semmi nincs itt implementálva, csupán a program működésének főbb osztályaiból készít példányokat, és azokat ellátja kezdőértékekkel.

A rendszer SDI jellegű, azaz egyszerre egyetlen nézet létezik, amely egyetlen dokumentumot jelenít meg, azonban a dokumentumok futás-időben cserélhetőek. A nézetet az Engine osztály (grafikus motor) reprezentálja, amely a Superstition osztályban létrehozott ablakhoz kapcsolódik, és azon jeleníti meg a programot. Az éppen aktuális dokumentumot mindig egy IModel interfészből származó példány testesíti meg, az ábrára a Model osztályt tettem, mivel a játék lényegi részének implementálása ebben található.

4.1 Motor architektúra

Az Engine csomag gyakorlatilag egy beékelődött réteg a DirectX és az IModel objektum között. Feladata az, hogy elfedje a modell elől a DirectX API hívásokat, és biztosítson egy grafikai elemkészletet a modell számára, valamint egy megfelelő eseménykezelést.



3. ábra: A motor rétegszerű elhelyezkedése

A DirectX kezelésének központi eleme az Engine osztály. A következő szolgáltatásokat nyújtja:

4.1.1 Inicializálás

A jelenlegi verzió a DirectX moduljai közül kettőt, a grafikáért felelős (Direct3D + Direct3DX) és a beviteli perifériákat kezelő (DirectInput) egységeket. Az ezekhez tartozó eszközöket (grafika, billentyűzet, egér) hozza létre megfelelő paraméterekkel és kezdőbeállításokkal.

4.1.2 Eseménykezelés

A billentyűzet és az egér figyelése a DirectInput segítségével, eseménykezeléssel oldható meg. Az esemény beérkezésekor a motor kifejti az üzenetet, és az éppen használt modell megfelelő eseménykezelő függvényét hívja meg. Így a modellben csupán a helyes függvények felüldefiniálásával implementálni lehet az adott eseményekre irányuló válaszokat. Továbbá a motoron keresztül folyik a fizikai rész időkezelése és a grafikus időzítés kezelése is, ami szintén a modell függvényeinek meghívásait végzik.

4.1.3 Szálkezelés

A beviteli perifériák, a fizikai és a grafikus egységek eseményei mind-mind külön szálon futnak. Ennek indoka, hogy az egyes folyamatok sebessége ne legyen hatással a többire. (pl. Ha a grafikai megjelenítés sebessége időről-időre változik, a fizikai cselekmények továbbra is egyenletesen folyjanak.) Az ehhez szükséges funkciókat, melyek közül a legfontosabb a kölcsönös kizárás, szintén itt oldottam meg.

4.1.4 Cserélhető modell

A motor a modell funkcióinak hívását egy leválasztott (IModel) interfészen keresztül végzi, ezért a modell bármikor átcserélhető egy másikra. Ehhez további segítséget nyújt az állapotkezelési modul, ahol az egyes állapotok mondják meg, mi annak a modellnek a referenciája, amit aktuálisan kezelni kell; továbbá a csomag által tartalmazott elemkészlet, amelynek használatával a motor és a modell között egy lazább függés alakulhatott ki.

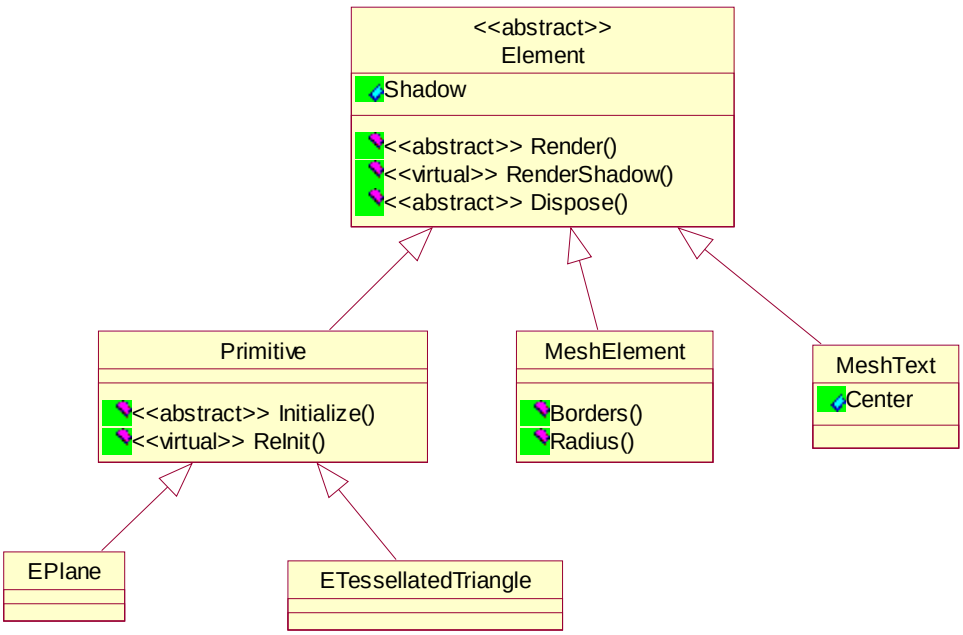
4.1.5 Motoron keresztül igényelhető elemkészlet

A motor csomagjába beépített grafikus elemek szintén tartalmaznak Direct3D API hívásokat, ezért ők a motorral szoros kapcsolatban állnak. Ezzel ellentétben pont az a szerepük, hogy a modell grafikai építőkövei legyenek. Éppen ezért ezeket az elemeket a motoron keresztül tudjuk igényelni, amely az elemek példányosításakor átadja saját referenciáját a példányoknak, így az elemek képesek a motor egyes szolgáltatásaira hivatkozni; másrészről a motor regisztrálja a kiadott elemeket, arra az esetre, ha a

modell nem gondoskodik a DirectX elemek takarításáról (memóriaszivárgás a grafikus kártyán), a motor megteszi helyette.

(Ez a szivárgás nem természetesen a felügyelt platformnak köszönhetően nem fog kijutni az alkalmazáson kívülre, hiszen a Managed DirectX kihasználja a .Net által nyújtott lehetőségeket ennek a problémának a megoldására. Viszont az alkalmazás futása során modellek létrejönnek és megszűnnek, és így hosszabb, változatos futtatás során így jobban kezelhetjük az erőforrásainkat.)

Mindez a modell elől el van rejtve, így a modell megírásakor ezekkel nem kell foglalkoznunk. Az elemkészlet osztályhierarchiáját az alábbi ábra szemlélteti:



4. ábra: A motor grafikai elemeinek hierarchiája

Az Element egy közös ősosztály az elemkészlethez, amely tartalmazza a minden elem esetében legfontosabb hívható funkciók listáját, és a közös beállítási lehetőségeket.

Ezen belül a Primitive a gyűjtőosztálya azon elemeknek, amelyek a DirectX grafikai rendszerét alacsony szinten (háromszögek szintjén) használja. Az EPlane egy diszkrét irányokba forgatható téglalapot reprezentál, az ETessellatedTriangle pedig egy bárhogyan elhelyezhető háromszöget. Ezek az alkalmazáshoz speciálisan létrehozott alakzatok, amelyeket a program fő modellje gyakran használ. További jellemzőjük,

hogy – beállíthatóan – apró háromszögekből pakolja össze őket a program, mivel a DirectX Garaud árnyalást használ, ezért a grafikai elemek vertex-beli felbontása határozza meg fény megjelenítésének pontosságát.

A MeshElement a DirectX Mesh osztályát csomagolja be a kívánt módon, azaz előre elkészített, bonyolult alakzatokat kezel magas szinten. A MeshText egy hasonló osztály, amely kizárólag szöveg objektumokat kezel, és – a megvalósítása miatt – csak a menürendszerben való szereplésre alkalmas.

4.1.6 Megjelenítési folyamat kezelése

Ez a dokumentáció a későbbiekben tárgyalni fogja, hogy milyen grafikai látványosságokat, algoritmusokat tartalmaz a megírt alkalmazás, amelyek azt is igénylik, hogy többszöri, több lépcsős renderelés történjen minden egyes képkocka előállításakor. A renderelések sorrendezését, paraméterezését a motor elvégzi, a pontos megjelenítések az elemkészletben vannak implementálva. A modellben a megjelenítéskor meghívott függvényt természetesen egyszer kell elkészíteni.

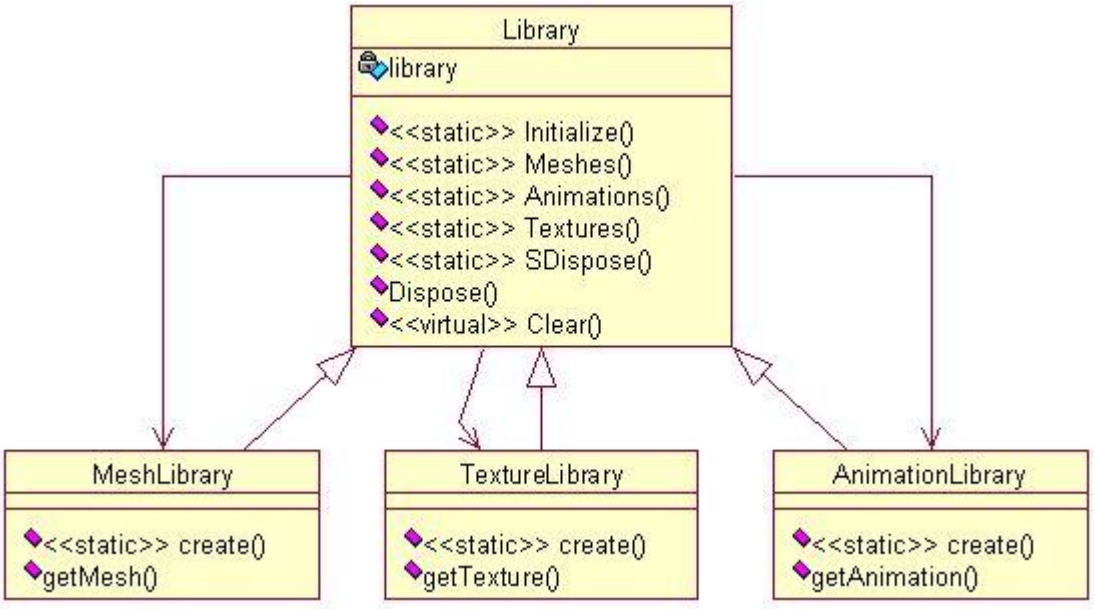
4.1.7 Modell-könyvtárak

A grafikai elemek, azaz textúrák, mesh objektumok és animációs jelenetek nagy erőforrást igényelnek, ezért egy játék során ezeket szeretnénk csak egyszer betölteni a rendszerbe. A grafikus megjelenítés alapelve segít minket, hiszen ezeket az elemeknek egyetlen példányát a világ-térben többször is felhasználhatjuk, ha megadjuk a megfelelő transzformációt. Az alkalmazásban implementáltam olyan osztályokat, amelyek segítségével a grafikus modellek betöltéseit kezelhetjük.

A Library osztály a Singleton tervezési mintán alapul, némi módosítással. Három könyvtárpéldányt tartalmaz, amelyek megfelelnek az egyes grafikai elemek típusának. Ezt az osztályt tovább örököltettem a típusoknak megfelelően, hogy a különböző elemek külön bánásmódban részesülhessenek, de az egyes könyvtárakat legegyszerűbben az ősosztály statikus függvényein keresztül érhetjük el.

Ennek a megoldásnak a szerepe, hogy ne kelljen a könyvtárakról minden olyan helyen referenciát tárolni, ahol felmerülhet a használatuk, hanem a Library osztályon keresztül könnyedén meg lehessen szerezni a modelleket.

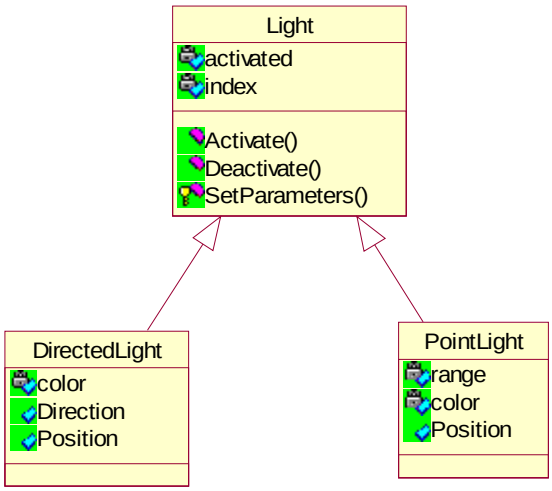
A könyvtár osztályok felépítése egyszerű, egy listában tárolják azoknak a grafikai elemeknek a referenciáját, amelyek már be vannak töltve, és hozzájuk társítva egy azonosítót, ami a betöltendő fájl neve. Így ha egy adott grafikai elemre hivatkoznunk, elég a fájlnévre emlékeznünk, és nem kell azonosítókat tárolnunk, ha egy – akár már betöltött - objektumhoz akarunk hozzáférni.



5. ábra: A könyvtárak osztály-szerkezete

4.1.8 Fénykezelés

A fényforrások kezeléséhez szintén rendelkezésre bocsátottam a modell számára csomagoló osztályokat, amelyek tartalmazzák a megfelelő DirectX hívásokat, a modelltől rejtetten. Két fajtája van: A játék során egy belső terű labirintus jelenik meg, amit fáklyák és lámpák világítanak be, ehhez a Point fények a legalkalmasabbak, de a menürendszernek szüksége van valamilyen nagyobb kiterjedésű, egyenletesebb fényre, ezért ott Directional fényforrást használok.



6. ábra: A motor fénykezelésének osztályai

Fontos azonban megjegyezni, hogy a DirectX egyszerre összesen nyolc fényforrással tud számolni. Ahhoz, hogy ezt a kevés helyet a lehető legoptimálisabban használjuk ki, a modellnek kell kezelnie, hogy az adott pillanatban melyik fényforrás aktív és melyik nem. Erre alkalmasak az Activate és Deactivate függvények, amelyekkel az adott fényforrás be- vagy kikapcsolhatjuk anélkül, hogy a DirectX felületével érintkeznünk kellene, a szabad hely keresését illetve a paraméterezést automatikusan elvégzi maga az osztály.

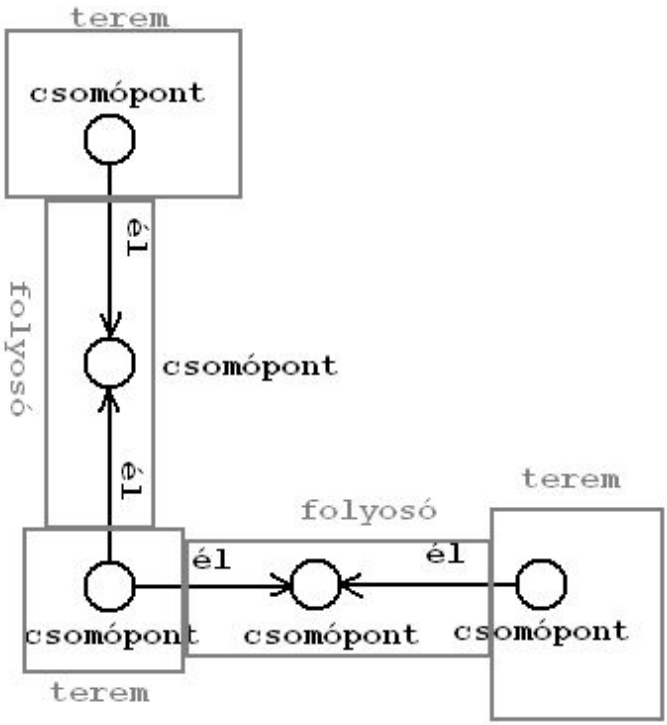
4.2 Modell architektúra

Az alkalmazás többi része több különböző csomag (egyben névtér) összessége. Ezek együttesen alkotják a fő architektúra dokumentum részét. Az alábbiakban olvasható, hogy az egyes csomagok milyen funkciókat tartalmaznak.

4.2.1 Model

Ez az egység testesíti meg magát a fizikai teret, annak előállítását, kezelését és takarítását. Maga a fizikai tér objektumok sokaságnak összessége, ahol az egyes példányok megadják a fizikai szimuláció működését, és hogy milyen grafikai elemeket használnak fel a motor csomagból. Külön tárolódik a játéktér és külön a játékosokhoz objektumok halmaza.

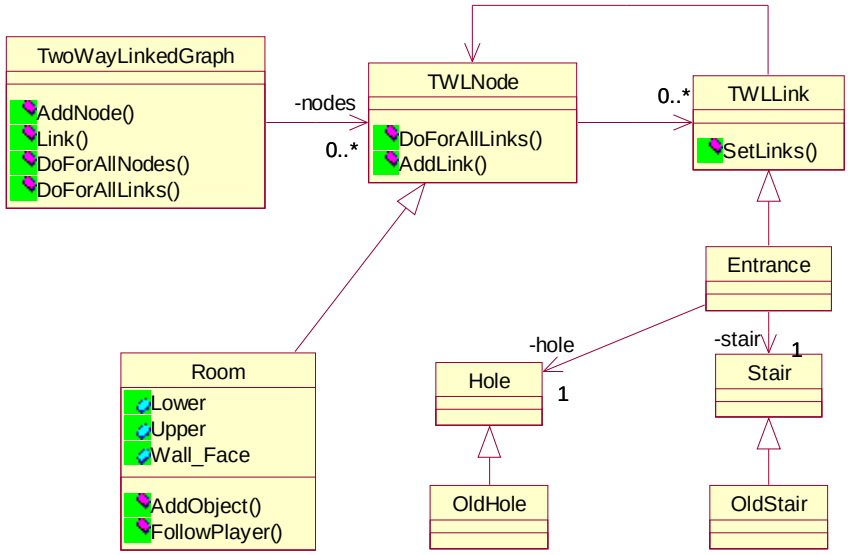
A játéktér architektúrája szoros kapcsolatban áll a Mathematics csomaggal, hiszen a labirintus felépítésének elve egy irányított gráf. A csomópontok termek vagy folyosók (a gyors megoldás miatt a folyosó is terem, csak mások a méretei), és az élek átjárók közöttük. Ezáltal a későbbi fejlesztésekben lehetőség nyílik gráf alapú algoritmusok (pl. útkeresés) használatára is.



7. ábra: A térkép gráfszerkezete

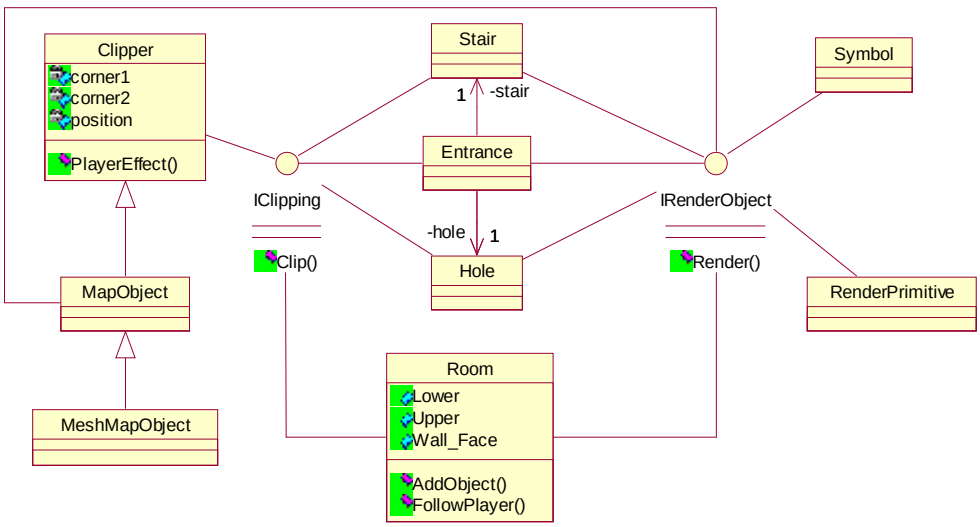
Osztályhierarchiát tekintve a játéktér egymáshoz illesztett terem (Room) példányokból áll, amelyeket átjárók (Entrance) kötnek össze. Az átjárók állnak egy nyílásból (Hole) és egy lépcsőből (Stair), amely a szomszédos termek szintkülönbségeit hidalják át. A termek felelősek a bennük levő tárgyakért, azok megjelenítései és fizikai változásai rajtuk keresztül történik.

Az átjárók, a termek és a bennük található pályaelemek kapcsolatát az alábbi ábra mutatja.



8. ábra: A játékteret alkotó elemek

Az IRenderObject azokat az osztályokat jelöli, amelyek grafikus megjelenítéssel rendelkeznek, ilyen elem például a Symbol, amely egy a földön forgó szimbólum, vagy a RenderPrimitive, ami csomagoló osztály a motor csomag nyújtotta elemkészletnek.

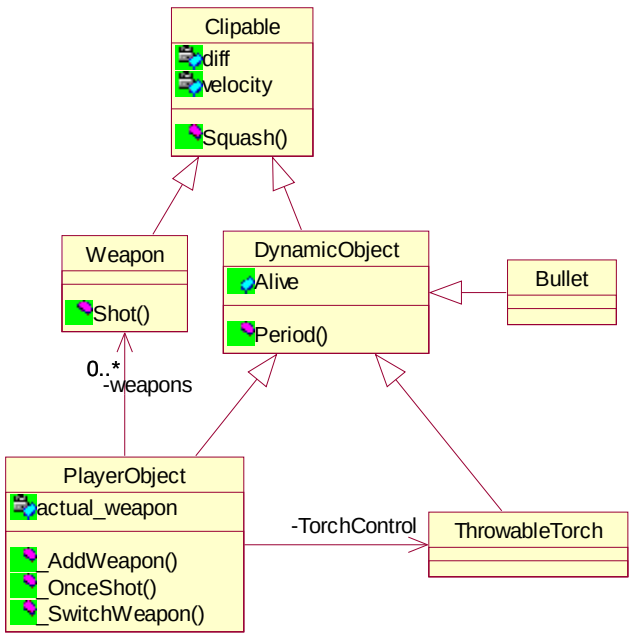


9. ábra: Ütköző objektumok osztály-hierarchiája

Az IClipping jelöli az ütközés-detektálásban résztvevő objektumokat. A Clipper a téglatesttel körbezárt ütköző objektumok ősosztálya, és mivel semmilyen grafikai elem alapból nem tartozik hozzá, ezért láthatatlan ütközőként és használható. A

MapObject a játéktéren megjelenő, mozdulatlan objektumok ősosztálya, ennek specifikus eleme a MeshMapObject, ami egy mesh elemet testesít meg.

A játékoshoz tartozó objektumok ettől függetlenül kezelendők, hiszen ezek az elemek a különböző termek között vándorolnak, így nem lehet egyértelműen egy adott terem megjelenítéséhez és fizikájához hozzárendelni. Az efféle objektumok osztályhierarchiája alább látható.



10. ábra: Mozgó objektumok

A DynamicObject olyan objektumok közös őse, amelyek bizonyos fizikai tulajdonsággal rendelkeznek (pl. hat rájuk a gravitáció). A PlayerObject reprezentálja a játékost, a ThrowableTorch a kezében a fáklyát, amely egyben a vetett árnyékokhoz felhasznált fényforrás is. Ezen kívül az ábrán fel van tűntetve a fegyvereket (Weapon) és a töltényeket (Bullet) megtestesítő osztályok is.

