Eötvös Lóránd Tudományegyetem

Informatikai kar

Algoritmusok és alkalmazásaik tanszék

Háromdimenziós, űrben játszódó akciójáték XNA környezetben

Témavezető
Valasek Gábor

Doktorandusz
Msc

Szerző
Thier Richárd István

nappali tagozatos
programtervező informatikus Bsc hallgató

2010. január. 17

Tartalomjegyzék

1. Bevezetés

A szakdolgozat célja egy TIE Fighter Forever című háromdimenziós űrben játszódó számítógépes akciójáték játszható demó verziójának megvalósítása az XNA 4.0 framework segítségével. A játék itt bemutatott demó verziója hagyományos PC-n fog futni Microsoft Windows XP(vagy annál újabb) operációs rendszeren, de a fejlesztés során tekintettel leszünk a továbbfejleszthetőségre és az XNA által támogatott másik fő platform, az XBOX 360 sajátosságaira a könnyű portolhatóság megőrzésével.

A dolgozat fő célkitűzése ezen felül lényegében, a háromdimenziós játékfejlesztésben felmerülő tipikus, illetve alapvető technikák, effektusok és problémakörök bemutatása és feloldása, egy konkrét játék kifejlesztése által.

1.1 A megoldandó feladatról

A játék egy belső nézetes, háromdimenziós akció-űrjáték így ennek megfelelően kell a grafikai, játékmenetbeli, illetve irányítási megoldásokat megvalósítani. A játék során elő fognak kerülni grafikai, színtérmenedzsment-béli, irányítási, és térbeli elmozdulások eredményeként ütközések miatt előálló problémák. A grafikai effektusok megvalósításakor és a megjelenítés egyéb hatásainak megvalósításakor a legtöbbször arra fogunk törekedni, hogy hardveresen gyorsított formában támogassuk az adott technikát. A fejlesztés során ezen felül arra is tekintettel leszünk, hogy a játék a megjelenése ellenére elfogadható sebességgel fusson a nem csúcskategóriás számítógépeken is. Ezen felül szem előtt kell tartanunk a program továbbfejleszthetőségi potenciálját is, mind játékmechanizmusban, grafikában, mind más platformok irányában. Az XBOX rendszerre történő portolás lehetőségét az ott nem létező(vagy rosszul támogatott) lehetőségek mellőzésével és rugalmas irányítási osztályokkal valósítjuk meg, az XNA által a 4.0 verzió óta támogatott Windows Phone-t pedig úgy támogatjuk, hogy a fejlesztés során a REACH beállításokkal fordítunk és csak a REACH-ben is elérhető lehetőségeket használjuk ki. Utóbbi kikötés mellett elméletileg a program Windows phone-ra történő portolása is lehetségessé válik, bár sajnos még így is sok nehézséggel jár a portolás, mert bár a phone tud szinte mindent, amit egy REACH profillal fordítva megvalósíthatunk, sajnos nem támogatja a custom shadereket, melyeket nagyon is sokszor használunk a megvalósításunkban, így phone-ra sajnos csak egy lebutított verzió portolható át a programból. A Windows phone platform kissé előnytelenebb helyzetének az ellenére viszont kijelenthető, hogy a program elvileg könnyen fejleszthető és kiegészíthető az XNA két fő csapásvonala, a PC és az XBOX-360 játékkonzol tekintetében.

Mivel játszható játék elkészítésére törekszünk nem elégszünk meg csupán grafikai és ütközésvizsgálati megoldások bemutatásával, hanem egy egyszerű, tényleges játékmenet is megvalósításra kerül. Mivel belső nézetes háromdimenziós játék esetén az egy gépen több játékos által egymás ellen játszható játékok megvalósíthatósága megkérdőjelezhető és mivel egy hálózaton keresztül játszható játék megvalósítása nem férne már bele a szakdolgozatba, kézenfekvő megoldás, hogy a játékban gépi ellenfelek legyenek jelen. A dolgozat nem tűzi ki maga elé célul egy bonyolult és jól működő mesterséges intelligencia rendszer implementálását is, úgy döntöttem, hogy egy egyszerűbb, mesterséges intelligenciát implementálok, de a programozás során meghagyjuk a lehetőséget a fejleszthetőségre ebben a tekintetben is, méghozzá úgy,

hogy a mesterséges intelligencia nem az adott osztályokba(pl. Űrhajó) lesz bedrótozva, hanem külön osztályrendszert alkot mely hivatkozásokon keresztül mozgathatja és változtathatja az objektumok és űrhajók állapotát amolyan dinamikus hozzárendelések segítségével. Ez a rendszer ebben a formában már könnyen kiterjeszthető, mivel egy másik objektum bevezetésével(virtuális tábornok), mely a játéktér állapotát időnként megfigyeli és egy virtuális katonai hierarchia kiépítésével, melynek legvégén a fent említett módosító jellegű osztályok vannak, nagy bonyolultságú mesterséges intelligenciával is felvértezhető a program egy lehetséges továbbfejlesztése.

1.2 A bemutatott technikák rövid felsorolása

- Buckaleképezési technika a nagyobb hajók és állomások számára normálvektor térképezéssel.
- Környezetleképezési technika a vadászgépek tükröződésének megvalósítására cubemap textúrával.
- Content pipeline kiterjesztések megvalósítása a művészek által előállított háromdimenziós űrhajómodellek, illetve textúrák a játékba történő helyes beolvasásának érdekében.
- Wrapper osztályok leprogramozása az irányítás felett, hogy a program könnyen portolható legyen az xbox konzolra, mely egyedi kontrollerrel rendelkezik. Ezen kívül az irányítást úgy tervezzük meg, hogy ne kelljen túl sok gombot használni a funkciók eléréséhez, mert a konzol kontrollerének a gombszáma igen limitált.
- A játék jól elkülöníthető részeinek komponensekre bontása az XNA framework által biztosított GameComponent osztály segítségével.
- Egyszerű színtérmenedzsment a játék menüje és a tényleges játék szinte teljes elkülönítésével.
- Skybox technika alkalmazása az űrbeli környezet megteremtése érdekében.
- Egy postprocessing technika megvalósítása és bemutatása: selective laserglow.
- Egy egyszerű, de kiterjeszthető részecskerendszer megvalósítása robbanásokhoz.
- Térbeli ütközések megvalósítása nagyságrendben is különböző méretű objektumok között térparticionálási fával és befoglaló hierarchiákkal.
- Egyszerű, de kiterjeszthető mesterséges intelligencia.
- Űrbéli objektumok forgatásának reprezentációja kvaterniók segítségével.

1.3 A megvalósításhoz felhasznált környezet rövid bemutatása

1.3.1 A program megírása során felhasznált szoftverkörnyezet és az XNA keretrendszer rövid leírása.

A programot C# programozási nyelven, a Visual Studio 2010 Ultimate fejlesztőkörnyezetben, az XNA grafikai framework 4.0 változatával valósítjuk meg a REACH profil használatával. A fejlesztés során a Visual Studio terminológiáját

alkalmazva a teljes programot **Solution**-nak(„megoldás”), azon belül pedig az egyes jól elkülöníthető és a fordítás szempontjából külön egységet képező alegységeit **Project**-nek(„projekt”) nevezzük.

A fejlesztés során fel fogjuk használni a framework előre elkészített osztályrendszerét, mint például a háromdimenziós matematikai segédfüggvények és osztályok, tartalmak kezelésére szolgáló osztályok, **Game** típusú játékosztályok, illetve **GameComponent** és **DrawableGameComponent** osztályok. Utóbbiak különösen fontosak, mert a fő játékosztályunk **Game** osztályból származtatásával egy komponensekkel kiegészíthető előre elkészített rendszert kapunk, a képernyők **DrawableGameComponent**-ből való származtatásával pedig képernyőnként egy, a fő játékosztályhoz hasonló környezetet kapunk, miközben a programot logikailag, képernyőnként elkülönülő részekre bonthatjuk. A játék komponenseket ezen felül használjuk az irányítási rendszer kialakítása során és a játék egyes, egyéb okok miatt jól elkülöníthető és újrahasznosítható részeinek a megvalósításakor (bár utóbbi esetben nem fogjuk kihasználni a keretrendszer beépített függvényhívási mechanizmusait).

1.3.1.1 Egy újonnan létrehozott XNA solution projektfájljai, hívási automatikái és a kiterjeszthetőség.

Mikor a Visual Studio-ban egy új **XNA Windows Game** típusú solution-t hozunk létre, megkapjuk a mi programunk által is használt projektek és osztályok egy részhalmazát, az alapértelmezett hívási automatikával és a **Game** osztályból öröklődő fő játékosztályunkkal.

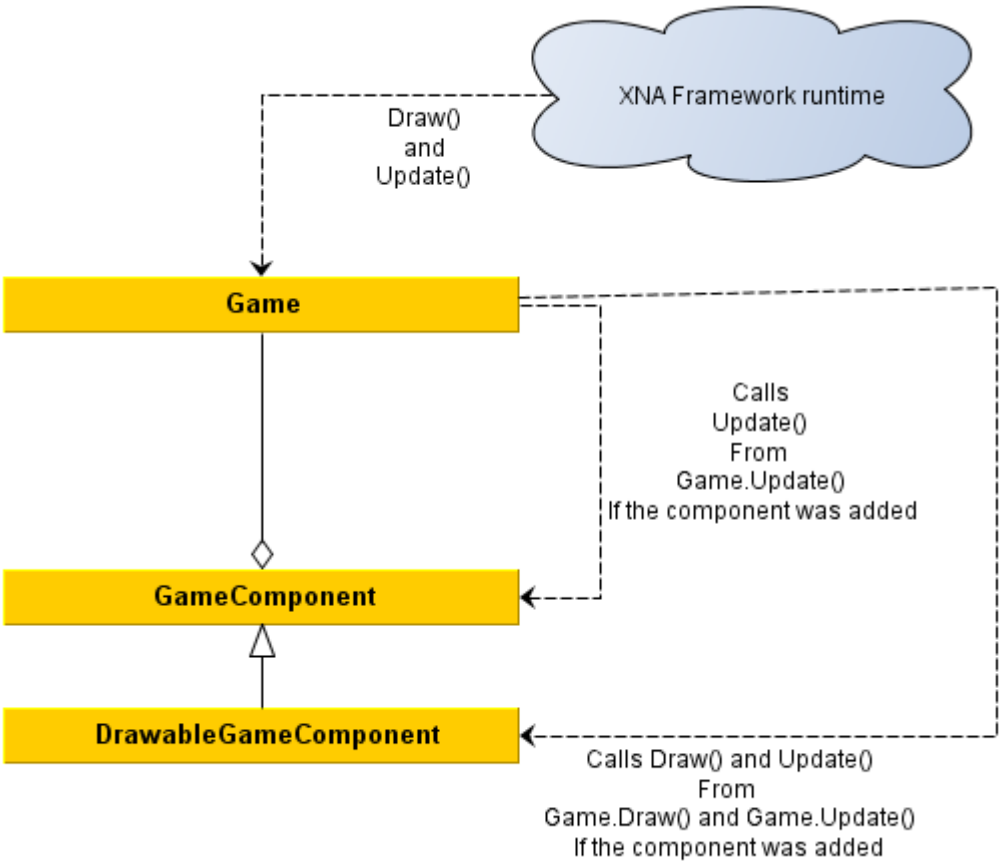
Két projektet hoz létre számunkra a rendszer. Egyet a játékunk fő osztályával és a belépési ponttal, egyszóval azon forráskódokkal melyek a célgépen történő futást határozzák meg és melyekből a végleges build a binárist előállítja, illetve a majdnem azonos névvel ellátott, de content kiegészítéssel ellátott, a játék során felhasznált tartalmakat tároló projekt, melynek elemeit a (később részletesebben leírt működésű) content pipeline fordítási időben futó transzformáló programjai alakítanak át a program által futási időben történő egyszerű felhasználásához/betöltéséhez.

Az XNA általunk is felhasznált **Game** osztálya, annak **Run()** metódusának meghívását követően beépített automatika alapján hívja meg a felüldefiniálható **Draw()** és **Update()** tagfüggvényeit úgy, hogy a játékállapot frissítésére vonatkozó kódot tartalmazó **Update()** metódus (alapesetben és gyors futás mellett) másodpercenként 60-szor lefusson a **Draw()**, kirajzolási műveleteket tartalmazó függvénnyel egyetemben. Amennyiben a kirajzolási, vagy a frissítési művelet túl hosszú, esetleg túl rövid, akkor a keretrendszer a játék objektum **isFixedTimeStep** propertyjének az értékétől függően megpróbálja biztosítani a frissítés 60-szor történő meghívását akár a kirajzolás meghívásának a hátrányára (ez az alapeset, ha be van állítva a property és mi is ezt használjuk), vagy egyszerűen ignorálja azt, hogy a program túl gyors vagy lassú ütemben zajlik.

Ezen kívül az XNA keretrendszer másik fő, általunk is felhasznált tulajdonsága, a játékkomponensek hívási automatikája. Az XNA framework használatakor ugyanis alapvetően az a szemlélet az uralkodó, hogy a fenti tulajdonsággal rendelkező fő játékosztály csupán a legalapvetőbb kódokat kell, hogy tartalmazza, míg a játék állapotának módosítását, illetve a kirajzolás nagy részét, a játék objektumhoz hozzáadható játékkomponensek valósítják meg.

Ez az elgondolás azon alapszik, hogy az XNA keretrendszer által meghívott fő játékobjektumbeli **Update()** és **Draw()** metódusok meghívják az eredeti, az XNA által biztosított **Game** ősosztály megfelelő metódusait, melyek végigiterálnak a játékosztály **Components** nevű **GameComponentCollection** típussal rendelkező komponens gyűjteményén, meghívva minden egyes, a gyűjteményhez hozzáadott komponens megfelelő metódusait úgy, hogy alapbeállítás szerint a komponensek eredeti hozzáadásának a sorrendjében hívják meg az egyes függvényeket (bár ezt módosíthatjuk az U**pdateOrder,** illetve **DrawOrder** property-k értékével). Mi ezt az automatikát csupán a játék egyes képernyőinek komponensekké alakításához, illetve az irányítási rendszer bárhonnan történő hozzáférhetőségéhez fogjuk felhasználni, de tekintettel leszünk arra is, hogy a nagy mértékben hordozható komponenseinket ne kelljen túlságosan átalakítani akkor, ha azokat egy másik solution-ben szeretnénk felhasználni, mint hozzáadott komponens.

A keretrendszer hívási automatikája, a benne szereplő, általunk is felhasznált ősosztályok és a „game loop” ciklus alapvető működése a következő ábráról is leolvasható:



1.3.2 A megvalósítás során felhasznált háromdimenziós videomegjelenítő rendszer architektúrája

A program megvalósítása során fel fogjuk használni az XNA által támogatott DirectX9.0 rendszer (illetve az ahhoz programozás tekintetében sokban hasonlító

XBox-os megjelenítő rendszer) architekturális sajátosságaiból következő hardveres gyorsítási lehetőségeket, a modern háromdimenziós megjelenítő rendszer programozható futószalagjának tulajdonságait, így ezek bevezető jellegű gyors áttekintése szükséges a dolgozatban leírtak megértéséhez.

1.3.2.1 A grafikus szerelőszalag szerepe

A mai megjelenítőkörnyezetekben, a háromdimenziós kirajzolás (renderelés) egy külön hardveresen gyorsított célhardver segítségével van megvalósítva, mely tulajdonképpen egy több állomásból felépülő futószalag, mely a bemeneti inputsorozatból előállítja a végleges raszterképet. Ez a futószalag régebben csupán egy néhány helyen paraméterezhető, állomásonként előre beégetett adattranszformációkat végző csővezeték volt, manapság pedig egy (a jelenlegi tendenciák szerint egyre több ponton) programozható szerelőszalag, melyben az egyes feldolgozó egységek paraméterezését, vagy programját adhatjuk meg.

A szerelőszalag programozható állomásain futó programokat a manapság általánosan használt és elfogadott terminológia shader programoknak nevezi. Ezek a programok a grafikai rendszert befoglaló host gépen fordításra, majd a grafikai alrendszer számára kiküldésre kerülnek a megjelenítés előtt, illetve közben, amennyiben az ott lévő programokat le kell cserélni másféle objektumokhoz felhasznált shader programokkal.

A shader programok tehát ténylegesen a videokártyán futó programok, melyek a z általunk használt architektúra esetén vertexenként (használt terminológia: **vertex shader**), illetve az egyes pixelek előállításához paraméterként szükséges ún. **fragment**enként (használt terminológia: **pixel shader, vagy fragment shader**) futnak le.

A grafikus csővezeték esetünkben, csak ezzel a két teljesen programozható állomással rendelkezik, de ezen kívül lehetőségünk van az összeszerelés paraméterezésére több ponton is, illetve vethetünk be trükköket az input adatok manipulálásával is.

1.3.2.2 A grafikus szerelőszalag felépítése, számunkra jelentős állomásai, a bemeneti és kimeneti pontok, illetve az adatok áramlása

A renderelési, menet (melynek célja, egy valamilyen célpontba kerülő raszterkép kialakítása) a host rendszer által kezdeményezett (valamilyen értelemben vett) draw utasítás kiküldésével kezdődik meg, melynek során megadjuk a megjelenítő rendszer számára fontos információkat (úgy mint a vertex deklaráció, ami a bemeneti adatfolyam struktúrájának a felosztását írja le, illetve például a kirajzolás indexelt vagy nem indexelt mivolta), melyet megelőz a grafikus megjelenítő eszköz(XNA-ban: **GraphicsDevice**) paraméterezése, amikor is kiküldésre kerülnek a megjelenítéshez szükséges áttranszformálandó bemeneti értékek, ami belső adatpufferből történő rajzoláskor nem történik meg, majd megkezdődik a szerelőszalagon az adatok átáramlása.

A szerelőszalag legfontosabb bemenetét a térbeli pontok, az ún. vertexek alkotják, melyekhez a térbeli koordinátáikon túl egyéb adatok(pl. használt textúra kifeszítési koordinátái, normálvektorok, saját adat, stb.) is hozzárendelhetőek. Ezek a host rendszer által kerülnek kiküldésre a megjelenítő alrendszer számára, vagy a belső és gyors memóriapufferekben előkészített tárhelyről olvasódnak be az ún. **vertex csatornák**ba,

ahonnan a vertex shader segítségével leírható programozott állomás bemeneteként olvashatóak.

A **vertex shader** speciális regiszterek formájában képes olvasni a bemeneti adatot a csatornákból, és még a host program által, a kirajzolás kezdeményezése előtt beállított, a renderelési menet folyamán változatlan ún. **shader konstansok** is átküldésre kerülnek a konstans regisztereken keresztül.

A **vertex shader** a kapott bemeneteinek és a belső programjának segítségével elvégzi a térbeli sokszögek egyes pontjaihoz tartozó adatok transzformációját, általában elvégezve a sokszög az ő magasabb szintű objektumához (pl. egy háromdimenziós alakzat) tartozó relatív koordináta-rendszeréből az ún. abszolút világtérbe való áttranszformálását a beállított világ-mátrix segítségével, az így kapott koordináta átalakítását a kamera által meghatározott koordináta-rendszerbe az ún. nézeti mátrix-szal történő transzformáció segítségével, majd végül az eredmény síkba történő projektálását a projekciós mátrix segítségével. Ezeken a lépéseken kívül szokás még átbocsátani a textúrakoordinátákat és a színértékeket, meg mindegy egyéb, a későbbi lépések számára bemenetként szolgáló adatot is.

Megj.: Természetesen mivel a vertex shader lépés teljes mértékben programozható, másféleképpen is eljárhatunk, mint a fent felvázolt transzformációsorozat, az itt csupán a kiindulópontként használt módszer megértését elősegítendő szerepel.

A vertex shader futása után az adatok néhány kevésbé programozható állomáson haladnak keresztül, ahol:

- A beállított paraméterezéstől függően kiválasztják az adott sokszög vertexeinek a sorrendje alapján (projektált képének órajárás szerinti, vagy azzal ellentétes sorrendjében történt-e a megadás), a háromszög hátsó, illetve elülső oldalát és a paraméterezéstől függően levágják a hátsónak megfelelő sorrendben álló háromszögeket a megjelenítési futószalagról. Ez az ún. **backface culling**, ami sokat gyorsít a futáson ha az objektumaink külső felén egy irányban definiáljuk a vertexeket az egyes háromszögekhez.
- Megtörténik a képernyőről a projekció után kilógó háromszögek, illetve a kamerához képest túl nagy távolságban lévő háromszögek kivágása a futószalagról. Ez az ún. **Frustum Culling**
- Megtörténik a négydimenziós vektorok w komponenseivel önmagával történő leosztása a homogén koordináták használata miatt.
- **A vertex shader kimeneti** csatornáiban szereplő, a háromszög csúcsaihoz rendelt **értékek interpolálásra kerülnek** a pixel shader bemenetébe való kerülés előtt, az adott síkbeli fragmentekhez.

Az adatok ezek után a **fragment-feldolgozó**, szintén programozható állomáshoz kerülnek, ahol az általában **pixel shadernek**, vagy **fragment shadernek** nevezett program fut le ahol manipulálhatjuk a pixelek előállításához szükséges adatot.

A **pixel shader** a bemeneti adatainak és a számára is felhasználható shader konstansoknak a segítségével meghatározhatja a kimeneti csatornájára helyezendő színértéket, illetve néhány másik értéket(a később emlegetett Z érték is módosítható szokott lenni), melynek kiszámításához a fenti kétféle típusú bemeneten kívül felhasználhat ún. **textúra-mintavételező**, angol terminológiával „**texture sampler**”

objektumok által a megjelenítő eszköz számára eltárolt síkbeli textúrákból (ezekre az egyszerűség kedvéért tekinthetünk úgy, mintha hagyományos képek lennének) mintavételezett színértékeket is.

Miután a pixel shader programunk lefutott, után újból néhány nemprogramozható, de általában paraméterezhető állomás következik:

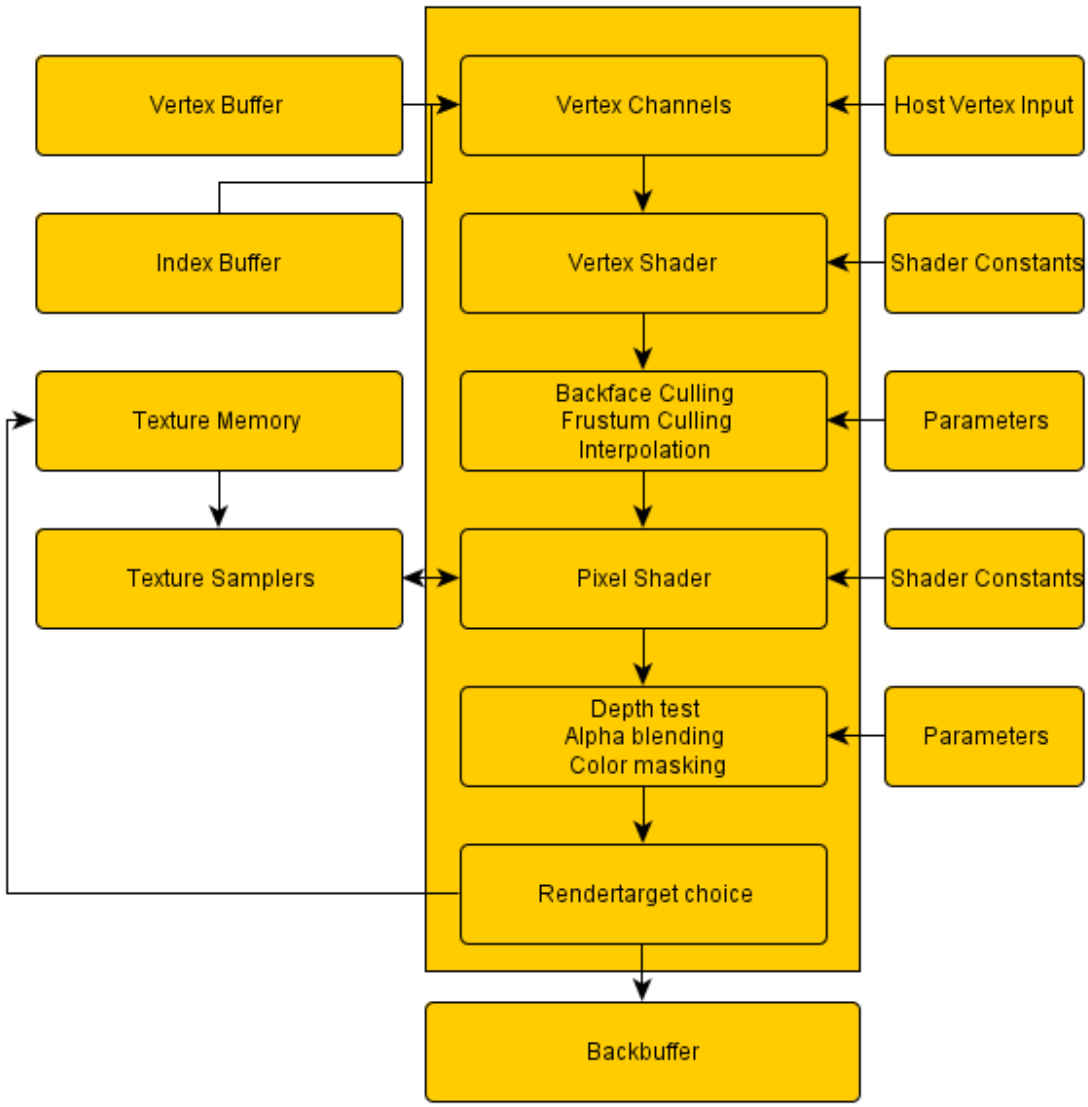
- Levágásra kerülnek a háromszögeknek az eddigi transzformációk ellenére mégis képernyőn kívül eső részei a **scissor test** során.
- Ún. „**alpha testing**” hajtódik végre, melynek segítségével szelektálhatunk kirajzolandó és eldobandó (ezzel teljesen átlátszó) pixelek között.
- Ún. **Depth/Stencil mélységi teszt** alapján meghatározzuk, hogy a most kirajzolásra kerülő pixel, melyhez kiszámolásra kerül egy **Z**, a kamera síkjától vett távolságérték közelebb van-e a kamerához, mint a korábban már a célpontban lévő pixel(ez az alapértelmezett tesztelés, de ez a lépés is állítható, amint majd azt látni fogjuk). A **Z** értékek a renderelés célpontjához külön hozzárendelt pufferében tárolódnak, az ún. **Z-Buffer**-ben
- Ún. **alpha blending** hajtódik végre, a félig átlátszó objektumok kirjzolásához, mely során paraméterezhető a korábban ott talált pixel (destination) és az új pixel (source) között elvégzendő művelet, illetve a két tényező súlyozható.
- Az alpha blending kiszámolása után **maszkolásra kerülnek** az előzőleg beállított írható **színcsatornák** úgy, hogy a nem írhatóak értéke az előző számítások eredményétől függetlenül megőrződik (ezt a ritkán kihasznált dolgot mi is felhasználjuk majd a robbanási effektusunknál)
- Az így kiszámolt színérték beíródik a renderelés célpontjául szolgáló textúrába (**rendertarget**), ami általában a **backbuffer**.
- Amennyiben a backbuffer-be történt a renderelés, az a már teljesen kész tartalmával kicserélődik az éppen megjelenített **frontbufferrel**, miközben az kerül besorolásra backbuffernek. Így a kész kép megjelenik a képernyőn.

