DIPLOMAMUNKA

Imre Norbert

Debrecen
2007

Debreceni Egyetem
Informatika Kar

OBJEKTUM ORIENTÁLT 3D MOTOR MEGVALÓSÍTÁSA C# ÉS .NET 2.0 ALATT

Témavezető:
Dr. Tornai Róbert
egyetemi adjunktus

Készítette:
Imre Norbert
programtervező
matematikus

Debrecen
2007

Tartalom

Bevezetés

Diplomamunkám célja bemutatni, hogyan építhető fel egy teljesen objektum orientált 3D motor, szem előtt tartva az optimalitást, a jól átláthatóságot és a könnyű bővíthetőséget. Az alkalmazást C# nyelven a Microsoft .Net Framework 2.0 eszközeit felhasználva fejlesztettem.

Az OpenGL függvénykönyvtár alapvetően eljárás-orientált szemléletet követve épül fel, a fő cél e függvények csoportosítása, majd osztályokba foglalása. Az osztályoknak a lehető legnagyobb mértékben követniük kell az OpenGL architektúrát, és szoros kapcsolatban kell állniuk az OpenGL környezetben létrehozható, a grafikus kártyák által hardver szinten támogatott objektumokkal.

A dolgozat célja, hogy bemutassa a motor felépítésének folyamatát, a tervezési, és implementálási fázisokat, a .Net és az OpenGL közötti alapvető kapcsolat megteremtésétől, egészen a végső, összetett osztályhierarchia megalkotásáig és a különböző látványfokozó technikák implementálásáig.

A motornak nem csak jól átlátható, objektum-orientált alapokon kell állnia, hanem mindezek mellett elég optimálisnak, gyorsnak is kell lennie. Olyanra kell tervezni, hogy a későbbiekben képes legyen befogadni más rétegeket, amelyek a tárgyak fizikai tulajdonságait definiálják, és valamilyen funkciót valósítanak meg. Elég erőforrást kell tartalékolni ahhoz, hogy akár egy játékot, vagy egy valós idejű, lakberendezési bemutató szoftvert megvalósító réteg beférjen mellé.

1. Alapvető kapcsolatteremtés

Ahhoz, hogy a motor jó alapokon álljon, találni kellett egy megfelelő keretrendszert, amely .Net alól lehetővé teszi az OpenGL környezet létrehozását, és használatát. Ehhez a legjobbnak a TAO keretrendszer bizonyult.

1.1. TAO Framework

A TAO Framework egy kis projectből nőtte ki magát. Mára már egy jelentős open-source keretrendszerré vált, amely folyamatosan fejlődik. Célja a különböző OpenGL-el kapcsolatos API-k használatának lehetővé tétele .Net alatt is. Én az 1.3.0 RC1 verzióját használtam, de már a 2.0 RC2 verziónál jár a fejlesztése. A jelentős különbség a két verzió között, hogy az 1.3-as verzió OpenGL 1.5-ig, a 2.0-ás viszont már OpenGL 2.1-ig tartalmazza az összes kiterjesztést. A keretrendszer tartalmazza még a GLU függvénykönyvtár 1.3-as verzióját is. A TAO Framework további API-kat is tartalmaz, ezek közül csak azokat említem, melyeket felhasználtam:

- Tao.OpenGL: A már említett OpenGL függvénykönyvtár implementációja, esetünkben az 1.5 verzióig, valamint ez tartalmazza a GLU 1.3 verziót is.
- Tao.FreeGlut: Az OpenGL-es berkekben jól ismert Glut függvénykönyvtár ingyenes változata.
- Tao.DevIl: Developers Image Library, képek betöltésére, különböző algoritmusokkal való manipulálására használható. Gyors és könnyedén összekapcsolható az OpenGL környezettel.

1.2. Tao.FreeGlut

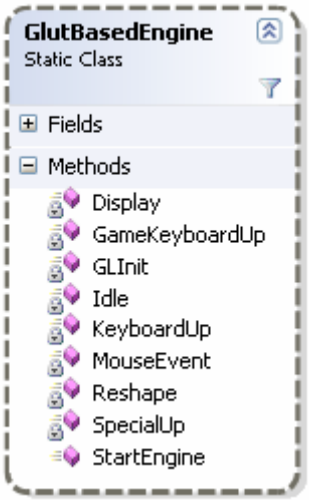
A FreeGlut függvénykönyvtár platform-függetlenül segít megvalósítani az OpenGL környezet felállítását, és összekapcsolását az aktuális megjelenítő rendszerrel, ami a mi esetünkben a Windows lesz. A Glut eszközeit használva a fejlesztőnek nem kell vesződnie ablak létrehozásával, majd ezen ablak látható felületének az OpenGL környezethez való kapcsolásával. A Glut megoldja még továbbá az OpenGL környezet alapvető beállítását, könnyedén lehetővé teszi a dupla képernyő puffer használatát, lehetőséget biztosít a

felhasználói interakciók megvalósítására, és egyszerűvé teszi az ablakos és a teljes képernyős módok közötti váltásokat.

1.3. Delegate-ek

A Tao.FreeGlut assembly .Net delegate-ek (MSDN Library – Delegate Class [2]) használatával paraméterezhető. Ezek lényegében függvény definíciók. Egy delegate definiálásakor meg kell adnunk a visszatérési értékét, a paramétereinek számát és típusát. Kapcsolatuk a függvényekkel (metódusokkal) lényegében úgy írható le, mint az osztályok kapcsolata az interfészekkel. Ha például egy metódus paraméterének típusa egy delegate, akkor ide paraméterül adható bármilyen metódus, amely ugyanolyan visszatérési értékkel és formális paraméterlistával rendelkezik, mint ami a delegate definíciójában szerepel. A .Net szabvány ezen még egy picit bonyolít, mivel a javasolt használati mód, hogy egy delegate helyére ne maga a konkrét metódus kerüljön, hanem egy delegate példány, amely konstruktorát a használni kívánt metódussal paraméterezzük. Ezután a létrehozott delegate ugyanazt a kódot fogja futtatni, mint amit az adott metódus tartalmaz. A delegate-ek sok helyen előjönnek a .Net használata során. Nagy jelentőségük van például a szálak közötti szinkronizáció megvalósításában is.

1.4. GLUTBasedEngine



1. ábra: GLUTBasedEngine osztály

A GLUT függvények egységbe foglalásához szükség volt egy konténer osztály létrehozására.

A *GLUTBasedEngine* (1. ábra) statikus osztály valósítja meg Glut függvényeken keresztül a motor magját. A motorban sok helyen statikus osztályokat használtam a könnyebb összekapcsolhatóság végett. Nem célkitűzés az, hogy egy program egyszerre több motort példányosítva, például több grafikus kártya használatával, több megjelenítőre egy időben rajzoljon. Ez a terület nyitott maradt, a statikus osztályok a későbbiekben helyettesíthetők nem statikusakkal, ez csak minimális többletadat tárolását jelenti, amit ennek az osztálynak kellhet majd megvalósítania.

A program, amely a motort használja a *StartEngine* metódus hívásával indíthatja el azt. Ennek paraméterei a kezdeti ablak szélessége, magassága és egy *String*, amely a létrehozandó Glut

ablak címsora lesz. Ez a metódus elvégzi a szükséges Glut inicializációkat, lepéldányosítja a megfelelő delegate-eket, amelyek az ábrán látható protected metódusokkal paraméterezettek, majd átadja ezeket a delegate-eket a megfelelő Glut függvényeknek. A Glut ezeket a főciklusában fogja tovább használni, amely majd külön szálon fut.

A metódusok nevei megegyeznek a Glut által előírt nevekkel, így azok funkcióit nem ismertetem részletesebben, feltételezvén a Glut általános ismeretét.

Bővebben erről az OpenGL Red Book [1] Introduction to OpenGL fejezetében olvashatunk.

Ez az osztály lehetővé teszi számunkra, hogy a *Display* metódusban, mintegy eljárás-orientált módon, OpenGL függvények hívásának sorozatával létrehozzuk, és megjelenítsük a kívánt kép elemeit.

A célunk viszont ezzel szemben az, hogy ezeket az OpenGL függvényhívásokat valamilyen módon csoportosítsuk, majd osztályokba rendezzük, ezzel elérve azt, hogy minden osztály az általa képviselt elem megjelenítéséért legyen felelős.

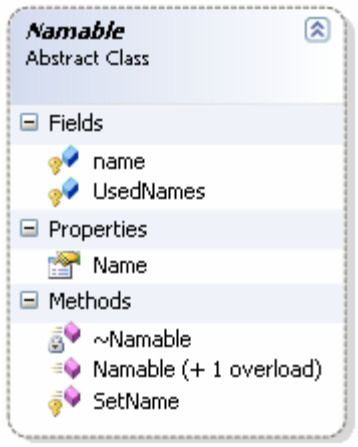
2. Osztály hierarchia

Ez a fejezet szorosan kapcsolódik az előzőhöz, mivel az alábbi osztályok bemutatása nélkül nem érthető meg a motor inicializációs, újrainicializációs és renderelési viselkedése.

A függvényhívásokat tehát, az alapján kell egységekbe foglalnunk, hogy a kép milyen elemeihez kapcsolódnak. Majd az egységeket osztályokba rendezzük, a közös tulajdonságaikat, és funkcióikat kiemeljük, és ősosztályokba helyezzük.

A tervezésben fontos szerepet kapott az egységesség. Ennek legfőbb eleme egy olyan ősosztály megalkotása volt, amely segíti a példányok egyedi azonosítását. Hogy emberi szemmel könnyebben átlátható legyen, ezt az egyedi azonosítást *String* típusú nevek végzik. Innen ered a *Namable* osztály elnevezése.

2.1. Namable



2. ábra: Namable osztály

A *Namable* (2. ábra) absztrakt osztályból származik minden egyes osztály, amely példányai futásidőben elérhetőek és manipulálhatóak a GUI (grafikus felhasználói interfész) felületek segítségével. Ez az osztály fontos eleme az egységes architektúrának.

A külvilág felé a *Name* property látható, amelyen keresztül lekérhető és beállítható az adott példány neve. Ennek a névnek egyedinek kell lenni, ezt az osztály a statikus *UsedNames* lista kezelésével éri el. Ebben a listában nyilván van tartva minden egyes kiosztott név. A destruktor végzi az adott példányhoz tatozó név felszabadítását, hogy az ezután újra felhasználható legyen.

Két konstruktora van, az egyik paraméterezetlen, ez egy alapértelmezett nevű példányt hoz létre, ehhez minden leszármazott osztálynak biztosítania kell a nevek indexelését. Erre csak a nagyobb csoportokat magában foglaló osztályoknál van szükség, mint például a *Solid* vagy *Texture*. Ha ezek valamely leszármazott osztályában létrehozunk paraméterezetlen konstruktorral egy példányt, az a „*SolidX*” vagy a „*TextureX*” nevet kapja majd, ahol *X* a következő szabad index. Tehát például a *LoDSolid* osztály példányát létrehozva annak „*SolidX*” neve lesz. A könnyebb azonosíthatóság végett ez a módszer lentebb vihető a

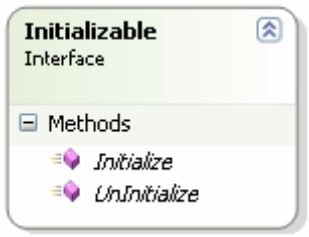
hierarchiában, tehát a *LoDSolid* felülírhatja a paraméter nélküli konstruktort, ezzel lehetővé téve, hogy a példányai a „*LoDSolidX”* nevet kapják.

A másik konstruktor egy *String*-et kap paraméterül, mely az új példány neve lesz, ez a konstruktor egyelőre nem használatos.

A hierarchia alapja öt egyszerű interfész, melyek az osztályok csoportosítására, viselkedésének leírására szolgálnak. Az interfészek szerepe az osztályok közötti kapcsolat megteremtésében van. Az interfészek kiemelnek egy-egy fontos tulajdonságot az osztály tulajdonságai közül, így egy másik, hivatkozó osztálynak elég azt tudni, hogy milyen interfészt implementáló osztállyal akar dolgozni, azaz milyen tulajdonságaik alapján akarja a többi osztály példányait elérni.

Mielőtt megismerkednénk a hierarchia magjával, néhány szóban szeretném bemutatni a készített interfészeket, és szerepüket.

2.2. Initializable



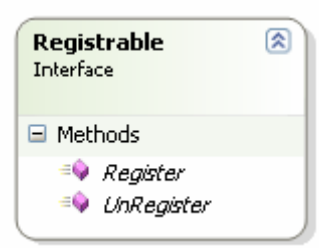
3. ábra: Initializable interfész

Az *Initializable* interfészt (3. ábra) azoknak az osztályoknak kell implementálniuk, amelyeknek szüksége van az OpenGL környezet felállítása után inicializációra (például display list-ek, textúra objektumok létrehozása). Erre azért van szükség, mert a példányok létrehozása kezdetben még az OpenGL környezet felállítása előtt történik meg a *Sceene* osztály használatával, ahogy ezt később majd látni fogjuk. Tehát ha a konstruktorba OpenGL hívásokat helyeznénk, akkor azoknak semmi hatása nem lenne, mert még nincs működő környezet.

Az *UnInitialize* metódusban a példányoknak le kell törölniük mindazon OpenGL objektumokat, melyeket használtak (például display list, textúra objektum, stb.). Ennek jelentősége csak akkor van, ha futás közben szerkesztjük a jelenetet, és ezáltal letörlünk példányokat, amelyeknek ekkor fel kell szabadítaniuk a foglalt erőforrásaikat a grafikus kártyán.

A program futásának befejezésekor a felépített OpenGL környezet lebomlik, így felszabadul minden foglalt erőforrás, tehát itt nincs szükség erre a metódusra.

2.3. Registrable



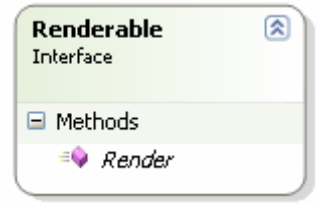
4. ábra: Registrable interfész

A *Registrable* interfészt (4. ábra) minden olyan osztálynak implementálni kell, amely szerepet akar kapni a motor főciklusában. A később ismertetett *EntityContainer* statikus osztály *Add* metódusának paraméteréül, ilyen interfészt implementáló osztálybeli példányt adhatunk meg. Ez a metódus semmi mást nem tesz, csak meghívja a kapott példány *Register* metódusát. A *Register* metódusban az osztályoknak hozzá kell adni magukat az *EntityContainer* megfelelő listáihoz, azaz tudniuk kell hovatartozásukról. Ez fogja majd később befolyásolni például, hogy milyen sorrendben kerülnek megjelenítésre.

Az *UnRegister* metódusban a példánynak le kell törölnie magát az *EntityContainer* mindazon listáiról, amelyekre előzőleg regisztrált, így kikerül a motor főciklusából. Erre akkor van szükség, ha törölni akarunk egy példányt, vagy például *Solid* esetén, áthelyezzük egy *SolidGroupba*.

Az *EntityContainer*, *Solid* és *SolidGroup* osztályokkal később foglalkozunk.

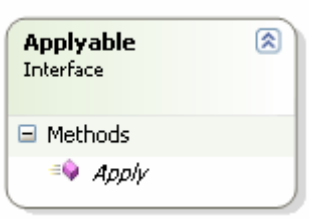
2.4. Renderable



5. ábra: Renderable interfész

A *Renderable* (5. ábra) interfészt azon osztályoknak kell implementálniuk, amelyek részét képezik a kép renderelésének, például *Solid* és *SolidGroup* osztályok, amelyek a látható tárgyakat és azok csoportját reprezentálják, vagy például a *ReflectionTexture* osztály, amely valós idejű tükröződő textúrát reprezentál. A *Render* metódusuk minden képkockában legalább egyszer meghívódik. Az *EntityContainer*-ben valamely render listába kell regisztrálniuk magukat, attól függően, hogy milyen sorrendben, vagy milyen renderelési fázisban kell a megjelenítésüknek bekövetkezni.

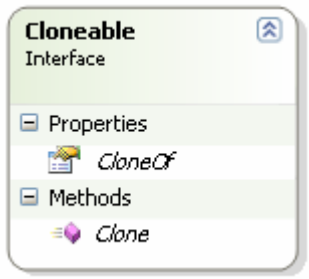
2.5. Applyable



6. ábra: Applyable interfész

Az *Applyable* interfész (6. ábra) nagyon hasonlít a *Renderable*-re, azzal a különbséggel, hogy ezt olyan osztályok implementálják, melyek nem közvetlenül megjelenő objektumokat reprezentálnak, nem szerepelnek render listában, hanem az utánuk megjelenítésre kerülő objektumok megjelenését befolyásolják. Például a *Material* és *Texture* osztályok, melyek az alkalmazásuk után megjelenített tárgyak felületének anyagát és mintáját állítják be.

