DEBRECENI EGYETEM
INFORMATIKAI KAR

C# ÉS KOMPUTERGRAFIKA

TÉMAVEZETŐ:

Dr. Kovács Emőd

Tanszékvezetõ fõiskolai docens
ESZTERHÁZY KÁROLY FŐISKOLA, EGER

KÉSZÍTETTE:

Bóta Balázs

Programtervező matematikus hallgató

DEBRECEN
2008

TARTALOMJEGYZÉK

BEVEZETÉS

A DOS operációs rendszer idejében az olyan szoftverek fejlesztése, amely bármilyen multimédiás képességgel rendelkezett, megkövetelte a hardver alacsony szintű programozását és magas szintű ismeretét. Mindemellett szükség volt az elméleti ismeretekre, amelyeket a programozó algoritmusain keresztül felhasznált. Ekkor szinte kizárólag a hivatásos játékfejlesztők sajátja volt például az a tudás, hogyan kell animációkat megjeleníteni.

Napjainkban egész más a helyzet: olyan eszközkészletek születtek, amelyek lehetővé teszik, hogy a multimédiás alkalmazások fejlesztése a hadverkörnyezettől minél kevésbé meghatározott módon történjen. Ezek az eszközkészletek nevezhetők API-nak (alkalmazásfejlesztői interfésznek) olyan értelemben, hogy az adott hardver szolgáltatásait teszik hozzáférhetővé egy magasabb absztrakciós szinten, illetve a hardver szolgáltatásaira vonatkozó minden kérést az eszközkészlet közvetít a hardver felé.

Ahhoz azonban, hogy jó alkalmazások szülessenek, az eszközök megléte, sőt, alapos ismerete sem elegendő. Régebben a programozónak kellett megmondania, mit szeretne látni, és meg kellett mondania azt is a hardvernek, hogyan kell ezt elérni. Ma nem kell közvetlenül ismerni a „hogyan”-ra a választ a hardver szempontjából. Elengedhetetlen viszont tisztában lenni a rendelkezésre álló eszközökkel, azaz hogy az adott API-t hogyan kell használni.

Másrészt az idők során nem változott, hogy a szükséges elmélet, algoritmusok ismerete is elengedhetetlen. Erre tipikus példa a számítógépi grafika, még pontosabban a háromdimenziós képi megjelenítés. Míg régebben gyakorlatilag a játékfejlesztők „kiváltsága” volt például az olyan specifikus módszerek ismerete, hogyan kell képeket megjeleníteni vagy animációt létrehozni, manapság bárki, aki általában szoftverfejlesztéssel foglalkozik, szembesülhet ilyen feladatokkal. Az API természetesen megmondja, mit tehetünk meg a hardverrel, annak milyen szolgáltatásai vannak, de nem mondanak túl sokat arról, ezek hogyan is működnek a gyakorlatban.

Tehát a mai multimédiás szoftverfejlesztésben két alapvető követelménynek kell a programozónak eleget tenni:

- A rendelkezésre álló API ismerete, illetve
- Az alkalmazási területnek megfelelő elméleti ismeretek.

A tapasztalat az, hogy egy API használatának elsajátítása több-kevesebb idő alatt, egyénileg történik. Gyakorlatilag minden, széles körben használt API mögött igen jelentős felhasználói-fejlesztői internetes közösség áll. A használat tényleges oktatása viszont—hazai vonatkozásban legalábbis—még várat magára. A megfelelő elméleti ismeretekre szintén elsősorban az internetről, illetve tudományos forrásokból jutnak a fejlesztők. Ebben a vonatkozásban—nyilván az adott területtől függően—a szakmai képzés, például az egyetemi tanulmányok is jelentős segítséget nyújthatnak.

Jelen dolgozatban mindkét, előbb felvázolt szempontot szem előtt tartva egy olyan szoftvert mutatok be, amely a DirectX API oktatásához és megismeréséhez nyújthat segítséget. A szoftver a háromdimenziós grafika területére koncentrál, mivel ez egy olyan terület, amely az egyetemi szakmai képzésben is szerepel.

Célom tehát kettős: szeretnék egy olyan szemléltető szoftvert bemutatni, amely egy konkrét API elméleti és gyakorlati alapjainak oktatását támogatja, illetve az API programozásával most ismerkedők számára hatékony segítséget nyújt. A célzott felhasználói réteget egyrészt azok a leendő vagy kezdő fejlesztők alkotják, akik már rendelkeznek bizonyos grafikai előismeretekkel, illetve a képzésük során most ismerkednek a háromdimenziós megjelenítéssel. A másik csoportban azok a programozók vannak, akik szeretnék a DirectX alapvető használatát elsajátítani. A szoftver kereteit elsődlegesen a Komputergrafika című tárgy törzsanyagánál igyekeztem meghúzni. Bizonyos esetekben ezért a szoftver képességei túlhaladják a DirectX által támogatott funkciókat.

A munkám során a DirectX 9.0c verziójával dolgoztam. A szoftver programozási nyelve a C# volt, a fejlesztés alapja a .NET Framework 2.0-ás verziója. Ezért jelen dolgozatban a DirectX programfejlesztési készlet .NET Framework-öt támogató Managed DirectX API kerül bemutatásra.

Bár a DirectX felhasználási területei miatt a legtöbb esetben az API programozása C++ nyelven történik, azért esett mégis a választásom a C# programozási nyelvre, mivel véleményem szerint a kezdő programozók illetve a DirectX programozásával most ismerkedők számára a DirectX megértését jobban elősegítő kód állítható így elő.

A DirectX felépítésének bemutatása után az oktatószoftver elemzésére térek rá. A programban demonstrált elméleti anyag a háromdimenziós grafika azon területeit érinti elsősorban, amelyek az API használata során igen hamar szükségessé válnak. Konkrétan megmutatom, hogyan történik a DirectX-ben egy háromdimenziós világ megjelenítése, az objektumok adatainak beolvasásától kezdve egészen a kamera tulajdonságainak beállításáig. Speciálisan vizsgálom, milyen többletet jelent programozási szempontból az animációk kezelése, külön hangsúlyt fektetve az animáció adatainak állományból való kinyerésére. Bemutatom továbbá, a program hogyan támogatja a háromdimenziós transzformációk illetve a homogén transzformációs mátrixok használatának elsajátítását, valamint a megvilágítás és árnyalás alapjainak megismerését.

A DirectX mint API

Mint említettem, a DirectX felfogható egy absztrakt rétegnek, amely a multimédiás hardver fölé ékelődve, annak szolgáltatásait közvetíti a saját maga által definiált módon; a programozó számára ez azt jelenti, hogy a DirectX minden szolgáltatása interfészeken keresztül érhető el. Maga a DirectX igazából nem egy konkrét API, hanem egy API-gyűjtemény. A különböző részek különböző jellegű szolgáltatásokat nyújtanak, és részben különböző típusú hardverelemeket céloznak meg. A jelenleg implementált részek a következők:

- DirectDraw: A videomemória és a grafikus hardver bizonyos funkcióinak elérését biztosítja. Segítségével vizsgálhatók az adott hardver tulajdonságai, illetve egy konkrét hardverfunkció támogatása is. Ezek miatt gyakorlatilag minden grafikát használó alkalmazás alapja, legyen szó két-, vagy háromdimenziós grafikáról.
- Direct3D: Háromdimenziós grafikát használó alkalmazások fejlesztését támogatja.
- DirectInput: Input- és outputeszközök kezelését teszi lehetővé,különös tekintettel a játékoknál használt speciális eszközökre, pl. joystick, kormány.
- DirectSound: A hardveres és szoftveres hangkeverést és lejátszást segíti.
- DirectMusic: Hardveres vagy szoftveres szintetizátor segítségével előállított hangmintákkal dolgozik.
- DirectPlay: Alkalmazások összekapcsolódását teszi lehetővé hálózaton keresztül, elsősorban többjátékos üzemmód megvalósítása céljából.
- DirectSetup: A szükséges DirectX futásidejű komponensek telepítését vagy a meglevő összetevők frissítését végzi el az alkalmazás telepítésekor, annak igényeihez mérten.

Az egyszerűség kedvéért a DirectX, Managed DirectX, Direct3D megnevezések helyett néhol az általános, API megjelölést használom a bemutatott eszközkészletre.

Jelen dolgozat közvetlenül a Direct3D, illetve részben—ezen keresztül, közvetve—a DirectDraw vizsgálatával foglalkozik.

A DirectDraw alapjai

A Direct3D specifikusan a háromdimenziós grafikai megjelenítésért felel, az alapvető grafikus szolgáltatásokat a DirectDraw-n keresztül éri el, még akkor is, ha az adott nyelvi implementáció ezt a

rétegződést transzparenssé teszi. A DirectDraw lényegében a legközelebbi absztrakt réteg a grafikus hardver fölött a DirectX-ben, amely a következő funkciókat és beállításokat kezeli:

- Együttműködési szint (cooperative level)

Az együttműködési szint azt modja meg, az alkalmazás milyen módon használja a grafikus erőforrásokat. Ez legegyszerűbben fogalmazva annak beállítását jelenti, hogy az alkalmazás teljes képernyős módban fut (és ekkor a grafikus erőforrásokat kizárólagosan használja), vagy normál alkalmazás-ablakban. Ez utóbbi esetben több alkalmazás is hozzáférhet ugyanahhoz a grafikus erőforráshoz (pl. a videomemóriához). Ez azt jelenti, hogy más futó programok tevékenysége hatással lehet az alkalmazás DirectX hívásainak eredményére.

- Megjelenítési módok lekérdezése, beállítása

A megjelenítési módot alapvetően egy adott felbontás és a színmélység azonosítja. A megjelenítési módok csoportosíthatók aszerint, hogy a színeket hogyan kódolják. Az indexelt színkezelésnél a színértékek egy megfelelő indextábla vagy paletta indexei. Ekkor a megjelítési mód által megadott színmélység ennek a palettának a méretét határozza meg. Hogy egy adott indexhez milyen konkrét szín fog tartozni, az a paletta beállításától függ.

Ha viszont a megjelenítési mód nem használ palettát, úgy a színadatokat közvetlenül kódolja; ekkor a színmélység az egy pixel színének leírására használható adatmennyiséget adja meg.

Nyilván minél nagyobb a felbontásban megadott szélesség, magasság illetve színmélység, annál több videomemória szükséges az adott mód használatához. Elképzelhető, hogy egyes módokat bizonyos régebbi hardverek nem támogatnak, például a megfelelő memória hiánya miatt. Ezért a DirectDraw lehetővé teszi az elérhető megjelenítési módok számbavételét és valamennyi elérhető módhoz támogatást nyújt.

- Surface objektumok kezelése

A Surface gyakorlatilag egy felület, amelyre a rajzolás történik. Pontosan fogalmazva a Surface egy összefüggő memóriablokk, amely képi információ tárolására használható. Ez a memóriablokk lehet a video-, vagy akár a rendszermemóriában is. Többféle felületet hozhatunk létre: elsődleges felületről beszélünk (primary surface), ha a felülethez rendelt memória közvetlen megjelenítésre kerül. Ha a felületen levő információ nem jelenik meg közvetlenül a képernyőn, off-screen felületről van szó.

Napjainkban elég általános az ún. kettős pufferelés használata, ilyenkor az elsődleges felülethez legalább egy off-screen felület tartozik. Az elsődleges puffert leggyakrabban elülső puffernek, a hozzá tartozó off-screen felületet hátsó puffernek hívják. Az ún. megjelenítési felület (render target) az a hátsó puffer (az esetlegesen hozzá tartozó mélységi puffer felülettel együtt), amelyre a kirajzolás legközelebb történni fog.

A DirectDraw által támogatott funkció a hardveres overlay is, azaz olyan felületet hozhatunk létre, amelynek tartalma mintegy az elsődleges felület „előtt” jelenik meg.

- Vágás

A DirectDraw által végzett vágás nem más, mint a képi információ adott, téglalap alakú tartományokon kívül eső részeinek elhagyása. Több ilyen tartomány is definiálható, amelyek ún. vágási láncot (clip chain) alkotnak.

- Flipping

A folyamatos mozgás hatásának megőrzése érdekében a képet egy másodperc alatt többször is frissíteni kell. Ezért ma már nem csak egy felületet használnak a rajzoláshoz, hanem legalább kettőt: mialatt az egyik kép látható, addig egy másik felületre történik a rajzolás. A flipping nem más, mint több felület periodikus felcserélése.

Az előbb már említettem a kettős pufferelést. Ekkor a flipping a következő mechanizmust jelenti:

1. Rajzolás a hátsó pufferre.
2. A hátsó puffer tartalmának megjelenítése.
3. Az előbbi hátsó puffer a jelenlegi elülső puffer lesz.
4. Vissza 1-re.

A flipping intenzitása nem lehet nagyobb, mint a monitor frissítési frekvenciája, ugyanis a flip mechanizmus a megjelenítő y-szinkron jeléhez van igazítva. Például egy CRT monitor esetén akkor történik a flip, amikor a képernyő végigpásztázása befejeződött és az elektronika deaktiválja a sugárvetőket, amíg a jobb alsó sarokból újra a bal felsőbe pozícionál. Ennek célja az animációk megjelnítésekor tapasztalható szakadási effektus kiküszöbölése. Szakadásról ugyanis akkor beszélünk, ha az objektum mozgása a frissítési mechanizmus közben történik, és az objektum egyik fele a régi, a másik fele az új helyen jelenik meg.

Ahhoz, hogy egy alkalmazás bármilyen megjelenítésbe kezdjen, először ezeket a funkciókat és paramétereket be kell állítania. Láttuk például azt, hogy ha az alkalmazásunk ablakos módban fut, akkor más programok hatást gyakorolhatnak a mi programunk grafikus hívásainak kimenetelére. Nagyon fontos ezért, hogy rendelkezzünk az erőforrásokhoz való hozzáférésről is bármilyen DirectX hívás előtt. Általánosan a DirectDraw inicializálása a következő lépéseket foglalja magában:

1. A DirectDraw objektum létrehozása a megfelelő COM interfészeken keresztül.
2. Az együttműködési szint beállítása.
3. A megfelelő megjelenítési mód kiválasztása.
4. A primary és offscreen felületek létrehozása, a flipping beállítása.
5. A paletta beállítása, ha szükséges.

Mint említettem, a DirectX és így a DirectDraw is azzal a céllal készült, hogy a hardvertől minél függetlenebb módon tegye használhatóvá annak szolgáltatásait. Jelenleg gyakorlatilag minden grafikus kártya támogatja a DirectX-et, viszont ez a támogatás a hardverek sokfélesége miatt különböző szintű. Ezért az inicializáló lépés sokszor kihasználja a DirectDraw információs szolgáltatásait is, például a 3. lépésnél.

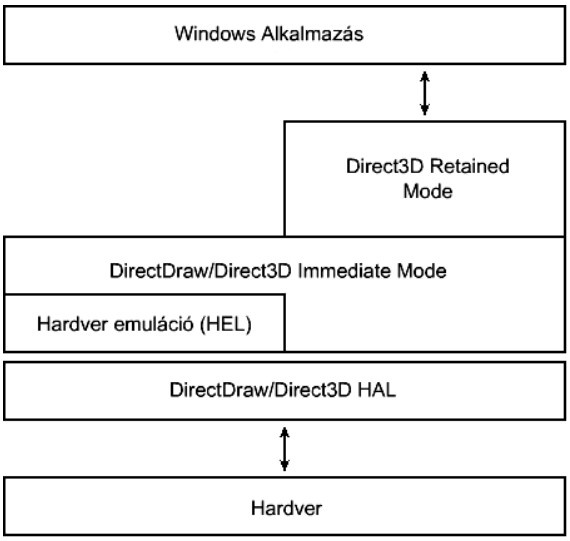
A DIRECT3D ALAPJAI

A Direct3D a DirectX 2-es verziójától kezdve része az API gyűjteménynek. A Direct3D eszközei két csoportra oszthatók:

Immediate Mode API: Alacsony szintű, hardverközeli és ezért elég rugalmas interfész. A programozó maga adhatja meg akár a megjelenítés minden lépését, az objektumok reprezentációját.

Retained Mode API: Az Immediate Mode-ra épülve és azt kiegészítve készült, a fő célja az egyszerű használat. A felhasználónak akár az objektumot reprezentáló adatok szerkezetét sem kell ismernie, egyszerűen megmondja, milyen alakzatot akar látni. A megjelenítés jelentősebb koncepcióival (pl. megvilágítás, vetítés) absztrakt eszközökön keresztül kerül kapcsolatba a felhasználó, ezek paramétereit bizonyos keretek között módosíthatja.

Az 1. ábrán a Direct3D API felépítése, illetve absztrakciós szerepe látható.



1. ábra: A Direct3D alapvető felépítése

Az API igyekszik minden, a hardver által támogatott funkciót a hardverrel elvégeztetni az optimális teljesítmény érdekében. Ez azt jelenti, hogy optimális esetben az API a HAL (Hardware Abstraction Layer) rétegen keresztül szolgálja ki az alkalmazás grafikus hardverhez érkező igényeit—tehát maga a hardver képes fogadni azokat. Ellenkező esetben, ha ezt a hardver képességei nem teszik lehetővé, a DirectX képes a szükséges funkciókat szoftveresen biztosítani, komoly teljesítményveszteséggel. Ekkor a grafikus szolgáltatásokat a HEL (Hardware Emulation Layer) nyújtja. Mint az ábráról is látszik, a HEL a hardvertől teljesen függetlenül nyújt szolgáltatást. Általában ezt a lehetőséget tesztelési célokra használják.

A DIRECT3D MEGJELENÍTÉSI FUTÓSZALAG

A DirectX-ben minden térbeli modell primitívekből épül fel. A primitív egy egyszerű térbeli alakzat, amely lényegében a megjelenítés feldolgozási egysége. Minden primitív bizonyos számú ponttal van megadva a háromdimenziós térben. A DirectX ezeket vertexeknek nevezi. A háromdimenziós világunk minden objektuma végső soron olyan adatfolyamokra képeződik le, amelyek vertexek sorozatát tárolják a hozzájuk tartozó információkkal együtt.

A megjelenítés előtt megadható a primitív típus, amely információt ad az API-nak, hogy az egyes vertexek milyen kapcsolatban vannak egymással.

A DirectX a következő primitív típusokat használja:

- Pont lista: A bemenő adatfolyam minden vertexe különálló pontként jelenik meg, az egyes pontok teljesen függetlenek egymástól. Ez a primitív típus a következő primitíveket generálja: (az i. vertex az input adatfolyamon, $0 \le i \le n$).
- Vonal lista: A vertexeket az API páronként tekinti. A megjelenítés során az egyes párok tagjai mint egyenes szakaszok végpontjai jelennek meg. A keletkező primitívek: . A vertexek száma páros kell hogy legyen és $n \ge 2$.
- Szakasz-sorozat: Az előbbi speciális esete: feltesszük, hogy a szomszédos szegmensek kapcsolódóak. A primitívek:
- Háromszög-lista: A vertexeket hármasával tekintjük, mint egy-egy különálló háromszög csúcsait. Ez a legáltalánosabban használt primitív típus a háromdimenziós modellek megjelenítése során, hiszen nem tételez fel semmilyen előismeretet a háromszögek közötti kapcsolatokról. Az előálló

primitívek a $[v_0, v_1, v_2], [v_3, v_4, v_5], \dots, [v_{n-2}, v_{n-1}, v_n]$ csúcsok által meghatározott háromszögek.

- Háromszög-sorozat: Az egymást követő háromszögeknek lesz közös élük, azaz a következő sorozat áll elő:
  $[v_0, v_1, v_2], [v_1, v_3, v_2], [v_2, v_3, v_4], [v_3, v_5, v_4], \dots, [v_{n-3}, v_{n-2}, v_{n-1}], [v_{n-2}, v_n, v_{n-1}]$.

Az objektumok megjelenítésekor az objektumot alkotó minden primitív minden vertexén egy meghatározott lépéssorozatot kell végrehajtani, amelynek igen leegyszerűsítve a lényege a háromdimenzós térben megadott csúcspontok leképezése a képernyő síkjára. Ez az ún. megjelenítési futószalag (rendering pipeline). Az elnevezés utal rá, hogy egy többlépcsős, egymásra épülő lépésekből álló folyamatról van szó.

A Direct3D a műveleteket három fő csoportra bontja. Most ezeket mutatom be A folyamat lépései a Függelékben található ábrán is láthatók.

ELŐKÉSZÍTŐ SZAKASZ

Az első szakasz opcionális; akkor szükséges, ha ún. magasabbrendű primitívek megjelenítése szükséges, és a hardver támogatja ezt a funkciót. A DirectX 9 a magasabbrendű primitívek két csoportját támogatja:

- N-Patch: Az eredeti modell felületi tulajdonságait hivatott javítani úgy, hogy ehhez semmilyen plusz információra nincs szükség. A Direct3D a vertexekben megadott pozíciók és a felületi normálisok adatai alapján a primitívekhez egy-egy patch-et generál, amelyet aztán háromszögekre bont.
- RT-Patch: A vertexeket kontrollpontoknak tekintjük, és megadjuk, hogyan álljon elő a sima felület ezen kontrollpontok alapján. A felület pontjai bizonyos függvények súlyozott összegeként állnak elő, amelyek paraméterei a kontrollpontok. Ezek a bázisfüggvények. Minden felületet a bázisfüggvények típusa és algebrai fokszáma határoz meg. Mindkét tulajdonság a Direct3D enumerációs típusain keresztül adható meg; a bázisfüggvények Bézier, B-Spline és interpoláló típusúak. A felületek másod-, harmad-, és negyedrendűek lehetnek.

A kiindulási primitívek tehát ekkor is ugyanúgy vannak megadva, mint normál esetben (általában háromszögekként). Ebben a szakaszban a primitívekhez tartozó kontrollpontok generálása történik, majd a keletkezett primitív háromszögesítése következik.

VERTEXMŰVELETEK

A második szakaszban az egyes csúcspontokon értelmezett műveletek lesznek végrehajtva. Ezek a műveletek a következők:

MEGVILÁGÍTÁS, TRANSZFORMÁCIÓ

Minden vertexhez hozzárendelünk a használt megvilágítási mód alapján számított színértéket, ehhez pedig a vertexeken koordináta-transzfromációt hajtunk végre. Konkrétan az objektum lokális koordinátarendszeréből a vetítési gúla koordinátarendszerébe transzformáljuk a pontokat. A transzformáció nem közvetlen, az egyes állomások a következők:

- Modell-tér (Model space): Minden objektum pontjai az objektum saját koordinátarendszerében adottak. Ez a kiindulási állapot.
- Jelenet-tér (World space): Az egyes objektumok a jelenet-térben egymáshoz képest vannak transzformálva, tehát olyan koordinátarendszerre térünk át, amely a háromdimenziós világunk minden objektumára nézve globális. Az ezt elérő transzformációt végző mátrix a szakirodalomban World matrix

néven ismert. A transzformáció az egyes objektumok forgatását, skálázását, eltolását is magában foglalhatja.

- Nézeti tér (View space): A nézőpont beállítása az objektumok globális koordináta-rendszerében. Ez az a pont, ahonnan a megfigyelő látja a háromdimenziós jelenetünket, ezért ezt gyakran nevezik kameratérnek (Camera space) is. A transzformáció mátrixa a nézeti mátrix (View matrix), amely rendszerint forgatást és eltolást ír le.
- Vetítési tér (Projection space): Az egyes pontokat a nézati gúlában adja meg. A vetítési mátrix (Projection matrix) skálázást végez az objektumokon a használt vetítési módnak megfelelően.

A koordináta-leképezések utolsó eleme a képernyő síkjára (Screen space) történő leképezés, amely csak később történik meg.

LAPELTÁVOLÍTÁS (CULLING)

Azokkal a lapokkal (illetve az ezeket meghatározó vertexekkel), amelyek az adott paraméterek mellett nem láthatóak, felesleges további számításokat végezni. Alapértelmezés szerint azok a lapok lesznek eldobva, amelyek vertexeinek sorrendje az óramutató járásával ellentétes körüljárási irányt határoz meg. Fontos, hogy a lapeltávolítás nem a vertexekben tárolt normálisok alapján történik, hanem az adott háromszöget tartalmazó sík normálvektorát vesszük alapul. Ez pedig a vertexek pozíciójából könnyen kapható: a háromszög síkjára merőleges vektor bármely két, lineárisan független élvektor vektoriális szorzataként áll elő. Ezután meghatározzuk a lapnormális és a nézőpontba mutató vektor szögét (a két vektor skaláris szorzata a szög koszinuszát adja meg). Ha ez tompaszög (a skaláris szorzat negatív), a lap eldobható. A lapeltávolítás módja a CullMode beállításban adható meg (beállítások alatt itt is és a továbbiakban is a megjelenítéshez létrehozott Device objektum RenderState adattagjában levő beállításokat fogom érteni). Ennek lehetséges értékei:

Érték	**Jelentés**
None	Nem dobunk el egyetlen lapot sem.
ClockWise	Azokat a lapokat tekintjük elülső lapoknak, amelyek vertexeinek sorrendje az óramutató járásával megegyező körüljárási irányt ad meg.
CounterClockWise	Azokat a lapokat tekintjük elülső lapoknak, amelyek vertexeinek sorrendje az óramutató járásával ellentétes körüljárási irányt ad meg.

VÁGÁS FELHASZNÁLÓI VÁGÓSÍKOKKAL

A vágás azt jelenti, hogy meghatározzuk a primitívek bizonyos határoló síkokon belül eső részét, a kívül eső részt pedig elhagyjuk. A határvonal mentén új vertexek fognak keletkezni. Az új vertexben a felületi normális, a textúrakoordináták, színadatok a metszett szegmensen levő két vertex értékei közötti interpolációval állnak elő.

A felhasználó a saját vágósíkokat négy együttható $[a, b, c, d]$ segítségével adhatja meg, amely a sík $ax + by + cz + d = 0$ alakú leírását takarja; az $[a, b, c]$ vektor a síkra merőleges. Összesen hat felhasználói vágósík definiálható.

LEHATÁROLÁS A VETÍTŐGÚLÁRA

Az előbbiek szerinti vágást hajtjuk végre a nézeti gúla síkjaira.