  Megemlítendő, hogy ez a csere analóg képernyő esetén általában akkor történik meg, amikor az elektronsugár nyalábja inaktív és a függőleges visszafutást végzi el, így nem keletkezik „félbevágott kép” a csere miatt.
- Amennyiben pedig nem a backbufferbe rendereltünk, esetleg további kirajzolási fázisokat is kezdeményezhetünk a jelen kirajzolás eredményéül szolgáló kimeneti textúra inputként való felhasználásával.

Tehát lényegében egy kirajzolási fázis során, a memóriából definiált térbeli pontokat (vertexek) transzformálhatjuk a vertex shaderrel, majd a kimenő adatok **interpolációja** után, pixelenként elvégezhetjük a konkrét színérték kiszámításához szükséges értékeket a textúra adatainak a felhasználásával a pixel shader program segítségével.

A shadereinket az XNA által is támogatott HLSL (**High Level Shading Language**) nyelven írjuk majd meg, méghozzá a pixel és vertex shader programok egyetlen *.fx kiterjesztésű fájlban történő elhelyezésével. A HLSL nyelv C-szerű szintaktikáját és a nyelvet bővebben, a dolgozat során nem ismertetjük, mert a legtöbb forrás könnyen érthető és átlátható, ezen felül pedig a programok működési leírása is adott.

1.3.2.3 A grafikus szerelőszalag egyszerűsített felépítését, számunkra jelentős állomásait, a bemeneti és kimeneti pontjai, illetve az adatok áramlását bemutató áttekintő-ábra



Bal oldalt láthatóak a megjelenítő eszközön található, de a futószalag nem szerves részét képező memóriaterületek, illetve a mintavételező egységek, jobb oldalon a befogadó architektúrával történő kommunikáció főbb csatornái, középen pedig kiemelve, az összeszerelő futószalag számunkra fontos állomásainak az egyszerűsített sorozata, melyeken az adatfolyam áthalad. Az adatok haladását a nyilak iránya jelzi.

Az ábra nem teljes, de a fenti ismertető mellett bőven elegendő a dolgozatban felhasznált módszerek

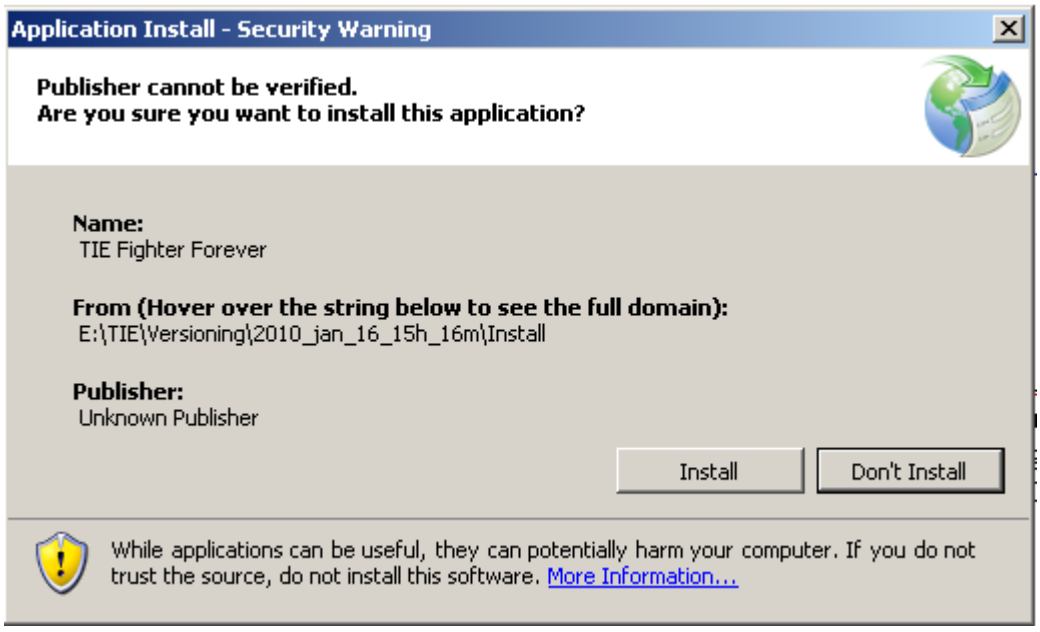
2. Felhasználói dokumentáció

A program, egy űrben játszódó háromdimenziós akciójáték, melyben a cél a minél magasabb pontszám elérése.

2.1 A program telepítése

A program öntelepítő setup.exe formájában van rajta a lemezen, az install könyvtár alatt, mely internetkapcsolat megléte esetén letölti a szükséges szoftver előfeltételeket is. Amennyiben hiba lépne fel, és a telepítő nem képes letölteni a megfelelő fájlokat, akkor azokat a rendszerkövetelmény, szoftveres vonzatánál felsorolt elemek kézi telepítésével is installálhatjuk.

Amennyiben az automatikus telepítés mellett döntöttünk, akkor internetkapcsolat megléte esetén először, egy valószínűleg nem túl bizalomgerjesztő képernyő fogad minket, mely azért ugrik elő. Mert a Microsoft szervere, ahonnan az előfeltételek letöltése zajlik majd, nem ismeri a fejlesztőt (vagyis engem), így rákérdez, hogy biztosan telepíteni kívánjuk-e a programot. Nyugodtan nyomjuk meg tehát az Install feliratú gombot és kövessük a megjelenő utasításokat.



Amennyiben az előfeltételek telepítve vannak, a program felinstallálódik a gépre, majd azonnal el is indul és a főmenü képernyőben találjuk magunkat. Ezzel szemben viszont, ha hiányzik valamilyen szoftveres követelmény, akkor várnunk kell, amíg a telepítő letölti azt az internet segítségével. Ilyenkor a telepítési folyamat hosszabb és újraindításokat is követelhet.

Megjegyzés: Amennyiben a telepítési folyamat során áramszünet, vagy egyéb ok miatt leállna a gép, indítsuk újra a telepítőt, az nagy valószínűség szerint be tudja majd fejezni a műveletet.

A telepítési folyamat befejeztével, kétféle kimenetellel találkozhat a felhasználó, vagy

valamilyen hibaüzenettel, amikor is javasolt utánanézni a szoftver és hardver követelményeknek, vagy a menü-képenyő elindulásával. Mi mostantól az utóbbi esettel foglalkozunk. Vagyis a továbbiakban feltételezzük, hogy a program sikeresen települt.

2.2 Navigáció a főmenüben és egyéb menükben

A játék menüiben a kurzormozgató billentyűkkel mozoghatunk és az enter lenyomásával aktiválhatjuk a kiválasztott menüpontot. A főmenüben két menüpnt található, melyek közül a „Start Battle” feliratú a tényleges csatát indítja el, az exit game pedig kilép a játékból.

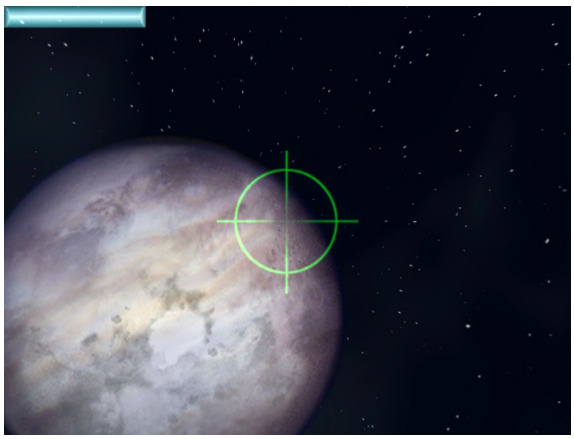
A játék során többször is találkozhatunk ehhez hasonló menü-képernyőkkel, ilyenkor a kezelés általában ugyanígy, a kurzormozgató billentyűkkel és az enterrel történik.

A játék főmenü-képernyője a következő ábrán látható:

A menüben az aktuálisan kijelölt objektumok mindig azok mozgásával vannak jelezve.

2.3 A Csata-képernyő felépítése

A továbbiakban feltételezzük, hogy a felhasználó elindította a csatát, ekkor a következő kép fogadja:



Ez pedig már a játékcsataképernyője, melynek a felépítése a következő:

- A bal felső sarokban a hajónk pajzsának az ereje látható egy kék csíkkal ábrázolva. Amennyiben ez elfogy, a játékos egyetlen kapott ellenséges lövést követően elveszti a játékot.
- Középen látható egy szálkereszt, ami segít az ellenséges hajókra történő célzásban.
- Látszik az űrbéli környezet és a benne található hajók, és a lövedékek
- A kamera valószínűleg vadul pörög, amin az irányítási leírás ismeretében majd változtathatunk

2.3 Irányítás a Csata-képernyőn

A játék során egy olyan űrhajót irányítunk, mely:

- Folyamatos, konstans sebességgel halad előre
- Képes forogni az előre mutató tengelye körül, az egér jobbra-balra kitérésének a függvényében.
- Képes emelkedni, illetve süllyedni az egész fel, illetve lefele kitérésének a függvényében
- A bal egérgomb lenyomása (vagy nyomva tartása) mellett képes négy lövedéket, másodpercenként legfeljebb egyszer előrefelé kilőni, az ellenség irányába.

A játékos ezen felül az **Escape** gomb lenyomásával előhozhatja a játék közbeni menüt, melyben választhat a csata-képernyő elhagyása(„Exit Battle”) és a játékban maradás („Return to Game”) között, miközben a játék futása szünetel.

A játékban gépi ellenfelek jelen vannak, melyek a játékos elpusztítására törekednek. A játékos ezek és a nagy űrállomások megsemmisítésével szerezhet pontokat, melyeknek összegét a játék végén, a játék vége képernyőn láthatjuk, ahova kilépéskor, vagy az életerő elfogyásakor juthatunk.

2.4 A játék vége képernyő

A játékos a csata elhagyásával, vagy az életpontjainak az elfogyásával, a játék vége képernyőre kerül, ahol megtekintheti a pontszámát és amit enter lenyomására hagyhat el, visszakerülve a főmenübe.

2.5 Hol találom a játékot?

A játékot annak telepítése után a start menü, megatron könyvtára alatt találjuk meg, TIE Fighter Forever nevű indítóikonnal ellátva.

2.6 Minimális Rendszerkövetelmény

2.6.1 Minimális szoftver követelmények

- Windows XP (vagy újabb) operációs rendszer, telepített SP2 javítócsomaggal
- Microsoft DirectX rendszer, legalább 9.0c verziója
- A felhasználó videokártyájához adott megfelelő képernyőillesztő programok
- .Net Framework 4 Client Runtime (a telepítő automatikusan installálja)
- .Net Framework 3.5 (a telepítő automatikusan installálja)
- XNA Game Studio 4.0 redistributable (a telepítő automatikusan installálja)

2.6.2 Hardver követelmények

- **Szükséges** DirectX9 kompatibilis videokártya, shader modell 2.0 támogatással
- **Szükséges** legalább 800x600-as felbontással rendelkező képernyő
- Ajánlott legalább 512 Mb rendszermemória
- Ajánlott legalább 1.5Ghz órajelű processzor

3. Fejlesztői dokumentáció

3.1 A program fő struktúrájának összefoglalása

A program solution 4 fő projektet tartalmaz, ezeket vesszük most röviden sorra, mert az áttekinthetőség érdekében már az elején jó tudni, miért van 4 projekt és hol mit találunk meg. A solutionben található projektek, továbbfejleszthetőségi értelemben véve fontossági sorrendben a következők:

i. **TIE Fighter Forever:**

   Ez a fő projekt, a játék szinte minden lényeges kódja ebben található.

ii. **TIE Fighter ForeverContent:**

   Ez a projekt foglalja magába a játékprogram által felhasznált külső tartalmakat, például háromdimenziós modelleket, textúrákat és shader effekt forráskódokat.

iii. **StarFighterEffect1Pipeline:**

   Ez a projekt tartalmazza az XNA content pipelinejának azon kiterjesztéseit, melyeket a vadászgépek beolvasásához használunk. Amennyiben továbbfejlesztési okokból szükség van egyéb vadászgépekhez tartozó modellbetöltésre, hasonló projektfájl írandó.

iv. **MothershipEffect1Pipeline:**

   Content pipeline kiegészítéseket tartalmaz nagyobb űrhajómodellek beolvasásához. Továbbfejlesztés az előző ponthoz analóg módon hozandó létre.

v. **CollisionPipeline:**
   Content Pipeline kiegészítések az ütközési modellek betöltéséhez.

vi. **CollisionPipelineRuntimeHelper**
   Az előző ponthoz köthető. Bővebb magyarázat az ütközéssel foglalkozó részben.

A fejlesztői dokumentáció a programot nem a 4 projekt külön ismertetésével mutatja be, hanem a fő modul komponenseinek és megoldásainak fejlesztői bemutatásával kapcsolatosan mutatjuk meg a többi modul tartalmát. A fő projektfájl (és a content-et tartalmazó is) könyvtárstruktúrába van szervezve a névterek könnyű átláthatósága, illetve a logikailag egy helyre tartozó osztályok szervezése érdekében. Ezen szervezésen túl a legtöbb tényleges forrásfájl csak néhány (vagy egy), szorosan összetartozó osztályt tartalmaz.

3.2 A belépési pont és a programfutás áttekintése

A program tényleges **belépési pontja** a TIE Fighter Forever projekt gyökérkönyvtárában található **Program.cs** elnevezésű fájlban található. Itt hozzuk létre az XNA osztálykönyvtárából származó **Game** osztályból öröklődő **TIEGame** osztályból származó objektumunkat és hívjuk meg annak **run()** metódusát, amivel megkezdődik a program lényegi részének a futása.

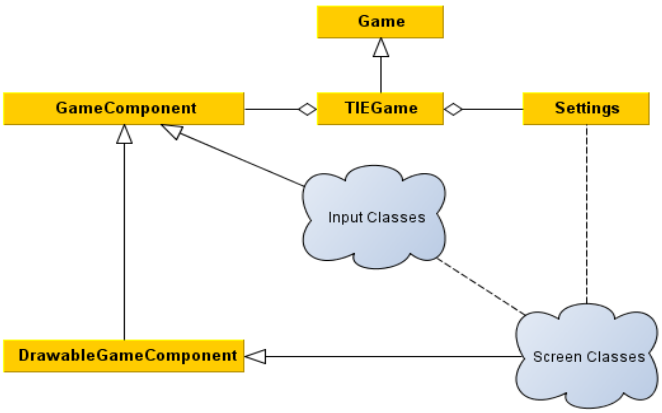
A **TIEGame.cs** fájl tartalmazza a fent említett osztály definícióját, melynek az XNA keretrendszer hívja meg az **Initialize()** metódusát és a **LoadContent()** metódusát,

illetve ezután meghívásra kerül bizonyos frissítési időközönként a **Draw(...)** és az **Update(...)** metódusok hívása is. A rendszerünk úgy épül fel alapvetően, hogy játék-komponenseket regisztrálunk, melyek meghívásra kerülnek a fő osztályunk megfelelő függvényeinek **base.Draw(...)** és **base.Update(...)** hívásakor. A játéktér/színtér-menedzsment úgy van megoldva, hogy az egyes játék képernyők, melyek a **Screen** névtérben vannak elhelyezve, bejelentik magukat a fő játék osztályba mint komponens és onnantól kezdve a kirajzolás és az update rajtuk keresztül folyik. Nincs külön rendszer, ami kezelné ezeket a screen-osztályokat, az egyikből a másikba való átlépés úgy manuálisan, az éppen aktuális screen osztályában zajlik le. Esetleges fejlesztés lehetne, ezeknek a screen-eknek egy közös ősosztályt és egy kezelőrendszert adni, ám a legtöbb esetben csupán nagyon kevés screen van egy ilyen jellegű játék során, így erre most energiát nem fordítottunk(bár könnyen kiegészíthető a program egy ilyen rendszerrel, ha szükség van rá), mert a screen-ek kezelése, már így is elég könnyen megoldható.

Megjegyzendő dolog, hogy csak azon a komponensek **Draw** és **Update** függvényei kerülnek automatikusan meghívásra, amik a fő játék osztályhoz mint komponens, hozzá vannak adva a **Components.Add(...)** tagfüggvénnyel. Ez azért is megemlítendő, mert sok dolog van megvalósítva komponensként, melyeket ennek ellenére közvetlenül, a tagfüggvényein keresztül kezelünk és nem a keretrendszer automatikáján keresztül. Általánosságban az mondható el, hogy logikus módon az irányítással összefüggő komponensosztályok és a játék aktuális képernyője van aktív komponensként hozzáadva, a többi komponensosztály pedig inkább csak azért rendelkezik a **GameComponent** vagy **DrawableGameComponent** ősszel, mert kód-hordozhatósági szempontból így értelmes megcsinálni az adott osztályt mert például egy laserglow komponens felhasználható egy másik játékban, amiben lehet, hogy máshogy van megoldva a screen-management. Természetesen az olyan osztályok, amik nem kirajzolhatóak a sima **GameComponent**-ből származtatottak, a többi komponens pedig a **DrawableGameComponent**-ből.

Ezen felüla **settings.cs** fájl gyűjt össze mindent, ami változtatható beállítás lehet a játékban.

A játékprogram magas szintű architekturális felépítése és a főbb kapcsolatok az alábbi ábrán könnyen áttekinthetőek:

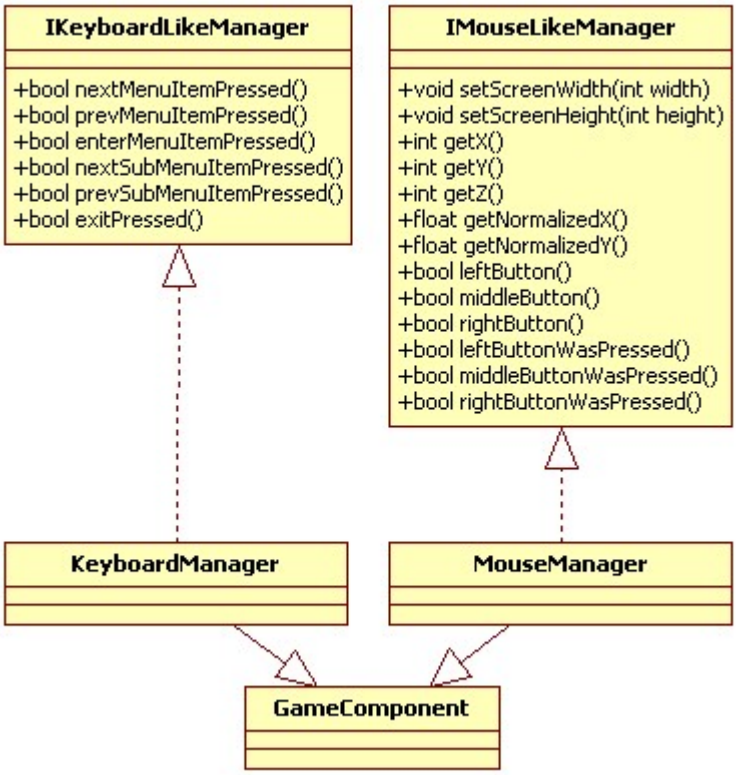


Megj.: Az ábrán csak a legfelső szinten is lényeges osztályok jelennek meg. A többi kimaradt rész az architektúra megértésének szempontjából itt még nem fontos.

3.3 Az irányítási rendszer és komponensei

A játék fontos alkotóeleme az XNA framework irányítással kapcsolatos megoldásai köré épített kis osztály és interfészrendszer, mely úgy van megalkotva, hogy könnyen lehessen hozzáadni más architektúrák alapkiszerelésű irányítóhardvereit lekezelő osztályokat. Ezt úgy érjük el, hogy definiálunk interfészeket, melyekből a tényleges irányítási osztályainkat származtatjuk a program összes többi részén pedig csak interfészeknek megfelelő változókat hozunk létre, így rejtve el az irányítás megvalósítását a rendszer elől.

Az irányítási rendszer áttekinthető az alábbi ábrán, mely tartalmazza az interfészek megvalósításához szükséges függvényeket is:



A **KeyboardManager** és **MouseManager** osztályok a **GameComponent**-ből származnak és megvalósítják a fenti két, általunk definiált interfészt.

A **KeyboardManager** osztály megvalósításának lényege, hogy nyílván tartunk kettő az XNA által számunkra adott **KeyboardState** objektumot, melyet a **Keyboard** osztály **GetState()** metódusával kérdezünk le a jelenlegi frame állapotát tükröző **newState** változóba, ezt felhasználva és az **oldState** változóval összehasonlítva beállítjuk a belső változóinkat, melyeket az interfész függvényeit megvalósító metódusokkal kérdezhetnek le az osztálytól. Ha megvagyunk a belső változók beállításával, akkor a **newState**-et az **oldState**-re mentjük. Az **oldState**-et a konstruktorban inicializáljuk előreolvasás jelleggel, hogy a fenti módszer működjön, a konstruktorban ezen felül **hozzáadjuk az objektumunkat a játék osztály szolgáltatásaihoz**, melyek közül lekérdezhetik típus alapján a komponensek(vagy más a játék objektumhoz hozzáférő

osztályok) az objektumunkat.

A **MouseManager** osztály megvalósítása teljesen hasonló módon történik, de van néhány említésre méltó dolog. Az egérgombok állapotát visszaadó függvényeket a billentyűzetet kezelő osztállyal analóg módon valósítjuk meg, de lekérdezhetővé tesszük azt is, hogy az adott jelenlegi függvényhívás pillanatában le van-e nyomva az adott gomb. Ez utóbbi hasznos lehet például akkor, ha egy gomb folytonos nyomva tartása esetén folyamatosan lőni szeretnénk megszakítás nélkül. **A felengedést megkövetelő függvények Pressed végződéssel vannak ellátva**, a jelenlegi állapotot lekérdező függvényektől való könnyű megkülönböztetés végett. A **MouseManager** osztály legfontosabb feladata a gombok állapotának lekérdezhetőségének biztosításán felül az egér jelenlegi koordinátáinak a visszaadása, melyet le lehet kérdezni abszolút koordinátaként és a képernyő méretéhez képest értett **relatív koordinátaként** is (ilyenkor [-1..1] – béli értéket kapunk visszatérési értékként). A relatív koordináták előállításához viszont szükség van a képernyő szélességére és magasságára, ami elméletileg akár dinamikusan is megváltozhat, így külön függvényeket biztosítottunk az interfészben ezek frissítésére. Az osztály **getZ()** metódusa az egér görgőjének a pozícióját adja vissza, ennek természetesen nincs relatív változata, hiszen nincs mihez képest osztani a Z koordinátát.

Felmerül még kérdésként az, honnan és ki hívja meg, az **Update** függvényünket. A válasz egyszerű: Az XNA keretrendszer hívja meg őket akkor, amikor a fő játékosztály Update függvényének a végén meghívásra kerül a **base.Update** ősosztálymetódus. Az irányító osztályok még a **TIEGame** osztály **Initialize** metódusában kerülnek hozzáadásra, mégpedig ezek az osztályok a leghamarabb az összes komponens közül.

Továbbfejleszthetőségi szempontból a rendszer úgy fejleszthető, vagy alakítható át a legeffektívebben irányítási szempontból, hogy:

(1) Az interfészeket bővítjük új gombok szükségessége esetén

(2) Új osztályokat hozunk létre, melyek a **GameComponent** osztályból származnak és megvalósítjuk bennük a megfelelő interfészt. Ilyen módon ha létrehozunk egy osztályt, ami az XBOX kontrollerével emulálja az egér működését és megfelel az interfésznek, minimális módosítással lecserélhető az irányítási rendszer.

(3) Tipp: Mivel a C# nyelv támogat többszörös öröklődést ha interfészekről van szó, kihasználhatjuk annak az előnyét, hogy egy osztállyal mindkét interfészt megvalósítsuk. Ez azért hasznos, mert például az XBOX esetén javallott egyetlen **XBOXContoller** osztály létrehozása, mely megvalósítja mind a billentyűzet, mind az egér működését, hiszen logikailag is így lesz átláthatóbb a kapott rendszer, hiszen a valóságban is egyetlen tényleges hardvertől – az XBox kontrollerétől, kapunk inputot.

3.4 A játék főmenüje és annak elemei

3.4.1 A TextMenus menümegjelenítő rendszer

Mivel a játékban a főmenün kívül máshol is előfordulhatnak menük, hasznos legalább egy darab egyszerű, de újrahasznosítható menümegjelenítő rendszert létrehozni. A **TextMenus** névtér osztályai és interfészei pont ezt hivatottak megvalósítani egy könnyed, de szerethető és használható, paraméterezhető felugró menü megvalósításával.

Ezt a menümegjelenítő rendszert fogjuk felhasználni mind a játék indulása után megjelenő főmenü, mind pedig a játék közben felugró menü megvalósítására, illetve akár arra is, hogy adott esetben sima szöveget írjunk a képernyőre. A rendszer ezen kívül tartalmaz egy továbbfejlesztett, de nem felhasznált megoldással is.

A menümegjelenítő rendszer három fontos komponensből áll:

(1) **MenuItem**-ből származó osztályok, melyek a menükbe pakolhatóak

(2) Az **ITextMenu** interfész és neki megfelelő menü-osztályok

(3) **ITextMenuNotifiable** interfész, mely callback függvényt biztosít nekünk a menüt befogadó osztályok részére.

3.4.1.1 A MenuItem osztályok

A **MenuItem** osztály egy egyszerű, de rugalmas menüpontot valósít meg egyetlen szöveges érték tárolásával és virtuális metódusokkal az öröklődő osztályok számára. Lehetőséget nyújtunk példának okáért a leszármazott osztályokban címszöveg elhelyezésére, illetve több szöveg elhelyezésére, és meg is valósítunk egy **MultipleChoiceMenuItem** nevű leszármazott osztályt, mely rendelkezik címszöveggel és több szöveges elemet tartalmaz melyek közül válogathatunk a **prevSelection()** és **nextSelection()** virtuális metódusok felüldefiniálásának eredményeként.