2.6. Cloneable



7. ábra: Cloneable interfész

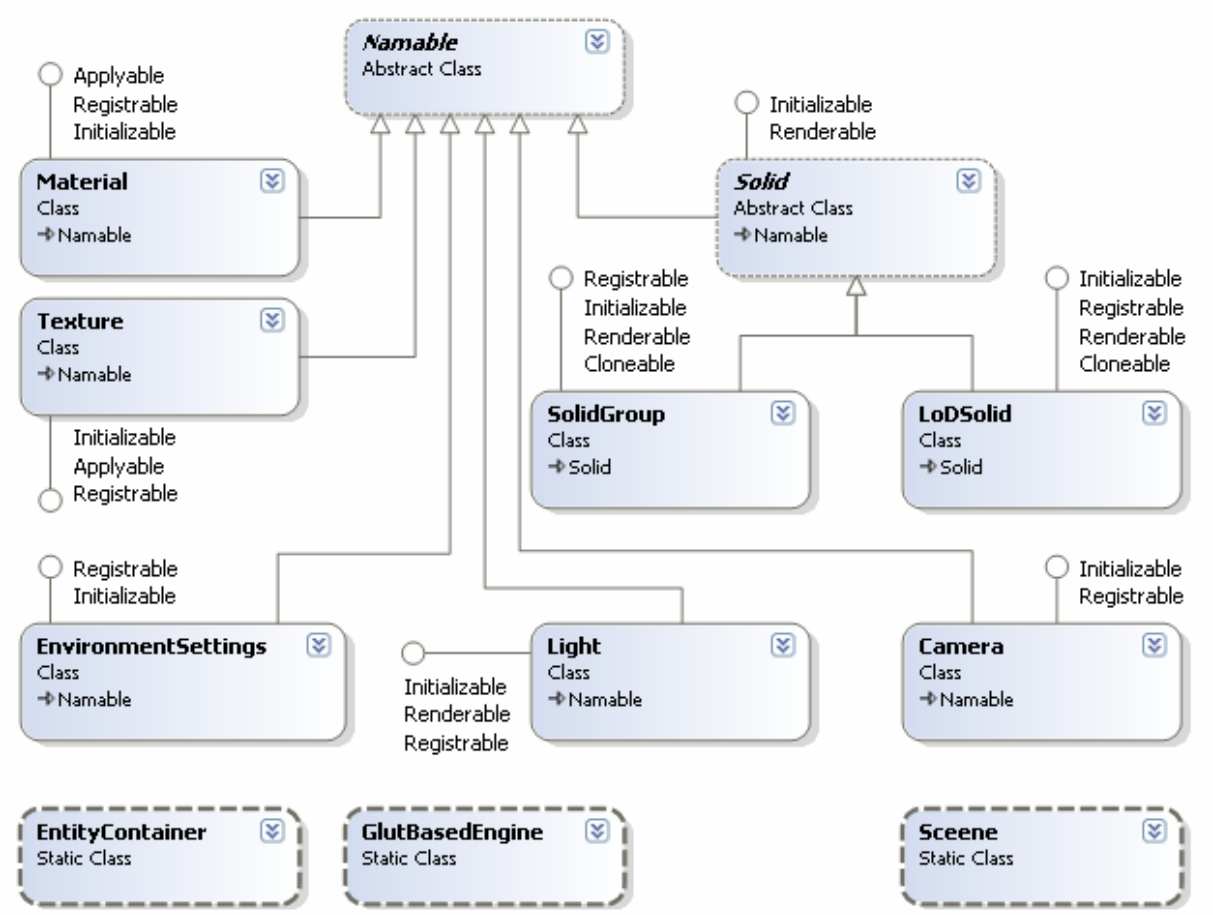
A *Cloneable* interfészt (7. ábra) olyan osztályok implementálják, melyek klónozható objektumokat reprezentálnak. A klónozás nagy jelentősége optimalizálásnál van, jelentősen csökkenthető a grafikus memóriában történő helyfoglalás, mivel az egyforma objektumok csak egyszer vannak letárolva. Ettől függetlenül minden klónnak saját transzformációs mátrixa van, így egymástól teljesen függetlenül mozgathatók, forgathatók.

A *CloneOf* property segítségével nyomon követhető, hogy egy adott példány klón-e, és ha igen, akkor melyik másik példányból jött létre. Ennek a jelenet elmentésénél van nagy szerepe.

2.7. Alapvető osztályok

Az alábbi osztály-diagramm (8. ábra) szemléletesen leírja az egyes osztályok hierarchián belüli helyét, kapcsolatát a többi osztállyal, valamint a hierarchián kívüli, de a működés szempontjából elengedhetetlen osztályokat is.

Az osztályok szerepe, és kapcsolata hosszas tervező munka eredménye, amely során figyelembe kellett venni a képalkotásban betöltött szerepüket valamint, hogy milyen OpenGL függvények találhatóak mögöttük, és hogy milyen hatást fejtenek ki az OpenGL környezetre.



8. ábra: Az osztályhierarchia alapját képező osztályok

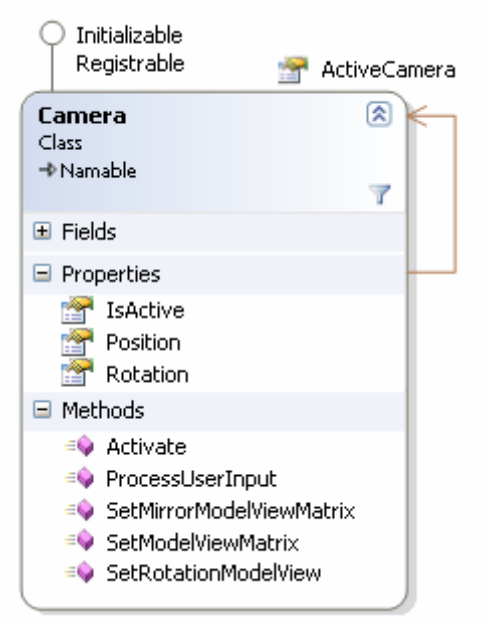
A *Namable* osztállyal már megismerkedtünk. A most bemutatásra kerülő osztályok a legalapvetőbb feladatok ellátásáért felelősek. Nem minden, az ábrán látható osztályról lesz a következőkben szó. A kimaradt osztályokkal részletesebben foglalkozunk majd a *Megvalósító osztályok és technikák* fejezetben.

Ahogy arról már volt szó, lesz majd egy konténer osztály, amelynek szerepe az összes példány nyilvántartása, listákba rendezése funkciójuk szerint, és a funkciókat megvalósító metódusaik hívása. Az *EntityContainer* osztály részletes megismeréséhez, előbb látnunk kell az osztályokat, amelyek példányait nyilván tartja majd.

2.7.1. Camera

A *Camera* osztály (9. ábra) példányai egy-egy kamerát reprezentálnak. Kikötés hogy mindig legalább egy kamerának lennie kell, és az összes kamera közül mindig csak egy lehet aktív. Az aktív kamera szemszögéből történik a kép kirajzolása, és az aktív kamerát mozgathatja a

felhasználó a billentyűzet és egér segítségével, mintha benne ülne, ha a *FlyMode* (lásd *EnvironmentSettings* osztály) aktív.



9. ábra: Camera osztály

Az aktív kamera a statikus *ActiveCamera* property-n keresztül érhető el, egy adott példány *IsActive* property-je jelzi, hogy épp ő-e az aktív kamera, valamint ezen property igazra állításával tehető aktívvá az adott példány.

A *Position* és *Rotation* property-k rendre a kamera helyzetét és az egyes tengelyeken való elforgatását adják meg, ezek a GUI felületen manuálisan is beállíthatóak, de az egérrel és billentyűzettel való mozgatás is ezeket manipulálja.

Visszatérve a *GLUTBasedEngine* osztályhoz: A *Display* metódus minden egyes képkocka renderelése előtt beállítja a *ModelView* mátrixot az aktív kamera *SetModelViewMatrix* metódusának hívásával. Az *Idle* metódusban meghívódik az aktív kamera *ProcessUserInput* metódusa, amely aszinkron Windows API függvényhívások segítségével feldolgozza a lenyomott billentyűket és az egér helyzetének változását.

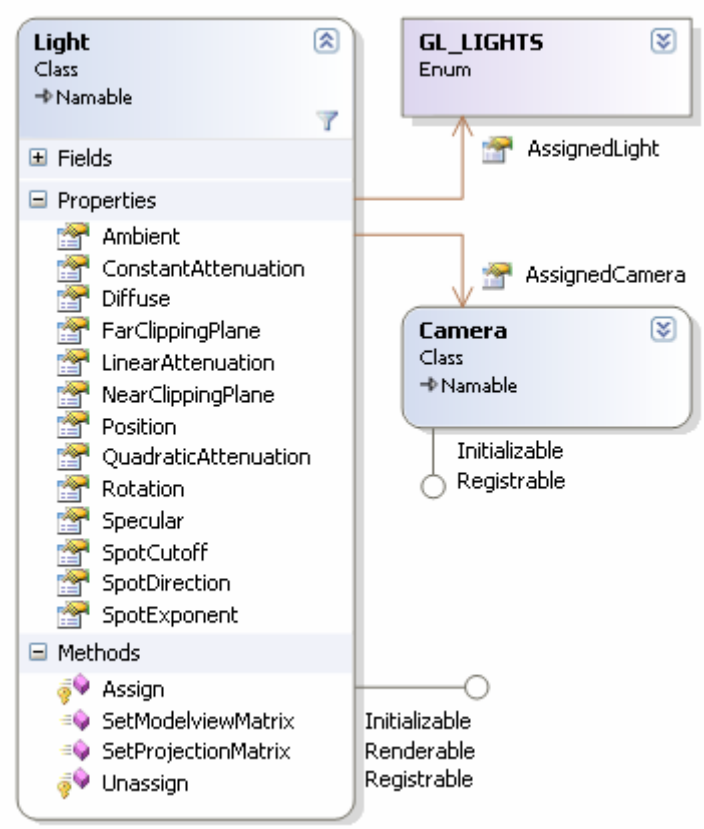
A *SetMirrorModelViewMatrix* metódus szerepét majd a *ReflectionTexture* osztály tárgyalásánál fejtjük ki. A *SetRotationModelView* metódus pedig a fényforrások kamerával való összekapcsolásánál, a fényforrás kamerával együtt való mozgatásánál fog szerepet kapni.

A mátrixok szerepét és működését nem fejtem ki részletesebben. Bővebb információ róluk az OpenGL Red Book [1] Viewing fejezetében található.

2.7.2. Light

A *Light* osztály (10. ábra), a fényforrásokat reprezentáló osztály. Az OpenGL környezetben az egyidejűleg használható fényforrások száma minimum nyolc. Ez az osztály bevezet erre egy korlátozást az *AssignedLight* property segítségével. Tetszőleges számú fényforrást létrehozhatunk, de ezekből alaphelyzetben csak az első nyolc kap érvényes GL_LIGHTi azonosítót (i = 0, 1…7). Az érvényes azonosítóval nem rendelkező fényforrások rendereléskor figyelmen kívül lesznek hagyva. Ha egy fényforrás megadott távolságon kívül

kerül az aktív kamerától, akkor lehetősége van az *UnAssign* metódus segítségével elengedni a hozzárendelt azonosítót, így hagyva más, közelebb levő, fényforrásokat szerephez jutni. Amikor megint közel kerül, az *Assign* metódussal megpróbál magának érvényes azonosítót szerezni. Ha van szabad azonosító, akkor ismét megjelenített fényforrássá válik.



10. ábra: Light osztály

Megjegyzés: A nagy mennyiségű dinamikus fényforrások használata még a mai grafikus processzoroknál is jelentős mértékű teljesítménycsökkenést okoz. Ennek kiküszöbölésére születtek különböző módszerek, például a *Light Map* technika, amely a fényforrást, mint egy textúrát képzi le azokra a testekre, amelyekre hatással van. Ez a technika statikus fényforrások esetén jelentős teljesítménynövekedést eredményez, de itt a továbbiakban nem foglalkozunk vele.

Lásd: Stefan Brabec, Hans-Peter Seidel – Extended Light Maps [3].

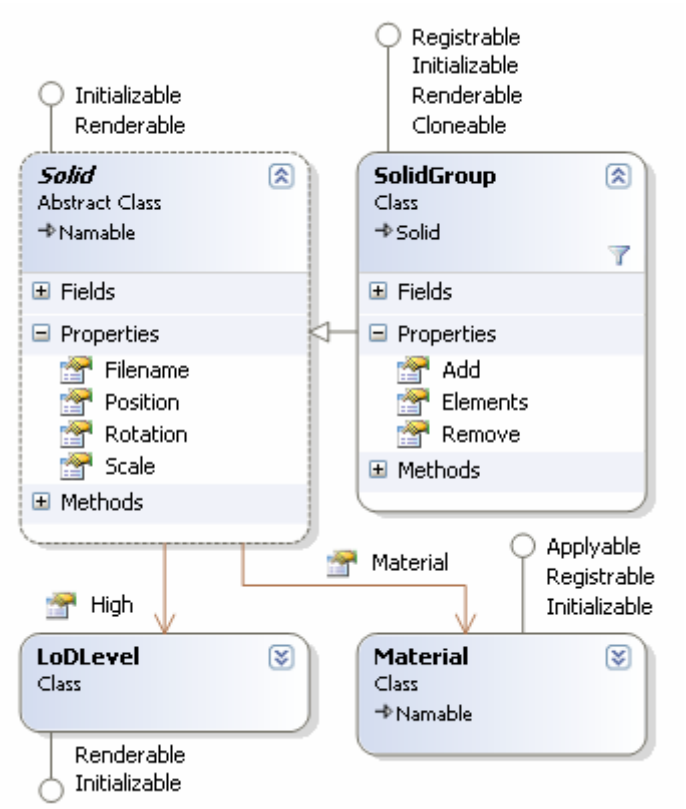
Az *AssignedCamera* property segítségével a fényforrás összekapcsolható egy kamerával, majd amikor a felhasználó a kamerát aktívvá teszi, és mozgatja, a fényforrás együtt mozog a kamerával. Ennek az irányított fényforrások beállításánál van nagy jelentősége.

Az *Ambient, Diffuse, Specular, Position, ConstantAttenuation, LinearAttenuation, QuadraticAttenuation, SpotCutoff, SpotDirection* és *SpotExponent* propertyk a fényforrás paramétereinek beállítására szolgálnak, és teljes mértékben megfelelnek az OpenGL specifikációban leírtaknak.

Bővebb információ az OpenGL Red Book [1] Lighting fejezetében található.

A *Rotation, Near-* és *FarClippingPlane* property-k és a *SetModelviewMatrix*, valamint a *SetProjectionMatrix* metódusoknak a vetett árnyékoknál lesz majd jelentőségük.

2.7.3. Solid és SolidGroup



11. ábra: Solid és SolidGroup osztályok

A *Solid* osztály (11. ábra) egy absztrakt osztály, ez az ősosztálya mindazon osztályoknak, melyek látható, úgymond kézzel fogható tárgyakat reprezentálnak. A *Position*, *Rotation* és *Scale* property-k segítségével a transzformációs mátrix állítható be, ezek rendre az objektumok helyzetéért, a középpontjuk körüli elforgatásukért és a skálázásukért felelősek.

A *Filename* property segítségével a testek fájlokból való betöltése valósítható meg. Ha ezt a property-t egy fájl nevére állítjuk, a példány automatikusan elvégzi a fájlban található vertex, face és normál koordináták betöltését, valamint ha a fájlban több LoD szint van, akkor azoknak a betöltését.

A *Material* property az adott testhez tartozó anyag beállításokat tartalmazó osztály. Ez végzi az egyes textúra szintekhez rendelt *Texture* példányok alkalmazását és beállítja a test felületének fényvisszaverési paramétereit, ahogy ezt majd a későbbiekben látni fogjuk.