HOMOGÉN OSZTÁS

Az eddigi transzformációk elvégzése után a vertexek homogén $[x, y, z, w]$ koordinátákkal adott térbeli pontok, amelyek az ún. kanonizált nézeti térben (lényegében egy téglatest alakú térrészben) helyezkednek el. Vagyis a következők teljesülnek:

$$\begin{aligned} -w \leq x \leq w, \\ -w \leq y \leq w, \\ 0 \leq z \leq w. \end{aligned}$$

A homogén osztás, vagyis a w komponenssel történő osztás elvégzése után a kanonizált nézeti koordinátákra a következők teljesülnek:

$$\begin{aligned} -1 \leq x/w \leq 1, \\ -1 \leq y/w \leq 1, \\ 0 \leq z/w \leq 1. \end{aligned}$$

VIEWPORT LEKÉPEZÉS

Vegyük észre, hogy a vertexek ezen a ponton még mindig térbeli pontok. A homogén osztás után történik meg a vertexek leképezése síkba, azaz a normalizált háromdimenziós térbeli koordinátákhoz a képernyő kétdimenziós síkján értelmezett pontokat rendelünk. A megjelenítési terület által definiált területet a koordinátatranszformáció előző lépéseihez hasonlóan szintén „térnek” nevezik—ez a transzformációs lépésnél már említett képernyőtér (screen space). A leképezést az ún. viewport definiálja. A viewport a képernyő síkjának egy téglalap alakú tartománya, amely a megjelenítést korlátozza egyrészt az xy síkban, másrészt a mélységet is korlátozza a $[0, 1]$ intervallum egy részére.

Alapértelmezés szerint a viewport a teljes képernyőteret (a megjelenítéshez rendelkezésre álló teljes területet) és a teljes mélységi tartományt használja. Az alkalmazás viszont ezt módosíthatja, a következő adatok megadásával:

- X, Y koordináták: A viewport bal felső sarkának koordinátái annak a területnek a bal felső sarkához képest, amelyen a megjelenítés történik.
- Szélesség, magasság (Width, Height).
- MinZ, MaxZ: A mélységi tatományt határozza meg. Ezek az értékek tehát nem a korábban látott vágósíkokhoz kapcsolódnak, hanem a Z-pufferben levő, $[0, 1]$-be eső értékek egy $[MinZ, MaxZ]$ intervallumra lesznek leképezve. Alapértelmezés szerint $MinZ$ értéke 0, $MaxZ$ értéke 1, ekkor a teljes mélységi tartomány ki lesz használva. Ha pl. $MinZ = MaxZ = 1$, minden objektum háttérbe fog kerülni.

  A Direct3D a következő mátrixot használja a viewport leképezéshez:

$$\begin{bmatrix} Width/2 & 0 & 0 & 0 \\ 0 & -Height/2 & 0 & 0 \\ 0 & 0 & MaxZ - MinZ & 0 \\ X + Width/2 & Height/2 + Y & MinZ & 1 \end{bmatrix}$$

A mátrix elvégzi a vertexek koordinátáinak skálázását a viewport méretének és a mélységi tartománynak megfelelően, és eltolja a csúcspontot a megfelelő pozícióba.

PIXELMŰVELETEK

A futószalag harmadik lépéssorozata már a viewport leképezés után kapott pixeleket dolgozza fel egyenként.

HÁROMSZÖG ELŐKÉSZÍTÉS

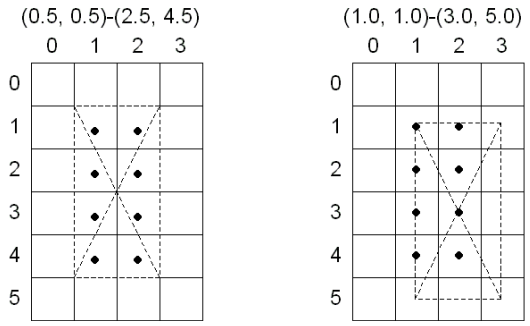
Itt történik az egyes vertexekhez számított színértékek közötti változásintenzitás meghatározása, amely szükséges a háromszögek helyes színkitöltéséhez.

RASZTERIZÁCIÓ

Ebben a lépésben az objektumokhoz a képernyő pontjainak egy véges halmazát rendeljük hozzá. Tehát a raszterizáció során eldől, hogy az adott primitívhez melyik pixelek tartoznak a képernyőn. Legtöbb esetben a pixelek legfeljebb egy objektumhoz rendelhetők hozzá. Viszont döntési helyzet áll elő, ha például bizonyos objektumoknak közös éleik vannak. A Direct3D ilyen esetekben az ún. felső-bal (top-left) kitöltési konvenciót alkalmazza.

Minden pixel esetében annak középpontja alapján hozunk döntést. Minden pixel középpontja egész koordinátákkal rendelkezik. A konvenció a nevét onnan kapta, hogy a vízszintes szakaszok esetén a szakasz bal szélén levő pixelek, illetve függőleges szakaszok esetén a felső pixelek akkor lesznek az objektum pontjai, ha ezen pixelek középpontja az objektumon belül van, vagy az objektum éle a középponton halad át. A szakaszok jobb, illetve alsó végén a képpont már nem lesz az objektum pontja, ha az él éppen a pixel középpontján halad keresztül. A top-left szabály lényegét a háromszögek raszterizációján keresztül vizsgáljuk meg.

A lenti ábrán két téglalap látható, amelyek ugyanazokkal a pixelekkel kerülnek kitöltésre. A bal oldali téglalap esetében nincs szükség a top-left szabályra, mivel minden pixel középpontja pontosan egy háromszögön belül esik, így az ábrán levő négy háromszög mindegyikéhez két-két pixel fog tartozni. A jobb oldali esetben (eltoltuk az előbbi téglalapot fél pixellel) a szabály értelmében az egyes háromszögekhez az óramutató járása szerinti sorrendben rendre négy, kettő, egy, egy pixel fog tartozni.



2. ábra: Raszterizáció a top-left konvencióval

TÖBBSZÖRÖS MINTAVÉTELEZÉS (MULTISAMPLING)

A supersampling élsimítási eljárás gyakorlatban elterjedt speciális esete. Minden pixelt elméleti szinten további alpixelekre bontunk (pl. négyszeres mintavételezés esetén minden pixelhez négy alpixel fog tartozni). Az eredeti supersampling koncepcióval ellentétben viszont nem számítjuk minden alpixelhez újra a szín,

textúra stb. értékeket. Az egyes subpixelekhez tartozó mélységértékeket összehasonlítjuk. Az élsimítást csak azokon a helyeken végezzük el, ahol ezek nem egyeznek (ez ugyanis azt jelenti, hogy egy élhez értünk).

KÖD-EFFEKTUSOK (DEPTH CUEING)

Az objektumok színét a kamerától vett távolság növekedésével egyenletesen egy meghatározott színnel mossuk össze (ez a köd színe). A köd-effektusok a pixelek RGB komponenseit befolyásolják.

Legyen az objektum színe ObjectColor, a köd színe FogColor. Szükség van még egy paraméterre, amely meghatározza, hogy az objektumok színe milyen módon változzon a távolság növelésével. Legyen ennek neve FogFactor. Ekkor a köd által elért effektus általánosan a következő:

$$ObjectColor = ObjectColor \cdot FogFactor + (1 - FogFactor) \cdot FogColor.$$

A köd-effektusok számításakor alapvetően két lehetőség van: a FogFactor paraméter értéke az egyes vertexekhez kerül kiszámításra (ekkor a raszterizáció során a közbülső pixelekre interpoláció fog történni), illetve minden egyes pixelhez újraszámoljuk (ez table fog néven is ismeretes). A számítások alapvetően mindkét esetben ugyanazok. A FogFactor értékei háromféleképpen kerülhetnek kiszámításra:

- Lineáris: Adott két távolságérték, z_s és z_e. A FogFactor értéke ezen $[z_s, z_e]$ intervallumon belül lineáris függvényt követve fog változni:

$$f(z) = \begin{cases} 1 & z < z_s \\ (z_e - z)/(z_e - z_s) & z \in [z_s, z_e] \\ 0 & z > z_e \end{cases}$$

- Exponenciális: Általában ezzel a módszerrel természetesebb színátmenet érhető el. A FogFactor átmenetét egy adott sűrűségi paraméter, a FogDensity is befolyásolja. Ebben az esetben a FogFactor értékei a következő függvény szerint kerülnek kiszámításra: $f(z) = e^{-dz}$.
- Négyzetes exponenciális: Az előbbihez hasonló, de $f(z) = e^{-(dz)^2}$.

Alapértelmezés szerint a számításokban szereplő z értéke az adott pixel Z-koordinátájának abszolútértéke (a nézeti térben). Az ún. távolsági köd (range fog) esetén viszont z a kamerapozíció és a pixel távolsága lesz.

LEHATÁROLÁS A VÁGÓTERÜLETRE (SCISSOR TEST)

A viewport mellett megadható egy téglalap alakú terület, amelyre a Direct3D szintén lehatárolást végez. Ez a vágótéglalap egy Rectangle struktúra, tehát rendszerint a bal felső sarkával és a méreteivel adjuk meg. A mérete nem lehet nagyobb, mint a megjelenítési felület, viszont meghaladhatja a viewport méretét. Ha a bal felső sarok koordinátái $Left, Top$ (a megjelenítési terület bal felső sarkához viszonyítva); a téglalap méretei pedig $Width$ és $Height$, akkor a scissor test pontosan akkor sikeres egy $[X, Y]$ pixel esetén, ha

$$Left \leq X < Left + Width,$$
$$Top \leq Y < Top + Height.$$

A lehatárolás a ScissorTestEnable beállítással kérhető, illetve tiltható le; a vágótéglalap a Device ScissorRectangle adattagjában adható meg.

ALPHA TESZT

Amennyiben a hardver ezt támogatja és az alkalmazás beállítása ezt előírja, minden pixel alpha-csatornájának értékét egy tesztnek vetjük alá. A teszt összehasonlítja a pixel alpha-értékét ($alpha$) egy ún. referencia értékkel ($reference$). A pixel csak akkor vesz részt a további lépésekben, ha ez a teszt sikeres eredményt ad vissza. Egyébként a Direct3D a pixelt eldobja.

Az alpha teszt lehetséges összehasonlító függvényei a Compare enumerációban vannak megadva:

Érték	Jelentés
Always	A referencia értéktől függetlenül a teszt sikeres lesz
Equal	A pixel átmegy a teszten, ha $alpha = reference$.
Greater	A pixel átmegy a teszten, ha $alpha > reference$.
GreaterEqual	A pixel átmegy a teszten, ha $alpha \geq reference$.
Less	A pixel átmegy a teszten, ha $alpha < reference$.
LessEqual	A pixel átmegy a teszten, ha $alpha \leq reference$.
Never	A pixel soha nem megy át a teszten
NotEqual	A pixel átmegy a teszten, ha $alpha \neq reference$.

A referencia érték, az összehasonlítási függvény és a teszt sikerességének meghatározása a következő beállításokon keresztül változtatható meg:

Név	Értékek	Megjegyzés
AlphaTestEnable	true, false	A teszt engedélyezése
AlphaFunction	Compare értékei	Az összehasonlítást végző függvény.
ReferenceAlpha	a [0, 255] intervallumban	0 a teljesen átlátszó, 255 a teljesen átlátszatlan megfelelője.

MÉLYSÉGI TESZT

A mélységi puffer (Z-puffer, illetve W-puffer) a Direct3D azon funkcióit támogatja, amelyekben az egyes objektumok takarási viszonyait kell vizsgálni. A Z-puffer (és az ehhez kapcsolódó stencilpuffer) egy off-screen felület, amely a lapozási láncban levő hátsó pufferekhez hozható létre. Az automatikus létrehozás a Device létrehozásakor a PresentParameters struktúra beállításaiban kérhető, a következőképpen:

```
presentParameters.EnableAutoDepthStencil = true;
presentParameters.AutoDepthStencilFormat = [...];
```

A mélységi puffer a megjelenítési felület minden pixeléhez egy mélységi értéket tárol (amely kiegészülhet egy stencilértékkel is). A Z-puffer támogatással történő megjelenítés a következőképpen történik:

1. A mélységi pufferben minden pixelhez a maximális mélységi értéket tároljuk el.
2. Ahogy a raszterizáció előállítja pixeleket, tesztelést hajtunk végre, amely lényegében egy összehasonlítás, akárcsak az alpha tesztelésnél. Amennyiben az összehasonlítási függvény úgy ítéli meg, hogy az aktuális pixel közelebb van a nézőponthoz, akkor annak a mélységi értékét tároljuk el a puffer megfelelő pozíciójában.

A Z-puffer használatának legnagyobb hátránya akkor válik láthatóvá, ha perspektivikus vetítést alkalmazunk, kicsi mélységi értékek mellett. Ekkor a perspektivikus rövidülés miatt az objektumok nagy része a mélységi tartomány egy kis részébe kerül, ami láthatósági problémákat okozhat. A helyzet orvosolható a közeli és távoli vágosíkok egymáshoz közelítésével, valamint ha a közeli vágósíkot olyan távol tartjuk a nézőponttól, amennyire csak lehet (a teljes mélységi tartomány egy kisebb térbeli tartománynak fog megfelelni).

A teljes megoldást ugyanakkor a W-pufferelés jelentheti. Ekkor ún. homogén reciprok mélységi értékeket használunk, azaz a mélységi pufferbe $1/w$ alakú értékek kerülnek. Ez lehetővé teszi a mélységi értékek egyenletes eloszlását a teljes mélységi tartományon.

A mélységi teszteléshez a következő beállítások tartoznak:

Név	**Értékek**	**Megjegyzés**
ZBufferEnable	true, false	A mélységi teszt engedélyezése, letiltása.
ZBufferFunction	Compare értékei	Összehasonlítás módja
ZBufferWriteEnable	true, false	A Z-pufferbe történő írás engedélyezése, tiltása. Hasznos lehet részlegesen átlátszó objektumok megjelenítésénél.
UseWBuffer	true, false	W-pufferelés engedélyezése, tiltása.

STENCIL TESZT

A stencil teszt segítségével a kapott pixelek tetszőlegesen eldobhatók a mélységi teszt eredménye és a stencilpufferben tárolt érték függvényében. A tesztelés menetében részt vesz a stencilpuffer aktuális értéke, egy referencia stencil érték (az összehasonlítások alapja), illetve az ún. stencil maszk. A tesztelés menete a következő:

1. Bitenkénti ÉS művelet a referencia érték és a stencil maszk között.
2. Bitenkénti ÉS művelet a stencil puffer értéke és a stencil maszk között.
3. 1. és 2. eredményének összehasonlítása az adott függvény segítségével.

A stencil maszk azt mondja meg, az értékek mely bitjei vegyenek részt az összehasonlításban.
A stencilpuffer értékei a következő három eset valamelyikének bekövetkeztekor számítódnak újra:

- A legutóbbi stencil teszt negatív választ adott.
- A stencil teszt sikeres volt, de a mélységi teszt sikertelen.
- A stencil teszt és a mélységi teszt is sikeres volt.

Mindhárom esethez külön meghatározható, a pufferben levő aktuális érték segítségével hogyan álljon elő az új stencilérték. A lehetőségeket a StencilOperation felsorolás tartalmazza:

Érték	**Jelentés**
Increment	Az új stencilérték a puffer értékének növelésével áll elő. A maximum elérésekor a túlcsordulás mértéke lesz az új érték.
IncrementSaturation	Az előbbihez hasonló, de a számítás nem indul újra a minimális értéktől.
Invert	Az új érték a puffer tartalmának bitenkénti negáltja lesz.

Keep	A puffer tartalma nem változik.
Decrement	Az új stencilérték a puffer értékének csökkentésével áll elő. A minimum (0) elérésekor az érték a felső határból lesz meghatározva.
DecrementSaturation	Az előbbihez hasonló, de a számítás nem indul újra a maximális értéktől.
Replace	Az új stencilérték a referencia értéke lesz.
Zero	Az új érték 0 lesz.

A stencil teszthez a következő beállítások érhetők el (a kétoldali tesztelés beállítása ezek mintájára történik, ezért azokat nem soroltam fel):

Név	Értékek	Megjegyzés
StencilEnable	true, false	A stencil teszt engedélyezése, letiltása.
StencilFunction	Compare értékei	Összehasonlítás módja
ReferenceStencil	A $[0, 2^{size-1}]$-ből, egész	$size$ a puffer mérete bitekben.
StencilMask	A $[0, 2^{size-1}]$-ből, egész	
StencilFail	StencilOperation értékei	Az új érték előállítási módja, ha a legutóbbi stencil teszt negatív választ adott.
StencilZBufferFail	StencilOperation értékei	Az új érték előállítási módja, ha a stencil teszt sikeres volt, de a mélységi teszt sikertelen.
StencilPass	StencilOperation értékei	Az új érték előállítási módja, ha a stencil teszt és a mélységi teszt is sikeres volt.
StencilWriteMask	A $[0, 2^{size-1}]$-ből, egész	A ténylegesen pufferbe írt adatokat korlátozhatja.
TwoSidedStencilMode	true, false	A primitívek mindkét oldalára végrehajthatjuk a stencil tesztet. Ehhez a hátsólap-eltávolítást ki kell kapcsolnunk.

ALPHA BLENDING

Egyszerre mindig két pixellel dolgozunk, és ezeket a forrás-, illetve célpixeleket összekombináljuk az alpha-csatornán tárolt értékeiken keresztül, egy adott függvény segítségével. Egész pontosan először a két pixel színét moduláljuk egy meghatározott (és külön-külön választható) módon, majd a két színt átadjuk a kombinációt végző függvénynek paraméterként. Azaz:

$Source = [r_s, g_s, b_s, a_s]$
$Dest = [r_d, g_d, b_d, a_d]$
$SourceModulated = [f(r_s), f(g_s), f(b_s), f(a_s)]$
$DestModulated = [h(r_d), h(g_d), h(b_d), h(a_d)]$
$BlendResult = BlendFunction(SourceModulated, DestModulated)$.

A modulációs faktorok a Blend enumeráció tagjai lehetnek, mégpedig a következők:

Érték	**Jelentés**
BlendFactor	A BlendFactor beállításban megadott szín komponenseit veszük: $[r_b, g_b, b_b, a_b]$.
InvBlendFactor	A BlendFactor beállításban megadott szín inverzének komponensei alkotják a modulációs faktort: $[1 - r_b, 1 - g_b, 1 - b_b, 1 - a_b]$.
SourceColor	A forráspixel színével modulálunk: $[r_s, g_s, b_s, a_s]$
InvSourceColor	A faktor a forráspixel színének inverzét tartalmazza: $[1 - r_s, 1 - g_s, 1 - b_s, 1 - a_s]$
SourceAlpha	A modulációs faktort a forráspixel alpha-csatornájának értékével töltjük fel: $[a_s, a_s, a_s, a_s]$.
InvSourceAlpha	A modulációs faktort a forráspixel alpha-csatornájának invertált értékével töltjük fel: $[1 - a_s, 1 - a_s, 1 - a_s, 1 - a_s]$.
SourceAlphaSat	A modulációs faktor komponenseit a forráspixel alpha-csatornájának értékével, vagy a célpixel alpha-értékének inverzével töltjük fel: $[f, f, f, f]$; $f = min(a_s, 1 - a_d)$.
BothInvSourceAlpha	Csak a forrásra állítható be, és a célpixel modulációs beállítását is felülírja. A faktorok rendre $[1 - a_s, 1 - a_s, 1 - a_s, 1 - a_s]$; $[a_s, a_s, a_s, a_s]$.
BothSourceAlpha	Hasonló az előzőhöz, az értékek fordítottak.
DestinationColor	Mint SourceColor, de a célpixel színértékeivel: $[r_d, g_d, b_d, a_d]$.
InvDestinationColor	Mint InvSourceColor, de a célpixel színértékeivel: $[1 - r_d, 1 - g_d, 1 - b_d, 1 - a_d]$

DestinationAlpha	Mint SourceAlpha, de a célpixel alpha-értékeivel: $[a_d, a_d, a_d, a_d]$.
InvDestinationAlpha	Mint InvSourceAlpha, de a célpixel alpha-értékeivel: $[1 - a_d, 1 - a_d, 1 - a_d, 1 - a_d]$.
One	A modulációs faktornak csupa 1 komponense lesz.
Zero	A modulációs faktornak csupa 0 komponense lesz.

A tényleges kombinációt végző függvényeket a BlendOperation felsorolás adja meg. Ezek leírásában az s, d paraméterek RGBA színértékeket jelölnek (gyakorlatilag négydimenziós vektorokat). Az egyes műveletek a színkomponensekre külön-külön vannak értelmezve.

Érték	**Jelentés**
Add	$Add(s,d) = s + d.$
Max	$Max(s,d) = max(s,d).$
Min	$Min(s,d) = min(s,d).$
RevSubtract	$RevSubtract(s,d) = d - s.$
Subtract	$Subtract(s,d) = s - d.$

Az alpha blending működését befolyásoló beállítások a következők:

Név	**Értékek**	**Megjegyzés**
AlphaBlendEnable	true, false	Az alpha blending engedélyezése, tiltása.
BlendOperation	BlendOperation értékei	A kombináló függvény.
SourceBlend	Blend értékei	A forráspixelek színének modulációjára használt faktor.
DestinationBlend	Blend értékei, kivéve BothSourceAlpha és BothInvSourceAlpha	A célpixelek színének modulációjára használt faktor.
SeparateAlphaBlendEnabled	true, false	A blending kizárólag az alpha csatornára lesz engedélyezett.

DITHERELÉS

A Direct3D eddig a pontig minden pixelműveletet RGBA színekkel végez, amelyben minden komponens legalább 8 biten van letárolva. A megjelenítési felület definíciójában viszont megadható olyan formátum is, amely az egyes színcsatornákhoz kevesebb bit információt tárol. Ezért mielőtt a színinformáció a megjelenítendő felületre íródna, a Direct3D elvégzi a színmélység szükséges csökkentését. Az információveszteségből eredő hibát az algoritmus egyenletesen próbálja meg elosztani a felületen.

MASZKOLÁS

A mostani helyzetben a következő adatok állnak rendelkezésre minden pixelhez: mélység- és stencilérték, szín. Az alkalmazás befolyásolhatja, hogy ezek közül melyek kerüljenek végül a megjelenítési felületre. Ez a szabályozás az ún. írási maszkok (write mask) segítségével történik, amelyek a következők:

- StencilWriteMask: Azt mondja meg, hogy a mélységi, illetve stencilértékek kikerüljenek-e a megjelenítési felülethez tartozó mélységi pufferterületre. A mélységi teszt beállíátsa független ettől a beállítástól, így lehetőség van arra, hogy a mélységi teszt olyan pixeleket is eldobhasson, amelyekre a puffer értéke nem változott. Ha a mélységi, illetve stencil puffer írása engedélyezett, akkor a StencilWriteMask határozza meg, hogy a puffer mely bitjei változnak a tesztek során.
- ColorWriteEnable: A színinformáció különálló színcsatornáit választja ki írásra. Ez a maszk egy vagy több ColorWriteEnable flaget tartalmaz (összesen négy ilyen maszk adható meg, amelyek közül a ColorWriteEnable1 (2, 3) maszkok több megjelenítési felület megléte esetén használatosak. Ha az alpha blending beállítás SourceBlend vagy DestinationBlend értékének Zero vagy One érték van beállítva, akkor a megjelenítési felület színinformációja nem kerül írásra. A maszk értéke a hasonló nevű enumerációban levő értékekből építhető fel:

Érték	**Jelentés**
Red	Vörös színcsatorna.
Green	Zöld színcsatorna.
Blue	Kék színcsatorna.
Alpha	Alpha színcsatorna.
RedGreenBlue	Vörös, zöld és kék színcsatornák.
RedGreenBlueAlpha	Vörös, zöld, kék és alpha színcsatornák.

- MultiSampleMask: Meghatározza, mely minták kerülnek írásra, ha a megjelenítési felülethez többszörös mintavételezés van beállítva. A maszk i. bitje az i. minta írását engedélyezi (az érték egész számként adható meg).

A DIRECTX FÁJLFORMÁTUMA

A háromdimenziós modellek, jelenetek megadásának alapvetően két útja van:

- A háromdimenziós világunkat teljes egészében az API segítségével, a program kódjából definiáljuk. Ekkor a DirectX által biztosított adatszerkezetekre, jól meghatározott API-hívásokra hagyatkozhatunk. Ezt nevezik procedurális modellezésnek is.
- Az általánosabb megoldás, hogy a jelenetet egy erre a célra létrehozott modellezőszoftverrel készítük el, a jelenet minden adatát pedig fájlba mentjük. A modellek egy másik alkalmazással is bármikor megnyithatók egészen addig, amíg a célalkalmazás is támogatja a modellezőprogram által használt formátumot. A jelentősebb modellezőszoftverek gyakorlatilag minden elterjedt, grafikában használatos formátumba képesek exportálni.

A DirectX által támogatott fájlformátum a .X kiterjesztéssel rendelkező DirectX állomány. Ez a fájlformátum úgy lett megalkotva, hogy az összes fontosabb, retained mode által használt objektumot tudja reprezentálni. A következőkben a fájlformátum legfontosabb tulajdonságait emelem ki, majd az állomány felépítését ismertetem.

A DIRECTX ÁLLOMÁNY SZERKEZETE

A Direct3D által a jelenetek leírására használt állomány rendelkezik a következő tulajdonságokkal:

- Sablon-vezérelt. Az állományban tárolt adatok szerkezetét az ún.template-ek (sablonok) definiálják. A sablonok egy része az SDK specifikációjában adott. A template-ek konkrét adatokat tartalmazó előfordulásait adatcsomópontnak (data node) nevezzük.
- Rugalmas. A felhasználó adhat meg saját template-eket az alkalmazás igényeinek megfelelően.
- Független a felhasználási környezettől.
- Hierarchikus. Minden template tartalmazhat újabb template-eket, tetszőleges mélységig.

A DirectX állományt szintaktikailag felépítő kategóriákat `<név>` alakban adtam meg. A `::=` jelsorozat definíciót vezet be. A választási lehetőségek `[opció1 | opció2]` alakban adottak, az opcionális elemeket `(elem)?` jelöli. A `*` tetszőleges számú, a `+` legalább egyszeri előfordulást jelöl; pontosan n darab előfordulást `(n)` fog jelölni. A szintaxis által rögzített kulcsszavakat és kötelező karakteres entitásokat félkövéren szedtem. Az értelemszerű kategóriák (pl. Számjegy) definícióit nem adtam meg.

Minden DirectX állomány a következő részekből áll: fejrész, template-definíciók, adatcsomópontok.

```
<DirectX_állomány> ::= <Header>
                       [ <Template> | <Adatcsomópont> ]+
```

HEADER

Minden DirectX állomány kötelezően egy négy részből álló szabványos fejrésszel kezdődik. Mind a négy rész pontosan négy ASCII karakterből áll. Az első tanúsítja, hogy az állomány tartalma a DirectX állományformátumának felel meg. A következő a fájlformátum verziószámát tartalmazza. A harmadik rész azt határozza meg, hogy az állomány tartalma egyszerű szövegként vagy bináris formában van kódolva, illetve tömörített tartalom esetén a tömörítési algoritmust azonosítja. Végül a negyedik rész a lebegőpontos számok pontosságát írja elő. A fejrész egyes elemeinek lehetséges értékeit a következő táblázat tartalmazza.