3.4.1.2 Az ITextMenu interfész

Egy új menürendszer implementálásakor az **ITextMenu** interfészből származó menümegvalósításoknak tudnia kell a menüt menüpontokkal bővíteni, a menüpontokat megjeleníteni és a rendszert frissíteni. Ezeket a funkciókat a menü-osztálynak az interfészben adott 3 függvény megvalósításával kell elérhetővé tennie:

- `void addMenuItem(MenuItem menuItem);`
  `-Adott menüelem hozzáadása`
- `void drawMenuItems(int fadeIn, SpriteBatch spriteBatch);`
  `-A menüpontok kirajzolása 0..255-ig terjedő halványítási „fade” érték mellett, ahol a 255 jelenti a teljesen megjelenő menüt, 0 felé pedig az egyre halványabbat.`
- `void updateMenuItems();`
  `-A menü állapotváltozásának megvalósítása`

3.4.1.3 Egyszerű megvalósítás: a SimpleMenu osztály

A játék által használt menü-osztály a **SimpleMenu** nevű osztályban van megvalósítva a **TextMenus** névtér alatt. Ez a menü-osztály képes egy, a konstruktorában megadott **SpriteFont** felhasználásával (melyet beépített módon betölthetünk) adott pozícióban

egy középre, vagy balra zárt, adott színnel rendelkező menüt megjeleníteni a képernyőn.

```
public SimpleMenu(ITextMenuNotifiable parent,
                  IKeyboardLikeManager keyLikeMan,
                  Vector2 position,
                  SpriteFont menuFont,
                  Color color,
                  SimpleMenuAlign align,
                  float lineSize)
```

A **color** és a **menuFont** paraméterek határozzák meg a menüpontok szövegének a kinézetét a szín és a felhasznált font meghatározásával. A **position**, **lineSize** és **align** paraméterek az egész menü kinézetének és elhelyezkedésének a paraméterezését tartalmazzák a menü adott képernyő koordinátákon történő elhelyezésével, a szövegigazítás meghatározásával, és az egyes sorok után történő lefele elmozdulás mértékével.

Miután létrehoztuk a menünket, az interfésznek megfelelő **addMenuItems(..)** függvény meghívásával adhatunk hozzá **MenuItem**-eket a menühöz, melyeket a **SimpleMenu** egy Listával tart számon.

Amennyiben feltöltöttük a menüt, annak használatához a **parent** kódban csupán el kell helyeznünk egy **drawMenuItems** függvényhívást azon a ponton, ahol ki szeretnénk rajzolni a menüt, illetve el kell helyeznünk egy **updateMenuItems** hívást is egy megfelelő helyen (pl. egy if után, ami eldönti, hogy a menü aktív-e).

Az elemek kirajzolása úgy történik, hogy lekérjük a keretrendszertől, a szöveg grafikus megjelenésének várható értékét, majd a kirajzolás adott koordinátájához képest a szöveg szélességének a felével balra elcsúsztatva kezdjük meg a menüpont kihelyezését. Amennyiben a balra igazítás be van kapcsolva, ezt az eltolást természetesen nullázzuk.

Az aktív elem megkülönböztetése úgy zajlik, hogy a kiválasztott indexet számon tartjuk és az **IKeyBoardLikeManager** interfészű objektumunktól való lekérdések alapján az **update** során ha kell frissítjük. Amennyiben kirajzoláskor az adott elem indexe éppen az aktív elem indexének megfelelő, akkor az előzőleg említett balra elcsúsztatás mértékét és a szöveg megjelenítésének alapméretét megszorozzuk a **scale** léptékezési faktorral. Így összességében hullámzó hatást érhetünk el a kiválasztott elemen.

A **parent** osztály megfelelő callback függvényének a meghívása akkor következik be, amennyiben a menü az update során az **enterMenuItemPressed()** hívásra igazat kap az **IKeyboardLikeManager** objektumtól. Ekkor hívódik meg a szülőként megadott **ITextMenuNotifiable** interfészt megvalósító objektumban definiált függvény, melynek segítségével megtudhatjuk melyik menü, melyik menüpontját választotta a felhasználó. Az említett függvény szignatúrája a következő:

```
void textMenuCallback(ITextMenu menu, MenuItem selectedMenuItem);
```

3.4.2 A főmenü képernyője

A főmenüképernyő egy **DrawableGameComponent**, mely implementálja az **ITextMenuNotifiable** interfészt és egy háttér felett megjeleníti a főmenüt, kilép a játékból, ha az Exit feliratú menüpontot választjuk, illetve hozzáadja a játék komponenseihez a csataképernyő osztályát, amennyiben a játék indítását választjuk.

A menü létrehozásához szükségünk van a fő játékosztály komponenseihez hozzáadott **IKeyLikeManagerre**, amit a következő technikával kérhetünk le a szolgáltatások

közül:

```
keyLikeMan = (IkeyboardLikeManager)game.Services.GetService
                                        (typeof(IKeyboardLikeManager));
```

A főmenüképernyő ezt követően a **konstruktor**ban, az **Initialize()** és a **LoadContent()** metódusban inicializálja a képernyőosztályt, majd megkezdődik a futás melynek során a fade nevű változó segítségével teljesen feketéből, lépésenként egyre erősebb színeket megjelenítve keltünk betöltődés-jellegű hatást.

A menüpontokat a **textMenuCallBack** függvényben kezeljük le, amiben ellenőrízzük, hogy a felhasználó a kilépés vagy az indítás pontot választja-e, majd ennek megfelelően meghívjuk, vagy a játék azonnali végét jelentő **Exit()** metódusát a fő **TIEGame** osztálynak, vagy beállítjuk a **goBattle** nevű bool változót és elkezdünk, most már az egyre feketébb megjelenés irányába elsötétíteni a képernyőt a **fadeIn** változóval.

Ha az update során azt tapasztaljuk, hogy újra elértük a sötét képernyő állapotot és a **goBattle** igazzá vált, akkor inaktívvá tesszük ezt a komponenst, létrehozunk egy új csatajelenetet (melynek megadjuk ezt a jelenetet szülőként) és annak Inicializálása után hozzáadjuk azt a komponensekhez. Így lényegében képernyőt váltunk.

3.5 A háromdimenziós csata és a felhasznált komponensek

A játék a csatajelenet során vált át 3D módba és a program legtöbb osztálya ehhez a részhez köthető, ez a játék magja.

Ez az alfejezet úgy épül fel, hogy felvetünk egy megoldandó grafikai, vagy egyéb jellegű problémát, aztán bemutatjuk a megoldásához kapcsolódó konkrét megvalósításokat. Nem forráskódról forráskódra haladunk tehát, hanem a megoldandó problémák övezte logikai útvonalon, melyen tulajdonképpen a fejlesztés is zajlott.

3.5.1 Environment Mapping környezetleképezési technika megvalósítása a vadászgépek kirajzolásához

3.5.1.1 A megoldandó feladat

A játék során lényegében kétféle űrhajó objektumot szeretnénk megjeleníteni: kicsi vadászokat és nagy állomásokat vagy anyahajókat. A nagy objektumok megjelenítésének a feljavításához, kézenfekvő ötlet a buckaleképezési technikák alkalmazása, ám ez kis hajóknál általában egyszerűen nem látszik meg, így más megjelenítési módszerhez folyamodunk.

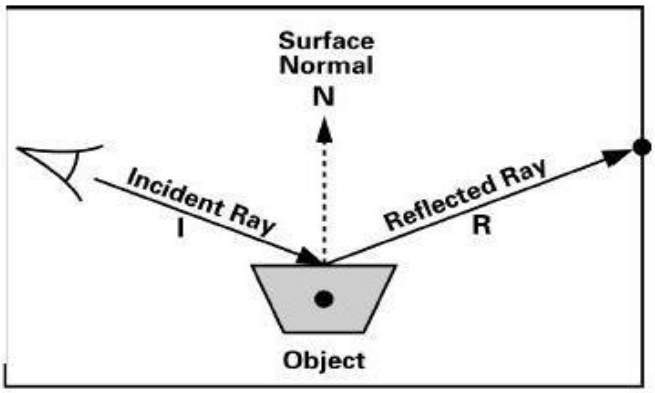
Általában kellemes hatás érhető el akkor, ha a kis hajók tükröződő hatást keltenek és fémes felületet utánoznak, mert ez gyors mozgás esetén, messziről is többé-kevésbé látszik a kis méret ellenére is. Azért, hogy elérjük a tükröződő hatást, a cube environment mapping elnevezésű grafikai megoldást implementáljuk, melyhez szükségünk lesz a megfelelő modellbeolvasást kiegészítő osztályokra is.

3.5.1.2 Az environment mapping technika elméleti alapjai és a megjelenítő rendszer alapvető architektúrája

Az environment mapping lényege, hogy a valós fénytükröződési számítások elkerülése mellett, hihetően utánozzuk a fényvisszaverődés érzetét. A módszer ehhez egy cubemap

textúrát használ fel, mely egy adott környezet egy kocka felületére vetített képe. A tükröződési effektust úgy érhetjük el, hogy a kamera szemvektorát felhasználva, majd azt a felület normál vektorával tükrözve indexelünk egy kocka oldalán a kapott vektorral.

A módszer egyszerűsített, síkbeli működését a következő ábra mutatja be:



Megj.: Amennyiben ezt a textúrát valós időben számoljuk ki, akkor már a valós tükröződés közelítő szimulációjáról beszélhetünk nem pedig annak utánzásáról.

Mivel az űrhajómodellek textúrázva is vannak és nem akarunk tökéletesen tükröződő objektumokat, a környezetleképezés által kinyert színértéket a modell eredeti textúrájából kinyert értékével együttesen fogjuk felhasználni a végleges szín előállításához így kombinálni fogjuk a fenti módszert a textúrázás és megvilágítás egyéb egyszerűbb modelljeivel is.

Az effektet hardveresen támogatott formában valósítjuk meg, a videó kártyán futó **pixel** és **vertex shader** programok írásával.

Az environment mapping megvalósításunk alapja, tehát egy vertex és pixel shader programpáros lesz, mely az űrhajómodellre alkalmazva annak textúrájának és a **cubemap** textúrának a felhasználásával előállítja a **framebufferbe** kerülő pixel színét.

3.5.1.3 Az environment mapping technika konkrét megvalósításának az áttekintése.

A korábban vázolt elméletet a programban úgy valósítjuk meg, hogy:

- Létrehozzuk a cubemap textúra felhasználásával, a környezetleképezést megvalósító shadereket tartamazó effect fájlt.
- Átalakítjuk az XNA keretrendszer modellbeolvasási rendszerének megfelelő részét, hogy annak kiterjesztésével egy betöltött háromdimenziós modellhez hozzárendelhessünk egy cubemap környezeti textúrát és a saját shader effect fileunkat.
- Egy trükkös megoldás alkalmazásával elérjük, hogy a cubemap környezeti térképet nem a kocka felületére eső 6 darab kép megadásával kelljen létrehoznunk, hanem csupán egyetlen, hagyományos síkbeli textúrából, automatikusan generáljuk ki egy többé-kevésbé megfelelő cubemapot.

3.5.1.4 A videó kártyára kiküldésre kerülő starfightereffect1.fx shader effect

A shader programjainkat ellátjuk ún. konstans paraméterekkel, melyek két renderelési fázis között között változhatnak csak meg(azaz egy render során végig konstansok).

Ezek az adatok a videokártya konstans regisztereibe kerülnek és a shader programokat tartalmazó effect fájlban változóként deklarálandók. Az így megadott konstansokat a processzoron futó főprogram minden, az adott effektust használó kirajzolás előtt be kell állítsa.

A használt konstans shader paraméterek:

```
float4x4 World;
float4x4 View;
float4x4 Projection;

float4 LightColor;
float3 LightDirection;
float3 AmbientColor;
float3 eyePosition;
float Glow;
bool TextureEnabled;

texture Texture;
texture EnvironmentMap;
```

Három darab 4x4-es mátrixra van tehát szükségünk, a világtérbe transzformáló **World**, a nézeti(vagy kamera)-térbe transzformáló **View** és a térbeli pontok projekcióját elvégző **Projection** mátrixra. Ezen kívül szükségünk lesz, a játékban használt egyetlen fényforrásunk színére és a vektorra ahonnan a fény érkezik, illetve a kamera pozíciójára is. A térbeli adatokon kívül deklarálunk egy logikai változót, amivel a külső program jelezheti, ha egy adott anyag nem textúrázott, illetve egy lebegőpontos változót, amit a **Glow** effektusunkhoz használhatunk fel és ami végül a pixel alfa csatornájába kerül. Ezen felül a külső programnak be kell állítania a színeket tartalmazó felületi textúrát és az environmentmap-nak megvelelő cubamap textúrát.

Az effect fájlban mind a beépített pixel, mind a vertex shader program felüldefiniálásra kerül. A vertexekre lefutó shader függvényünk számára a vertex streamből kiolvasott bemeneti paramétereinek listáját és deklarációját a **VertexShaderInput** struktúra, míg ezen függvény outputját a **VertexShaderOutput** struktúra írja le. Ez utóbbi struktúra viszont bemeneti paramétere a pixel shadernek, mely kimenetként egy 4 komponensből (RGBA) álló színértéket állít elő. Az említett struktúrák, persze magas szintű nyelvi eszközök, de a valóságban a videokártya egyes komponensein futó programok közötti kommunikáció úgy valósul meg, hogy a bemeneti és kimeneti struktúrák változódeklarációjában ún. Szemantikák segítségével megadjuk, hogy az adott objektum, melyik regiszterben kerül tárolásra. Egy példával könnyen szemléltethető a deklarációk felépítése:

```
struct VertexShaderInput
{
      float4 Position    : POSITION0;
      float3 Normal      : NORMAL0;
      float2 TexCoord    : TEXCOORD0;
};
```

Ez a struktúradeklaráció azt jelzi a fordítónak, hogy szemantika szerint, a **Position** változó egy négykomponensű lebegőpontos vektor legyen, mely a **POSITION0** regiszterben található, illetve, hogy ezen kívül érkezni fog a **NORMAL0** csatornán egy 3komponensú normálvektor és a **TEXCOORD0** csatornán egy, a vertexhez tartozó kétdimenziós textúrakoordináta. Ezek az adatok a külső program által lesznek adagolva (**drawUserPrimitives(..)** hívás) vagy pedig a videokártya saját memóriájában található

vertex bufferből(**drawPrimitives(..), drawIndexedPrimitives(..)**) lesznek kiolvasva.

```
struct VertexShaderOutput
{
    float4 Position      : POSITION0;
    float2 TexCoord      : TEXCOORD0;
    float3 Reflection    : TEXCOORD1;
    float3 Fresnel       : COLOR0;
    float3 Lighting      : COLOR1;
};
```

A fent látható struktúradeklarációval tulajdonképpen azt határozzuk meg, hogy a struktúra **Position** nevű eleme egy float4 vektor típusú, szemantikailag a **POSITION0** regiszterben tárolódó változó. Ezzel analóg módon helyezzük el a textúrakoordinátákat, a visszatükröződési vektort, az ún. **fresnel együttható** közelítését és a fényszámításhoz szükséges információkat.

A **vertex shaderünk** a futása során először minden egyes bejövő vertexet **transzformál** a **world**, a **view** és a **projekciós mátrixokkal**, előállítva ezzel a végleges pont koordinátáját, majd a textúrakoordináták szimpla átemelését követően a következő lépéseket hajtja végre:

1. Kiszámítjuk a vertex világtérbéli normálvektorát a felület által meghatározott, tangenstérbeli normálvektornak a **World** mátrix-szal való szorzásával.
2. Kiszámítjuk a kamera pozícióját, úgy, hogy a **View** mátrix eltolási komponenseinek negáltját szorozzuk nézeti mátrix transzponáltjával(a transzponálás a felső 3x3-as forgatási mátrixon invertálást jelent, mert az a mátrixrész ortonormált). Ezzel a művelettel a kamera világtér-béli pozícióját kapjuk meg, hiszen a nézeti mátrixot pont egy eltolási és egy forgatási mátrix ilyen sorrendben vett szorzataként definiáljuk.
3. A kettes pont eredményét felhasználva kiszámítjuk a kamera térbeli pozíciójától a vertex felé mutató vektort(**viewVector**), amit a **reflect(..)** beépített shader függvény segítségével tükrözünk a szintén világtérben lévő normálvektorra, megalkotván a visszaverődési vektor inverzét.
4. A kamera irányából a vertexhez mutató **viewVector** normalizáltjának és a felületi normálisnak a skaláris szorzatának eggyel történő növelésével, majd az eredménynek a **saturate(..)** függvény segítségével az [0..1] intervallumba szorításával kapjuk meg a **fresnel** együttható közelítését, amit majd arra használunk fel a pixel shaderben, hogy a tükröződés és az eredeti textúraszín értéke között lineárisan interpoláljunk. Ezzel a megoldással azt érjük el, hogy azon felületek, melyekre a kamera közel derékszögben néz nem fognak tükröződni, amelyeket pedig közel a felülettel párhuzamosan néz erősen fognak tükröződni.
5. Végül hasonló megfontolásokból a felületi normális és a fény irányvektorának a skalárszorzásának eredménye és a 0 érték maximumaként kiszámoljuk, a fény mennyire erősen világítja meg a felületet, majd a **Lighting** vektorba beleírjuk az **ambiens fény + diffúz fény * fényerő értéket**.

Ezek után a pixel shader már csak annyit csinál, hogy kiolvassa az alapszínt a textúrából

a **tex2D(..)** függvény és a megfelelő **TextureSampler** objektum segítségével, majd kiolvassa a cubeMap textúrából is a megfelelő színértéket a **texCube(..)** függvény segítségével és lineárisan interpolál a kettő között a fresnel összefüggés alapján. Az interpoláció után a keletkező színértéket a fényszámítások eredményeként előállt vektorral szorozzuk meg, majd visszaadjuk a kimeneti színt, kiegészítve az eddig számoltakat a negyedik, **Glow** komponenssel.

Az effect fájl végén a shaderek mögött még megadjuk, hogy a futás egyetlen menetet igényel és deklaráljuk a használt fordítót is a **shader modell 2.0** megadásával.

3.5.1.5 Megjegyzések a shader effect filehoz

- A shader program fény irányvektorral dolgozik, de a főprogramunkban a fény koordinátája is megváltoztatható, ami a nagyobb hajók esetében megvalósult pontosabb fényszámítások miatt szükséges. A helyes eredmény azonban közelíthető a főprogramban egyetlen változó használatával, ami a jelenettől egy kellően távoli fényforrás pozíciót reprezentál. Ezt tesszük mi is.
- A cubamap jellegű textúrában való helyes lookup művelet elvégzéséhez felhasználtuk a **texCube** beépített intristic függvényt, mely előre adott a **HLSL** shader környezetben, így nem kellett magunknak kiszámolni az indexelést.
- A cubemapból olvasáshoz deklarált **TextureSampler** objektum beállításánál az u és v textúrakoordináták **clamp**elését(azaz 0 és 1 közé történő szorítását) állítottuk be, hogy számítási hibák esetén úgy olvassunk ki helytelen színértékeket, a cubeMap sarkainál, hogy azok a legutolsó indexű koordinátából kerüljenek ki. Hasonló módszert alkalmazunk majd a **skybox**nál is, mert ott is fontos, hogy a kocka szélein ne jelenjenek meg oda nem illó színérték artifactok, hanem inkább legyen illeszkedő a széle a textúráknak és kicsit ismétlődjön a széleken pár pixel.
- A shader programok paramétereinek a beállítása a külső programból úgy történik, hogy

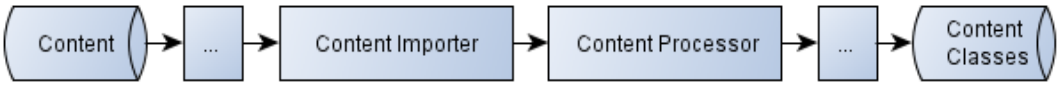
```
foreach (ModelMesh mesh in m.Meshes){
              foreach (Effect effect in mesh.Effects){
                  effect.Parameters["Glow"].SetValue(1.0f);
                  effect.Parameters["eyePosition"].SetValue(eye);
                   ...
```

A fenti módszerrel egy adott **Model** osztály meshjeinek a rajzoláskor átküldendő paramétereit beállíthatjuk. Ezeket nem törli ki egy draw hívás, így amennyiben a paraméter értéke nem változik meg, nyugodtan hagyhatjuk őket változatlanul, nem kell szüntelenül beírni az értékeket.

3.5.1.6 Az XNA framework content pipeline rendszerének áttekintése

Az XNA biztosít egy ún. content pipeline rendszert, melynek lényege, hogy különböző fájlformátummal rendelkező, de alapvetően azonos típusú tartalmak fordítási időben preprocesszálásra és átalakításra kerülnek és úgy helyeződnek el a végleges bináris mellé. Ezen átalakított tartalmakat az XNA alapkiszerelésű belső osztályreprezentációjában már átlátszatlanul kezelhetjük, textúraként, modellként vagy egyéb osztályba tartozó tartalomobjektumként.

A rendszer a fent említett átalakítást részekre bontva végzi el, csővezetékszerűen, melynek 2 fő pontján is felüldefiniálhatjuk az alapvető működést:



Számunkra csupán a csővezeték Importer és Processor lépése fontos igazán, hiszen a legtöbb esetben ezeket kell felüldefiniálnunk, mert az ábra kipontozott részein olyan osztályok vannak, amik az egyes Importerek fájljainak olvasását, írását és egyéb menedzselését végzik.

A **Content Importer** objektumok azért felelnek, hogy különböző fájlformátumú, de azonos jellegű tartalmakat köztes nyers adatokká alakítsanak. Így lehetővé téve például a *.x és a *.fbx típusú fájlok számunkra (már a **ContentProcessor** osztályban is) átlátszó kezelését. Míg a **ContentProcessor** osztályok felelnek a nyers köztes adatok végleges formára alakításáért és mindenféle fordítási idejű pre-processing lépésért.

A programunk során csak **ContentProcessorokat** fogunk írni, mivel mi csupán átalakítani akarjuk az amúgy is esetlegesen különféle formátumban adott modelljeinket, illetve mert a **ContentImporter** osztályokat csak abban az esetben célszerű használni, amennyiben ki szeretnénk terjeszteni az XNA framework által beolvasható tartalomfájlok listáját, ami nem célunk a beépített sokféle tartalmat kezelni képes importer osztályok miatt.

3.5.1.7 A környezetleképezést használó, a vadászgépek betöltéséhez átalakított tartalombetöltő rendszer

Mivel XNA alap modellbetöltő rendszere nem támogatja az environment mapok beolvasását, főleg nem úgy, hogy egy egyszerű, nem cubemap textúrából alakítjuk ki őket, szükségessé válik a tartalombetöltő rendszer átalakítása. Ezt a feladatot úgy oldhatjuk meg, hogy létrehozunk egy projektet, melyben megvalósítunk néhány **content pipeline extension**-t, melyet felhasználhatunk adott modellek speciális betöltéséhez, a modellhez fordítási időben meghatározható pre-processing adatok hozzáadásához.

A rendszert ezen kívül úgy alakítjuk ki, hogy az elérhető legyen a fejlesztői környezetből is, így téve egyszerűvé a modellekhez való betöltő hozzárendelését.

A rendszer legmagasabb szintű eleme a **StarfighterEffect1ModelProcessor** osztály, mely a beépített **ModelProcessor** osztályból öröklődik. Ez a **ContentProcessor** kiterjesztés lesz hozzárendelve a vadászgépek űrhajómodelljeihez.

A **StarfighterEffect1ModelProcessor** annyiban különbözik a hagyományos modellprocesszortól, hogy:

- Felüldefiniálja az eredeti modellbetöltő, anyagkonvertáló függvényét:

```
MaterialContent ConvertMaterial(MaterialContent material,
                                ContentProcessorContext context)
```

- Bevezet egy új publikus property-t mely a fejlesztőkörnyezetből is látható:

```
private string environmentMap = "env_green.bmp";
  [DisplayName("Environment Map")]
  [DefaultValue("env_green.bmp")]
  [Description("The environment map applied to the model.")]
```

```
public string EnvironmentMap
{
    get { return environmentMap; }
    set { environmentMap = value; }
}
```

- Az anyagok konvertálását a **ConvertMaterial** függvényben pedig a saját **StarfighterEffect1MaterialProcessor** nevű saját anyagátalakító osztállyal végzi el, melynek átadja az új **EnvironmentMap** property paramétert is beállításra az ősosztály szükséges paramétereivel egyetemben.

A modell betöltése során tehát a következő lépcsőfok a **MaterialProcessor**-ból származó **StarfighterEffect1MaterialProcessor** megvalósítása, mely abban csupán különbözik ősosztályától, hogy:

- Felüldefiniáljuk az átalakító **Process(..)** függvényt
- Felüldefiniáljuk az anyagban lévő textúrából osztályokat képező **BuildTxture(..)** függvényt
- Itt is bevezetjük a **ModellProcessor** osztályunknál látott environmentMap property-t.

A Process függvény átalakítása azt eredményezi, hogy az anyaghoz a mi saját „Shaders\\StarFighterEffect1.fx” shader effektusunk lesz hozzárendelve, illetve hozzárendeljük az anyaghoz az eredeti és az általunk adott environment textúráját is.

A BuildTexture függvényt azért definiáljuk felül, mert hátra van még a cubemap textúra preprocessz-jellegű legyártása, melyet egy saját **CubemapProcessor** osztállyal valósítunk meg, mely az alap generikus **ContentProcessor** osztályból származik:

```
class CubemapProcessor :
ContentProcessor<TextureContent,TextureCubeContent>
```

A generic paraméterezés alapján látható is, hogy egy hagyományos textúrából Cube típusú textúrát állítunk elő. Valószínűleg ez a legbonyolultabb **ContentProcessor** osztályunk, hiszen egy adott típusú tartalom teljes, típusán is átívelő átalakítását végezzük el. Az osztály első ránézésre esetleg bonyolultnak tűnhet, de a megvalósított módszer elméleti leírása után, valószínűleg könnyen felfoghatóvá válnak az egyes metódusainak a szerepei.

A textúrabeolvasó **ContentProcessorunk működési elve** a következő:

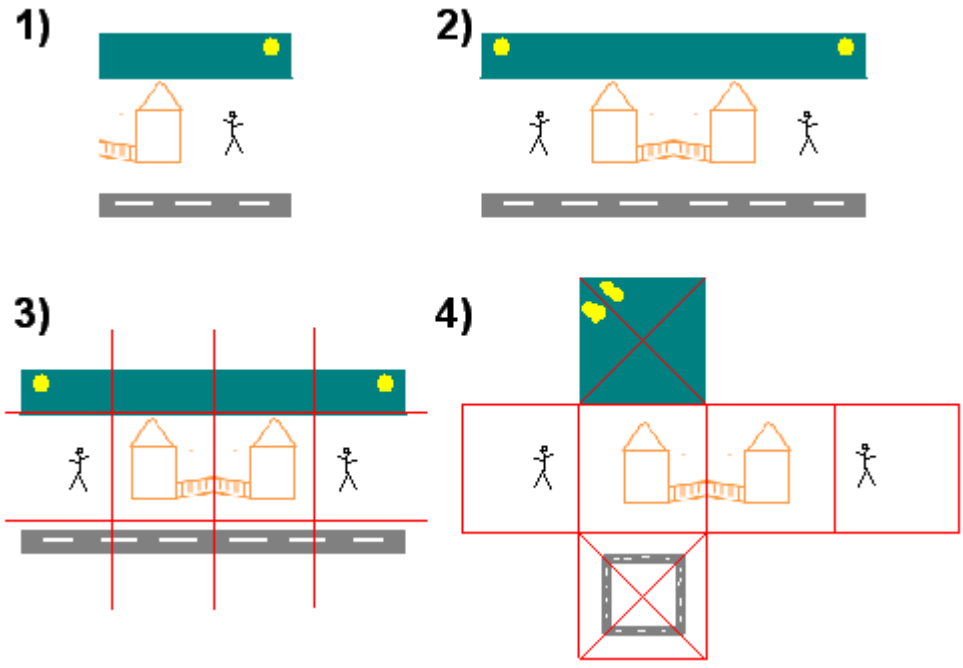
- Átalakítjuk a textúrát biztosan tömörítetlen formátumba
- Előállítjuk a bemeneti alap textúra horizontálisan kétszer akkora megfelelőjét úgy, hogy vízszintesen tükrözzük a textúrát és maga mellé helyezzük. Ezt a műveletet végzi el a **MirrorBitmap(..)** függvény.
- Létrehozunk egy új **TextureCubeContent** típusú objektumot, amibe előállítjuk a kocka 6 oldalának megfelelő textúrákat úgy, hogy:
    - A vízszintesen kétszer akkora textúrát 3 sorra és 4 oszlopra bontjuk.
    - A középső sor 4 oszlopa által meghatározott négyzetekből előállítjuk a kocka oldalsó lapjait a **CreateSideFace(..)** függvény segítségével.
    - Az alsó sor 4 oszlopa által meghatározott négyzeteket háromszöggé fajuló trapezoiddá nyomjuk össze és az alsó oldal előállításához

behajtjuk, a **CreateBottomFace(..)** függvénnyel.