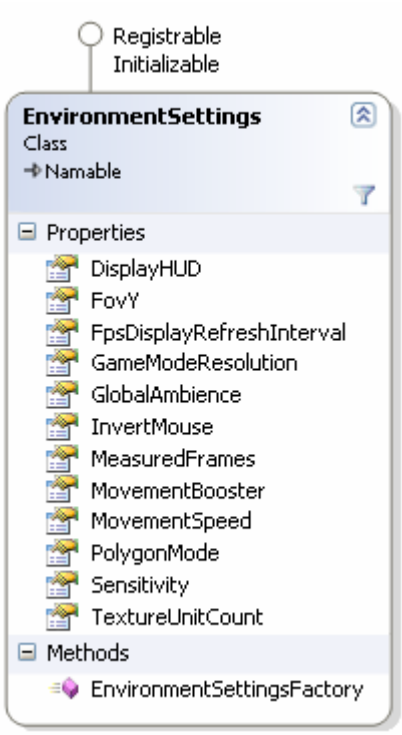
A *High* property elnevezését majd a LoD technikák, és a *LoDSolid* osztály tárgyalásánál fogjuk tisztázni, egyelőre csak annyit, hogy ez egy *LoDLevel* osztálybeli példány, amely a testhez tatozó legnagyobb felbontású LoD szintet reprezentálja, kezeli az ehhez tartozó display list-et, és újraépíti azt, a meglévő információk alapján, ha arra szükség van. A motor szerves részét képezi a LoD megvalósítása, ezért jelenik meg már ilyen szinten is ez a fogalom, viszont a *Solid* mint olyan nem valósítja meg a LoD technikát, ezért azt úgy tekintjük, mintha csak egyetlen, nagy felbontású LoD szintje lenne. Természetesen lehetőség van e szint, és a hozzá tartozó elemek teljes figyelmen kívül hagyására, ha egy olyan *Solid* leszármazott osztályt kívánunk bevezetni, amelynek nincs szüksége ilyesmire.

A *SolidGroup* osztály már nem egy absztrakt osztály. Ez *Solid* osztálybeli példányok csoportba foglalását, a teljes csoport egy testként kezelését, és klónozását teszi lehetővé. Így a *Solid* property-ei közül az előző hármat (*Filename, Material, High*) el is rejti, mert ezek nem használhatók egy csoportnál.

Megjegyzés: A célkitűzések között szerepel a *Filename* property implementálása, amely lehetővé tenné teljes csoportok script fájlból való betöltését, ugyanolyan módszerrel, mint amit majd a *sceene* fájlok betöltésénél ismertetek.

A *Solid* osztály tartalmaz egy display listet, ennek neve *TransformationDisplayList*, melybe az inicializáláskor belefordítódik az adott példány transzformációs mátrixát létrehozó OpenGL függvényhívások csoportja. Nem tartozik ide a *Modelview* mátrixot kiürítő művelet, helyette a mátrix verembe lementésre kerül az aktuális *Modelview* mátrix, majd hozzászorzódnak az adott példány, pozícionálás, elforgatás és skálázás mátrixai, melyek megvalósítására a függvénykönyvtár biztosítja az eszközöket. Miután megtörtént a *Solid* renderelése, a mátrix veremből visszaállítódik a *Modelview* mátrix előző állapota. Ez a működés lehetővé teszi a *SolidGroup*-ok esetében a tetszőleges mélységben egymásba ágyazott transzformációk megvalósítását.

2.7.4. EnvironmentSettings



12. ábra: EnvironmentSettings osztály

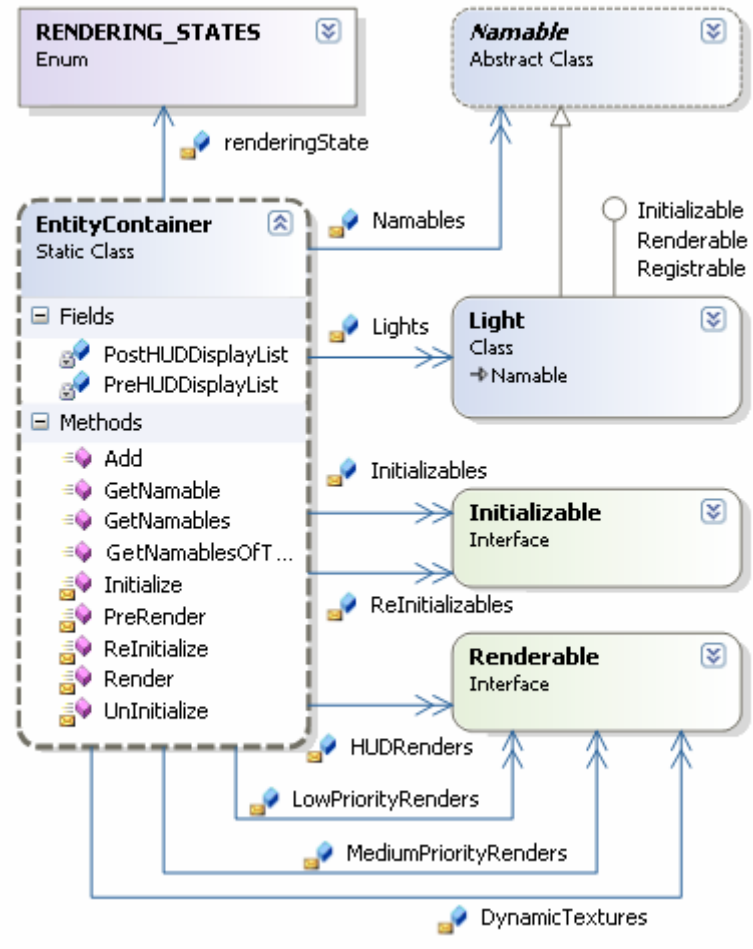
Az *EnvironmentSettings* (12. ábra) osztály egy eredetileg statikus osztálynak tervezett osztály. Később vált mégis nem statikussá, hogy futás időben, az egységes architektúrába illeszkedve, manipulálható legyen. Emiatt ennek az osztálynak nincs látható konstruktora, a példányosítást az *EnvironmentSettingsFactory* statikus metódus végzi, mely egyetlen példány létrehozását teszi lehetővé, ennek property-jei pedig a statikus területen tárolt adatokhoz engednek hozzáférést. Ennek a módszernek a jelentősége majd a GUI felületek tárgyalásánál válik világossá.

Az osztály property-jei a különböző környezeti beállítások módosítását végzik, ez az osztály a fejlesztés során folyamatosan bővül a felhasználó által módosítható, a motor

globális működésére kiható tulajdonságok megjelenésével. Az egyes property-k szerepére nem kívánok kitérni, némelyik csak a fejlesztő számára bír jelentőséggel, a többi pedig a neve és a program futása során megjelenített leírása alapján magától értetődő.

Megjegyzés: A *FlyMode* mező az, amire korábban már hivatkoztam, ennek szerepe ki- vagy bekapcsolni a repülés módot. A repülés mód alatt az egér az ablakhoz van 'láncolva' és a kamera forgatását végzi, a billentyűzeten a *W, S, A* és *D* betűk pedig a kamerát mozgatják. Ha a repülés mód inaktív, az egér szabadon mozgatható. Ez a tervezéshez, a GUI felületek használatához szükséges. (A *FlyMode* állapota a jobb egérgomb segítségével változtatható.)

2.7.5. EntityContainer



13. ábra: EntityContainer osztály

Az *EntityContainer* (13. ábra), statikus osztály szolgál az összes egyéb osztály példányainak nyilvántartására. Listákban tárolja az egyes példányokat, minden példány abban a listában található, amelyikbe a szerepe szerint beleillik. Az osztályok tudják, hogy példányaik mely listákba kell, hogy kerüljenek, ahogy erről már a *Registrable* interfész leírásában szó volt.

A *GLUTBasedEngine* osztály *GLInit* metódusa meghívja az *EntityContainer* osztálynak, az *Initialize* metódusát, amely végig hívja az összes *Initializables* listára regisztrált példány *Initialize* metódusát. Ugyanez történik a *ReInitializables* lista példányaival, mikor az ablak átméretezésének hatására meghívódik a motor *Reshape* metódusa. A *Namables* listára be kell regisztrálnia mindegyik osztály

példányainak. A *GetNamable*, *GetNamables* és *GetNamablesOfType* metódusok segítségével kapja majd meg a GUI felület azokat a példányokat, melyekre épp szüksége van.

A *GLUTBasedEngine Display* metódusának működése már kicsit bonyolultabb. Miután alkalmazta az aktív kamera *Modelview* mátrixát, a *renderingState* mezőt *PRE_RENDERING* értékre állítja, majd meghívja a *PreRender* metódust. Ez egyelőre a *DynamicTextures* listára regisztrált példányok *Render* metódusát hívja meg. Ide kerülnek a valós idejű tükröződést, a vetett árnyékokat és a félig statikus tükröződéseket megvalósító *Texture* példányok, ahogy ezt majd a későbbiekben látni fogjuk. Ezután a *renderingState* mezőt *TEXTURING_PASS_1* értékre állítja, és meghívja a *Render* metódust. Ez a vetett árnyékokhoz tartozó textúrák felhelyezését végzi a testekre. Majd ugyanez végbe megy *TEXTURING_PASS_2* értékre állított *renderingState* mellett is, ekkor kerülnek megjelenítésre a *Standard, Bump, Reflection* és *Refraction* textúra rétegek, melyekre később még visszatérünk. A *Material* példányok alkalmazásukkor, a *renderingState* alapján tudják, hogy milyen kombináló függvényeket alkalmazzanak az egyes pass-ok kirajzolásakor.

A *Medium-* és *LowPriorityRenders* listák a nem átlátszó és átlátszó testek elkülönítésére szolgálnak. Az átlátszó testeknek később kell kirajzolásra kerülniük, hogy ezzel csökkentsék az átlátszóság okozta hibák esélyét. Ha egy átlátszó test hamarabb kerül kirajzolásra, mint egy mögötte levő másik test, akkor a mögötte levő, később kirajzolt test nem fog látszani az átlátszó testen keresztül.

Megjegyzés: Az átlátszóság tökéletesítésére a testek kamerától való távolság alapján történő rendezése, és a listában a legtávolabbitól kezdődően történő kirajzolás szolgál. Ennél viszont figyelembe kell venni a sok test rendezése esetén megnövekvő CPU igényt. E módszer bevezetése, és optimális alkalmazása még a jövő feladata a motor számára. Játékokban, azt a minimálisan észrevehető hibát, amit az átlátszó testek rendezetlen kirajzolása okoz, általában elfogadhatónak tekintik, mert a sebesség sokkal fontosabb.

A *HUDRenders* listában a legutoljára kirajzolandó, a Heads Up Display részét képező elemek szerepelnek. Ezekkel nem foglalkozom bővebben, az alkalmazásban a logo megjelenítését és az aktuális sebesség kiírását végzik.

A *Megvalósító osztályok és technikák* (5) fejezetben látni fogjuk, hogyan illeszkednek a származtatott osztályok a bemutatott hierarchiába.

3. Egységes architektúra

Az egységes architektúra kifejezés, az osztályok, a program által történő egységes kezelését takarja. A célja, hogy a GUI felületek tervezésekor, valamint a jelenetek fájlból történő reprodukálásakor, betöltésekor a fejlesztőnek ne kelljen tudnia, hogy milyen osztály property-eit fogja az adott algoritmus módosítani, vagy épp betöltéskor ne kelljen minden egyes osztály példányainak létrehozására, és property-einek beállítására külön algoritmust készíteni.

Az alapja, a *Namable* osztály, mint ősosztály, amely a példányok egyedi nevekkel történő azonosítását teszi lehetővé. Ez azért is fontos, mert így a felhasználó számára is átláthatóbb és jobban kezelhetőbb lesz a struktúra. A felhasználó alatt nem a legvégső felhasználót értem, aki például egy játék előtt ülve, mit sem tud a mögötte rejlő szerkezetről, hanem például egy játék tervezőt, aki a motort felhasználva, beépíti alá a játék funkcióit megvalósító réteget. Ha a motort egy tervező program fájljait kezelő, és a tervezett jeleneteket valós időben megjelenítő alkalmazás használja, akkor viszont az apróbb módosítások végett, a végfelhasználó is betekintést nyerhet a struktúrába. Ez esetben, az alkalmazás fejlesztői teljes mértékben új GUI-t tervezhetnek a példányok property-einek, mert a motorban levő GUI semmit sem tud az épp általa megjelenített példányról, azaz a GUI nem tesz hozzá semmi plusz funkciót, nem ad semmilyen többletjelentést a motor példányaihoz, példányainak.

3.1. Property

A következőkben egy egyszerű példán keresztül szeretném bemutatni a property-k működését, és jelentőségét. A *Solid* osztály *Material* property-je:

```
[TypeConverter(typeof(NamableTypeConverter)),
Description("The associated material")]
public virtual Material Material { get { return material; } set { material = value; } }
```

A property típusa *Material*, azaz neki értékül egy *Material* osztálybeli példány adható. A *get* és *set* blokkokban kell a property viselkedését megadni, amikor az értékét lekérik, vagy beállítják. Ebben az esetben ezek a blokkok a legegyszerűbb funkciót látják el, egyszerűen a kívülről nem látható *material* mező értékét adják vissza, vagy állítják be.

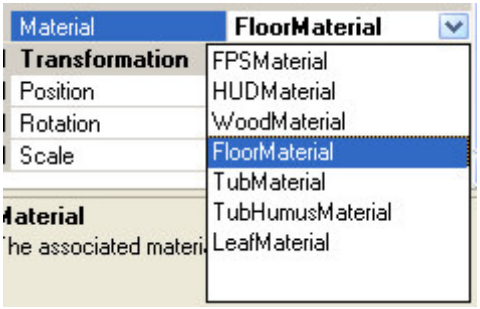
A szögletes zárójelek között a property-hez tartozó attribútumok adhatóak meg. A *Description* a property leírását állítja be, amit a .Net majd a property-nek, az úgynevezett *PropertyGrid* kontrolban való megjelenítésekor használ fel. Ez egy leírás a felhasználó felé, az adott property-ről.

3.2. TypeConverter

A property-k erejének egyik igazi eleme a hozzájuk csatolható *TypeConverter*. Képzeljük el, hogy ez a property megjelenik egy *PropertyGrid*-ben, ahogy ezt később majd a GUI felületeknél látni fogjuk részletesen. Ezután a felhasználó értékül szeretne neki adni egy *Material* példányt, vagy akár csak meg szeretné nézni, hogy melyik *Material* van hozzákapcsolva az adott *Solid*-hoz. *TypeConverter* nélkül ez teljességgel lehetetlen, mivel ekkor a felhasználó csak annyit lát, hogy a property értéke egy furcsa hash érték, ami a .Net számára egyedien azonosítja a *Material* példányt, de élő ember ezzel nem sokat ér. Ugyanígy mikor meg szeretne neki egy másik példányt adni, kizárt, hogy tudja a futás időben generált hash kódját.

Itt lép a képbe a *NamableTypeConverter*, ami egy saját *TypeConverter*. Ez az osztály a beépített *StringConverter*-t terjeszti ki, mivel *String*-é szeretnénk konvertálni a property értékét, ami egy példány.

```
public class NamableTypeConverter : StringConverter
{
    public override bool CanConvertTo(ITypeDescriptorContext context, Type destinationType)
    public override object ConvertTo(ITypeDescriptorContext context,
System.Globalization.CultureInfo culture, object value, Type destinationType)
    public override bool CanConvertFrom(ITypeDescriptorContext context, Type sourceType)
    public override object ConvertFrom(ITypeDescriptorContext context,
System.Globalization.CultureInfo culture, object value)
    public override bool GetStandardValuesSupported(ITypeDescriptorContext context)
    public override StandardValuesCollection GetStandardValues(ITypeDescriptorContext context)
    public override bool GetStandardValuesExclusive(ITypeDescriptorContext context)
}
```



14. ábra: Material property, működés közben

Ez a *TypeConverter* teszi lehetővé, hogy a képen (14. ábra) látható módon, egy listából válasszunk az összes létező *Material* példány közül.

A *CanConvertTo* és *CanConvertFrom* metódusok esetünkben egyszerűen csak *true* értékkel térnek vissza, mivel legördülő listával valósítjuk meg a választást. Ezek ugyanis előzetes integritás ellenőrzésre valók, ha például a felhasználó kézzel írná be a példány nevét. Ahogy az a listában is látszik, a példányok azonosítására *Name* property-jüket használjuk, így a felhasználónak csak a számára ismert, általa adott nevek közül kell választania.

A *ConvertTo* metódus a property értékének kiolvasásakor hívódik meg, tehát ez egyszerűen visszaadja a *Material* példány *Name* property-ének értékét. A *ConvertFrom* metódus szerepe

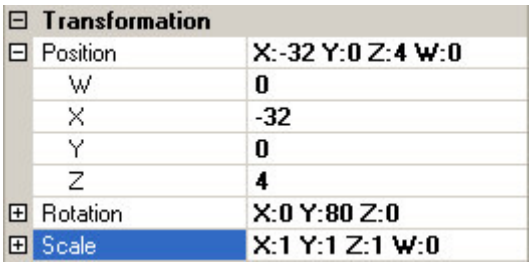
fordított, egy paraméterül kapott *String* alapján, amely egy létező példány neve (erre szolgál a *CanConvertFrom*, vagy esetünkben a *GetStandardValues* metódus, ahogy azt később látni fogjuk) kikeresi a hozzá tartozó példányt, és visszaadja azt. Ez a kikeresés történhetne a .Net *Reflection* namespace által biztosított eszközökkel, melyek képesek az éppen futó, vagy egy tetszőleges assembly-ben kikeresni az összes megadott típusú példányt, és ezután már csak ezek közt kéne név alapján megkeresni a nekünk kellőt. Az optimalitás kedvéért, mivel többek között pont ezért van az *EntityContainer*-ben a *Namables* listában regisztrálva az összes *Namable* példány, felhasználjuk az *EntityContainer.GetNamable* metódust, amely visszaadja a megadott nevű példányt.