4.2.2 Egyéb csomagok

Mathematics: Ez a csomag néhány (vektorral, gráfokkal, véletlen számokkal, stb. foglalkozó) matematikai algoritmus gyűjtőterülete.

Menu: A játék menüjének megjelenítéséért, kezeléséért felelős csomag.

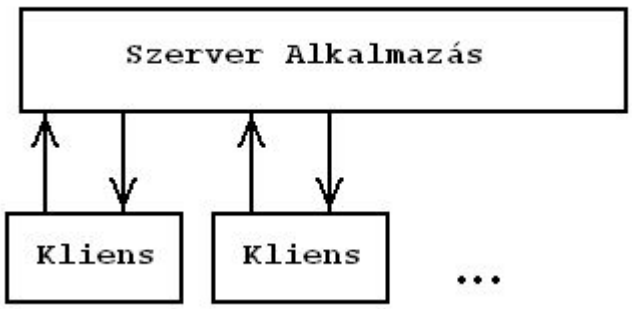
StateControl: Az alkalmazás állapotkezelését valósítja meg.

Network: A hálózati működés implementációja.

GameLib: Más alkalmazásokkal közös osztályokat tartalmazó könyvtár

XmlDataBinding: XML kezelést megvalósító könyvtár

4.2.3 Hálózati architektúra



11. ábra: Kliens-szerver architektúra

A hálózati megoldás kliens-szerver alapú, ahol a szerver egy külön, a többitől független alkalmazás. Ennek értelmes, hogy akár magát a menetet indító kliens kilépésével is folytatható a játék. Természetesen a szerver és a kliensek azonos gépen is futtathatóak. Mivel a szerveralkalmazásnak gyorsnak és egyszerűnek kell lennie, valamint hogy a kliens változásakor minél kevesebbet kelljen rajta módosítani, csak leegyszerűsített adatokkal dolgozik, a többi szükséges adatot (pl. a játéktér térképe) binárisan tárolja. Nem tartalmaz háromdimenziós megjelenítést sem, és a számolások nagy részét továbbra is a kliensek a megszokott módon végzik.

5 Labirintus generálása

5.1 Algoritmus

Az alkalmazás fejlesztése során szerettem volna a játékba véletlenszerű elemeket tenni. Ez kettős célt szolgál: Először is a játékos minden újrakezdésnél új kihívással szembesül, másodszor pedig a fejlesztőnek kevesebb munkába kerül egy új szint létrehozása. Másrészről változatos generátorok előállítása költséges programozási feladat.

A játékban szereplő szinteket labirintus-generáló algoritmussal állítom elő. Azonban szerettem volna megőrizni azt a lehetőséget, hogy a pálya készítésekor a fejlesztő befolyásolja a generálás menetét, és így lehetőség szerint könnyebb és nehezebb szinteket állíthasson elő. Ennek módszere a következő:

A pályakészítő megnevez olyan fő termeket, amelyeket a játékosnak (opcionálisan) mindenképp meg kell látogatnia. Ezek után megmondja, hogy melyik fő terem melyik másikból érhető el (mindenképp). Meghatározza, hogy az egyes fő termekben milyen szimbólumok, illetve tárgyak találhatóak. Ezt a formátumot neveztem el „cselekmény”-nek, ezek alapján a szoftvernek automatikusan létre kell tudni hoznia az adott szintet.

Mivel a játék zárt térben játszódik, a legkézenfekvőbb ötlet egy labirintus generálása, amely az előzőekben leírtak alapján a következő folyamattal írható le:

1. **Létrehozunk egy megfelelő méretű játékmezőt.**
2. **Kijelölünk helyet a fejlesztő által meghatározott fő termeknek.**
3. **Összekötjük őket.**

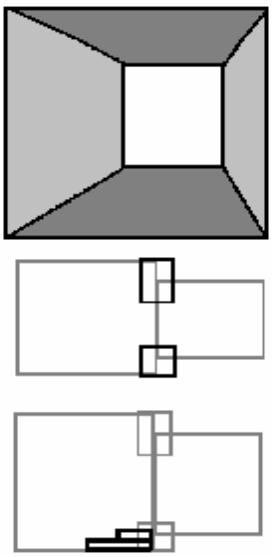
Matematikai értelemben tehát veszünk egy gráfot, amelyet síkba rajzolunk úgy, hogy a csomópontoknak véletlenszerű pozíciót adunk meg. A feladat jellege miatt a „síkba rajzolás” kifejezést itt nem a klasszikus értelemben használtam, hiszen a felhasználó spontán előállíthat egy olyan térképet is, amely bizonyítottan nem

rajzolható le élek keresztezése nélkül. Ezért az adott pálya cselekményét szerkesztőnek figyelembe kell vennie azokat az eseteket, amikor bizonyos pontokat a véletlen folyamán akár egy szimpla folyosóval összeköt.

A labirintus generálásánál két fontos tulajdonságot tartottam szem előtt:

- **A labirintus termek és folyosók sokaságából álljon össze.**
- **Korlátozott legyen a megjelenítendő környezet.**

Ha komplex, véletlenszerűen előállított szerkezetet akarunk létrehozni, akkor ahhoz tetszőlegesen variálható alapelemekre van szükségünk. Tervezési döntésként azt hoztam, hogy a labirintust termekből, azaz – gyakorlatilag – belső nézetű dobozokból állítom össze, amelyek mérete tetszőleges értékeket vehet fel, és a benne található különböző díszek, elemek a méretek alapján arányosan generálódnak. Megjegyzendő, hogy a fejlesztést tekintve a „terem” és a „folyosó” fogalmak egyenértékűek.



12. ábra: A labirintus építőelemei

Az ábra azt mutatja meg, hogyan készül el egy több teremből álló szerkezet.

Először is felvesszük a megfelelő dobozokat, és eldobjuk azokat az oldalakat, amelyekből átjáró fog nyílni másik terem felé. Azután a két összeérő terem közé szegélyt építünk, úgy, hogy a nyílást a lehető legnagyobbra vesszük, ami még egyik teremnél sem nagyobb. Végül vesszük a termek szintkülönbségét, és az alacsonyabb aljzatú terem széléhez lépcsőt teszünk.

A módszer egyszerű, és ha tudjuk a termek pontos méreteit, akkor automatizálható.

A második pont ennek alapján a következő magyarázattal egészül ki: Egy terem renderelése meghatározott időt és erőforrást igényel. A gyorsítás érdekében arra

törekedtem, hogy egyszerre csak meghatározott számú terembe láthasson bele a felhasználó, így egyszerre csak kevés termet kelljen megjeleníteni.

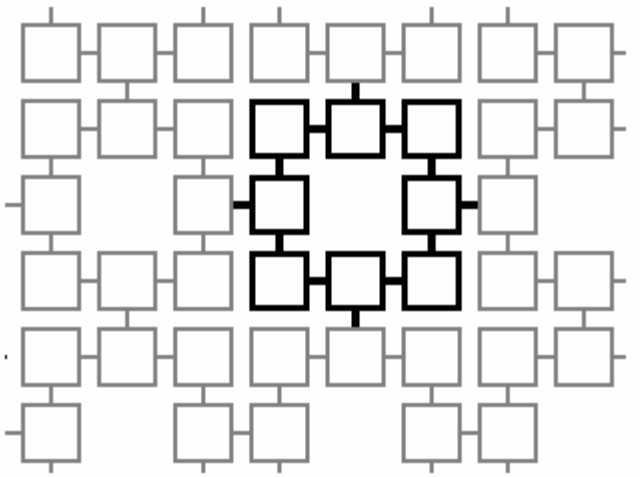
Azt, hogy egy ilyen előre megfogalmazott tulajdonsággal épüljön fel a labirintus, két módszerre érhetjük el.

On-Line A labirintus építése közben állandóan figyelünk a korlátozó tényezőkre, a bővítést mindig úgy folytatjuk, hogy a feltétel továbbra is teljesüljön. Bonyolult algoritmust, és a szerkesztés során sok erőforrást igényel, viszont az adott feladatkör összes megoldását megkaphatjuk.

Off-Line Előre meghatározunk korlátozásokat, és a szerkesztéshez csak olyan eszközöket nyújtunk, amivel nem lehet olyan labirintust létrehozni, ami a feltételt nem teljesíteni. Ezzel a feladatkört szűkítjük, kevesebb megoldást kaphatunk meg, de a megoldások száma továbbra is kielégítő lehet. Viszont mindenképp gyorsabb, takarékosabb módszer, és kevesebb kódolást igényel.

Az egyszerűsége miatt az Off-Line módszert használtam. Nevezetesen létrehoztam egy olyan alapszerkezetet a pályához, amelyben biztosítva van, hogy ha e fölött szerkesztjük meg a labirintust, akkor a játékos csak korlátozott számú termet láthat. A szerkezetet az alábbi ábra mutatja.

Az alapszerkezetben csomópontok vannak elhelyezve rács formában. Közéjük be van rajzolva, hogy a szerkesztés folyamán melyik csomópont melyik másik csomóponttal lehet egyáltalán összekötve. Az összeköttetést megvalósító szűk folyosók miatt a játékos több termet csak egyenes vonalban láthat be, az alapszerkezet így biztosítani tudja, hogy egyszerre legfeljebb öt termet lehessen látni.



13. ábra: Labirintusok alapszerkezete

A labirintus megszerkesztése így a következőképpen folyik:

1. **Megszámoljuk, hogy a fejlesztő hány fő termet kért. Ennek a számnak az arányában létrehozunk egy akkora méretű alapszerkezetet, amelyben a labirintus részeit kényelmesen el tudjuk helyezni.**

2. **A fő termek pozícióját e szerkezet csomópontjai közül választjuk ki véletlenszerűen.**

3. **Távolsági függvényen alapuló heurisztikus útkereséssel képzünk egy járatot két-két összetartozó fő terem között. Később ezt a járatot csomópontokkal és folyosókkal „kikövezzük”.**

4. **Igény szerint a labirintust zsákutcákkal bővíthetjük.**

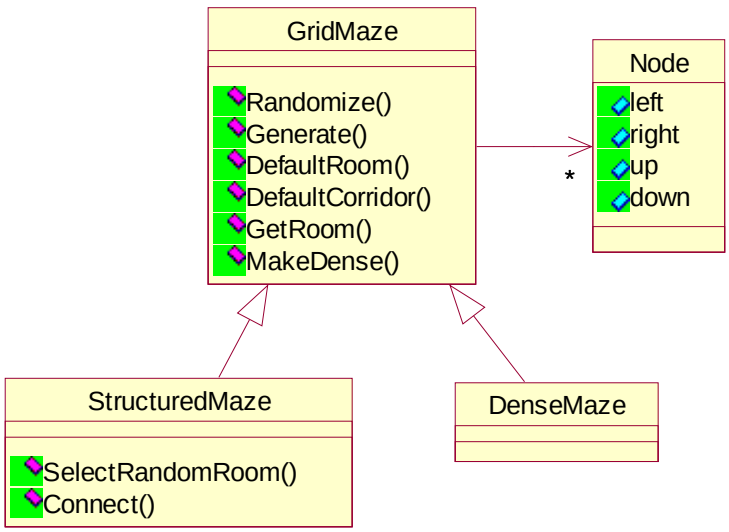
5.2 Implementáció

Az egyszerű generálásnak köszönhetően a termek, a hozzájuk tartozó szegélyekkel és lépcsőkkel együtt is felépíthetőek olyan téglalap elemekkel, amelyek oldalai a koordináta-tengelyekkel párhuzamosak, és a felületük normálisai a térbeli hat fő irányba mutatnak. Ennek megvalósítására alkottam meg az EPlane osztályt. Ennek feladata egyrészt, hogy a megadott iránynak megfelelően a téglalap grafikai elemei úgy készüljenek el, hogy a megfelelő culling módban lehessen megjeleníteni. Másrészt, hogy a téglalapot feldarabolja apró háromszögekre.

Ez utóbbi azért kell, mert a DirectX Garaud árnyalást használ a megjelenítés során. Ezt programozási szempontból azt a megkötést teszi, hogy a téglalap árnyalásának pontossága attól fog függeni, mennyire sűrűn helyezkednek el vertex-ek a téglalapon belül. Ha például a téglalapnak csak a négy sarkába tennénk vertex-eket, akkor előfordulna, hogy egy a téglalap közepénél elhelyezett fényforrás hatása meg sem jelenne. Úgy írtam meg a daraboló algoritmust, hogy a következő két szempontot teljesítse:

- **Legyen egy minimum vertex sűrűség.**
- **A téglalapon mindenhol egyenletes legyen a vertexek sűrűsége.**

A labirintus elkészítéséért külön osztályok a felelősek. Ezek hierarchiáját az alábbi ábra mutatja:



14. ábra: Labirintusok osztály-szerkezete

A GridMaze egy általános ősosztály, amely megvalósítja a korábban elemzett labirintus-alapszerkezetet a Node segédosztály segítségével. A szerkezetben kijelölhetünk termeket, és megadhatjuk, hogy melyek legyenek egymással összekötve. Egyetlen függvényhívással lefuttathatunk egy olyan algoritmust, amely véletlenszerűen zsákutcákkal bővíti a térképet, mindaddig, amíg terem rendelkezésre állhat. Ezeken

kívül véletlenszerűsíthetjük a termek paramétereit, végül legeneráltathatjuk azt az adatszerkezetet, amelyet a program fel tud használni a játék során.

A StructuredMaze implementálja az olyan funkciókat, mint a főtermek véletlenszerű kijelölése, valamint azok összekötése útkereső algoritmussal létrehozott vonalon.

Mivel a GridMaze ősosztály implementálja a labirintussal kapcsolatos fontos követelményeket, újabb labirintus algoritmusokat könnyedén hozzáadhatunk a szoftverhez, öröklés segítségével. Ilyen például a DenseMaze algoritmus, amely az összes teremlehetőséget kihasználva alkot meg egy labirintust.

6 Díszítő elemek

Míg a játékban szereplő termek és folyosók szerepe, hogy bizonyos minimális mennyiségű adatból véletlenszerűsítéssel értelmes játékteret generáljon a program, addig a díszítő elemek gondoskodnak arról, hogy a pálya esztétikusabb és ingergazdagabb legyen. Ezért ezek esetében mindenképpen kell valamiféle alakbeli vagy színbeli változatosság.

6.1 Mesh objektumok

Ha a játékba dísz szerepét betöltő, bonyolultabb alakzatokat szeretnénk elhelyezni, az alakzatokat létre lehet hozni programkóddal (pl.: szabályszerű formák, véletlen generálás) is. Ez a megoldás azonban nehézkes, mivel minden egyes alakzatnak külön kódot kell írni, és többletmunka árán lehet őket csak egységesen kezelni.

Érdemesebb tehát inkább egy előre elkészített szerkesztővel létrehozni a különböző alakzatokat. Az egyik legismertebb szerkesztő a 3D Studio Max.

A DirectX támogatást is ad nekünk előre elkészített alakzatok tárolására, kezelésére. A Mesh osztály lehetőséget ad a programunkban ezen alakzatok reprezentálására, továbbá előre elkészített algoritmusokkal egységesen tudjuk őket használni, a fájlból betöltéssel kezdve, különböző beépített transzformációkon keresztül, egészen a kirajzolásig.

A probléma csupán az, hogy a 3D Studio Max és a DirectX egymástól függetlenül lettek kifejlesztve, így nincs közöttük semmiféle szabvány vagy egységes formátum. Azonban a mesh (háromszögekből összeállított test) szemlélet adja azt a lehetőséget, hogy a két rendszert együtt használjuk.

A 3D Studio Max alap formátuma a .max állomány, ami kizárólag saját felhasználásra tartalmaz adatokat, amelyeket más szoftverek, lehet, hogy nem is tudnak értelmezni. Azonban lehetőség van .3DS formátumú fájl exportálására, amely már hasonlít más rendszerek állományaira.

A DirectX kizárólag egy saját .x formátumot fogad el, amely speciálisan ehhez a rendszerhez készült el. Szerencsére az Interneten megjelent, és széles körben elterjedt

egy conv3ds.exe program, amely képes .3DS állományokat .x formátumra átkonvertálni, anélkül, hogy különösebben belemélyednénk a két szükséges formátum tanulmányozásában.

6.2 Fénykezelés

A DirectX sajnos egyszerre csak nyolc fényforrással tud számolni. Mivel az egész pályát be kell világítani, ehhez sokkal több fényre van szükség. Fények származnak a termek minden olyan falára elhelyezett lámpákból, ahol nincsen átjáró, ezenkívül a fáklyából és némely egyéb tárgyból.

Mivel a fáklya mozgó tárgy, ezért fontos, hogy mindig belevegyük a számításokba a tárgyak renderelésekor. A termekben elhelyezett tárgyak és a statikus lámpák azonban mindig ugyanabban a teremben vannak. Úgy állítottam be a fényerejüket, hogy jelentős hatással csak abban a teremben bírjanak, amelyikben éppen szerepelnek. Így megoldható, hogy egy fény csak akkor legyen bekapcsolva, amikor a hozzá tartozó termet jeleníti meg a program, előtte és utána kivesszük a fényforrásokat tartalmazó felsorolásból.

Így egy teremben legfeljebb három statikus lámpáról kell gondoskodnunk, egy szimbólumról, és a fáklya fényéről, ez öt darab összesen. Így mindig bennmaradunk a nyolcas határon belül, és a későbbi fejlesztések során további három fényforrásról gondoskodhatunk.

A statikus lámpák esetében alkalmazott fényforrásoknál továbbá arra kellett ügyelnem, ha a fényt valóban odahelyeztem, ahol a lámpa volt, a fal közelébe, akkor a fényszámításoknál a fal normálvektorai nagy szöget zártak be a fénysugarakkal, ezért azon a falon, amelyik közelében volt a fény, alig látszott belőle valami, viszont a szomszéd falat túl erősen világította meg. Ezért a fényeket kicsit eltávolítottam a faltól, aminek következményében nem kevésbé élethű a megvilágítottság iránya, de ez a játékban nem feltűnő.

A DirectX háromféle fényforrást használ: Directed (irányított), Point (egy pontú forrás), valamint Spot. Mivel maga a játék cselekménye belső térben folyik, ezért az irányított fényforrás szóba sem jöhetett, a falon elhelyezkedő lámpák és a fáklya fénye viszont inkább a pont típusú fényforráshoz hasonlít. Így a megjelenítés során ezeket

használtam. Egy külön osztály (PointLight) gondoskodik arról, hogy a megfelelő paraméterek el legyenek tárolva, így egyetlen függvényhívással kapcsolni lehet a fényforrást.

Ebben maga az Engine osztály is támogatást nyújt, amelyik számon tartja, hogy a nyolc fényforrásnak szánt helyek közül melyik üres, és melyik áll használat alatt, így a fényforrás bekapcsolásakor keres egy üres helyet, ahová bemásolja a paramétereket.

6.3 Menü

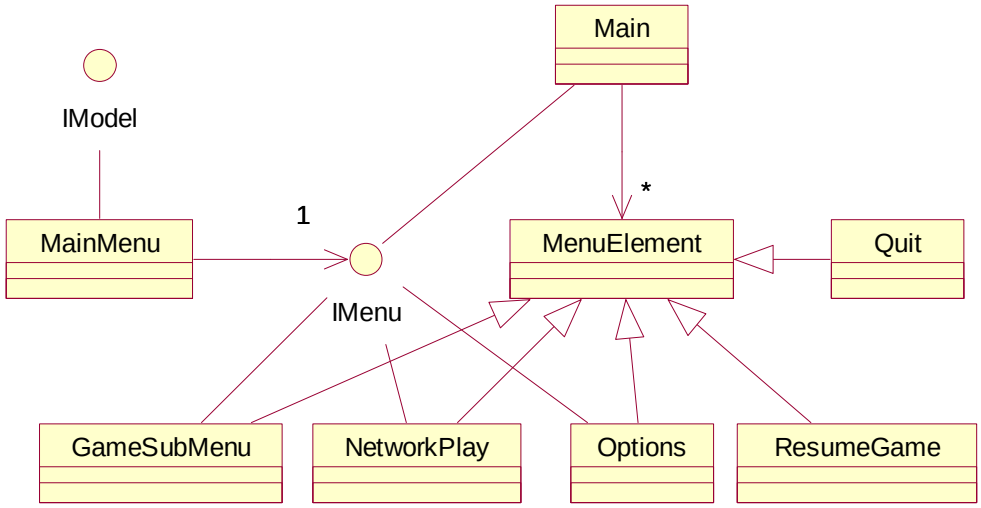
A menürendszerre mondhatjuk, hogy nem magának a konkrét játéknak a díszítő eleme, hanem az alkalmazásnak és a forráskódnak a szépségét adja. Ennek segítségével a felhasználó be tudja állítani az általa fontosnak vélt opciókat, de a tesztelésben is fontos szerepe van, így tudunk használni külön-külön minden olyan fontos funkcionalitást, amit éppen implementáltunk, izolálva a többi zavaró tényezőtől.

A menürendszer megvalósításához a következő szempontokat vettem figyelembe:

- **DirectX-es megjelenítés (nem kell a prezentációt az ablakos móddal váltogatni)**
- **Játék közben is lehessen menüt használni**
- **Elég jól skálázhatónak kell lennie**
- **Programkódban átlátható, bővíthető legyen**
- **Jól nézzen ki**

A szoftver architektúrája lehetővé tette, hogy az időközben megírt grafikus menüt integrálni lehessen a játékkal. Míg a motor folyamatos működés alatt áll, az állapotkezelés bevezetésével a különböző modellek cserélhető váltak. Így a menü megjelenítéséhez nem volt másra szükség, mint az IModel interfész implementálására. Ezzel együtt az új modell architektúráját is meg kellett tervezni, úgy, hogy az minél könnyebben bővíthető legyen a fejlesztés során.

Az osztályhierarchia a Menu névtérben található, amelynek egy részét a következő ábra szemlélteti, csak hogy érthető legyen a felépítése.



15. ábra: Menüelemek osztály-hierarchiája

A MainMenu osztály az IModel interfész implementálásával teszi lehetővé a grafikus integrációt. Ez az osztály felelős tehát a megjelenítésért: A menü elemei három dimenziós szövegalakzatok, amelyek különféle mozgó fénypontokkal van véletlenszerűen megvilágítva. Mivel a háttért képző textúrázott négyzetre ezek a fények nem hatnak, olyan, mintha a betűk mintája lenne színváltó textúra. A megjelenítést kameramozgatással próbáltam érdekesebbé tenni.

A menüpontok egy ősosztályból, a MenuElement osztályból származnak, ahol implementálva vannak a menüelemek általános grafikai és menükezelési rutinjai. Azok az elemek, amelyek almenük, az IMenu interfészt implementálják. Azt, hogy melyik menüben vagyunk éppen, a MainMenu osztály referenciával tárolja, az objektumok referenciáinak cseréjével barangolhatjuk be a teljes teret.

A menü szoros kapcsolatban áll az állapotkezeléssel. Nem csak azért, mert az architektúrája hasonló alapokra épült, hanem mert a legtöbb állapotváltás innen érhető el.

7 Ütközésdetektálás

7.1 Algoritmus

Az ütközésdetektálás, mint fogalom nem csupán a testek egymásba érésének vizsgálatát jelenti, hanem lefedi a fizikai térben található testek közötti kölcsönhatásokat. Feladatai a testek közötti érintkezések felfedezése, annak jellemzőinek kiszámolása, valamint a visszahatás irányítása.