- A felső sor 4 oszlopa által meghatározott négyzetekkel az alsó oldalhoz tartozó négyzetekhez analóg módon konstruáljuk meg a felső kockaoldalt a **CreateTopFace(..)** függvény felhasználásával.
- Az utóbbi két lépés véglegesítése előtt egy egyszerű blur filterrel elmossuk először függőlegesen, aztán vízszintesen a keletkező eredményt, hogy elrejtsük a pontatlanságokat. Ezt a lépést végzi el a **BlurCubemapFace(..)** függvény, mely ideiglenes textúrák segítségével alkalmaz a paraméterén két különböző irány menti **ApplyBlurPass(..)**-t. A blur filter egy egyszerű lineáris általgolás(nem a glow effektushoz használt gaussian blur) annyi különlegességgel, hogy a kockaoldal közepéhez közelebb lévő pixeleket erősebben, míg a szélekhez közel eső pixeleket egyáltalán nem bluroljuk. Ez utóbbi azért fontos, mert a kocka többi oldalára nem alkalmaztunk blur filtert, így feltűnő lenne az illeszkedésnél a turpisság.

- A cube textúrához ezután kigeneráljuk a mipmapokat és visszaalakítjuk dx1 formátumúvá a képet, majd visszaadjuk a hívó félnek.

A létrehozás lépéseit nem részletezzük jobban, de a következő ábra jól szemlélteti a lényeget, hogy mi történik:



Megjegyzendő, hogy a keletkező cubemap textúra dimenziója a **cubemapSize** nevű konstans integer változóban van megadva, ami jelenleg **256x256** pixel, vagyis viszonylag alacsony felbontású. Ezt azonban nem vihetjük ki a **Settings** osztályba mint paraméter, hiszen a **Settings** osztály ilyen jellegű megvalósításának eredeti célkitűzése az, hogy egy helyre kerüljenek a programból változtatható megjelenési és egyéb paraméterek, viszont ez a lépés még fordítási időben történik meg, így nem lehetséges

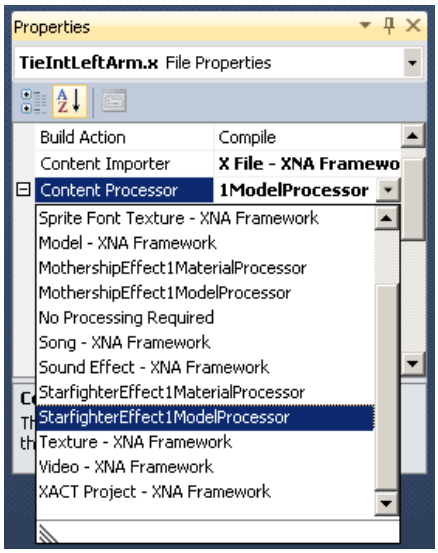
és ezt a számot paraméterezhetővé tenni.

3.5.1.8 A tartalombetöltő projekt osztályainak használata

A tartalombetöltő osztályok használatához, a felhasználó projekt referenciái (References) közé fel kell venni a **StarFighterEffect1Pipeline** projektet és minden, a betöltővel betöltendő modellhez ezt a modellbetöltőt kell hozzárendelni.

Ezt a hozzárendelést a fejlesztőkörnyezetben is megtehetjük ha kijelölünk egy ezzel a rendszerrel betöltésre szánt modellt és a property ablakban a **ContentProcessor** property értékét a legördülő menüvel a mi tartalombetöltő osztályukra állítjuk.

Ugyanezen a helyen, a processzor al-propertyjeként elérhető a modellhez hozzácsatolt cubeMappá alakított textúra is.



3.5.1.9 Megjegyzés a megvalósított módszerhez

A program szerkezetét és a támogatott effektusokat ismerve kritizálható az itt látható megvalósítás, hiszen a játékban alkalmazunk skyboxot, ami eleve 6 darab textúrát jelent és azt a környezetet is használhatnánk cubemap gyanánt. A jelenlegi megoldás persze továbbfejleszthető úgy is, hogy a textúraátalakító fázist elhagyjuk és helyette a skyboxhoz is használt 6 textúrát használjuk fel közvetlenül cubeMap gyanánt, a jelenlegi megoldás implementációja túlbonyolításnak is tűnhet.

Az itt ismertetett megoldás igazából, azért került implementálásra, hogy bemutassuk a preprocess tartalmak megvalósítási módszerét az xna környezetben.

3.5.2 Normal Mapping buckaleképezési technika megvalósítása a nagyobb objektumok kirajzolásához

3.5.2.1 A megoldandó feladat

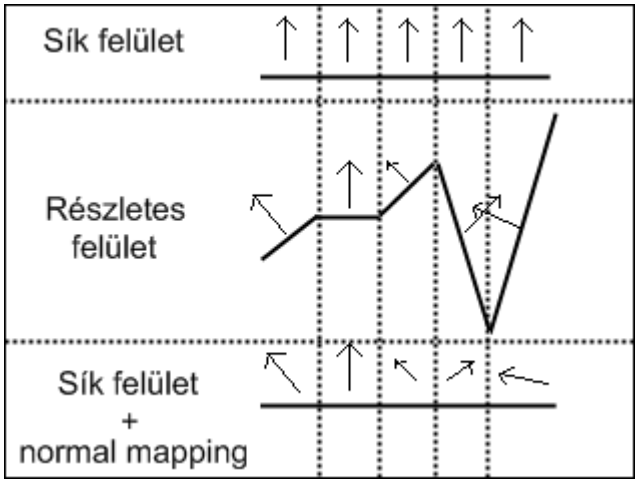
Mint az előző fejezetben említettük, a nagy hajók megjelenésének a szebbé tételéhez érdemes buckaleképezési technikákat használni. Ezek egyik formája a normal mapping, mely viszonylag gyors, könnyen előállítható textúrákat használ és hardveres gyorsítással megvalósítva gyors és szép eredményt képes produkálni.

3.5.2.2 A normal mapping technika elméleti alapjai

A normal mapping, az elnevezésének megfelelően azon alapszik, hogy egy egyszerű felületet bonyolultnak(„buckásnak") tüntessen fel a fényszámítások szokványos modelljeit kihasználva.

Az előző fejezetben láttuk, hogy a fényhatások kiszámításában jelentős szerepe van a felületi normálvektornak, hiszen arra kell tükröznünk a szem, illetve fényforrás vektorokat a számítások elvégzése érdekében. A módszer lényege az, hogy a felületet úgy próbáljuk érdessé tenni, hogy a felülethez a sima textúráján kívül hozzárendelünk egy ún. normal mapot is, mely az adott pixel alatti normálvektor értéket fogja tartalmazni. A pixel shader programban ezentúl nem a felület normálvektorát használjuk a fényszámításra, hanem a normal map textúrából kiolvasott értéket, ezzel a tényleges geometria nem változik meg ugyan, de egy forgó modell, illetve különböző helyzetekből nézett modell felülete, a kamera, illetve a fényforrás és a modell relatív transzformációinak megfelelően változni fog olyan hatást keltve, mintha a modellünk részletessége sokkal több lenne annál, ami látszik belőle.

A következő ábrán kettő dimenzióban, egyszerűsítve demonstráljuk, hogy egy sík felület pontjai milyen normál vektorokkal rendelkeznek, egy részletes felület milyen normálvektorokkal rendelkezik, illetve azt, hogy hogyan kell elképzelni egy sík felületet, melynek pontjaihoz normálvektor-értékeket rendelünk (az ábrán látható függőleges vonalak által elválasztott pontok normálvektorai azonosak):



A normal mapping technikát gyakorlatilag kétféle populáris megközelítésben szokták implementálni aszerint, hogy a fényvektort a vertex shaderben viszik át a felület

bázisrendszerébe, vagy a felület bázisrendszerében adott normálvektorokat visszük át világtérbe a pixel shader futásakor. Mindkét módszernek vannak előnyei és hátrányai, de amíg külön okunk nincs arra, hogy a pixel shaderes megoldást használjuk, jobb a tangenstérbeli normal mappingot alkalmazni, mivel így a lassabb mátrixszorzások a vertex shaderbe kerülnek, ami általában kevesebbszer fut le, mint a minden pixelre (akár többször is) aktivizálódó pixel shader.

Egy felülethez tartozó normal map textúra manapság már generáltatható különböző háromdimenziós modellező programokban, egy kisebb felbontású modellhez egy nagyobb felbontású geometriájúból, illetve léteznek olyan eszközök is, melyek egy egyszerű színértékeket tartamazó textúrából próbálnak ügyes módszerekkel normal mapot előállítani úgyhogy előállítási lehetőségekkel el vagyunk látva. Viszont az XNA környezet alapkiszerelésben nem támogatja ezeknek a normal mapoknak a beolvasását, így megint nekünk kell saját **ContentProcessort** írnunk, melynek implementálási fázisait itt már kevésbé fogjuk részletezni, mint az előző fejezetben.

3.5.2.3 A normal mapping technika konkrét megvalósításának az áttekintése

Mivel nem vagyunk rászorulva arra, hogy a pixel shaderben transzformáljuk ki világtérbe a normálvektorokat (ezt a megkötést például a postprocesszként megvalósított fénybevilágítási rendszerek – deferred shading technikák, hozhatnak be) és mivel a nem csúcskategóriás videó kártyák pixelenkénti fillrate-je alacsony, azt a megoldást választjuk, hogy a vertex shader kódjában transzformáljuk be a szükséges vektorokat a felület bázisrendszerébe és használjuk a pixel shaderben, tehát röviden tangenstérbeli buckaleképezést használunk.

3.5.2.4 A buckaleképezést használó, a nagy objektumok betöltéséhez átalakított tartalombetöltő rendszer

A tangenstérbe transzformáló mátrix előállításához szükségünk lesz 3 vektorra, melyek ortonormált bázisrendszert alkotnak úgy, hogy a textúratér síkjában van két bázisvektor, a harmadik pedig a felületre merőlegesen. Ezt kiszámolhatnánk manuálisan is, minden egyes háromszögre, mert a csúcspontok ismeretében vektoriális szorzással kihozható az elvárt eredmény, de szerencsére ezzel már azonban nem kell törődnünk, mert az xna keretrendszer **ModelProcessor** osztályát be lehet úgy állítani, hogy generálják ki nekünk az ún. tangens és binormális vektorokat, melyek a normálvektorral együtt ortonormált bázist adnak nekünk a tangenstérben, így elég a Processor osztály tagjait csak annyira felüldefiniálni, hogy a **GenerateTangentFrames** property értéke mindig igaz legyen és módosítani a modellek beolvasásához szükséges egyéb, egyszerűbb technikai jellegű dolgokat.

Arra is ügyelnünk kell, hogy egy nagyobb űrhajó szinte biztosan tartalmaz több textúrát is és nagy valószínűség szerint szükség van valamilyen megoldásra, az egyes színtextúrákhoz tartozó normal mapok könnyen kezelhető, automatikus beolvasására. Ezt a beolvasást úgy is csinálhatnánk, hogy a modellfájlban eltárolt speciális adatokkal, vagy a modellező program által hozzárendelt normal map adatokkal dolgozunk és ez alapján olvassuk be a modellt, ám ezzel a megoldással (minden szépsége ellenére) előáll az a probléma, hogy különböző modellező eszközök különböző mértékű támogatást nyújtanak az egyébként sokszor szabványos modellformátumokhoz is. Ez az út tehát

igazából csak akkor járható biztosan, ha stabil modellezői gárda van egy projekt mögött és tudjuk pontosan ki milyen formátumot használ és a formátumba milyen programmal exportál. A mi esetünkben nem ilyen jó a helyzet és mi szeretnénk minél szélesebb spektrumot lefedni a lehetséges felhasználható tartalmak tekintetében, ezért a fent vázoltnál egy sokkal rusztikusabb megoldást választunk: Egy adott modell vagy az előző fejezet environment textúrájához hasonlóan egyetlen normális térképhez köthető lesz, melyet egy propertyben állíthatunk be, vagy pedig ezt a property-t az **<auto>** értékkel látjuk el, ami azt fogja jelenteni, hogy amennyiben a tartalom betöltésekor találunk egy * nevű valamilyen (grafikus kép) kiterjesztésű textúrát, azonnal megpróbálkozunk a ***_bump** nevű, azonos kiterjesztésű textúra keresésével. Ez egy egyszerű megoldás, de a gyakorlatban igen könnyen használható és ha kell kiterjeszthető.

A tartalombetöltőnk most is 3 osztályból áll:

- **MotherShipEffect1ModelProcessor**
- **MotherShipEffect1MaterialProcessor**
- **NormalMapTextureProcessor**

Melyek az előző fejezethez analóg módon származnak a beépített tartalomprocesszor osztályokból és hasonló struktúrájúak.

A **MotherShipEffect1ModelProcessor** osztály a **GenerateTangentFrames** property felüldefiniálásán túl tartalmaz egy **normalMapTexture** publikus és a fejlesztőkörnyezet számára is megjelenítődő stringet(„<auto>” kezdőértékkel), leellenőrzí hogy a string nem-e üres, definiálja a modell importer osztály által készített vertex csatornák közül a számunkra hasznos, elfogadhatóakat (ezek a `TextureCoordinate(0), Normal(0), Tangent(0)` és a `Binormal(0)`csatornák) majd a **ProcessVertexChannel** metódus felülírásával megvalósítjuk, hogy csak ezeket a csatornákat fogadjuk el az importertől, végül pedig az előző fejezethez analóg módon, a **ConvertMaterial** felüldefiniálásával átirányítjuk az anyagkonvertálást, a mi **MotherShipEffect1MaterialProcessor** osztályunk irányába, annak felparaméterezését követően.

A **MotherShipEffect1MaterialProcessor** osztályban felüldefiniáljuk a **Process()** metódust, melyet ezen a ponton módosítunk úgy, hogy a meglévő textúrákhoz hozzáadjuk a fent említett „_bump”-al kiegészített nevű normal mapokat amennyiben <auto> beállítást érzékelünk, illetve minden egyéb esetben a propertyben megadott normal mapot töltjük be a második textúrának. A MaterialProcessor végül, de nem utolsó sorban felüldefiniálja a BuildTexture metódust is, melyben attól függően, hogy a mi normal mapunk textúráiról van-e szó, a saját **NormalMapTextureProcessor** osztályunk felé irányítja.

A **NormalMapTextureProcessor** osztály által felüldefiniált Process metódusa pedig egy meglehetősen egyszerű preprocessz lépést valósít meg: normalizálja a vektorok hosszát, hogy ezt a shader programban ne kelljen runtime folyamatosan megtennünk!

3.5.2.5 A buckaleképezést megvalósító MotherShipEffect1.fx shader effect

Bemeneti shaderkonstansok:

- **World**, **View**, **Projection** mátrixok.

- **LightPosition**, **eyePosition**, **LightColor** és **AmbientLightColor** 3-3 illetve 4-4 dimenziós vektorok.
- **Shininess** és **SpecularPower** lebegőpontos változók a phong shading-jellegű fényképzési modell paraméterezéséhez.
- **Texture** és **NormalMap** textúrák a megfelelő sampler objekumokkal.

A vertex shader futása:

- Meghatározzuk a végleges kimeneti vertexpozíciót a **World, View, Procejtion** mátrixokkal szorzás alapján.
- Keresztülküldjük a kapott textúrakoordinátákat
- Kiszámoljuk a fényforrás irányát, a **lightDirection**-t.
- Kiszámoljuk a szemtől a vertex felé mutató **viewDirection** vektort
- Előállítjuk a tangenstérből a világtérbe transzformáló ortonormális mátrixot:
    - Létrehozunk egy üres 3x3-as mátrixot
    - Oszlopvektorainak megadjuk a tangens, normál és binormál vektorok világtérbe transzformált változatát.
    - Így egy ortonormált rendszert kapunk, mellyel a felület koordináta-rendszeréből a világtér koordináta-rendszerébe tudunk alakítani.
    - Eredmény: **float3x3 tangentToWorld;**
- A fent előállított mátrix transzponáltja egyben annak inverze is (ez az ortonormális mátrixok egyik tulajdonsága), így könnyen előállíthatjuk a fordított irányban transzformáló mátrixot is(transzponálással), mellyel a világtér-béli vektorokat vihetjük be a tangenstérbeli bázisba.
- A fent számolt lightDirection és viewDirection vektorokat bevisszük a tangenstér-szerinti bázisba, a **tangentToWorld** mátrix transzponáltjával való szorzással. Közben kihasználjuk a HLSL egy lehetőségét, miszerint támogat oszlop-orientált és sor-orientált mátrixszorzási módot is, attól függően, hogy mul(float3, float3x3), vagy mul(float3x3, float3) alakban végezzük a szorzást, így a szokásostól eltérően most az utóbbit választjuk, mely pont megfelel a transzponálttal szorzásnak és nem igényel tényleges transzponálási műveletet.

A pixel shader futása:

- Normalizáljuk a kapott **viewDirection** és **lightDirection** értékeket
- Mintavételezünk a **normalMapSampler** segítségével, majd normalizáljuk az értéket.
- A kapott normálvektorral az előző fejezetben ismertetett módon kiszámoljuk a diffúz bevilágítást.
- Specular lightingot számolunk úgy, hogy a fényirányvektort a normálvektorral tükrözzük és vizsgáljuk a tükrözött vektor, illetve a view vektor kapcsolatát. Azaz, hogy mennyire néz egy irányba a két vektor. Az eredményt a

SpecularPower nevű shaderkonstanssal hatványozzuk és kiszámoljuk a **specular** float4 színértéket.

- Kiolvassuk a hagyományos textúrából az ott található színértéket.
- Végül visszaadjuk a (**(diffuse** + **AmbientLightColor) * diffuseTexture** + **specular)** értéket.

Megj.: Kihasználjuk, hogy a PixelShader textúrakoordináta inputját(meg a legtöbb másikat is) a rendszer interpolálja a vertexek közötti pixelekre a vertex shader által kiszámított értékekből, így nem vesztünk különösen minőséget a nem tangenstérbeli normal mappinghoz képest!

3.5.3 Skybox technika megvalósítása a játék csillagközi háttérkörnyezetének a megteremtéséhez

3.5.3.1 A megoldandó feladat

Mivel a játék űrben játszódik, az ember elvárja a különböző égitesteknek, csillagoknak és bolygóknak, azaz az űrbéli környezetnek a hiteles megjelenítését. Mivel ezek az objektumok nagyon sokan vannak, viszont szinte elérhetetlenül távol, nem érdemes ténylegesen háromdimenziósan jeleníteni meg őket.

A programunkban egy, a háromdimenziós játékfejlesztést már a kezdetektől jellemző trükköt vetünk be a háttérkörnyezet megjelenítésének az érdekében, méghozzá a skybox technikát, mely lényegében egy dobozra festett környezetábrázolás.

3.5.3.2 A skybox technika elméleti alapjai

A skybox techika lényege, hogy minden egyéb rajzolás előtt kirajzolunk egy kockát a kamera jelenlegi pozíciója köré, figyelembe véve annak elforgatását méghozzá úgy, hogy a kocka oldalait távoli tájak képével textúrázzuk. Mivel az ilyen módon megrajzolt objektumok nagyon távoliak, a játékos nem veszi észre, hogy nem valódi háromdimenziós képet lát (mondjuk egy szép bolygóról), hanem csupán egy előre elkészített képet, mely a kocka 6 oldalára van ráhúzva.

Ennél a nagyon régi megoldásnál, ma már sokkal fejlettebbeket is szokás használni, mert a rosszul előállított skybox textúrák illeszkedési pontjaiban észrevehetővé válik a trükk, mégis egy űrhajós játéknál az űr többnyire fekete üressége miatt ez a probléma elhanyagolhatóvá válik és az sem utolsó szempont, hogy igazából csupán a skybox-ok minősége miatt kellene aggódnunk, mert elméletileg lehetséges egy skyboxot úgy is előállítani, hogy az bizonyos feltételek mellett perspektívahelyes legyen (csak az emberek lustasága szab határt egy skybox minőségének).

3.5.3.3 A skybox technika konkrét megvalósításának az áttekintése

A mi skybox megvalósításunk a **DrawableGameComponent** osztályból fog származni, mivel azt szeretnénk, hogy a játékunk minél több komponense hordozható legyen és más projektekbe könnyen legyen beépíthető, akár olyan helyen is, ahol a játékképernyő menedzsmentet nem a játékkomponensekkel oldották meg hanem másként, és igény van a komponensek rendes játék-objektumhoz való hozzáadására.

A **SkyBox** komponensen kívül definiálunk egy saját kis shadert a **SkyBox.fx**-et is, melyben elhagyunk minden felesleges számítást(pl. fényeket és hasonlókat) az overhead legminimálisabb megjelenését is kizárva.

Ezen kívül a kocka vertexeit előre ki fogjuk számolni és a videókártya saját memóriájába tároljuk el, vertex és index bufferek segítségével és attól is eltekintünk, hogy olyan kockaoldalt jelenítsünk meg, ami a kamera mögött helyezkedik el. Megjegyzendő, hogy ez utóbbi nem azért szükséges, mert a kocka néhány vertexe komoly overhead-et jelentene a vertex shadernek, hanem maga a draw hívás miatt létrejövő kommunikációs költség az, ami miatt megéri spórolni ezen a ponton, mivel a kocka egyes oldalait a különböző textúrájuk miatt, különböző draw hívásokkal rajzoljuk, ami egy minimális adatküldési költséget jelent.

A skybox technika egyébként viszonylag egyszerű játékfejlesztési technika.

3.5.3.4 A skybox inicializációjának áttekintése

A skybox inicializációját a **konstruktora** és az **Initialize()** metódusa végzi el.

A **konstruktor**ának paraméterül a fő játék-osztályt (kommunikációs megfontolásokból), a használni kívánt skybox helyét és a skybox X,Y és Z irányban az origótól vett maximum távolságát – kiterjedését kell megadni. A skybox nevének megadásával azt érjük el, hogy az alapértelmezett content projekt SkyBoxes alkönyvtárában lévő, megadott nevű elérési úttal rendelkező alkönyvtárában lévő **back, bottom, front, left, right, top** elnevezésű asset-et használja textúrának a skybox. Fontos, hogy nem a filenév az ami megadandó, hanem a fájlhoz hozzárendelt **Asset Name**. Ez utóbbi így van minden egyéb helyen is, ahol külön nem említünk külső fájlt. Az **Asset Name** property egyébként a fejlesztőkörnyezetben könnyen látható és módosítható az adott tartalom kiválasztásával annak properties ablakában. Itt kell módosítani tehát, amennyiben új tartalmat adunk hozzá (vagy akár kihasználhatjuk, hogy alapesetben, az adott tartalomhoz a kiterjesztés nélküli neve kerül **Asset Name**-ként beírásra)

A konstruktor tehát menti a fent kapott három paramétert, de semmi egyebet nem csinál. A legtöbb inicializáló lépést tehát az **Initialize()** metódus fog végrehajtani.

Az Initialize() elvégzi a következőket:

- Textúrák betöltése
- Shader effect fájl betöltése
- Vertexbuffer létrehozása saját vertexdeklarációval
- Vertexbuffer adatainak kiszámolása és áttöltése a videomemóriába
- Indexbuffer adatainak kiszámítása és átküldése a videomemóriába

3.5.3.5 Vertex és Index bufferek, vertexdeklarációk

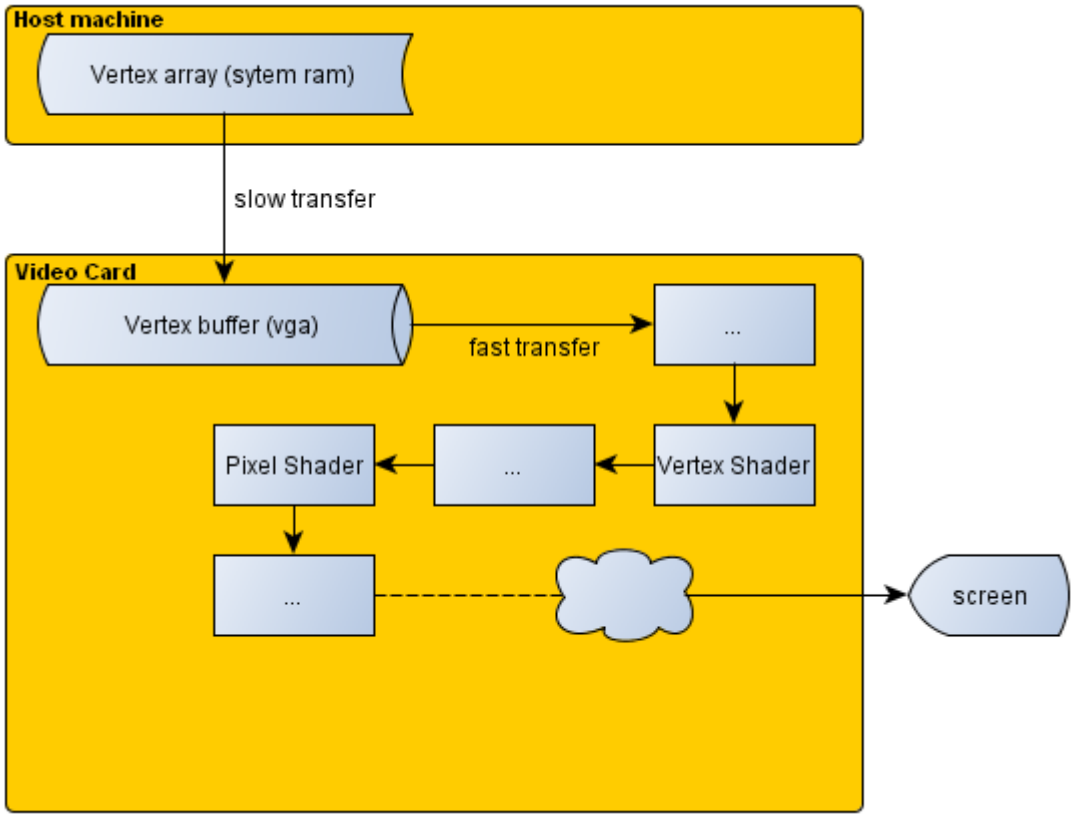
Az inicializáló metódus megértéséhez szükséges áttekintenünk azt, hogyan működnek a videó kártyán található vertex és index pufferek, illetve hogyan határozzuk meg a vertex shader bemenetét képező adatfolyam beosztását, illetve azt, hogy milyen input-csomag érkezik a vertex shaderhez vertexenként.