Már csak azt kell megoldani, hogy a felhasználó ne kézzel írja be a példány nevét, hanem egy legördülő listában kiválaszthassa azt az összes létező példány közül. Erre szolgál a *GetStandardValues* metódus. Ez a következőképp van implementálva. A paraméterül kapott *context* példányból kinyerhető a property típusa, amihez az adott *TypeCoverter* hozzá van kapcsolva. Ezt a típust az *ITypeDescriptorContext.PropertyDescriptor.PropertyType* tartalmazza. Az *EntityContainer GetNamablesOfType* metódusa segítségével, egy *Dictionary<String, Namable>*, név alapján indexelt listában, visszakapjuk az összes adott típusú, regisztrált példányt. Ezután már csak a lista kulcsaiból fel kell építenünk egy *StandardValuesCollection* példányt, és visszaadni azt.

A *GetStandardValuesSupported* és *GetStandardValuesExclusive* metódusok egyszerűen *true* értéket adnak vissza. Az első a legördülő lista támogatását adja meg a keretrendszer fele, a második pedig azt állítja be, hogy a felhasználó csak a legördülő listában található értékek közül választhat, nem írhat be saját kézzel. Ezzel biztosítjuk a nevek integritását.

Amint az észrevehető, ugyan a *Material* property-n keresztül láttuk a *NamableTypeConverter* működését, magában az implementációban sehol nem szerepel a *Material* explicit módon. A *PropertyDescriptor.PropertyType* lehetővé teszi, hogy ez a *NamableTypeConverter* bármilyen típusú property-re működjön, ha a típus ősei közt szerepel a *Namable* osztály. Ennek köszönhetően ezt a *TypeConverter*-t használjuk mindenhol, ahol példányok közötti áthivatkozásra van szükség.

Érdemes megemlíteni még az *ExpandableTypeConverter* osztályt, amelyből olyan saját *TypeConverter* származtatható, mely lehetővé teszi összetett propertyk szerkesztését. Az ilyen propertyk előtt egy ’+’ jel jelenik meg, amelyet megnyomva legördülnek a propertynek



15. ábra: ExpandableTypeConverter-t használó property-k

értékül adott példány propertyjei, így azokat is szerkeszthetjük, a képen (15. ábra) látható módon.

A *TypeConverter* osztályról, implementálásáról és alkalmazási példákról az MSDN Library – TypeConverter Class [4] fejezetében olvashatunk bővebben.

3.3. UITypeEditor

A property-k felhasználói interakcióját még tovább bonyolítja egy másik nagyon fontos elem, az *UITypeEditor*. Az előző példa fonalán továbbhaladva, nézzük most a *Material* osztály egy másik property-ét:

```
[Description("The first texture level for standard maps"),
Category("2nd Pass"),
DisplayName("1. Standard")]
public TextureLevel Standard { get { return standard; } }
```

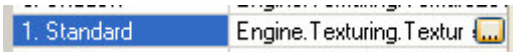
A *Standard* property, amely a sztenderd textúra szintet reprezentálja, a második renderelési pass-ban. A *Category* attribútumról még nem volt szó, ez a megjelenítéskor való kategóriába sorolást adja meg. A *DisplayName* attribútum pedig a property megjelenített nevét állítja be, ha nincs megadva, akkor a property a saját nevével jelenik meg. Itt nem látható attribútumként hozzárendelve semmilyen *UITypeEditor*, ahogy az előzőekben a *TypeConverter* esetében láttuk. Ennek oka az, hogy míg az előző esetben a *Material* osztály felhasználási területe sokkal tágabb volt, itt a *TextureLevel* osztály egy specifikus osztály. A *TextureLevel* nem *Namable* leszármazott, hanem a *Material*-hoz kötődik nagyon szorosan. Minden *Material* példánynak pontosan nyolc *TextureLevel* példánya van (pass-onként négy-négy), és ezek együtt jönnek létre, és együtt kerülnek lebontásra az adott *Material* példánnyal. Ebből kifolyólag a *TextureLevel* osztályhoz, magához van hozzákapcsolva a megfelelő *UITypeEditor*.

```
[Editor(typeof(TextureLevelTypeEditor), typeof(UITypeEditor))]
public class TextureLevel : Applyable { … }
```

Az *UITypeEditor*, amit használunk, egy saját *TextureLevelTypeEditor*, kifejezetten a *TextureLevel* osztályhoz lett létrehozva.

```
public class TextureLevelTypeEditor : UITypeEditor
{
    public override UITypeEditorEditStyle GetEditStyle(ITypeDescriptorContext context)
    public override object EditValue(ITypeDescriptorContext context, IServiceProvider
provider, object value)
}
```

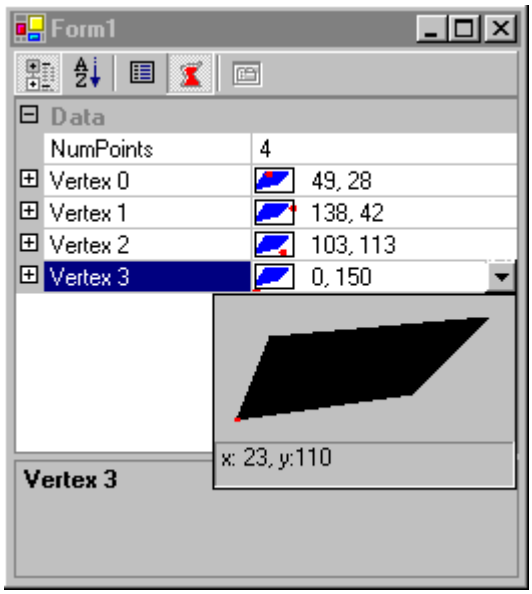
A *GetEditStyle* metódus szerepe, hogy megmondja a keretrendszer számára, milyen stílusú lesz az *UITypeEditor*. Ez lehet *None, Modal* és *DropDown*. A *None* és *DropDown* most számunkra nem lesz érdekes. A *Modal* egy olyan *UITypeEditor*-t definiál, amely felhasználói felülete egy felugró ablak. Ez a képen (16. ábra) látható módon jelenik meg a felhasználó felé, egy gomb, amelynek felirata „…”. Amikor a felhasználó a gombot megnyomja, lefut az *EditValue* metódus. Ez esetünkben nem csinál mást, mint lepéldányosít egy *EntityEditor* formot, az adott *TextureLevel* példánnyal paraméterezve, majd megjeleníti azt. Az *EntityEditor* form szerepéről majd a GUI fejezetben bővebben lesz szó.



16. ábra: A Standard property

Bővebb információ az MSDN Library – UITypeEditor Class [5] című fejezetében olvasható.

Az *UITypeEditor*, ha egyszer egy osztályhoz lett társítva, annak minden leszármazott osztályához is társítva lesz, hacsak az adott osztály nem definiálja felül. Erre való a *None* stílus, amellyel egy már meglévő társítást definiálhatunk felül, mégpedig kikapcsolva az egyedi felhasználói felületet.



17. ábra: Példa legördülő UITypeEditor-ra

A *DropDown* stílus hasonló a *TypeEditor GetStandardValues* funkciójának működéséhez, azzal a különbséggel, hogy itt teljesen saját felhasználói felületet adhatunk meg, amely legördítéskor megjelenik.

A képen látható egy példa, amely az MSDN Library – Getting the Most out of the .Net Framework PropertyGrid Control [6] fejezetében található.

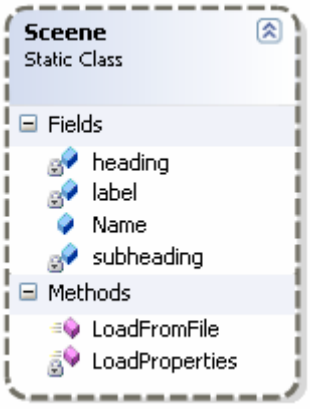
Ez a cikk részletesen tárgyalja a *PropertyGrid* osztályt, és a hozzá tartozó attribútumokat, átfogó képet adva azok szerepéről, működéséről, példákkal szemléltetve.

3.4. Reflection

A .Net *Reflection* namespace nyújtotta lehetőségekről már tettem említést. Ez a namespace olyan eszközöket biztosít, melyekkel futás időben visszakereshető egy adott példány típusa, maga típus mint objektum kezelhető, rengeteg lehetőség adva ezzel a példányok futási időben, kódból való manipulálására. Egy adott assemblyben visszakereshetők megadott típusú példányok, vagy akár a típust reprezentáló objektum kikereshető a típus neve alapján. Létrehozhatóak olyan típus példányai, melynek nevét például egy szöveges fájlból olvastuk be. Ezeket a lehetőségeket kiaknázva működik a *Sceene* osztály betöltő algoritmusa, amely egy script-hez nagy mértékben hasonlító, szöveges fájl segítségével építi fel az adott jelenetben szereplő példányokat, anélkül, hogy bármilyen ismerettel rendelkezne az adott példány osztályának specifikus elemeiről.

Bővebb információ a *Reflection* namespace-ről és a kapcsolódó namespace-ekről az MSDN Library – Reflection [7] fejezetében található.

3.5. Sceene



18. ábra: Sceene osztály

A *Sceene* statikus osztály () felelős a jelenetek fájlból való betöltéséért, és később ennek szerepe lesz a jelenetek elmentése is.

A *LoadFromFile* metódus megnyitja a szöveges (.sceene kiterjesztésű) fájlt, az aktuális elérési útvonalat a fájl elérési útvonalára állítja, hogy ezzel megkönnyítse a fájlban található relatív elérési utak feldolgozását, majd megkezdi egy tördelő osztály segítségével a benne található szöveg feldolgozását. A következőkben részleteket mutatok be a példaprogram jelenetét leíró fájlból, rajtuk keresztül tárgyalva a betöltő algoritmus működését.

3.5.1. Új példány létrehozása

```
Engine.LoD.LoDSolid {
    Name "chairleg"
    Filename "models\chairleg.ase"
    Position "3 8 3"
    Material "WoodMaterial"
    LoDBias 0.5
}
```

Itt a kapcsos zárójelek előtt meg kell adnunk a típus teljes, minősített nevét, amelyből új példányt szeretnénk készíteni. Az algoritmus egy ilyen részhez érve, meghívja az *Activator* osztály *CreateInstance* metódusát, amely szintén a *Reflection* namespace eszköze. Ezzel létrejön az új példány, majd a kapcsos zárójelek közti rész kerül feldolgozásra. A *LoadProperties* metódus végzi a feldolgozást, ha talál egy nevet, az adott példány *PropertyDescriptor*-a segítségével megkeresi az adott nevű property-t, majd szintén a *PropertyDescriptor* segítségével megkeresi az adott property-hez rendelt *TypeConverter*-t, és meghívja a *ConvertFrom* metódusát a név után talált értékkel paraméterezve.

Ha a feldolgozás során valamilyen hiba történik, arról ablakban értesíti a felhasználót, megjelenítve a hiba pontos helyét a fájlban, és a hiba okát.

Látható a *Filename* property értékének beállítása. Amikor ez a property új értéket kap, a *Solid* példány automatikusan betölti a megadott fájlt, ahogy erről már korábban szó esett.

3.5.2. Meglévő példány klónozása

```
Engine.LoD.LoDSolid : "chairleg" {
    Name "chairleg.clone1"
    Position "-3 8 3"
    Rotation "0 30"
}
```

A típus után „:”-al megadva egy már létező példány nevét, az algoritmus a név alapján kikeresi a példányt, és meghívja annak *Clone* metódusát, majd a létrejött klónra meghívja a *LoadProperties* metódust.

3.5.3. Meglévő példány beállításai

```
EnvironmentSettings {
    FovY 65
}
```

A már létező példány nevét megadva, kapcsos zárójelek közt felsoroljuk a property-érték párosokat, ez esetben egyet. A betöltő kikeresi a név alapján a példányt, és meghívja rá a *LoadProperties* metódust. A már előzőekben ismertetett *EnvironmentSettings* osztály egyetlen, *EnvironmentSettings* nevű példányának *FovY* property-ét állítjuk 65-re.

3.5.4. Komplex property-k

```
Engine.Material {
    Name "FloorMaterial"
    Diffuse "1 1 1 1"
```

```
    Ambient "-0.5 -0.5 -0.5 1"
    Standard {
        Texture "FloorTexture"
        Scaling "8 8"
    }
    Bump {
        Texture "FloorBump"
        Scaling "32 32"
        LoDBias -0.75
        TexEnvMode "GL_COMBINE"
        CombineRGB "GL_ADD_SIGNED"
    }
    Reflection {
        Texture "ReflectionTexture"
        Position "0.5 0.5"
        TexEnvMode "GL_COMBINE"
        CombineRGB "GL_INTERPOLATE"
        CombineAlpha "GL_INTERPOLATE"
        ConstantColor "0 0 0 0.15"
    }
}
```

Ez a kódrészlet a *FloorMaterial* nevű példány létrehozását, és beállítását végzi, megfigyelhető a *Standard, Bump* és *Reflection* property-k mássága. Ezek ugyanis *TextureLevel* típusúak, ahogy azt a *Standard* esetében már korábban láthattuk. Tehát összetett property-k, a nevük után kapcsos zárójel jelzi a betöltő számára, hogy ilyen property következik, ekkor kikeresi a név alapján a megfelelő property példányt, majd erre rekurzívan meghívja a *LoadProperties* metódust. Ezzel a módszerrel az összetett property-k tetszőleges mélységben egymásba ágyazhatók.

3.5.5. Elmentés

Az elmentő algoritmus implementálása még a jövő feladata, de a lényegi működése nem sokban tér el a betöltőétől. Ami miatt nem jutott rá idő, az hogy az optimális fájl készítéséhez az algoritmusnak tudnia kell, hogy egy property értéke alapértelmezetten lett-e hagyva, vagy megváltozott, és csak akkor kell fájlba írnia, ha megváltozott. Ez azért fontos, mert rengeteg property van, amelyeknek most csak egy kis szeletét volt hely bemutatni, de ha minden property kikerül a fájlba, hatalmas méretű *sceene* fájlok jönnek létre. Erre a property-hez rendelhető *Default* attribútum szolgál, amelyben megadható a property alapértelmezett értéke, így ha az elmentő algoritmus olyan property-t talál, amelynek értéke nem változott az alapértelmezetthez képest, azt egyszerűen figyelmen kívül hagyja. Az összes property-hez utólag megadni az alapértelmezett értéket, viszont nem kissé időigényes munka. Voltak ettől sokkal fontosabb implementálandó dolgok, amikkel foglalkoznom kellett.

A másik, a klónok nyilvántartása, amely az elmentésnél kap fontos szerepet. Ennek a problémának a megoldására született a *Cloneable* interfész *CloneOf* property-e, amely megmondja az elmentő számára, hogy klónként kell-e elmentenie az adott példányt vagy sem.

A *sceene* fájlokat így egyelőre kézzel kell megírni.

3.6. Konklúzió

Láthattuk, hogy ugyan a property-k működésének megértése, és az attribútumaik implementálása kezdetben elég sok időt és energiát elvisz, mindenképp érdemes velük vesződni, mert használatukkal a betöltő és elmentő algoritmusok lényegesen egyszerűsödnek, nem is beszélve a GUI felületek tervezésének egyszerűségéről, ahogy azt a következő fejezetben látni fogjuk.

4. GUI

Egy ilyen motor GUI felülete gyakorlatilag csak a fejlesztői réteget célozza meg. A szerepe, hogy futási időben hozzáférést biztosítson az egyes példányok beállításaihoz, megkönnyítve ezzel a fejlesztést. A GUI koncepciója felhasználható a későbbi, a motorra épülő alkalmazás tervezésekor is, bár ott valószínűleg egy teljesen saját GUI felület szükséges, amely nem tesz hozzáférhetővé ekkora mennyiségű információt a példányokról.

Tehát a következőkben bemutatott GUI a demo tervezést, a fejlesztést, a bevezetett új technikák tesztelését, egyszerű paraméterezését teszi lehetővé. A cél, hogy ezt minél egyszerűbben, minél kevesebb specifikus részt tartalmazva tegye. Ahogy azt látni fogjuk, a megvalósító formok gyakorlatilag semmilyen explicit információval nem rendelkeznek azokról a példányokról, amelyek paramétereinek beállítását lehetővé teszik majd.