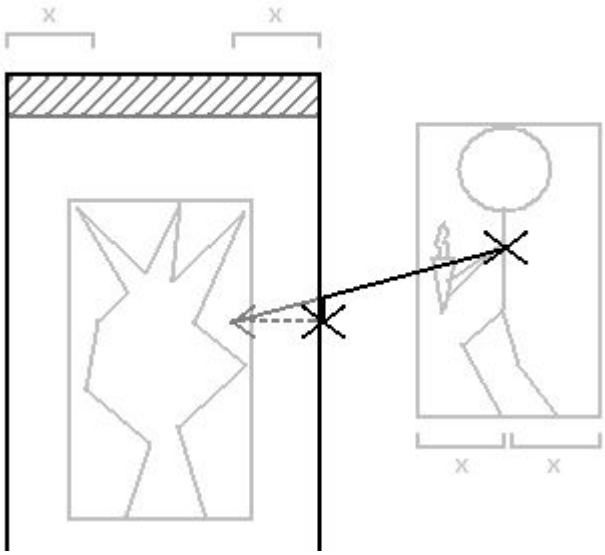
Általában a testek grafikai megjelenítése minél részletesebb, az ütközések vizsgálatánál inkább arra törekszünk, hogy minél egyszerűbb modelleket kezeljünk, minél kevesebb számolás és erőforrás felhasználása mellett. Ilyen modellek például a befoglaló gömbök, a koordináta-tengelyekkel párhuzamos élű befoglaló téglatestek, és ezek finomított illetve kombinált változatai.

A játékban befoglaló téglatesteket használtam, minden testhez egyet, hogy egyszerű, gyors művelet legyen az ütközésvizsgálat. Az alkalmazás szempontjából az vált fontosság, hogy a játékos beleütközzön a játéktérbe. Olyan algoritmust akartam megvalósítani, amelynél a játékos nem tud belenyomódni a pályaelemekbe, hanem a leendő pozíciója úgy kerül kiszámításra, hogy a karakter a fal felülete előtt megálljon. Ennek menete a következő:

1. Szükséges transzformációk

Egyrészt fontos, hogy minden test pozícióját világ-koordinátákban értsük. Ehhez a világ-transzformációt lefuttatjuk a szükséges helyeken, és úgy alakítjuk ki a befoglaló téglatestek paramétereit.

A második transzformáció egy egyszerű művelet: A tárgyak befoglaló téglatesteit megnöveljük akkora kiterjedéssel, mint amekkora a játékos (vagy más ütköztetni kívánt alakzat) kiterjedése. Ennek eredménye az lesz, hogy az ütköző testeknél nagyobb téglatestekkel kell számolnunk, ugyanakkor az ütköztetni kívánt alakzatot elég egyetlen ponttal reprezentálni. Ezt szemlélteti az alábbi ábra.



16. ábra: Ütközés-detektálási algoritmus szemléltetése

2. Az ütközés észrevétele

Ha arra hagyatkoznánk, hogy ütközés során a mozgás közbeni leendő pozíció benne van a test belsejében, sok esetben (pl.: túl vékony falak) az algoritmus hibásan futna le. Ugyanakkor az aktuális és a leendő pozíciót összekötve egy irányított szakaszt kapunk. Azt kell megvizsgálnunk, hogy létezik-e olyan oldala a befoglaló téglatestnek, amelyik metszésben van a szakasszal. Ha igen, akkor a két test ütközésben van.

3. Az ütközés feloldása

Ha ütközést találtunk, akkor általánosságban az a teendő, hogy a mozgás irányított szakaszát módosítsuk. Itt többféleképp járhatunk el. Ha valami hozzátapad a falhoz, akkor az irányított szakaszt arányosan meg kell rövidíteni. Ha viszont a játékost szeretnénk, ha a fallal ütközve az iránytól függően csússzon a fal mellett, akkor a megfelelő koordinátákat kell csak átírnunk (ez látható az ábrán). De előfordulnak más esetek is, például a fáklya megpattan a falon, ekkor a mozgás célpontját a befoglaló téglalap oldalára kell tükröznünk.

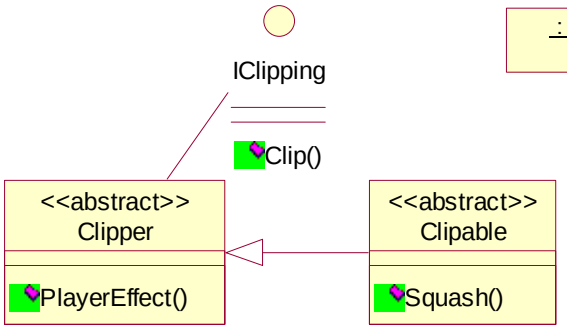
Ezen kívül léteznek kivételes esetek is. Például a lépcső. Ha szigorúan vesszük az ütközést, akkor a játékos képtelen lenne felmászni a lépcsőfokokra, hiszen

előremenetel közben az egyes fokokkal ütközik. Ezért az ütköző testek befoglaló téglatestében létrehoztam egy tűrési sávot. Ha a mozgás végpozíciója e fölött maradna, akkor úgy módosítjuk a mozgás irányát, hogy a játékos felkerüljön a tárgy tetejére.

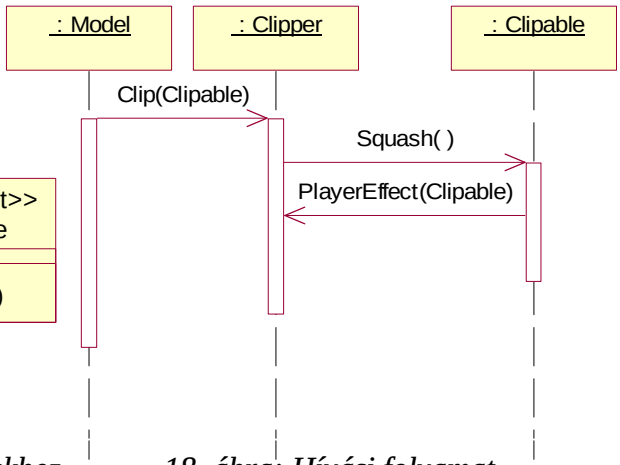
Végül foglalkoznunk kell a mozgás megváltozásán kívül az egyéb kölcsönhatásokkal is. Például ha egy töltény eltalál egy játékost, akkor a töltény eltűnik, a játékos életereje lecsökken.

7.2 Implementáció

Az implementált folyamatot az alábbi két ábra szemlélteti:



17. ábra: Ősosztályok ütközésekhez



18. ábra: Hívási folyamat

A Clipable osztály reprezentálja azokat a tárgyakat, amelyek mozognak, és a Clipper osztály példányaiban ütköznek. A Model adja ki a parancsot az ütközés vizsgálatára, amit a Clipper osztály Clip függvénye végez el, aszerint, hogy mi a két test jellemzői. Ha ütközés történt, akkor az ütközés feloldására kerül sor a Clipable Squash függvényén keresztül. Ebben tárolódik az, hogy az adott tárgy speciálisan hogyan viselkedik ütközés közben. Két fajtája van, a második akkor hajtódik végre, ha a Clipper példány rendelkezik olyan tulajdonsággal, ami hat a vele ütköző tárgyakra. Ebben az esetben meghívódik a PlayerEffect függvény.

8 Vetett árnyék

8.1 Árnyékok általánosságban

A vetett árnyék problémáját az inkrementális képszintézis hozta magával. Más megjelenítő módszerekben ez a probléma esetleg fel sem merül. A globális és lokális illuminációs algoritmusokban valamint a sugárkövetéses módszerben az adott tárgy adott pontjába érkező fényrészecskék energiáját számolják ki, azonban a fényből a testek legtöbbször valamennyit elnyelnek, egymásra a tárgyak kölcsönhatással vannak. Értelemszerűen oda, ahova árnyék vetül, kevesebb fényrészecske energiája jut, így a végső képen is sötétebb foltot kapunk.

Az inkrementális képszintézisben a tárgyakat reprezentáló alakzatok részei egymástól függetlenül kerülnek kiszámolásra, nem jelenik meg közöttük az efféle kölcsönhatás. Viszont fontos, hogy a szoftverünk továbbra se vesződjön túl sokat az ilyen grafikai megoldásokkal. Bár az inkrementális képszintézis nem az efféle problémák megoldására született, mégis léteznek – viszonylag – gyors algoritmusok a vetett árnyékok megjelenítésére is. Elméleti alapon ezeket a módszereket két csoportba lehet osztani:

Statikus módszerek Ebben az esetben egy Off-Line módon, előre kiszámított árnyékkal dolgozunk (pl. egy adott tárgy textúrájában egy újabb rétegre ráfestjük az árnyékot). Mivel fizikai szimuláció futási időben nincs, ezért ez egy mindenképp gyors és egyszerű megoldás. Fontos azonban tudni, hogy a módszer csak statikus fényforrásokra alkalmazható, és nincsen kölcsönhatással újonnan érkező szereplőkkel.

Dinamikus módszerek Ha a fényforrások vagy az árnyékban megjelenő tárgyak időközben változnak, érdemes a fizikai szimulációt futási időben, On-Line végezni. Ugyanakkor ez a megoldás több erőforrást is igényel.

A dinamikus módszerek közül a két legelterjedtebb algoritmus a következő:

Shadow Map Ez a módszer a fényforrás útjába kerülő alakzatot leképezi egy adott síkra, és a sík felületére rajzolja elsötétítve. Előnye, hogy mindössze egy plusz leképezést kell végrehajtani, amire például mesh objektumok esetében már létezik előre beépített metódus. Hátránya viszont, hogy általában csak síkfelületre lehet az árnyékot vetni, továbbá mindig a teljes árnyékot ki kell rajzolni.

Shadow Volume Ez az árnyék testek módszere. Minden alakzathoz készítünk egy testet, amelynek elülső felülete a tárgy azon része, amely a fény szemszögéből látható, oldalainak egyenesei a fényből indulnak és onnan széttartva a végtelenben haladnak. A képernyő minden egyes pixeléhez számon kell tartani azt az információt, hogy a virtuális világbeli megfelelője ezen az árnyéktesten belül helyezkedik-e el. Több erőforrást és számítást igényel, azonban ideális folyton változó térbeli árnyékok generálásához, továbbá a testek saját magukra is vetnek árnyékot.

A Shadow Volume algoritmust választottam, amelyhez kiválóan használhatóak a DirectX renderelési algoritmusai, valamint a Stencil Buffer.

A Stencil Buffer egy virtuális tár, amely a képernyő minden egyes pontjához tárol információt valamilyen szám értékeként. Fizikai elhelyezkedését tekintve a háttértár, azaz a Z-buffer része. Minden egyes pixelhez egy közös érték tartozik, amelynek valahány bitje a Z-buffer-beli értéket írja le, a többi a Stencil Buffer-é.

A kettő közötti megfelelő megosztáshoz a DirectX többféle formátumot is támogat. A legtöbb videó kártya manapság már támogatja a D24S8-as elosztást, ami azt jelenti, hogy a háttértár minden egyes eleme egy double word, azaz egy 32 bites szám, aminek 24 bitje a Z-buffer-hez tartozik, és 8 bitet biztosít a Stencil Buffer-nek. (Megjegyzés: Bár sok példaprogramban a Shadow Volume algoritmust egy bites Stencil Buffer segítségével valósítják meg, egy komplex vagy több test esetében az árnyékok kiszámításához ennél több értékre van szükségünk. Ez a bitelosztás ebből a szempontból az egyik legelőnyösebb.)

A Stencil Buffer olvasása és írása mindig valamilyen alakzat renderelésekor történik, gyakorlatilag majdnem hardware szintű programozással. Lehetőség van törlésre, ahol konkrét értéket is meg tudunk adni a Buffer minden egyes elemének, valamint meg lehet adni, hogy renderelés közben milyen feltételnek kell egy pixelre teljesülnie (referencia érték, maszk, a Buffer aktuális értékének és egy feltétel függvénynek a felhasználásával), és – attól függően, hogy a feltétel teljesült-e vagy sem – milyen érték kerüljön vissza a Stencil-be (nullázza, növelje, csökkentse).

A Shadow Volume módszer ötlete az, hogy a Stencil Buffer-ben tároljuk, az adott képen melyek azok a pixelek, amelyek árnyékosak, és melyek azok, amelyek nem. Először legeneráljuk a sötét képet, majd utána minden egyes fényforrásra nézve elkészítjük a megfelelő Stencil Buffer értékeket, és ahol megengedett, ott egy újabb renderelés segítségével, Blend funkcióval fényességet adunk az egyes pixeleknek.

A Stencil Buffer értékét ezután úgy kapjuk meg, hogy először elkészítjük az árnyéktesteket, majd utána, az első renderelés után előállt mélységi információ segítségével rendereljük le az árnyéktest oldalait. Ez a renderelés sem a képernyőn megjelenítendő képet, sem a Z-buffer-t nem szabad, hogy megváltoztassa, csupán a Stencil Buffer változik.

Mivel nekem jelenleg csak egy fényforrásra nézve kell vetett árnyékot szerkesztenem, a következőképpen programoztam le az algoritmus folyamatát:

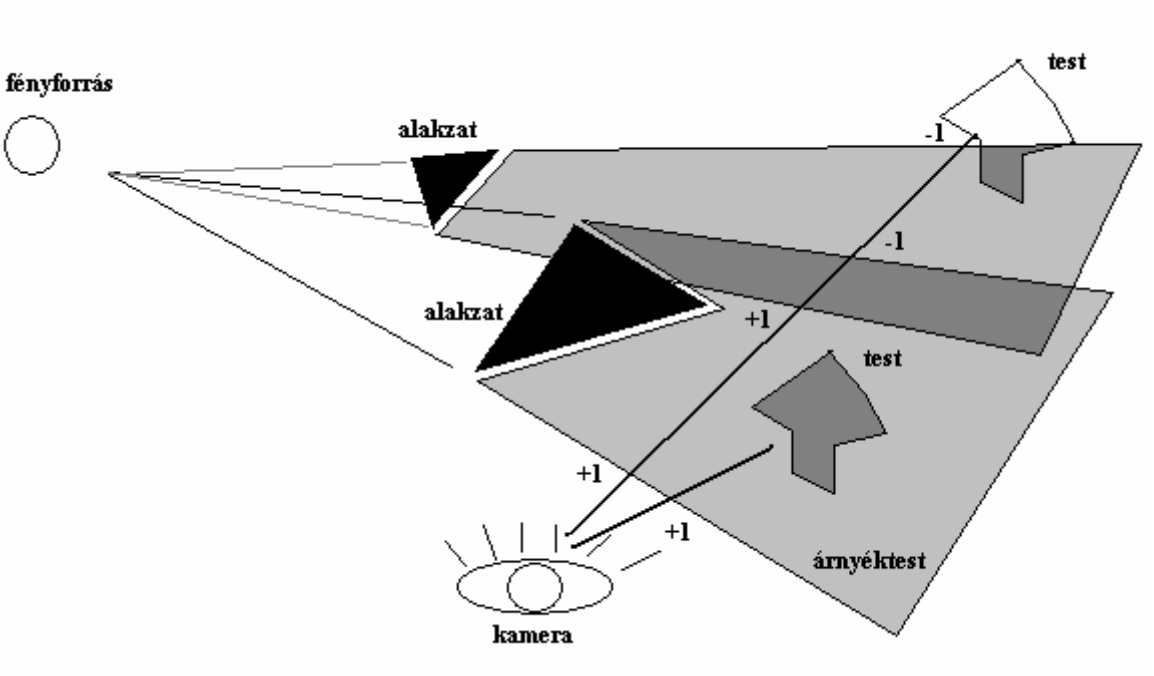
1. **Renderelem a megjelenítendő képet a fáklya fényével együtt.**

2. **Letiltom a képernyőre és a Z-buffer-re való írást, minden alakzatra nézve renderelem az árnyéktestet.**

3. **Megállapítom a Stencil Buffer-re vonatkozó feltételt, engedélyezem a képernyőre és a Z-buffer-re való íratást, törlöm a Z-Buffer-t, és újra renderelem a képet, ezúttal a fáklya fénye nélkül.**

A Stencil Buffer gyakorlatilag azt az információt fogja tárolni, hogy egy adott képpont világbeli megfelelője hány árnyéktesten belül helyezkedik el, átfogalmazva, hány árnyéktest oldalát kell renderelni a kamera és az adott test között. Tehát mindig,

mikor az árnyéktest oldalait rendereljük le, a pixelhez tartozó Stencil értéket eggyel meg kell növelnünk.



19. ábra: Árnyéktestek használata

Ezt a kapott számot hasonlítjuk össze a Stencilben felprogramozható függvények segítségével egy referenciaszámhoz, és amennyiben a referencia számnál többet kapunk, azaz volt árnyéktest széle a kamera és a test között, akkor ott az árnyékos pixelt kell megjelenítenünk a képernyőn.

Ezzel a megoldással együtt azonban felmerül két probléma:

A. Mi történjen, ha a test és a kamera között ugyan van egy árnyéktest, de a test nincs benne?

Ez azt jelenti, hogy az adott árnyéktestnek nem csak a felénk eső oldalát látjuk, hanem a test nem takarja el az árnyéktest túlsó lapját. Tehát amikor a túlsó lapját rendereljük, akkor a pixelhez tartozó Stencil értéket csökkenteni kell. Azt, hogy most éppen innenső, vagy túlsó lapot látunk, a következőképpen tudjuk megkülönböztetni. A testet felépítő háromszögeknek a DirectX kiszámol egy normális vektort, ami ha a kamera felé áll, akkor a háromszöget megjeleníti, egyébként nem. A normális vektor irányában azonban az is szerepet játszik, hogy a kiszámításához milyen sorrendben

vesszük fel a háromszögek csúcspontjait. A megfelelő sorrendet (Cullmode) renderelés előtt megadhatjuk.

B. Mi történjen, ha mi magunk is egy adott árnyéktesten belül helyezkedünk el?

Ha a kamera szintén egy adott árnyéktesten belül van, az azt jelenti, hogy minden lap, amit az árnyéktestből látunk, „túlsó lap”-nak számít, vagyis nem történik meg egy innenső lap megjelenítése. Következésképpen a Buffer-ben tárolt minden egyes referencia értéket eggyel meg kellene növelni, amit egyszerűbben úgy érhetünk el, ha magát a referenciaértéket csökkentjük eggyel.

Az árnyéktestek leképezését megpróbáltam egységessé tenni, amit úgy próbáltam elérni, hogy mivel minden egyes test a renderelés folyamán háromszögekre lesz bontva, írtam egy metódust, amely egy adott háromszöghöz készít egy csonka gúla alakú testet, és rendereli.

Természetesen csak azoknak a háromszögeknek kell árnyéktestet készíteni, amelyek felszíne a fény szemszögéből látható.

Az algoritmus eredendően végtelen hosszú testekkel dolgozik, amit egyszerűen úgy érhetünk el, hogy a háromdimenziós vektort átírunk homogén koordinátás alakra, és a homogén (negyedik) koordinátát nullára állítjuk. De tekintve, hogy az így kapott négydimenziós vektorokat nehezebb, körülményesebb kezelni, továbbá, hogy már korábban lekorlátoztam a megjelenítendő tér nagyságát, inkább csak elegendően hosszú testeket gyártok.

Egy árnyéktest teljes megjelenítése tehát a következő pontokból áll:

1. **Megvizsgáljuk, hogy a fényforrás a háromszög síkjának megfelelő oldalán van-e. Amennyiben nem, kilépünk az algoritmusból.**

2. **Létrehozzuk a csonka gúlát.**

3. **Beállítom, hogy a Cullmode órajárással ellentétes legyen, és hogy rendereléskor a Stencil értéke növekedjen.**

4. **Renderelem az árnyéktestet.**

5. **Beállítom, hogy a Cullmode órajárással megegyező legyen, és hogy rendereléskor a Stencil értéke csökkenjen.**

6. **Újból renderelem az árnyéktestet.**

7. **Megállapítom, hogy a kamera az árnyéktesten belül van-e. Amennyiben igen, csökkentse a referencia értékét eggyel.**

Azt, hogy a kamera az árnyéktesten belül van-e, a következőképpen lehet egyszerűen megállapítani. Az árnyétest minden oldalára megnézzük, hogy a kamera pozíciója az adott sík megfelelő oldalán van-e.

Láthatjuk tehát, hogy az 1. és 7. pontban ugyanazt a műveletet kell végrehajtani: Meg kell állapítani, hogy egy adott pont a síknak melyik oldalán van. Ha a síkot „a”, „b” és „c” pontokkal adjuk meg, valamint a vizsgálandó pont „p”, akkor kiszámítjuk a következő értéket:

$$((\underline{b}-\underline{a})\times(\underline{c}-\underline{a}))\cdot(\underline{p}-\underline{a})$$

A pontok megfelelő sorrendjének függvényében a kiszámított érték előjele adja meg, hogy a „p” pont a megadott sík melyik oldalán helyezkedik el.

8.2 Egyszerű alakzatok árnyéka

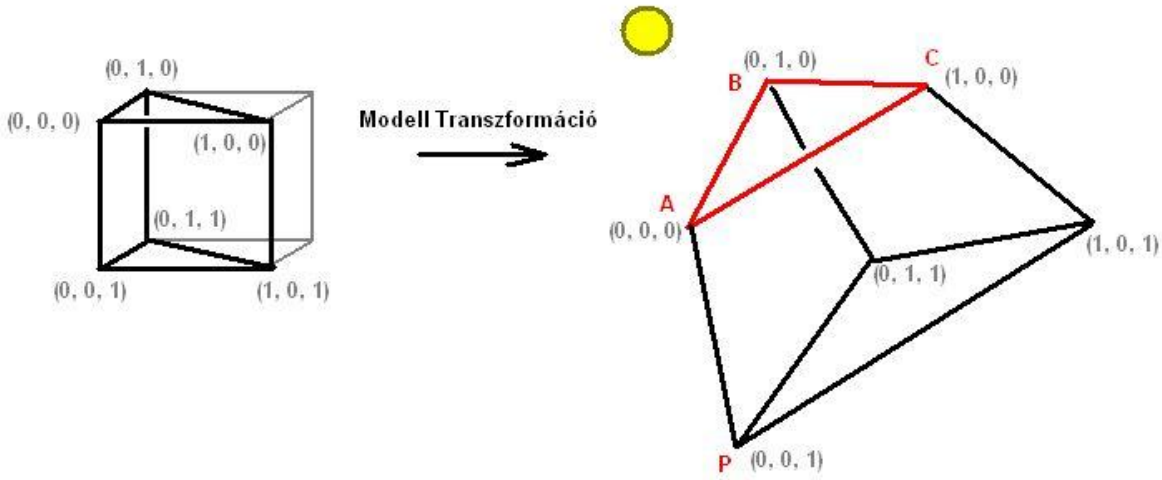
Az Engine osztályt kibővítettem egy egyszerű, általános algoritmussal, ami egyetlen háromszög árnyéktestét jeleníti meg. Olyan egyszerű testeknél, mint a terem falait alkotó téglalapok ez teljesen megfelelő: Bár megjelenítés szempontjából sok apró háromszögből tevődnek össze, geometriáját tekintve két háromszög elegendő a leírásához.