Vertex pufferek:

Korábban említettük, hogy a videokártya tekinthető egy csővezetéknek, mely tartalmaz jó néhány feldolgozó egységet és a host architektúra által küldött adatfolyamot alakítja át végleges képpé, mely a képernyőn megjelenik. Ez így többé-kevésbé igaz is, de ennél a lehetőségek sokkal bővebbek. Jelen esetben ez a fent vázolt architektúra annyiban változik meg, hogy a videokártyáknak nagyméretű belső memóriájuk van, melyeket főleg belső célra használnak, de arra is használhatóak, hogy olyan vertex-inputot tegyünk beléjük, amelyek szinte sose változnak meg, így spórolva meg, a rendszermemóriából a kártyára történő felesleges adatküldés kommunikációs idejét.

A vertex adatoknak a videókártyán történő tárolása ún. vertex pufferekkel történik. Ezek olyan objektumok, melyek a videókártyán lévő memóriát reprezentálják és általában lassan írhatók és olvashatók cpu oldalról nézve (de ha már egyszer ott vannak akkor gyorsan eléri őket a videokártya), ezért tényleg csak azt érdemes bele tenni, ami a programfutás alatt ritkán változik meg. Ez a legutolsó mondat persze csalóka, mert szinte minden belefér a fent említett kategóriába a jelenlegi játékunk grafikus elemei közül, ugyanis a vertexek térbeli koordinátáit egyenként nem változtatjuk meg például egy űrhajó mozgásakor (vagy forgatásakor), mert ilyenkor csak a vertexeket transzformáló mátrixok változnak. Általában elmondható, hogy csak akkor nem érdemes vertex puffert használni, ha a térbeli vertex-koordináták folyamatosan és dinamikusan alakulnak a cpu-n történő számítások alapján mondjuk úgy, hogy a modell vertexeit hullámoztatjuk vagy valami hasonló. Ilyeneket mi viszont nem csinálunk, nálunk a modellek és a saját objektumok megtartják alakjukat.

A következő ábra szemlélteti a vertex puffer használatot:



Index pufferek:

A fentiek alapján tehát elmondható, hogy spórolhatunk a kommunikációs költségen az állandó vertex-adatoknak a videokártyára helyezésével, ám ha csak a fent említett pufferrel rendelkeznénk, sok esetben szükségünk lenne vertex adatoknak a megismétlésére – olyan vertexek újbóli tárolására, melyek már korábban szerepeltek.

Erre azért van szükség, mert igazából a legtöbb esetben nem csak térbeli pontok halmazát szeretnénk megjeleníteni, hanem ezekből valamilyen rendszert alkotva például háromszögeket – ahogy azt mi is tesszük, tehát meg kell tudnunk mondani, hogy a vertex pufferben lévő adatok háromszögeknek felelnek meg és mondjuk minden harmadik adatcsomag a háromszög egy pontját jelenti. Ezzel pedig máris eljutottunk a vertexek felesleges ismétlődésének a problémájához, hiszen ha ki akarunk rajzolni egy négyzetet, két háromszögből, akkor 6 vertexet kell majd a vertex pufferbe helyeznünk, holott a valóságban egyetlen négyszögről van csak szó. Ezt a problémát hivatottak orvosolni a vertex pufferek a videokártyán, méghozzá úgy, hogy lehetőséget adnak (a fenti példával élve) csupán a 4 vertex deklarációjára és egy kiegészítő index puffer deklarációra, melyben minden háromszöghöz már három index fog tartozni, melyek a vertex pufferen belülre indexelve mutatják meg a vertex helyét. Ezzel a megoldással úgy spórolhatunk helyet a videokártya memóriáján, hogy az index puffereket kisebb indexekkel(általában 16 bitesek) látjuk el, mint a vertexeket leíró adatcsomagok.

Vertex deklarációk:

Eddig tehát odáig jutottunk el, hogy közvetlen adatküldés helyett a videokártyára előzetesen letároljuk a vertexadatot, amit spórolásból indexekkel is elláthatunk, ám arról még nem esett szó, honnan tudja a rendszer, hogy milyen jellegű adatcsomag tartozik egy-egy vertexhez. A vertexek ilyen jellegű típusának megadását a Vertex deklarációkkal végezhetjük el, amiben leírjuk, hogy hány darab adat, milyen típusú csatornán érkezik a videokártyára. Az adatok ennek megfelelően tárolódnak a belső vertex pufferben is.

```
VertexDeclaration vDecl = new VertexDeclaration(new VertexElement[]{
                new VertexElement(0, VertexElementFormat.Vector3,
VertexElementUsage.Position, 0),
                new VertexElement(12, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate,0)});
```

Az itt látható deklaráció például azt adja meg, hogy minden vertexhez tartozik egy háromdimenziós vektor, mely a pozíció csatornába kerül, illetve rendelkezik minden vertex egy kétdimenziós textúrakoordinátával is. A **VertexElementUsage** csak egy csatornát határoz meg, az adat felhasználása sok esetben csakis rajtunk múlik. Mi a jelen esetben a vertexhez tartozó két adatot pont arra használjuk majd, amire valók. Egyikben a síkbeli textúrakoordinátát adjuk meg, ami a háromszög adott csúcsához tartozik, a másikban a térbeli pozíciót a háromszög csúcsához.

3.5.3.6 A kirajzolás

A skybox kirajzolását a draw metódus végzi el, amely a következőket csinálja futása során:

- Kiszámítja a rot nevű mátrixot a következőképp:
  - Összeszorozza a nézeti és a projekciós mátrixokat
  - A keletkező mátrixban nullázza az eltolási tényezőt(legalsó sor)

    - Így pontosan azt érjük el, hogy a skyboxot a kamerának megfelelően elforgatjuk, de el sose nem mozgatjuk, ha a kamera egyébként mozog, tehát a skybox a „kamerával együtt halad”.
- Minden oldalára a skyboxnak:
    - Meghatározzuk, hogy az adott oldalt ki kell-e rajzolni, vagy a kamera mögött van. Ezt a vizsgálatot a kamera irányvektorának és a tengellyel párhuzamosan megválasztott vektoroknak a segítségével ejthetjük meg. Ha nem kell kirajzolni, akkor a második pont nem hajtódik végre.
    - Beállítjuk a használandó vertex és index puffert, illetve a textúrát.
    - Aktiváljuk a shader nulladik menetét (csak az az egy menet létezik)
    - Kirajzoljuk az adott kockaoldalt. A draw hívás során megadjuk, hogy háromszögekként kell értelmezni az adatot és használunk index puffert.

3.5.3.7 Megjegyzések

- A draw hívás során mindig zajlik kommunikáció, hiszen kezdeményeznünk kell egy kirajzolási fázist. Mivel oldalanként külön rajzolunk, célszerű a fenti, olcsón végrehajtható teszt végrehajtása a felesleges draw hívások elkerülésére.
- A rendszer nem a kirajzolások sorba rendezésével, vagy egyéb módon rejti el a távolabbi objektumokat amennyiben egy közelebbi fedi azt, hanem egy ún. **mélységi puffert(Z-Buffer)** használ, mely lényegében annyit jelent, hogy a képernyő minden pixeléhez a 4 színértékén kívül tartozik egy Z(általában 16 vagy 24 bites) érték, mely azt reprezentálja, hogy az adott pixel milyen „mélyen” van a képernyő, azaz esetünkben a kamera síkjában. Amikor egy pixel kiírásra kerül és a **depth-test** engedélyezve van, az újonnan kiírandó pixel mindig megvizsgálja, hogy az ő beírandó mélysége kisebb-e, mint az ott lévő társáé. Ha igen, akkor kirajzolódik az adott pixel és amennyiben a **depth-write** be van kapcsolva beíródik az új mélységi érték is. Ez utóbbi számunkra azért hasznos, mert a skybox potencionálisan végtelen távol lévő objektumokat jelenít meg a felületén, így nem akarjuk, hogy valami közeli objektum felülíródjon a skybox egyik felületével, mert mondjuk a skybox 2000 egység széles a kamera pedig tolatva eltávolodik egy űrhajótól 2000 egységre és a skybox „belevágna” az objektumba és eltüntetné a felét. Ezt a problémát gyökerében elfojthatjuk úgy, ha a skybox kirajzolása előtt beállítjuk, hogy a mélységi puffer ne legyen írható. Ezt a jelenlegi implementációban nem a modul teszi meg, hanem a modult felhasználó programrész!

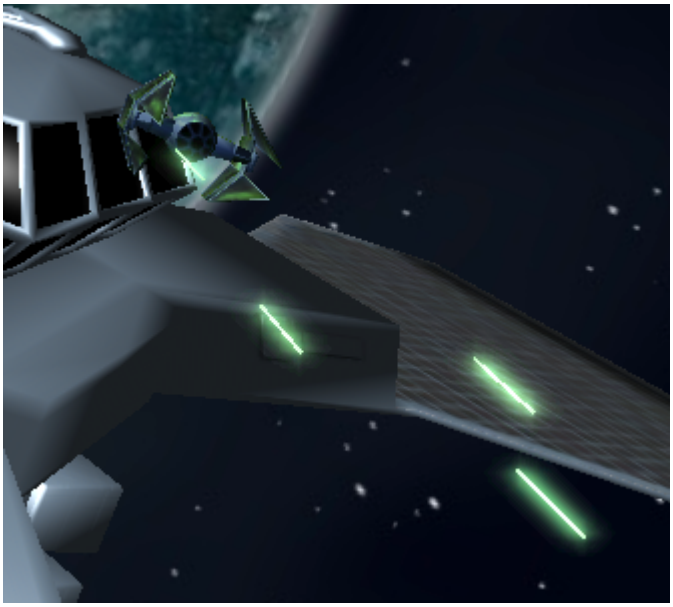
3.5.4 Egy post-processing technika megvalósítása: A selective laserglow effekt

3.5.4.1 A megoldandó feladat

Célunk a csillagok háborúja filmekből jól ismert lézer-effektus implementációja, mely

az űrben száguldó lövedékek köré egyfajta holdudvar-jellegű glow mezőt jelenít meg.

A megoldandó feladat a legegyszerűbben az elkészült játékból vett alábbi kép segítségével fogható meg:



A feladat részét képezi az is, hogy a fent ábrázolt effektust úgy érjük el post-processing műveletként, hogy szelektíven megadható legyen melyik objektum, milyen erős glow hatással rendelkezzen, illetve a feladat megoldása során a lehető legjobban függetleníteni kell a technikát a többi megoldástól.

Megjegyzendő, hogy a képen a lövedék tényleges háromdimenziós kiterjedése csupán az izzó „magja”, a többi már a holdudvar, mely a végleges képre a magból generálódik!

3.5.4.2 Post-processing műveletek megvalósítása modern videomegjelenítő környezetben

Modern megjelenítő környezetekben megkérhetjük a videokártyát, hogy ne a képernyőre, illetve a backbufferbe renderelje ki az adott képet, hanem egy általunk megadott textúrába, melyet később (akár ugyanehhez a képkockához tartozó további renderelési fázisokhoz is) felhasználhatunk.

Egy post-processing technika a névéből értetődően már a kész, síkbeli, elvileg kirajzolt képen fut le mint utókezelés, mely általában több lépésből, menetből áll. Egy ilyen technika futását a fenti architektúrával a hátunk mögött úgy érhetünk el, hogy több lépésben rajzoljuk ki a végleges képet, a köztes lépések végeredményeit ideiglenes textúrákba renderelve, melyeket aztán a legutolsó lépés során egy végleges képpé alakítunk.

Ezzel a módszerrel elvileg tetszőlegesen sok post-process lépést végrehajthatunk, azonban a képkockánként történő túl sokszori kirajzolásnak hamar meglesz az ára, szóval nem érdemes pazarolni és illik minimálisra csökkenteni a kirajzolási menetek számát. Ez utóbbi xboxon vagy kevésbé modern PC-ken még fontosabb, hiszen kevés memória áll a rendelkezésünkre ilyen célból. Az xbox-on például 10megabájtos nagyságrendben van ilyen és hasonló célokra egy gyorsítótár, melyet ezen kívül még

belső tárolásra is használ a gép és bár az alap grafikus memória sokkal nagyobb, artifactok(zajok és hibák) jelentkezhetnek ha túllépjük speciális tárterületet.

3.5.4.3 A megoldás alapötlete

Az ilyen ragyogás-jellegű effektusok megvalósítására nagyon sok példát találhatunk, ám szinte az összes megvalósítás azon alapszik, hogy az egész képernyőre ráengednek egy ragyogást keltő útókezelést és minden dolognak lesz valamekkora holdudvara, ami bár sokszor megfelelő, számunkra nem cél. Mi csak néhány objektumot, szelektív módon akarunk ragyogtatni.

Az eredeti egész-képernyős megvalósítás egyébként azon alapszik, hogy kirajzoljuk az egész-képernyős képet egy textúrába, majd ebből a textúrából kiindulva egy következő textúrába kiválogatjuk az eredeti kép egy bizony határértéknél világosabb pontjait, amit aztán további lépésekkel elmosunk, hogy a környező pixelein kisebb színértékkel, de szerepeljenek, aztán végül az így keletkezett képet hozzáadják az eredetihez.

Az első ötlet ami valószínűleg az ember eszébe jut a szelektív megvalósítást illetően az, hogy a lövedékeket és az egyéb objektumokat külön rendereljük ki egy-egy textúrába és a lövedékek elmosása után az elmosott lövedékek képét az egyéb objektumok kirajzolásából származó képpel összegezzük. Ez a megoldás ígéretesnek és kézenfekvőnek tűnhet első látásra, de megvannak a buktatói, ugyanis a kirajzolás során

- Ha mindenféle körültekintés nélkül kombináljuk a két képet, akkor problémánk lesz a mélységi puffer miatt, ugyanis ez a textúrákba nem mentődik át.
- Bár a mélységi puffereket elérjük pixel shaderből és emiatt tartalmuk kimenthetővé válik, ezzel a megoldással sok tovább renderelési fázis szükségessége merül fel.
- Konvertálások miatt megjelenítési pontatlanságok keletkezhetnek.
- Nem elég megőrizni a lövedékek Z értékeit, hanem szükségesé válik a holdudvarhoz tartozó Z-értékek kiszámítása is.
- XBox-on a szűk speciális memóriát pazaroljuk a sok renderelési fázissal, holott az esetleg más, egyéb effektusok számára is hasznos lehetne. Ez számunkra azért fontos, mert a programnak egyrészt könnyen Xbox-ra átalakíthatónak, másrészt kiterjeszthetőnek kell lennie.

A fentieket ismeretében tehát meggondolandó ez a fajta megközelítés. Szerencsére van egy másik, egyszerűen megvalósítható és gyors megoldás, mely egy akár szigorúnak is tűnő megkötés mellett tökéletesen működik akár egy lassabb gépen is.

Alfa-csatorna használata:

Elhangzott már korábban, hogy a pixeleket nem 3 színkomponenssel, hanem négy színkomponenssel reprezentáljuk(RGBA), ahol a negyediket alfa-csatornának nevezik és szinte minden esetben a félig átlátszó objektumok megvalósítására használják úgy, hogy minél nagyobb ez az érték, annál tömörebb, átlátszatlanabb az adott objektum és minél kisebb, annál átlátszóbb. Ezt hívják alpha blendingnek és ha nem használnánk fel, akkor lenne pixelenként egy szabad lebegőpontos értékünk, melyben tetszőlegesen értéket tárolhatnánk.

Vizsgáljuk meg gyorsan, hol szükséges számunkra az átlátszóság a játékban:

- Lövedékek holdudvara kicsit áttetsző
- Robbanások részecskéinek megjelenítéséhez a részecskék átfedésekor
- Esetleges fejlesztésként a játékos űrhajójának belső nézetének kialakításakor
- Menü szövegének megjelenítésekor
- Űrbéli objektumok félig átlátszó részeihez

A lövedékek átlátszósága megoldható manuálisan, amikor összeállítjuk a végleges képet, a részecskék megjelenítéséhez használható additive blending (majd a robbanás effektussal és a részecskékkel foglalkozó részben részletesebben kifejtjük mi is az), a belső nézet kirajzolható a post-processing után, sőt mivel az mindennél közelebb van a kamerához még Z-buffer se nagyon kell, a menüszöveg pedig ezzel teljesen analóg módon kezelendő. Az egyetlen dolog tehát amit elvesztünk az űrhajók félig-áteresztő ablakainak a megjelenítése, ami sok esetben fel se tűnik és így nem olyan vészes megkötés.

A megoldásunk végül tehát az eredeti, szokványosabb egész-képernyős ragyogási technikákhoz teljesen hasonló lesz, csupán annyit kell ügyeskednünk, hogy a szelekciós lépés során ne a színerősség, vagy valami egyéb dolog, hanem az alfa csatorna szerint szelektáljunk és a dolgokat ezentúl úgy jelenítsük meg, hogy tekintettel legyünk arra amit az alfa csatorna jelent számunkra.

Megjegyzés:

Paradox módon nálunk az alfa csatorna kisebb értéke jelenti majd az erősebb glow-t és a nagyobb értéke (1.0f-el bezárólag) a gyengébbet. Ennek a választásnak technológiai oka van: a legtöbb textúra, anyagjellemző és színérték úgy van optimalizálva, hogy alapból átlátszatlan legyen, azaz 1.0f-et tartalmaz az alfa csatorna. Ha ezt vennénk maximális glow-nak, akkor vagy minden glowolna, vagy minden shadert írhatnánk át úgy, hogy megfordítjuk kirajzoláskor az alfa értéket a nekünk megfelelőre.

3.5.4.4 A megvalósítás főbb komponensei

Megoldásunk 3 fő részből épül fel:

- **GlowComponent** osztály
- **GlowSetting** osztály
- A **Content** projekt **Shaders\Postprocess** alkönyvtárában található effect fájlok

A **GlowComponent** osztály egy **DrawableGameComponent**, mely képes inicializálni a szükséges rendertargeteket és változókat, lehetőséget nyújt a kirajzolások átirányítására, képes kiszámítani és beállítani az effect shaderek konstans paramétereit, illetve a draw metódus segítségével képes kezdeményezni és végrehajtani az egyes post-processing lépéseket a végleges kép előállítása érdekében.

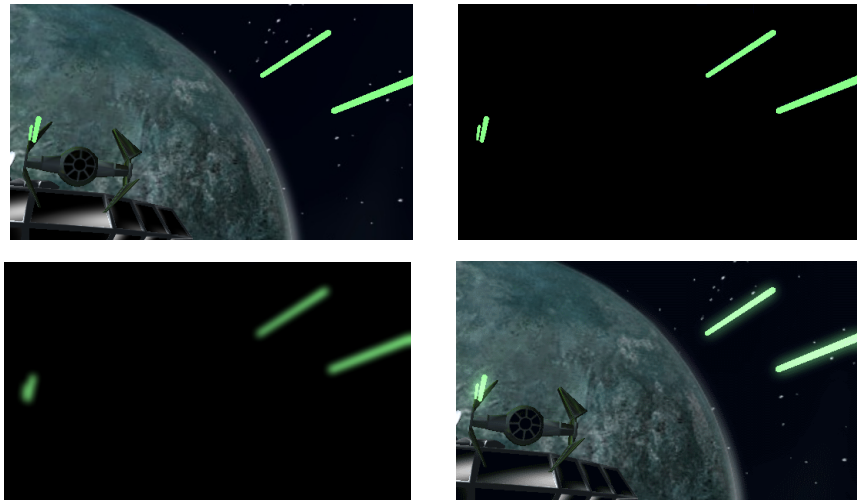
A **GlowSetting** osztály a Glow effektushoz tartalmaz paramétereket. Minden **GlowComponent** tartalmaz egy **Setting** property-t, mely a laserglow effektus paraméterezését tartalmazza.

A **Shaders\Postprocess** könyvtár alatt található *.fx fájlok az egyes post-process kirajzolási menetek által használt pixel shader programokat tartalmazzák.

3.5.4.5 A kirajzolási menetek

A laserglow effektus létrehozásához összesen 5 kirajzolási fázisra van szükségünk.

Először is ki kell rajzolnunk az eredeti képernyőképet egy, a képernyő felbontásának megfelelő méretű textúrába, ezután ki kell válogatnunk az így keletkezett képből a glow effektus számára fontos pontokat, el kell homályosítanunk, azaz blurolnunk a kiválasztott pontokat, majd végül az így kapott képet az eredetivel összekeverve a végleges képet is meg kell jelenítenünk. Az egyes lépésekhez tartozó képeket a következő ábrasorozaton láthatjuk:



A bal felső kép az eredeti, ebből a szelekció során állítjuk elő a jobb felső képet, melyre alkalmazunk egy gaussian blur filtert(a bal alsó kép lesz az eredmény) és végül a bal alsó és a bal felső képek segítségével előállítjuk a végleges képernyőképet, a jobb alsót.

3.5.4.6 A szelekciós fázis

A szelekciós fázis, az eredeti képből válogatja ki a gaussian blur filterezésben előforduló pontokat. A kiválogatást a **SelectGlowEffect.fx** fájlban található pixel shader végzi el a következőképpen:

- Az adott képernyőpont alfa csatornájának értékét csökkenti a **GlowBonus** nevű konstans paraméterértékkel.
- Kiszámolja a **selector** szelekciós együtthatót úgy, hogy a **GlowBonus**-al módosított eredeti képpont alfa csatornájának értékét kivonja 1.0-ból.
- A pixel eredeti színértékeit szorozzuk a kapott **selector**-al.

Tehát összefoglalva a nullához közeli alfa csatornájú elemek az eredeti színükkel

szerepelnek a szelekciós fázis kimenetének képpontjai között, az 1.0 közeliek pedig szinte egyáltalán nem. A **GlowBonus** paraméter szerepe pedig az, hogy a teljes képernyőt is glowoltathassuk, ha mégis ezt szeretnénk elérni.

A szelekciós fázis a bemeneti textúrát az s0 sampler regiszter segítségével olvassa.

3.5.4.7 A gaussian blur filter

A fázisok végrehajtása viszonylag egyszerű módon történik, a legbonyolultabb utófeldolgozás melyet alkalmazunk a gaussian blur filterezés.

A gaussian blur filterezés lényege, hogy a kép pixeleit úgy mossuk el, hogy a környező pixelek átlagát vesszük az adott pontban, méghozzá úgy, hogy átlagszámítás előtt az adott pixeleket, azok távolsága alapján a haranggörbének adott pontjához megfelelő súlyozással tekintjük.

A gaussian blur a mi esetünkben úgy kerül megvalósításra, hogy a pixel shaderben, az előzőleg kiszámolt textúrából a megfelelő ponton mintavételezünk. Az egyetlen probléma ami így előáll, a pixelenkénti mintavételezések korlátolt számossága, mely 32 körüli érték szokott lenni jelentősen gátat szabva ezzel az egy pixelhez történő minták számát és kiterjedését.

A problémát az oldja fel, hogy a gaussian blur filter lineárisan szeparálható tulajdonságú, azaz egyetlen kétdimenziós filterezési fázist helyettesíthetünk két egydimenziós filterezési menettel. Ennek a tulajdonságnak a felhasználásával jó minőségű, de eggyel több menetből álló filterezést fogunk megvalósítani úgy, hogy először vízszintesen, majd ezután függőlegesen mossuk el a képet.

A gaussian blur filterezést megvalósító pixel shader a **GaussianBlurEffect.fx** fájlban található és a pixelenként lefutó program feltűnően egyszerű. Ez azért van, mert a shader számára két tömbben (**SampleOffsets** és **SampleWeights** tömbök), konstans paraméterként kerül átadásra a blurhoz használt mintavételezési pontok koordinátái és azok súlyai, illetve a súlyok úgy érkeznek meg, hogy összegük 1 legyen.

Ennek az ismeretében a shader feladata csupán annyi, hogy pixelenként elvégezze a mintavételezést és a súlyozott átlag kiszámítását, egy ciklussal és nem kell, hogy pixelenként haranggörbét közelítsünk.

A gaussian blur fázisok a bemeneti textúrát az s0 sampler regiszter segítségével olvassák

3.5.4.8 A véglegesítő fázis

A véglegesítő fázist a **FinalGlowEffect.fx** nevű fájlban lévő pixel shader végzi el, mely az s0 samplerben várja a gaussian blur filter kimenetét és az s1 samplerben pedig az eredeti képet. A shader ezen kívül paraméterezhető a **BaseIntensity** és a **GlowIntensity** nevű konstans shader-paraméterekkel is, melyeket a végleges képpont színének az alábbi képlet szerint történő kiszámításához használunk fel:

Pixel = BaseIntensity * base + GlowIntensity * glow

Ahol a base és a glow a két bemeneti textúra megfelelő képpontjának a színértékeit jelenti.

Megjegyzés: Az utolsó fázis shader fájlja tartalmaz egy kikommentezett részt, mellyel paraméterezhetővé válik az egyes bemeneti textúrák színtelítettsége is. Ezt azért nem használjuk fel a végleges megoldásban, mert a megvalósítás által használt lineáris interpoláció nagyon belassítja a program futását egy gyengébb gépen annak ellenére, hogy a legtöbbször nem használjuk ki a paraméterezés által nyújtott lehetőséget.

3.5.4.9 A GlowComponent osztály és használata

A fent leírt lépések végrehajtását a **GlowComponent** osztály segítségével valósíthatjuk meg. Itt ismertetésre kerül az osztály főbb metódusainak a futása.

A LoadContent metódus futása:

- Létrehoz egy spritebatch objektumot síkbeli teljes-képernyős rajzolásokhoz.
- Beolvassa a felhasznált shader effektusokat
- Lekérjük a backbuffer (ahova az eredeti rajzolás történik) méreteit és egyéb tulajdonságait, hogy később megfelelő méretű textúrákat hozhassunk létre.
- Létrehozzuk a **scene** nevű **RenderTarget2D** típusú textúrát a backbuffer tulajdonságainak megfelelőt, melybe majd az eredeti, még nem utókezelt kép fog renderelődni.
- Létrehozzuk a **render0** és a **render1** nevű ideiglenes textúrákat, melyek takarékossági okokból nem rendelkeznek mélységi (Z) pufferrel és melyeknek a méretét a fő játék objektumunk **settings** mezőjében lévő **glowBadness** értékkel az eredeti backbuffer méretet elosztva határozhatunk meg.

A PrepareDraw metódus futása:

A **PrepareDraw** metódus szolgál arra, hogy a kirajzolás célpontját a saját rendertargetünkre irányítsuk át. Ezt a metódust így természetesen akkor kell meghívni, amikor a komponenst felhasználó program kirajzolásai megkezdődnek.