Név	**Érték**	**Megjegyzés**
Állomány azonosító	xof	
Verziószám	MMmm	M: major verzió, m: minor verzió. Pl.: 0302
Tartalom kódolása	txt	Egyszerű szöveg, a tartalom tekinthető karakteres adatfolyamnak.
	bin	Bináris kódolás, a tartalom bájtok adatfolyamaként kezelhető.
	bzip	
	gzip	
Lebegőpontos precizitás	32	32 bites (float) pontosság.

	64	64 bites (double) pontosság.

TEMPLATE

A template leginkább a magasszintű nyelvek struktúráihoz hasonlító szerkezet. A fájl adatainak szerkezetét definiálja, egyszersmind az egyes összetartozó adatokat csoportosítja.

Az API által biztosított fájlbeolvasók nem használnak beépített template-készletet, hiszen az alkalmazásnak lehetősége van saját template-eket létrehozni. Ezért a beolvasás előtt mindig regisztrálni kell a beolvasó számára azt a template-készletet, amely az állományban előfordul.

A template felépítése:

```
<Template> ::= template <Azonosító>
               { <GUID>
               <Adattag>*
               [ <Template_név>* | […] ] }
<Azonosító> ::= <Angol_ábécé_betű>
               [ <Angol_ábécé_betű> | <Számjegy> | _ ]*
<GUID> ::=     < <Hexa_számjegy>(8) - <Hexa_számjegy>(4) - <Hexa_számjegy>(4) -
               <Hexa_számjegy>(4) - <Hexa_számjegy>(12) >
<Adattag> ::=  (array [ [<Számjegy>+ | <Azonosító> ] ])? <Típusnév>
               <Azonosító> ;
```

Az azonosító egy karaktersorozat, amely betűvel kezdődik, és betűkkel vagy számjegyekkel, vagy _ karakterekkel folytatódhat. Nem egyezhet meg a DirectX által meghatározott foglalt azonosítókkal, amelyek a DirectX template-ek adattagjainak típusát jelölik. Az API által biztosított beolvasók az azonosítókban a kis-és nagybetűket nem különböztetik meg.

A GUID egy hexadecimális karaktersorozat, amely a Windowsban használatos GUID értékekkel azonos felépítésű. Az API a GUID értéken keresztül éri el a template-eket. Ha megváltozik a template által leírt adatstruktúra, akkor új GUID-ot kell megadni. Ilyen módon a template-ek nem definiálhatják felül és nem fedhetik el egymást. Így biztosítható az, hogy a rendszer minden template-et egyértelműen tud azonosítani ezen érték segítségével.

Minden template nulla vagy több adattagot tartalmaz. A template nemcsak az adattagok számát, típusát határozza meg, hanem ezek sorrendjét is. A template-ek ismeretében tehát a DirectX állomány egyszerűen validálható.

Az adattag egy névből és típusmegjelölésből áll. Minden adattag típusa alaptípus, tömb vagy egy már definiált template megjelöléséből áll. A tömböket a DirectX állományban az `array` kulcsszóval vezetjük be, amit a típusmegjelölés követ, majd az adattag neve és végül a dimenziómegjelölés. Tömbök típusa bármelyik alaptípus vagy már definiált template-típus lehet. A tömb hossza változó és rögzített is lehet. Változó hosszúságú tömb dimenziómegjelölésében egy már definiált adattag neve kell hogy szerepeljen.

A DirectX alaptípusai a következők:

- `BYTE`: előjel nélküli 8 bites egész
- `WORD`: előjel nélküli 16 bites egész
- `DWORD`: előjel nélküli 32 bites egész
- `CHAR`: előjeles 8 bites egész

- `SWORD`: előjeles 16 bites egész
- `INT`: előjeles 32 bites egész
- `FLOAT`: 32 bites lebegőpontos
- `STRING`: ANSI sztring

A template megadhatja azokat a template-eket is, amelyek benne beágyazott template-ként előfordulhatnak. A beágyazott template-ek száma szerint bármelyik template a következő három csoport valamelyikébe sorolható:

- Zárt template: ha a template nem ad meg beágyazott template-ekre vonatkozó listát, akkor a template nem tartalmazhat ilyeneket. A zárt template-nek tartalmaznia kell legalább egy adattagot.
- Szűkített (restricted) template-ről beszélünk ha a template megadja a beágyazott template-ek konkrét listáját.
- Nyílt template esetén bármilyen, előzőleg definiált template szerepelhet beágyazott template-ként. Ekkor a template-lista utolsó eleme a […] karaktersorozat.

ADATCSOMÓPONT

Egy DirectX állomány legnagyobb részében adatcsomópontok vannak felsorolva egymás után; sőt, legtöbbször egymásba is ágyazva. A csomópontok hierarchiája közvetlenül képes reprezentálni a háromdimenziós jelenet objektumainak hierarchiáját is, és képes kiküszöbölni az egyező adatokból eredő redundáns felsorolást. Az adatcsomópont általános leírása:

```
<Adatcsomópont> ::= <Template_név> (<Azonosító>)?
                    { [ <Adatcsomópont> | <Referencia> | <Bináris_adat> ]+ }
<Referencia> ::=  { <Azonosító> }
```

Minden adatcsomópontot annak a template-nek a nevével vezetünk be, amelynek az adott csomópont egy konkrét példánya. Ezt követheti egy egyedi azonosító. A csomópont tartalma { és } karakterek között áll. Ennek a résznek az elején adhatunk meg egy GUID azonosítót is, amely kizárólag ezt a csomópontot (és annak minden alcsomópontját) fogja azonosítani az állomány további részében. Ettől eltekintve a csomópont adatdefiníciós része (a { és a megfelelő } zárójelek közötti rész) tartalmazza a template konkrét adattagjait, majd opcionálisan a template által megengedett beágyazott adatokat.

Az adattagok a template definíciójában megadott sorrendben, egymástól pontosvesszővel elválasztva követik egymást. A tömb elemei egymástól vesszővel vannak elválasztva.

Beágyazott adat alatt a következő három elemet értjük: beágyazott csomópont, bináris adat és referencia.

Az adott csomópont template-je közvetlenül meghatározza, mely template-ek konkrét példányai fordulhatnak elő beágyazott csomópontként. A beágyazott csomópontok felépítése megegyezik a felső szinten levő csomópontok felépítésével.

Bináris adatok akkor is lehetnek egy DirectX állományban, ha maga az állomány egyszerű szövegként van kódolva. Ezt egy speciális token teszi lehetővé, amely elválasztja a szöveges adatokat a bináris adatoktól. Az SDK beépített beolvasó interfészei erre fel vannak készítve.

Egy csomópont alcsomópontja lehet egy olyan referencia is, amely egy korábban megadott csomópontra hivatkozik. A hivatkozás történhet név vagy GUID alapján. A referenciát formailag {és } karakterek között adjuk meg.

Láttuk, hogy a DirectX állomány ugyan sok tekintetben rugalmas, de alapvetően mégis rögzített formát biztosít a háromdimenziós világ adatainak tárolásához. Láttuk azt is, hogy az állományt felépítő

adatcsomópontok szerkezete rögzített a template-ek által. Ezért tehát az egész állomány leírható egy meghatározott entitáshalmaz elemeinek véges sorozataként. Ezeket az entitásokat tokeneknek nevezzük. A token olyan jelsorozat az állományban, amelyet a feldolgozás során nem bontunk további részekre.

Az állomány alapvetően szöveges vagy bináris formában írható le. Most megvizsgáljuk a két lehetőséget az állomány tokenekre bontásának szempontjából:

- Szöveges reprezentáció esetén a tokenek halmaza bizonyos kulcsszavakból és speciális karakterekből áll. A kulcsszavak feldolgozhatóságához szükséges, hogy az ilyen tokeneket szóköz határolja. A speciális karakterek esetén ilyen feltételezésre nincs szükség, hiszen ők is szeparátor funkciót töltenek be. A kulcsszavak feldolgozása során a kis-és nagybetűket nem tekintjük különbözőknek. A DirectX-ben a következő szavak kulcsszavak:

array	float	ulonglong
binary	lpstr	unicode
binary_resource	sdword	void
char	string	word
cstring	sword	
double	template	
dword	uchar	

- A bináris esetben minden tokent egy tizenhat bites érték azonosít. Megintcsak két csoportra oszthatjuk a tokeneket: az egyszerű tokenek mellett vannak olyanok is, amelyeket meghatározott egyéb adatok követnek. Ezeket az adatokat token rekordoknak nevezzük.
  Az egyszerű tokenek jelölik az egyes kulcsszavakat és az elválasztó szerepet betöltő karaktereket. Ez utóbbiak tipikusan olyanok, hogy egy nyitó és egy záró változat is van belőlük (pl. {}, []). Ezt a párosítást a bináris tokenek követik, azaz vannak nyitó és záró tokenek.

Bináris reprezentáció esetén a szeparátorok szerepe csökken, ugyanis ekkor az adatok maximális összefüggő csoportokban jelennek meg. Például ha van egy adatstruktúránk, amely több adattagban egyenként három egész számot tárol, bináris kódolásnál ez a több adattag egyetlen, egész számokból álló listát fog alkotni.

A következőkben a szöveges ábrázolású állományok feldolgozásával fogok foglalkozni, ugyanis ez a forma jelentősen könnyíti a leírt objektumok adatainak vizsgálatát, valamint annak bemutatását, miként juthatunk el ezekből az adatokból a megjelenített alakzatig.

AZ X-SHARP OKTATÓSZOFTVER

Eddig arról volt szó, milyen képességekkel támogatja a DirectX-Direct3D API a háromdimenziós objektumok megjelenítését. Ezeket a következő komplex feladattal szeretném szemléltetni, részben ki is egészíteni: adott a háromdimenziós jelenetet leíró DirectX állomány. Célunk a jelenet megjelenítése és a megjelenítés jelentősebb paramétereinek valós idejű módosítása, a paraméterek hatásainak bemutatása.

A szoftver ezt a feladatot a következő négy alapvető lépésben oldja meg:

1. A szoftverkomponensek inicializálása: Létre kell hozni azokat a DirectX-DirectDraw objektumokat, amelyek az alkalmazás grafikai funkcióinak az alapvető infrastruktúrát szolgáltatják, ezenkívül fel kell készíteni az alkalmazást a grafikai funkciók intenzív használatára.
2. A háromdimenziós világunk objektumainak létrehozása: A DirectX állomány feldolgozása és az adatok transzformációja a Direct3D szabványos retained mode objektumaiba. Esetleg optimalizációkat is kell végeznünk a meglevő adatokon.
3. A jelenet megjelenítése.
4. A megjelenítés paramétereinek módosítása.

A dolgozat hátralevő részében lényegében ezt a négy lépést követem végig. Ennek megfelelően először azt fogom bemutatni, hogyan kell egy C# alkalmazást a DirectX használatára felkészíteni.

A második lépéshez egy saját fájlbeolvasót készítettem, ugyanis ilyen módon könnyebben meg tudom mutatni a fájlokban leírt adatok leképezése során felmerülő részfeladatokat, illetve a Direct3D olyan funkcióit is, amelyek kevésbé dokumentáltak, viszont elég jelentősek ahhoz, hogy egy DirectX (és általában grafikai területen dolgozó) programozó tisztában legyen velük.

A főbb paraméterek valós idejű módosíthatóságát és azok hatásának szemléltetését különösen fontosnak tartottam a fejlesztés során. Ezért a paraméterek az egyes területek szerint rendszerezve, több modulon keresztül jelennek meg. Ezeken kívül a szoftver bemutatja az állomány és a megjelenített objektumok közötti kapcsolatot is, az állományban levő hierarchiát szemléltető modulok segítségével.

A DIRECTX INICIALIZÁLÁSA

Láttuk korábban, hogy a Direct3D grafikai funkciói alapvetően a DirectDraw szolgáltatásaira épülnek. Ezért nem meglepően a Direct3D inicializálása során a DirectDraw alapjait leíró pontban felsorolt lépések közül sok vissza fog köszönni.

Ahhoz, hogy egy C# program a Managed DirectX funkcióit tudja használni, mindenképp először létre kell hoznia egy Device objektumot.

A Device objektumot alapvetően kétféle szerepben kell látnunk. Először is a Device az összekötő kapocs az alkalmazás és az összes grafikus erőforrás között. Ezek az erőforrások két részre oszthatók:

- Lapozási láncok (swap chain-ek): Az alkalmazás által használt hátsó pufferek és az aktuális elülső puffer alkotja. A Device létrehozásakor létrejön egy alapértelmezett swap chain, egy alapértelmezett hátsó pufferrel, amely egyben az alapértelmezett render target is. Az alkalmazás igény szerint újabb offscreen felületeket és render target felületeket definiálhat, a felületeket lekérdezheti, illetve közvetlenül megadhatja, melyik felületre történjen kirajzolás.
- Resource objektumok: minden olyan erőforrás idetartozik, amely nem közvetlenül a megjelenítésért felel. Közös tulajdonságaik: Type, Pool, Format, Usage. Ezeket mind az adott objektum létrehozásakor határozzuk meg, és az értékük nem változtatható ezután. Mindegyikük értékei felsorolásos típusokból kerülnek ki.

A Type az objektum tényleges természetét mondja meg. Értékei a ResourceType enumerációban adottak. Számunkra a fontosak ezek közül az IndexBuffer, a VertexBuffer, a Surface és a Texture.

A Pool tulajdonság az objektum memóriabeli elhelyezését és a Direct3D általi memóriabeli kezelését határozza meg. A lehetséges értékeket a hasonló nevű felsorolásos típus tartalmazza:

Érték	**Jelentés**
Default	Az erőforrás csak a grafikus memóriában létezik. Mivel soha nem készül az adatairól másolat a rendszermemóriában, a Device elvesztésekor az erőforrást újra létre kell hozni.
Managed	Az erőforrás először a rendszermemóriában jön létre és szükség esetén átmásolódik a grafikus memóriába.
SystemMemory	Az erőforrás adatai soha nem kerülnek át a grafikus memóriába.
Scratch	Az előbbihez hasonló, de a hardver képességei nem jelentenek megkötést a létrehozáskor. Azaz olyan paraméterekkel is létrehozható, amelyeket a hardver nem támogat.

A Format tulajdonság az erőforrás által tartalmazott adatok bitszintű felbontását adja meg.

A Usage értékét több flag egyidejű állításával adhatjuk meg. Ezek a beállítások alapvetően azt mondják meg, az alkalmazás hogyan fogja használni az erőforrást (például dinamikusan változik-e az erőforrás).

Más közelítésből a Device felfogható úgy is, mint a korábban leírt megjelenítési futószalag absztrakt reprezentációja a programban. Ugyanis minden, a megjelenítéssel kapcsolatos beállítás ezen az objektumon keresztül érhető el. A megjelenítés egészének beállításai nagyrészt a RenderState adattagba vannak összegyűjtve, kisebb részben pedig egyéb tulajdonságok és metódusok segítségével módosíthatók.

Minden DirectX programnak tartalmaznia kell olyan logikát, amely valamennyi grafikus funkció használata előtt, a program indulásakor fut le, és létrehoz egy olyan Device példányt, amely az alkalmazás igényeinek megfelel. Ennek a következő lépésekből kell állnia:

1. Az elérhető grafikus illesztők felderítése. Annak ellenére, hogy a DirectX egy absztrakciós réteg a hardver fölött, a Device mindig egy adott hardveres illesztőhöz kötődve jön létre. Az inicializáció első lépéseként ezért rögzítenünk kell, melyik grafikus hardverelem erőforrásait fogja az API használni.

   Meg kell jegyeznünk, hogy a hardveres illesztő fogalma nem azonos a grafikus illesztőkártyával. Ma már általános, hogy egyetlen kártya több megjelenítőt is képes meghajtani (általában egyszerre kettőt—ezt nevezik dual head megjelenítésnek). Ekkor a DirectX számára a két csatlakozó két külön illesztőként fog látszani.

   Meg kell jegyezni továbbá, hogy ugyanazon illesztőhöz kapcsolódva több Device is létrehozható. Ekkor viszont mind ugyanazokat az erőforrásokat fogja használni, jelentősen rontva a teljesítményt.

   A grafikus illesztők a DirectX számára egész értékekkel vannak jelölve. A 0 mindig az alapértelmezett illesztőhöz tartozik.

   A DirectX biztosít lehetőséget az alkalmazás számára, hogy megvizsgálja az elérhető összes grafikus illesztőt. Ez a Direct3D Manager osztályán keresztül tehető meg, amelynek Adapters nevű kollekciója tartalmazza az összes illesztőt, és minden illesztőhöz egy AdapterInformation objektumot tárol, amely a következő információkat teszi közzé:

   - Adapter: Az illesztőhöz tartozó indexet (egész számot) adja vissza.
   - CurrentDisplayMode: Az illesztőn érvényben levő megjelenítési mód (felbontás, képfrissítési frekvencia, felületformátum—lényegében a színmélységet határozza meg).
   - Information: az illesztő működését támogató grafikus meghajtóról szolgáltat információkat.

- SupportedDisplayModes: Az illesztő által támogatott megjelenítési módok kollekcióját adja vissza.

2. A grafikus illesztők által támogatott megjelenítési módok felderítése az előbb említett SupportedDisplayModes kollekció segítségével, és az alkalmazás igényeinek megfelelően a következő támogatások meglétének ellenőrzése:

   - Hardveres gyorsítás megléte az adott megjelenítési módhoz
   - Egy adott felületformátum támogatottsága megjelenítési formátumként (a render target fomrátumaként)
   - Egy adott felületformátum támogatottsága a hátsó puffer formátumaként.

   Ehhez a CheckDeviceType metódus használható, melynek általános definíciója a következő:

```
public static System.Boolean CheckDeviceType ( System.Int32 adapter ,
       Microsoft.DirectX.Direct3D.DeviceType checkType ,
       Microsoft.DirectX.Direct3D.Format displayFormat ,
       Microsoft.DirectX.Direct3D.Format backBufferFormat ,
       System.Boolean windowed [, out System.Int32 result] ).
```

   Tehát meg kell adnunk az adapter indexét, a Device (igényelt) típusát—ez gyakorlatilag mindig a DeviceType.Hardver érték lesz. Ezután a tesztelni kívánt formátumok következnek. A `windowed` paraméterrel azt állítjuk be, az alkalmazásunkat ablakos vagy teljes képernyős módban kívánjuk-e futtatni. Ha a `result` paraméterrel is rendelkező változatot használjuk, itt a COM által definiált HRESULT értéknek megfelelő értéket fogunk visszakapni.

   A paraméterek között szereplő Format enumeráció felületeknél használt formátumokat jelöl, és egységes formában írja le, hogy az egyes formátumokban egyetlen pixelhez milyen célt szolgáló információk vannak rendelve, milyen sorrendben, és ezek az információk egyenként hány biten vannak tárolva.

   A teljes képernyős alkalmazások esetén a hátsó puffer és a megjelenítési puffer formátumának lényegében meg kell egyeznie (a hátsó puffer tartalmazhat alpha információkat is). Ablakos üzemmódban viszont lehetőség van arra, hogy a két formátum különböző legyen. Ebben az esetben az a megkötés, hogy a hátsó pufferben levő információ konvertálható legyen a megjelenítési puffer információira. Emlékezzünk vissza ugyanis arra, hogy a megjelenítési futószalag ditherelést is végez közvetlenül a megjelenítési felületre írás előtt. A Direct3D Manager CheckDeviceType metódusa azt is ellenőrzi, hogy ez a lépés elvégezhető lesz-e az adott formátumok esetén. Ha ez nem teljesül, holott a megjelenítés hardveresen nem ütközik akadályokba az adott formátumokkal, a metódus akkor is negatív választ fog adni.

3. Az alkalmazás által használt speciális funkciók támogatásának ellenőrzése. Ennek eszköze a Manager osztály GetDeviceCaps metódusa:

```
public static Caps GetDeviceCaps(int adapter, DeviceType deviceType)
```

   A paraméterek a 2. pontban leírtakhoz hasonlóak. A kívánt funkció támogatásának ellenőrzéséhez a visszakapott Caps struktúra megfelelő tulajdonságának értékét kell megvizsgálnunk.

4. Egy PresentParameters objektum létrehozása, majd paramétereinek beállítása. Elsősorban a hátsó puffer beállításai, az együttműködési mód és a lapozás beállítása lényeges.

5. A Device objektum létrehozása. A konstruktor definíciója:

```
public Device ( System.Int32 adapter , Microsoft.DirectX.Direct3D.DeviceType  deviceType,
   System.Windows.Forms.Control renderWindow , Microsoft.DirectX.Direct3D.CreateFlags
   behaviorFlags, Microsoft.DirectX.Direct3D.PresentParameters presentationParameters )
```

   Egy Managed DirectX alkalmazás rendszerint ezt az alakot fogja használni. A többi konstruktor unmanaged (COM) környezetet feltételez.

   A renderWindow paraméterben a megjelenítéshez használt objektum referenciáját adjuk át. Ez tulajdonképpen a felhasználói felület egy eleme lesz, illetve az egész ablak (mint Form objektum) teljes képernyős mód esetén.

A paraméterek között szerepel a CreateFlags felsorolásos típus is. Az ebben szereplő tulajdonságok közül itt talán elegendő kiemelni a vertex-számításokat befolyásolókat:

- HardverVertexProcessing: A vertexekkel kapcsolatos minden számítást a grafikus hardver fog végezni.
- SoftwareVertexProcessing: A számításokat a processzor fogja végezni. Ezt főleg a hardveres támogatás hiányosságai esetén használják.
- MixedVertexProcessing: A számítások helye szabadon változtatható lesz. Kihasználja a hardveres feldolgozás teljesítményfokozó hatását, de fenntartja a processzor támogatását a hardveresen nem támogatott funkciókhoz.

6. A Device eseménykezelőinek regisztrálása. A leggyakrabban a Managed DirectX programban a Device következő eseményeit kezeljük:

   - DeviceLost

     Az alkalmazás futása során egy Device objektum két állapot valamelyikében van: aktív, illetve passzív. Ez utóbbi esetén az objektumra vonatkozó minden hívás hatástalan, a megjelenítés látszólag leáll. Fontos megjegyezni, hogy a metódusok ekkor is lefutnak és sikeresen térnek vissza, semmilyen kivétel nem váltódik ki. Viszont látható eredmény nem lesz, mivel ebben az állapotban a Device kapcsolata a grafikus erőforrásokkal megszakadt. Ezt az állapotot a szakirodalom „device lost” néven ismeri; tipikus példa rá, amikor teljes képernyős alkalmazás esetén az Alt-Tab billenytűkombináció vagy más esemény hatására a program ablaka inaktív lesz, illetve ablakos alkalmazás esetén, ha az ablakot a felhasználó átméretezi.

     Ez az állapot a grafikus erőforrások (pl. a felületek) újbóli létrehozását teheti szükségessé. Fontos tehát, hogy az alkalmazás erre az átmenetre fel legyen készítve. Ezért elég általános, hogy egy Managed DirectX alkalmazás saját eseménykezelőt ad meg a Device DeviceLost eseményére.

     Az eseménykezelőnek a következőket kell tennie az aktív állapotba való visszatéréshez:

     1. A Device vizsgálata, hogy az aktív állapotba való visszatérés lehetséges-e.
     2. A programnak minden olyan erőforrástól meg kell szabadulnia, amely az adott Device objektumhoz kapcsolódik, és videomemóriát (Default pool-ban levő erőforrásokat) használ. Tipikusan ilyen objektumok a felületek.
     3. Az alkalmazás meghívja a Device objektum Reset metódusát. A Reset az egyetlen metódus, amelynek egy inaktív állapotban levő Device-ra van hatása. Ezért ez az egyetlen módja az aktív állapotba való visszatérésnek. A Reset csak akkor fog sikeresen visszatérni, ha az előző lépésben az alkalmazás minden video-erőforrást felszabadított. A Reset alkalmával tehát lényegében újra el kell végezni a hátsó pufferek, esetleg a mélységi puffer és egyéb erőforrások inicializálását, azaz részben ugyanaz a kód fut le, amely a Device létrehozásánál.

   - DeviceReset

     Az előbbiekben már szóltam az itt végrehajtandó tevékenységekről. Fontos leszögezni, hogy a Device inaktívvá válása és az ehhez kapcsolódó DeviceReset esemény nem elsősorban kivételes események, hibák nyomán következnek be. A folyamat akkor is lezajlik, ha a felhasználó egyszerűen átméretezi az alkalmazás ablakát.

A DirectX hívások közvetlenül nem nyújtanak információt arról, hogy a Device objektum épp aktív vagy inaktív állapotban van; pontosabban azok eredményét ez közvetlenül nem befolyásolja. Viszont az alkalmazás bármikor tesztelheti ezt az állapotot a CheckCooperativeLevel metódussal.

Mindezek után lássuk ezen koncepciók működését az XSharp programban!

Először a Program.cs forrásállományban található Main függvényt vizsgáljuk meg, mivel ez az alkalmazás belépési pontja. A Main metódus egy általános C# programban a következő:

```
        [STAThread]
        static void Main()
        {
                Application.EnableVisualStyles();
                Application.SetCompatibleTextRenderingDefault(false);

                Application.Run(new MainForm());
        }
```

A Managed DirectX-et használó alkalamzásoknak ennél nem sokkal kell több; tulajdonképpen egyetlen dolgot kell elvégeznünk még, mielőtt az alkalmazásunk főablakára kerülne a vezérlés: a DirectX inicializálását, illetve a programunk felkészítését az intenzív grafikus funkciókra. Erre a fő alkalmazás-ablakot definiáló MainForm osztály InitGraphics metódusát fogjuk használni.

```
[STAThread]
static void Main()
{

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);

        using (MainForm mainForm = new MainForm())
        {
                if (!mainForm.InitGraphics())
                {
                    MessageBox.Show("A DirectX inicializálása nem sikerült!",
                        "Error",
                        MessageBoxButtons.OK, MessageBoxIcon.Error);
                    Application.Exit();
                }

                System.Windows.Forms.Application.Idle +=
                    new EventHandler(mainForm.OnApplicationIdle);
                Application.Run(mainForm);
        }
}
```

Amennyiben az InitGraphics sikertelenül tér vissza, kénytelenek vagyunk tudomásul venni, hogy a program futtatása végzetes akadályokba ütközött—ez ugyanis azt jelenti, hogy az alapvető Device objektum és az ehhez kapcsolódó erőforrások létrehozása nem sikerült. Ekkor nem leszünk képesek semmilyen DirectX funkció igénybevételére, ezért az alkalmazás hibaüzenetet ad, majd véget ér.