A dinamikus árnyéktestekre jellemző, hogy képkockáról képkockára változik az alakjuk. Másképp mondva minden új rendereléskor el kell mozgatni a vertex-eiket. Általánosan elfogadott szabály viszont, hogy az algoritmus során minél kevesebbszer szabad csak DirectX metódusokat hívni, mivel azok tényleges futtatása több índirekción keresztül történik, ezért lassítja a programot. Éppen ezért a helyett, hogy egy adott

alakzat vertex-einek koordinátáit ténylegesen elmozgatnák, Vertex Shader-eket hívnak segítségül, megspórolva a videokártya memóriájának megnyitását, átírását és lezárását.

Ebben az esetben azonban egyszerűbb megoldást próbáltam ki érdekességképp, kihasználva, hogy a háromszög, mint árnyékot vető test, csak kétdimenziós, de a DirectX háromdimenziós homogén transzformációkat alkalmaz.

Az algoritmus alapja az, hogy a motor lefoglal egy háromszög alapú egység élű hasábnak megfelelő alakzatot a videokártya memóriájában. Ez lesz az árnyékteste minden külön árnyékot vető háromszögnek. Ezután kiszámol egy olyan transzformációt, amely az alakzatot a megadott háromszög árnyéktestébe viszi át. Ezt megadva, mint modell transzformációt, a szokásos renderelési folyamat alkalmas lesz az árnyéktest kezelésére.



20. ábra: Árnyéktest képzése egyetlen háromszögre

A keresett transzformáció a következőképp áll össze. Ha a háromszög koordinátái rendre A, B és C, akkor a (0, 0, 0) koordinátát eltoljuk az A csúcs helyébe. A transzformációnak ezután az (1, 0, 0) koordinátára B - A vektort kell adnia, a (0, 1, 0) koordinátára a C-A vektort. Azaz a homogén transzformáció mátrixának felső 3x3-as részmátrixának első sora megfelel B - A, a második sora a C - A koordinátáinak. Így ha az A pont homogén koordinátáit írjuk a mátrix legalsó sorába, akkor egy olyan transzformációt kapunk, amely a hasáb felső háromszögét az árnyékot vető háromszögbe viszi át.

Ha ezek után meghatározzuk, hogy mekkora legyen az árnyéktest palástjának hossza (d) – aminek nem kell végtelen nagynak lennie, csak „elég hosszúnak” –, továbbá, hogy melyek P koordinátái – azaz a hasáb fényforrástól távolabbi háromszögének csúcspontjai közül A párja hol fog elhelyezkedni –, akkor a hasáb két alapháromszögének hasonlóságát segítségül véve bebizonyítható, hogy a mátrixnak létezik egy egyértelmű kitöltése, amely a kívánt transzformációt adja. Képletek formájában az alább megtalálható.

$$\underline{\underline{M}} = \begin{bmatrix} Bx - Ax & By - Ay & Bz - Az & 0 \\ Cx - Ax & Cy - Ay & Cz - Az & 0 \\ Px * d - Ax & Py * d - Ay & Pz * d - Az & d\text{-}1 \\ Ax & Ay & Az & 1 \end{bmatrix}$$

$$d = \frac{|\ \text{fénypont} - A\ |}{|\ \text{fénypont} - A\ | + \text{palásthossz}}$$

$$P = (A - \text{fénypont}) * \left(1 + \frac{20}{|\ \text{fénypont} - A\ |} \right) + \text{fénypont}$$

21. ábra: Felhasznált képletek

Implementáció szintjén elmondhatjuk, hogy az árnyéktest alapjaira nincs is szükség, hiszen az egyiket takarni fogja az eredeti alakzat, a másik sohasem fog megjelenni, hiszen elméletileg valahol a végtelenben helyezkedik el. Ami fontos, az a palástja, amit tökéletesen leír egy hatelemű Triangle Strip. Ezt az alakzatot indításkor betehetjük a videokártya memóriájába, és a megfelelő transzformációt megszerkesztve mindig felhasználhatjuk.

8.3 Mesh-ek árnyéka

A mesh, mint grafikai objektum legfőképpen abban különbözik az egyszerű elemekből, hogy apró, különböző méretű, formájú, és elhelyezkedésű háromszögek sokaságából áll. Általánosságban véve az alakjuk – ha pontos árnyékot szeretnénk előállítani – nem modellezhető egy leegyszerűsített formával. Ugyanakkor értelemszerű, hogy háromszögenként sem állíthatjuk elő a vetett árnyékot.

A fent vázolt folyamat itt is megfelelő lesz, azonban két fontos kérdés merül fel:

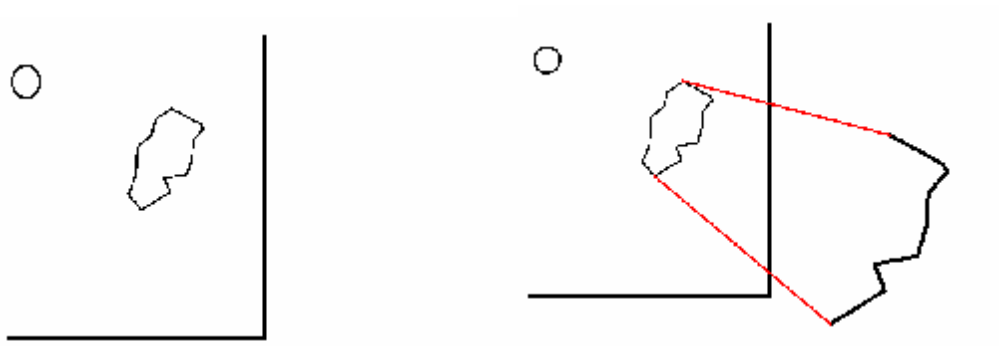
1. **Hogyan állítható elő (gyorsan) a mesh alakzathoz tartozó árnyéktest?**
2. **Hogyan állapítható meg, hogy a kamera az árnyéktesten belül van?**

Kissé elemezve a az első problémát rájöhetünk, hogy a mesh test árnyékteste sok hasonlóságot mutat egy egyszerű háromszögével. Nevezetesen mindkettő egy olyan csonka gúla, amely palástjának élei a fénypontban egymást metsző egyenesekre illeszkedik, és az alapjai – matematikai értelemben – hasonló poligonok. A különbség, hogy ebben az esetben az alap nem egyetlen háromszög, hanem a test határoló körvonala lesz. Továbbá szem előtt kell tartanunk, hogy a gúla alapja időben változik, attól függően, hogy a fény éppen melyik irányból világítja meg a testet, és ezzel együtt a palást formája és lapjainak a száma is módosul. Ezért egy olyan megoldás szükséges, ahol magát az árnyéktestet is valós időben ki tudjuk számolni.

Az algoritmust a DirectX SDK-ban található Shadow Volume példaprogram alapján készítettem el, amelyhez rendelkezésre állt részletes dokumentáció, és C++ nyelvű forráskód.

Az először is a régi mesh objektum mellett létre kell hozni egy újat – texture és material információ nélkül -, ami formára megegyezik az eredeti objektummal, azonban hozzávesz rengeteg nullaméretű háromszöget. Ezek szerepe, hogy – míg az eredeti háromszögek az árnyéktest alapjait alkotják, az új háromszögek közül azok, amelyek a test megvilágított felének határvonalain helyezkednek el, megnyúlva fogják alkotni az árnyéktest palástját. Ahol pedig nincs átmenet fényes és árnyékos fél között, ott az új háromszögek nullaméretűek maradnak, így nem jelennek meg a képernyőn. Végeredményben tehát kapunk egy mesh objektumot, amelyet ha a fénytől eltávolodva

megnyújtunk, akkor megkapjuk az árnyéktestet. A korábbi megfontolásokat figyelembe véve az árnyéktest megjelenítésekor ebben az esetben sem kell az árnyéktest alapjait – azaz a mesh eredeti háromszögeit – kirajzoltatni, csupán a palástot, ezzel ellentétben a valós játéktér megjelenítésekor a palást lesz láthatatlan. Ezért a megvalósításban külön tároltam el az objektumot, és külön az árnyéktestet alkotó, változó méretű háromszögeket.

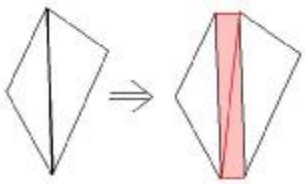


22. ábra: Árnyéktest transzformációja

A palástot alkotó háromszögek előállítása is azonban egy bonyolult feladat. Először elemezni kell az objektumot, és kiszűrni az eredeti objektumok null-háromszögeit. (Ennek oka, hogy a palást elkészítésékor fontos, hogy az eredeti objektum háromszögei milyen irányba néznek. Null-háromszögek esetén ez nem megállapítható, hiszen a felületi normálisuk null-vektor.) Erre talán a legjobb a háromszög-egyenlőtlenség teljesülésének vizsgálata. Azaz, ha a két kisebb oldal méretének összege egyenlő a harmadik oldal hosszával, akkor a háromszög területe is nulla. Azért is kell ezt a szabályt használnunk, mert null-háromszögek nem csak úgy állhatnak elő, hogy a háromszög két pontja egyenlő. Ha null-háromszöget találtunk, a háromszög felületének normálisát nem az oldalvektorokból számoljuk ki, hanem feltételezzük, hogy a vertex-ekben tárolt normálisok közelítőleg ezzel megegyeznek.

(Itt és a továbbiakban is érvényes, hogy a lebegő pontos (float) számok véges pontosságú ábrázolása miatt kritériumként érdemes nem a pontok pozíciójának vagy az oldalhosszak méretének megegyezését vizsgálni, hanem azt, hogy a különbségük legyen egy minél kisebb szám. Azonban mivel az általam használt objektumok nem használják ki a teljes pontosságot ebben a számábrázolásban, én mégis az egyenlőséget választottam.)

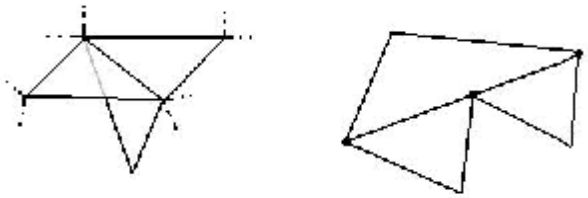
A következő lépés elméleti megfogalmazása egyszerű: Tegyünk minden két egymásra illeszkedő háromszög közé egy nullaméretű négyszöget. Ezek a négyszögek fogják alkotni a palástot.



23. ábra: Közbülső háromszögek képzése

Általánosságban elmondható egy helyesen definiált mesh objektumról, hogy a felületét egymáshoz pontosan illeszkedő háromszögek határolják. A mesh leírásában „adjacency” néven jelölték azt az adatszerkezetet, amely felsorolja, hogy az egyes háromszögekhez mely három másik háromszög illeszkedik teljes pontossággal. Plusz információként azt kell az objektumból kinyerni, hogy az illeszkedő háromszögek esetén mely csúcsok találhatóak ugyanabban a pozícióban. Az egyező két csúcspont segítségével definiálnunk kell két darab háromszöget. A DirectX vertex reprezentációja lehetővé teszi, hogy – bár csak két csúcspont koordináta áll a rendelkezésünkre – a négyszög egyes csúcsait megkülönböztessük. Az egyes vertex-ek normálvektorának megadjuk a hozzá tartozó háromszög felületi normálisát.

Az algoritmus feltételezi tehát, hogy az eredeti mesh objektum háromszögei egy zárt felületet adnak, méghozzá úgy, hogy minden háromszög a illeszkedik másik háromhoz, és ez az illeszkedés a csúcspontok egyezőségével fordulhat csak elő. Azaz, ha van egy levegőben lógó háromszögünk, vagy az alakzat felépítése olyan, hogy egy háromszög éle megegyezik két másik háromszög együttes élhosszával, az algoritmus megbukott. Szerencsére a legtöbb 3D formázó program (pl: 3D Studio Max is) megfelelő használatot feltételezve ilyen hibát nem generál.



24. ábra: Hibalehetőségek a mesh fájlban

Így már készen van az a grafikai objektum, amelyet megnyújtva megkapjuk az árnyéktest palástját. A „megnyújtás a következőket jelenti: Megvizsgáljuk a kapott négyszögekhez tartozó két háromszöget. Ha az egyik a készítendő árnyéktest felső lapjához tartozik (azaz a felülete a fényforrás felé néz), a másik az alsó lapjához (azaz a felülete a fényforrásnak háttal van), azaz a test megvilágított felének határvonalán helyezkedik el, akkor „szétrántjuk” a két háromszöget, és az alsó részt a végtelenbe visszük.

Azt, hogy a háromszög felülete milyen irányba néz a fényforráshoz képest, egy egyszerű skaláris szorzással meg tudjuk állapítani. A háromszög egy pontját és a fényforrást összekötjük, a kapott vektort szorozzuk a háromszög felületének a normálisával, és megvizsgáljuk, hogy az eredménynek milyen előjele van. A felületi normálisok viszont már ki vannak számolva, a null-háromszögek vertex-eiben tároljuk őket.

Így a feladat leegyszerűsödik a következőre: A vertex-et és a fényforrást összekötjük egy vektorral, és beszorozzuk a vertex normális összetevőjével. Az előjeltől függően a kapott vertex-et a helyén hagyjuk, vagy a megfelelő irányban a távolba (esetleg: nagyon messzire) visszük. A kapott vertex-ekkel kirakjuk az árnyéktest palástját.

A mesh alakzatok bonyolultsága miatt az árnyéktestek valós idejű számításához elengedhetetlenné válik a hardveres segítség. A palást háromszögeinek megnyújtása illeszkedik a Vertex Shader-ek problémakörére. A feladat a háromszögek pontjaiként végzi az amúgy elég egyszerű műveletet, függetlenül a többi pont állapotától. Az algoritmushoz tartozó vertex programot HLSL nyelven írtam meg, amelynek betöltéséhez és futtatásához a felügyelt DirectX a megszokott módon nyújt segítő kezet.

```
float4 pos = mul(inPos, World);
float3 normal = mul(inNormal, World);
float s = LightPos.w;
float3 l = float4(LightPos.xyz, 1);
float3 p = float3(pos.x / pos.w, pos.y /
pos.w, pos.z / pos.w);

if( dot ( p - l, normal ) >= 0){
    float len = length (p - l);
    p = (p - l) * s / len + p;
}

return mul( float4(p.xyz, 1), ViewProj );
```

25. ábra: Vertex Shader vetett árnyékhoz

Minden esetben, mielőtt a programot aktiválnánk, előtte a kódot fel kell paraméterezni, azaz megfelelő adatokat kell közvetíteni a grafikus kártyának. Mivel a saját Vertex Shader használata érvénytelenné teszi az alapértelmezett transzformációkat, a mesh világ-transzformációja (World) mellett szükség van a nézeti- és képernyő-transzformációkra is, amelyeket elég csupán egy szorzatban átadni (ViewProj). A fény pozícióját egy négydimenziós vektorban (LightPos) adom át, mert ennek a kezelése ebben a környezetben könnyebb, és kihasználom a negyedik koordinátát arra, hogy átadjam azt az értéket, amennyivel megnyújtani kívánom az árnyéktesteket. (Tervezési döntés eredménye, hogy nem a végtelenbe visszük a mozgatni kívánt pontokat, csupán elég messzire.)

Megjegyzendő, hogy a transzformáció során a vertex-ek másképp viselkednek, mint a vektorok. Ezért a világ transzformáció elvégzésekor a normál vektorok külön bánásmódban kell, hogy részesüljenek. Egyik megoldás, hogy a transzformáció során nem magával a világ transzformációs mátrixszal szorozzuk be, hanem inverz transzponáltjával. Egyszerűbb megoldás – lévén homogén koordinátákról szó – a negyedik koordinátát nullára állítjuk, és úgy végezzük el rajta a transzformációt.

Végezetül azt kell megállapítanunk, hogy a kamera pozíciója az árnyéktesten belül van-e, vagy azon kívül. Az árnyéktest geometriájából látszik, hogy minden pont, ami azon belül helyezkedik el, olyan egyenesre illeszkedik, amely áthalad a fényponton, és metszi a mesh alakzatot. A DirectX Mesh osztálya rendelkezik egy Intersect tagfüggvénnyel. Ennek paraméterül meg kell adni egy sugarat, amely a tér valamely pontjából egy megadott irányba indul, és visszatérésként megadja, történt-e metszés az

objektummal, és ha igen, akkor a kezdőponttól mekkora távolságra. A kezdőpontnak megfeleltetjük a kamera pozícióját, és az iránynak azt az egyenest, amely a fényforrással összeköti. Ha a metszés távolsága kisebb, mint a fényforrás és a kapott távolság közötti szakasz, akkor az árnyéktesten belül vagyunk.

Figyelnünk kell azonban arra, hogy a fényforrás és a kamera pozíciója a világ koordináta-rendszerben helyezkednek ez, azonban az Intersect hívás az objektum modell-terében végzi a számításokat. Ezért a függvény hívása előtt ezt a két pozíciót egy inverz transzformációval át kell számolni a modell koordináta-rendszerébe.

8.4 Szöveg árnyéka

A játék menüjében található háromdimenziós szövegek a Mesh beépített osztály segítségével elkészített alakzatok, így az árnyékuk ezzel a módszerrel megkonstruálható és megjeleníthető az árnyékuk. Problémát okoz viszont, hogy – mivel nagyon részletesek – rengeteg primitívből állnak, ezért az árnyékuk elkészítése idő- és tárigényes.

Közelről szemügyre véve a szövegalakzatokat azonban észre vettem, hogy ezek gyakorlatilag olyan hasábok, amelyek alap alakzata maga a szöveg, oldaléleik párhuzamosak, és hosszuk paraméterezhető. Ha tehát úgy állítjuk be a menüt, hogy a szövegek mindig az árnyékot vető fényforrás felé mutassák az alapjukat, az alakzatok saját maguk árnyékai lehetnek.

Közelebbről megnézve a feladat matematikai részét észrevehetjük, hogy ez az egyszerű háromszögek esetében készített árnyéktest-generálás speciális esete, így a megfelelő transzformáció könnyedén előállítható.

Ebben az esetben nem kell azzal foglalkozni, hogy a kamera pozíciója az árnyéktesten belül van-e, mivel a szöveg objektumok használata kizárólag a menürendszerre korlátozódik.

8.5 Carmack's Reverse (Depth-Fail technika)

A fent vázolt algoritmus kitűnő megoldás arra az esetre, amikor a játékteret valamilyen speciális nézetből tekintjük, mint például a menü esetében, vagy egy madártávlatból megjelenített játék során. Ezzel szemben az első személyű játékoknak további olyan problémákkal is meg küzdeniük, amelyek a grafikus megjelenítő környezetek implementációjából fakadnak, a puszta matematikai modellben nem merülnek fel.

Az egyik ilyen fontos probléma, hogy a matematikai reprezentációban a kamera egyetlen pontként szerepelt, ugyanakkor maga a képernyő a virtuális térben is megfelel egy – viszonylag – nagyobb kiterjedéssel rendelkező négyszögnek. Ekkor előfordulhat, hogy a stencil bufferben megrajzolt háromszögek, a kamerával az árnyéktest széléhez érve, beleütköznek a képzeletbeli kiterjedésbe, így annak – a közelebbi síkbeli vágásnak köszönhetően – olyan részei nem kerülnek megrajzolásra, amelyek a kamera pozíciója előtt helyezkednének el, és így zavart keltenek az árnyékok megrajzolásában.

John Carmack az ID Software grafikus motorjainak fejlesztője, és egyben annak az algoritmusnak a feltalálója, amelyet tudományos körökben Depth-Fail vagy Z-Fail néven emlegetnek. Itt az árnyéktesteket megjelenítő algoritmus a következőképpen változik meg:

Egy árnyéktest teljes megjelenítése tehát a következő pontokból áll:

1. **Megjelenítem a teret a fényforrással együtt.**

2. **Beállítom, hogy a Cullmode órajárással ellentétes legyen, és hogy rendereléskor a Stencil értéke növekedjen. (Ezzel az árnyéktest belső lapjai kerülnek renderelésre.)**

3. **Renderelem az árnyéktestet. A stencil buffer értékei akkor módosulnak, ha a z-teszt nem teljesül. (Az árnyéktest lapjait a testek takarják.)**

4. **Beállítom, hogy a Cullmode órajárással ellentétes legyen, és hogy rendereléskor a Stencil értéke csökkenjen. (Ezzel az árnyéktest külső lapjai kerülnek renderelésre.)**

5. **Újra renderelem az árnyéktestet. A stencil buffer értékei akkor módosulnak, ha a z-teszt nem teljesül.**

6. **Megjelenítem a teret a fényforrás nélkül. A képernyő ott rajzolódik felül, ahol a stencil buffer értéke nem nulla.**

Mint látható, az algoritmus abban különbözik a korábbi (Depth-Pass) algoritmustól, hogy az árnyéktest oldalait más sorrendben, más stencil értékmódosítással rendereljük le, és a Stencil Buffer-t akkor módosítjuk, ha a távolsági értékek kedvezőtlenek.

Előnye, hogy nem kell megvizsgálnunk, hogy a test az árnyéktesten belül van-e, az algoritmus során a képernyő stencil értékei megfelelően módosulnak minden esetben. Továbbá az árnyéktestek szélénél sem keletkezik zavar, mivel a rosszul módosult értékeket kijavítják a test gúlaformájának alapjai, amelyek a távolsági vizsgálat megfordításával váltak láthatóvá.

Ebből viszont következik, hogy ezúttal az árnyéktestnek teljesen zártnak kell lennie, azaz renderelni kell a gúla alapjait is, ami meghosszabbítja egy árnyéktest kirajzolását. (Az árnyéktest memóriabeli tárolását azonban nem kellett megváltoztatni, hiszen egy alap nem más, mint az eredeti alakzat fényforrás felé vagy azzal ellentétesen elhelyezkedő fele. A palástot úgy kapjuk, hogy ezeket a vertex-eket megfelelő módon indexeljük. Így ha a palástot akarjuk megjeleníteni, index-ekkel együtt renderelünk, az alapok esetében index-ek nélkül, mindkét esetben ugyanazon vertexek felett.)

Másik probléma, hogy az árnyéktest kisebbik alapjának felszíne egybeesik az árnyékot vető alakzat felszínével. Ekkor megjelenítés során z-fighting alakul ki. Ez a jelenség a renderelés során a pixelek távolsági értékeinek számolásakor felmerülő pontatlanságából fakad, és a következménye, hogy a két felszín kirajzolása keveredik, ahelyett, hogy valamelyik felszín elsőbbséget élvezne.

Elegánsan ezt a problémát Bias értékek bevezetésével oldják meg. Ez nem más, mint az egyes kirajzolt felületekhez rendelt rangsor. Mikor tehát a két felszín szinte ugyanabban a pozícióban helyezkedik el, és esély nyílik z-fighting kialakulására, akkor a teljes pontosság helyett ennek az értéknek a segítségével határozzák meg, hogy melyik felszín kerül tényleges kirajzolásra.

Szerencsétlenségre, a rendelkezésemre álló videokártya nem rendelkezik ezzel a technikával, ezért a kisebbik alapot úgy renderelem, hogy a távolsági vizsgálatok mindig kedvezőtlen eredményeket adjanak, hiszen a Bias értékekkel történő megvalósítás során is takarná az árnyékot vető alakzat az árnyéktestet.