4.1. EntityLister

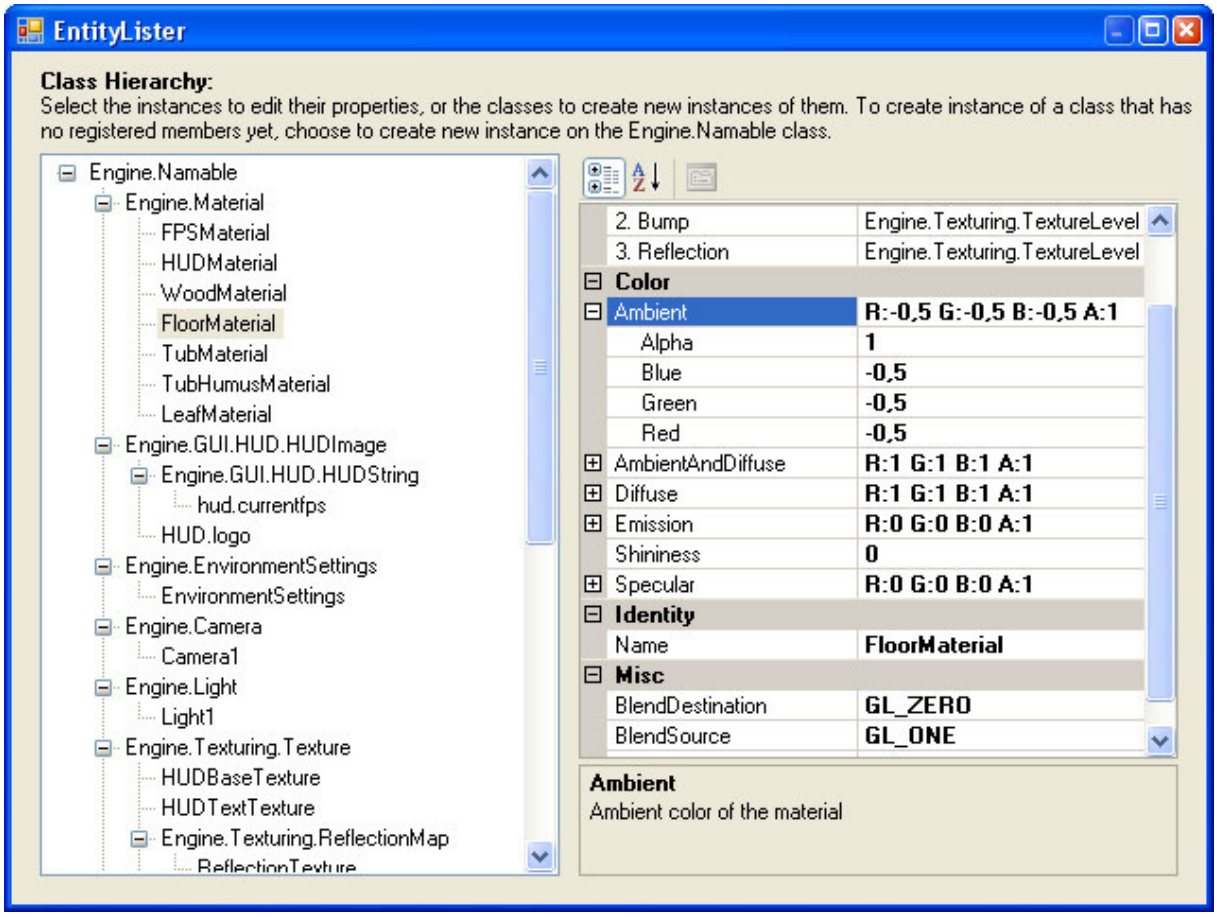
Az *EntityLister* form (19. ábra), a legelső felhasználói felület, amellyel találkozunk (F1 lenyomásával) a példaprogramban. Feladata, lehetővé tenni az osztályok példányainak menedzselését.

Az *EntityContainer getNamables* metódusának segítségével lekéri az összes regisztrált példányt. Ezután, ahogy az az ábrán látható, a *Reflection* eszközeinek segítségével bejárja minden példány ősosztályait, egészen a *Namable*-ig, és ezek alapján fába rendezi a példányokat.

Minden példány alapvetően rendelkezik egy *GetType* metódussal, amely visszaadja az adott példány osztályához tartozó *Type* osztálybeli példányt. E példány *BaseType* nevű property-je hivatkozza az ősosztály *Type* osztálybeli példányát. Így rekurzívan bejárható egy tetszőleges példány minden ősosztálya.

Bővebb információ erről az MSDN Library – Type Class [8] fejezetében található.

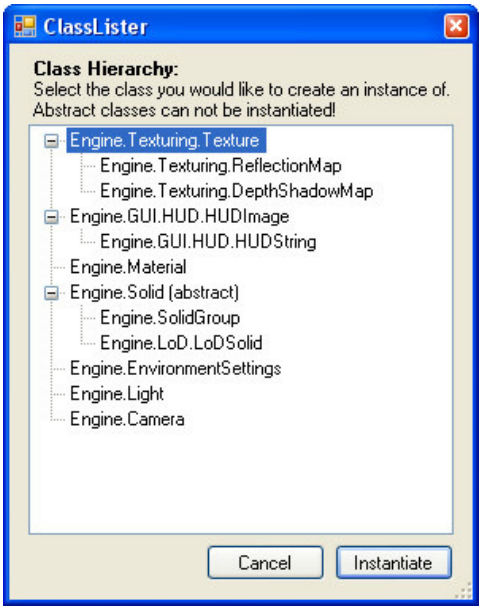
A form lehetőséget ad, példányok létrehozására, törlésére és klónozására egy felugró menü segítségével.



19. ábra: Az EntityLister form

A jobb oldalon, a részleteiben már bemutatott *PropertyGrid* grafikus vezérlő látható. Ennek az eszköznek paraméterül adva egy tetszőleges osztály példányát, megjeleníti annak property-eit, használva mindazon eszközöket, melyeket az előző fejezetben bemutattam.

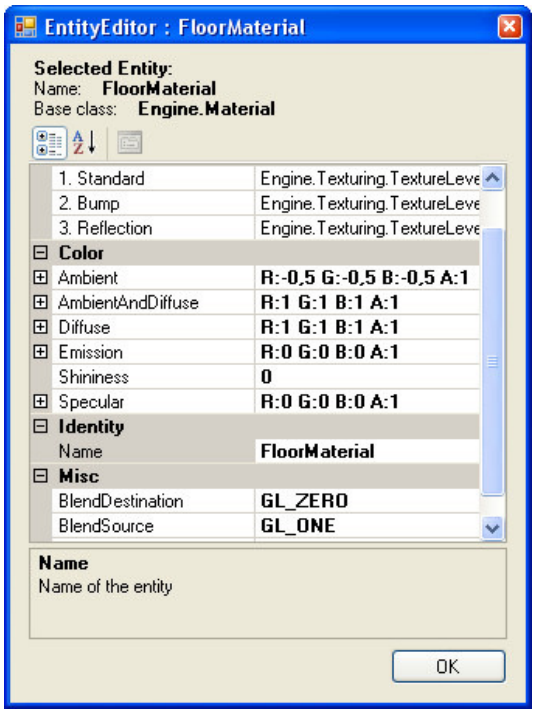
4.2. ClassLister



20. ábra: ClassLister form

A *ClassLister* form (20. ábra) segítségével bármely, a *Namable*-ből származó osztály példányosítható. Érdekessége, hogy egy, az eddigieknél is magasabb szintű *Reflection* eszközt használ. Itt ugyanis semmilyen konkrét példány nincs, amely alapul szolgálhatna a keresésnek.

A *typeof* kulcsszó segítségével lekéri a *Namable* nevű osztály típusát, mint egy *Type* osztálybeli példányt. E példány *Module* nevű property-je hivatkozza azt az assembly modult, amelyben a *Namable* osztály található. E *Module* példány *FindTypes* metódusának segítségével visszakeresi az összes olyan osztály *Type osztálybeli* példányát, amely ősei közt szerepel a *Namable*. Ehhez szüksége van még egy általunk megírt *TypeFilter* delegate típusú függvényre, amely a szűrést végzi. Majd a visszakapott *Type* osztálybeli példányokat az előző formhoz hasonlóan, osztályhierarchia szerint, fába rendezi.



21. ábra: EntityEditor form

Bővebb információ erről az MSDN Library – Module Class [9] fejezetében található.

4.3. EntityEditor

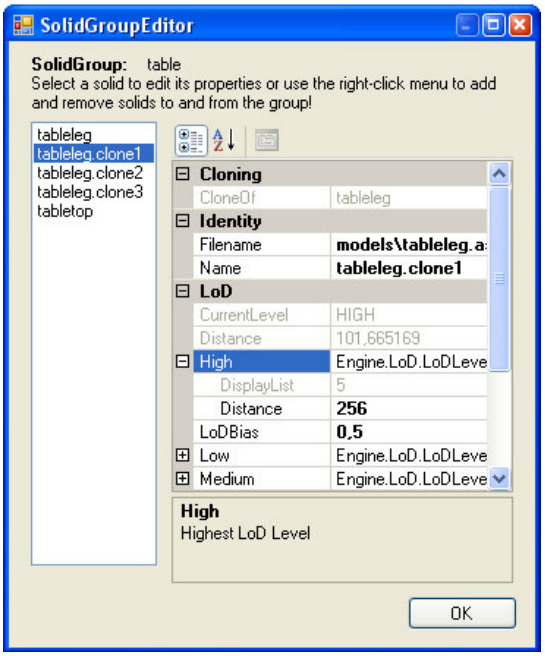
Az *EntityEditor* form (21. ábra) szolgál az egyes példányok önálló megjelenítésére. Szerepe csak az átláthatóság növelésében van.

Semmi újat nem hoz be, csak a teljesség kedvéért került megemlítésre. A már tárgyalt *PropertyGrid* kontrol segítségével megjeleníti a paraméterül kapott példány property-eit.

Illetőleg nagy jelentőséget kap a *Material* osztály *TextureLevel* típusú property-einek

szerkesztésekor. Ez az ablak ugrik fel, amikor a felhasználó megnyomja a '…' feliratú gombot, ahogy azt az *UITypeEditor* osztály tárgyalásakor bemutatott példában, már az előző fejezetben láthattuk.

4.4. SolidGroupEditor



22. ábra: SolidGroupEditor form

Ez a form valamivel specifikusabb, mint a többi. Szerepe a *SolidGroup* osztálybeli példányok elemeihez biztosítani hozzáférést. Ezt ugyanolyan eszközök segítségével éri el, mint a korábban bemutatott formok.

A megszorítás annyi, hogy csak *SolidGroup* osztálybeli példánnyal paraméterezhető, mert tartalmaz *SolidGroup* specifikus részeket.

5. Megvalósító osztályok és technikák

Ebben a fejezetben részletesen foglalkozunk azokkal az osztályokkal, amelyek egy magasabb szintet képviselnek, implementálják a ténylegesen megjeleníthető dolgokat.

5.1. A tárgyak megjelenítésének alapja, a LoD

A LoD (Level of Detail) szerepe jelentős a megjelenített háromszögek mennyiségének optimalizálásában. A technika lényege, hogy azoknak a tárgyaknak, amelyek messze vannak a kamerától, drasztikusan csökkentsük a megjelenítési idejét, azaz a megjelenített háromszögeik számát úgy, hogy ez minél kevésbé legyen észrevehető a néző szemszögéből. A LoD algoritmustól elvárjuk, hogy a részletesség csökkentése közötti átmenetek ne legyenek nagyon szembetűnők, és hogy a lerontott részletességű tárgyak alakja, és főbb ismertető jelei ne romoljanak nagymértékben.

5.1.1. LoD fajták

A LoD technikák alapvetően két csoportba sorolhatók. Az első a dinamikus LoD. Ennek lényege, hogy a LoD algoritmus egy előre definiált és kiszámolt, többletinformációval rendelkező, geometriai leírás alapján, valós időben változtatja a tárgyak részletességét. Az ilyen technikák teljes mértékben kiküszöbölik az átmeneteket az egyes részletességi szintek között. Hátrányuk viszont a magas CPU igény, amely erős optimalizálást követel meg, valamint pontosan definiálni kell a tárgyak éleit, a különös ismertető jeleit, jobb esetben még prioritással is ellátva ahhoz, hogy a részletesség csökkenésével ne deformálódjanak el észrevehetően.

Egy nagyon jó dinamikus LoD algoritmus leírása található Michael Garland, Paul S. Heckbert – Surface Simplification Using Quadric Error Metrics [10] cikkében. Ez az algoritmus páronként összeválogatja a tárgy vertexeit, majd minden párhoz négyzetes hibabecslési mátrixokat készít. Figyelembe veszi, hogy a párokat összeköti-e él, azaz érvényesek-e, és hogy a párok eltávolítása mennyire rontaná a tárgy alakjának jellegzetes vonásait. Majd a párokat ez alapján sorba rendezi és futási időben, minden egyes iterációval, eltávolítja a legkisebb hibájú párt. Az eltávolítás lényegében a két vertex, egy harmadikká való összeolvasztását jelenti. A vertexek él-listáját összefűzi, eltávolítja az érvénytelenné vált éleket, a négyzetes hibabecslési mátrixaikat egyszerűen összeszorozza, új párokat hoz létre a szomszédos vertexekkel, és elhelyezi őket az eltávolítási listában.

A mi esetünkben ez a technika viszont nem megfelelő, mert az ilyen tárgyak kezelése csak vertex array segítségével valósítható meg optimálisan. Ahogy a következő fejezetben látni fogjuk, a motornál minden display listbe lett helyezve, mert a display listek messze a legoptimálisabb eszközök az OpenGL-ben. Statikus mivoltuk miatt viszont, nem képesek ilyen dinamikus testek tárolására, és megjelenítésére.

A vertex array működéséről bővebb információ az OpenGL Red Book [1] State Management and Drawing Geometric Objects fejezetében található. A display listekről pedig ugyanezen könyv Dislpay Lists fejezetében olvashatunk.

A fentebb bemutatott dinamikus LoD technika tehát, nem lett implementálva a motorban. Későbbi célkitűzésként szerepel az implementálása, de csak segédalgoritmusként, amely automatikusan legenerálja egy nagyfelbontású tárgy megadott LoD szintjeit. Ezeket a LoD szinteket aztán a statikus LoD technika segítségével használjuk tovább.

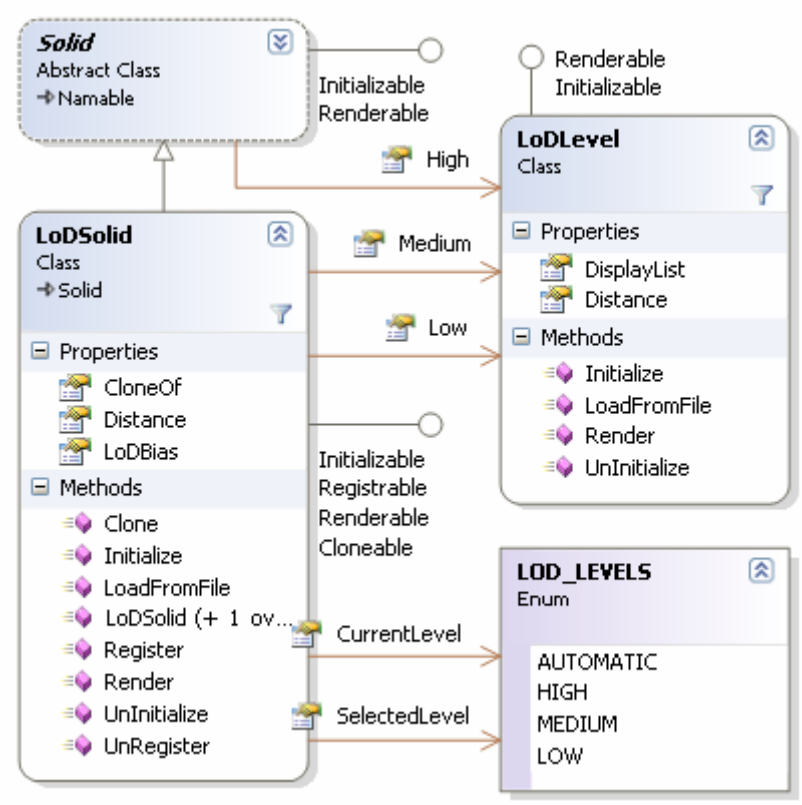
A statikus LoD technika lényege, hogy a LoD szintek előre el vannak készítve, így futási időben nincs számolási igény, csak egyszerűen, a távolság függvényében választani kell egy megfelelő LoD szintet, és kirajzolni. A hátránya, hogy rosszul választott állandók esetén, észrevehető az egyes szintek közötti váltás, valamint nagyobb memória igénye van, mivel minden LoD szintet külön tárolni kell. Előnye viszont, hogy minden LoD szint előre belefordítható egy-egy display listbe, így a kirajzolási sebessége maximális.