A fellelhető DirectX bevezetők és példák ezen a ponton általában egyszerűen csak elindítják az alkalmazást, az összes megjelenítési logikát az alkalmazás ablakának OnPaint eseménykezelőjébe sűrítve, és a megjelenítés végén az Invalidate metódust hívják. Ez nem más, mint egy egyszerű végtelen ciklus, ami biztosítja a képkockák kirajzolásának és a megjelenítésnek a folyamatosságát, viszont biztosítja a processzor folyamatos kilencven százalékos terhelését is, ha az alkalmazás ablaka aktív. Ezt elkerülendő, az XSharp indulásakor az alkalmazás Idle eseményére egy kezelőt regisztrálunk. Ilyen módon azt fogjuk elérni, hogy a megjelenítési logika és mindenféle kirajzolás kizárólag akkor fusson le, amikor a program semmilyen más tevékenysége nem foglalja le az erőforrásokat. Mielőtt azonban ezt részletezném, az inicializációnál felvázolt lépéssorozatnak megfelelően a Device objektum létrehozását végző InitGraphics metódusra térek rá.

```
public bool InitGraphics()
{

Format[] validBackBufferFormats = new Format[] {Format.A8B8G8R8,
    Format.X8B8G8R8, Format.X1R5G5B5, Format.R5G6B5, Format.A1R5G5B5,
    Format.X1R5G5B5, Format.A2R10G10B10};
Format[] validDisplayFormats = new Format[] { Format.X8B8G8R8,
    Format.X1R5G5B5, Format.R5G6B5 };
```

```
Caps hardwareCaps;
CreateFlags flags;
bool formatFound = false;
int adapterIndex;

PresentParameters pp = new PresentParameters();
pp.Windowed = true;
pp.SwapEffect = SwapEffect.Discard;
pp.DeviceWindow = renderPane;
pp.AutoDepthStencilFormat = DepthFormat.D24X8;
pp.EnableAutoDepthStencil = true;

for (adapterIndex= 0; adapterIndex < Manager.Adapters.Count; adapterIndex++)
{
     foreach (Format backFormat in validBackBufferFormats)
     {
          foreach (Format displayFormat in validDisplayFormats)
               if (Manager.CheckDeviceType(adapterIndex, DeviceType.Hardware, displayFormat,
                    backFormat, true))
               {
                    pp.BackBufferFormat = backFormat;
                    formatFound = true;
                    break;
               }
          if (formatFound == true)
               break;
     }
     if (formatFound == true)
     break;
}
if (adapterIndex < Manager.Adapters.Count)
{
        if (hardwareCaps.DeviceCaps.SupportsHardwareTransformAndLight)
        {
                flags = CreateFlags.HardwareVertexProcessing;
                if (hardwareCaps.DeviceCaps.SupportsPureDevice)
                        flags |= CreateFlags.PureDevice;
        }
        else
        {
                flags = CreateFlags.SoftwareVertexProcessing;
        }
        device = new Device(adapterIndex, DeviceType.Hardware, renderPane,
                flags, pp);

        [...]
}

return skinSupport;
}
```

A validBackBufferFormats tömb azon megjelenítési formátumokat tartalmazza, amelyek használhatóak a hátsó puffer formátumaként (Referencia: [11]). Ezek közül kerülhet ki az elülső puffer (végső soron a render target) formátuma is. Az elülső puffer viszont nem tárolhat alpha-csatorna értékeket (az A2R10G10B10 kivételével, de ez is csak teljes képernyős alkalmazásoknál elérhető). A validDisplayFormats tömb tehát az alpha csatornát nem tartalmazó formátumokból áll.

Először létrehozunk egy PresentParameters struktúrát, amely sok olyan tulajdonságot tárol, amelyek a megjelenítés alapvető működését határozzák meg, és amelyek közül a legtöbbet a DirectDraw alapjairól szóló részben láttunk. Ezek a beállítások tehát közvetlenül befolyásolják a Direct3D alatt működő DirectDraw funkciók működését. Konkrétan a következő tulajdonságokat állítjuk be:

- Windowed: lényegében ez a DirectDraw-nál látott együttműködési szint

- SwapEffect: azt mondja meg, hogy többszörös pufferelés esetén hogyan történjen a pufferek közötti váltás. Az elérhető lehetőségeket a SwapEffect felsorolásos típus adja meg:

    1. Discard: A lapozási lánc ekkor egy olyan lista lesz, amelyből minden hátső puffer kikerül a tartalmának a megjelenítése után. Ez a módszer a leggyakoribb, mivel nem igényel extra erőforrásokat.

    2. Flip: A lapozási lánc egy cirkuláris lista, amelyek számozása n db hátsó puffer esetén 0,..,(n-1), és 0 jelöli a legrégebben elülső pufferként használt felületet. A megjelenítés (a Device.Present hívása után) a listában forgatás történik, azaz a legutóbbi elülső puffer az (n-1)-edik elemként bekerül a listába, míg a nulladik elem lesz az új elülső puffer.

    3. Copy: Csak akkor használható, ha a lapozási láncban pontosan egy hátsó puffer van. A Present hívás során nem változik meg a hátsó puffer tartalma, tehát nem íródik felül az elülső pufferen levő tartalommal.

- DeviceWindow: A felhasználói felületnek az a komponense, amelynek területén a DirectX a megjelenítést fogja végezni. Ez mindig egy Control objektum. Ha az egész alkalmazás-ablakot a DirectX fogja használni (pl. teljes képernyős módban), akkor a this kulcsszót adjuk itt meg.

- BackBufferFormat: A használt hátsó puffer(ek) formátuma.

- Opcionálisan kérhetjük, hogy a hátsó pufferhez automatikusan jöjjön létre olyan felület is, amely mélységi pufferként fog funkcionálni. Erre szolgál az EnableAutoDepthStencil tulajdonság. Enélkül a DirectX nem fog mélységi tesztelést végezni az objektumok megjelenítése során, tehát feltételezi, hogy az objektumok láthatóság szerint a megfelelő sorrendben vannak.
  A mélységi puffer által használt formátum szintén beállítható az AutoDepthStencilFormat tulajdonságon keresztül. A lehetséges formátumértékekben a számok az adott funkció által felhasznált bitek számát jelentik egy képpontra. A formátum a DepthFormat felsorolásos típus eleme.

Ezt követően, az előző részben definiált mintát követve, a Manager osztály metódusai segítségével végignézzük az elérhető adaptereket, és a

```
Manager.CheckDeviceType(adapterIndex, DeviceType.Hardware, displayFormat, backFormat, true)
```

metódus segítségével kiderítjük, az adott adapteren, ablakos üzemmód mellett milyen elülső, illetve hátsó puffer-formátum párosítás esetén kapunk hardveres támogatást. Ha találunk ilyet, akkor a program a hátsó pufferként használt felületek formátumát a megtalált párnak megfelelően állítja be.

Mivel az alkalmazásnak semmilyen speciális hardverfunkcióra nincs szüksége, a keresést ezután meg is szakítjuk.

Amennyiben az aktuális adapter-index kisebb, mint az összes adapterek száma (tehát a keresés nem indexelt túl az elérhető adapterek tömbjén), akkor az inicializálás egy CreateFlags felsorolásos típusú érték beállításával folytatódik, amely elsősorban a program erőforrás-használatát befolyásolja, illetve a hardver képességeitől függően bizonyos optimaizációt tesz lehetővé. Ezen beállítások közül minimálisan minden alkalmazásnak rendelkeznie kell arról, hogy a vertexekkel kapcsolatos műveletekhez a processzor vagy a grafikus hardver számítási idejét foglalje-e le. Az előbbi a SoftwareVertexProcessing, az utóbbi a HardwareVertexProcessing értékkel kérhető. Jelen alkalmazás ez utóbbit részesíti előnyben a hatékonyság miatt. A legtöbb példaprogram a CPU használatát preferálja, mivel ez nem igényel a grafikus hardvertől támogatást. Ma már viszont gyakorlatilag minden grafikus kártya képes a vertexműveletek támogatására; ennek ellenére a kompatibilitás miatt az XSharp ellenőrzi ezen képesség meglétét (az adapter Caps struktúrájában a DeviceCaps tulajdonságban található, SupportsHardwareTransformAndLight nevű érték ellenőrzésével). Ha a támogatás nincs meg, akkor természetesen a SoftwareVertexProcessing-et állítjuk be.

Ezek után, az inicializáció végső lépéseként létrehozzuk a Device objektumot. A metódus abban az esetben tér vissza sikertelenül, ha minden hátsó puffer-elülső puffer formátumpár esetén a CheckDeviceType hamis értékkel tért vissza.

Az alkalmazás megjelenítési ciklusa kizárólag a program futásának „üresjárati idejében” indul el. Ennek célja az erőforrások, elsősorban a processzoridő használatának elfogadható szinten tartása, eszköze pedig a program belépési pontjában regisztrált OnApplicationIdle eseménykezelő.

Ez két kritériumtól teszi függővé a megjelenítés megkezdését:

- A megjelenítési ablak éppen aktív.
- Egyetlen esemény feldolgozása sincs folyamatban.

Ez utóbbi eldöntésére a Windows API egy függvényét használom, amely az aktuálisan feldolgozás alatt álló üzenetet adja vissza, illetve hamis értéket, ha az üzenetsor üres.

Mindkét feltétel teljesülése esetén a Render metódus fog lefutni:

```
internal void Render()
{
    if (env.RenderState == RenderState.DeviceLost ||
        env.RenderState == RenderState.Paused)
    {
        System.Threading.Thread.Sleep(100);
    }
    if (env.RenderState == RenderState.Inactive)
    {
        System.Threading.Thread.Sleep(20);
    }
    if (env.RenderState == RenderState.DeviceLost)
    {
        int result;
        if (!device.CheckCooperativeLevel(out result))
        {
                if (result == (int)ResultCode.DeviceLost)
                {
                        System.Threading.Thread.Sleep(50);
                        return;
                }
                try
                {
                        device.Reset(device.PresentationParameters);
                }
                catch (DeviceLostException)
                {
                        System.Threading.Thread.Sleep(50);
                        return;
                }
        }
        env.RenderState = RenderState.Normal;
    }

    float elapsedTime = env.AppTimer.GetElapsedTime();
    OnFrameMove(elapsedTime);
    if (env.RenderState == RenderState.Normal)
    {
        try
        {
                env.RenderScene();
        }
        catch (DeviceLostException)
        {
                env.RenderState = RenderState.DeviceLost;
        }
        catch (DriverInternalErrorException)
        {
                env.RenderState = RenderState.DeviceLost;
        }
    }
}
```

Alapvetően a metódus egy elágazást definiál a RenderState objektumban nyilvántartott megjelenítési állapot szerint. Ez a RenderState felsorolásos típussal azonosított állapot a következők egyike lehet:

- Normal: Ez az alapértelmezett állapot; azt jelzi, hogy a megjelenítés nem ütközik semmilyen akadályba.
- Inactive: Az alkalmazás ablaka nem aktív.
- Paused: A megjelenítési ciklus szünetel. Ennek két oka lehet:
  1. A felhasználó másik fület aktivált az alkalmazás főablakában.
  2. Egyéb intenzív tevékenység indult el, például a felhasználó egy másik állomány megnyitását kezdeményezte.
- DeviceLost: A megjelenítés nem történhet meg, mert a Device jelenleg inaktív állapotban van. Ezt az állapotot a DeviceLost esemény váltja ki.

Ha az aktuális állapot Inactive vagy Paused, akkor felesleges további erőforrást fordítani a kirajzolásra; ezért más alkalmazások számára szabadítjuk fel a processzoridőt.

A DeviceLost állapot kezelése a fentebb felvázolt séma szerint történik. Mivel az alkalmazásnak nincsenek külön felszabadítandó erőforrásai, ezért rögtön a CheckCooperativeLevel metódust hívjuk meg. Ha ez sikertelenül tér vissza, a Microsoft ajánlása szerint az alkalmazás várakozik, amíg a Device újraaktiválása meg nem kísérelhető. Tehát ebben az ágban is más alkalmazásoknak adjuk át az erőforrásokat. Egyébként megtörténik a Reset metódus hívása; ennek eredményétől függően a megjelenítési állapot marad DeviceLost, illetve sikeres aktiválás után Normal.

Ezek után megmérjük az előző megjelenítési ciklus óta eltelt időt. Ez egyrészt az animációk pontos megjelenítése, másrészt az egérrel történő finom mozgatások helyes kezelése miatt fontos. Ez utóbbiakat az OnFrameMove metódus fogja lekezelni.

A lényegi megjelenítés csak akkor kezdődik el, ha ezen a ponton az állapot még mindig Normal. Ha a megjelenítés során hiba lépett fel, az állapot DeviceLost lesz.

A HÁROMDIMENZIÓS VILÁG OBJEKTUMAINAK LEKÉPEZÉSE

Ebben a fejezetben azzal foglalkozok, hogyan lehet a DirectX állományban tárolt adatokat a Direct3D retained mode objektumaiba leképezni. Leképezés alatt a template-ben tárolt adatok és az objektumok adattagjainak egymásnak megfeleltetését értem.

Erre a feladatra a Direct3D rendelkezik eszközökkel. A háromdimenziós objektumok beolvasására DirectX állományból két metóduscsoport tagjai használhatók.

Az első csoport tagjai Mesh objektumot adnak vissza. Ha az állomány több objektumot tartalmaz, az API az összes adatot egyetlen Mesh-be koncentrálja. Amennyiben az adatok kép-hierarchiában vannak megadva, a metódusok az ebben definiált transzformációkat is figyelembe veszik. A csoport tagjai a következők:

- FromFile
- FromX
- FromStream

A második csoportba olyan metódusok tartoznak, amelyek az animációra vonatkozó adatokat is figyelembe veszik:

- LoadHierarchy
- LoadHierarchyFromX

Ezek egy AnimationRootFrame struktúrát adnak vissza, amely igazából két fontos információt hordoz: az objektumok hierarchiáját és adatait tartalmazó Frame, és az animáció adatait tartalmazó AnimationController objektumot. A csoport metódusainak használata általában a következő módon történik:

1. Az AllocateHierarchy osztály származtatása, a CreateFrame és CreateMeshContainer metódusainak felüldefiniálása. Tehát meghatározzuk, a program hogyan képezze le a képkockák információit a Frame, illetve az objektumok adatait a Mesh objektumokra.

2. Amennyiben az alkalmazás saját maga által definiált adatleképezéseket is használ (tehát a DirectX állományban vannak saját template-ek is), azt a program a LoadUserData metódusain keresztül tudja kezelni. Ezt az osztályt származtatva felül kell definiálnunk a LoadFrameChildData, LoadMeshChildData, és LoadTopLevelData metódusokat (annak megfelelően, hogy a speciális adatok hol helyezkednek el a hierarchiában).

3. Meghívjuk a LoadHierarchy(FromX) metódust a megfelelő AllocateHierarchy és LoadUserData példányokkal. A beolvasás során az API ezek megfelelő metódusait fogja hívni.

Meg kell jegyezni, hogy bár ezek az eszközök bizonyos körülmények között jól használhatóak, a tapasztalat azt mutatja, hogy a hibatűrő képességük nem túl nagy és ezért rugalmatlanok, ez pedig eléggé nehezíti az általános célú alkalmazásukat.

Az XSharp program ezek helyett saját maga végzi el a DirectX állomány feldolgozását. Ezt a döntést az indokolja, hogy az alkalmazás így gyakorlatilag tetszőleges tevékenységet végezhet el már a beolvasás közben; például alkalom nyílik a felfedezett hibák helyéről és természetéről információt adni a felhasználó felé. Másrészt ilyen módon az adatok leképezése és az adatok esetleges optimalizálása, a hiányzó adatok pótlása egyetlen fázisban elvégezhető.

A DirectX állományok beolvasását a programban az XReader osztály végzi. Ez a System.IO.StreamReader osztály leszármazottja.

A beolvasó osztály hét fő funkciót tölt be. Ezekhez a funkciókhoz hét különböző osztály kötődik:

1. Az XReader beolvasást indító metódusa a Parse metódus, amelynek visszatérési értéke egy XAnimRoot példány, amely az API LoadHierarchy metódusainál látott AnimationRootFrame-hez hasonlóan az objektumhierarchia, illetve az animációk adatait adja vissza.

2. Elképzelhető, hogy a DirectX állomány egy tetszőleges szövegszerkesztővel megnyitva helytelenül jelenik meg, tördelés nélkül. Ezért a beolvasás során az állomány, mint szöveges tartalom tördelését is elvégezzük, és az eredményt eltároljuk. Később a felhasználó bármikor meg tudja tekinteni ezt az áttördelt változatot. A beolvasó ehhez egy egyszerű System.Text.StringBuilder objektumot használ; amelynek tartalmát egy HTML-lapként jelenítjük meg.

3. Láttuk, hogy a DirectX állománya képes közvetlenül definiálni az objektumok hierarchiáját a template-ek hierarchiáján keresztül. Ezt a program arra is kihasználja, hogy a felhasználó számára láthatóvá tegye a jelenet felépítését. A megjelenített háromdimenziós világ objektumait egy fastruktúra elemeiként reprezentáljuk. Minden elemnek egy XElement példány felel meg, amely a beépített System.Windows.Forms.TreeNode leszármazottja. A beolvasást végző metódus a hierarchiát tartalmazó felső szintű XElement példányt is visszaadja.

4. Mivel a jelenetet a legritkább esetben adjuk meg procedurális modellezés segítségével, fontos, hogy a program részletesebb információt is adjon a DirectX állományban található adatokkal kapcsolatban. Ezt szolgálja a HtmlCodeGenerator osztály. Az alkalmazás ennek segítségével olyan HTML lapot tud készíteni, ahol nem elsősorban az egyes template-ek hierarchiája szemléltethető, hanem azok adatai és az adatok szerepe, az esetleges egyéb elérhető információkkal együtt:

    - Az adatok rendeltetésének megnevezése. A konkrét adatcsomópontok felépítése nem mond semmit a csomópontban tárolt adatok céljáról, tehát arról, hogy milyen adatokról van szó; azt egyedül a template definíciója tartalmazza. Ezért a diagramszerű nézet feltünteti az egyes adatok rendeltetését (pl. vertexkoordináták vagy textúra-koordináták)

    - Az egyes összetartozó adatokat csoportokba rendezi, például egyetlen csúcspont három koordinátája egymás mellé rendezve jelenik meg.

- Az állományban levő adatcsomópontok nem mind függetlenek egymástól. Például minden olyan template-nek, amely csúcspontokhoz tartozó adatokat sorol fel, minden egyes csúcsponthoz kell adatot megadnia. Az ilyen feltételeket a diagram-nézet szintén megjeleníti, a szükséges és aktuális adatmennyiség összehasonlításán keresztül.
- Egyéb, template-specifikus információk, a programnak a template feldolgozása közben végzett tevékenységével kapcsolatban.

5. A beolvasó nyilvántartja a feldolgozás során talált anyag-információkat. A Mesh template-ben adott anyaginformációk nagyrészt a megvilágítási tulajdonságokat befolyásolják vertex-szinten; végső soron így az objektum színét határozzák meg. Az API beépített beolvasó metódusaihoz hasonlóan a program az anyaginformációkat az objektum adataitól függetlenül is elérhetővé teszi. A későbbiekben az anyagtulajdonságokat bemutató modul fogja ezt az információt felhasználni.
6. Az anyagok adatai mellett a beolvasó az animációk adatait is külön nyillvántartja, szintén a megfelelő modul számára. Ezt a funkciót az AnimationContainer osztály látja el.
7. A beolvasás során keletkezett hibákról a program a Logger osztály segítségével listát készít.

A beolvasó működése azokon az elveken alapul, amiket korábban a szöveges DirectX állomány feldolgozásáról leírtam. Tehát a feldolgozás alapjai a kulcsszavak és az elhatároló jelek.

A kulcsszavak a programban nem a DirectX szabványos kulcsszavai; ugyanis jelen változatában a beolvasó nem támogatja felhasználói template-ek feldolgozását—a gyakorlati felhasználói munkafolyamatban (tehát a jelenet elkészítése valamely erre a célra készült szoftverrel, majd DirectX formátumba történő konvertálása, és az így kapott állományok feldolgozása a programmal) gyakorlatilag ezekre nincs is igény. Kulcsszó alatt tehát a beolvasó a DirectX szabványban előre definiált template-ek egy csoportját érti. Ez a csoport a leggyakrabban előforduló template-ekből áll:

Animation	Material	SkinWeights
AnimationKey	Mesh	TextureFilename
AnimationSet	MeshNormals	XSkinMeshHeader
Frame	MeshMaterialList	
FrameTransformMatrix	MeshTextureCoords	

A DirectX állomány felépítésének megfelelően a Parse metódus először a fejlécet keresi meg. Ha az állományban nincs fejléc, a program figyelmeztetést ad, és megpróbálja az állományt szöveges állományként feldolgozni. Amennyiben ez nem sikerül, a felhasználó erről hibaüzenetet fog kapni.

Az esetleges template-definíciókat a program figyelmen kívül hagyja.

Az adatcsomópontok feldolgozása a következő séma szerint történik:

```
while (true)
{
    if (megtaláltuk az aktuális template-et lezáró elhatárolót ( } ))
    {
        A bemeneti adatfolyamból az illeszkedő rész törlése;
        break;
        //Végtelen ciklus vége
    }

    if (az aktuális helyen az adatfolyamban template-nevet találtunk)
    {
        A bemeneti adatfolyamból az illeszkedő rész törlése;
        Az állomány-hierarchiát szemléltető fastruktúra bővítése az adott
                template-nek megfelelően;
        A fastruktúra gyökere (az az elem, amely alá a bővítés történik) az
                újonnan felvett elem;
        A tördelt tartalmat tároló StringBuilder bővítése;
```

```
        A diagram-nézetet generáló HtmlCodeGenerator segítségével új HTML-blokk
                nyitása;
        Elágazás a template neve szerint
        {
                [...]
                //Az egyes tartalmazott template-eknek megfelelő tevékenységek
        }
        A tördelt tartalmat tároló StringBuilder-ben az aktuális blokk lezárása;
        A diagram-nézetet generáló HtmlCodeGenerator segítségével az utolsó HTML-blokk lezárása;
        A fastruktúra gyökere (az az elem, amely alá a bővítés történik) az aktuális elem szülője

    }
    else
    {
        A bemeneti adatfolyam következő sorának beolvasása feldolgozásra;
        if (A bemenő adatfolyam nem adott vissza több sort)
                break;
                //Végtelen ciklus vége
    }
}
```

A megengedett kulcsszavak halmaza kontextusfüggő, tehát nem explicit módon adott (sehol sincsenek az elemei közvetlenül felsorolva)—a különböző template-ek különböző tartalmazott template-eket definiálnak. Ezek közül a program csak a kötelezőket (a template-definícióban név szerint felsoroltakat) kezeli a név alapján történő elágaztatásban. Ha a program az adatfolyamon olyan, template-ként azonosítható entitást talál, amely az aktuális kontextusban nem támogatott, akkor automatikusan az adott blokkot lezáró } karaktertől folytatja a feldolgozást.

Látható, hogy a program a feldolgozást—a háttérben levő StreamReader metódusait használva—soronként végzi. Ennek ellenére nem várjuk el, hogy az eredeti állományban az adatok bármilyen újsor-szekvenciával el legyenek választva; más szóval a program nem tekinti az újsor-karaktert vagy annak megfelelő bármilyen karakter-kombinációt elválasztónak. Az elválasztójelek a DirectX specifikációt követve a pontosvessző, a vessző, és a { } zárójelek.

A specifikációnak megfelelően feltételezzük azt is, hogy a kulcsszavak és egyéb opcionális azonosítók között az elválasztó legalább egy üreshely-karakter (szóköz, tabulátor, vagy újsor).

A program az elválasztók meglétét ellenőrzi; azok hiánya esetén hibaüzenetet generál. Ez a szintaktikai ellenőrzés nem feltétlenül befolyásolja a beolvasás kimenetelét. Például a pontosvessző hiánya nem feltétlenül teszi sikertelenné a beolvasást.

Az adatok megfeleltetése a retained mode objektumainak

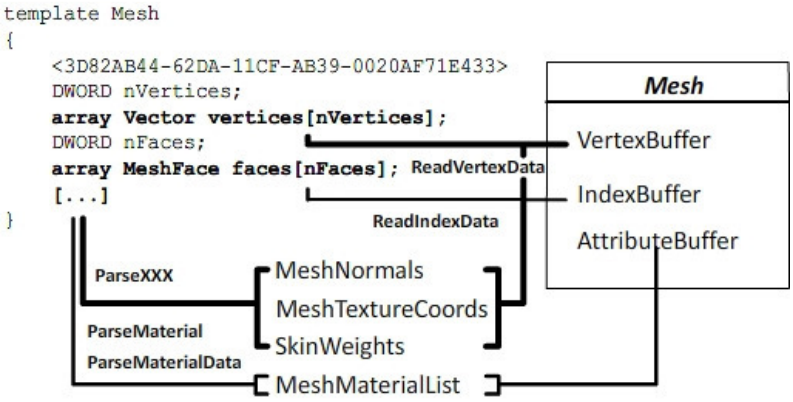
A retained mode API osztályai nagyrészt követik az előre definiált template-ek felépítését, ami jelentősen megkönnyíti az adatok átvitelét. Most a program által használt retained mode objektumokat és azokra történő adatleképezési lépéseket veszem sorra, a nekik megfelelő template-ekkel együtt.

Mesh

A 3. ábrán látható, hogy a Mesh template a következő adatokból áll:

1. A csúcspontok száma.
2. A vertexkoordináták, csúcspontonként csoportosítva.
3. Az indexadatok száma.
4. Indexbejegyzések felsorolása.
5. Opcionálisan tartalmazott template-ek.

A feldolgozás során ezeket ebben a sorrendben kell beolvasnunk.



3. ábra: A Mesh template feldolgozása

A Mesh az adatokat három adatcsoportban helyezi el: az elsőben az adott objektum összes csúcspontjának koordinátái és minden vertex-specifikus adatot helyezzük el, a másodikban azt tároljuk, hogyan kell a vertexekből az alakzat lapjait felépíteni; a harmadik pedig azt mondja meg, milyen csoportokra oszthatók az alakzat lapjai az anyagtulajdonságok és egyéb jellemzők alapján.

A Mesh osztály ún. indexelt alakzatot képes reprezentálni. A vertexek adatait az ún. vertex puffer tárolja. A puffer az egyes vertexekhez tartozó adatokat egymás után sorolja fel—lényegében ez egy adatfolyam. Hogy egy vertexnek milyen adatai vannak, az a Device létrehozásánál rögzített vertexformátumtól függ. A lapok felépítésére vonatkozó információ az index pufferben van. Ennek a tartalma lényegében rendezett számhármasokból áll. Minden számhármas egy háromszöglapra vonatkozik; tagjai az egyes csúcspontok sorszámát adják meg a vertex pufferben. A vertex adatokat a programban a ReadVertexData, az indexfelsorolást a ReadIndexData metódus olvassa be.

VERTEXADATOK FELDOLGOZÁSA A READVERTEXDATA METÓDUSSAL

A ReadVertexData metódus a csúcspontokat olyan struktúrában helyezi el, amely a kötelezően rendelkezésre álló adatok (pl. koordináták) mellett az opcionális adatok számára is biztosít helyet. Ez a BlendedVertex struktúra a következő adatokat rendeli az egyes vertexekhez:

- Koordináták (objektumtérben)
- Normális koordinátái az adott csúcspontban
- Textúra-koordináták
- Ha az objektum bőrözést használ (skinning), akkor maximálisan négy darab, [0, 1]-be eső súlyérték
- Maximálisan négy egész szám, amelyek megmondják, a fenti súlyok milyen mátrix-indexhez tartoznak az ún. mátrix-palettából.