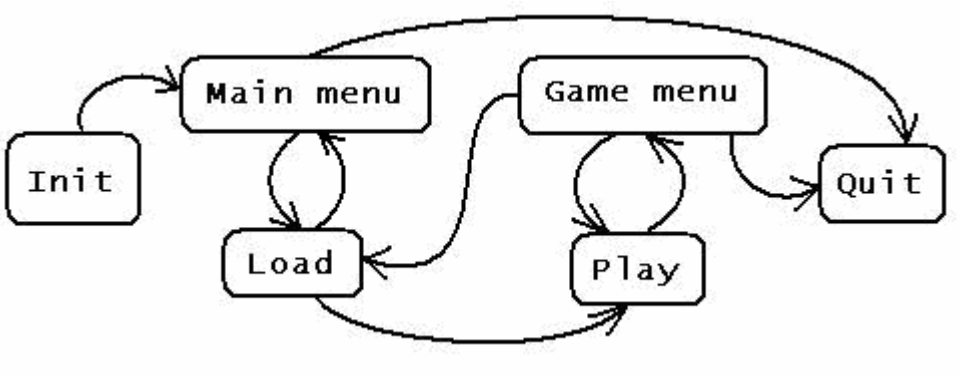
Végül érdemes megemlíteni egy újabb problémát, ami az első személyű megjelenítésnél jelentkezett: Úgy tapasztaltam, hogy az árnyéktest vertex-eit a projektív transzformáció utáni vágás megzavarja. Magyarázatot nem találtam, csak indirekt bizonyítékot. (A zavar akkor állt elő, amikor az árnyéktest a képernyő oldalán túlmutatott, és a vertex pozíciója esetenként átkerült a fényponttal összekötő egyenes másik oldalára; pont, mint az átfordulási probléma esetén.)

Ezért az árnyéktestek renderelésének idejére ezt a vágási műveletet kikapcsolva tartom.

9 Állapotkezelés

9.1 Az alkalmazás állapottere

Az alkalmazás állapotterét a következő ábra szemlélteti:

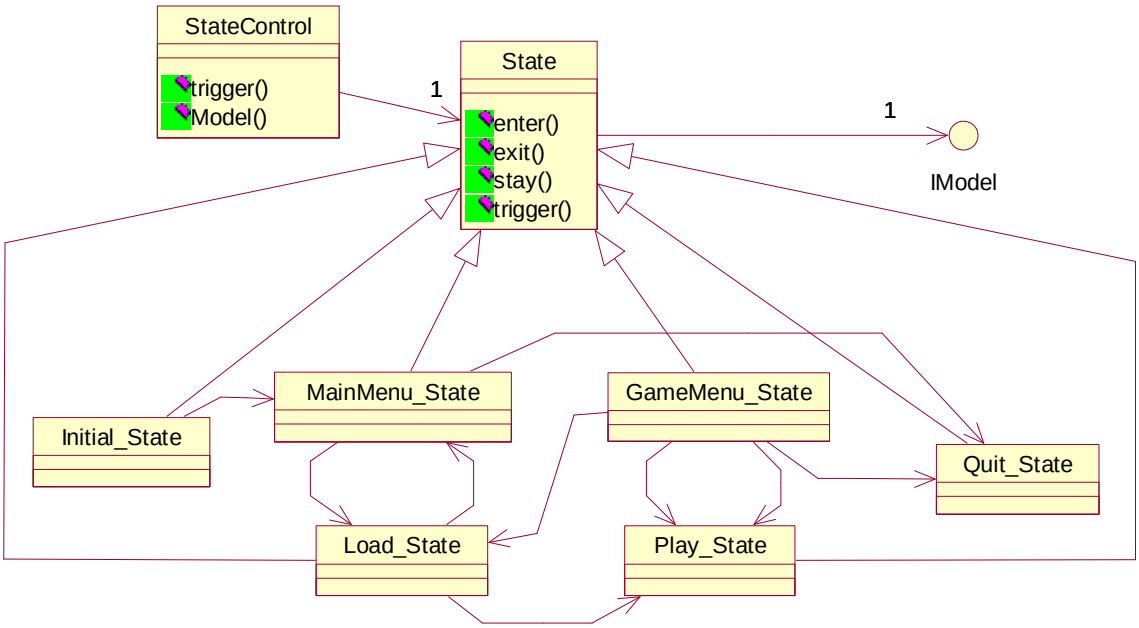


26. ábra: Állapottér

Az Init állapot jelzi, hogy elindul a grafikai motor, az irányítás utána a Main Menu-höz kerül át. A menü segítségével elindíthatjuk a játékot, a szükséges betöltések viszont hosszadalmasak lehetnek. Ezért a Load állapotban egy forgó felirat jelenik meg, és marad, amíg a játék el nem kezdődik. A játék során meghívhatjuk újra a menüt, de a Game Menu állapotban újra visszatérhetünk az eredeti játékhoz, vagy újat kezdhetünk. Kilépés során a Quit állapothoz kerül az irányítás, amely gondoskodik a modellek és a motor felszabadításáról.

9.2 Implementáció

Az állapotgép megtervezéshez felhasználtam a State Design Pattern-t. Minden egyes állapot egy osztálynak felel meg. Ebben definiáltam az állapotok közötti átmenetet, a be- és kilépéskor végrehajtódó funkciókat. Az aktuális állapotot egy példány jelenti, és a példányok közötti váltás jelenti az állapotváltást. Minden állapot rendelkezik egy IModel referenciával, ami azt határozza meg, hogy a motornak éppen melyik modellt kell megjelenítenie. Az architektúrát az alábbi ábra jeleníti meg:



27. ábra: Állapotok osztály-szerkezete

A megvalósításnál figyelni kellett arra, hogy minden állapot rendelkezzen valamilyen megjeleníthető modellel, azok is, amelyeknél a grafika nem fontos elem. Például az Initial és a Quit

Az állapotokhoz egy-egy JunkModel példány van rendelve, amely gyakorlatilag egy üres modell, amely kapcsolatot tart a motorral addig is, amíg nincs igazi modell a rendszerben. Másik példa a Load állapot, amikor a képernyőn egyetlen „Loading...” szöveg forog, tudatva a felhasználóval, hogy az alkalmazás továbbra is működik.

Másik fontos tényező a trigger, mint eseményt jelző függvényhívás, paraméterének megválasztása. A jelen alkalmazásban nincs semmilyen egységes forma, amely meghatározná, hogy ilyen esetben mit adhatunk át paraméterben, hiszen szinte minden állapotnak más-más objektumokra van szüksége a helyes működéshez. Ezért object típusú adatokkal dolgozok, amelyeket minden állapot maga értelmez.

Végül fontos probléma azon eset, amikor az állapotváltást kérő szál az állapotváltással saját magát blokkolná. Ebben az esetben egy új, ideiglenes szál jön létre, amellyel segít a teljes folyamat lebonyolításában.

10 Adatkezelés

Ha nem túl nagy mennyiségű adathalmazt szeretnénk tárolni, érdemes szabványos megoldásokra hagyatkozni. Esetünkben ugyanis nem az elmentés és visszatöltés sebessége, vagy az adat sűrűsége fontos szempont, hanem a fejlesztési idő. Ebben a részben írom le, hogyan használtam fel a .Net sorosítási és XML kezelő rutinjait.

10.1 Bináris sorosítás

A .Net Reflection modulja teszi lehetővég, hogy saját magunk definiálta osztályok példányait egy előre megírt algoritmus folyammá alakítsa, amelye szekvenciálisan ki tudunk íratni háttértárra, vagy át tudunk küldeni a hálózaton. Többféle megoldás közül választhatjuk például a bináris sorosítást és az XML alapú sorosítást is.

Az XML alapú sorosítás előnye, hogy az XML nyelv platform-független. Abban az esetben is, ha a kommunikáló felek más-más platformot igényelnek, a bennük szereplő adatstruktúrák könnyedén átadhatók. További előny, hogy az XML fájlok olvashatóbbak, redundánsak, ezért hiba esetén manuálisan, illetve automatikusan korrigálhatóak.

A bináris sorosítás ezzel szemben egy sokkal tömörebb leírást ad. Ugyan egyáltalán nem olvasható a felhasználó számára, ugyanakkor gyors adatkezelést tesz lehetővé, és kevesebb ideig tart a teljes adatmennyiség továbbítása. Éppen ezért ezt a megoldást használom a játék aktuális állapotának elmentésekor, visszatöltésekor, valamint hálózati kommunikációnál.

Független azonban attól, hogy melyik konkrét megoldást alkalmazzuk, a sorosítási folyamat egy elég fontos hátránnyal rendelkezik. Nevezetesen, ha az adatstruktúrában szereplő definícióknak bármilyen kis része megváltozik, a teljes adathalmaz használhatatlanná válik. Ezért sorosítást csak rövid élettartalmú adatok esetén használok.

10.2 XML kezelés

A .Net felajánlja a két leggyakrabban használt XML kezelési módszer teljes implementációját: SAX a szekvenciális olvasáshoz, és DOM a dinamikus kezeléshez. A platform-függetlenség, az olvashatóság és a javíthatóság itt is előnyként jelenik meg, és ha az általánosságban vett XML kezelés szintjén maradunk, a sorosítással ellentétben a korábbi adathalmazok részadatait is fel tudjuk használni. Ez például akkor lehet fontos, ha az adatstruktúra időben is változhat.

Például, a mi esetünkben a játékprogramon kívül létezik egy külön fejlesztett alkalmazás, amellyel útmutatást tudunk adni a játékszintek véletlenszerű generálásához. Az adatok XML állományokban tárolódnak. Ha időközben bizonyos definíciók több tulajdonsággal bővülnek, kár lenne elveszteni régebben megalkotott, komplex, sok munkát igénylő adathalmazokat.

Az általános XML kezelés hátránya azonban, hogy nem egy specifikus adathalmazt kapunk a memóriában, hanem sablonszerű objektumokat. Lemondunk a típusságról és az olvasható, rendezett forráskódról.

A XML data binding egy módszer, amely fogalmi szinten valahol a natív XML kezelés és az XML sorosítás között helyezkedik el. Lényege, hogy az XML leírásban szereplő adatot megfelelteti egy típusos adatszerkezetnek, melynek definíciója illeszkedik az adott programnyelvhez. Az adatstruktúra fordítási időben is ismert, amit előre elkészített, vagy utólag kinyert séma alapján manuálisan és automatikusan is létre lehet hozni. Ennek a módszernek a létjogosultsága az XML sorosítással szemben abban rejlik, hogy míg a sorosítás valamilyen objektum-szerkezet elmentését és visszatöltését végzi, megszabott struktúrával, a data binding egy előre elkészített, általános sémájú adatállományt dolgoz fel.

Ennek egy egyszerű változatát implementáltam, szem előtt tartva a korábban említett követelményeket, továbbá, hogy tudjon használni olyan alapelemeket, amelyek a szoftver többi részében jelen vannak, de esetleg nem olyan módon lettek megalkotva, amire nekünk valóban szükségünk lenne.

Megvalósítást tekintve az XML kezelő modul szintén a Reflection modult használja, hiszen ezzel automatizálható a különböző típussal rendelkező tagváltozók és osztálydefiníciók helyes kezelése. Az XML állomány beolvasása DOM használatával történt, csupán hogy a fejlesztés gyors legyen.

11 Hálózati kommunikáció

11.1 Hálózati protokollok

Fontos kérdés a szoftverek esetén legelőször is, hogy kapcsolattal rendelkező (TCP), vagy kapcsolat nélküli (UDP) hálózati forgalmat bonyolítsanak le.

Általánosan elfogadott szabály, hogy egy multimédiás szoftver esetén az UDP kommunikációt tartják előnyben, mivel a hálózati csomagoknak real-time időben kell megérkezniük a forrástól a célig. Szakmai körökben ezt úgy mondják, hogy „a késett csomag már nem is érdekes”. Éppen ezért a szokásos minőségbiztosítási módszerek (pl.: garantáltan nem veszik el csomag; minden csomag egyszer érkezik meg) helyett arra törekednek, hogy bármely időpontban elő lehessen állítani egy konzisztens állapotot.

A TCP kapcsolat ezzel szemben könnyen kezelhető, kevesebb mellékes programozási munkát igényel. Tökéletes megoldás nagy méretű adathalmaz precíz átvitelére a hálózaton. Már korábban említettem, hogy a programomban a kapcsolódás elején át kell küldeni a hálózaton azt a játékteret, amelyen a játék folyni fog. Erre ez a fajta átviteli módszer tökéletes.

Éppen ezért a hálózati játékhoz való csatlakozás két fázissal történik. Az adminisztrációs fázisban TCP protokollt használok, a játék fázisában viszont UDP protokollt.

Az adminisztrációs fázisban nem történnek olyan adatok átvitele, ami kritikus átviteli idejűek, inkább – a felesleges programozási feladatokat elhárítandó – érdemes kihasználni a TCP protokoll minőségbiztosítási kritériumait. Itt történik a szerverhez való kapcsolódás, a szerver információinak lekérdezése, új játék indítása vagy egy játékhoz való csatlakozás, valamint a játéktér fel- vagy letöltése. Ha mindez lebonyolódott, akkor a TCP kapcsolat lebomlik.

Ezután a kliensek a szervert UDP csomagokkal értesítik saját állapotukról, a szerver kiszámolja azokat az információkat, amelyeket a kliensekkel meg kell osztania, és kiértesíti a kapott információkat. A szoftver jelenlegi állapotából adódóan kevés információval le lehet írni a játéktérben bekövetkezett változásokat, ezért a szerver a klienseket folyamatosan az aktuális állapotról tájékoztatja. Így akár ideiglenes hálózati hibák esetén is konzisztens állapot hamar létre tud jönni a kliensek közül.

11.2 A hálózati egységek logikai felépítése

Amíg a munkám el nem ért arra a pontra, hogy a hálózati egységek fejlesztésébe is hozzákezdjek, elmondható volt, hogy teljes mértékben a kliens alkalmazás írására koncentráltam. A korábban említettek következtében azonban kiderült, hogy a szerver egy külön futtatható alkalmazás keretén belül kerülne bele a munkába. Ezért a szerver programmal szemben a következő kritériumokat fogalmaztam meg:

1. Minél kisebb és gyorsabb legyen.

Mivel az új alkalmazást a nulláról kezdve kellett írnom, célom volt az, hogy minél korábban egy használható verzió álljon elő. Másrészről az is igaz, hogy mivel a szerveralkalmazások több klienst szolgálnak ki egyszerre, a teljesítmény fontos szempont. A megfelelő gyorsaság eléréséhez az egyik út az, ha lemondunk a háromdimenziós megjelenítésről és minden egyéb csupán esztétikai funkcionalitásról, és csak a legfontosabb, elengedhetetlen részeket írjuk meg.

2. A szerver legyen minél függetlenebb a kliensektől

Párhuzamosan több alkalmazás fejlesztése még a mai korszerű forráskezeléssel is bonyolult, hosszadalmas feladat. Sőt, mivel a kliensalkalmazásba már minden fontos funkciót megírtunk, értelmetlen lenne újra megírni őket egy másik programba. Így a szerver megírásánál arra törekedtem, hogy minél kevesebb, a játékkal kapcsolatos funkcionalitást végezzen el, inkább építsen a kliens modulok erőforrásainak kihasználására.

Ami azonban a hálózati forgalom lebonyolításán kívül fontos része a szervernek, az, hogy a kliensek közötti kölcsönhatást konzisztensen kezelje. Olyan feladatoknál, mint a játékosok egymással való ütközése, fontos, hogy a számolás egy centralizált, elkülönített helyen történjen, különben az is lehet, hogy akár az összes kliens más eredményt ad ugyanarra a helyzetre.

Másrészről fontos, hogy a szerveren belül és a kliens és a szerver közötti adatforgalomban használ adatreprezentáció lehető legjobban független legyen a kliens

belső adatkezelésétől, hogy a kliens alkalmazás megváltozásával minél kevésbé kelljen a szerver alkalmazásba belenyúlni.

11.2.1 Szerver oldal

A szerveralkalmazás tevékenységi két külön logikai részre bontható.

Az adminisztrációs egység egy közös felület, amelyen keresztül a kliensalkalmazások kapcsolódhatnak a szerverhez, információt kérnek le annak állapotáról, majd a megfelelő módon indíthatnak új játékot, vagy csatlakozhatnak egy régihez.

A játék egység feladata, hogy a szerveren különböző játékmeneteket futtasson, tárolva azok jelenlegi állását, és közvetítve a kliens moduloknak.

1. Adminisztrációs egység

Az adminisztrációs egységen a kliensek egy közös felületen keresztül irányítják a szerver felé a kéréseiket. A csatlakozási kérelmek és kérések fogadása egy közös szálon hajtódik végre, összeköttetés alapú protokollon keresztül. A csatlakozás után a kapcsolatokat az alkalmazás eltárolja, és egységesen kezeli őket. A kommunikáció kérés-válasz alapú, szinkron folyamat, egészen addig, amíg a felhasználó vagy ki nem lép a programból, vagy nem jelzi játékindítási- (publish) vagy csatlakozási (join) szándékát. Ekkor szükséges adatok átvitele után a szerver lebontja a kapcsolatot.

2. Játék egység

Minden játékmenet egy-egy külön szálon fut. Az egyes szálakhoz vannak azok a játékosok regisztrálva, akik az adott játékmenetben részt vesznek. (Úgy is lehet mondani, hogy a rendszernek ez a része manuálisan megvalósított kapcsolat alapján működik.) A kliensek folyamatosan értesítik a szervert jelenlegi állapotukról.

A szerver feladata, hogy a kapott információkat összegyűjtse, egybedolgozza őket, és a kapott eredményről az összes klienst értesítse. Mindig tárolja a kliensek legutolsó állapotát. Ha új információ érkezik, akkor úgy veszi, mintha a többi kliens állapota nem változott volna, így mindig biztosítható, hogy egy konzisztens állapot éppen tárolódni fog a játékról a szerver memóriájában.

11.2.2 Kliens oldal

A kliens oldal hálózati folyamatai is az elmondottak alapján két részre osztható. Az adminisztrációs rész gyakorlatilag azon időszak, amikor a felhasználó a menüben kiválasztja a megfelelő opciókat a hálózat kezeléséhez. Majd a szerver játék egységével való kommunikáció akkor indul el, mikor a felhasználó is visszakerül a menüből.

1. Adminisztrációs rész

A kapcsolat felépítése után a kliens lekérdezi a szerver megfelelő információit, és átadja az irányítást a felhasználónak, aki választhat, hogy új játékot indít-e a szerveren, vagy egy meglévőhöz kíván csatlakozni. Amennyiben megtörtént a megfelelő művelet a szerveren, a szükséges adatok elküldése illetve fogadása után az alkalmazás kilép a menüből, és informálja a szervert arról, hogy megkapta a szükséges adatokat. Ezt már egy a kapcsolat nélküli protokollal elküldött csomaggal teszi meg.

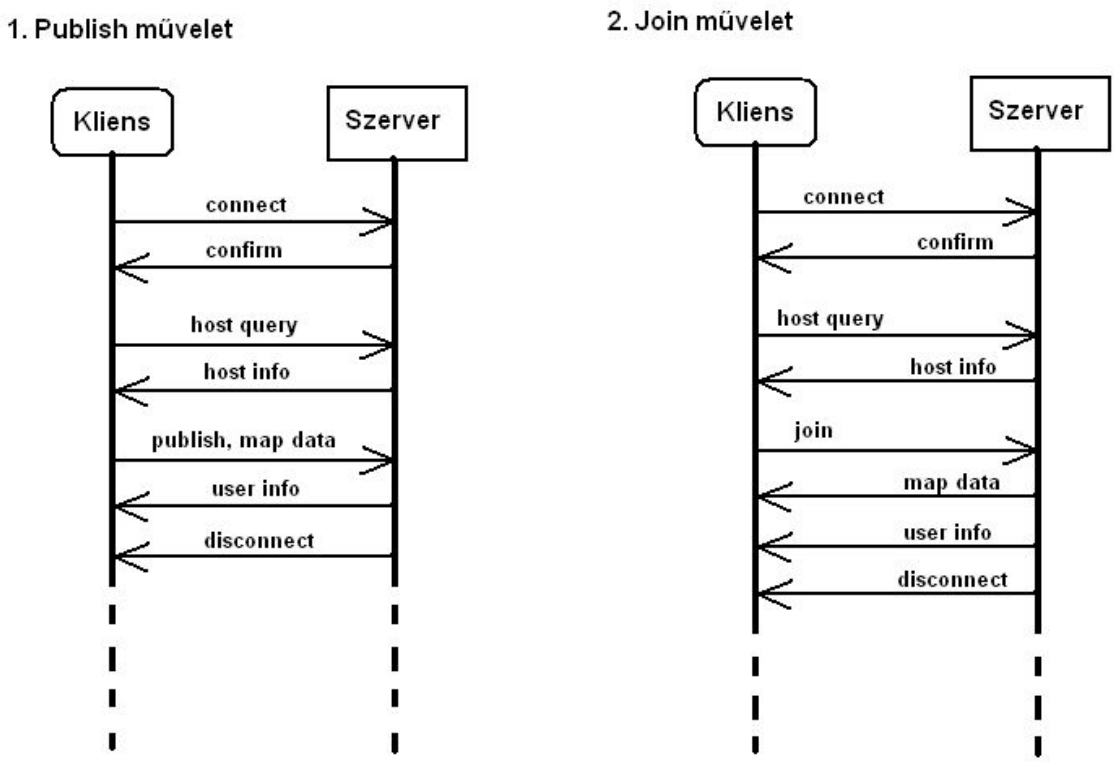
2. Játék rész

A játék folyamán időközönként a kliens program összeállít saját adataiból egy csomagot, majd azt küldi a szervernek. Mivel nincs garancia a csomagok megérkezésére egyik oldalon sem ezért, a játék folytatása és a csomagok küldözgetése akkor is megtörténhet, ha már régóta nem érkezett válasz a szerver felől. A játékból való kilépéskor a kliens informálja a szervert.

11.2.3 A hálózati csomagok forgalma

1. Adminisztrációs rész

Az alábbi ábra szemlélteti, hogy miképp történik a hálózati forgalom az adminisztrációs rész ideje alatt, optimális esetben. Baloldalon az az eset látható, amikor a felhasználó új játékot akar indítani a szerveren, jobb oldalon, pedig mikor csatlakozni kíván egy más meglévőhöz.