A Draw metódus futása:

Ezt a metódust akkor kell megrajzolni, amikor már minden érdemi kirajzolás megtörtént és már csak a menüpontok vagy belsőnézetes, mindig megjelenő objektumok kirajzolása van hátra, ugyanis elveszítjük a Z-Buffer értékeit a végleges kirajzolás során. A metódus a futása során:

- Beállítja a GlowBonus paramétert és kirajzolja a scene textúrát teljes képernyőben a render0 textúrába a szelektáló shaderrel, ezzel az első fázist megvalósítva.
- Kiszámolja a vízszintes gaussian blur filter shaderparamétereit, majd a render0 textúra és a gaussian blur shader felhasználásával a render1 textúrába rajzolja a szelektált kép, vízszintesen történő elmosottját, ezzel a második fázist megvalósítva. A shader paraméterek kiszámítását, a **SetBlurEffectParameter** függvény valósítja meg.
- Kiszámolja a függőleges gaussian blur filter paramétereket, majd a render1 textúrát a render0 textúrába rajzolja a gaussian blur shaderrel, ezzel a harmadik fázist is megvalósítva. A shader paraméterek kiszámítását, a

SetBlurEffectParameter függvény valósítja meg.

- Végül a render0 és a scene textúrák felhasználásával a backbufferbe, azaz lényegében a képernyőre rajzoljuk a kész képet a **FinalGlowEffect** shaderrel.

A draw során felhasznált **DrawFullscreenQuad** függvény**,** a képernyőre rajzol a bal felső sarokból kiindulva egy **width** és **height** mérettel rendelkező négyzetet az adott textúrával és shaderrel, a **DrawFullscreenQuadToTarget** függvény pedig egy adott renderelési céltextúrát kap és arra rajzolja ki teljes-képernyős négyzetként textúrát.

A Gauss függvény:

Haranggörbe függvény. A függvény visszatérési értékét a következő képlettel számítjuk ki:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

Ahol a theta megfelel a **setting.BlurAmount**-nak. Mivel a haranggörbe magasságát a szorzat első tagja adja meg, az e kitevőjében a tört alatt lévő theta pedig görbe szélességét befolyásolja, a **BlurAmount** beállításával ezt a két tulajdonságát változtatjuk a súlyoknak.

A SetBlurEffectParameters függvény:

Ez a függvény állítja elő a súlyokat a gauss függvény felhasználásával, illetve a mintavételezési pontok relatív koordinátáit a bemeneti dx és dy paraméterek alapján.

A koordinátákat, szimmetrikusan, ide-oda lépkedve számítjuk ki, kihasználva a görbe szimmetrikusságát. A lépések nagysága 2.0, de ehhez 1.5-et hozzáadunk, hogy az így kapott textúrakoordináta majd két pixel közé essen, ami azért jó nekünk, mert nem egész koordinátával megadott pixelt szeretnénk olvasni a shaderből, akkor a sampler a lineárisan interpolált értéket adja vissza számunkra, kiátlagolva ezzel az ottani két pixelt „ingyenesen”. Ezzel a megoldással tulajdonképpen kétszer akkora területet fedhetünk le, mint ha mi magunk átlagoljuk az egész-koordinátás pixeleket.

A relatív koordináták, illetve a hozzárendelt súlyok kiszámolása után, az eredményeket átküldjük a shadernek a konstans paramétereinek a beállításával.

A komponens használata következőképp zajlik, ha nem adjuk hozzá az osztályt a fő játékobjektum komponenseihez:

- **GlowComponent** objektum létrehozása és a **LoadContent** metódus hívása
- Minden rajzolás előtt a **PrepareDraw** metódus meghívása
- A rajzolások után (kivéve a menü és a belső nézetes objektumok kirajzolását) a komponens Draw metódusának meghívása.

A komponens használata következőképp zajlik, ha hozzáadjuk az osztályt a fő játékobjektum komponenseihez:

- **GlowComponent** objektum létrehozása és a **LoadContent** metódus hívása
- Minden rajzolás előtt a **PrepareDraw** metódus meghívása

- Meg kell győződni róla, hogy a **base.Draw()** hívás során meghívódó Draw komponensmetódus a kirajzolások után hívódik meg(beleértve egyéb játék komponenseket is), a főprogram komponenssorrendiségének **DrawOrder-**rel történő megadásával.

Mi nem adjuk hozzá a főosztályhoz a komponenst, hanem manuálisan végezzük el a hívást, mert az átláthatóbb.

3.5.5 Hardveresen gyorsított részecskerendszer létrehozása robbanások megjelenítéséhez

3.5.5.1 A megoldandó feladat

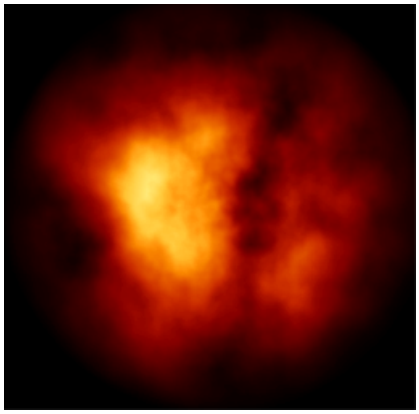
A játékban szükségünk van robbanások megjelenítésére egy ellenséges hajó megsemmisülésekor. A robbanási effektusokat játékokban az adott ponton megjelenő kétdimenziós robbanási sprite textúrával, vagy az ennél minőségibb látványt nyújtó részecskerendszer technikával szokás megoldani. Mi az utóbbi mellett döntünk.

3.5.5.2 A részecskerendszer technika elméleti alapjai

A részecskerendszer technika lényege, hogy a robbanást (vagy más hatást) úgy állítjuk elő, hogy sok, egymástól független háromdimenziós részecske mozgását szimuláljuk a térben a hatás helyszínén.

Tényleges térbeli pontokkal viszont nem érdemes dolgoznunk, mert úgy nagyon sok részecskét kellene használni, már egy viszonylag egyszerű robbanáshoz is, hiszen (amennyiben minden részecskéből egy pixel lesz) párszázas nagyságrendű részecskeszám mellett is azt érzékelnénk, hogy „lukacsos” a robbanás. Ezt a problémát kiküszöbölendő, a részecskerendszerek ún. billboarding technikát alkalmaznak és nem pontokat, hanem textúrázott síklapokat (általában négyzeteket) jelenítenek meg, méghozzá úgy, hogy azok folyamatosan a kamera felé nézzenek.

A programban is felhasznált egyik részecsketextúra a következő:



Ilyen textúrával rendelkező részecskék jelennek tehát meg százas nagyságrendben, különböző pontokban a robbanás helyszínén és mozognak különböző irányokban.

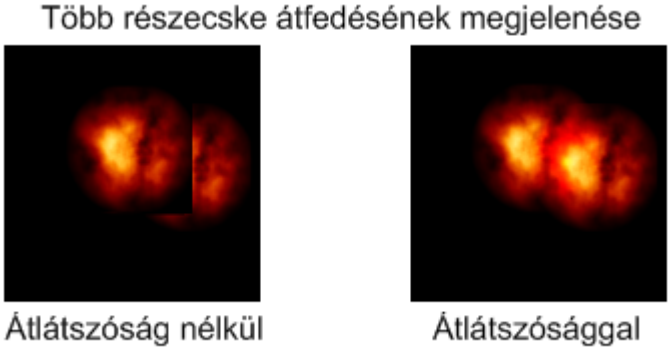
Ez a spherical billboarding technika egyébként még a régi időkből maradt fent, amikor a háromdimenziós kirajzolás nagy költséggel járt és az ellenfeleket, illetve egyéb mozgó objektumokat úgy rajzolták ki, hogy kiszámolták a középpontjának térbeli koordinátáit,

annak távolságát a kamerától, aztán ezek ismeretében síkban rajzoltak ki egy objektumot a megfelelő léptékelés mellett. Ez a megoldás azonban ebben a formában ma már nem túl járható út, a kettő és háromdimenziós kirajzolás keverésének nagy ráfordítási költsége miatt, ezért nekünk jobban megéri térbeli négyszöget, a megfelelő irányba fordítani, mint síkban rajzolni az itt vázolt módszerrel.

Elforgatás alatt viszont nem tényleges, mátrix-szal történő transzformációt kell érteni, hanem azt, hogy a részecske középpontja, a kamera pozíciója és a kamera felfele mutató vektora alapján vektoriális szorzattal számolt, a kamera síkjával párhuzamos tengelyek mentén történő eltolással kialakítjuk a tényleges négyszöget.

Ez a megoldás így már életképes, viszont felveti az átlátszóság problémáját, hiszen két részecske átfedésekor átlátszóság alkalmazása nélkül a kamerához éppen közelebb eső részecske textúrájának a fekete része egy kis négyzet alakú részt metszene ki a korábban ott lévő részecske képéből, ami nagyon csúnya hatást váltana ki.

Az átlátszóság használatának fontossága a következő ábrán jól látszik:



A fenti második képen látható átlátszóságot azonban nagyon könnyű elérni, elég csupán észrevennünk azt a tényt, hogy a részecske bezavaró háttér-komponense fekete, ami rgb-vel kifejezve csupa nullásból álló színértékeknek felel meg. Ennek a tudatában az a megoldási ötletünk támadhat, hogy a részecskék pixeleinek színértékeit additív módon adjuk hozzá a korábban már a képernyőn található pixelek színértékeihez. Ezzel a megoldással a fekete pixelek alatt megmarad a korábbi részecske színe(vagyis általában véve a korábbi ott lévő szín) és ezzel egyben még azt is elértük, hogy a robbanás részecskéinek egymást átfedő részén a színek erősebbek, világosabbak legyenek.

3.5.5.3 A hardveres gyorsítás szükségességének az okai

Az elméleti részben vázolt módszert akár úgy is implementálhatnánk, hogy létrehozunk egy tömböt a részecskéknek megfelelő struktúrából, majd a kamera irányában elforgatott négyzeteket kirajzoltatjuk a videokártya segítségével. Ezzel a naiv megoldással kapcsolatban azonban több probléma is felmerül.

Először is, ebben a formában az adatok nem lennének a gyors vertex pufferben tárolhatóak, mert a vertexek koordinátáit egyesével a CPU-nak kellene folyton újraszámolnia és a pufferbe helyeznie. Ez ebben a formában még nem is olyan nagy probléma, hiszen vannak külön ehhez hasonló a célokra dinamikus vertex pufferek, melyeket kicsit gyorsabban lehet írni, de ebben a formában a részecskéket a központi feldolgozó egységen történő számításokkal mozgatnánk el, ami azt jelentené, hogy a

teljes tömbön végig kéne mennünk és minden egyes vertexre ki kéne számolnunk az elmozdulást egy olyan hardverrel, ami nem háromdimenziós számításokra van optimalizálva, hanem általános célokra, miközben a videokártya pihenne és várná az eredményt.

Az lenne a legjobb számunkra, ha el tudnánk érni, hogy a részecskék frissítését teljes egészében a videokártyára végezze el, minimális CPU terhelés mellett. Szerencsére ha nem akarunk túl bonyolult számításokat elvégezni részecskénként (pl. fluid dynamics, vagy hasonló), akkor a mai architektúrák jó megoldást adnak nekünk a részecskekoordináták videokártyán történő kiszámításához.

3.5.5.4 A hardveres gyorsítás megvalósításának az áttekintése

A hardveres gyorsítási módszerünk alapja az architektúra azon sajátosságán alapul, ahogyan a vertex shader program kétféle inputja megérkezik egy draw hívás során:

- Megérkeznek a draw hívás típusától függően a host számítógépről a videókártyára küldött, vagy a videókártyán található vertex pufferből kiolvasott vertex-értékek, melyek egy korábban megadott vertex deklarációnak felelnek meg és az adott vertexre éppen futó vertex shader, adott vertexre vonatkozó adatait tartalmazzák.
- Megérkeznek a shader konstansok, melyek a képkocka kirajzolása, azaz a renderelés során minden vertexre(pixel shader esetén pixelre) azonosak maradnak.

Észre is vehetjük a lehetőséget, hogy mivel a vertex deklarációkat mi adhatjuk meg, (bizonyos korlátok között persze) a részecskenégyszögünk csúcspontjainak a reprezentációját az igényeink szerint megváltoztathatjuk. Ez azért fontos számunkra, mert így megtehetjük azt, hogy egy robbanás létrehozásakor feltöltünk egy vertex puffert egy saját vertex deklarációval úgy, hogy minden vertexhez megadjuk:

- A robbanás középpontjának koordinátáit a **POSITION0** csatornában.
- A kétdimenziós textúrakoordináta mellett a vertex születési idejét és a maximum korát a négydimenziós **TEXCOORD0** csatornában.
- A vertexhez tartozó részecske háromdimenziós elmozdulásvektorát és egy, a részecskék egyediségének biztosításához felhasznált véletlenszerűsítő tényezőt, a szintén négy dimenziós **TEXCOORD1** csatornában.

Illetve a fentieken kívül, minden draw hívás előtt beállításra kerülnek a következő konstans shaderparaméterek is:

- **world**, **view** és **projection** mátrixok.
- Kamera pozíciója és felfele mutató vektora (billboardinghoz szükséges).
- Egy **force** nevű háromdimenziós vektor, ami az adott irányban fellépő konstans erőhatást hozza létre a részecskékhez (gravitáció, esetleg mozgó hajó húzó hatása a robbanásra).
- A jelenlegi időpont, melyből kiszámíthatjuk a pozíciót.
- A robbanás hatására beírásra kerülő alfa csatorna értéke.

Megj.: Ezt nem fogjuk kihasználni, mert az additív blendinget kevés gépen lehet úgy beállítani, hogy külön módszerrel adja össze a pixelek színértékeit és az alfa csatornájukat, ha viszont simán összeadjuk, akkor az alfa érték 1.0-ra nő.

- Egy léptékelési faktor a részecskék méretének a növelésére.

Ezen információk birtokában pedig már a videokártyán tudjuk kiszámítani a részecskék pozícióját.

3.5.5.5 A megoldás részei és a kapcsolatuk

A részecskerendszer jelenlegi formájában 4 részből áll:

- **ExplosionVertex** osztály, a speciális vertexdeklarációnk leírására
- **Explosion** osztály, egy adott robbanás inicilizációjára és kirajzolására
- **ParticleSystem** osztály, mely robbanások tömbjeit kezeli, robbanás objektumok igény szerinti létrehozásával és megjelenítésével, illetve elméletileg kiegészíthető más, részecskerendszert igénylő effektusok kezelésére is.
- **ExplosionEffect.fx** shader effect fájl, mely a videokártyán futó pixel és vertex shader programokat tartalmazza.

Az **ExplosionVertex** osztály lényegében egy egyszerű **struct**, ami egy háromdimenziós és két négydimenziós vektort tartalmaz, illetve képes visszaadni az elemeinek megfelelő vertex deklarációt, de ezen kívül semmi egyebet nem tud, a **ParticleSystem** osztály pedig csupán létrehoz egy **Explosion** elemeket tartalmazó tömb-puffert, melyet cirkuláris pufferként használva tölt fel új robbanással kérésre, melyeket lineáris tömbeléréssel rajzoltat ki úgy, hogy minden létező elemre meghívja a **Draw()** metódust. Ezen felül a **ParticleSystem** osztály betölt (jelenleg) kétféle textúrát is a robbanások részecskéihez és ezeket váltogatja úgy, hogy minden második robbanás másféle részecsketextúrákkal jelenjen meg a változatosság kedvéért.

A **ParticleSystem** osztály tehát jelenleg viszonylag egyszerű, fő célja egy köztes réteg létrehozása egy játék komponens alakjában, ami esetleg később több, hasonló jellegű részecskeeffektus létrehozásával bővülhet, például hajtóművek anyagkiáramlását szimuláló másik részecskeeffektussal.

A rendszer két másik része kicsit bonyolultabb, azokat külön alfejezetekben ismertetjük.

3.5.5.6 A videokártyán futó programrész: ExplosionEffect.fx

Az effekt fájl tartalmazza a hardveres gyorsítás áttekintésénél látott bemeneti deklarációkat, illetve shaderkonstansokat, egy **BillBoardVertex** nevű segédfüggvényt az adott vertex helyes koordinátára hozásához, illetve a Vertex és Pixel shader főprogramokat.

A BillBoardVertex segédfüggvény futása:

- Megkapja bemenetként a vertex középpontját, azaz a **billboardCenter**-t, a vertexhez tartozó textúrakoordinátákat, melyekkel a vertex azonosítható (**cornerID**) és a részecskének a nagyítási faktorát (**size**).
- Kiszámolja a kamerától a részecske középpontja felé mutató **eyeVector**-t.

- Az **eyeVector** és a kamera felfele vektorának keresztszorzatának a normáltjával meghatározza a billboard oldalirányú **sideVector**-át.
- A sideVector és az eyeVector segítségével előállítjuk keresztszorzattal és normálással a billboard felfele vektorát (**upVector**)
- A textúrakoordináták segítségével, melyek -1.0 és 1.0 közötti értékek (bal felső saroktól a jobb alsó felé nőnek a koordináták), illetve az előbb kiszámolt vektorokkal és a size értékkel kiszámítjuk a vertex tényleges pozícióját. Amit visszadunk.

A vertex shader program futása:

- Kiszámoljuk a **world** mátrix-al transzformált részecskepozíciót, fejlesztési lehetőséget biztosítva arra, hogy a robbanást például egy modell saját koordináta-rendszerében definiáljuk, mely objektum világ mátrixát például itt megadva a robbanás könnyen követi az adott objektumot annak mozgása esetén.

  Megj.: A pozíció vektort emiatt kiterjesztjük a negyedik komponenssel, hogy az eltolások is működjenek a világ mátrix transzformációjából ne csak a forgatás.
- Dekódoljuk a bemenő vertex csatorna paramétereit külön változókba helyezvén az egyes, számunkra fontos elemeket.
- Kiszámoljuk a jelen vertexhez tartozó részecske aktuális, halálhoz relatív korát. Ezzel egy [0..1]-beli értéket kapunk, hogy hol tart az életútja a részecskének.
- Kiszámoljuk a részecske méretezésére és odébb helyezésére szolgáló együtthatókat a következő formulák alapján:

$$\text{size} = (1 - \frac{\text{relAge}^2}{2}) * \text{scale} * \text{random}$$

$$\text{displacement} = \sin(\frac{\text{relAge} * 6.28}{4}) * 3.0 * \text{random}$$

  Tehát a részecskék mérete idővel négyzetesen csökken, eltolása pedig a szinusz függvény első negyedének felszorzása szerint számítódik az idő függvényében.
- Kiszámítjuk a részecske tényleges, az adott időpillanathoz tartozó középpontját az eredeti koordinátájának és a **delta** elmozdulásvektor **displacement**-szeresének az összege alapján.
- Az előző pontban kiszámolt középpontot manipuláljuk a force vektor segítségével a részecskekor függvényében.
- A fent leírt segédfüggvény felhasználásával és a projekciós, illetve nézeti mátrix-al történő szorzással meghatározzuk a jelenlegi vertexpozíciót.
- A pixel shader egyik bemenetéül szolgáló **a** értékét kiszámoljuk a következőképp:

$$\mathbf{a} = 1 - \text{relAge}^2$$

  Mert ezzel fogjuk majd a pixel shaderben sötétíteni a részecske színértékeit.

- A textúrakoordinátákat pedig változtatás nélkül adjuk tovább

A pixel shader program futása:

- Kiszedjük a textúrából az alapszínt.
- Megszorozzuk **a**-val a komponenseit.
- A végleges szín így kapott 3 komponensét a konstans shaderparaméter **alpha** értékével kiegészítve alfa csatornával is ellátjuk.

3.5.5.7 Az Explosion osztály

Az **Explosion** osztály reprezentálja magát a robbanást a játékban. Létrehozásakor meg kell neki adni a szülő részecskerendszert, illetve a használt textúrát és shadert, inicializációjakor (**InitExplosion**) pedig meg kell adni:

- A jelenlegi időpillanatot reprezentáló **GameTime** típusú objektumot
- A robbanás pozícióját
- A robbanás világtranszformációs mátrixát
- A robbanás méretét
- A robbanás sebességét
- A robbanáshoz felhasznált részecskék számát
- A robbanás időbeli hosszát milliszekundumban
- A részecskékhez beírandó alfa csatornát
- A force húzóerőt vektor formájában

Az inicializáció ezek után létrehozza a robbanásvertexeket leíró vertex puffert és feltölti azt a megfelelő paraméterértékekkel, illetve a majdani **random** változóba kerülő véletlen számokkal és a feltöltés során felhasználunk még három véletlen számot, melyeket a mozgásvektorok megszorítására használunk fel a tengelyek irányában így érve el még nagyobb fokú változatosságot a robbanásaink között. Az inicializáció ezen kívül eltárolja az utolsó részecske teljes eltűnésének a várható idejét is.

A robbanás megjelenítését a **draw** metódus végzi el, melynek szintén meg kell kapnia az aktuális játékidőt, melyet majd a shadernek átad konstans paraméterként. A draw metódus a futása során:

- Lekérdezi a játékidő objektumból az aktuális milliszekundumot.
- Összehasonlítja az utolsó részecske meghalásának idejével és ha nagyobb, akkor kilépünk és az aktivitást false-ra állítjuk.
- Beállítjuk a shaderparamétereket
- Beállítjuk a vertex bufferünket
- Aktivizáljuk a shaderünk nullás menetét (vagyis magát a shadert mert az egymenetes)
- Végül kirajzoljuk a vertex buffer tartalmát

Megj.: Az **active** property azért került megvalósításra, hogy esetleges fejlesztés gyanánt később a **ParticleSystem** osztály valami okosabb, kevésbé triviális módszerrel tárolja a robbanásokat, mindig a legrégebbit cserélve ki, ha túl sok robbanás van.

3.5.5.8 A rendszer használatához szükséges lépések a rendszert befoglaló osztály számára

A komponens használatához néhány módosításra van szükségünk a felhasználó osztályban. A megoldandó feladatok a következőek:

- Az átlátszósági megoldás kiválasztása (de az additív blending a támogatott)
- Z-Buffer működésének meghatározása, hogy a részecskék melyek a kamerához egy objektumnál közelebb vannak az objektumra rajzolódjanak, melyeket egy objektum eltakar ne legyenek kirajzolva, viszont az egyes részecskék lehetőleg ne takarják el egymást!
- A robbanások és egyéb más objektumok közötti kirajzolási sorrend meghatározása, csak a két halmaz között, azaz lényegében a **ParticleSystem draw()** metódushívásának programszövegbeli helyének meghatározása a befoglaló osztályban.

Ezeket a problémákat sokféleképpen meg lehet oldani. Mi a programunkban a következő megoldásrendszert választottuk a fentiekre válaszolva:

- Az átlátszóságot teljes additív blending-el valósítjuk meg, azaz a részecskék pixeleit és a már korábban ott lévő pixeleket összegezzük. Ehhez a megoldáshoz sajnos nem használhatjuk fel a beépített **BlendState.Additive** beállítást, mert az paradox módon felhasználja az alfa csatornát is az XNA framework 4.0-s verziószámától kezdve bevezetett premultiplied alpha alapértelmezett használata miatt. Mivel nincs is nekünk megfelelő beépített BlendState, nekünk magunknak kell az objektumot felparamétereznünk a következőképpen:

```
realAdditiveBlend = new BlendState();
realAdditiveBlend.ColorBlendFunction = BlendFunction.Add;
realAdditiveBlend.AlphaBlendFunction = BlendFunction.Add;
realAdditiveBlend.ColorDestinationBlend = Blend.One;
realAdditiveBlend.AlphaDestinationBlend = Blend.One;
realAdditiveBlend.ColorSourceBlend = Blend.One;
realAdditiveBlend.AlphaSourceBlend = Blend.One;
realAdditiveBlend.ColorWriteChannels = ColorWriteChannels.Red
                                     | ColorWriteChannels.Green
                                     | ColorWriteChannels.Blue;
```

  Ez pont azt jelenti, hogy a pixelek színértékei összegezve lesznek (A **Blend.One** értékek mint szorzótényezők állnak az összegzendő régi és az új pixelértékek mellett), illetve az utolsó sor alapján az alfa csatorna értékét meg sem változtatjuk. Ez utóbbi azért kell, mert enélkül ha egy robbanás mögött halad el egy lövedék, akkor a lövedék „elveszíti” egy időre a glow tulajdonságát. Sajnálatos módon ez **Xbox-on** mint **hiba** jelentkezik állítólag, XBox-on viszont általában külön lehet állítani az alfa csatornához és a színkomponensekhez tartozó paramétereket, amik nálunk a glow problémáját okozzák csakis egyben történő beállíthatóságukkal. **Így a probléma megoldható Xboxon is.**

- A Z-Buffer működési szemantikája a programunkban az lesz, hogy a skyboxtól

eltekintve minden objektum írja és olvassa is a Z-Buffert, a részecskerendszer pedig csak olvassa(így a részecske megjelenésekor nem frissül a mélységi puffer és a később rajzolt távoli részecske nem fog eltűnni.

- Az előző ponthoz alkalmazkodva a helyesség érdekében a részecskék rajzolódnak ki utoljára, hogy a Z-Bufferbe írás hiánya miatt semmi probléma ne keletkezzen.

3.5.6 Az űrben található objektumok reprezentációja és osztályhierarchiája

3.5.6.1 A megoldandó feladat

Szükségünk van a játék űrobjektumainak átfogó jellegű, hierarchikus reprezentációjára a közös tulajdonságok kiemelésével a továbbfejleszthetőséget is szem előtt tartva.

Megjegyzendő, hogy az ellenfelek mesterséges intelligenciája nem tartozik bele ebbe a feladatkörbe, hiszen azt szeretnénk különválasztani az űrbeli objektumok osztályaitól, annak dinamikusan változtathatóságát mint továbbfejlesztési lehetőséget támogatva. Itt nem célunk leírni a mesterséges intelligenciához tartozó osztályaink szerepét.

Az űrobjektumaink ütközéseinek kezelése miatt külön kiegészítéseket teszünk majd az itt látható hierarchiában található osztályokhoz, illetve külön projekteket is készítünk. Ezek azonban itt nem kerülnek ismertetésre, mert az ott található többször fejlettebb megvalósítások miatt azok kitöltenek egy külön fejezetet.

3.5.6.2 Egy térbeli, forgatható objektum orientációjának és helyzetének a tárolásának a lehetséges megközelítései

Az akciójátékunk megvalósításához elengedhetetlen, az űrhajóink és egyéb az űrben található objektumaink helyes reprezentációja azoknak frissítésének, mozgatásának a céljából. Egy térbeli tetszőleges tengelyek körül forgatható objektumot, annak koordinátáival egyetemben azonban többféleképpen is reprezentálhatunk, melyek közül igyekszünk a lehető legmegfelelőbbet kiválasztani a megvalósítandó feladat ismeretében.

Az orientáció és a pozíció tárolására általában a következő megoldások egyikét szokás alkalmazni:

- Egy 4x4-es transzformációs mátrix.
- Az objektum saját koordinátatengelyeinek és a térbeli pozíciónak az eltárolása.
- Az objektum 3 alaptengely körül történő elforgatási mértékének(yaw, pitch, roll) és a koordinátáinak a tárolása
- Az objektum forgatását reprezentáló kvaternió és az térbeli pozíció eltárolása
- Egyéb vagy ötvözött, speciális megközelítések.