A motor ez utóbbi technikát valósítja meg. Három LoD szintet használ, melyek külső tervező program segítségével készültek, és egy fájlba vannak elmentve, innen kerülnek betöltésre. A tesztek alapján, jól megválasztott, tárgyanként definiálható LoD bias értékek használatával, a részletességi szintek közötti átmenet észrevehetősége minimalizálható.

A megvalósító osztály a *LoDSolid*, amely a *Solid* osztály leszármazottja, és teljes mértékben implementál egy három LoD szinttel rendelkező tárgyat.

A LoD példában (Függelék, 26. ábra) jól megfigyelhető, hogy a három szint kiosztása nem lineáris. Ha a *High* szintet vesszük 100%-os részletességűnek, akkor a *Medium* 60-70%os, a *Low* pedig 20-30%-os részletességű.

5.1.2. LoDSolid



23. ábra: LoDSolid osztály

Ahogy azt a *Solid* osztály tárgyalásánál már említettem, a *High*, *LoDLevel* típusú property már felsőbb szinten definiálásra került. Ez reprezentálja a legnagyobb részletességi szintet, és azért van *Solid* szinten, mert a későbbiek során, ha esetleg LoD technikát nem használó osztály kerül felvételre, annak is lesz egy képletesen értett LoD szintje, amit tárolni kell.

Ehhez bevezetjük még a *Medium* és *Low* LoD szinteket, így áll elő a három LoD szint. A *LoDSolid* osztály (23. ábra) tehát egy háromszintű, statikus LoD technikát megvalósító osztály.

A *LoDLevel*-ek property-jei *ExpandableTypeConverter* használatával szerkeszthetők. Minden *LoDLevel*-nek van egy *DisplayList* property-je, amelybe az általa reprezentált LoD szint bele van fordítva. A *Distance* property-jük pedig azt a távolságot állítja be, amely távolságig aktívak lesznek. A végső LoD szint kiválasztásakor, ezek a távolságok felszorzódnak a *LoDSolid LoDBias* property-ének értékével, és ez alapján választódik ki a megjelenített részletességi szint. Ezt a kiválasztást a *LoDSolid Render* metódusa végzi, a kiválasztott szint pedig saját magát jeleníti meg a *Render* metódusa által. A *LoDSolid Distance* property-je csak olvasható, megadja a test aktuális távolságát.

A *SelectedLevel* property segítségével, kézzel beállítható az aktuális LoD szint, vagy automatikussá tehető a szintkiválasztás. Automatikus szintkiválasztáskor a *CurrentLevel*, csak olvasható property adja meg az épp kiválasztott LoD szintet.

A *LoadFromFile* metódus segítségével a *LoDSolid* betölthető egy ASCII Export v2.0 formátumú, szöveges fájlból. Betöltéskor az egyes *LoDLevel* példányok *LoadFromFile* metódusai hívódnak meg, amelyek kikeresik az adott LoD szint adatait a fájlban, és felépítik magukat a talált információ alapján.

Az egyes LoD szintek a geometriát saját, belső reprezentáció szerint eltárolják, és az *Initialize* metódusukban display listbe fordítják azt. Ezután sem kerül törlésre a belső reprezentáció, mert minden egyes teljes képernyős és ablakos mód közötti váltáskor az OpenGL környezetet újra kell inicializálni, azaz ekkor a display listeknek is újra kell fordítódniuk.

5.2. Textúrázás

A textúrázás az a folyamat, amely során a tárgyak felszínére valamilyen bittérképet feszítünk. Ez történhet explicit módon, úgynevezett textúra koordináták megadásával, vagy az OpenGL által biztosított textúra koordináta leképezések használatával.

A textúráknak nagyon nagy szerepük van a tárgyak élethű megjelenítésében, valamint az optimalizálás során is, mert nem szükséges minden kis apró részletet kidolgozni egy tárgyon, elég azok nagy részét, csak textúraként rátenni.

Az OpenGL első implementációiban csak egy rétegű textúrázás volt lehetséges, majd később kiterjesztésként, megjelentek a többrétegű textúrázást megvalósító eszközök, ahogy a grafikus kártyák hardver szinten is kezdték támogatni ezt. A mai grafikus kártyák átlag négy textúra réteg egyidejű megjelenítésére képesek, de a különböző kombináló függvények segítségével tetszőlegesen sok, négy rétegű egység illeszthető egymásra. Ezek persze a teljesítmény rovására mennek, mert minden négy réteg kirajzolásához a teljes képet újra kell rajzolni.

A motor egyelőre négy réteg textúra alkalmazását teszi lehetővé, de a későbbiekben ennek bővülnie kell nyolcra, a több fényforrásból érkező vetett árnyékok, és a fénytörés bevezetésével.

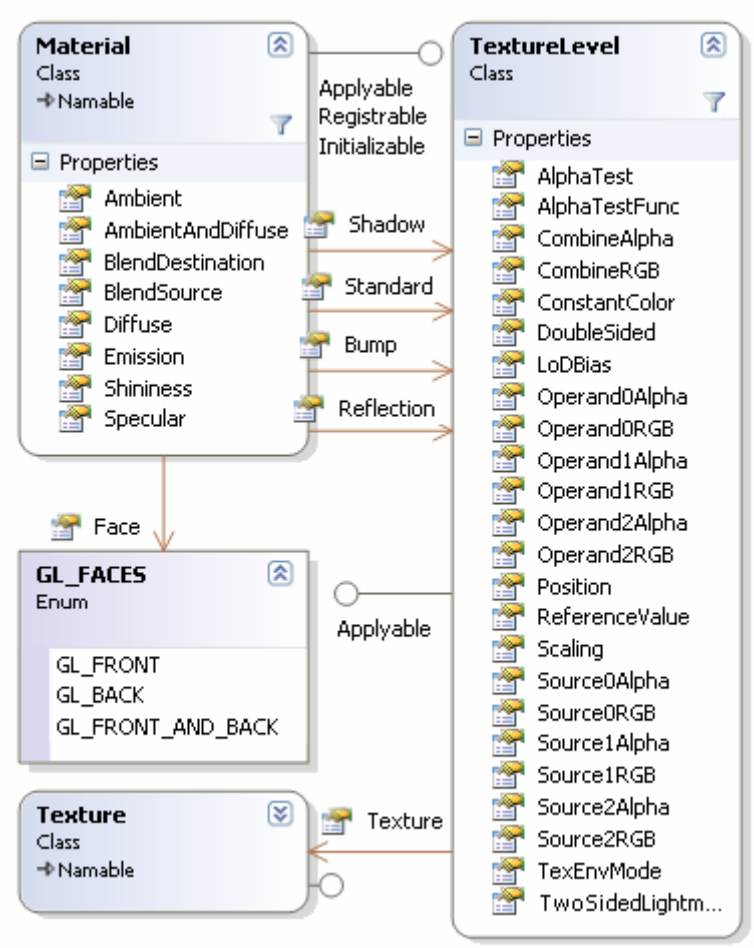
A textúrázáshoz szorosan hozzá tartozik az úgynevezett *material*. Ez definiálja egy adott tárgy anyagát. A *Material* osztály tárgyalásánál részletesen foglalkozunk majd vele.

Az OpenGL pipeline rendszerű feldolgozása miatt az anyag tulajdonságait, és a használt textúrákat a tárgy megjelenítése előtt meg kell adni. Az egyes textúra rétegeket be vagy ki kell kapcsolni annak megfelelően, hogy van-e rájuk szükség vagy nincs. A bekapcsolt textúra

rétegekhez hozzá kell rendelni a megfelelő textúra objektumokat, amelyek tárolják a textúrát, és az egyéb hozzátartozó beállításokat.

Bővebben a materialokról az OpenGL Red Book [1] Lighting, a textúrázásról és a textúra objektumokról pedig a Texture Mapping fejezetében olvashatunk.

5.2.1. Material



24. ábra: Material osztály

A *Material* osztály (24. ábra) felelős egy adott anyagú felület tulajdonságainak beállításáért. Minden *Solid* példányhoz, ahogy azt már láthattuk, egy *Material* példány lehet hozzárendelve. Ez a *Material* példány végzi el a tárgy anyagával kapcsolatos beállításokat, sőt az egyes textúra rétegek alkalmazását is. A négy textúra réteget a *Shadow, Standard, Bump* és *Refleciton* propertyk tartalmazzák. A textúra rétegek *TextureLevel* osztálybeli példányok, property-eik beállítását *UITypeEditor* segítségével végezhetjük el, ahogy ezt egy korábbi példában már láthattuk is.

A *TextureLevel* példányok a *Material* példánnyal együtt jönnek létre, és bontódnak le. Minden *TextureLevel* példányhoz egy *Texture* osztálybeli példány lehet hozzárendelve. Ha nincs hozzárendelt *Texture* példány, akkor az adott *TextureLevel* nem kerül alkalmazásra.

A *Material* alkalmazása a *Solid* renderelése előtt történik. A *Material* alkalmazza az ábrán látható property-jei által reprezentált beállításokat OpenGL függvények segítségével. A property-k nevei megegyeznek az OpenGL specifikációban foglaltakkal, így azokkal nem

kívánok foglalkozni. Bővebb információ az OpenGL Red Book [1] Lighting fejezetében található.

Ezután a *Material* meghívja az egyes *TextureLevel* példányai *Apply* metódusait. Ha érvényes a textúra szint, azaz van hozzárendelt *Texture* példány, akkor alkalmazásra kerülnek a *TextureLevel* property-jei által reprezentált beállítások OpenGL függvények segítségével. A property-k nevei megegyeznek az OpenGL specifikációban foglaltakkal, így azokkal nem kívánok foglalkozni. Bővebb információ az OpenGL Red Book [1] Texture Mapping fejezetében található.

Végül meghívja a hozzárendelt *Texture* példány *Apply* metódusát, amely elvégzi a textúra objektum alkalmazását az adott rétegen.

5.2.2. Események

Az optimalitás miatt mindez a folyamat egy display list segítségével történik, amelyet a *Material* példányok tartanak nyilván, és fordítanak le. Ennek következményeként, ha futás közben valamit megváltoztatunk egy *Texture* példányban, vagy akár egy *TextureLevel* példányban, a változások nem lesznek láthatóak, mert a *Material* display listjét még újra kell fordítani. Ebben nehézséget okoz, hogy a *Texture* és *TextureLevel* példányok nem tudnak semmit arról a *Material* példányról, amelyikhez hozzá vannak rendelve. Egy *Texture* egyszerre akárhány *Material* példányhoz is hozzá lehet rendelve, ezért azok nyilvántartása a *Texture* osztályban, nem lehetséges.

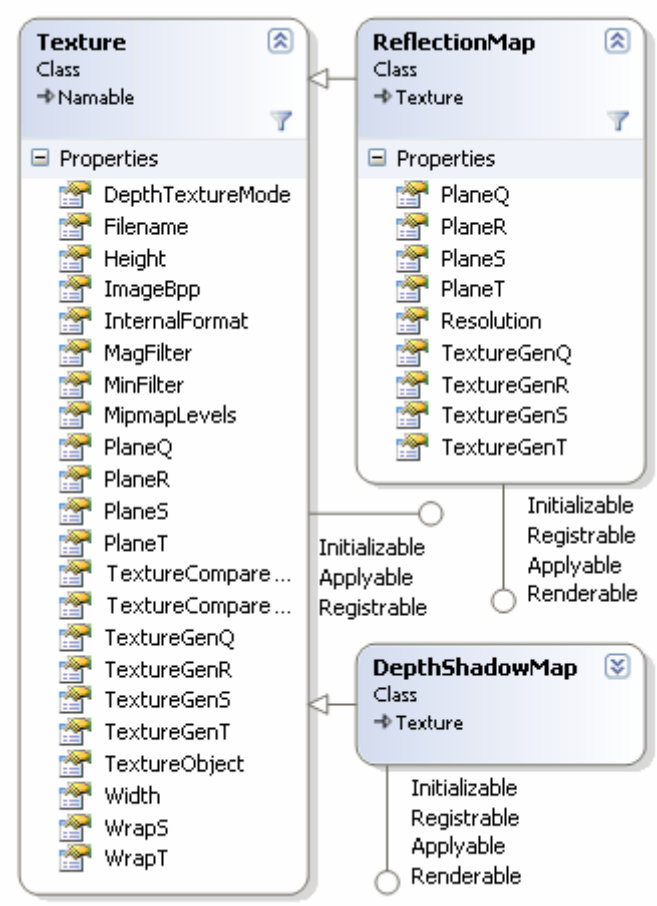
Itt lépnek képbe az események. Az események a Windows által támogatott elemei az alkalmazások közötti, és az alkalmazáson belüli kommunikációnak. Alapjai a teljes Windows felhasználói felületnek. Most mi is hasznukat vesszük, a következő módon. Minden *Texture* példánynak van egy *OnChange* nevű eseménye. Amikor egy *TextureLevel* példányhoz hozzárendelünk egy *Texture* példányt, a *TextureLevel* példány feliratkozik a *Texture* *OnChange* eseményére. Ha előzőleg hozzá volt rendelve egy másik *Texture* példány, akkor annak leiratkozik az *OnChange* eseményéről.

Ugyanilyen módon a *Material* példány fel van iratkozva a hozzárendelt *TextureLevel* példányok *OnChange* eseményeire.

Amikor egy *Texture* példány valamely tulajdonságát megváltoztatjuk, kiváltódik az *OnChange* eseménye. Ezt elkapják a megfelelő *TextureLevel* példányok eseménykezelői, és

kiváltják a saját *OnChange* eseményüket, amit pedig a megfelelő *Material* példányok kapnak el. A *Material* példányok ekkor újrafordítják a display listjeiket, és ezzel a változások láthatóvá válnak, úgymond érvényesek lesznek.

5.2.3. Texture és leszármazott osztályai



25. ábra: Texture, ReflectionMap és DepthShadowMap osztályok

A *Texture* osztály (25. ábra) lényegében az OpenGL és a grafikus kártyák által hardver szinten kezelt textúra objektum megfelelője. A property-jei olyan OpenGL beállításokat reprezentálnak, amelyeket az OpenGL textúra objektumokhoz képes kapcsolni, és olyan formátumban tárolni, amely a grafikus kártya számára a leggyorsabban feldolgozható.

A textúra objektum létrehozása után, csak hozzá kell rendelnünk egy textúra szinthez, majd meghívni azokat az OpenGL függvényeket, amelyek elvégzik a megfelelő beállításokat. Ezek a beállítások a továbbiakban a textúra objektum, újbóli textúra szinthez történő hozzárendelésével alkalmazhatók.

Megjegyzés: nem összekeverendő a textúra objektum és a textúra szint a *Texture* és *TextureLevel* osztályokkal vagy azok példányaival. Ezek az OpenGL által támogatott belső textúra objektumok és azok a textúra szintek, amelyekhez az ilyen objektumok hozzárendelhetők.

A textúra objektumok működése és optimalizációs szerepe legjobban a display listek működéséhez és ilyesfajta szerepéhez hasonlít.

A *Texture* osztály *Apply* metódusának hívásával alkalmazható az általa reprezentált textúra objektum. Ez a metódus semmi mást nem tesz, csak hozzárendeli a textúra objektumot az aktuális textúra szinthez.

A *Texture* osztály property-jei által reprezentált beállítások OpenGL függvények segítségével az *Initialize* metódusban kerülnek alkalmazásra. A property-k nevei megegyeznek az OpenGL specifikációban foglaltakkal, így ezekkel bővebben nem foglalkozom. Információ erről az OpenGL Red Book [1] Texture Mapping fejezetében található.

A *LoadFromFile* metódus segítségével tölthető be textúra bitkép, tetszőleges grafikus fájlformátumból. Ezt a betöltést a DevIl függvénykönyvtár által biztosított eszközök teszik lehetővé. A *MipmapLevels* property segítségével beállítható, hogy hány mipmap szintje legyen az adott textúrának. A mipmap szintek elkészítése autómatikusan történik, szintén a DevIl függvénykönyvtár által biztosított képátméretező eszközök segítségével.

A DevIl függvénykönyvtár által biztosított lehetőségekről Denton Woods – Developer's Image Library Manual [12] című könyvében olvashatunk bővebben.