28. ábra: A kliens és a szerver közti kommunikáció az adminisztrációs fázisban

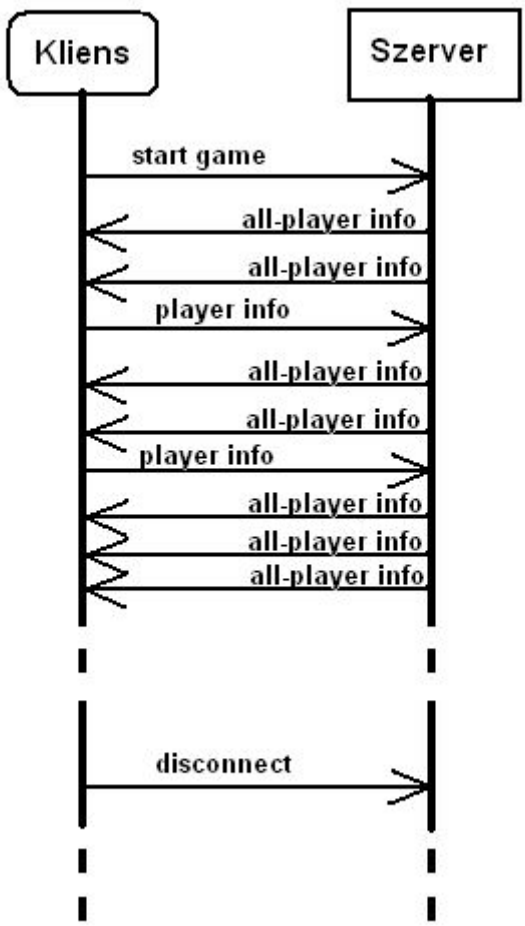
Mindkét esetben az látható, hogy legelőször is a kliens a közös csatornán csatlakozni kíván a szerverhez. Ha ez sikerül, akkor a kliens küld egy „host query” csomagot, amely különösebb információt nem tartalmaz. Annál inkább fontosabb a szervernek erre adott válasza, a „host info”, amelyben a szerver leírja, hogy hány játékmenet indítható egyszerre rajta, továbbá, hogy az egyes férőhelyek közül melyik üres még, és melyiken folyik már játék. Továbbá azt is leírja, hogy az egyes játékokhoz legfeljebb hány játékos csatlakozhat, és hogy jelenleg mennyien játszanak rajta.

Ha a kliens úgy dönt, hogy saját játékát megosztja a szerverrel, akkor az indítási kérelemmel együtt csatolja a játéktér kezdő állapotát is. Cserébe a szerver elküld egy „user info” csomagot a kliensnek, ami tartalmazza, hogy milyen csatornán él a játéka, ahová őt automatikusan bevonja. Ezután a szerver lebontja a kapcsolatot, és a megadott csatornán várja a kliens jelentkezését.

Abban az esetben, ha a kliens egy meglévő játékhoz kíván csatlakozni, egy csomaggal jelzi szándékát a szervernek, aki válaszként elküldi a játéktér kezdő

állapotát, továbbá ebben az esetben is küld egy „user info” csomagot, és regisztrálja a klienst az adott játékban. Miután lebontotta a kapcsolatot, a megadott csatornán várja a kliens jelentkezését.

A játékrész kommunikációját a következő ábra szemlélteti.



28. ábra: A kliens és a szerver közti kommunikáció a játék fázisában

A kliens először is küld a „user info” csomagban kapott adatok szerint egy „start game” üzentet, ami arról biztosítja a szervert, hogy megkapta a szükséges információkat, és hogy ezután a szerver folyamatosan küldheti neki a játékosok adatait. Majd ahogy változik a kliens állapota, egy „player info” csomagban elküldi saját adatait a szervernek. A szerver a kapott adatokat összesíti, és egy „all-player info” üzenetben

adja tovább a klienseknek. A kliens jelezheti továbbá a szervernek, hogy kilép az adott játékból.

11.3 A hálózat egységek implementálási részletei

A kommunikáció tényleges implementálásához választanom kellett egy gyorsan elkészíthető alternatívát. Két lehetőség állt előttem: A .Net Framework által biztosított megoldás, valamint a DirectX által kezdeményezett DirectPlay. Az idő azonban ezt a kérdést is megoldotta, mivel a DirectPlay-t visszavonták. A .Net Framework-ben szerepel a már megszokott Socket alapú kommunikáció, a munkám során ezt használtam fel.

Mint ahogy korábban említettem, az adminisztrációs rész – azaz a kapcsolat orientált rész – a TCP protokollt használja, amely programozási értelemben megkönnyítette a helyzetemet. Ezzel szemben a játék résznél UDP protokollt használtam, ahol a kívánatos minőségi követelményeket magamnak kellett megoldanom.

Mivel a szerver és a kliens is minden esetben tárol magánál egy értelmes állapotot, még ha a kettő egymással nem is konzisztens, idővel az információ csere során egyeztetésre kerülnek. Így a csomagvesztéssel nem kell törődni, hiszen a rendszer jellege miatt a hiba idővel kiküszöbölődnek. A probléma a csomagok megkettőződése vagy sorrendiségének változása.

Az üzenetek, amik a hálózaton keresztül mennek, azok az egyes felek legutolsó teljes állapotát tükrözik, ezért elég, ha a beérkező csomagok közül mindig a legutolsót, a legaktuálisabbat vesszük figyelembe. Ezért a kellő minőség biztosítására az UDP csomagok esetében a számozást vezettem be.

A hálózati forgalomban részt vevő csomagok definícióit egy közös könyvtár projektben helyeztem el, a legfontosabbak alább láthatóak.

```
[Serializable]
public class Packet
{
      public int sender;
      public long counter;
      public
GameObjectData data;
}
```

```
[Serializable]
public class GameObjectData
{
      protected string description;

      public string Description {
            get { return description; } }

      public GameObjectData(){
            description = null; }

      public GameObjectData(
            string description){
       this.description = description;
      }

      public virtual object create()
      {
            throw new Exception(
            "Illegal    create    function
call.");
      }
}
```

29. ábra: Hálózati csomagoló osztályok

A jobb oldalon található a hálózaton keresztül átküldhető adatok ősosztálya. Ezekre az adatokra jellemző, hogy van egy leírásuk (description), ami megadja a csomag jelentését, így konvertálás nélkül is megállapítható, hogy a megfelelő információt tartalmazza-e. Továbbá az osztály sorosítható, hiszen a hálózaton való átküldéshez először bitfolyammá kell alakítani.

Az kliens alkalmazásban minden objektumnak, amelynek állapotát a hálózaton közölni kívánunk, létezik egy GameObjectData osztályból örököltetett segédosztálya, ahol a kiválasztott, sorosítható adatok szerepelnek. Ezek a segédosztályok felüldefiniálják a „create” függvényt, amelyben meghatározzák, hogy a bennük található adatokból hogyan hozható létre egy eredetileg használt objektum.

Tulajdonságai miatt a GameObjectData példányait használom a hálózaton történő kommunikáció lebonyolításához is. UDP protokoll esetében az előbb vázolt okok miatt ezt az objektumot becsomagolom egy Packet példányba elküldés előtt. Ennek az osztálynak a definíciója látható a baloldalon. Tartalmazza a csomag küldőjét, a sorszámát, és természetesen a küldeni kívánt adatot.

A hálózati forgalom elemzéséhez szemügyre kell venni a szerveralkalmazással szemben megfogalmazott második kritériumot is, miszerint a szervert lehető legjobban el kell különíteni a kliens alkalmazás implementációjától.

Ugyanis a hálózaton történő adat elküldése a következő módon történik: Kiválasztjuk a küldeni kívánt adatokat, becsomagoljuk őket a megfelelő osztályok példányaiba, sorosítjuk a kapott adathalmazt, és úgy tesszük ki a hálózatra. A másik fél kiolvassa a kapott adatot, visszaalakítja a sorosított biteket, megvizsgálja a csomag értelmét, és elvégzi a kiadott feladatot.

A szervert úgy programoztam, hogy minél kevesebb adatot kelljen feldolgozni a játék során, nevezetesen megkapja a játékosok pozícióját, ütközteti őket, és a kapott információkat visszaküldi a klienseknek. A többi vizsgálatot a kliensek végzik, hiszen azok már eredetileg is implementálva voltak kliens oldalon.

Ebből következik, hogy a játéktér többi részéről a szerveralkalmazásnak semmit sem kell tudni, vagyis érdektelen, hogy mennyi minden más objektum tartózkodik éppen a pályán. Értelmesnek tűnik tehát az a megfontolás, hogy az ezekhez tartozó osztályokat ne osszuk meg a szerver forráskódjával, hiszen úgysem értelmezi őket. Ezzel ellentétben, ahhoz, hogy a sorosítás visszaalakítása sikeres legyen, az kell, hogy a szerver ismerje a sorosított adathalmazban szereplő osztályok definícióit.

Ezért, amikor a kliens feltölti a szervernek a játékteret, akkor azt kétszeresen sorosítja. Az első sorosításkor kapott eredményt eltárolja a memóriában, becsomagolja egy GameObjectData osztálybeli példánnyal, és a kapott eredmény újbóli sorosítását küldi el a szervernek. Ennek értelmében az első visszaállításkor a szerver az általa ismeretlen osztályokból álló adathalmazt egy bájtsorozatot tartalmazó memóriaterületnek érzékeli, és nem fejtegeti tovább, mit tartalmazhat.

11.3.1 Szerveralkalmazás

A szerver fő modulja egy egyszálú végtelen ciklus, amely két nem blokkoló hálózati műveletet vizsgál időről időre.

Az egyik a kapcsolódó kérelmek fogadása. A szerver nyit egy közös TCP típusú szoftvercsatornát, előre meghirdetett címmel és port-tal. Az érkező kérelmek elfogadásakor az újonnan keletkezett kapcsolatot egy ServerControl típusú objektumba tárolja el, és teszi egy listába.

A másik művelet a kapcsolatok figyelése, hogy érkezett-e azokon kérelem. Ezek vizsgálatát statikus multiplexálással végzi, amely lehetőséget ad, hogy egyszerre egy szálon több szoftvercsatorna állapotát vizsgáljuk aszerint, hogy érkezett-e üzenet.

Amennyiben a kliens új játékot hoz létre, a szerver létrehoz egy Host példányt, és regisztrálja hozzá az adott klienst, végül felbontja a kapcsolatot.

A Host osztály reprezentál egy éppen futó játékot. Nyit egy a játékosok számára közös szoftvercsatornát, amelyen várja a kliensek adatait. Minden Host a többitől külön szálon fut, és külön port-on várakozik, így nincs szükség multiplexálásra. Valahányszor információ érkezik, az adott Host eltárolja a megfelelő játékos állapotát, és új közös állapotot hoz létre a játékosoknak, végül kiértesíti őket az eredményről.

A Host egy játékoshoz tartozó információt egy Player típusú objektumban tárol. Minden játékost jellemez a címe, egyénileg kiosztott port-ja, pozíciója, továbbá ez az objektum kezeli az adott játékos irányába történő hálózati kommunikációt, beleértve a csomagok megfelelő számozását is.

A Host szabadon kezeli a játékosokat, olyan értelemben, hogy akár a játékot indítható kliens is bonthatja a kapcsolatot, a játék tovább folytatódik. Egy játék akkor válik érvénytelenné, ha azt az összes játékos elhagyta, ilyenkor a szerver gondoskodik arról, hogy a host példány törlődjön.

Az első kritérium szerint a az alkalmazásnak mind fejlesztési időben, mind teljesítményben gyorsnak kellett lennie, ezért a szokásos grafikus megjelenítés helyett a .Net Framework Windows Forms támogatását használtam megjelenítésre. Indításkor egyetlen ablak jelenik meg, ahol megadhatjuk a közös TCP szoftvercsatorna port-számát. A szerver elindítása után kijelzi, hogy éppen, milyen ServerControl kapcsolatokat tart nyílván, és mik az éppen futó játékok paraméterei.

11.3.2 Tesztkliens

Mivel az alkalmazás grafikus része már elég komplex volt, gondot jelentett volna azon fejleszteni a hálózati részt, hiszen a betöltés hosszassá vált, és nehezebb lett volna abban az alkalmazásban a hibákat megkeresni. Ezért létrehoztam egy tesztalkalmazást a szokásos Windows Forms megjelenítéssel, amely szimulálja egy

kliens hálózati kommunikációját. A hálózat kezelésének implementációját egy külön osztályban írtam meg, amelyiket könnyedén lehetett integrálni a grafikus alkalmazásba.

A program működésének folyamata a következő.

1. Beírjuk a szerver címét

Ekkor egy alatta levő ablakban megjelennek azok az IP-címek, amelyeken elérhető a beírt cím. Ezek közül kell kiválasztanunk egyet.

2. Kiválasztjuk a szerver közös port-ját

3. Kapcsolódunk

A „Connect” gomb megnyomásával a teszt program létrehoz egy NetworkClient példányt, amely a szerverrel való kapcsolatunkat fogja reprezentálni. Ennek tagfüggvényének meghívásával a kliens kapcsolódik a szerverhez, várja a küldendő üzeneteket, és folyamatosan figyeli, hogy érkezik-e válasz.

4. Szerver információt kérünk

A „Host Query” megnyomásával lekérdezzük a szerver állapotát. Egy erre külön készített ablakban megjelenik a szerver Host példányainak listája, annak paramétereivel együtt.

5. Publish / Join

A kapott listából kiválasztva egy elemet és a megfelelő gombot lenyomva új játékot indíthatunk, vagy egy meglévőhöz csatlakozhatunk. Mindenképpen kapunk két UDP port számot. Az első az, ahová a kliensnek az információkat küldenie kell, a második, amelyiket figyelnie kell. Továbbá a kapcsolat a szerverrel megszűnik.

6. Készenlét jelzése

A kommunikáció további, azaz UDP részét a PlayClient objektum fogja irányítani. A „Start Game” lenyomásakor egy rövid üzenetet küld a megadott port-ra, hogy készen áll, és várja a szervertől az információkat. Ezeket tovább küldi a program többi moduljának.

7. Játékos állapotának frissítése

A teszt kliensen van egy ablak, amelyben egy vörös keresztet irányíthatunk úgy, hogy a közelében egérrel az ablakra kattintunk. Ezzel az alkalmazásunk által megtestesített játékosának koordinátáit változtatjuk meg, és küldjük át a többi kliensnek. Kék keresztek jelzik azt, hogy a többi játékosok hol tartózkodnak.

8. Kilépés

A „Quit Game” hatására a PlayClient objektum egy kilépő üzenetet küld a szervernek, és törli a hálózati kommunikációhoz tartozó szoftvercsatornát.

11.3.3 Grafikus kliensalkalmazás

A grafikus kliens esetében ugyanezt az utat kell végigjárnunk a hálózati játékhoz, csupán a folyamat a grafikus környezetbe integrálva kezelhető. Ehhez azonban először az alkalmazás állományaiba kell nyúlnunk.

A grafikus kliens futtatható állományának könyvtár-hierarchiájában kell a „network” könyvtárhoz férnünk. Itt tárolódnak az előre ismert hálózati címek és port-számok, amelyekkel kapcsolódni lehet a játékon belül a szerverhez.

A könyvtárban található „.srv” kiterjesztésű XML állományokban tárolódnak a fontos információk. Ezek az állományok egyetlen „server” gyökérelemből állnak, amelynek három attribútuma van: A szerver neve (name), címe (address) és a port-szám (port).

Az alkalmazás futása során, ha csatlakozni kívánunk egy megadott szerverhez, akkor a menüben ezek közül a fájlok közül választhatunk. A program kiolvassa a címet, és a kapott IP-címeket felkínálja. Ezekből egyet választva a Network Client osztály kapcsolódik a szerverhez, és automatikusan lekéri annak információját. Egy újabb almenüben megjeleníti a listát, ahol a kiválaszthatjuk, hogy melyik játékrekeszben próbálunk játékot indítani vagy játékhoz kapcsolódni.

Ha ez megtörtént, az alkalmazás (az esedékes betöltések után) átkerül a játéktérhez. A rendszer belső állapotváltása gondoskodik arról, hogy a szükséges információkat kiolvassa a kliens a hálózatból, és felszabadítsa a NetworkClient objektumot. Ezután létrejön a PlayClient objektum, és kapcsolódik a kapott információk alapján a szerver játékrészéhez.

Amíg a NetworkClient konkrétan a menübe lett beintegrálva, a PlayClient a PlayerObject-hez kötődik szorosan. A PlayClient felelőssége a korábban leírtak szerint csupán a hálózati kommunikációra korlátozódik, az adatok értelmezését a PlayerObject objektum végzi. Adott időközönként kiválogatja azokat az adatokat, amikre a szervernek a számolás során szüksége van, és átadja a PlayerClient objektumnak.

Ezzel aszinkron módon a folyamatosan beérkező adatokat a PlayClient közvetíti a PlayerObject példánynak, aki ez alapján kijavítja a saját adatait, és megjeleníti egy modell segítségével a többi játékost.

Amikor a felhasználó befejezi a játékot, a rendszer állapotváltása gondoskodik arról, hogy a PlayClient kilépő üzenetet küldjön a szervernek.

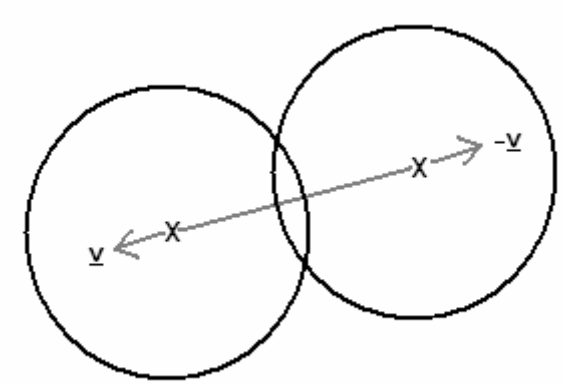
Megjegyzendő, hogy a teszt kliens és a grafikus alkalmazás képes egy szerveren, egyazon játékhoz kapcsolódni. Ennek azonban az a követelménye, hogy a játékot a grafikus kliens indítsa. Mivel a teszt kliens a grafikus alkalmazástól teljesen független, nem ismeri annak adatszerkezeteit, nem tud feltölteni a szerverre olyan játékteret, amit a grafikus kliens értelmezni tudna. Ezzel ellentétben a teszt kliens nem foglalkozik a játéktérrel, így ő bármilyen játékhoz kapcsolódhat.

Ebből fakad az a tény is, hogy hálózati forgalomban levő adatok a kétfajta kliens esetében logikailag nem kompatibilisek egymással, így nem tudjuk az egyik működését a másikkal nyomon követni.

11.3.4 Ütközések számolása

A korábban leírt algoritmus, amely a játékos és a játéktér ütközését vizsgálja és kezeli, továbbra is működő képes. Azonban egy játékosra nézve is elég erőforrás igényes, nem várható el a szervertől, hogy minden éppen folyó játék esetében legyen tisztában a játéktérrel, és minden játékossal végezze el az ütközésvizsgálatot. Ezért ezt a fajta vizsgálatot továbbra is a kliensek maguknak végzik.

A játékosok egymás között történő ütközése más elvárásokat vet fel. A játéktérrel való ütköztetés során az elv az volt, hogy a pálya elemei nem mozdulnak, ezért csak a játékos mozgását befolyásolhatjuk. Másrészről tervezői döntés volt, hogy olyan ütközési algoritmust választottam, ahol a játékos nem süllyedhet bele a játéktér részeibe, de szem előtt kell tartanunk, hogy ez csak egy mozgó objektum ütközésének vizsgálatánál használható megoldás. (Külön problémát vetne az az eset, amikor a szerver úgy kapja kézhez a játékosok pozícióját, hogy azok már egymásba érnek.)



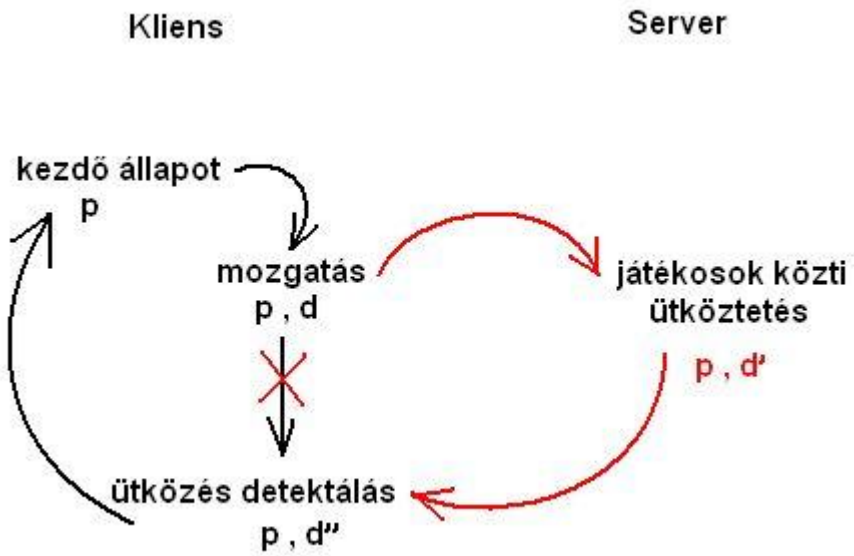
30. ábra: Ütköztetés befoglaló gömbökkel

Éppen ezért a szerver a következőképpen számolja a játékosok ütközését: Megkapja a játékosok pozícióját, és megnézi, hogy a befoglaló testek mennyire érnek bele egymásba. Kiszámítja, hogy az egyes objektumokat milyen irányba kellene elmozdítani, és mennyivel, ahhoz, hogy az ütközés előbb-utóbb feloldódjon. Végül hozzáadja a játékosok paramétereihez a kapott értékeket. Az egyszerűség kedvéért itt befoglaló gömböket alkalmaztam.

Azt is látatjuk, hogy ennek a módszernek a hatására a vizsgált objektumok valóban kétirányú kölcsönhatásban állnak egymással, így arra is lehetőség akad, hogy az egyik játékos a másikat eltolja.

Az kliens eddigi ütközésvizsgálata tehát úgy működött, hogy adott volt a játékos kezdő pozíciója (p), amihez az irányítás során hozzávettünk egy elmozdulás-vektort (d). Az ütközés detektálása során azt vizsgáltuk, hogy a kapott vektor belelóg-e egy kijelölt térbeli alakzatba. Ha beleért, akkor az elmozdulás-vektort lerövidítettük (d''), és ezt adtuk hozzá a pozícióhoz.

Az új ütközés-detektálási folyamat ezt a láncot szakítja meg, és a pozíciót és az elmozdulás-vektort a kliens elküldi a szervernek. A szerver összeadja őket, és megvizsgálja, hogy a játékosok a leendő pozícióban ütköznek-e egymással. Kiértékeli az eredményeket, és így megállapítja, hogy az adott játékos pozícióját mennyivel kellene megváltoztatni ahhoz, hogy az ütközés feloldódjon ($\underline{v}$). A kapott vektort hozzáadja az elmozdulás-vektorhoz, és az eredeti pozícióval együtt, mellékelve a többi játékos adatait is, visszaküldi a kliensnek. Ezután a kliens az új pozícióval és elmozdulás-vektorral végzi el azt ütköztetést a játéktéren, és az így kialakult adatokat írja vissza a játékos állapotába.



31. ábra: Ütközés-detektálási folyamat a szerver bevonásával

12 Kiegészítő eszközök

Egy nagyobb projekt során gyakran szükségünk van arra, hogy – miután az alapokat, nagyobb elemeket megírtuk – részletesebb anyagokkal szépítsük a tovább a munkákat. Ezekhez általában kész szoftvertermékeket használunk fel, még ha nem is olcsó, a határidő szempontjából legtöbbször ez a hasznosabb megoldás.

Előfordulhat azonban, hogy a termék valamilyen szempontból nem felel meg az elvárásainknak, például valamilyen műveletet nem tudunk a programmal megvalósítani, vagy a kimeneti adatok formátuma nem megfelelő. Ezekben az esetekben még mindig előttünk áll a lehetőség, hogy a szükséges szoftvereket magunk írjuk meg.

Előnyünkre válhat az is, hogy a magunk írt kiegészítőket sokkal testhezállóbbra tudjuk formálni. (Amíg a program belső használatra készül, ez semmilyen téren nem árt.) Nem csak a kezelését tudjuk személyre szabni, hanem időközben felmerülő problémákra tudunk új megoldásokat az alkalmazáshoz adni.