Ezen adatok közül nem mindegyik kerül ténylegesen a végleges Mesh objektumba. Ha nincs bőrözési információ, az utolsó két adatcsoport nem szerepel majd a Mesh csúcsainak adatai között.

A tartalmazott template-ek nagy részének adataihoz tehát nincs külön adattag fenntartva a Mesh osztályban; minden vertex-specifikus információ a csúcspontot reprezentáló struktúrában kell hogy helyet kapjon. Az API-nak mégis valahogy tudnia kell, hogyan épül fel a vertex, hiszen már láttuk, hogy a DirectX minden objektumot végső soron adatfolyamokra képez le; ehhez nélkülözhetetlen az adatfolyamon belüli adatok szerkezetének ismerete.

A Direct3D sok előre definiált vertexstruktúrát tartalmaz; az alkalmazásnak pedig lehetősége van saját struktúrát megadnia. Ez a rugalmas vertexformátum (flexible vertex format, vagy FVF) teszi igazán rugalmassá az egész API-t. A Managed DirectX esetén a VertexElement osztály segítségével adhatunk meg saját vertexformátumot; egy vertexformátumot VertexElement objektumok tömbje ír le. A program két ilyen tömböt definiál; bőrözött illetve nem bőrözött objektumokhoz.

INDEXADATOK FELDOLGOZÁSA A READINDEXDATA METÓDUSSAL

Az egyes lapok felépítésére semmilyen megkötés nincs. Tehát semmi sem garantálja azt, hogy minden lap háromszöglap lesz. Ezért minden lap leírása az indexadatok felsorolásában az adott lapon levő csúcspontok számával kezdődik, és ezt követi az érintett csúcsindexek felsorolása.

Számunkra ez azt jelenti, hogy amennyiben szükséges, már a feldolgozás alatt el kell végeznünk a lapok háromszögesítését—a DirectX csak háromszöglapokból álló Mesh-t tud kezelni. A ReadIndexData metódus a következő egyszerű tesszellációt alkalmazza:

```
[...] //A csúcspontok számának beolvasása (size)

if (size > 3)
{
    Nyilvántartásba vesszük, hanyadik lapnál történt háromszögesítés, és hány háromszögre
osztottunk.
}
A lap csúcsindexeinek beolvasása egy egész tömbbe (tmp);
int a = 1;
while (a < size - 1)
{
        Írás az indexlistába: tmp[0];
        Írás az indexlistába: tmp[a];
        Írás az indexlistába: tmp[++a];
}
```

A keletkezett háromszögeknek lesz közös csúcsuk, és az indexléptetés biztosítja, hogy az új háromszögek körüljárási iránya az eredeti lap körüljárási irányával egyezzen meg.

A metódus az összes keletkezett indexet folyamatosan felsoroló listát adja vissza; a Mesh objektum ilyen formátumban fogadja el az indexadatokat.

A ReadIndexData kimenő paraméterként viszaadja az olyan lapok sorszámát, amelyeknél tesszelláció történt, a keletkezett lapok számával együtt. Ez az információ kulcs-érték párokként lesz nyilvántartva, és gyakorlatilag minden tartalmazott opcionális template feldolgozásakor fel fogjuk használni, ugyanis a háromszögesítés során plusz lapok keletkeztek, amelyek nem szerepelnek az ottani felsorolásokban.

Ezzel feldolgoztuk a Mesh template kötelező adatait.

OPCIONÁLIS TEMPLATE-EK FELDOLGOZÁSA

A tartalmazott template-ek a következők lehetnek:

- MeshNormals: A megvilágításhoz minden csúcspontban szükségünk van a normális koordinátáira. A csúcspontban vett normális természetesen egy absztrakció, ugyanis a lapok közvetve, csúcspontjaikon keresztül vannak tárolva. Erre az adatra akkor is szükség van, ha maga az állomány nem adja meg. Ebben az esetben a normálisok generáltathatók az API-n keresztül is; a program viszont saját metódust használ ennek elvégzésére.
- MeshTextureCoords: Textúrázáshoz szükséges koordináta-pár minden egyes csúcsponthoz. Ezek a koordináták a kétdimenziós textúrára vonatkoznak, és annak egy pontját jelölik ki egy $[x, y]$ párral $(x, y \epsilon [0, 1])$.
- MeshMaterialList: Megadja, hány anyagot használ az adott objektum, és melyik laphoz melyik anyag tulajdonságait kell rendelni. Az anyagok ebben tartalmazott template-ként, vagy egy már definiált Material template-re történő hivatkozással vannak megadva.
- SkinWeights: Lényegében azt tartalmazza, hogy egy mátrix az ún. mátrix-palettából mely csúcspontokra van hatással, és mekkora súllyal.
- XSkinMeshHeader: Három információt hordoz a bőrözéssel kapcsolatban:

1. nMaxSkinWeightsPerVertex: Az egyetlen csúcsponttal kapcsolatban levő mátrixok maximális száma (a DirectX hivatalosan legfeljebb négy mátrixot támogat vertexenként).
2. nMaxSkinWeightsPerFace: Egy lap három csúcspontját befolyásoló mátrixok maximális száma.
3. nBones: Lényegében azt mondja meg, összesen hány mátrix vesz részt a bőrözésben.

Ezek az információk a program működését nem befolyásolják, viszont használhatók ellenőrzésre a beolvasás során (pl. a SkinWeights példányok száma és nBones egyenlő-e).

A Mesh adatcsomópontjainak feldolgozása után következik a szükséges adatok generálása (normálisok), majd a DirectX Mesh objektumának létrehozása.

VERTEX-NORMÁLISOK GENERÁLÁSA

A normálisok generáltatására a Mesh osztály GenerateNormals metódusa is használható. A program egy hasonló nevű, saját metódust használ erre a célra:

```
private void GenerateNormals(ref int[] inds, ref BlendedVertex[] verts)
{
    Vector3 v1, v2, v3, fnorm, vnorm;
    int[] numAdjacentFaces = new int[verts.Length];

    for (int a = 0; a < inds.Length; a+=3)
    {
        v1 = verts[inds[a]].Position;
        v2 = verts[inds[a + 1]].Position;
        v3 = verts[inds[a + 2]].Position;
        fnorm = Vector3.Normalize(Vector3.Cross(v3 - v2, v1 - v2));

        verts[inds[a]].Normal += fnorm;
        verts[inds[a + 1]].Normal += fnorm;
        verts[inds[a + 2]].Normal += fnorm;
        numAdjacentFaces[inds[a]]++;
        numAdjacentFaces[inds[a + 1]]++;
        numAdjacentFaces[inds[a + 2]]++;
    }
    for (int a = 0; a < verts.Length; a++)
    {
        verts[a].Normal = Vector3.Normalize(Vector3.Scale(verts[a].Normal,
        1f / numAdjacentFaces[a]));
    }
}
```

Ez lényegében nem más, mint a Gouraud által javasolt egyszerű algoritmus. A metódus paraméterként megkapja az indexek és a vertexek listáját. Az indexek listája megmondja, hogy egy háromszöglapnak melyek a csúcspontjai. Minden vertexre összeadjuk azon lapok normálisait, amelyeknek a csúcspont közös pontja, és nyilvántartjuk, hogy az adott vertexben hány lap találkozik (a numAdjacentFaces tömbben). A kapott normálisokat skálázzuk aszerint, hogy hány lap befolyásolta az értéküket, majd normalizálunk. A szakirodalomban ez a módszer MWE (Mean Weighted Equally) néven is ismert.

AZ ATTRIBÚTUMPUFFER

Egyetlen alakzat több anyagot is használhat. Az anyaghozzárandelést a lapok szintjén a MeshMaterialList template-ből tudjuk. A kirajzolás hatékonysága érdekében egy osztályozást kell végeznünk a lapokra (végeredményben azok csúcspontjaira), az azokhoz rendelt anyagok alapján (ugyanis a hozzárendelés megadása nem feltétlenül—sőt, általában nem—az anyagok sorszáma szerint csoportosítva történik). Ezt az osztályozást a Mesh objektum az ún. attribútum táblában tárolja. Ez a tábla az egyes anyagok sorszámaihoz a lapok összefüggő sorozatát rendeli; megadja ezen sorozatok kezdő-, illetve végpontját (az indexpufferből vett indexekkel). Ez azt jelenti, hogy az objektum lapjait egyező anyaghozzárendelések alapján rendezni kell, ennek megfelelően pedig

frissíteni az indexpuffer tartalmát. Szerencsére ezt a nem triviális feladatot az API képes elvégezni; a Mesh osztály OptimizeInPlace metódusa számos szempont szerint képes az adatokat megfelelő formába hozni. Az attribútum tábla generálásához az OptimizeVertexCache paramétert kell a metódusnak megadnunk.

Mindezek után a Mesh több lapcsoportból fog állni, és az egyes anyagsorszámok szerinti csoportok a DrawSubset metódussal rajzoltathatók ki, mint ahogy az a programban is történik.

Megjegyezzük, hogy a csoportokra osztás másként történik például akkor, ha bőrözött alakzatról (skinned mesh) van szó. Ekkor a generálás során nem pusztán a lapokhoz rendelt anyagokat vesszük figyelembe; egy osztályt az anyag indexe mellett mátrixindexek egy közös csoportja azonosít.

A MESH OBJEKTUM LÉTREHOZÁSA

Kétféle Mesh objektumot hozhatunk létre aszerint, hogy az állományban találtunk-e bőrözési információkat vagy nem. Utóbbi esetben több adatot rendeltünk minden vertexhez, tehát más vertexformátumot kell megadnunk a létrehozásnál. A lehetőségek a következők:

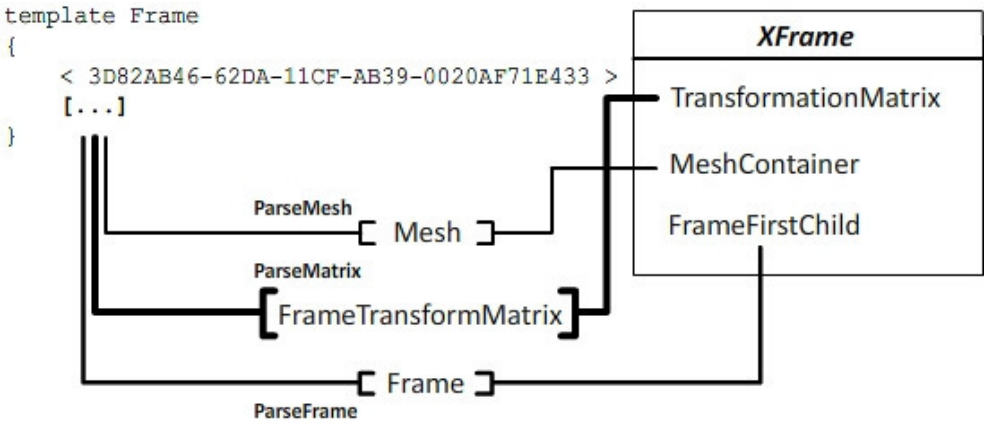
```
//Bőrözéssel
mesh = new Mesh(nTris, nVerts, MeshFlags.Managed | MeshFlags.Use32Bit,
     ElementDescription.BlendedElements, renderDevice);
//Bőrözés nélkül
mesh = new Mesh(nTris, nVerts, MeshFlags.Managed | MeshFlags.Use32Bit,
     ElementDescription.PlainElements, renderDevice);
```

Mindkét esetben közös paraméterek sorrendben:

- Lapok száma
- Csúcspontok száma
- A MeshFlags beállítások elsősorban az objektum memóriahasználatát befolyásolják. Jelen esetben azt írjuk elő, hogy az objektum összes erőforrása a .Net által kezelt memóriaterületen legyen, illetve az indexpufferben levő indexek méretét 32 bitben határozzuk meg. Utóbbira azért van szükség, mert alapértelmezés szerint az indexek 16 bitesek, viszont az alkalmazásban használt indexlista 32 bites egész értékeket tartalmaz.
- A konstruktor negyedik paraméterében a vertexek felépítését kell megadnunk. Jelen esetben ezt VertexElement tömbök segítségével tesszük meg.
- A Device objektum; jelen esetben ez ugyanaz a Device, amelyet az inicializálás során létrehoztunk.

FRAME

Ahogy a 4. ábráról is látható, hogy ez egy nyílt template. Az API beolvasó metódusai jelenleg három tartalmazott template-et ismernek fel a Frame példányaiban: Mesh, Frame, FrameTransformMatrix. Ez utóbbi adja meg a transzformációkat az összes Mesh példány globális terében. A transzformációk az aktuális Frame összes tartalmazott template-jére (végső soron minden alsóbb szinten levő Mesh példányra) hatással vannak. Ezek a mátrixok alkotják bőrözés esetén a mátrix paletta elemeit is.



4. ábra: A Frame template feldolgozása

A lehetséges tartalmazott template-ek feldolgozása a következőképpen történik:

- FrameTransformMatrix: A DirectX Frame osztálya tartalmaz egy TransformationMatrix adattagot, amely egy 4 × 4-es mátrix. A beolvasó osztály ParseMatrix metódusa egy lebegőpontos tömb elemeibe olvassa be a template-ben levő tizenhat számot, amelyekkel a megfelelő DirectX Matrix struktúrát tölti fel.
- Mesh: Az adatok a Frame MeshData struktúrájának Mesh adattagjába kerülnek.
- Frame: a beolvasó ParseFrame metódusa bővíti a képkockák hierarchiáját, majd rekurzívan saját magát hívja.

AZ ANIMATIONSET ÉS A KULCSKOCKA ALAPÚ ANIMÁCIÓ

A harmadik, felső szinten is előforduló template az animációkat írja le. Ez a template nem közvetlenül az animációs adatokat tartalmazza. Minden AnimationSet felfogható úgy, mint egy önálló mozgássorozat, egy külön „jelenet”, amelyben az egyes Animation template-ek különböző objektumok bizonyos idő alatt bekövetkező transzformációit írják le. A mozgási adatok meghatározott időpillanatokban vannak megadva. Ezekben a pillanatokban igazából az adott alakzaton bekövetkezett transzformációkat tároljuk le. Az ilyen időpillanat-állapot párokat nevezzük kulcskockáknak.

Az Animation úgyanúgy nyílt template, mint a Frame. Az API által meghonosított gyakorlat szerint a mozgás leírásához a template-en belül szerepelnie kell egy hivatkozásnak arra a Frame template-re, amelyben a mozgó alakzat leírása található. A tényleges kulcskocka adatokat az AnimationKey template tartalmazza:

```
template AnimationKey
{
    < 10DD46A8-775B-11CF-8F52-0040333594A3 >
    DWORD keyType;
    DWORD nKeys;
    array TimedFloatKeys keys[nKeys];
}
```

A DirectX minden animációt a kulcskockák időben növekvő sorozatával ír le. A közbülső állapotok számítása mindig két konkrét kulcskocka közötti lineáris interpolációval történik. Az API olyan animációkat tud kezelni, amelyekben az időbeli változás leírható három elemi transzformációval: forgatás, skálázás, eltolás. Így végső soron az időbeli változás leírható 4 × 4-es homogén transzformációs mátrixokkal, tehát megfeleltethetők a hierarchiában levő képkockák traszformációs mátrixának. Ennek jelentősége a későbbiekben a bőrözés leírásakor derül ki.

A kulcskockák megadására a DirectX állományban alapvetően kétféle lehetőség van aszerint, hogy az eltolási, forgatási és skálázási értékeket külön-külön adjuk meg, vagy a teljes transzformációs mátrix adott az egyes időpillanatokban. A template által definiált első érték ezt a megadási módot határozza meg; a lehetséges értékei a következők:

Érték	Jelentés
0	A kulcskocka-adatok forgatási adatokat tartalmaznak. Az értékek négy lebegőpontos számmal vannak megadva, amelyek egy-egy quaternion komponenseinek felelnek meg.
1	Skálázási adatok, a megfelelő vektorok komponensei vannak megadva három-három lebegőpontos szám formájában.
2	Eltolási adatok, az előbbihez hasonló formában.
3, 4	A transzformációk kompozit, mátrixos alakban adottak.

Az AnimationKey következő adattagja az összes kulcskocka számát adja meg, majd a kulcskockák felsorolása következik. A definícióban szereplő TimedFloatKeys gyakorlatilag az egyes időpillanatokat, majd az ezekhez rendelt értékeket sorolja fel.

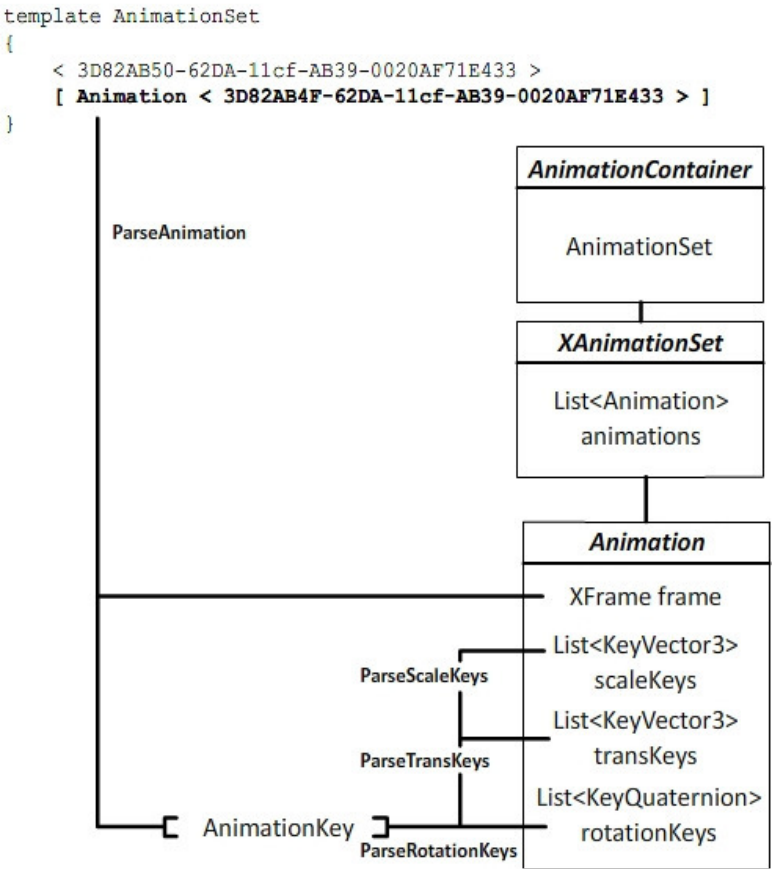
A kulcskocka-animációk feldolgozásának sémája az 5. ábrán látható.

Az időbeli változások aggregált formában, azaz mátrix-alakban történő tárolása rendkívül hatékony az erőforrás-felhasználás szempontjából. Viszont a kulcskockák közötti állapotok számítása nem történhet meg a mátrix minden elemének egyenkénti interpolációjával (főleg a forgatási komponens miatt); az így keletkezett mátrix jó eséllyel nem lesz a megfelelő transzformációs mátrix. Ezért a közbülső állapotok számítása mindig három részműveletből tevődik össze: a skálázási és eltolási vektorok lineáris interpolációja, illetve a forgatást leíró két megfelelő quaternion közötti gömbi lineáris interpoláció. Ezek után rendelkezésünkre fognak állni a három transzformációt az adott közbülső időpillanatban leíró struktúrák, amelyekhez elkészítjük a megfelelő homogén transzformációs mátrixot.

Az AnimationController minden pillanatban pontosan egy animációs csoportot (AnimationSet-et) tart nyilván. A beolvasás során ez a megtalált első AnimationSet lesz. A program képes a további mozgássorozatokat is kezelni (ellentétben a legtöbb ingyenes DirectX állomány-beolvasóval). Ezek adatai viszont rengeteg erőforrást igényelnének, és mivel egyszerre legfeljebb csak egyet játszunk le közülük, felesleges valamennyiüket a memóriában tartani.

Ezért a program a további mozgássorozatokhoz csak ún. referencia pontokat tárol el a beolvasás alatt—tehát nyilvántarjuk azt, hol találhatók még AnimationSet template-ek az állományban. A referenciák kezdő-, és végindexszel adottak. A hivatkozási alap pedig nem más, mint az egyszerű forrásállomány-nézet alapjául szolgáló StringBuilder tartalma. Mivel ez mindig elérhető lesz, egy újabb AnimationSet kiválasztásánál egyszerűen kikeressük annak referenciapontját, és az ez által meghatározott sztringből beolvassuk az adatokat. A sztringből történő beolvasásra az StrReader osztály szolgál, amely lényegében az XReader módosított metódusait használja.

Az XReader a többlet-adatok esetén nem végzi el az objektumleképezést, minden más felgolgozási lépés (és ezzel együtt a szintaxis ellenőrzése) ugyanúgy végbemegy.



5. ábra: A kulcskocka alapú animációk feldolgozása

Minden animáció egy adott Mesh objektumra vonatkozik. Egy mozgássorozatban résztvevő összes animáció összefogására szolgál az AnimationSet template. A programban ennek reprezentációja az XAnimationSet osztály. Ez az osztály tehát az AnimationSet template-ben levő adatok leképezésének eszköze; az ott leírt animációkat tárolja.

Az egyes kulcskockák információi az Animation objektumba kerülnek. Az osztály példányai a kulcskockákat a Direct3D KeyVector3, illetve KeyQuaternion struktúráiban helyezik el. Azt az objektumot, amelyre az időbeli változás vonatkozik, az adott objektumot tartalmazó XFrame segítségével tároljuk. Ennek oka az, hogy az animáció végső soron nem más, mint egy konkrét objektum helyzetét leíró transzformációs mátrix időbeli változása. Az XFrame pedig mind a mátrixot, mind magát az objektumot tartalmazza.

Vektorok esetén a kulcskockák közötti állapotok számítására a Vector3 struktúra Lerp metódusa használható:

```
Vector3 Lerp(Vector3 left, Vector3 right, float interpolater)
```

A metódus eredménye a $left + interpolater(right - left)$ vektor, ahol $interpolater \epsilon [0,1]$.

A forgatási komponens interpolációjának leírása megkívánja a quaternion általános ismertetését.

QUATERNIONOK ÉS A GÖMBI INTERPOLÁCIÓ

A quaternion általánosan egy $q = [v, \theta]$ pár, ahol v egy háromdimenziós vektor, θ pedig egy, a v körüli pozitív irányú forgatást leíró szög, radiánban. Tehát hatékonyan képes leírni egy általános tengely körüli forgatást. A DirectX-ben a quaternion egy $[x, y, z, w]$ négyes, amelynek tagjai az előbbi jelöléseket használva a v komponensei, majd a θ szög. Ha feltételezzük, hogy a v általános forgatási tengely normalizált, a q négy komponense a következőképpen adódik:

$$q.X = sin(\theta/2) \cdot v.X$$
$$q.Y = sin(\theta/2) \cdot v.Y$$
$$q.Z = sin(\theta/2) \cdot v.Z$$
$$q.W = cos(\theta/2)$$

A quaternion esetében az egyszerű lineáris interpoláció nem mindig kielégítő. Ugyanis ezzel a módszerrel a forgatási szög változása nem lesz egyenletes az animáció ideje alatt. Ezért a forgatás interpolációjakor az ún. gömbi interpoláció (Slerp—spherical linear interpolation) módszerét használjuk. Matematikailag ez a következő alakban írható, ha adott $left, right$ quaternion és $interpolater \in [0, 1]$: $left^{1-interpolater} \cdot right^{interpolater}$

Az animáció vezérlésére alapvetően két megközelítés kínálkozik. Ezek egy naív elnevezése lehet: passzív illetve aktív módszer.

A passzív megjelölés arra vonatkozik, hogy a megjelenítési logika futásakor mindig az utolsó megjelenítési ciklus óta eltelt idő függvényében változtatjuk a jelenetet. Ez a módszer a rendszer terheltsége esetén szaggatott mozgást eredményezhet. Az aktuálisan eltelt időt a Windows API számlálóitól kérdezzük le, és leképezzük az aktuális animációs készlet teljes hosszára. Az így kapott t időértékhez megkeressük annak a három kulcskockának az indexét (keyrot, keyscale, keytrans), amelyekre

$$t_{keyrot} \leq t < t_{keyrot+1}$$
$$t_{keyscale} \leq t < t_{keyscale+1}$$
$$t_{keytrans} \leq t < t_{keytrans+1}$$

Ezek alapján elvégezzük az interpolációt. A módszer hátránya, hogy az animáció sebessége lényegében csak a megjelenítési függvény hívásának intenzitásától függ.

Az aktív módszer lényege, hogy magának a megjelenítési logikának a lefutását egy meghatározott eseményhez, egy adott időmennyiség elteltéhez kötjük. Tehát lehetőség nyílik a frissítés gyakoriságának szabályozására. Ebben az esetben sincs garancia arra, hogy az időzítés pontos; csak azt tudjuk elérni, hogy a megjelenítés gyakorisága ne legyen kisebb, mint egy adott időküszöb. A kulcskockákon sorrendben haladunk végig, nincs meg annak a kockázata, hogy „átugrunk” kulcskockákat. A program ez utóbbi módszert valósítja meg a nagyobb fokú szabályozhatóság miatt.

BŐRÖZÉS

Egy Frame template általánosan egy transzformációs mátrixot tartalmaz, és azoknak a térbeli objektumoknak az adatait, amelyek térbeli elhelyezkedését ez a mátrix közvetlenül befolyásolja. Az itt leírt lapokat, amelyek az objektum megjelenését, alakját meghatározzák, összességükben nevezhetjük az alakzat bőrének (skin). Maguk a transzformációs mátrixok (illetve ezek közül egyesek) tekinthetők az alakzat csontvázának (skeleton). Az analógia lényege az, hogy ha a mátrixok változnak, a megfelelő háromszöglapok is követik ezt a változást, a tartalmazott alakzatokkal együtt, hasonlóan ahhoz, ahogy egy élő szervezet tagjai mozognak. Ezért minden Frame-et tekinthetünk az alakzat csontvázában egy csontnak (bone). Matematikailag egy bone nem más, mint maga az adott Frame-ben levő transzformációs mátrix.