Az itt felsoroltak közül mindegyik rendelkezik előnyökkel és hátrányokkal. Például az alaptengelyek körüli forgatás nagyon jól működik olyan környezetben, ahol nem szükséges minden tengely körül forgatni, csak például az x és az y tengely körül

(ilyenek például a belső nézetes, lövöldözős katonai játékok, vagy a modellező szoftverek), de mivel a harmadik tengely használatba vételével szabadsági fokot veszítünk és beállhat az ún. **gimbal lock** nevű állapot (szélsőséges forgatások esetén előforduló állapot, melyben az egyik tengely körüli forgatás hatása minimálisra csökken), ez számunkra nem használható módszer. A koordinátatengelyekkel történő reprezentációnak maximum a lövedékek megjelenítéséhez lenne értelme, mert azok kellően egyszerű objektumok, de akkor elvesztenénk az objektumaink egyeséges reprezentálásának a lehetőségét. A meggondolandó megoldások halmaza tehát lényegében a mátrix-alapú, illetve a kvaterniós megoldásokra csökken.

Korábbi tapasztalatokból kiindulva azonban elmondható, hogy egy ilyen típusú játék esetén, bár csábítónak tűnhet a mátrixos megközelítés, mivel nagyon bonyolult és lassú számításokat kell elvégezni egy adott frissítési vagy mozgatási fázisban, ezt az ötletet is elvethetjük. Így tehát a kvaternióval és a koordinátáinkkal történő reprezentáció mellett döntünk, melyet implementációs megfontolások miatt végül kicsit vegyítünk a mátrix-os megvalósítással.

3.5.6.3 A forgatási transzformációk és a kvaterniók kapcsolatának az elméleti gyors áttekintése

A dolgozatnak nem célja a kvaterniók algebrai szintű ismertetése (sőt, annak konkrét ismerete az implementációnkhoz sem szükséges) de azok és a térbeli forgatások kapcsolatát valamilyen szinten át kell tekintenünk, bizonyos, a kvaterniókra jellemző alaptulajdonságok ismertetése mellett.

A kvaterniók egy négydimenziós vektorteret alkotnak a valós számok felett, összeadás, skalárral való szorzás és kvaterniószorzás műveletekkel. Melyek közül az első kettő a hagyományos négydimenziós vektorokkal megszokott műveletet, az utóbbi pedig egy, az 1, i, j, k-val jelölt báziskvaterniók esetén az (i * i = j * j = k * k = i * j * k = -1) báziselemekre vonatkozó tulajdonságból levezethető, következőképpen számolandó szorzás:

$$\begin{aligned}
&a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 \\
&+ (a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)i \\
&+ (a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)j \\
&+ (a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)k.
\end{aligned}$$

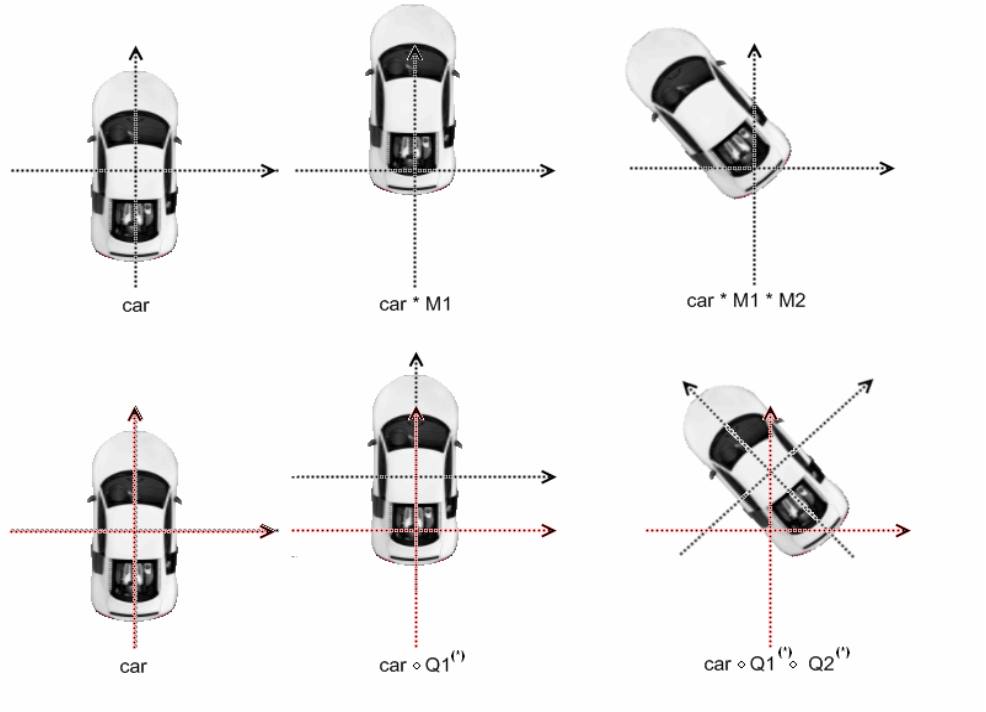
Ahol az a,b,c,d betűk páronként az eredeti báziskoordinátáknak felelnek meg, indexük alapján a baloldali(1), illetve jobboldali(2) szorzótényezőkben. Érdemes már itt megjegyezni, hogy a kvaterniók egymással történő szorzása nem kommutatív. Ezen felül a kvaterniókra értelmezzük a konjugálás műveletet is, melynek során a valós (az 1 bázishoz tartozó) komponenstől eltekintve negáljuk a koordinátákat.

A kvaterniók geometriai célú felhasználhatósága abból származik, hogy egy háromdimenziós vektornak egy **egységkvaternióval** történő konjugálása (a vektort, mint 0 valós komponensű kvaterniót tekintve, majd balról a kvaternióval, jobbról annak konjugáljával szorozva) egy olyan, az eredeti vektor elforgatottjaként előálló vektort kapunk, mely az eredeti vektorunk, egy **u** vektor körüli, az óramutató iránya szerinti, **alfa** fokkal történő elforgatásának az eredménye, amennyiben konjugáló q kvaternióra:

$$q = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} = w + (x, y, z) = \cos(\alpha/2) + \vec{u}\sin(\alpha/2)$$

Számunkra ez röviden összefoglalva csupán annyit jelent, hogy egy kvaterniót tulajdonképpen reprezentálhatunk egy négydimenziós vektorral, mely lényegében egy tengelyt reprezentál(**u** vektor) és egy elforgatási szöget(**alfa** szög). Ezen kívül a fent leírt kvaterniószorzást sem szükséges magunknak implementálnunk, mert az XNA környezet **Quaternion** osztálya tud mindent, amire csak szükségünk lehet. Itt csupán azért szerepelnek ezek az elméleti alapok, hogy a dolgozat segítséget nyújtson kvaternió osztályok megvalósításához, illetve, hogy az olvasónak legyen egy elképzelése a kvaterniókkal végzett műveletek (nem magas) költségét illetően.

Számunkra sokkal lényegesebb viszont az a gyakorlati kérdés, hogy mit is jelent, mint transzformációt tekintve, egy több kvaternió szorzataként előálló kvaternió és egy pozícióvektor felhasználásával transzformálni egy vektort. Ezt egy egyszerűsített felülnézetben látható objektumon, csak az adott síkhoz tartozó szorzat-transzformációkat tartalmazó képsorozaton mutatjuk be, összehasonlítva a mátrixokkal történő transzformációk hasonló képsorozatával:



A fenti képsorozat egy autónak először egy M1 eltolási mátrix-szal, majd egy M2 forgatási mátrix-szal történő transzformálását láthatjuk. Ilyen esetben a transzformációra „ökölszabály-jelleggel” úgy is tekinthetünk, hogy balról jobbra olvasva a transzformációkat, azokat folyamatosan a helyben maradó eredeti koordináta-rendszerhez képest hajtjuk végre, így kialakítva a végleges transzformációt, míg – amint azt az alsó képsorozaton láthatjuk, kvaterniókat használva az egyes transzformációkat mindig a már transzformált tengelyeket alapul véve végezzük el.

(<u>'): Fontos megjegyzés!</u> Az ábrán megjelenített példa „hamis”, mivel a kvaterniók csupán a forgatásokat reprezentálják és az eltolásokat ehhez még külön, az elhelyezkedési vektorokkal kell kiszámítani. A „kör” jelzésű művelet tehát nem csak a kvaternióval történő transzformálást jelenti, hanem „kvaternió felhasználásával létrehozott általános eltolási és forgatási transzformációpáros realizálását”.

A fent említett „kör” transzformációs műveletet a következő transzformációs (és egyéb) lehetőségek megvalósításával támogatjuk majd:

- Lehetőséget adunk az objektum 3 jelenlegi alaptengelye körül történő elforgatásra. Ez jelenti a reprezentációs kvaternió szorzását a koordinátatengelyek körül forgató 3 kvaternióval.
- Lehetőséget adunk abszolút koordináták szerinti eltolásra. Ez a reprezentációnkban tárol háromdimenziós pozícióvektor frissítését jelenti.
- Lehetőséget adunk az objektum saját koordináta-rendszerében történő eltolására, mindhárom tengelye mentén. Ezt a kvaternióból előállított forgatási transzformáció, forgatási mátrix-szá történő alakításával, illetve az eredeti bázisvektorok ezzel történő szorzásával, majd az így kapott vektorok mentén a koordinátáknak megfelelő eltolással valósítjuk meg.
- Mivel gyakran használjuk, megvalósítjuk az előző függvény egy egyszerűbb változatát, mely csak az előre-irányuló vektor mentén képes eltolni.
- Ezen felül lehetőséget nyújtunk a kvaternió, a pozíció és a segítségükkel előálló irányvektorok és mátrix lekérdezésére.

Megjegyzés: A kvaternió forgatási mátrix-szá alakítása és úgy történő alkalmazása szinte nem, vagy egyáltalán nem lassabb, mintha a kvaternióval konjugálnánk a vektort, viszont kézenfekvő a használata, hiszen mindenképpen lehetővé kell tennünk a forgatási mátrix lekérdezését, illetve mert az átalakítást az XNA elvégzi helyettünk egy beépített tagfüggvénnyel, a **Matrix.CreateFromQuaternion()** függvénnyel.

3.5.6.4 Az űrbéli objektumok reprezentációját megvalósító alaposztály, a SpaceObject, illetve az űrobjektumaink osztályhierarchiájának az áttekintése: A BattleField névtér osztályai

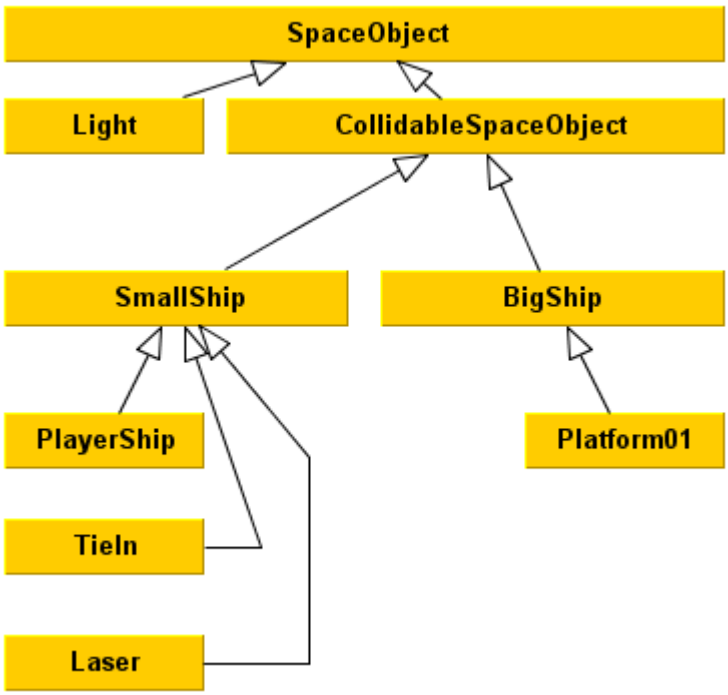
Az űrben létező objektumaink közös ősosztályaként tehát létrehozzuk a **SpaceObject** nevű osztályunkat, ami rendelkezik, az előző alfejezet végén sorba vett transzformációs és lekérdezési lehetőségek megvalósításával.

Ebből származtatjuk a játékbeli jelenetünk egyetlen fényforrását, illetve az ütközéseket is kezelni képes, illetve megjeleníthető **CollidableSpaceObject** osztályunkat, melyből két fő hajóosztályt, a **BigShip** és **SmallShip** osztályainkat fogjuk örökíteni. Az egyes hajótípusok ezek alatt külön osztályt kapnak majd az említett két fő ősosztályból származtatva, és általában olyan, speciális elemeket fognak tartalmazni, mint egyéni ütközési jellemzők, illetve maguk a háromdimenziós modellek.

A hajók méret alapján történő osztályozása mellett azért döntöttünk, mert a programunk során majd – mint azt látni fogjuk, ütközési szempontból különféleképpen kezeljük a kis és a nagy hajókat. Érdekes megemlíteni, hogy a kamera objektumunk, vagyis maga a játékos képzeletbeli hajója a **PlayerShip** is egy, a **SmallShip** osztályba tartozó űrhajó lesz, mely ki van egészítve a nézeti mátrix megkonstruálásához szükséges forráskóddal, annak lekérdezését biztosítva, illetve felül van benne definiálva a tengelyek körüli forgatást megvalósító függvény, melyben azért, hogy a többi objektummal való kompatibilitást megőrizzük, negatív forgatási szögekkel forgatunk, így biztosítva egyszerű megoldást a nézeti mátrix egyszerű előállítására (negatív pozícióval történő eltolási mátrix * kvaternióból nyerhető forgatási mátrix) a könnyű kezelhetőség és az

említett többi objektummal történő kompatibilitás megőrzése mellett.

A **SpaceObject** osztályból származó, a jelenlegi megvalósításban implementált osztályhierarchia a következő ábrán látható:



Megjegyzés: A lövedékeket reprezentáló **Laser** osztályunk **SmallShip**-ből történő öröklődésének az oka továbbfejlesztési és azon belül ütközésvizsgálati jellegű lehetőség támogatása miatt van megvalósítva. Ennek a pontosabb magyarázata a következő, ütközéssel foglalkozó fejezet végében található. Megjegyzendő továbbá, hogy a fában a nem absztrakt, levél-osztályok a **Light, PlayerShip, TieIn, Laser, Platform01** osztályok.

A fent vázolt osztályhierarchia elég általános és így könnyen bővíthető újabb hajókkal, de az ütközési megvalósításokon és az objektumok megjelenését, illetve elhelyezkedését leíró információkon és az azokon dolgozó metódusokon kívül egyebet nem tartalmaz. A konvenciónk tehát az, hogy minden egyéb megvalósítandó, az űrobjektumokkal kapcsolatos művelet az itt látható osztályhierarchiától szeparáltan kerüljön megvalósításra, az itt létrehozott osztályok befoglalásának segítségével. A legfontosabb ilyen befoglaló osztályunk a **BattleFieldComponent** nevű osztály, mely a programunk összes űrobjektumát egybefogja, ezzel lényegében a csatamező állapotterét megvalósítva. Az említett osztály ezen felül egy játékkomponens is, mely rendelkezik a megfelelő **Draw** és **Update** hívásokkal.

3.5.6.5 A BattleFieldComponent osztály

A **BattleFieldComponent** osztályunk magvát az általa menedzselt három tömb, a **bigShips**, a **smallShips**, illetve a **lasers** tömbök alkotják, melyek a megfelelő típusú űrobjektumokat tárolják, illetve a fényforrást tároló **light** és a játékos virtuális űrhajóját reprezentáló **playerCamera** változók. A komponens ezen kívül tartalmaz egy

referenciát a fő játékobjektumunkra, a játék során használt projekciós mátrixra, illetve egy, már inicializált részecskerendszerre robbanások létrehozásához.

A csatamező-komponensünk ezen elemeinek a felhasználásával képes a **Draw** metódusának végrehajtásával megjeleníteni a csatamező összes objektumát. Ehhez a tartalmazott űrobjektumok saját kirajzoló függvényét használja, mely nagy hajók esetén a buckaleképezéses technika felparaméterezésével és a modell megjelenítésével, kis hajóknál pedig a **TieIn** osztály esetén környezetleképezéssel történő felparaméterezés melletti modellkirajzolást, **Laser** osztály esetén a beépített **BasicEffect** shaderrel történő kirajzolást, **Playership** esetén pedig üres függvényhívást eredményez.

A csatamező-komponens másik fontos függvénye az **Update** pedig elvégzi a tömbökben tárolt objektumok ütköztetését, egy egyszerű algoritmussal (ez is majd a következő fejezetben kerül bővebb ismertetésre, az egyes ütközési modellek ismertetése mellett).

A komponens a fent említetteken kívül tagfüggvényeket biztosít új objektumok hozzáadásának a megkísérlésére, melyek csak akkor történnek tényleges elhelyezésre, ha a konstruktorként kapott maximum hajószámoknak megfelelő méretű tömbökben felszabadul egy hely, az illető objektum **life** propertyjének negatívra csökkenésével. Ilyenkor egy egyszerű next fit algoritmus fut le, amely a jelenlegi indexpozíciótól számítva, a tömb hosszáig keres cirkuláris módon (modulo aritmetikával) egy üres helyet és helyezi el az objektumot amennyiben egy ilyet talál.

3.5.7 Az űrobjektumok közötti ütközések megvalósítása

3.5.7.1 A megoldandó feladat

A programnak megoldást kell nyújtania a nagyszámú (lövedékekkel együtt 70-100 körüli darabszámmal rendelkező), és a nagy és kis hajók miatt nagyságrendben is eltérő méretű, illetve eltérő sebességű háromdimenziós modellek közötti ütközések ellenőrzésére gyors, illetve skálázható megoldások alkalmazásával.

3.5.7.2 Észrevételek és egy alapvető ötlet: a három ütközési család

A játékban három, ütközési szempontból különféle jellemzőkkel bíró űrobjektum-család található:

- Vékony, csíkszerű és gyors lövedékek
- Kicsi és közepesen gyors vadászok
- Nagy és lassú (a megvalósított példajáték során nem is mozgó) hajók, állomások

A fentieket elolvasva az az ötletünk támadhat, hogy a felvázolt három kategóriába tartozó objektumok ütköztetését talán érdemes lenne különböző módszerekkel megvalósítani, egy általános, ám túlzottan pazarló megoldás implementációja helyett, ügyelve az egyes ütközési családokba tartozó egyedek közötti, családok között, keresztben történő ütközések megvalósítására is. Pontosan ezt fogjuk tenni, méghozzá a közösen felhasználható megoldások és tulajdonságok, az ősosztályokba történő kitranszformálása mellett.

3.5.7.3 CollidableSpaceObject, avagy az ütközési családok közös elemeit összefogó osztály

A legalapvetőbb közös jellemzőket, a **SpaceObject**-ből származtatott, **CollidableSpaceObject** nevű osztályban helyezhetjük el, amiből a kis és nagy hajók egyaránt származnak így egy ideális helyen található közös ősosztályt képezve, ahol a lövedék kishajónak számít a mérete miatt (majd a fejezet végén megtudjuk, hogy milyen megfontolásból).

Úgy döntöttünk, hogy az ütközéssel kapcsolatban, a következő dolgok fontosak, mind a három ütközési család számára:

- Objektumot befoglaló **kocka** lekérdezésének lehetősége
- Objektumot befoglaló **kocka léptékezett** lekérdezésének a lehetősége
- Objektum előző koordinátája és jelenlegi koordinátája által meghatározott sugár, lekérdezhetősége, méghozzá úgy, hogy a sugár reprezentációjában az utolsó elmozulás irányított szakaszát is tároljuk (az XNA beépített **Ray** osztálya szerencsére pont ilyen)
- Objektum életpontjainak a száma (játékmenetbeli megfontolásból)
- Az objektum által ütközéskor okozott életpontcsökkenés száma (szintén a játékmenet érdekében)
- Az objektum életerejének a csökkentése megadott számmal (programozás későbbi kényelmessé tétele miatt)
- Implementálandó **collosionTest** abstract függvény, mely megadja hogy történt-e ütközés a bemenetként kapott objektummal, vagy sem.
- Implemetálandó **proximityTest** függvény, ami igazat ad vissza, ha az objektum az adott másik objektum közelében van (továbbfejlesztési lehetőségek miatt került be)
- Adott irányított szakasz, ütközési modellbetöltővel betöltött modellel történő ütközését vizsgáló **raysegmentModelCollision** függvény. Ez főként a nagy hajók ütközéséhez használható, de a továbbfejleszthetőséget szem előtt tartva ide illik.

Mivel nem szeretnénk implementálni a **collisionTest** függvényt minden lehetséges objektumcsaládból minden lehetséges objektumcsalád felé nézve, azt a megoldást választjuk, hogy az egyes hajók között végzett ütközési teszt nem lesz kommutatív, azaz **a.collisionTest(b) != b.collisionTest(a)**. Az egyes ütközéseket megvalósító függvények programozásakor pedig azt az ökölszabályt alkalmazzuk, hogy a speciálisabb igényű ütközési objektum collisionTest-je kerül meghívásra egy ugyanolyan, vagy alacsonyabb specialitásigényűvel szemben.

Az igények specialitás szerinti sorrendje a következőképpen alakul(felül a legigényesebb):

1. Nagy hajók
2. Lövedékek
3. Kis hajók

3.5.7.4 Kishajó ütközése kishajóval

A kishajók közötti ütközések megvalósítására egyszerű, a tengelyekkel párhuzamos befoglaló kockát használunk, melyek metszése esetén az objektumokat ütközőknek tekintjük.

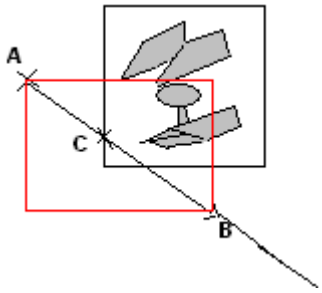
Ebben a formában a kishajó-kishajó ütközés pontatlannak tűnhet, ám nagyon gyors és a célnak megfelel. Amennyiben pontosabb ütköztetési igény merül fel, az később implementálásra kerülhet a megfelelő ütköztető függvény pontosabb megvalósításával.

3.5.7.5 Lézerek ütközése kishajókkal

Mivel a lézerek gyorsan haladnak, illetve mert vékonyak, a fent leírt módszer nem megfelelő az ütköztetésükre a nagy várható hibaarány miatt. A lézerek ütköztetésének alapötlete az, hogy felhasználjuk a lézer elmozdulásának irányított szakaszát tartalmazó sugár-objektumot és ezt az irányított szakaszt metsszük a kishajó befoglaló kockájával.

Ezt úgy ellenőrizzük le, hogy az irányított szakaszköré írt befoglaló doboz (ez már nem kocka) és a kishajó befoglaló kockájának a vizsgáljuk van-e metszete (a lenti ábrán piros és fekete dobozok tesztje), majd amennyiben van, akkor először optimista módon le ellenőrizzük, hogy az irányított-vektor sugár jelenlegi végpontja a kishajó befoglaló kockájában van-e, mert ha benne van akkor azt ütközésnek vesszük, amennyiben pedig nincs benne, úgy vesszük a **Ray** objektumunk **intersects(BoundingBox)** metódusa által visszaadott metszési távolságot (ha nincs metszet, null-t kapunk és kilépünk false-al), amiről megvizsgáljuk, hogy közelebb van-e az irányított szakaszunk kezdőpontjához, mint annak a hossza.

A módszert röviden a következő ábra szemlélteti, ahol az elmozdulás irányított szakaszának kezdőpontja A, végpontja B a lövedék esetén, C pedig az ütközési pont, melynek távolságát, a szakasz hosszával vetjük össze ha kell:



Ez a módszer megfelelő megoldást nyújt a kishajókkal történő lövedéktköztetésre, hiszen nem túl észrevehető pontatlanság mellett a nagy sebesség ellenére sem tudja a lövedék „átugorni” egy képkocka alatt a kishajó objektumot.

3.5.7.6 Nagy hajók ütközésének vizsgálata egyéb objektumokkal, áttekintés

A nagy űrobjektumaink ütköztetésekor a fentieknél sokkal bonyolultabb és egyben pontosabb megoldásra van szükségünk, ugyanis a méretkülönbségek növekedésével a fenti egyszerű módszerek hibája is rendkívüli mértékben megnő, így szükségessé válik a nagy űrhajó háromdimenziós modelljének, mint háromszögekből álló térbeli testnek a precíziós ütköztetése az egyéb objektumokkal.

Két megoldandó feladat is keletkezik a precíz, háromszögenkénti ütköztetéssel, az egyik a megvalósításra egy módszer kidolgozása, a másik a megvalósítás optimalizálása. Utóbbi azért fontos, mert a program során felhasznált modelljeink sok ezer háromszögből állnak, melyeknek a közel százas nagyságrendű egyéb objektumokkal történő „nyers-erő” jellegű ütköztetése hihetetlen mértékben lassítaná le a program futását.

Az első feladatra a válaszunk egy, a lézerlövedékek ütköztetéséhez hasonló, irányított szakasz alapú ütköztetés lesz, a második feladatra pedig egy fordítási időben a modellhez hozzárendelt, annak saját koordináta-rendszerében értelmezett térparticionáló fa lesz. A nagyobb hajók ütköztetésének a megoldását így, két kisebb feladatra bontással az elkövetkező alfejezetekben írjuk le.

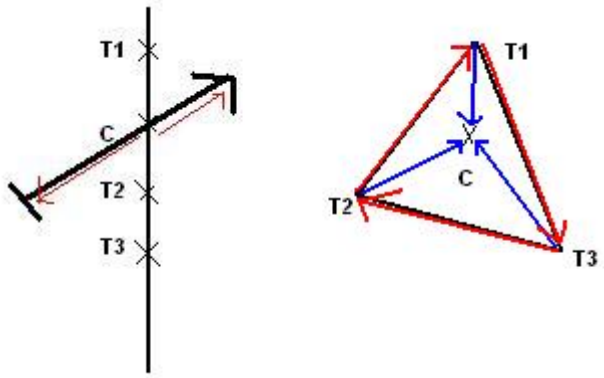
3.5.7.7 A nagy hajók ütköztetéséhez használt irányított szakasz, térbeli háromszöggel való metszetén alapú ütközési modell

Az alapötletünk, hogy a hatalmas nagyságrendbeli különbségek miatt, a kis hajók és a lövedékek is tekinthetőek nagyon vékony objektumnak a nagy hajóval történő ütközés esetére, így elég megvalósítanunk térbeli háromszögeknek, irányított szakaszokkal történő metszésének vizsgálatát, majd a nagy hajó összes háromszögét megvizsgálni a kisebb objektum sugár objektummal reprezentált, irányított szakaszának az ütközésével.