Az ebben a részben leírtak két fontos szempont miatt különböznek az előzőektől.

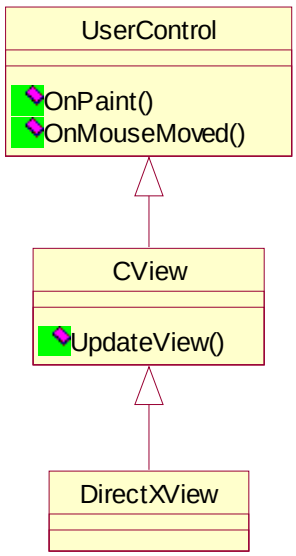
Először is a korábbiakban olyan alkalmazásokról volt szó, amelyek vagy teljes mértékben DirectX grafikával bírtak – ez volt maga a játék – vagy csak Windows Forms megjelenítést használtak. A kiegészítő alkalmazásoknál viszont keveredik a kettő, mivel az alkalmazás is kettős céllal készül. Gyorsan és folyamatosan fejleszthetőnek kell lennie. Ebben segít a Microsoft Visual Studio .Net 2003. grafikus ablakszerkesztő komponense, amellyel már fejlesztés közben is pontos képet kaphatunk arról, hogyan fog a programunk kinézni. A .Net eseménykezelő rendszere ehhez biztosítja azt, hogy a felhasználói felületet könnyedén a megvalósított logikához kapcsolhassuk. Ugyanakkor szeretnénk látni, hogy az eszközzel mit valósítunk meg, méghozzá szerkesztés közben. Így biztosítanunk kell, hogy a DirectX grafikus megjelenítés is integrálódjon az alkalmazásba.

A második fő szempont, hogy az így elkészült szoftver nem önálló eszközként fog működni, hanem egy másik alkalmazás munkáját fogja alátámasztani. Arra fogjuk használni, hogy a játék gyorsabb, látványosabb, vagy változatosabb legyen. Ezért a fejlesztése minden szempontból összefügg a fő program szerkesztésével, gyakran közös

osztálydefiníciókkal dolgoznak. Más oldalról nézve a két alkalmazás gyakorlatilag sosem fog egyszerre futni, futás közben együttműködni. Az eszközzel csupán off-line, azaz előre kiszámolt, megformázott adatokat készítünk el, amelyeket a játék később épít be a saját rendszerébe.

12.1 Grafikus integráció

Első feladat az volt tehát, hogy a DirectX grafikus megjelenítésnek olyan formát adjunk, amellyel beilleszthetővé válik a már megszokott ablakos rajzolásba és eseménykezelésbe. Egyik oldalról a .Net ablakkezelő osztályai lehetőséget adnak a felhasználónak arra, hogy egy irányító elemet (control) saját maga tervezzen meg. Ugyanúgy, mint az ablakok szerkesztésénél, könnyen összeállítható egy Windows Forms elemekből összeállított struktúra. Másik oldalról, a DirectX szorosan kapcsolódik az ablakkezelő rendszerekhez, hiszen a megjelenítés paraméterezésénél lehetőségünk van (és kötelező is) megadni azt az irányító elemet, amelyen a kirajzolást végzi.



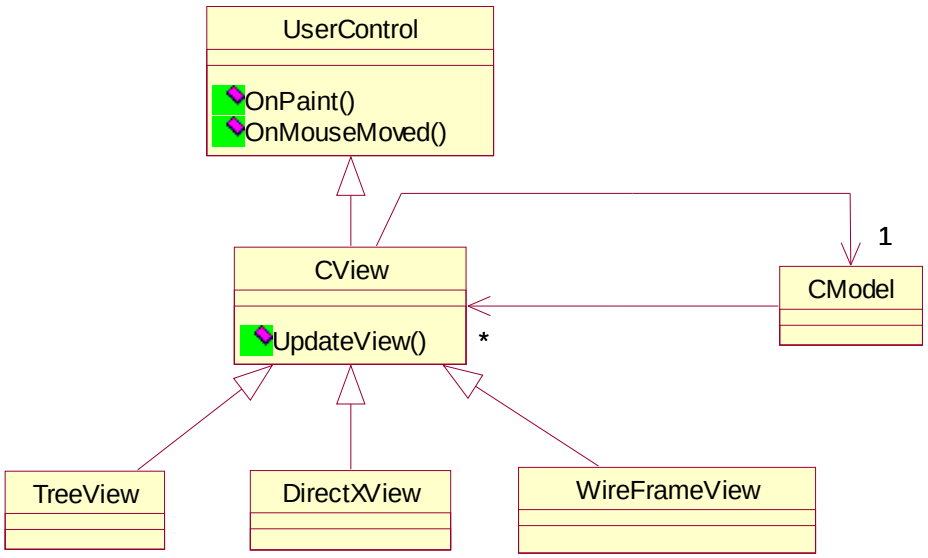
32. ábra: A DirectXView komponens helye az osztályhierachiában

A fenti ábra szimbolizálja, hogy a DirectX megjelenítésű komponens a UserControl osztályból öröklődik (közvetetten), és annak a megfelelő függvényeit definiálja felül. Az OnPaint függvényben értesítjük a DirectX-et, hogy újra ki kell rajzolnia a reprezentált objektumokat, és az irányítást vezérlő felülírható függvények és

események segítségével irányítjuk ezt az egységet, amellyel a Windows ablakkezelőjének irányítási folyamatába léphetünk bele. Így kapunk egy önálló, egységbe zárt komponenst, amelyet a Microsoft Visual Studio .Net 2003. szerkesztőjében ugyanúgy feltehetünk egy ablakra, mint például egy egyszerű gombot.

A CView osztály szerepe, hogy az eddig megszokott SDI (Single Document Interface) architektúráról MDI (Multiple Document Interface) architektúrára, hiszen a DirectX megjelenítésű komponens ezúttal osztozni fog más nézeteken is. Az új architektúra alapja a Observe tervezési minta és a Document-View típusú szétválasztás. Ebben a kialakításban – a játékhoz hasonlóan – első lépésben különválasztjuk a modellt, amely az információk tárolására és azon speciális műveletek elvégzésére szakosodott struktúra. A CModel osztály további feladatai a nézetek regisztrálása, majd értesítése a változásokról, amely utasítja a nézeteket önmaguk átrajzolására.

A fentieket mutatja be a következő ábra:



33. ábra: Az Observe tervezési minta implementációja

Ilyen megoldásoknál általában a Windows komponensek Invalidate függvényét használják a nézetek értesítésére, mivel a legtöbb nézet egyszerű, gyors függvényhívásokkal meg tudja rajzolni önmagát. A DirectXView komponens működése azonban ettől lényegesen eltér. Nem csak hogy magát a megjelenítő eszközt kell valamilyen inicializáló programmal működésre bírni, de minden kirajzolandó elem – legyen az egy egyszerű rácsvonal, vagy egy bonyolult alakzat – felkerül a videokártya

memóriájába. A feltöltés azonban hosszadalmas művelet, főként, mivel nem felügyelt komponensek kódjait hívja meg. Ezért különbséget kell tennünk két alapeset között. Az elsőben, amikor például csak egy ablak eltakarta a komponensünket, és annak eltűnésével újra kell rajzolnunk a felületet, akkor elég lenne egy Invalidate hívás. Ezzel ellentétben, ha bármilyen megjelenítésre szánt alakzat formája megváltozott, újra a videokártya tartalmához kell nyúlnunk. Éppen ezért a CModel a nézetek kiértesítésénél egy UpdateView függvényre hivatkozik, amely eldönti, csupán egy újabb megjelenítésre van-e szükség, vagy át kell állítani az megjelenítendő objektumokat.

12.2 Architektúrális integráció

Ebben a részben csupán arról van szó, hogy mivel a kiegészítő eszközök különálló futó alkalmazások, a Microsoft Visual Studio .Net 2003-ban is külön projektekként jelennek meg. Azonban a létrehozott adatokat és információkat maga a játék fogja felhasználni, ezért a külön futó programokat mind fájl, mind adat szintjén kell, hogy kapcsolódjon vele.

Fájlszinten az integráció annyit jelent, hogy a szerkesztő program lehetőség szerint nem a saját könyvtáraiban dolgozik, hanem a játékprogram alkönyvtáraiban, továbbá hogy a fájlok előre megegyezett nyelven kell, hogy íródjanak. A .Net keretrendszert használó programok esetén az xml csak jó választás lehet, hiszen ez egy szabványos megoldás, amely a felhasználó által is olvasható, ellenőrizhető, és az előre megírt megoldásokkal (Xml Serialization, Xml Data Binding) könnyedén a memóriába tölthető.

Azonban hogy a fájlokban leírt adatok ténylegesen mind a két fél által azonos értelmezésen essenek át, a legegyszerűbb módszer, hogy reprezentált adatok struktúráját egy közös osztálykönyvtárba helyezzük el. Jelen esetben ez a GameLib osztálykönyvtár.

13 Szerkesztő programok

Jelen esetben a kiegészítő eszközök logikai háttere nem tartalmaz tudományos megfontolásokat, vagy kutatási területet. Céljuk csupán az, hogy az eddig megírt programot kiegészítsék valamilyen látványos, interaktív dekorációval. Mégis szeretném pár oldalban összefoglalni, hogy az egyes alkalmazásoknak mi a feladata, és milyen főbb funkciókat valósítottak meg.

13.1 Mesh szerkesztő

A játéknál hangsúlyt helyeztem a mesh objektumok helyességére, a vetett árnyékok készítése miatt. Ezért olyan eszközt terveztem meg, amellyel lehetőségünk van különböző, változatos alakzatok megmunkálására, olyan műveletekkel, amelyek garantálják a megszabott feltételeket. Ez nem más, mint hogy a mesh alakzatok szigorúan csak teljesen zárt fedő háromszöghálóval rendelkezhetnek.

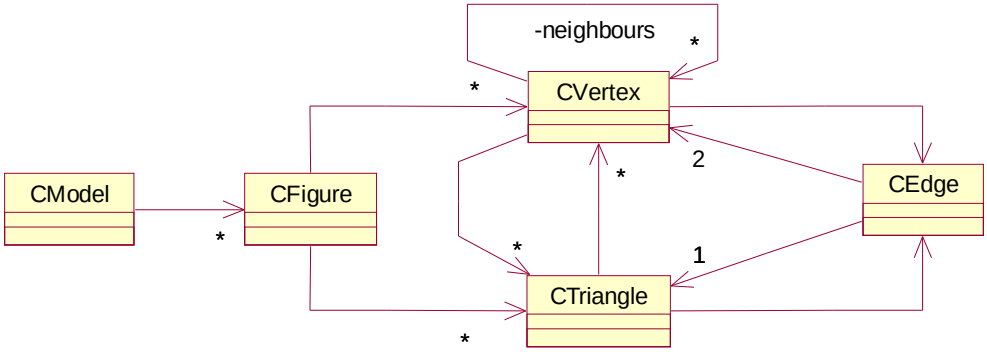
A szerkesztő a szoftvercsomagban található T3DCreator (Triangle 3D Creator) program, amely a nevét onnan kapta, hogy a modelleket háromszögek és azok vertex-einek szintjén van lehetőségünk megszerkeszteni.

13.1.1 Alapstruktúra

Mivel a cél az volt, hogy a program a megszerkesztett adatokat könnyen átírja mesh objektummá, az adatok struktúrájának felépítése tükrözi ezt.

A csúcspontokat, azaz vertex-eket a CVertex osztály reprezentálja, a háromszögeknek megfelelő CTriangle osztály ezekre hivatkozik. A CEdge osztály azonosítható a háromszög éleivel, azonban ennek az osztályra inkább azért van szükség, hogy megkönnyítsük vele a navigációt a különböző objektumok között.

Ezzel a szerkezettel nyomon tudjuk követni, hogy mely elemeknek mely másik elemek a szomszédjai, egyéb hozzátartozói, így könnyen megvizsgálható, hogy a háromdimenziós felület, amit létrehoztunk, zárt-e.



34. ábra: A T3DCreator modell felépítése

13.1.2 Műveletek

Ha a szerkesztés lépéseit matematikai operátorokkal azonosítjuk, kettős előnnyel jár.

Egyrészt formális bizonyítást adhatunk arra vonatkozóan, hogy a szerkesztett anyagunkra a megkötéseink igazak. Jelen esetben ez fordítva működik: A szerkesztés alatt csak és kizárólag olyan műveletek elvégzését engedjük meg, amelyről triviálisan látszik, hogy használatával a megkötések minden esetben fennmaradnak. Az ötletet az Euler-operátorok adták, amelyek lehetőséget adnak olyan térbeli modellek megszerkesztésére, melyekre érvényben marad a gráfok csúcsainak, éleinek, és azok közötti tartományok közt fennálló Euler-tétel. A mi esetünkben is hasonló gondolatmenetről van szó, itt a megkötés, amelyiket fenn akarjuk tartani, a testek felületének helyes felépítése, zártsága.

Másrészt a legtöbb műveletnek létezik inverz párja. A transzformációkat és inverzüket eltárolva folyamatos Undo-Redo, azaz Visszavonás-Újra lehetőséget biztosítunk a felhasználónak, jól megírt inverz műveletek esetén kevés erőforrás-pazarlással.

Az eszköz műveleteit a következőképp használhatjuk:

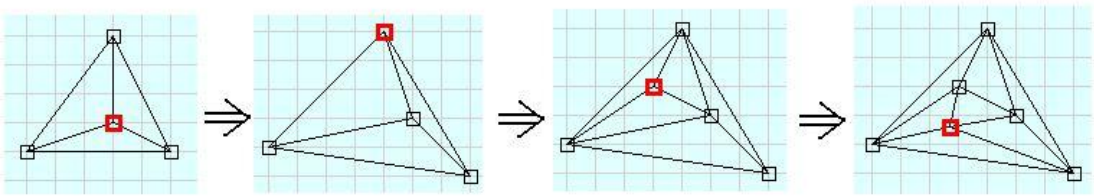
Első lépésben veszünk egy kész testet (Figure), amire már teljesül a zártsági feltétel. Ez valamilyen egyszerű forma lesz, például egyetlen tetraéder, vagy kocka. A testet a térben bármerre mozgathatjuk. A többi művelet az objektumot fedő háromszöghálóra vonatkozik.

A test vertex-eit megfogva elmozgathatjuk, így a háromszögeket egyenként manuálisan megfelelő alakúra igazíthatjuk.

Kiválaszthatunk egy háromszöget a testben, és egy középponti vertex behelyezésével három kisebb háromszögre oszthatjuk.

Kiválaszthatunk egy élet, és kettéoszthatjuk. Ezzel az élre bekerül egy újabb csúcs az élre, és az él két oldalán elhelyezkedő háromszögek a súlyvonal mentén megfeleződnek.

(Megjegyzés: könnyen belátható, hogy ezek a műveletek is Euler-operátorok.)



34. ábra: A T3DCreator modell felépítése

Egyetlen testből természetesen nem lehet bonyolult vagy nem összefüggő alakzatokat létrehozni, ezért a modell több ilyen formát tárol, amelyek egyetlen mesh-ként kerülnek elmentésre.

13.1.3 Textúrázás

A drótházós és a háromdimenziós megjelenítés mellett szerepet kapott egy harmadik grafikus nézet is, amely a textúrák helyes felhelyezését hivatott megoldani. A csúcspontokat a nézet a textúra-koordinátái alapján rávetíti a textúra képére, ahol őket mozgatva beállíthatjuk a helyes mintázatot.

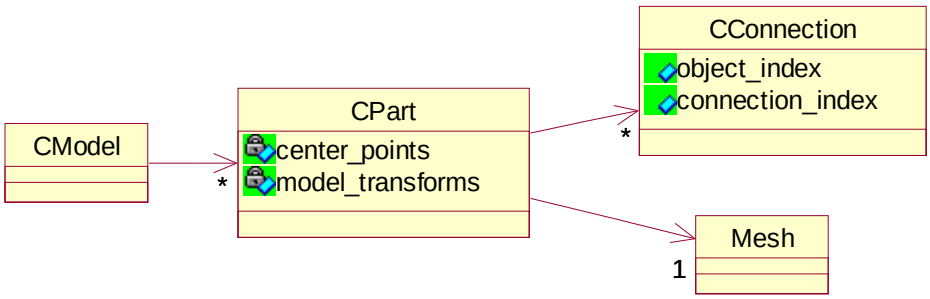
Nem minden test esetében lehet jól illeszteni a textúrát az alakzathoz, ilyenkor megadhatunk valamilyen automatikus kiosztást, amely a vertex adott pozíciójából számítja ki az új koordinátákat, vagy akár véletlenszerűen generálja őket.

13.2 Jelenetszerkesztő

Az AnimTool eszköz arra lett tervezve, hogy több előre – akár saját szoftverrel, akár más termékkel – elkészített mesh objektumokat összefűzzön, és megfelelő paraméterek beállításával animációt lehessen készíteni velük.

13.2.1 A modell struktúrája

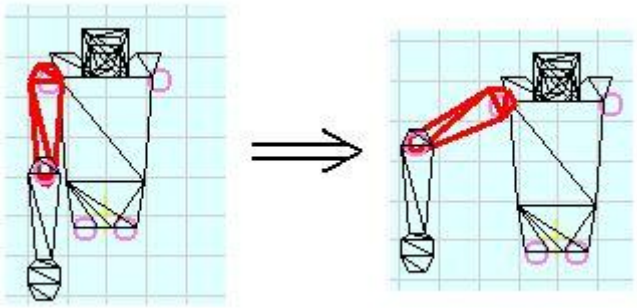
A modell felépítése ebben az esetben egyszerűbb, és mivel nem igazodtam semmilyen formátumhoz, a különböző osztályok csak a szükséges adatokat tartalmazzák.



35. ábra: Az AnimTool modell felépítése

A modell CPart elemekből épül fel, ami tartalmaz egy mesh-t, amelyet megjelenít, valamint a középpontjának helyzetét és az állását mutató transzformációs mátrixot. Ezen felül minden részlethez a szerkesztés folyamán rendelünk egy kapcsolati elemet (CConnection). Ezen keresztül illeszthetjük össze a különböző alakzatokat.

Egy részletet úgy rendelhetünk egy másikhoz, hogy az első kapcsolatán beállítjuk a megfelelő indexeket. Ez a kapcsolat egy irányú, az értelme pedig, ha a második elemet elmozdítjuk a helyéről, akkor az első vele együtt mozdul el, a másik oldalról megközelítve az első elemet csak úgy tudjuk változtatni, hogy a kapcsolati pontja mindvégig igazodni fog a másik elemhez.



36. ábra: A modell mozgatása

Ahhoz, hogy érvényes modellt kapjunk, azaz az illesztő algoritmus helyesen működjön, fontos betartanunk azt a szabályt, miszerint minden részletnek csak egyetlen kapcsolata lehet egy másik részlethez rögzítve, valamint nem lehet körben hivatkozni. Matematikai nyelven megfogalmazva a részletek által alkotott gráfnak szigorúan körmentes irányított erdőnek kell lennie. Ebben az esetben az egyes csoportok a gyökérelemen keresztül pozícionálhatóak, és a gyermek elemek forgatással módosíthatóak.

Az illesztés művelete egyszerű: Először is ki kell számolni, hogy a megadott transzformációkkal és eltolásokkal a részlet rögzített csatlakozási pontja hol helyezkedik el, majd ugyanezt el kell végezni a kölcsönhatásban szereplő másik részlettel is. A kettő különbségét kiszámítva megkapjuk, hogy mennyivel kell eltolni a rögzített részletet.

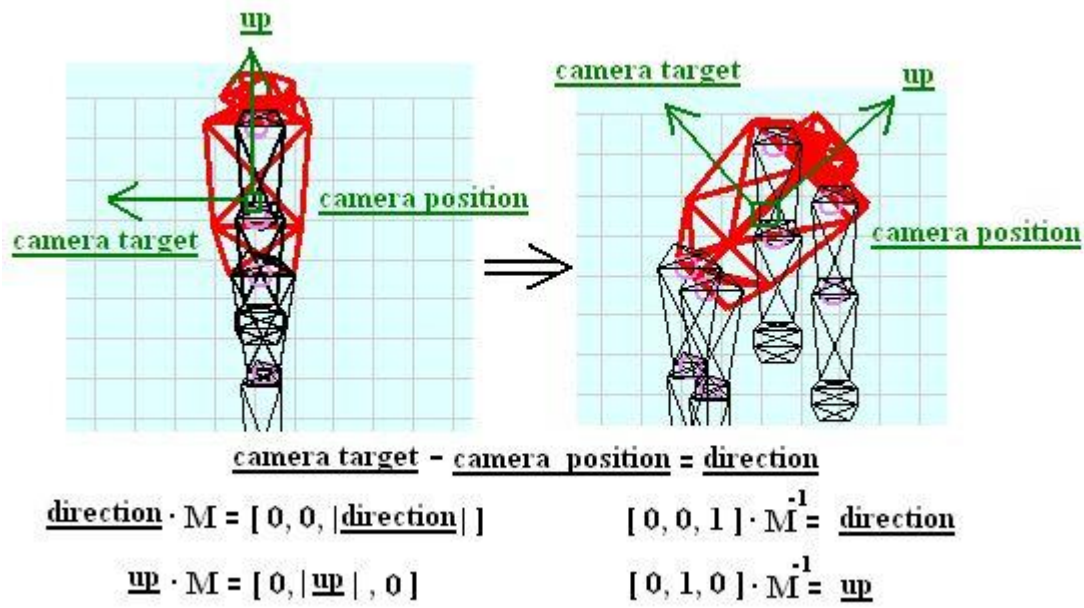
13.2.2 Mozdulatok összefűzése

Mivel minden mozdulatot egyenként beállítva ember feletti türelmet és kitartást igényel, a mozdulatok folytonosságát lineáris interpolációval biztosítottam. Ez az egyes részletek középpontjának eltolásánál nem okozott gondot, mivel szimpla vektorok interpolációja a Vector3 osztályban implementálva van. Ezzel szemben a transzformációs mátrixoknál egy trükköt kellett alkalmaznom.

Kihasználtam, hogy a transzformációs mátrixok az egységmátrixtól annyiban különböznek, hogy meg vannak szorozva néhány forgatást végző mátrixszal. Ezek a transzformációk azonban csak a térbeli koordinátákat változtatják meg, azaz értékei a bal felső 3x3-as részre korlátozódnak. Azonban minden olyan transzformáció, amely csak ezeket az értékeket használja fel, leírható egy speciális mátrix-konstrukcióval. Erre

közvetett bizonyíték szolgál, hiszen beláthatjuk, hogy a világ koordináta-rendszeréből a nézet koordináta-rendszerébe transzformáló mátrixok pontosan ezt teszik.

Jelöljük a világbeli koordinátákat nézeti koordinátákra konvertáló mátrixot M-mel. Ennek lényegéből fakad, hogy a kamerapozíciót a középpontba viszi át, a kamera célpontját egy Z tengelyen levő pontba, a felfelé mutató vektort az Y tengelyre helyezi, abban az esetben, ha a kamera iránya merőleges a felfelé mutató vektorra. Abban az esetben, ha a kamera pozícióját a világ koordináta-rendszerében is a középpontba helyezzük, és a felfelé mutató vektort merőlegesnek vesszük a kamera irányára, elmondható, hogy az M mátrix a kamera irányát egy tetszőleges hosszú, Z tengellyel párhuzamos vektorba viszi át.