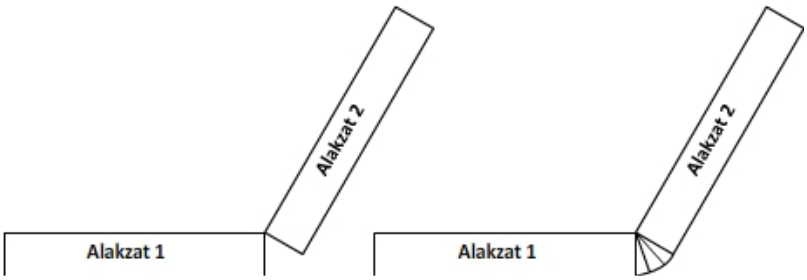
Minden csont helyzetét két transzformáció írja le:

- Lokális transzformáció: tegyük fel, hogy a csont a saját lokális terében (bone space) úgy helyezkedik el, hogy az ízülete (a közös terület, amelyen a szülőjével érintkezik) az origóba esik. Ekkor a lokális transzformáció egyrészt egy forgatást ad meg ekörül az origó körül, másrészt egy eltolást, amellyel elérjük, hogy a csont ténylegesen csatlakozzon a szülőjéhez.
- Kombinált transzformáció: az adott csontot az objektum csontvázában helyezi el (transzformációt végzünk az objektum lokális terébe). Ez az, ami ténylegesen az alakzat lapjainak helyzetét is befolyásolja.

A kombinált transzformációs mátrix meghatározása a következőképpen történik: először alkalmazzuk az adott csont lokális transzformációját, majd a szülőjének lokális transzformációját, és így tovább, egész a legfelső szintig.

A DirectX Frame objektuma—és a programban használt XFrame osztály—egy univerzális szerkezet, ugyanis nemcsak az objektumok hierarchiáját adja meg, hanem bőrözött alakzat (skinned mesh) esetén ugyanez a hierarchia az alakzat csontvázát is megadja, ahol a lokális transzformáció a TransformationMatrix adattagban helyezkedik el. Ezt egészíti ki a program a CombinedTransformMatrix adattaggal, ami a kombinált transzformációt írja le.

Mindezzel együtt eddig nem definiáltunk semmilyen közvetlen kapcsolatot az alakzat csontváza és bőre között. A kézenfekvő az lenne, hogy minden csontnak megfelel egy alakzat (amelyet az adott Frame objektumban helyezünk el), és ezek geometriai adatai a megfelelő csont lokális terében lennének megadva, ezzel a kombinált transzformációk helyesen elvégeznék az alakzatok illesztését a csontváznak megfelelően. Viszont ekkor az alakzat bőre általában nem lesz összefüggő. A problémát a 6. ábra mutatja.



6. ábra: Alakzatok pozícionálása bőrözés nélkül, illetve bőrözéssel

A megoldás az, hogy az objektumunk bőrét egyetlen, összefüggő alakzatnak tekintjük. Ezáltal lehetőség van arra, hogy a kapcsolódások közelében levő vertexek pozícióját több csont is befolyásolja; mégpedig a pozíciót a megfelelő mátrixok súlyozott összege adja. Ez a megközelítés a bőrözés vagy vertex blending alapja.

Az elméletből egy dolog hiányzik még: az előbb az egyes alakzatok a csont lokális terében voltak definiálva, így használhattuk a kombinált transzformációs mátrixokat a pozíciók meghatározásához, amely által az alakzatok egy közös objektumtérbe kerültek. Most viszont a teljes alakzat közvetlenül az objektumtérben lesz megadva, ezért bevezetünk még egy, ún. offszet transzformációt. Minden csonthoz rendelünk egy offszet mátrixot, amely a vertexeket a csont lokális terébe viszi át. Innentől kezdve a kombinált transzformáció ismét használható lesz. Tehát az i. csonthoz a következő transzformációt rendeltük hozzá:

$$F_i = O_i C_i$$

O_i az i. csont offszet mátrixa, C_i a kombinált transzformációs mátrix.

A gyakorlatban a DirectX maximum négy mátrix befolyását engedélyezi egy vertexre. Korábban már láttuk a program által használt vertex-struktúrát, amely a súlyokat és a mátrixok indexeit is tartalmazta. Ezek az indexek az ún. mátrix palettára vonatkoznak, amely valójában minden egyes csont F mátrixát tartalmazza.

Ilyen módon minden csúcspont pozíciója a következőképpen kapható:

$$P = \sum_{i=0}^{n-1} w_i \boldsymbol{v} F_i$$

ahol n az adott csúcspontot befolyásoló mátrixok száma, $w_i \in [0,1]$ az i. mátrixhoz rendelt súly, $\sum w_i = 1$; $\boldsymbol{v}$ az inputként adott vertexpozíció.

A JELENET MEGJELENÍTÉSE

A Direct3D-ben két út létezik a háromdimenziós világ megjelenítésére: az API beépített funkcióit használó ún. fixed function futószalag, illetve a programozható futószalag. Az előbbi főleg a retained mode, az utóbbi elsősorban az immediate mode szemléletét tükrözi. Tehát míg a fixed function futószalag esetén a felhasználó a megjelenítést paramétereken keresztül befolyásolja, addig a programozható futószalag esetén lehetőség van a megjelenítés több lépésébe is a konkrét algoritmusok szintjén beavatkozni, ami nagyobb flexibilitást jelent.

A program ezen flexibilitás miatt a programozható futószalag lehetőségeit használja fel arra, hogy a fixed function futószalag által nyújtott szolgáltatások közül minél többet megmutasson.

A Direct3D a futószalag hatékony kezeléséhez az Effect nevű eszközt biztosítja. Az Effect igazából a megjelenítési futószalag vertex-, és pixelmanipulációs lépéseit fogja össze, hogy ezekhez bizonyos közös támogatásokat nyújtson. Ennek a támogatásnak a főbb eszközei:

- Lehetőség van a beépített, vagy a programozható futószalag funkcióit használnunk a vertex- és pixelműveletekhez. Utóbbi esetben a két csoportot jól elkülönítő vertex, illetve pixel shader programokat definiálunk. Ezek olyan kódok, amelyeket az API közvetlenül a grafikus hardver gépi kódjára fordít le. Régebben a shader programokat az assemblyhez hasonló nyelven kódolták, ma már elterjedt a nagyobb kényelmet biztosító HLSL (High Level Shader Language) használata.
- Globális változókat definiálhatunk, amelyek az effekt és az alkalmazás által is módosíthatók, például a megjelenítés paraméterei kívülről és valós időben is változtathatók lesznek.
- A futószalag működése a render state beállítások állításával módosítható. Tehát közvetlenül az effekt képes módosítani a megvilágítás, árnyalás és egyéb funkciók működését; ugyanúgy, ahogy azt az alkalmazás a Device RenderState objektumában levő beállításokon keresztül teszi.
- Az effekt képes a textúrázás, a mintavételezés működését befolyásolni: lehetőség van textúrák, ún. sampler objektumok megadására (ezek szükségesek ahhoz, hogy egy pixel shader egy képponthoz a textúra megfelelő pontját tudja hozzárendelni).
- Több, független megjelenítési technikát (technique) definiálhatunk. Tipikusan a támogatott vagy felhasznált funkcionalitás, illetve a hardver képességei alapján képzett választási lehetőségek szerint több technikát adunk meg. Minden technika legalább egy megjelenítési menetet (pass) tartalmaz, amely a futószalag minden lépését (vertex-, és pixelműveletek) magába foglalja. Többmenetes megjelenítést általában speciális effektusok létrehozására használnak.

Az effekt tehát igen erős eszközt biztosít az alkalmazások számára, hogy a megjelenítés paramétereit, és gyakorlatilag az egész futószalag működését egész a részletekig a kezükben tartsák. Az API ugyanis tartalmaz eszközöket a globális változók, a technikák és az azokban levő menetek közvetlen elérésére. Lehetőség van akár futási időben újabb shader programok megadására, fordítására és használatára is.

Az, hogy az effekt által leírt vertex és pixel shader hogyan illeszkedik a DirectX megjelenítési futószalag működésébe, a Függelékben levő ábrákon látható.

A program által alapértelmezettként használt effekt beállítása a már bemutatott InitGraphics metódus hatására történik. A teljes megjelenítési folyamatban több metódus is részt vesz, amelyeket a hívás időrendjében veszek sorra.

INITSHADER

Itt történik a megjelenítéshez használt effekt létrehozása az API Effect osztályán keresztül. Az effekt maga egy különálló állomány, amely a DirectX szabvány szerint a .fx kiterjesztéssel van ellátva. Ebből a Direct3D Effect osztályának egy példányát készítjük el az Effect osztály FromFile statikus metódusának használatával. Ennek definíciója a következő:

```
public static Effect FromFile(Device, string, Include, string, ShaderFlags,
EffectPool);
```

Általában az alkalmazásnak nem kell a harmadik és utolsó paraméterrel foglalkoznia, ezek nagy valószínűséggel null referenciák lesznek. A második paraméter az effektet tartalmazó állomány elérési útja, a negyedik paraméterként például az effekt által hivatkozott esetleges egyéb adatokat definiáló ún. effekt header (.fxh) állomány elérési útja adható meg. A ShaderFlags enumeráció további fordítási és optimalizációs beállításokat ad meg.

A metódus másik fontos feladata az effektben található globális változókhoz olyan hatékony hozzáférési pontok regisztrálása, amelyek lehetővé teszik azok értékének gyakori módosítását a program futása alatt. Erre az ún. EffectHandle szolgál; a specifikáció szerint minden globális változóhoz ajánlott ilyen objektumot létrehozni, ugyanis az ilyen elérés hatékonyabb, mint a név szerinti elérés (tekintve, hogy az EffectHandle állandó összeköttetést valósít meg az alkalmazás és az adott változó között). A program mindig ezt a metódust hívja, ha a használt effekt cseréje szükséges, minden esetben a betöltés és egy EffectHandle tömb feltöltése fog történni.

RENDERSCENE

Az összes megjelenítési tevékenység kiindulási pontja. A metódus által végrehajtott főbb lépések a következők:

- Animáció esetén az egyes transzformációs mátrixok frissítése
- UpdateFrameMatrices hívása
- A megvilágításhoz szükséges effekt-paraméterek beállítása
- A jelenet rekurzív kirajzolása a DrawFrame metódussal

Maga a kirajzolás egy rögzített megjelenítési séma szerint történik, amely hat különálló lépésre bontható:

- A megjelenítési felület beállítása. Több felület használata esetén a Device SetRenderTarget metódusa használható erre. A GetBackBuffer metódus egy adott, hátsó puffer szerepet játszó felületet ad vissza. Alapértelmezés szerint a render target automatikusan kerül beállításra, így az alkalmazás megspórolhatja ezt a lépést, ha a lapozási lánc egy elülső és egy hátsó pufferből áll. A programban ezt a lépést akkor használjuk, ha a Megjelenítés modul aktív, a kamera által látott kép kirajzolásához.
- A megjelenítési felület és a hozzárendelt mélységi és stencil puffer törlése. Erre a Device Clear metódusa szolgál. Itt meg kell adnunk, milyen felületeket szeretnénk törölni, és milyen színnel; illetve milyen értékekkel szeretnénk a puffert feltölteni.
- A megjelenítési logika kezdetét jelző BeginScene hívása.
- A jelenet kirajzolása az alkalmazás igényei szerint (most ez a DrawFrame metódus hívása).
- A megjelenítési logika végét jelző EndScene hívása.
- A Present metódus hívása. Ekkor történik meg az elülső-hátsó pufferek cseréje a beállított lapozási mód szerint.

UPDATEFRAMEMATRICES

Láttuk, hogy a DirectX állományban tárolt hierarchia egysége a Frame, amely tartalmaz egy transzformációs mátrixot, esetleg egy Mesh adatait, vagy újabb Frame példányokat. Azt is láttuk, hogy az itt leírt hierarchia közvetlenül megfeleltethető az objektumok hierarchiájának. Nem meglepő tehát, hogy a transzformációs mátrixok öröklődnek a tartalmazott Frame template-ek felé is. Ez a metódus a felső szintű képkockától elindulva rekurzív módon újraszámítja a Frame példányok transzformációs mátrixait. Erre a lépésre azért van szükség, mert a mátrixokat az animáció időben változtatja. A metódus működése a következő:

```
private void UpdateFrameMatrices(XFrame frame, Matrix parentMatrix)
{
        frame.CombinedTransformationMatrix = frame.TransformationMatrix *
                                parentMatrix;
        if (frame.FrameSibling != null)
        {
                UpdateFrameMatrices((XFrame)frame.FrameSibling, parentMatrix);
        }
        if (frame.FrameFirstChild != null)
```

```
        {
                UpdateFrameMatrices((XFrame)frame.FrameFirstChild,
                        frame.CombinedTransformationMatrix);
        }
}
```

A metódus első hívásánál a parentMatrix a teljes jelenet aktuális forgatását, eltolását, skálázását leíró mátrix lesz (amelyet a nézeti és vetítési paramétereket magában foglaló ModelViewerCamera ad meg). A hierarchiában történő navigáláshoz a Frame osztály két metódusát használjuk: a FrameSibling az aktuális képkocka első testvérét adja vissza (tehát a soron következő, az aktuálissal egy szinten levő képkockát); a FrameFirstChild értelemszerűen az aktuális képkocka első gyermekét adja meg.

DRAWFRAME

A hierarchiában tárolt objektumok rekurzív kirajzolását kezdeményezi.

```
private void DrawFrame(XFrame frame)
{
        XMeshContainer mesh = (XMeshContainer)frame.MeshContainer;
        while (mesh != null)
        {
                DrawMeshContainer(mesh, frame);
                mesh = (XMeshContainer)mesh.NextContainer;
        }
        if (frame.FrameSibling != null)
        {
                DrawFrame((XFrame)frame.FrameSibling);
        }
        if (frame.FrameFirstChild != null)
        {
                DrawFrame((XFrame)frame.FrameFirstChild);
        }
}
```

DRAWMESHCONTAINER

A Frame objektumokban tárolt Mesh megjelenítése. A metódusban elágaztatás történik aszerint, hogy az adott Frame aktuális MeshContainer-ében bőrözött alakzat van-e tárolva vagy nem (ez az alakzat beolvasása alatt eldőlt). Az elágaztatás a megjelenítéshez használt effektus megfelelő technikájának kiválasztásához szükséges. Ettől kezdve a tevékenységek gyakorlatilag megegyeznek. Ezek váza a következő:

```
Mátrix-paraméterek beállítása az effekt számára (vetítési és egyéb mátrixok);
A Device vertexformátumának beállítása;
Ciklus az alakzat által használt minden anyagra
{
        Anyag-paraméterek beállítása (diffuse, specular, ambient és egyéb
        tulajdonságok);
        if (az anyaghoz tartozik textúra)
        {
                Textúra-paraméterek beállítása;
        }
        //Ez az effekttel történő megjelenítés általános sémája
        int passes = effect.Begin(0);
        for (int pass = 0; pass < passes; pass++)
        {
                effect.BeginPass(pass);
                A Mesh adott anyaghoz rendelt lapjainak kirajzolása;
                effect.EndPass();
        }
        effect.End();
}
```

Az effekt használata során tehát először az alkalmazott technikát választottuk ki, majd a Begin metódust hívva megtudjuk, az adott technika hány menetet tartalmaz. Minden egyes menetben meg kell ismételnünk a kirajzolást, amelyet az effekt BeginPass és EndPass metódusa között végezhetünk el. Az összes menet végeztével az End metódus hívása szükséges.

A megjelenítés paramétereinek változtatása

A paraméterek változtatására szolgáló eszközök leírása a következő részben található.

Az XSharp szemléltető eszközei

Modulok

A programnak az oktatási célokat megvalósító eszközrendszere négyszintű. Az első szintet az egyes paraméterek közvetlen és valós idejű változtatását lehetővé tevő modulok alkotják. A következőkben ezeket veszem sorra.

Nézet és vetítés

Korábban megismertük a megjelenítési folyamatot, mint koordinátaleképezések sorozatát. Ezen sorozat egyik tagja a nézeti, majd a vetítési transzformáció, amely a háromdimenziós tér egy részét egy kétdimenziós síkra képezi le.

A Direct3D általánosan síkbeli geometriai vetítést használ. Ez olyan félegyenesekkel szemléltethető, amelyek egy kitüntetett pontból, az ún. vetítési középpontból indulnak ki. A középpont és az ún. vetítési sík egyértelműen megadják a vetítést. A háromdimenziós objektum minden csúcspontján keresztül húzunk ilyen félegyenest (vetítősugarat vagy projektort). Az objektum síkbeli képét a csúcspontok képe határozza meg, amelyek a projektorok és az adott vetítési sík metszéspontjai lesznek.

Ha a vetítési középpontot, mint a projektorok közös metszéspontját végtelen távolinak tételezzük fel, akkor a vetítősugarak egymással párhuzamosan haladnak, és a párhuzamos vetítéshez jutunk. Ellenkező esetben perspektivikus vetítésről van szó.

A párhuzamos vetítés megőrzi a modellen belül a hosszúságok arányait és a szögeket is jól szemlélteti. Ezen belül kétféle vetítési módot különböztetünk meg aszerint, hogy a vetítősugarak milyen szögben érik a vetítési síkot.

Merőleges vetítésről van szó, ha a párhuzamosan haladó projektorok a vetítési síkra merőlegesek. Az ilyen vetítés nem árul el túl sokat az objektum térbeli tulajdonságairól, ezért általában egyszerre több merőleges vetítés használatával, mintegy több nézetből mutatják be az objektumot. (Ezt hívják többnézetes—multiview—merőleges vetítésnek, sokszor találkozhatunk vele háromdimenziós tervezőprogramoknál, ahol rendszerint elölnézet, oldalnézet és felülnézet egyszerre látható.) Merőleges vetítés esetén a vetítési sík az objektum terének valamelyik tengelyével párhuzamos.

Amennyiben a vetítési sík általános helyzetben van az objektum koordináta-tengelyeihez viszonyítva, akkor axonometrikus vetítésről beszélünk. Ha minden koordináta-tengely ugyanolyan szögben metszi a vetítési síkot, izometrikus vetítést kapunk. Viszont ha csak két tengely metszi a síkot ugyanolyan szögben, illetve ha minden szög különböző, akkor rendre dimetrikus, és trimetrikus vetítéshez jutunk.

A párhuzamos vetítések másik csoportját a ferde vetítések (oblique projection) alkotják. Egy ferde vetítést három dolog határoz meg: a vetítési sík helyzete az objektumhoz képest, a projektorok és a vetítési sík által bezárt szög, illetve a projektorok helyzete a vetítési sík normálvektorához képest. A leggyakoribb ferde vetítések egyike a Kavalier axonometria, ahol a vetítősugarak és a vetítési sík 45 fokos szöget zárnak be.

A vetítések másik típusa a perspektivikus vetítés, ahol a vetítési sík irányából nézve a projektorok a meghatározott távolságban levő vetítési középpontban találkoznak. Ez a vetítési mód a valósághoz közelebb álló képet állít elő:

- Párhuzamos szakaszok, amelyek nem a vetítési síkkal párhuzamos síkon vannak, egy ún. eltűnési pont irányában összetartanak.
- Az objektumok mérete függ a középponttól vett távolságuktól.
- Az objektumok alakja torzulhat.

A perspektivikus vetítések szintén tovább osztályozhatók aszerint, hogy milyen a vetítési sík elhelyezkedése az objektumok koordinátarendszerének tengelyeihez képest. Egypontos perspektíváról beszélünk, ha egy tengely metszi a vetítési síkot; kettő, illetve három metsző tengely esetén két-, illetve hárompontos perspektíváról van szó.

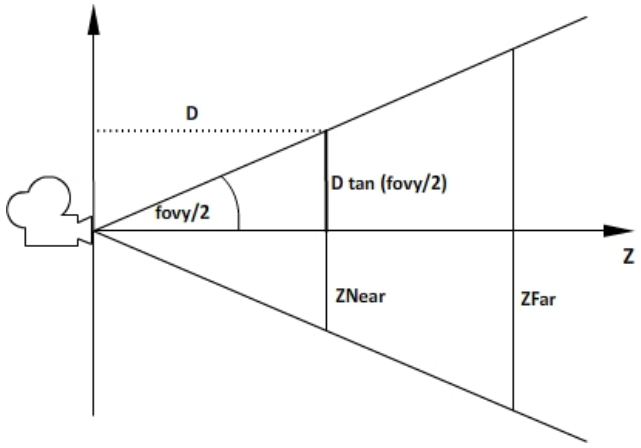
A Direct3D számára a vetítés természetesen egy homogén transzformációs mátrix. A perspektivikus vetítési mátrix a DirectX-ben kétféleképpen származtatható.

Az első megközelítés egyszerű leírásához tegyük fel, hogy a vetítési sík merőleges a z-tengelyre. Legyen a vetítési középpont a koordináta-rendszer origójában, a vetítési sík távolsága a középpottól legyen D. Ekkor a vetítés által végrehajtott transzformáció két lépésre osztható: egy eltolás (-D-vel), illetve egy nemlineáris skálázás. Mátrixokkal leírva:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -D & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & -D & 1 \end{bmatrix}$$

A második mátrix a homogén w komponenst osztja D-vel. Erre azért van szükség, mert a DirectX megköveteli, hogy a perspektív vetítési mátrix (3,4) helyen levő eleme 1 legyen; a későbbi mélységi és stencilműveletek miatt. Ez a megközelítés nem veszi figyelembe a nézeti gúla nyílási szögeit, és az előálló z-értékek a távolság növekedésével igen közel eshetnek egymáshoz; megnehezítve a mélységi tesztelést.

A másik megközelítésnél számításba vesszük a megjelenítési gúla arányait, egyszerűen a szélesség és magasság, vagy a gúla egyik nyílásszöge felhasználásával (ezt hívják field-of-view alapú megközelítésnek). A 7. ábra ezen megközelítést szemlélteti.



7. ábra: A perspektivikus vetítés field-of-view alapú megközelítése

A Direct3D által közvetlenül támogatott vetítési módok a merőleges és a perspektív vetítés. A vetítési mátrixok a DirectX beépített Matrix struktúrájához definiált statikus metódusok segítségével generáltathatók. Lehetőség van egyaránt jobb-, illetve balsodrású koordinátarendszerhez a megfelelő mátrixok elkészítésére. Mi a balsodrású rendszerhez készült változatokat fogjuk bemutatni, mivel a DirectX-ben ez a rendszer az alapértelmezett; a program is ezeket használja.

- Matrix.OrthoLH(float width, float height, float zNearPlane, float zFarPlane): merőleges vetítési mátrixot definiál. A mátrix a paraméterek alapján a következőképp áll elő:

$$\begin{bmatrix} 2/width & 0 & 0 & 0 \\ 0 & 2/height & 0 & 0 \\ 0 & 0 & \frac{1}{zFarPlane - zNearPlane} & 0 \\ 0 & 0 & \frac{zNearPlane}{zFarPlane - zNearPlane} & 1 \end{bmatrix}$$

Itt $width$ és $height$ a nézeti gúla (amely gyakorlatilag egy téglatest) szélességét és magasságát jelentik, a mélységét $zFarPlane$ és $zNearPlane$ különbsége adja meg. Ezek a közeli és távoli vágósíkok távolságai.

- Matrix.OrthoOffCenterLH(float left, float right, float bottom, float top, float znearPlane, float zfarPlane): Az előbbi esetben a koordinátarendszer origója a nézeti gúla középpontjába esett. Ez egy általánosabb eset; a metódus paraméterei gyakorlatilag a gúla vágósíkjainak távolságát adja meg explicit módon, rendre a bal oldali, jobb oldali, alsó, felső, illetve közeli és távoli vágósíkok távolságát. Az OrthoLH ennek speciális esete, a következő értékek mellett: $left = -width/2, right = width/2, bottom = -height/2, top = height/2$. A mátrix alakja a következő lesz:

$$\begin{bmatrix} \frac{2}{right - left} & 0 & 0 & 0 \\ 0 & \frac{2}{top - bottom} & 0 & 0 \\ 0 & 0 & \frac{1}{zFarPlane - zNearPlane} & 0 \\ \frac{(left + right)}{(left - right)} & \frac{(bottom + top)}{(bottom - top)} & \frac{zNearPlane}{zFarPlane - zNearPlane} & 1 \end{bmatrix}$$

- Matrix.PerspectiveLH(float width, float height, float znearPlane, float zfarPlane): A nézeti gúla megadásához annak méreteit használja. Az előálló mátrix a következő alakú lesz:

$$\begin{bmatrix} \frac{2 \cdot zNearPlane}{width} & 0 & 0 & 0 \\ 0 & \frac{2}{height} & 0 & 0 \\ 0 & 0 & \frac{zFarPlane}{zFarPlane - zNearPlane} & 1 \\ 0 & 0 & \frac{zNearPlane \cdot zFarPlane}{zNearPlane - zFarPlane} & 0 \end{bmatrix}$$

- Matrix.PerspectiveOffCenterLH(float left, float right, float bottom, float top, float znearPlane, float zfarPlane): A merőleges vetítéshez hasonlóan ez is az előbbi általános esete. Mátrixa:

$$\begin{bmatrix} \frac{2 \cdot zNearPlane}{right - left} & 0 & 0 & 0 \\ 0 & \frac{2 \cdot zNearPlane}{top - bottom} & 0 & 0 \\ \frac{(left + right)}{(left - right)} & \frac{(bottom + top)}{(bottom - top)} & \frac{zFarPlane}{zFarPlane - zNearPlane} & 1 \\ 0 & 0 & \frac{zNearPlane \cdot zFarPlane}{zNearPlane - zFarPlane} & 0 \end{bmatrix}$$

- Matrix.PerspectiveFovLH(float fieldOfViewY, float aspectRatio, float znearPlane, float zfarPlane): A perspektivikus vetítés mátrixában a gúla függőleges nyílásszögét is figyelembe veszi. Az aspectRatio paraméter a nézeti gúla szélesség-magasság aránya. A mátrix alakja:

$$\begin{bmatrix} width & 0 & 0 & 0 \\ 0 & height & 0 & 0 \\ 0 & 0 & \frac{zFarPlane}{zFarPlane - zNearPlane} & 1 \\ 0 & 0 & \frac{-zNearPlane \cdot zFarPlane}{zFarPlane - zNearPlane} & 0 \end{bmatrix}$$

Itt h a nézeti térben vett magasság, amely $h = cot(fieldOfViewY/2)$ alakban adódik; w a nézeti térben vett szélesség, $h = w / aspectRatio$-ból számítható.

A VIRTUÁLIS KAMERA

A Direct3D koncepciójában a háromdimenziós jelenetet egy virtuális kamerán keresztül szemléljük, amelyet a jelenet koordinátarendszerében helyezünk el. A kamera pozíciója és iránya segítségével egy újabb koordinátarendszert definiálunk, amely a nézeti rendszer (view space vagy camera space). Ebben a kamera az origóban helyezkedik el, és a z-tengely pozitív iránya felé tekint, az y-értékek felfelé, míg x értékei balra növekednek.

A DirectX-ben a kamera modellje három adatból áll: a kamera pozíciója (P), fókuszpontja (F), illetve a $\overrightarrow{PF}$ vektorra merőleges síkban a pozitív irányt meghatározó vektor (U) (szemléletesen ez a „felfelé" irányt adja meg). Tehát a nézeti transzformáció mátrixa három összetevőből áll: eltolás P-be, forgatás F felé ($\overrightarrow{PF}$ a z-tengelyen lesz), forgatás az U síkjában.