A mipmap technika lényege, hogy minden textúrához megadunk bizonyos számú, csökkentett felbontású bitképet. Például egy 256×256 pixel méretű textúrához, létrehozunk 128×128, 64×64 és 32×32 méretű mipmap szinteket. Amikor a textúra kirajzolásra kerül, a távolság alapján kiválasztódik egy mipmap szint. E technika segítségével elkerülhetők az úgynevezett *flickering* (villódzás, csíkozódás) effektusok, ha sűrű mintázatú textúrát jelenítünk meg a kamerától távol. A tapasztalataim alapján majdnem minden textúrához elég kettő, a nagyon sűrű mintázatúakhoz három, maximum négy mipmap szint.

A mipmap technikáról, és az OpenGL által biztosított mipmap eszközökről bővebb információ az OpenGL Red Book [1] Texture Mapping fejezetében található.

A *ReflectionMap* és *DepthShadowMap* (25. ábra) leszármazott osztályok valós időben renderelt, tükröződés és Depth Shadow Map alapú vetett árnyék megvalósítását végzik. Melléjük kell, hogy kerüljön még a *MultiLevelShadowMap* és *CubeMap* osztály. Az általuk reprezentált technikákról a következőkben kívánok néhány szót ejteni.

5.3. Implementált és implementálásra váró technikák

5.3.1. Detail Texture, Bump Map

Az úgynevezett Detail Texture technika a tárgyak kidolgozottságát növeli azáltal, hogy a sztenderd textúra rétegre egy másik textúra réteg kerül. Ez a textúra egy olyan képből áll, amely az anyagok természetes felületi hibáit adja meg. A kép általában szürke árnyalatos. Az 50%-os szürke szín a homogén felületet, az ezen felüli színek a kitüremkedéseket, az ez alatti színek pedig a mélyedéseket reprezentálják. Az ilyen, részletességet növelő textúrák sokkal sűrűbben tapétázhatók a felületre, mint a sztenderd réteg, így nincs szükség arra, hogy nagy felbontásúak legyenek, mégis jelentősen növelik a tárgyak élethűségét. Az első, ide kapcsolódó példán (Függelék, 27. ábra) látható, hogy milyen a tárgyak felülete detail texture alkalmazásával, és nélküle. A második példán (Függelék, 28. ábra) pedig egy asztal tölgyfa lapja látható, amelyen a karmolások imitálásra használtam ezt a technikát.

A technika hátránya, hogy nem dinamikus, azaz a fényforrás helyzetének változásával, nem változik például, az így előállított barázdák megvilágítása. Az esetek nagy többségében ez nem jelent gondot, mert megfelelően elkészített képekkel ez a laikus szemek számára szinte észrevétlenné tehető, mivel kevesen nézik azt, hogy mi is történik a karcolások, barázdák, göcsörtök árnyalásával. A modern grafikus kártyák számolási képességei lehetővé teszik egy fejlettebb, az úgynevezett Bump Map, technika használatát. Ez a technika egy olyan képet használ, amely színkomponenseiben normálvektorok koordinátáit tartalmazza, és rendereléskor ezek alapján számítja ki az egyes pontok színét. A motorban az utóbbi technika nem került implementálásra, de shader-ek bevezetésével megvalósítható a jövőben.

A shader-ekkel, és többek között a Bump Map technikával részletesen az OpenGL Orange Book [13] foglalkozik.

5.3.2. Alpha blending és alpha vágás

Az úgynevezett alpha blending technika segítségével, egy átlátszósági bittérkép alkalmazásával, olyan felületeket készíthetünk, melyek részben átlátszóak, részben nem. Ehhez a textúrákat RGBA módban kell tárolnunk, ahol az A csatorna tartalmazza az adott pont átlátszósági értékét. Az így elkészített felületeknél viszont, a korábban már említett probléma merül fel. Mivel az egyes pontjaik mélységi értékére nem hat az alpha komponensük, így az olyan tárgyak, amik később kerültek kirajzolásra nem fognak látszani, a

már hamarabb kirajzolt, átlátszó tárgyakon keresztül. Ennek a problémának a kiküszöbölésére egy, az optimalitást meglehetősen veszélyeztető technika a mélységi rendezés. Az esetek többségében erre nincs szükség, ha használjuk az alpha vágás technikát, amely egy bizonyos alpha érték alatt az adott pontot eldobja, így az nem kerül ki a képernyő, és a mélységi pufferbe sem, vagyis nem fogja kitakarni a később érkező, mögötte levő testeket. Az ide kapcsolódó példán (Függelék, 29. ábra) egy pálmalevél látható, amely ilyen módon kerül kirajzolásra.

Az alpha vágásról, és a kapcsolódó módszerekről bővebb információ az OpenGL Red Book [1] The Framebuffer című fejezetében található.

5.3.3. Valós idejű tükröződés

A valós idejű tükröződés lényege, hogy egy képkocka alatt a képet kétszer kirajzolva, tökéletes, dinamikus tükörkép megjelenítését tegye lehetővé nagy felületű tárgyakon, mint például padlón, vagy vízfelszínen. A folyamat során a kamerát tükrözzük annak a felületnek a síkjára, amelyen a tükörkép megjelenik majd. Ez a sík legtöbb esetben a vízszintes helyzetű koordináta sík. Ebből a szemszögből kirajzolunk, általában egy kisebb felbontású képet, ez lesz a tükörkép. Majd a tényleges, megjelenített kép rajzolásakor, rátextúrázzuk a tükörképet a tükröződő felületre, perspektivikus leképezést használva a textúra koordináták meghatározásához. Ez a technika veszélyes lehet, mert minden egyes ilyen tükörkép, egy többlet-képkocka kirajzolásával jár, ezért ebből egy, maximum kettő használata javasolt egyszerre. A példaprogramban a padlón megjelenő tükröződéshez (Függelék, 30. ábra) használtam ezt a módszert.

5.3.4. Cube Map alapú, statikus vagy félig statikus tükröződés és fénytörés

A Cube Map technika konvex, nem túl nagyméretű tárgyakon használható fénytörés és tükröződés megvalósítására. Lényege, hogy a tárgy középpontjából kirajzolunk hat képet, az x, y és z tengelyek negatív és pozitív irányába állított kamerával. Ezekből a képekből létrejön a cube map. Rendereléskor a grafikus kártya minden vertex-hez meghatároz egy-egy textúra koordinátát a normálvektora alapján, és kirajzolja a tárgyra a megfelelő képet. A technika ereje abban rejlik, hogy egyrészt a már beállított képek párosával megcserélhetők (egyszerű textúra transzformációval), így mind tükröződés, mind fénytörés megvalósítására képes, valamint a kocka hat oldalát alkotó textúrákat elég csak egyszer kirenderelni, ha a tárgyunk és környezete nem mozog. Bevezethető a félig statikusság fogalma, azaz a cube map-et nem

fixen használjuk, hanem újrarajzoljuk, amikor arra szükség van, például nagyobb, a tárgy környezetében bekövetkezett változás esetén, vagy a tárgy mozgásakor. Mivel, egy képkocka alatt a kocka mind a hat oldalának újrarajzolása nagyon nagy teljesítmény csökkenést okozna, az újrarajzolást inkább lépésenként érdemes megvalósítani, minden képkocka alatt 1-1 oldalt.

Az OpenGL nyújtotta Cube Map megvalósítási eszközökről az OpenGL Red Book [1] Texture Mapping fejezetében olvashatunk bővebben.

5.3.5. Depth Shadow Map alapú vetett árnyék

A vetett árnyékok készítésének legegyszerűbb módja az úgynevezett Depth Shadow Map használatán alapuló technika. A lényeg, hogy a képet kirajzoljuk a fényforrás szemszögéből, de a puffernek, csak a mélységi értékeket tartalmazó részével foglalkozunk. A mélységi értékeket kimentjük egy depth texture formátumú textúra objektumba, majd a végleges kép kirajzolásakor, ezt a textúrát perspektivikusan leképezzük a tárgyak felületére, a fényforrás szemszögéből, és összehasonlítjuk a benne található mélységi értékeket a leképzett textúra koordináták R értékeivel. Ha az R érték kisebb, mint a textúrában tárolt érték, akkor az adott pont színe fekete, egyébként fehér lesz. Ez a technika nagyon pontatlan, rossz felbontású, esetenként villódzó vetett árnyékokat fog eredményezni. Ugyan implementálásra került a motorban, de alkalmazni nem alkalmaztam, mert nem volt elfogadható a kapott eredmény.

A technika részletes megvalósítása megtalálható az OpenGL Red Book [1] Texture Mapping fejezetének legvégén.

5.3.6. Perspective Shadow Mapping technika

A Perspective Shadow Mapping technika az előző technika továbbfejlesztése. Lényege, hogy az árnyékokat tartalmazó textúra megrajzolását nem a fényforrás eredeti helyéről végzi, hanem a fényforrást transzformálja úgy, hogy a kamera helyzetét figyelembe véve, a lehető legnagyobb felbontású árnyék textúrát tudja előállítani. Lényege, hogy amit a kamera nem lát, azokat az árnyékokat nem is kell megrajzolni. Tökéletesen alkalmazható kisméretű, gyorsan mozgó tárgyak árnyékának megrajzolására. A technika részletes tanulmányozása és implementálása még a jövő feladata.

Bővebb információ Marc Stamminger, George Drettakis – Perspective Shadow Maps [11] cikkében olvasható.

5.3.7. Többrétegű, RGBA textúra alapú vetett árnyék

Ez egy saját módszer, amelynek megvalósítása még csak elméleti fázisban jár. A lényege, hogy a nem gyorsan mozgó, vagy álló tárgyak árnyékainak rajzolására is teljes színkomponenseket tartalmazó textúrákat alkalmazunk, ezáltal az árnyékok színezhetőek, valamint részletesebbek, szebbek lesznek, és a távolság függvényében halványodhatnak is. A probléma az, hogy így nem rendelkezünk információval arról, hogy melyik vetett árnyéknak melyik tárgyon kéne megjelenni, nem tudjuk kivédeni az öneltakarást, vagy az olyan tárgyak kitakarását, amelyek közelebb vannak a fényforráshoz, mint az adott árnyékot vető tárgy. Ennek megoldására a módszer egy három-dimenziós textúrába tárolja a vetett árnyékokat. A textúra egyes rétegei adott azonosítójú testek vetett árnyékait tartalmazzák. Kirajzoláskor minden tárgy a saját azonosítójának megfelelő textúra szintet kapja meg, mint árnyék textúrát. Az azonosítók kiosztása lehet statikus, vagy dinamikus (a fényforrástól való távolságuk alapján). A textúra szinteket elég csak akkor újrarajzolni, ha valamelyik tárgy megmozdul, és akkor is csak az adott tárgy azonosítójától kisebb azonosítójú szinteket. Előzetes becslések szerint egy 128 szintből álló textúra elég lenne tetszőleges összetettségű jelenetek árnyékainak tárolására. A szintek újrarajzolása természetesen nem egy képkocka alatt történne, hanem 1-1 képkocka alatt 1-1 szint kerülne újrarajzolásra. Ez 128 szintnél, akár 3-4 másodpercig is eltarthat, emiatt nem megfelelő ez a technika gyorsan, és sokat mozgó tárgyak vetett árnyékának megvalósítására. Épületek, hegyek, sziklák, vagy nem mozgó tárgyak vetett árnyékát viszont sokkal szebben, és gyorsabban lehetne képes megjeleníteni, mint a mélységi értékeken alapuló technikák.

Emellett kombinálható a már korábban említett Light Map alkalmazásával, ami nagyon nagy optimalizációt jelentene a lassan, vagy egyáltalán nem mozgó fényforrások megjelenítésében.

6. Optimalizálás

Egy valós idejű 3D motorral szemben támasztott legnagyobb elvárás, még a mai nagy teljesítményű hardverek mellett is, a sebesség. A motornak a lehető legkevesebb CPU erőforrást kell elvinnie, mert arra szüksége lesz majd a mögé kerülő alkalmazásnak. Játékoknál például a fizikai számításokhoz, és a mesterséges intelligencia működtetéséhez. Mindemellett a motornak a lehető legjobban ki kell használni minden grafikus kártya nyújtotta optimalizációs lehetőséget.

Az OpenGL egyik optimalizációra szánt eszköze a vertex array, ennek a dinamikus geometriával rendelkező tárgyak esetében van jelentősége. A lényege, hogy a fejlesztőnek a tárgy vertexeit (háromdimenziós pont) egy tömbbe kell rendeznie. Ezután ennek a tömbnek a mutatóját átadja az OpenGL környezetnek, és megad mellé egy másik tömböt, amely a háromszögelés folyamatát írja le a vertexek indexeinek sorozatával. Ez alapján a megjelenítéskor az OpenGL bejárja a tömböt, és megjeleníti a háromszögeket. A tömb tartalma dinamikusan változtatható, ezért jó ez a technika például vízfelszín megjelenítésére. Ugyanilyen tömbben tárolódnak a vertexek normái és textúra koordinátái is. Bővebb információ az OpenGL Red Book [1] State Management and Drawing Geometric Objects fejezetében található.

A statikus testek, OpenGL környezeti változtatások, mátrixműveletek és textúra hozzárendelések optimalizálására az OpenGL az úgynevezett display listeket használja. Ezek egyedi azonosítóval rendelkeznek. Létrehozásukkor a megadott OpenGL parancsok belefordítódnak egy display list objektumba. Ezt a fordítást a grafikus kártya végzi, és az eredményt a saját formátumában tárolja. Amikor megjelenítésre kerül a sor, csak meg kell hívni a már lefordított display listet. A belefordított műveletek sebessége így a lehető legjobb lesz. Hátrányuk, hogy statikusak, a fordításukkor megadott parancsok hatása mindig ugyanaz lesz minden lefuttatáskor. Viszont, ahogy arra már láthattunk példát az előző fejezetben, megfelelő egymásba ágyazásukkal, és újrafordításukkal nincs olyan, amit ne tudnának megvalósítani. Bővebb információ az OpenGL Red Book [1] Display Lists fejezetében található.

Egy manapság egyre nagyobb teret nyerő eszközrendszer, a shaderek segítségével, még a dinamikus geometriájú tárgyak is display listbe foglalhatóak. Például egy vízfelszín esetében, a vizet mozgató függvény egy vertex-shaderben megvalósítható. Mivel a vertex-shader a

display list feldolgozása után fut le, képes a display listben foglalt vertexek helyzetét manipulálni. A shaderekről bővebben az OpenGL Orange Bookban [13] olvashatunk.

A pusztán egymás után helyezett, vertex arrayben tárolt és display listben tárolt tárgyak kirajzolásának sebességét jól tükrözi a következő teszt, melyet 2006-ban végeztem.

Az alkalmazás feladata egy körülbelül 2000 háromszögből álló tárgy kirajzolása volt, egy NVidia GeForce 6600 grafikuskártyán:

- Vertex array és display list használata nélkül, pusztán az OpenGL parancsok egymás utáni hívásával történő kirajzolás sebessége **~40fps** (képkocka per másodperc) volt.
- Vertex array használatával ez megnövekedett **~200fps** magasságába.
- Az első fázisban hívott OpenGL parancsok display listbe fordításával az elért sebesség viszont **~750fps** volt.

Ebből nagyon jól látszik, hogy egy modern motornak display listek segítségével kell megvalósítania minden kirajzolást, a dinamikus geometriájú tárgyak alakváltoztatását pedig szintén display list és vertex-shader segítségével kell megoldani a maximális sebesség elérése érdekében.

A példaprogram, melyet készítettem, jelenlegi állapotában, egy olyan jelenet kirajzolását, amely több mint 10 000 háromszöget tartalmaz, négy réteg textúrával és valós idejű tükröződéssel (egy effektív képkocka alatt a jelenetet kétszer rendereli ki) 640×480 pixeles felbontás mellett **~450fps**, 1680×1050 pixeles felbontás mellett pedig **~250fps** átlagsebességre képes. Ez az érték várhatóan nem csökken drasztikusan a jelenet összetettségének növelésével, egészen addig, amíg el nem éri a grafikus kártya állította korlátokat.

Az értékek mérésénél használt konfiguráció:

ASUS P5P800-SE alaplap (Intel 865PE chipset), Intel Celeron D 331 @ 3200MHz processzor, 1GB Dual DDR400 memória és NVidia GeForce 7600GT AGP (560/1450MHz, 256MB) grafikuskártya.

7. Összefoglalás

A diplomamunkám célja egy jól működő, optimalizált, emellett teljes mértékben objektum-orientált alapokra fektetett 3D motor alapjainak elkészítése volt. Ezt a célt, véleményem szerint, sikerült elérni, nyitva hagyva a folytatás lehetőségét.

A fejlesztés során sikerült létrehozni, egy .Net assemblybe (dll fájl) foglalva, egy olyan motort, amelyben mindent osztályok reprezentálnak, a köztük lévő kapcsolatok jól definiáltak és szorosan kötődnek az OpenGL architektúrához, így tökéletesen képesek optimalizálni a működésüket. Valamint sikerült megmutatni, hogy egy alapvetően C nyelvre készült, eljárás-orientált függvénykönyvtár és az arra épülő 3D alkalmazások sebességveszteség nélkül átvihetők C# alá, .Net környezetbe, objektum-orientált szemléletet követve.