37. ábra: A mátrixok interpolálása

Megfordítva a gondolatmenetet, az M mátrix inverzét képezve találhatunk egy olyan irányvektor-felfelé mutató vektor párost, amely egyértelműen meghatározza az M mátrixot. Ha feltételezzük tehát, hogy az adott részlethez tartozó modell transzformáció egy képzeletbeli nézeti koordinátarendszerbe átszámoló mátrix, az interpolálás a következőképpen megoldható:

Először a kezdeti és végső mozdulatok kinyerjük a meghatározó vektorpárost. Ezeket a vektorokat a Vector3 osztály Lerp függvényével interpoláljuk, majd minden lépésnél újra elkészítjük a nézeti transzformációk mátrixait.

13.3 Cselekményszerkesztő

Korábban már említettem, hogy a pályák véletlengenerátorának alapjául egy gráf-szerkezet szolgál, amely biztosítja a labirintusok kellő bonyolultságát és szabályozza annak bonyolultságát. Ezt az információt saját konvenció szerint cselekménynek hívom. Ennek megszerkesztését kezeli a StoryGenerator program, megvalósítást tekintve a legegyszerűbb a kiegészítő eszközök közül.

A modell struktúrájáért egyetlen osztály, a csomópont (Node) felelős, amely megadja, hogy mely másik csomópontokkal van, valamint, hogy az adott csomópontnak mik a fő paraméterei. Ezeket a főablakban megjelenő PropertyGrid segítségével állíthatjuk át. Megadhatjuk például a terem textúráját, vagy egy benne található mesh alakzatot.

A csomópontok a labirintus generálása során fő termekké válnak, amelyeket más termek kötnek össze. Ezt befolyásolja, hogy mely termek közé rendelünk a gráfban kapcsolatot. (Az algoritmus jellemzője ugyanakkor, hogy a véletlengenerálás miatt olyan termek lehetnek közvetlenül összekötve, melyek közt nem adtunk meg kapcsolatot.)

38. ábra: Egy cselekmény, és egy hozzá tartozó példalabirintus

Azonban fontos, hogy a megszerkesztett gráf megjelenítése semmit nem mond arról, hogy a főtermek a valóságban hol helyezkednek el, ezért lehetőségünk van előnézet (Preview) megtekintésére is. A program e funkciója generál egy példalabirintust, amelyen megfigyelhetjük a generált szintek jellegzetességeit.

14 Konklúzió, perspektívák

A diplomamunkám megmutatja, hogyan lehet – viszonylag rövid időn belül – egy teljes multimédiás alkalmazást megírni, háromdimenziós megjelenítéssel, a .Net Framework alá illesztett DirectX komponensek segítségével. Bár maga a játék nem sikerült kerek egészre, az implementáció további kiegészítései ebben az irányba terelhetik a fejlesztést.

A jövőben szeretném a játékot változatosabbá, játszhatóbbá tenni, igény szerint új kiegészítő eszközökkel bővíteni. Ezen kívül fontosnak tartom az állandó javításokat és optimalizációt, amelyekre ebben a környezetben különösképpen szükség van.

Irodalomjegyzék

[1] Albert – Balássy – Dr. Charaf – Erdélyi – Horváth – Levendovszky – Péteri - Rajacsics: A .Net Framework programozása
ISBN 963 9131 62 8

[2] Dr. Szirmay-Kalos László : Számítógépes grafika
ISBN 963 618 208 6

[3] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides: Programtervezési Minták
ISBN 963 930 1779

[4] Craig Andera : Managed Direct3D Tutorial
http://pluralsight.com/wiki/default.aspx/Craig.DirectX/Direct3DTutorialIndex.html

[5] Shadow, Trancparency, & Fog
http://developer.nvidia.com/object/shadows_transparency_fog.html

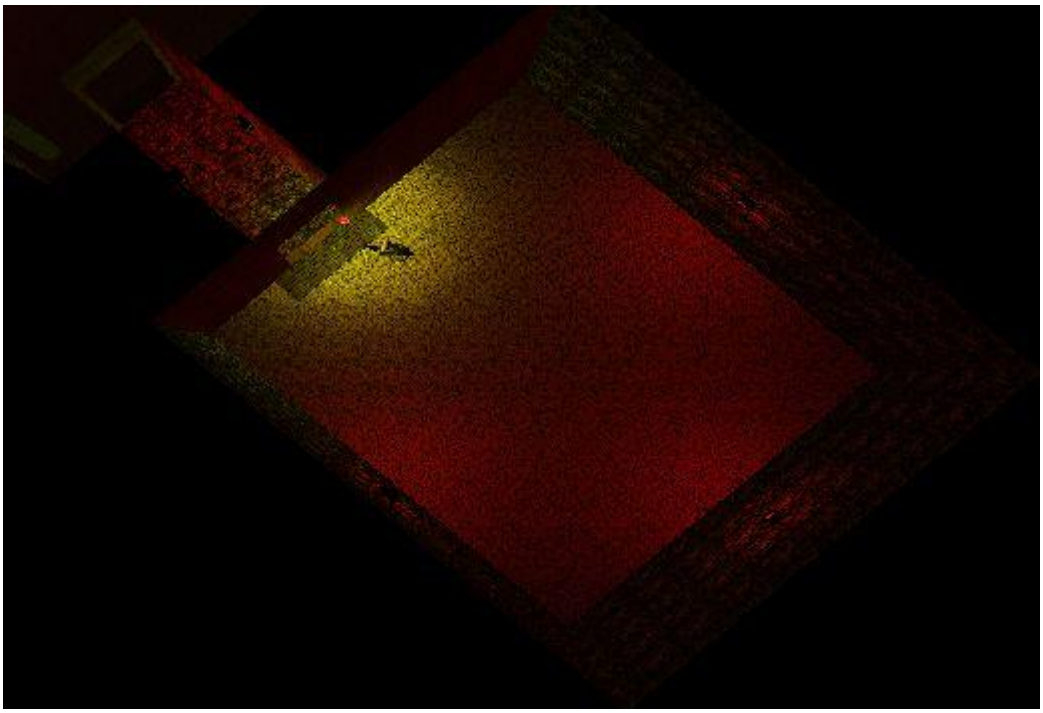
[6] Shadow Volume Sample
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/ShadowVolume_Sample.asp

[7] Hun Yen Kwoon : The Theory of Stencil Shadow Volumes
http://www.gamedev.net/columns/hardcore/shadowvolume/

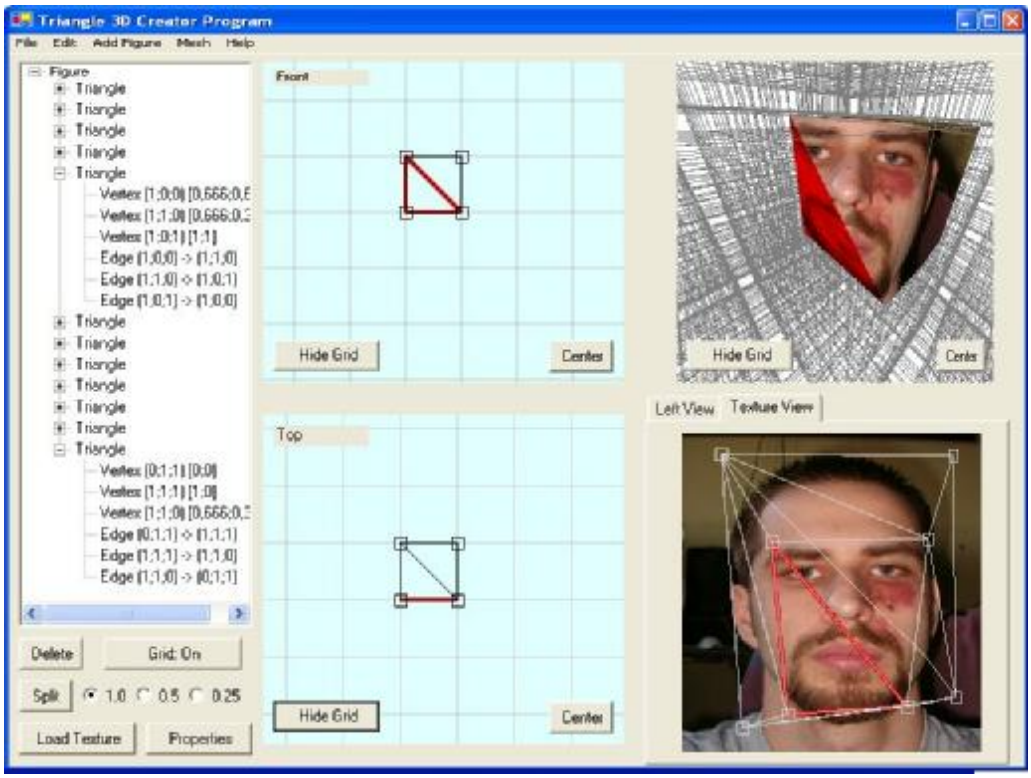
[8] Xml Focus Topics : Xml Data Binding
http://www.xml.org/xml/resources_focus_databinding.shtml

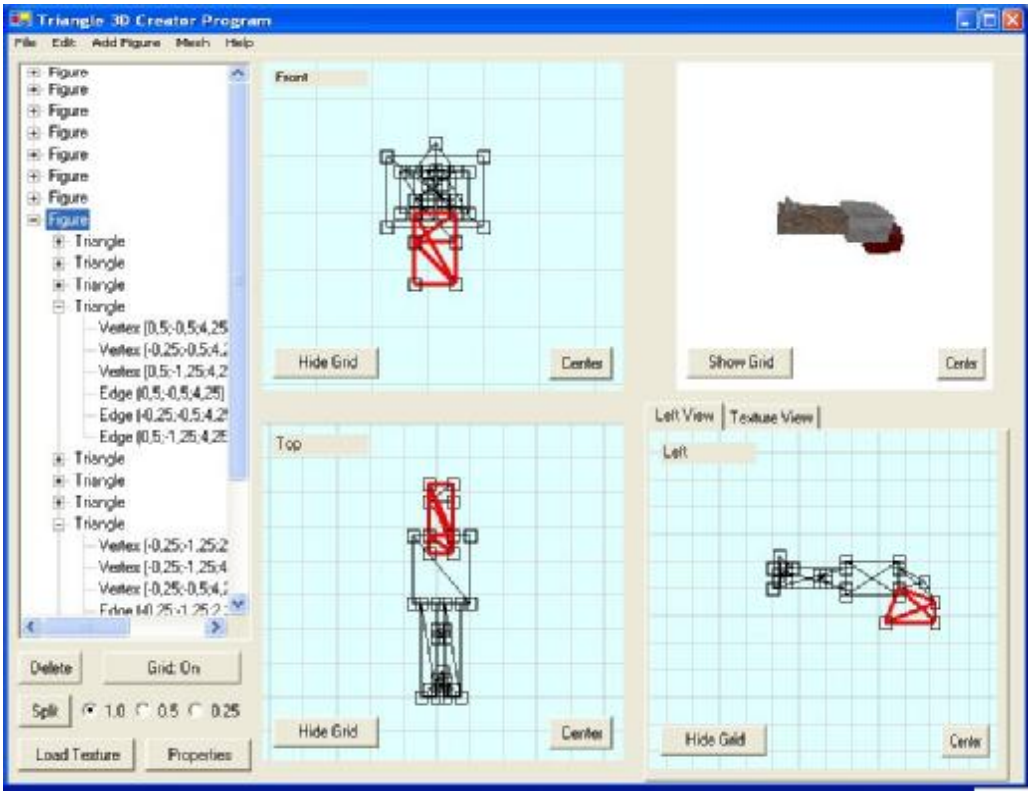
Függelék

F.1. ábra: A játékos vetett árnyéka

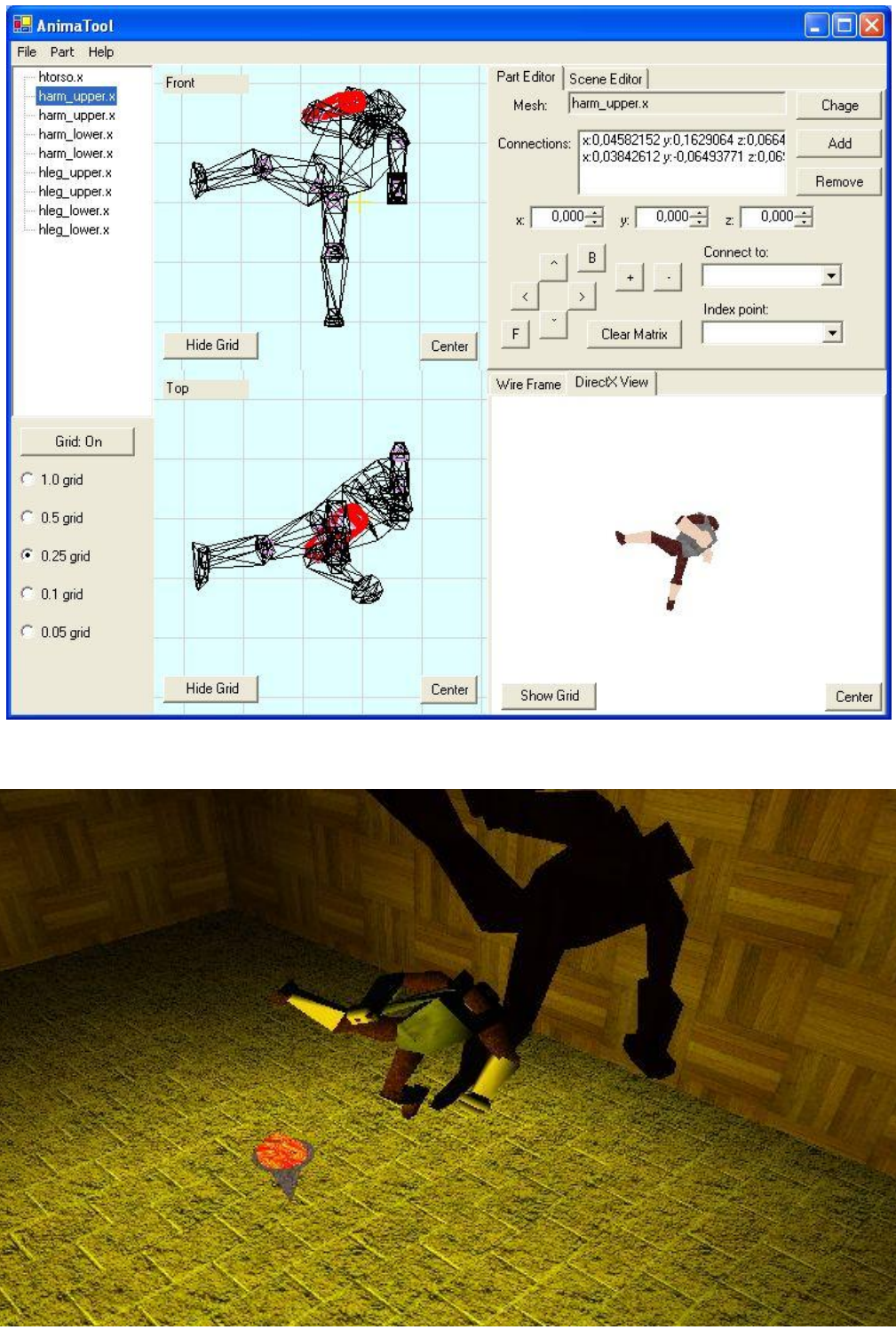


F.2. ábra: A játéktér felülnézetből

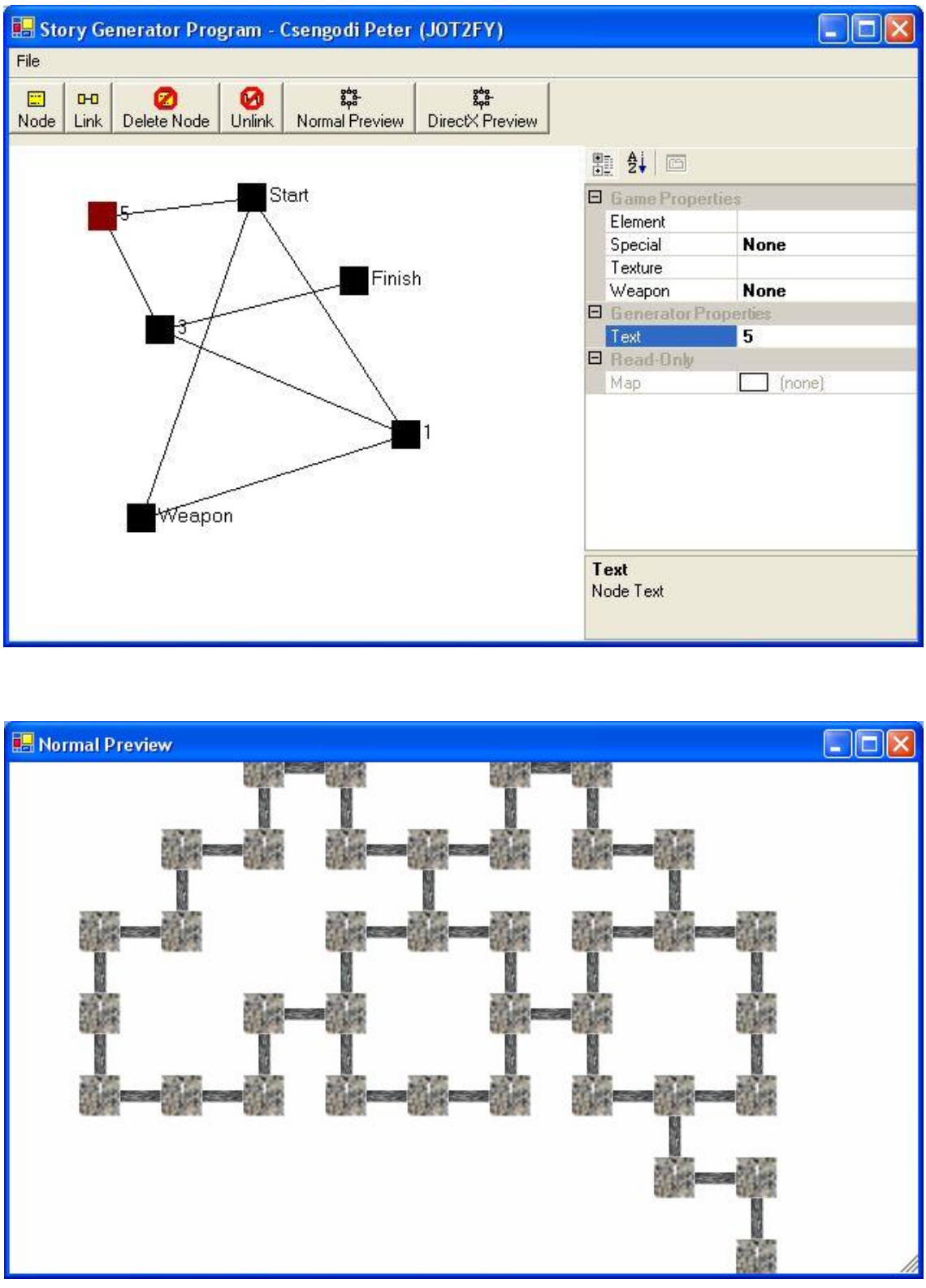




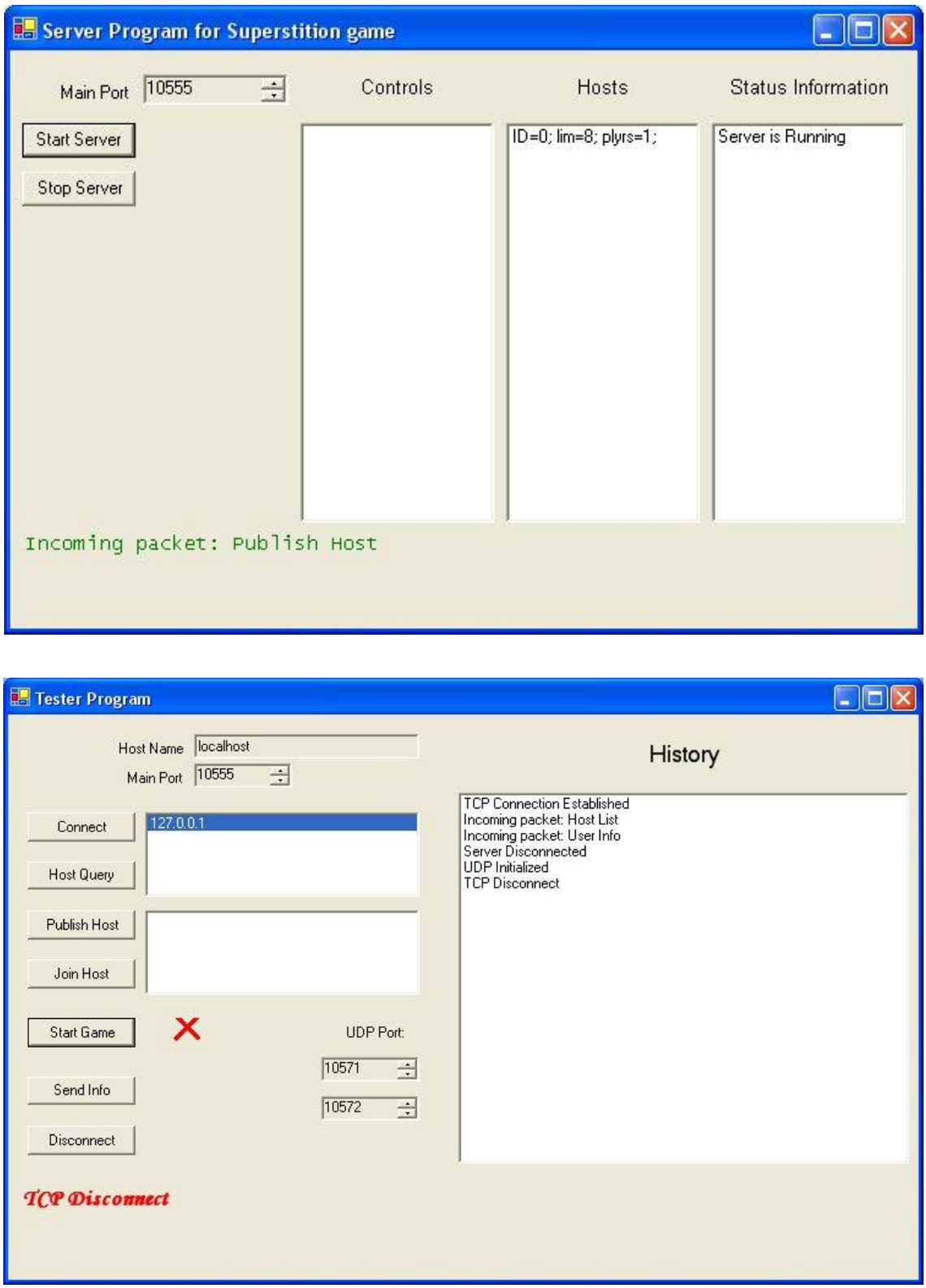
F.3.. ábra: A mesh szerkesztő



F.4. ábra: Egy animáció a szerkesztőben és a játékban



F.5. ábra: A cselekményszerkesztő és egy példalabirintus



F.6. ábra: A szerver és a tesztkliens futás közben