Az API tartalmaz eszközöket a nézeti mátrix generáltatására is; ismét a Matrix metódusait használhatjuk. Balsodrású koordinátarendszer esetén a

```
Matrix.LookAtLH(Vector3 cameraPosition, Vector3 cameraTarget, Vector3 cameraUpVector)
```

metódus a megfelelő.

A program futása során a kamera helyzete folyamatosan változtatható—ezzel együtt a nézeti mátrix is állandóan változik.

A Nézet és vetítés modul által változtatható beállítások:

- A vetítés módja (perspektív vagy párhuzamos).
- A vetítési gúla minden síkjának távolsága.
- Perspektív vetítés esetén: függőleges nyílásszög, képarány.

ANYAGTULAJDONSÁGOK

Láttuk, hogy a DirectX állományban definiálhatók ún. anyagok, amelyek aztán az egyes lapokhoz rendelhetőek. Az anyagok biztosítják azt, hogy az egyes lapokat úgy tudjuk kezelni, mint felületeket, amelyek a környezetből jövő hatásokra (pl. fény) meghatározott módon reagálnak.

A programban az anyagokat a MaterialEx struktúra reprezentálja. A Direct3D ExtendedMaterial struktúrájához hasonlóan a következő adatokat tartjuk benne nyilván:

```
ColorValue ambientColor;          //ambiens komponens
ColorValue diffuseColor;          //diffúz komponens
ColorValue specularColor;         //spekuláris komponens
ColorValue emissiveColor;         //emissziós komponens
float materialPower;              //tükröződési hatások erőssége
String textureFilename;           //a textúra elérési útja, amennyiben elérhető
```

Ezeken kívül a struktúra tartalmaz egy Id nevű adattagot az azonosítás megkönnyítésére. Értéke az állományban megadott anyagnév, ennek hiányában egy egyedi generált név.

A DirectX állományban definiált anyagok leképezése a MaterialEx példányaiba történik. Amennyiben az állomány egyetlen anyagot sem definiált, az API specifikációt követve az alapértelmezett, fehér színhatású anyag lesz az objektumhoz rendelve.

A ColorValue struktúra RGBA színértékek tárolására szolgál.

Láttuk, hogy a megjelenítéshez az objektumok lapjait az API alcsoportokra bontja, és az osztályozás alapja (esetlegesen más szempontok mellett) a lapokhoz rendelt anyag. Ezért megjelenítéskor elérhető kell hogy legyen az összes anyag adata. A DirectX fixed function pipeline ezeket az információkat a Frame osztályban tartja nyilván, a Mesh összes adatát tartalmazó MeshContainer példányban. Az XSharp esetén a nyilvántartást a MaterialContainer osztály végzi. Ennek oka az, hogy az anyagok adatait olyan kontextusban is elérhetővé kell tenni, ahol a képkockák hierarchiája (és az abban tárolt MeshContainer-ek) nem elérhetők közvetlenül. A MaterialContainer igazából az összekötő kapocs a beolvasó és az Anyagtulajdonságok modul között; ilyen módon nemcsak a megjelenítést támogatja, hanem az egyes anyagok minden tulajdonságát változtathatóvá teszi.

Mindkét funkciónak az alapjai az anyagok és a textúrák tárolására szolgáló Dictionary példányok (lényegében ezek tábla típusú adatszerkezetek; kulcs-érték párok):

```
private List<MaterialEx> materials = new List<MaterialEx>();
private Dictionary<string, Texture> textures = new Dictionary<string,
Texture>();
private Dictionary<string, int> matIndex = new Dictionary<string, int>();
```

A textúrák nyilvántartása az elérési út alapján történik. Ilyen módon elkerülhető a redundáns tárolás. Az anyagok tényleges adatai a materials listába kerülnek. A matIndex adatszerkezetben a DirectX állományban levő anyagnevekhez egyedi sorszámokat rendelünk. A MeshContainer számára elegendő a szükséges anyagok sorszámait eltárolni.

Erre a látszólag bonyolult mechanizmusra azért van szükség, mert a DirectX állományban semmi sem akadályozza meg az anyagok többszöri definícióját. A tapasztalat azt mutatja, hogy nem minden generálószoftver elég intelligens ebben a dologban, és több objektum esetén gyakran a közös anyagokat többször újradefiniálják. A MaterialContainer új anyag hozzáadásakor ezért ellenőrzést végez:

```
//mat a hozzáadni kívánt MaterialEx példány
if (már van ilyen nevű anyag)
{
    //akkor a név alapján kikeressük a neki megfelelő sorszámot
   return matIndex[mat.Id];
}
int a = 0;
foreach (MaterialEx m in materials)
{
//Az összes tulajdonság egyezése esetén a két anyagot megegyezőnek tekintjük
       if (m.AmbientColor.ToArgb() == mat.AmbientColor.ToArgb() &&
              m.DiffuseColor.ToArgb() == mat.DiffuseColor.ToArgb() &&
              m.EmissiveColor.ToArgb() == mat.EmissiveColor.ToArgb() &&
              m.SpecularColor.ToArgb() == mat.SpecularColor.ToArgb() &&
              m.SpecularSharpness == mat.SpecularSharpness &&
              m.TextureFilename == mat.TextureFilename)
       {
              //Már egyszer eltároltuk ezt az anyagot, név alapján kikeressük az indexét;
              ugyanakkor feljegyezzük, hogy az aktuális névhez is az az index tartozik

              matIndex.Add(mat.Id, a);
              return a;
       }
       a++;
}
```

A ColorValue struktúra ToArgb metódusa az RGBA színkódokat egyszerű int értékekké alakítja, így egyszerűbb az összehasonlítás.

Az anyagtulajdonságokkal kapcsolatos számításokhoz a megjelenítéshez használt effekt az aktuális anyag minden tulajdonságát nyilvántartja. Ezek a tulajdonságok egyrészt az effekt saját Material struktúráján, részben a textúra-deklaráción és a hozzá tartozó sampler-en keresztül érhetők el:

```
struct Material {
     float4 materialAmbient;
     float4 materialDiffuse;
     float4 materialSpecular;
     float4 materialEmissive;
     float materialPower;
};
Material material;

texture tex1;
sampler SampTexture =
sampler_state
{
    Texture = <tex1>;
    MipFilter = LINEAR;
    MinFilter = LINEAR;
    MagFilter = LINEAR;
};
```

A modul lehetővé teszi az összes definiált anyag valamennyi nyilvántartott tulajdonságának változtatását, beleértve az anyaghoz rendelt textúra-hivatkozást is. Ez utóbbi globálisan is megtehető, tehát a felhasználónak lehetősége van valamennyi anyagban lecserélni az adott textúrára történő hivatkozást, ahol az előfordul. A módosítások egy anyagra vonatkozóan és globálisan is érvényteleníthetők.

MEGVILÁGÍTÁS, ÁRNYALÁS

A megvilágítás az egyes vertexekhez rendel színértéket a következő információk alapján: az adott vertexnél használt anyag tulajdonságai, a fényforrás tulajdonságai, illetve az ún. visszaverődési modell, amely a számításokat konkrétan definiálja.

A Direct3D ún. lokális megvilágítási modellt használ. Tehát minden vertexhez úgy számítjuk ki a színét, hogy az összes többi vertextől függetlennek tekintjük. Ebből következően a rendszer nem képes közvetlenül árnyékokat előállítani, mivel nem vizsgálja az objektumok egymáshoz való viszonyait.

Az anyagtulajdonságok ismeretében a Direct3D minden egyes vertexre meghatározza, hogy az adott vertexet megvilágítva milyen visszaverődés következik be—a visszavert fény mértéke határozza meg, az objektum az adott vertexben milyen színű lesz.

Az API a visszavert fénynek a következő formáit különbözteti meg:

- Ambiens: Olyan fényt jelöl, amelynek a forrása a jelenetben minden objektumot egyformán világít meg. Legegyszerűbben háttérmegvilágításként képzelhető el. Pozíció és irány tulajdonságai nem értelmezettek. Minden objektum színintenzitását egyenletesen növeli.
- Diffúz: Olyan felületek esetén tapasztalható, amelyek az adott szögben érkező fénysugarakat minden lehetséges irányban megközelítőleg egyenlő intenzitással szórják szét (a hétköznapokban ezeket matt felületeknek nevezzük). A Direct3D itt a Lambert-féle koszinusz-törvény segítségével számol. Az intenzitás függ a vertexhez tartozó felületi normális és a fény terjedési irányának viszonyától, de független a megfigyelő helyzetétől.
- Spekuláris: A felület viselkedése leginkább egy tükörhöz hasonlítható: a beérkező fény nagy részét a felület egyetlen sugárként veri vissza (a gyakorlatban ezek a tükröződő felületek). Ebben az esetben a számítások az optimális visszaverődést tételezik fel, azaz a beérkező fénysugár és a visszavert fénysugár ugyanakkora szöget zár be a pontban vett felületi normálissal. Az intenzitás a megfigyelő helyzetétől is függ. Az API itt a Phong-Blinn modellt használja.
- Emissziós: Igazából itt nem is visszavert fényről, hanem az objektum által kibocsátott fényről van szó. Ennek ellenére a Direct3D az ilyen tulajdonságú objektumokat nem tekinti fényforrásoknak. Segítségével fluoreszkáló és hasonló felületeket lehet modellezni.

A Direct3D háromféle fényforrást képes modellezni: irányított, pontszerű és spot fényforrásokat. Közös tulajdonságuk a fényforrás által kibocsátott fény intenzitása, amely külön ambiens, diffúz, spekuláris komponenseivel adható meg.

Az irányított fényforrás jellemzője a fénysugarak közös irányát meghatározó vektor. A vertexekben számított visszverődés intenzitása nem függ a fényforrás helyzetétől; minden objektumot egyenletesen világít meg. Távoli fényforrások, elsősorban a Nap modellezésére használják.

A pontszerű fényforrás minden irányban egyenlő intenzitással bocsát ki fényt, egy meghatározott pontból. Ha a fényforrás távol van az objektumtól, hatását tekintve az irányított fényforrást közelíti; ellenkező esetben a különböző csúcspontokba különböző szögben érkeznek be a fénysugarak. A pontszerű fényforrás hatását egy meghatározott hatókörben ($Range$) fejti ki, amelyen belül a fényintenzitás egyenletesen gyengül. Ennek a gyengülésnek a mértékét egy tompítási faktor ($attenuation$) szabályozza.

A spot fényforrás hatását egy kúpszerű tartományon belül fejti ki, hasonlóan egy zseblámpához. A kúpon belül a fényintenzitás nemcsak a fényforrástól vett távolsággal változik; a megvilágított területen belül is két tartomány lesz elkülöníthető: egy belső, világosabb rész, amelyet a θ szög határoz meg; és egy külső, gyengébben megvilágított tartomány, amelyet a φ szög határoz meg. A kettő közötti intenzitásátmenet a Falloff paraméterben adható meg. Ha $Falloff > 1$, az átmenet hirtelen és éles, míg $Falloff < 1$ esetén folyamatosabb átmenetet érhetünk el.

Minden vertexben a visszavert fény intenzitását a tompítási faktorral moduláljuk. Irányított fényforrás esetén $attenuation = 1$. Pontszerű fényforrás esetén tompítás értéke a $Range$ paramétertől függ, mégpedig

$$attenuation = \begin{cases} 0 & d > Range \\ \dfrac{1}{A_0 + A_1 d + A_2 d^2} & d \leq Range \end{cases}$$

d az aktuális vertex távolsága a fényforrástól. A_0, A_1 és A_2 rendre a tompítás konstans, lineáris és kvadratikus tagjait jelölik. Értékük a [0, ∞) intervallumból kerülhet ki, legalább egyikük nemnulla.

Spot fényforrás esetén a tompítás:

$$attenuation = \begin{cases} 0 & \\ \left(\dfrac{Spot_0 - Spot_1}{Spot_2 - Spot_1}\right)^{Falloff} & Spot_0 \leq Spot_1 \\ & Spot_1 < Spot_0 \leq Spot_2 \\ 1 & Spot_2 < Spot_0 \end{cases}$$

$Spot_0$ a fénykibocsátás irányának és a vertexbe húzott vektornak skalárszorzata; $Spot_1 = \cos(\theta/2)$, $Spot_2 = \cos(\varphi/2)$.

A programban egyszerre mindig egyetlen fényforrás aktív. A megvilágítást végző shader egy saját Light nevű struktúrában fogja össze a fényforrás összes tulajdonságát. Ez a struktúra lényegében megfelel a Direct3D Light struktúrájának; felépítése a következő:

```
struct Light
{
   int iType;
   float3 vPos;
   float3 vDir;
   float4 vAmbient;
   float4 vDiffuse;
   float4 vSpecular;
   float  fRange;
   float3 vAttenuation;          //1, D, D^2;
   float3 vSpot;                 //cos(theta/2), cos(phi/2), falloff
};
```

A Spot adattag a $Falloff$, a $Spot_1$ és $Spot_2$ értékét tárolja.

VISSZAVERŐDÉSI KOMPONENSEK

A DirectX által a vertexekhez rendelt színértékek általánosan az

$$Intenzitás = Ambient + Diffuse + Specular + Emissive$$

képlet szerint kerülnek kiszámításra, ahol a tagok mind ún. megvilágítási termek; önállóan számított értékek. Az ambiens term a következőképpen számítható:

$$Ambient = Material.Ambient(GlobalAmbient + attenuation \cdot Light.Ambient).$$

A tompítási moduláció mindenütt a fényforrás típusától függ. A $GlobalAmbient$ az objektumoktól független megvilágítást adja meg, amely a RenderState.Ambient tulajdonsággal adható meg.

A diffúz intenzitás nem más, mint a vertexbe beérkező összes diffúz fény intenzitása, amelyet a tompítási faktorral, majd az anyag diffúz komponensével modulálunk. A számítás során alkalmazzuk Lambert visszaverődési képletét. A Lambert-féle visszaverődési modellben a visszavert fény intenzitása a vertexben számított felületi normális és a fény terjedési iránya által bezárt szög koszinuszaként adódik. Ha mindkét vektor normalizált, a koszinusz a skaláris szorzatból kapható, tehát $Id = dot(n, -Light.Dir)$. A negatív előjelre azért van szükség, mivel n a vertexből kifelé, $Light.Dir$ pedig a vertexbe mutat. Mindezek alapján

$$Diffuse = Material.Diffuse(attenuation \cdot Light.Diffuse \cdot Id).$$

A spekuláris intenzitás, mint már említettem, a Phong-Blinn modell alapján kerül kiszámításra. Eszerint a visszavert fény intenzitása ls a vertexben vett felületi normális és egy ún. félutas vektor által bezárt szög koszinuszától függ. Ezt ismét skaláris szorzattal kapjuk, amelyet az anyag által definiált erősségi együttható hatványára emeljük. Ez az együttható a felületen megjelenő tükröződési foltok (specular highlight) kiterjedését és fényintenzitását határozza meg. Tehát

$$Specular = Material.Specular(attenuation \cdot Light.Specular \cdot ls^{Material.SpecularPower})$$

Az objektum által közvetlenül kibocsátott fény intenzitása nem függ egyetlen fényforrás tulajdonságaitól, sem a vertex egyéb adataitól. Ezért

$$Emissive = Material.Emissive.$$

ÁRNYALÁS

Az árnyalás azt határozza meg, hogy a vertexekhez rendelt színértékek alapján hogyan rendelünk színértéket a primitívek (háromszögek) belső pontjaihoz.

A Direct3D két árnyalási módot ismer, ezek a flat, illetve a Gouraud-féle árnyalás.

Flat árnyalási mód esetén az aktuális primitív minden pontjának színe az első vertexének színinformációjából kerül meghatározásra. Az egyes primitívek között általában élesen kivehetők lesznek a határvonalak. A program vertex shadere ehhez az árnyaláshoz nem tartalmaz külön kódot; a fixed function pipeline szolgáltatásait veszi igénybe. Ehhez a hozzá tartozó technika leírásában beállítjuk a megfelelő render state tulajdonságot: ShadeMode = Flat.

A Gouraud-féle árnyalás során a vertexekben számított normálisokból indulunk ki. Ezek alapján minden vertexhez számítunk intenzitásértékeket egy tetszőleges visszaverődési modell alapján. Ezen intenzitásértékekkel azután először az adott poligon élei mentén, majd a poligon közbülső pontjain lineáris interpolációt hajtunk végre.

A DirectX-ben jelenleg hivatalosan nincs implementálva, de a program a teljesség igénye miatt támogatja a Phong árnyalási módot is.

A Phong árnyalás során minden ponthoz külön számítunk normálist. A vertexekben adott normálisok felhasználásával először az élek mentén, majd a poligon belső pontjaira interpoláció segítségével kapjuk a normálist. Az intenzitás értékei ezek alapján pontonként számíthatók tetszőleges visszaverődési modell használatával. Mivel ez az árnyalási mód pixelszintű, ezért a hozzá tartozó számításokat a pixel shader végzi.

A program a Phong-Blinn-féle visszaverődési modellt használja, tehát minden pixelre az intenzitásérték

$$I = Ambient + (L \cdot N) \cdot Diffuse + (H \cdot N)^{Material.Power} Specular + Emissive$$

alakban adódik, ahol L a fényforrás pozíciójába mutató vektor; N a normalizált normális, és H az úgynevezett félutas (halfway) vektor, $H = (L + V)/|L + V|$, V a nézőpontba mutat. A Phong-Blinn modellnek lényegében ez a vektor az alapja. Az eredeti Phong-modellben az optimális visszaverődési vektorral (R) számoló $R \cdot V$ term szerepel, amely a két vektor által bezárt szög koszinuszát adja. A $H \cdot N$ ezen szög koszinuszának a felét közelíti (innen kapta a H a félutas vektor nevet), ezáltal a számítás egyszerűsödik.

A Megvilágítás, árnyalás modul által módosítható beállítások:

- A fényforrás típusa.
- A kibocsátott fény színe.
- Az árnyalás módja.
- Hatósugár.
- A tompítási faktor konstans, lineáris, négyzetes tagja, ezek figyelembevételének engedélyezése és tiltása.
- Spot fényforrás esetén a kúp két nyílásszöge, valamint az átmenet erősségét szabályozó $Falloff$ paraméter.

TRANSZFORMÁCIÓK

A program lehetőséget biztosít a felhasználó számára, hogy közvetlenül a megjelenítés ablakában az egér segítségével mindhárom alapvető transzformációt elvégezhesse a teljes jelenetre nézve. A bal egérgombbal forgatás, a jobb egérgombbal eltolás végezhető; a görgő mozgatásával a skálázás módosítható.

Ezek a műveletek közvetlenül a nézeti transzformációra vannak hatással. A programban a nézeti leképezést a ModelViewerCamera osztály foglalja magában.

A ModelViewerCamera egy olyan kamerát modellez, amely az objektumok körül egy gömbhéj pontjaiban helyezkedhet el; azaz lényegében szabadon pozícionálható a jelenet körül, akár egy műhold a Föld körüli pályán. Innen ered a koncepció neve is: orbitális kamera.

A transzformációk szempontjából a leglényegesebb metódus a FrameMove, amely a kamera helyzetét frissíti a felhasználó interakcióknak megfelelően. A metódus váza a következő:

1. Ha az egérgombok valamelyike le van nyomva, lekérdezzük a pozícióváltozás nagyságát az utolsó hívás óta eltelt idő függvényében (ezt az időt a mozgásmennyiség modulálására használjuk, ilyen módon finomabb és folyamatosabb mozgáshatás érhető el kis változások esetén is).
2. Meghatározzuk a bekövetkezett mozgás irányát az UpdateVelocity metódussal.
3. A görgő mozgásával összhangban változtatjuk a kamera távolságát a modell középpontjától.
4. A kamera fókuszpontjának változtatása az egér mozgásának megfelelően. Az egérrel történő mozgatáshoz a program az ún. arcball koncepciót használja. Ez nem más, mint egy gömb, amely a modellt is magában foglalja. Az egér mozgatása során a mozgás kezdő- és végpontjaihoz (amelyek a képernyő pontjai) megkeressük ennek a gömbnek a megfelelő pontjait, és a gömb középpontjából a pontokba mutató vektorokat. Ezen vektorok vektoriális szorzata olyan vektort eredményez, amely mindkét vektorra merőleges. Ez a merőleges lesz a forgatás tengelye. A két előbbi vektor skaláris szorzata pedig a forgatás szögének koszinuszát adja. Az adatok felhasználásával a forgatáshoz tartozó Quaternion struktúrát állítunk elő.

   A ModelViewerCamera egy ilyen arcball segítségével kezeli a kamera forgatását; ebből származtatja a nézeti mátrixot is. Először is az arcball mátrixát invertáljuk, ugyanis az arcball forgatása az objektum forgatását vonja maga után, a kamera viszont ezzel ellentétesen fog mozogni. Ennek a mátrixnak a segítségével előállítjuk a kamera új fókuszpontját:

```
Vector3 localUp = new Vector3(0, 1, 0);
Vector3 localAhead = new Vector3(0, 0, 1);
Vector3 worldUp = Vector3.TransformCoordinate(localUp,
    cameraRotation);
Vector3 worldAhead = Vector3.TransformCoordinate(localAhead,
```

```
    cameraRotation);
// posDelta az egérmozgásnak megfelelő modulált elmozdulás
Vector3 posDeltaWorld = Vector3.TransformCoordinate(posDelta, cameraRotation);
lookAt += posDeltaWorld;
```

5. A kamera eltolása a gömbön levő megfelelő pozícióba:

```
eye = lookAt - worldAhead * radius;
```

6. Az új nézeti mátrix előállítása:

```
viewMatrix = Matrix.LookAtLH(eye, lookAt, worldUp);
Matrix invView = Matrix.Invert(viewMatrix);
invView.M41 = invView.M42 = invView.M43 = 0;
// a kamera az arcball forgatásához képest ellentétesen mozog
Matrix modelLastRotInv = Matrix.Invert(lastModelRotation);
```

7. Az új forgatási mátrix előállítása. A mátrix konstrukciója akkumulatív, az aktuális forgatási mátrixot a következő alkalommal fel fogjuk használni. A számítás a kamera saját nézeti terében történik:

```
// az elfordulást az arcball forgatási mátrixából származtatjuk
Matrix localModel = worldArcball.RotationMatrix;
// az elforgatás nézeti térben történik, ezért szükséges a vewMatrix és az invView tag
modelRotation *= viewMatrix * modelLastRotInv * localModel * invView;
lastCameraRotation = cameraRotation;
lastModelRotation = localModel;
```

8. Ortogonalizáció. A forgatási mátrix növekményes előállítása magában hordozza a hiba halmozódásának veszélyét, ebből adódóan a forgatási mátrix az objektum torzulását válthatja ki. Ezt elkerülendő biztosítjuk azt, hogy a mátrix felső 3 × 3-as főminorának sorvektorai ortonormált vektorok legyenek (ilyen módon a forgatás koordinátatengelyei is merőlegesek lesznek egymásra). A konkrét lépések (xBasis, yBasis, zBasis rendre a főminor első, második, és harmadik sorvektora):

   - xBasis normalizálása.
   - zBasis és xBasis vektoriális szorzataként előállítjuk yBasis-t.
   - yBasis normalizálása.
   - xBasis és yBasis vektoriális szorzataként kapjuk az új zBasis-t.

9. Az így kapott forgatási mátrix eltolása a kamera pozíciójába:

```
modelRotation.M41 = lookAt.X;
modelRotation.M42 = lookAt.Y;
modelRotation.M43 = lookAt.Z;
```

10. A jelenettér koordinátatranszformációját leíró world matrix előállítása az eltolási, skálázási és forgatási komponensekből. Az eltolás magában foglalja a modell középpontjának eltolását a jelenettér origójába—a kamera fókuszpontjába—(trans mátrix), illetve az egér segítségével történő eltolást (userTranslation):

$$world = trans \cdot scale \cdot modelRotation \cdot userTranslation.$$

A Transzformáció modul szintén a world mátrixot módosítja. A felhasználónak lehetősége van az eltolás, skálázás, forgatás egyenként történő módosítására.

A forgatást a DirectX háromféleképpen képes kezelni:

- Forgatási mátrix: A forgatást a szokásos módon, homogén transzformációs mátrixszal reprezentáljuk. A Matrix struktúra gyakorlatilag bármilyen forgatási mátrixot képes generálni.
- Quaternion: Ezzel a módszerrel közvetlenül az arcball forgatását leíró Quaternion módosul. Az API megteremti a mátrixos alak és a Quaternion alapú reprezentáció közötti átjárhatóságot.
- RPY (roll, pitch, yaw) szögekkel történő megadás: A megjelölés az aerodinamikában használatos elnevezésekből származik. A három forgatási szög gyakorlatilag a három koordinátatengely körüli forgatást adja meg. Először a z tengely körül forgatunk, majd az elforgatott, y' tengely körül, végül a kétszeres

forgatás után kapott x'' tengely körül. Mivel a forgatások szögét mindig az előző forgatás utáni koordinátarendszerben értelmezzük, a sorrend nem cserélhető fel. Az egyes forgatási lépések jelentése rendre: csavarás, billenés, fordulás. RPY-orientációs mátrixok szintén készíttethetők a Matrix struktúra RotationYawPitchRoll metódusával.

MÁTRIX ELEMZŐ

A háromdimenziós transzformációk nagy része lineáris transzformáció; azaz leírhatók egy 3 × 3-as mátrix segítségével. Ebből következően a transzformációk sorozata mátrixokkal való szorzások sorozatával is elvégezhető.

Az eltolás sajnos nem illik bele ebbe a mintába; az nem lineáris transzformáció. Ezzel együtt bármilyen háromdimenziós transzformáció leírható a következő alakban:

$$[x', y', z'] = [x, y, z] \cdot M + [t_x, t_y, t_z],$$

ahol M tetszőleges lineáris transzformáció (vagy ezek konkatenációja), $[t_x, t_y, t_z]$ az eltolás vektora. A mátrix-szorzást elvégezve (M_i az M i. oszlopvektora):

$$[x', y', z'] = [xM_1, yM_2, zM_3] + [t_x, t_y, t_z] = [xM_1 + t_x, yM_2 + t_y, zM_3 + t_z].$$

Ez ekvivalens egy négysoros mátrixszal való szorzással, ami azt jelenti, hogy ha a transzformációkat 4 × 4-es mátrixokkal írjuk le, akkor minden műveletet mátrix-szorzással lehetne kifejezni, beleértve az eltolást is.

A fenti leírás pontosan az affin transzformációk leírása (az affin transzformáció nem más, mint egy lineáris transzformációs mátrixszal való szorzás, amit egy eltolás követ).