A későbbi fejlesztések célja lehet a shaderek bevezetése, egy-egy osztály létrehozása VertexShader és PixelShader néven, majd megoldani, hogy ezek hozzárendelhetők legyenek Solid és esetleg Texture vagy Material osztályokhoz is.

Az ismertetett technikák közül, a Cube Map segítségével megvalósított félig statikus tükröződés és fénytörés, a saját, több rétegű RGBA shadow map alapú technika és a gyorsan mozgó, kis tárgyak esetében a Perspective Shadow Map technika még nem került, vagy csak részben került implementálásra, így ezek teljes megvalósítása még a jövő feladata.

I. Irodalomjegyzék

[1] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo and Jackie Neider – OpenGL Programming Guide, 5th Edition (Red Book) (2005) Addison-Wesley

[2] MSDN Library – Delegate Class
http://msdn2.microsoft.com/en-us/library/system.delegate.aspx

[3] Stefan Brabec, Hans-Peter Seidel – Extended Light Maps

[4] MSDN Library – TypeConverter Class
http://msdn2.microsoft.com/en-us/library/system.componentmodel.typeconverter.aspx

[5] MSDN Library – UITypeEditor Class
http://msdn2.microsoft.com/en-us/library/system.drawing.design.uitypeeditor.aspx

[6] MSDN Library – Getting the most out of the .Net Framework PropertyGrid control
http://msdn2.microsoft.com/en-us/library/aa302326.aspx

[7] MSDN Library – Reflection
http://msdn2.microsoft.com/en-us/library/cxz4wk15.aspx

[8] MSDN Library – Type Class
http://msdn2.microsoft.com/en-us/library/system.type.aspx

[9] MSDN Library – Module Class
http://msdn2.microsoft.com/en-us/library/system.reflection.module.aspx

[10] Michael Garland, Paul S. Heckbert – Surface Simplification Using Quadric Error Metrics

[11] Marc Stamminger, George Drettakis – Perspective Shadow Maps

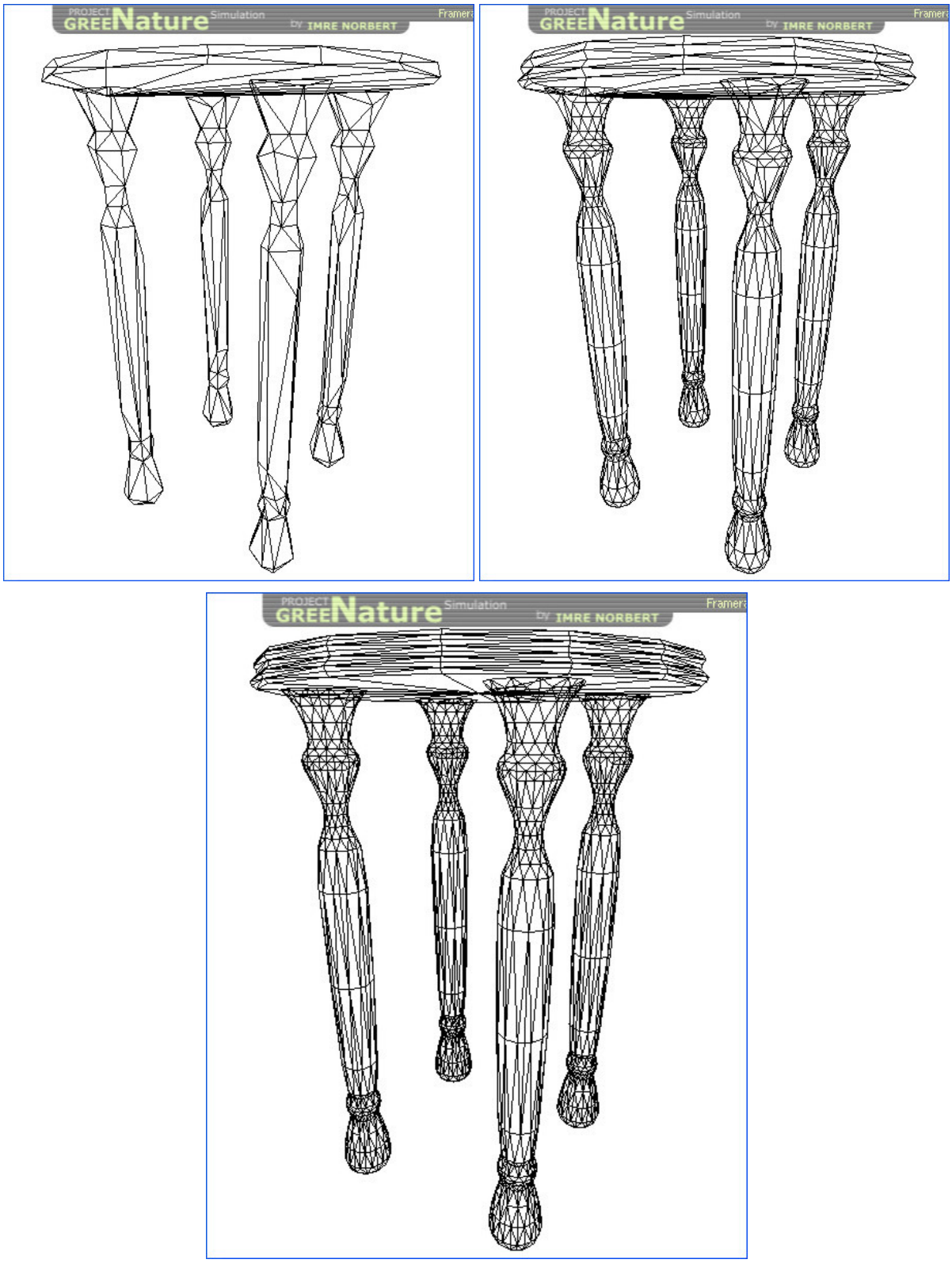
[12] Denton Woods – Developer's Image Library Manual (2002)

[13] Randi J. Rost – OpenGL Shading Language, 2nd Edition (Orange Book) (2006) Addison-Wesley

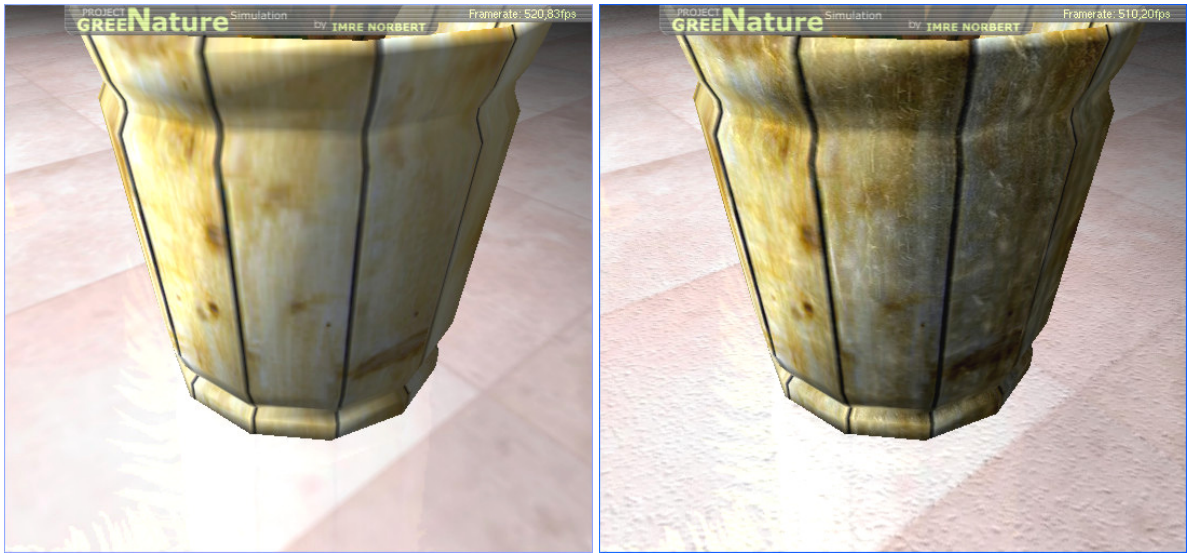
[14] OpenGL Architecture Review Board, Dave Shreiner, Randi J. Rost – OpenGL Library, 3rd Edition (Blue Book) (2006) Addison-Wesley

[15] Paul Martz – OpenGL Distilled (2006) Addison-Wesley

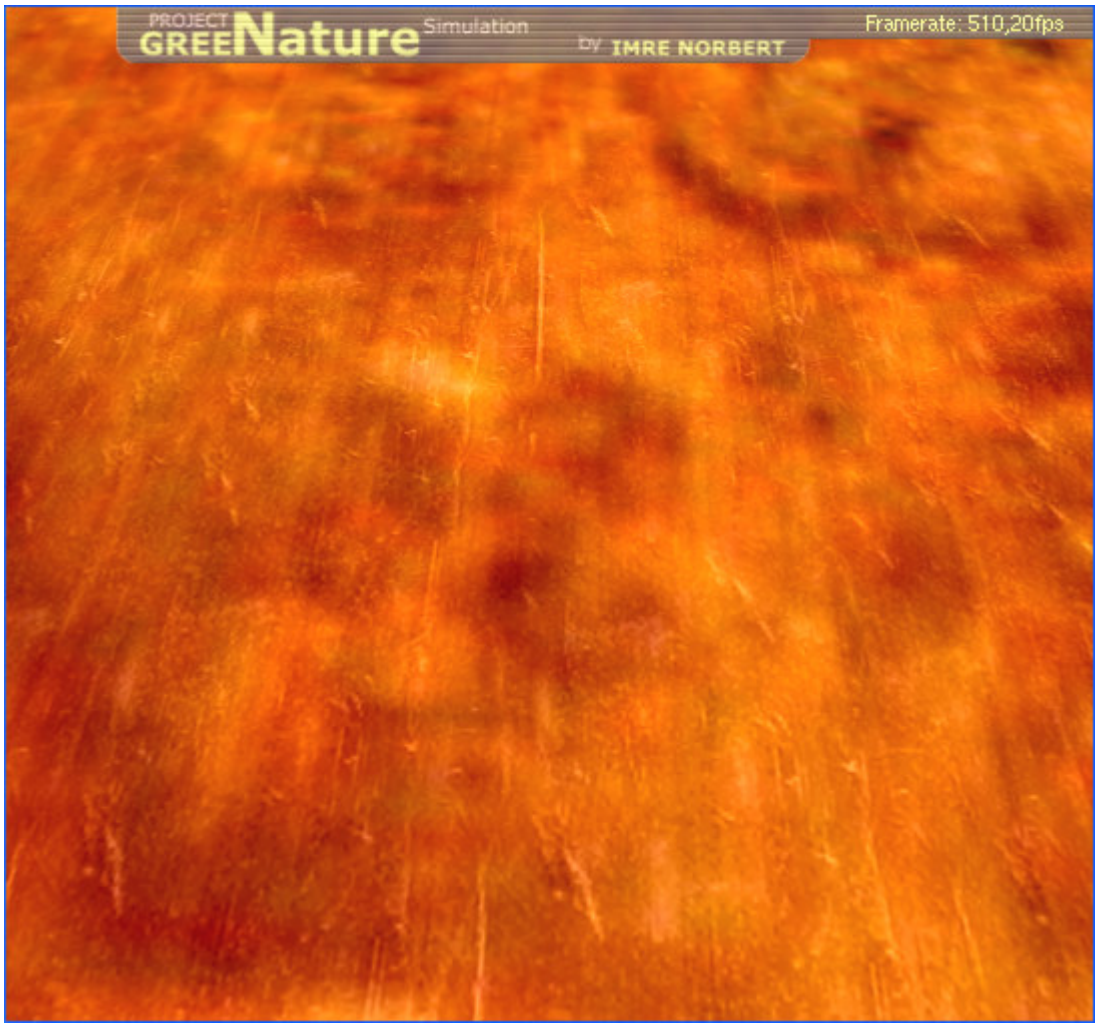
II. Függelék



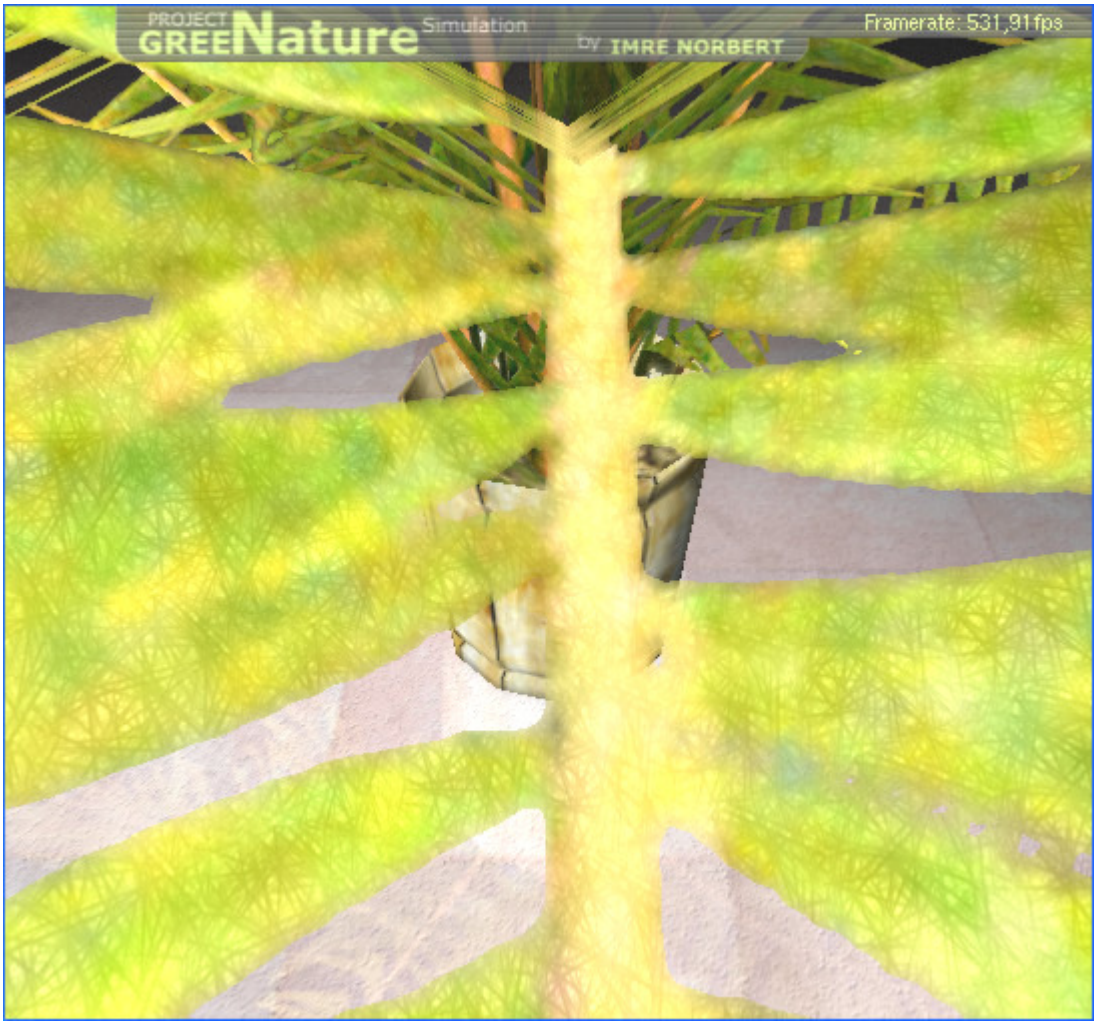
26. ábra: LoD példa – Sorban egy szék Low, Medium és High szintjei.



27. ábra: Detail Texture példa – Dézsa és padló detail texture nélkül, és detail texture-rel.



28. ábra: Detail Texture példa – Fa asztal lapja. A karcolások detail texture segítségével kerültek rá.



29. ábra: Alpha Test példa – Pálmalevél alpha átlátszóság és alpha vágás, valamint kétoldalas textúra segítségével.

30. ábra: Reflection példa – Padló valós idejű tükröződés nélkül, és vele.

31. ábra: Példa program – Egy kép a példaprogramból.

Köszönetnyilvánítás

A diplomamunkám elkészítésében nyújtott segítségéért, a megfelelő könyvek biztosításáért, és a korábbi, általa tartott tárgyak keretében, az OpenGL alapjaiba való bevezetésért szeretnék köszönetet mondani a témavezetőmnek, Dr. Tornai Róbertnek.