Ha az affin transzformációk reprezentációjára ún. kibővített mátrixot használunk, a pontokat pedig homogén koordináták segítségével írjuk fel, a következőt kapjuk:

$$[x', y', z', 1] = [[x, y, z] \cdot A + [t_x, t_y, t_z], 1].$$

Arra jutottunk tehát, hogy minden háromdimenziós transzformáció leírható homogén transzformációs mátrixok segítségével.

Már láttuk korábban, hogy a DirectX-ben az alapvető transzformációk a forgatás, eltolás és a skálázás. Az API számára minden homogén transzformációs mátrix ezen komponensekből épül fel—mint ahogy erre már az animáció leírásánál utaltam.

A Mátrix elemző elsődleges célja, hogy megmutassa, hogy egy tetszőleges homogén transzformációs mátrix hogyan épül fel eltolási, skálázási, illetve forgatási komponensekből.

A transzformációk visszafejtését a Decomposer osztály végzi. Az osztály metódusait az animációs adatok beolvasásánál is felhasználjuk.

A visszafejtés általános M homogén transzformációs mátrixra működik. A szingularitás kiküszöbölése miatt az egyetlen feltétel, hogy a felső 3 × 3-as főminor determinánsának és a (4, 4) helyen levő elemnek a szorzata nemnulla legyen. Az UnMatrix metódus a mátrix elemei ismeretében a megfelelő eltolási és skálázási vektorokat (t, s), illetve a forgatást leíró Quaternion struktúrát (q) adja vissza.

Az algoritmus a visszafejtés során az egyes transzformációk hatásait „visszavonja", mégpedig fordított sorrendben.

Az eltolási komponens igen könnyen kinyerhető; a fentebb leírtak miatt $t = (M(4,1), M(4,2), M(4,3))$. Innentől kezdve a felső 3 × 3-as főminorral foglalkozunk, jelölje ezt M'.

A skálázási faktorok meghatározásával automatikusan az esetleges nyírás együtthatóit $(Sh_{xy}, Sh_{yz}, Sh_{xz})$ is meghatározzuk. Legyen először s_x az M' első sorvektorának hossza. Ezután az Sh_{xy} együttható értékét számítjuk ki: $Sh_{xy} = dot(M'_1, M'_2)$. Legyen az M' második sora ezek után $M'_2 = M'_2 - Sh_{xy}M'_1$. Az így kapott sorvektor hossza lesz s_y. A számítás további menete ezekhez a lépésekhez hasonló:

$$Sh_{xz} = dot(M'_1, M'_3)$$
$$M'_3 = M'_3 - Sh_{xz}M'_0,$$
$$Sh_{yz} = dot(M'_2, M'_3)$$
$$M'_3 = M'_3 - Sh_{yz}M'_2,$$

$s_z = Length(M'_3)$.

Ezek után az M' egy forgatási mátrix lesz, egy esetleges -1-es skálázási faktortól eltekintve. Amennyiben az M' determinánsa negatív, a mátrix minden elemét -1-gyel szorozzuk. Az így kapott mátrixból a forgatási Quaternion egyszerűen generáltatható.

A kiszámított komponenseket a Mátrix elemző grafikus formában jeleníti meg. A forgatási komponenshez a mátrixos és a Quaternion reprezentációt is megjelenítjük.

A Mátrix elemző egyrészt a Transzformáció panellel, másrészt a forrásállományt megjelenítő nézetekkel van kapcsolatban. Utóbbiak esetén bármelyik megjelenő mátrix közvetlenül komponensekre bontható a megfelelő menüpont kiválasztásával.

FORRÁSHIERARCHIA

A Forráshierarchia panel a forrásállományban levő template-hierarchiát jeleníti meg.

A panel minden template-et megjelenít, függetlenül attól, hogy tartalmaz-e a beolvasó által feldolgozható adatokat vagy sem. Template-definíciók esetén a hierarchiában a template neve, adatcsomópont esetén az opcionális név, ennek hiányában egy generált név jelenik meg.

Ennek a modulnak a haszna a forrás nézettel, illetve a diagramos nézettel való kapcsolat. A néven történő dupla kattintással ugyanis a megfelelő (akár mindkét) nézetben láthatóvá válik az adott template kódja, illetve részletei.

BEOLVASÓ ÜZENETEI

A panel a DirectX állomány beolvasása során talált esetleges kódszintű, szintaktikai hibákra hívja fel a figyelmet. A visszajelzés kétféle lehet:

- Figyelmeztetés: A talált hiba nem befolyásolja a beolvasás kimenetelét. Ilyen lehet például egy elválasztójel hiánya.
- Hiba: A rendellenesség hatására a beolvasás nem fejezhető be. Például kevesebb adat áll rendelkezésre, mint amennyit a template definíciója deklarált.

A program minden esetben igyekszik a beolvasást befejezni. Végzetes hiba esetén, ha elegendő adat állt rendelkezésre, a megjelenítés nem lesz letiltva. A hiba helyéig rendelkezésre álló template-hierarchia szintén elérhető és teljesen használható.

Bármelyik üzenetre történő dupla kattintásra a hiba helye a forrásnézetben láthatóvá válik.

ANIMÁCIÓ

A panel a korábban már bemutatott, a beolvasás során feltöltött AnimationContainer osztályban tárolt AnimationSet mozgássorozataihoz nyújt alapvető kezelő funkciókat:

- A lejátszás leállítása, újraindítása.
- A lejátszás menetének grafikus kijelzése.
- A lejátszás sebességének (képkocka / másodperc) változtatása.
- Több AnimationSet esetén ezek neveinek listázása (illetve generált nevek megjelenítése), a kiválasztott AnimationSet betöltése igény szerint.

Ezeken kívül a panel a legfontosabb információkat is megjeleníti:

- A mozgássorozat teljes hossza (képkockákban és másodpercben)
- Az aktuális pozíció (képkockákban és másodpercben)
- Az animált csontok száma

PROGRAMOZÓI TÁMOGATÁS

A program egyik célja a DirectX-szel ismerkedő programozók támogatása. Ezt a célt szolgálja a szoftver szemléltető eszközeinek második szintje, amely két eszközcsoportot tartalmaz:

1. A Kódgenerátor eszköz: képes az aktuális beállítások felhasználásával azt a C#-kódot előállítani, amellyel a megfelelő állapot a fixed function pipeline programozásakor reprodukálható. A kódrészletek az alábbi beállításokra vonatkoznak:

   - Megvilágítás.
   - Vetítés.
   - Nézet, virtuális kamera.
   - Transzformáció.

   Másrészt rendelkezésre bocsátja a program által használt shader programok kódjait, amelyekben a fontos számítások, az alkalmazandó módszerek könnyen láthatók:

   - Fényforrások.
   - Phong-Blinn árnyalás.
   - Alapvető vertex, illetve pixel shader mintákon bemutatja az árnyalók felépítését.
   - Bőrözés, vertex blending.

   Ezeken kívül lehetőség van kisebb hasznos kódrészletek generálására, amelyek bizonyos általános mintákatadnak meg a DirectX alkalmazása során felmerülő részfeladatok megoldására, például:

   - Frame hierarchia megjelenítése, a transzformációk öröklődésének kezelése.
   - Effect-ek használata.

2. Fontos, hogy a DirectX működésével ismerkedő programozók ismerjék meg a megjelenítési futószalag működését is. Ezt segítik a modulokhoz hasonló, a futószalag bizonyos lépéseit szemléltető panelek. Ezek a fixed function futószalag képességeire és az adott lépéseknél elérhető render state beállításokra épülnek. Az egyes funkciók különálló Effect példányokat használnak, amelyeket igény szerint töltenek be. A shader programok működése az adott lépésnél leírtak alapján történik. Ezek közül egyszerre csak egy lehet aktív. A panelek a futószalag következő lépéseit mutatják be:

   - Hátsó lap eltávolítás (culling): A vertex shader összegyűjti egy lap két élvektorát, majd kiszámítja ezekből az él felületi normálisát. A nézőpontba mutató vektor és ennek szöge alapján bizonyos vertexeket eldobhatunk.
   - Viewport leképezés: Lehetőség van a sarokkoordináták és a méretek, valamint a mélységi tartomány változtatására. A beállításoknak megfelelően a vertex shader megkapja a leképezésnek megfelelő mátrixot, amelyet a vertexek transzformációjakor felhasznál. A kívül eső vertexeket eldobjuk.
   - Vágás felhasználói vágósíkokkal: a DirectX szabványt követve hat vágósík adható meg, amelyek $[a, b, c, d]$ paraméterei ismeretében a vertex shader az $ax + by + cz + d < 0$ tesztelést hajtja végre az egyes síkokra. Ennek teljesülése esetén a vertexet eldobjuk.
   - Köd-effektusok: Lehetőség van vertex-, és pixelszintű köd megadására is, a számítások helye ennek függvényében változik.
   - Alpha blending: A számításokban alapértelmezés szerint a forráspixelek színei az objektum pixeleiből, a célpixelek színei az aktuális anyaghoz megadott textúrából származnak. Ennek hiányában megadható egy „pszeudo” színérték, amely egy egyszínű textúrát imitál.
   - Írási maszkok alkalmazása: A három maszk közül a ColorWriteEnable értékei változtathatók.

- Alpha teszt: A pixel shader a referencia értékkel történő összehasonlítás alapján dobja el, illetve tartja meg a pixeleket.

SÚGÓ

Az oktatást támogató eszközök harmadik szintje a program súgója, amelynek elsődleges célja az elméleti alapok feltárása. A Súgó témakörei a következők:

- A DirectX alapjai, a megjelenítési futószalag lépései. A fontosabb lépések leírásánál a súgó megadja az adott terület által érintett render state beállításokat is.
- A program használata.
- DirectX állományok.

EGYÉB ESZKÖZÖK

A negyedik csoportba két eszköz tartozik:

- A Nyomkövető célja, hogy a különböző területeken előforduló homogén mátrixokban felfedje az egyes elemek szerepét. Segítségével egy adott mátrix változásai valós időben követhetők. Az eszköz a Transzformáció és a Megjelenítés panelről aktiválható.
- A Device tulajdonságok segítségével egyrészt az aktuálisan használt Device példány paraméterei válnak láthatóvá és változtathatóvá. Az eszköz lehetővé teszi az elülső és a hátulsó puffer formátumának, valamint a mélységi és stencil puffer formátumának változtatását, illetve a mélységi puffer használatának engedélyezését és letiltását. A kiválasztott formátumok tesztelésére is lehetőség van, ehhez a program a Manager osztály bemutatott metódusait használja. Amennyiben a kiválasztott formátumok kompaitbilisek, a változtatások érvénybe lépnek.

  Az eszköz szintén lehetővé teszi a hardveres tulajdonságok (a DeviceCaps tulajdonság elemeinek) megtekintését.

ÖSSZEFOGLALÁS

Az előbbiekben bemutattam a DirectX API-t mint grafikai programozási eszközrendszert. Áttekintettem, milyen alapvető képességei vannak. Konkrét célul tűztem ki egy olyan szoftver elkészítését, amely ezeket a képességeket mutatja be és programozásukat támogatja.

A dolgozatban bemutatott program több eszközzel is szolgálja a DirectX (és általánosan a háromdimenziós grafika) alapjainak elsajátítását:

Az egyes funkciókat, fontosabb területeket, illetve ezek implementációit a DirectX-ben ún. modulok segítségével mutattam be. A modulok első csoportja a megjelenítést közvetlenül befolyásoló területekre koncentrál. Ezek konkrétan a következők:

- Nézet, virtuális kamera: A modul teljesen lefedi a DirectX-ben elérhető megjelenítési módokat. A nézeti gúla vizualizációja nagyban segítheti a vetítés alapjainak megértését.
- Megvilágítás, árnyalás: A program a DirectX által támogatott valamennyi megvilágítási modellt támogatja, a Direct3D árnyalási módjai mellett a Phong árnyalást is bemutatja.
- Animáció: Az API által támogatott kulcskocka alapú animációk teljes kezelése megoldott. A beolvasó több AnimationSet template jelenlétére is fel van készítve, az API beépített metódusai ezt nem támogatják.
- Anyagtulajdonságok: A modul segítségével az anyagok színérzetre, és a megvilágításra gyakorolt hatása válik láthatóvá.

A modulok következő csoportja a DirectX állományt mutatja be:

- Forráshierarchia: Az állományban levő template-hierarchia megjelenítése.
- Beolvasó üzenetei: A beolvasás kimeneteléről ad tákékoztatást.

A harmadik csoport a DirectX-ben oly fontos homogén transzformációs mátrixok koncepciójára, használatára koncentrál:

- Transzformációk: A felhasználó közvetlenül megtekintheti a forgatás, eltolás, skálázás hatását. A változások valós időben követhetők a Nyomkövető eszközzel.
- Mátrix elemző: Megmutatja, hogy egy homogén transzformációs mátrix hogyan értelmezhető; milyen kapcsolatban áll a külön-külön vett forgatás, skálázás, eltolás műveletével.

Szintén a DirectX állományra koncentrál a Forrás nézet, amely a beolvasott állomány szövegét jeleníti meg olvashatóbb formában, míg a Diagram nézet elsősorban az állományban tárolt adatokat és azok rendeltetését igyekszik megmutatni—az összefüggések megértése érdekében a két nézet egyszerre is megjeleníthető, és mindkét nézet össze van kapcsolva a Forráshierarchia modullal.

A program képes hatékonyan segíteni a grafikai alapok elsajátítása mellett a tényleges Managed DirectX programozást is. Ezt egyrészt a Kódgenerátor nevű eszköz támogatja, amely a modulokban beállított értékek felhasználásával C#-kódot generál. Másrészt a program szemlélteti a megjelenítési futószalag összes olyan lépését, amely vizuálíisan bemutatható. Ezek a lépések: köd-effektusok, alpha blending, alpha teszt, vágás felhasználói vágósíkokkal, viewport leképezés és scissor test, maszkolás.

A dolgozatban bemutatott szoftver tehát—a dolgozat elején kitűzött kettős célt szem előtt tartva—az alapvető elméleti grafikai ismereteket is szemléltető, illetve konkrétan a Managed DirectX programozását is bemutató komplex eszközrendszerrel rendelkezik. Ez az eszközrendszer teljesnek tekinthető, hiszen a DirectX megjelenítési futószalag valamennyi olyan lépését demonstrálja, amely egyszerűen vizualizálható shader programok segítségével, a DirectX specifikáció által megadott funkciókat pedig teljes mértékben támogatják. Ezen kívül számos kényelmi funkció, egyszerűen használható eszköz szolgálja a fontos területek megértését. Mindezek miatt a szoftver a vele szemben támasztott követelményeknek megfelel. A kitűzött célokat lényegében elértem.

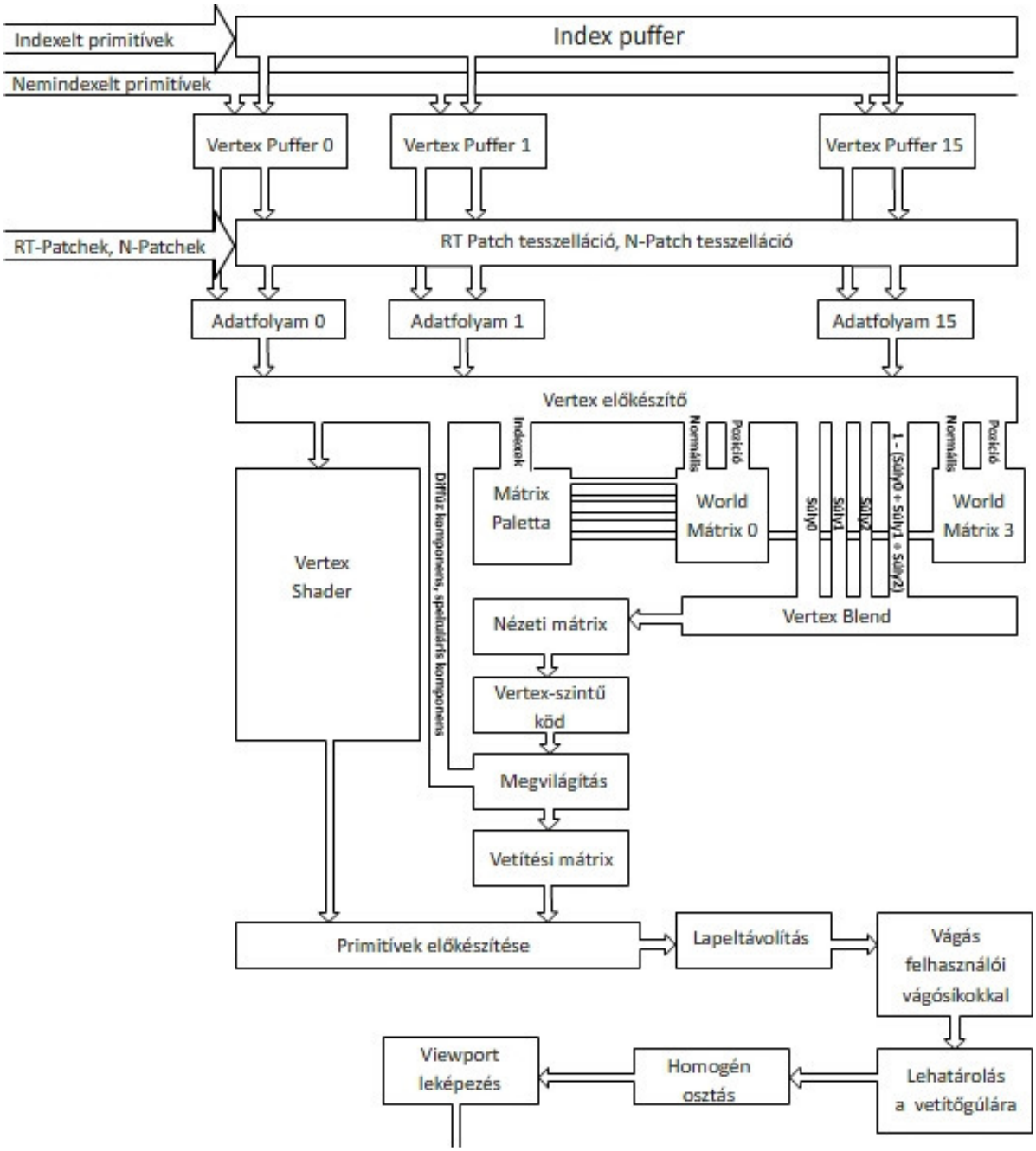
IRODALOMJEGYZÉK

1. Arvo, J. (szerk.): Graphic Gems vol. 2
   Academic Press, 1991
2. Chen, J.: Computational Geometry: Methods and Applications
   Előadás-vázlat; elérhető a
   http://faculty.cs.tamu.edu/chen/notes/geo.pdf címen.
   A hivatkozás ellenőrizve: 2008. 11. 05.
3. Dam, E. B. – Koch, M. – Lillholm, M.: Quaternions, Interpolation and Animation
   Publikáció; elérhető a
   http://www.itu.dk/people/erikdam/DOWNLOAD/98-5.pdf címen.
   A hivatkozás ellenőrizve: 2008. 11. 05.
4. Glassner, A. (szerk.): Graphic Gems vol. 1
   Academic Press, 1990
5. Gray, K.: Microsoft DirectX 9 Programmable Graphics Pipeline
   Microsoft Press, 2003
6. Jin, S. – Lewis, R. L. – West, R.: A Comparison of Algorithms for Vertex Normal Computation
   Publikáció; elérhető a
   http://www.tricity.wsu.edu/~bobl/personal/mypubs/2003_vertnorm_tvc.pdf címen.
   A hivatkozás ellenőrizve: 2008. 11. 05.
7. Kirk, D. (szerk.): Graphic Gems vol. 3
   Academic Press, 1992
8. Kovach, P.J.: Inside Direct3D
   Microsoft Press, 2000
9. Leiterman, J. C.: Learn Vertex and Pixel Shader Programming with DirectX 9
   Wordware Publishing, 2004
10. Luna, F.: Skinned Mesh Character Animation with Direct3D 9.0c
    Publikáció; elérhető a
    http://mathinfo.ens.univ-reims.fr/image/dxMesh/extra/d3dx_skinnedmesh.pdf címen.
    A hivatkozás ellenőrizve: 2008. 11. 05.
11. Miller, T.: Managed DirectX 9 Kick Start: Graphics and Game Programming
    Sams Publishing, 2003
12. Peeper, C. – Mitchell, J.C.: Introduction to the DirectX 9 High Level Shading Language
    Publikáció; elérhető a
    http://ati.amd.com/developer/ShaderX2_IntroductionToHLSL.pdf címen.
    A hivatkozás ellenőrizve: 2008. 11. 05.
13. Sanchez, J. – Canton, M. P.: The PC Graphics Handbook: A Reference Manual on PC Graphics Hardware and Software
    CRC Press, 2003.
14. Sanchez, J. – Canton, M. P.: DirectX 3D Graphics Programming Bible
    IDG Books Worldwide
15. Sander, P. V.: A Fixed Function Shader in HLSL
    Publikáció; elérhető a
    http://www2.ati.com/misc/samples/dx9/FixedFuncShader.pdf címen.
    A hivatkozás ellenőrizve: 2008. 11. 05.
16. Shoemake, K. – Duff, T.: Matrix Animation and Polar Decomposition
    Publikáció; megjelenés helye:
    Proceedings of the conference on Graphics interface '92, 258-264. o.

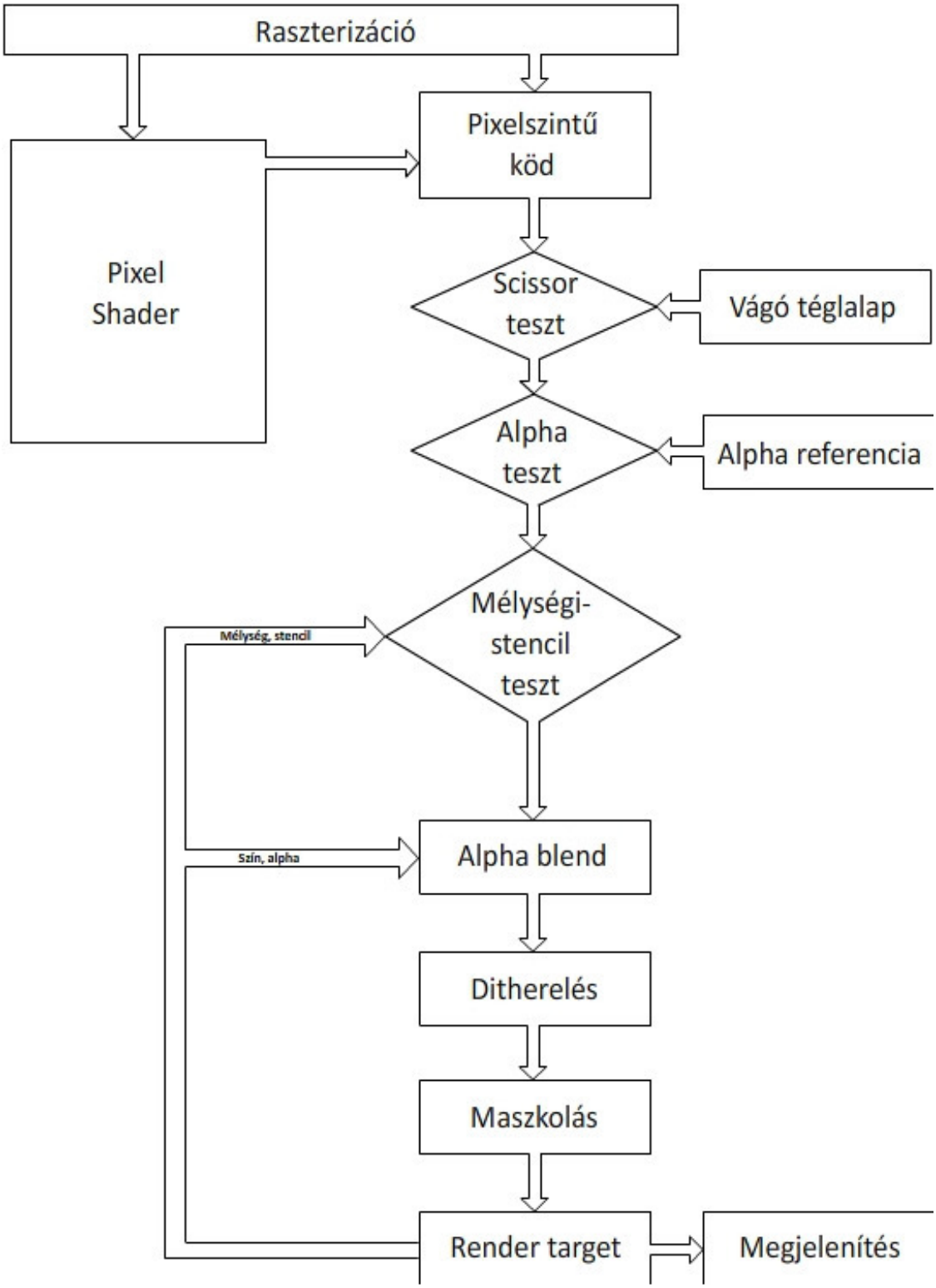
17. Szirmay-Kalos L. – Antal Gy. – Csonka F.: Háromdimenziós grafika, animáció és játékfejlesztés ComputerBooks, 2005
18. Thomson, R.: The Direct3D Graphics Pipeline
Kézirat; elérhető a http://www.xmission.com/~legalize/book/download/index.html címen. A hivatkozás ellenőrizve: 2008. 11. 05.
19. van Waveren, J. M. P.: From Quaternion to Matrix and Back
Publikáció; elérhető a
http://www.intel.com/cd/ids/developer/asmo-na/eng/293748.htm
címen. A hivatkozás ellenőrizve: 2008. 11. 05.
20. DirectX SDK (August 2005) Managed
A Microsoft hivatalos dokumentációja a
http://msdn.microsoft.com/en-us/library/aa139769.aspx címen.
A hivatkozás ellenőrizve: 2008. 11. 05.
21. X File Format Reference
A Microsoft hivatalos dokumentációja a
http://msdn.microsoft.com/en-us/library/bb173014(VS.85).aspx címen.
A hivatkozás ellenőrizve: 2008. 11. 05.
22. Reference for HLSL A Microsoft hivatalos dokumentációja a
http://msdn.microsoft.com/en-us/library/bb509638(VS.85).aspx címen.
A hivatkozás ellenőrizve: 2008. 11. 05.

FÜGGELÉK

A DIRECTX VERTEXFELDOLGOZÓ LÉPÉSEI



A DIRECTX PIXELFELDOLGOZÓ LÉPÉSEI



KÖSZÖNETNYILVÁNÍTÁS

A dolgozat készítője köszönetét fejezi ki dr Kovács Emődnek, a szakmai segítségért és iránymutatásért, valamint a hasznos tanácsokért, amelyekkel hozzájárult mind a szoftver, mind az elméleti rész elkészüléséhez.