Az ütközés vizsgálata igen egyszerűen, a következő lépésekkel zajlik le:

1) Előállítjuk a háromszög által meghatározott síkot (van beépített **Plane** osztály az XNA-ban)
2) A **Ray.Intersect(Plane)** függvény segítségével meghatározzuk a sugár és a sík metszéspontjának távolságát a sugár kezdőpontjától, illetve ha null-t kapunk vissza, azaz nincs metszéspont, akkor visszatérünk hamis értékkel, jelezve, hogy nincs ütközés.
3) A sugár irányának a felhasználásával meghatározzuk a tényleges ütközési pontot.
4) Vektoriális szorzat, illetve az újonnan keletkezett ütközési pont segítségével eldöntjük, hogy a sík két felén van-e az irányított szakaszunk két pontja. (ez kell az ütközéshez minden esetben)
5) Ha igen, akkor megnézzük, hogy a háromszög síkjára eső síkkal történő ütközési pont a háromszögön belül van-e. Ezt úgy határozzuk meg, hogy a háromszög oldalaiból vektorokat képezünk, miközben a kiindulási pontból az ütközési pont felé is vektorokat képezünk és ezek vektoriális szorzatáról leellenőrizzük skaláris szorzás segítségével, hogy egy irányba mutató vektorokat eredményeztek-e. Ugyanis mivel a vektoriális szorzat eredménye az operandusaival mindig jobbkezes koordináta-rendszert eredményez, az így kapott szorzatvektoroknak mindig, minden körülmények között egy irányba kell mutatniuk, amennyiben az ütközési pont a háromszög belsejében van. Ha a pont belül van és idáig eljutottunk, akkor ütközés történt!

A módszer két fő lépése a következő ábrán látható, a háromszög oldalnézetében, illetve szemből.:



Ahol a bal oldal a síkkal való metszéspont meghatározásának a fázisát jelzi, míg jobbra a síkon, a háromszög belsejében lévőség ellenőrzését. Ebben a fázisban a kék vektorokat kell vektoriálisan szorozni, az ugyanabból a kezdőpontból induló piros vektorokkal, majd a szorzatok egy irányba mutatását skaláris szorzattal(előjelvizsgálat) ellenőrizni.

3.5.7.8 A nagy hajók ütköztetéséhez használt térparticionálási rendszer áttekintése

A fenti módszer ebben a formában nem alkalmas a gyakorlati megvalósításra, hiszen túl nagy műveletigény szükséges minden egyes háromszög ilyen módon történő vizsgálatához. Az így keletkező problémát mi úgy oldottuk fel, hogy a nagy hajókhoz hozzárendeltünk az alap háromdimenziós modelljükhöz egy ütközési modellt is, melynek **Tag** tulajdonságában eltároltunk egy térparticionálási fát, ami a modell háromszögeit tartalmazza. Ezt az információt (mivel se a futást, se a betöltést nem akarjuk lassítani), egy **content pipeline kiegészítés**ként megvalósított ütközési beolvasóval oldjuk meg, még fordítási időben.

3.5.7.9 Az octree adatstruktúra

Számunkra éppen ideális az **Octree**-nevű, minden szintjén nyolc (vagy levél esetén nulla) irányba elágazó fa adatstruktúra, ami egy skálázható megoldást nyújt a modellünkben szétszórt háromszögek könnyen kezelhető particionálására.

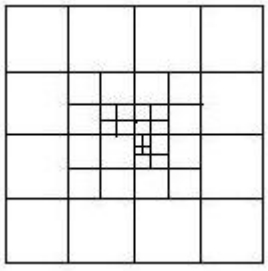
Az octree, működési elve az, hogy a térbeli objektumok(nálunk modell háromszögei) köré írható befoglaló dobozt, szétosztja 8 azonos méretű albobozra(bal-felső-első, bal-felső-hátsó, jobb-felső-hátsó, …, jobb-alsó-első) és azokban szétosztja a fődoboz objektumait (nálunk háromszögeit).

Az octree adatstruktúra számára adott a maximális famagasság, illetve az egy dobozban található maximális elemszám, amin túl fel kell bontani az adott csúcsot, ha ez a famagassági kritériummal nem ellenkezik.

Az octree tehát lényegében egy, a leveleit kivéve minden pontján nyolc felé elágazó fa, melytől azt várjuk, hogy (kellően szétszórt input esetén) nyolcadára csökkenti famagasságonként a várható dobozban kerülő háromszögek számát. Ezt általában a

gyakorlatban használt modellek esetén el is várhatjuk tőle.

Ezen felül az octree rendelkezik egy nagyon szép adaptív tulajdonsággal is, ugyanis a fa felosztását vizualizálva, azt tapasztalhatjuk, hogy a particionálás a sűrű részeken pont sűrűbb, a ritka részeken pedig ritkább lesz, ahogy az a következő (felülnézeti) ábrán szemléltethető:



Így néz ki egy octree felülnézete olyan objektum esetén, melynek a közepén magas csak a részletessége.

Megjegyzés: Az octree térbeli adatstruktúra, nem síkbeli (ami az ábra miatt az embernek eszébe juthat), az ábra csupán az octree, egy felülnézetből vett projektált képét mutatja. Az octree síkbeli megfelelője egyébként a hasonló elven működő quadtree.

A mi octree megvalósításunk egy rekurzív adatstruktúra, azaz minden egyes csúcs a fában egy fa maga is. Egy fa létrehozásakor pedig meg kell adnunk a dobozának a méretét (a legelső fához ezt előre kiszámoljuk a háromszögek befoglaló dobozának a meghatározásával), a potencionális benne lévő háromszögeket(első szintnél az összeset átadjuk) a maximális szintet és a jelenlegi szintet, ami a kezdőelemnél nulla kell legyen.

Ezután a létrejött fa, kiválogatja és megszámolja a ténylegesen őt metsző háromszögeket(*), majd amennyiben a túl sok háromszöget talált létrehozza a 8 gyerek fát és mindet inicializálja, a már általa kiválogatott háromszögekkel.

A (*)-al jelölt művelet számunkra egy kicsit trükkös, hiszen azt kell eldöntenünk, hogy egy adott háromszög, mint térbeli síkidom, metszi-e az adott, a koordináta-tengelyekkel párhuzamosan (a modell koordináta-rendszerében dolgozunk végig) elhelyezkedő téglatestet. Ez nem is olyan könnyű probléma, hiszen nem elég csupán befoglaló dobozokat, vagy pontok tartalmazását megvizsgálnunk, sőt, még a háromszög éleinek a kocka síkjaival vett metszete se elég a teljes pontossághoz. A programunk során ennek a problémának a megoldására a SAT nevű tételt (Separable Axis Theory) használjuk ki, melynek lényege, hogy minden térbeli, akármilyen kicsi térfogattal rendelkező két konvex test elválasztható egy valamilyen tengelyre történő egydimenziós projekciójában a ponthalmazok metszetét tekintve, akkor csak akkor, ha a két konvex objektum nem metszi egymást

3.5.7.10 A háromszögek halmazából történő kiválogatás módszere és a SAT tétel

A SAT tétel tehát kimondja, hogy létezik az előző alfejezet végén említett tengely és a tétel konstrukciót is ad egy ilyen elválasztó egyenes meghatározásához (a két nem metsző objektum két legközelebbi pontja által meghatározott vektor által meghatározott

tengely pont jó lesz). A tételt nem bizonyítjuk, de alkalmazni fogjuk, méghozzá úgy, hogy megvizsgáljuk a háromszög, illetve a doboz összes érintősíkjára merőleges tengelyek mentén a szétválaszthatóságot, illetve a háromszög és a doboz minden élpárjára egyszerre merőleges tengelyek mentén (ezekre vetítve pontszerűvé alakulnak az adott élek).

Ez tehát összesen 13 darab tengelyt jelent, ismerve a doboz koordinátatengelyekkel való párhuzamosságát:

- 3 projekciós teszt a koordinátatengelyek mentén (doboz oldallapjainak az érintősíkjai)
- 1 teszt a háromszög normálvektora mentén (háromszög oldallapjának az érintősíkja)
- 9 teszt: a háromszög mindhárom élének a 3 tengellyel való vektoriális szorzásával kapott, mindkettőre merőleges vektor mellett.

Megjegyzés: Azzal az esettel, amikor a háromszög és a doboz adott éle párhuzamos, nem foglalkozunk tovább, mert ez az eset elfajult hibákat eredményezne a számításokban (bele is sikerült futni ebbe hibába a rendszer első implementációja során). Ezzel a megoldással, hogy ezen élpárok között a teszteket nem hajtjuk végre információt viszont nem sokat veszítünk és vegyük észre, hogy nekünk nem célunk a tökéletes ütközés ellenőrzés, csupán a háromszögek „minél jobb” szétszórása.

3.5.7.11 A nagy hajók ütközési rendszerének a megvalósításának az összefoglalása és a megvalósításhoz használt fájlok és osztályok rövid leírása. TypeWriter és TypeReader osztályok szükségessége

A nagy hajók ütközési rendszere tehát irányított szakaszok és háromszögek ütköztetésén alapul, melyhez fordítási időben egy content pipeline kiegészítésként megvalósítottunk egy ütközési modellek beolvasását végző modellbetöltőt, mely a modelljeinket ellátja egy **ütközési Octree**-vel, mely adatstruktúrának a **COctree** nevet adtuk, ami egy rekurzív adatstruktúraként van megvalósítva, melynek egyes csúcsai vagy egyéb **COctree** fákat, vagy háromszögek tömbjeit tartalmazzák. Ez a saját modellbetöltő, a **CollisionPipeline** projekt **CollisionModelProcessor**, míg a runtime is felhasználható osztályokat a **CollosionPipelineRuntimeHelper** névtér alatt került megvalósításra és részletezésre itt már nem kerül, hiszen nem sok újdonságot tartalmaz a korábban már más content pipeline kiterjesztéseknél megemlítettekhez képest. Az egyedüli különlegesség a **TypeWriter** osztály definiálása a **Triangle** és a **COctree** objektumok számára. A **Triangle** nevű osztállyal pedig egyszerű háromszögeket reprezentálunk.

A TypeWriter és a TypeReader:

Ezek az osztályok számítanak újdonságnak a Content pipeline kiegészítésünkben. A writer szerepe az, hogy az importer és a processor által átalakított kimenetben lévő osztályokat sorosítsa, míg a megfelelő reader osztály szerepe a sorosított objektum beolvasása. Szerencsére nincs szükségünk az objektumokban nekünk magunknak fájlokat kezelni, azt megoldja a keretrendszer, mi csupán azt vagyunk kénytelenek megadni, hogy egy nem ismert osztályt, melynek adatai a kimeneti tartalomba kerülnek, milyen, már beépített osztályok segítségével írhatunk le. A háromszögünket mi térbeli koordinátákat jelképező vektorokká (**Vector3**), míg az C**Octree** fát egyéb fákká és

háromszögek (illetve néhány egyéb érték, pl az integer. Famagasság) sorosítjuk.

Megjegyzés: A tényleges megvalósításban technikai okokból nem modellek, hanem modelmesh-ek(részmodellek) befoglaló hiearchiáit hozzuk létre.

3.5.7.12 Az így létrehozott ütközési modell felhasználása

Az ütközési modellünket úgy használjuk fel, hogy a **CollidableSpaceObject** osztályunkban megvalósítjuk egy, várhatóan a **Tag** tulajdonságában egy fent definiált **COctree** adatstruktúrát tartalmazó ütközési modellt egy adott irányított szakassz történő ütköztetését, a fánk **collisionTest** metódusának a segítségével, méghozzá az objektum inverz világnézeti mátrix-szával előzőleg modell-térbe transzformálva az irányított szakaszt.

Az előző bekezdésben vázolt megoldás segítségével pedig már képesek vagyunk, a három ütközési családba tartozó objektumok közötti ütközések, páronként történő megvalósítására, a korábban említett „ökölszabályunk" alapján. A **BattleFieldComponent** osztályunk **Update** metódusában lévő ciklusok segítségével.

3.5.7.13 Egy lehetséges továbbfejlesztési irányvonal

A rendszer már ebben a formájában is sok hajót kezel, de van egy gyenge pontja, mégpedig a **BattleFieldComponent** osztályban található négyzetes jellegű megközelítés, melynek során egy adott hajót, az összes többivel is megpróbálunk ütköztetni.

Ezt kikerülendő, érdemes lenne egy dinamikusan változó felosztású octree megvalósítására, mely tartalmazná az ütköztethető űrobjektumainkat, elérve ezzel, hogy csak az egy dobozba kerülő hajók között vizsgáljuk meg az ütközést, további nagyságrendekkel bővítve a játékban felhasználható hajók számát.

Egy ilyen megvalósítás során előnyhöz juthatunk, ha a nagyobb és a kisebb hajókat megkülönböztetjük(mert a nagy hajó sok dobozon átnyúlhat a kicsi pedig nem feltétlenül), így ennek a fejlesztési lehetőségnek a tudatában helyeztük a **Laser** osztályt a kishajók osztályhierarchiájába.

3.5.8 Egy egyszerű, de kiterjeszthető mesterséges intelligencia megvalósítása

3.5.8.1 A megoldandó feladat

A játék ténylegesen játszhatóvá tételéhez szükség van egy minimális szintű mesterséges intelligencia kialakítására, melyet úgy célszerű megalkotni, hogy könnyen továbbfejleszthető legyen minél többféle megközelítést támogatva.

3.5.8.2 Mennyit segít nekünk ebben az eddig meglévő architektúránk?

Abban a szerencsés helyzetben vagyunk, hogy a jelenlegi architektúránk szépen különválasztja az irányítani szánt hajókat reprezentáló osztályokat és rajtuk műveleteket végző osztályokat. Ezen kívül adott egy, a játék csataképernyőjének a mesterséges intelligencia számára hasznos állapotait jól reprezentáló állapottér osztálya, a

BattleFieldComponent személyében.

3.5.8.3 Mi lesz a kiterjeszthető mesterséges intelligencia megvalósításunk alap keretarchitektúrája?

A mesterséges intelligencia megvalósításunk 3 fő komponensből fog összeállni:

- **Strategist** (Stratéga) osztályokba tartozó egyedek, melyek látják a **BattleFieldComponent** osztályunkat, így elvileg képesek átfogó döntések meghozatalára.
- **CollidableSpaceObjectAI** osztályból származó egyedek, melyek esetleg több, általuk irányított alegység-szerűen működő **CollidableSpaceObjectAI**-t tartalmazhatnak egyfajta virtuális katonai hierarchiát jelentő fastruktúrát megvalósítva, miközben saját ütköztethető objektumuk frissítéséért is felelősek.
- **AICommand** általános parancs osztályból származó parancsok, melyek a fában üzenetként terjednek.

Mivel azonban egy ilyen mesterséges intelligencia precíz megvalósítása (különösen a harctéri elemzéseket elvégző stratéga komponens megvalósítása) egy külön dolgozattal is felérne, mi csupán a lehetőségét és a keretét adjuk meg egy ilyen bonyolult hierarchia megvalósításának, az alaposztályok létrehozásával, miközben implementálunk egy egyszerű, egyetlen szintből álló fát, melyet a **BruteForceStrategist** osztályba tartozó stratéga irányít, aki nem csinál mást, mint hogy parancsok kiadása nélkül rábízza a döntést az egyes **BruteForceStarFighterAI, CollidableSpaceObjectAI**-ből származó hajókhoz tartozó mesterséges intelligenciákra, melyek csupán egy egyszerű állapotgépet valósítanak meg a következő állapotokkal:

- Megközelítik az ellenfelet

  `CollidableSpaceObjectAIState.ApproachingTarget`
- Lőnek az ellenfélre

  CollidableSpaceObjectAIState.`AttackingTarget`
- Kitérnek az ellenfél elől

  `CollidableSpaceObjectAIState.EvadingTarget`

3.5.8.4 A BruteForceStrategist osztály és a BattleFieldComponent osztály kapcsolata

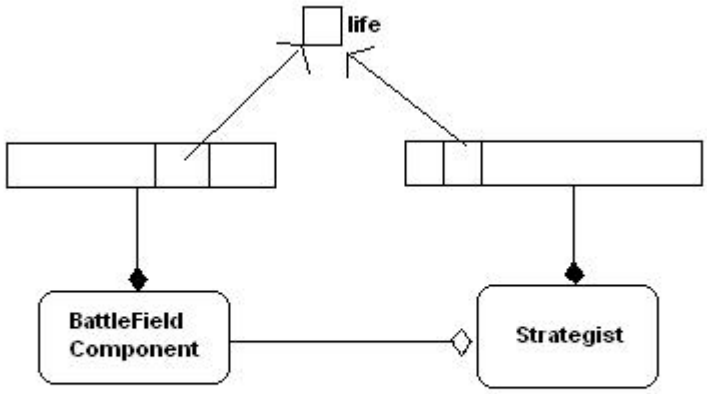
A játékmenetünk során majd azt szeretnénk elérni, hogy bizonyos időközönként létrejöjjenek vadászok, melyek rögtön kapnak egy mesterséges intelligenciát is, de amennyiben a pályán már található a maximálisnak megfelelő vadász, akkor ne jöjjön létre, csak annyi amennyi még elfér.

Ezt jelenleg könnyen megvalósíthatjuk a kész csatamező-osztályunkkal, hiszen annak a hajólétrehozó függvényei pont így működnek. A célunk tehát az, hogy a stratéga osztályunk megfelelő létrehozó függvényei, működésben feleljenek meg a korábban látott megoldásnak, illetve kapcsolódjanak ahhoz, szinkronizálva a hajók pusztulását és születését.

Kapcsolódási pont gyanánt logikus választás az életerő pont figyelése, hiszen az mind a

mesterséges intelligencia osztályaiban történő hivatkozások számára, mint a játéktérosztályunk számára akkor csökken 0 alá, ha az adott objektum elpusztul(ilyenkor robbanást is kitesz a helyére a **BattleFieldComponent**).

A mesterséges intelligencia jelenleg megvalósított egyszintes architektúrája a következő ábrán tekinthető át, annak kapcsolataival egyetemben:



Ahol a **BattleFieldComponent** által a **life** közvetlenül, annak **SmallShips** tömbjében lévő elem publikus tulajdonságaként érhető el, míg a **Strategist** számára az általa számon tartott **CollidableSpaceObjectAI**-k egy absztrakt publikus, getter-setter metódussal megvalósítandó **life** tulajdonságán át, mely jelen esetünk az adott **BruteForceStarFighterAI** által irányított **goverdedObject** változójának a megfelelő tulajdonsága, mely ugyanazt az értéket tartalmazza, amennyiben a csatamezőn frissített hajó azonos az adott mesterséges intelligencia által irányított hajóval.

Ezen felül a két rendszer egységes feltöltéséhez egy **Spawner** nevű osztályt is létrehoztunk, mely megpróbálkozik az adott vadászt, mindkét alrendszerben történő létrehozásával, illetve amely (egyszerűségi megfontolásokból) rendelkezik egy egyszerre négy lövedéket, az adott ütköztethető űrobjektum pozíciójának és orientációjának megfelelő létrehozását biztosító függvénnyel is.

3.5.8.5 A BruteForceStarFighterAI osztály működési alapelve

Az alap működési elv a következő:

- Amennyiben Támadó, Megközelítő, vagy Kitérő állapotban van, igyekszik a **targetedPosition** térbeli koordinátái felé.
- Kezdetben megközelítő állapotban van.
- Ha a **targetedPosition**-tól vett távolság **400** alá csökken és megközelítési állapot van, akkor átmegy támadó állapotba.
- Támadó állapot esetén másodpercenként lead egy négy lövedékből álló lövést a mesterséges intelligencia által irányított űrobjektum aktuális irányában.

- Ha a **targetedPosition**-tól vett távolság 60 alá csökken, akkor átmegy:
  - Kitérő állapotba, amennyiben támadó, vagy megközelítő álapotban volt
  - Megközelítő állapotba, ha kitérő állapotban volt
- Kitérő állapotba kerüléskor inicializálódik a **targetedPosition** egy véletlenszerű célpont irányába.
- Egyébként a **targetedPosition** folyamatosan a **target** tulajdonság által célzott hajó elé van egy kicsivel beállítva, annak sebességétől függően.

Megjegyzés: A jelen megvalósításban a **target** objektum minden mesterséges intelligencia által irányított hajóra, a játékos hajója.

A **targetPosition** irányába történő elfordulások meghatározásához a hajó mesterséges intelligenciája:

- Kiszámítja a jelenlegi pozíciójából a cél felé mutató vektort
- Normálja azt.
- Majd egy trükkös projekciót végez, melynek során egy, a saját koordináta-rendszerének fel és jobbra vektora által meghatározott térbe egy kör pontjaira projektálja az előzőleg kapott normált vektort úgy, hogy ha a z koordináta szerint előtte van az objektum, akkor hagyományos projekciót végez amit a végén felére zsugorít (így 0.5 sugarú kört kapunk a lehetséges pontokból amik előttünk vannak), amennyiben pedig mögöttünk volt a pont, akkor úgy alakítja a koordinátákat 1-ből, illetve -1-ből történő kivonással, hogy azok nagyobbak legyenek ha a z tengellyel bezárt szög kicsi(ezt esetszétválasztással oldjuk meg), illetve az így kapott koordinátákat projektáljuk és 0.5-el növeljük, egységkört előállítva. Ezzel lényegében egy képzeletbeli projektált félgömböt hoztunk létre, melyben a pont elhelyezkedése azt jelzi, hogy az ellenséges célpont megtalálásához merre mekkorát kell elfordulnunk.
- Az előzőleg előállított síkbeli koordinátapár alapján meghatározzuk az elfordulásokat, melyeknek felfele-lefele történő mennyiségét az előbb kiszámolt koordináta y komponense, a jobbra-balra, az irányvektorunk körül történő forgatásának mennyiségét pedig a fenti síkban lévő felfele mutató vektornak és a kapott x, y koordinátapár által meghatározott vektornak a bezárt szöge határozza meg úgy, hogy a forgatás irányát az x komponens előjeléből számoljuk.
- Az így kapott két forgatási értéket -1 és 1 közé normáljuk, majd egy konstans értékkel szorozzuk(alapból túl lassú volt a fordulás) és elvégezzük a forgatást.

Röviden tehát előállítunk egy egységgömböt, melyet leprojektálunk trükkösen, ami azt jelenti, hogy az első félgömb egy 0.5 sugarú kör ponthalmazának felel meg, míg a hátulsó félgömb azt ezt a ponthalmazt egy 1 sugarú körré kiegészítő tórusz pontjainak felel meg, méghozzá úgy, hogy (mivel ha a célpont pont mögöttünk van, akkor a fordulás pont a legerősebb kell legyen) kifordítjuk előtte a hátsó félgömb pontjait, a leghátsó pont melletti pontokat az oldalsó körgyűrű pontjaival kicserélve és így tovább. Aztán az így kapott kör alapján előre-vektor körüli szöget, illetve emelkedés/süllyedési szöget számolunk.

A forgás és az állapotváltozások kiszámolásán túl, az irányított objektumot a

mesterséges intelligencia folyamatosan, az előre vektora mentén mozgatja annak a **goForward** függvényének a segítségével.

3.5.9 A játékmenet és a pontozás megvalósítása illetve a program készre fordítása

3.5.9.1 A megoldandó feladat

A tényleges játék megvalósulásától már csak egy hajszál választ el minket. Ez a hajszál pedig a játékmenethez kapcsolódó kiegészítő programkódok implementálása, egy szálkereszt a célzáshoz, egy pontozási szisztéma, és egy kilépés vagy elhalálozás esetén előkerülő játék vége képernyő megvalósításából áll.

3.5.9.2 Kiegészítések a játékmenettel kapcsolatosan

- Hozzáadtuk a képernyőhöz a szükséges részecskerendszer, csatamező, stratéga és játékos osztályokat, illetve inicializáltuk ezeket.
- A csatajelenet képernyőben lévő forráskódot módosítottuk úgy, hogy az egér relatív x, y elhelyezkedéséből kiindulva a játékos a mesterséges intelligenciához hasonló módon képes legyen felfele emelkedni, illetve süllyedni és forogni az előre mutató irányvektora körül.
- Bal egérgomb (akár folytonos) lenyomásához hozzárendeltünk egy másodpercenként történő lövedéklétrehozást.
- A korábbi főmenü-képernyővel analóg módon implementáltunk egy játék-vége képernyőt, ami a konstruktorában kap egy **int** számot, a **score**-t.
- A csatamező osztályt módosítottuk, hogy az ütközésvizsgálatok esetén, amikor ütközések vannak növeljen egy kezdetben nullára inicializált publikus **score** nevű **int** változót, melyet elhalálozáskor, vagy kilépéskor a játék-vége képernyő inicializálásához használunk. Így a játékos pontos kap minden egyes hajó megsemmisülésekor, ha él az adott pillanatban, amikor a hajó megsemmisül.

  Megjegyzés: Ezen felül a nagyobb hajók megsemmisülését több ponttal jutalmaztuk.
- A csataképernyő forrásában elhelyeztük a szálkereszt kirajzolását (egyszerű teljes képernyős **SpriteBatch** rajzolás) közvetlenül a játékbeli menüpontok megjelenítését megelőző helyen.
- Kirajzoltunk egy a játékos hajójának **life** tulajdonságával egyenesen arányos csíkot a bal felső sarokba.
- A csataképernyő inicializálásakor véletlenszerűen kihelyeztünk, maximum 10 darab, űrállomást úgy, hogy ezek nem érnek egymásba, mert minden létrehozás előtt ellenőrizzük, hogy az adott pozíción az adott befoglaló dobozzal ütközünk-e valamelyik korábbi objektummal és ha nem, csak akkor helyezzük ki az állomást.
- Beleírtuk az **Update** metódusba, hogy 30 másodpercenként hozzunk létre 20 új vadászt, amennyiben ez lehetséges.

3.5.9.3 A program készre fordítása

A programot a végleges állapotában lefordítottuk Debug, majd Release módban is, hogy szinkronban legyen a két bináris állapota. Ezt követően a fő projektünkre ráengedtük a Visual Studio ClickOnce telepítővarázslóját, melyet felparamétereztünk, hogy töltse le a megfelelő szoftver követelményeket a program futtatásához az internetről, így egy kompakt telepítési csomagot létrehozva.

3.5.10 Pár szó a tesztelésről

A program könnyen debugolható, tesztelhető részeit általában alávetettük a megfelelő tesztelésnek(egy menüpont, több menüpont, menüpontok kipróbálása, stb.), de a grafikai effektusokat megvalósító shader programokat jobb híján csak szemre teszteltük, hiszen abban a környezetben természetesen nem működik a debugger.

A grafikai effektusoknál az egyes megoldások implementációjának tesztjei a következők voltak:

- Environment mapping technika tesztelése kikapcsolt ütközések és álló hajók mellett, a hajó állandó forgatásával és annak körberepülésével, több szögből történő megvizsgálásával.
- A normal mapping technikát először nem tangens térben és mindenféle egyéb optimalizációktól teljesen mentesen implementáltuk, aztán az így készült képeket összehasonlítottuk a később megvalósításra került, átírt változattal
- A részecskerendszer tesztelésekor ügyeltünk az egyes objektumok egymáson történő fedésének mindenféle lehetőségét leteszteltni, így akadtunk például rá arra a hibára, amit az additív blending okozott, miszerint elvesztek a robbanás mögött lévő lövedékek glow effektusai a robbanás idejére (bár ez egy alig észrevehető hiba volt)
- A skybox technika teszteléséhez nagyon távolra is elrepültünk a jelenet objektumaitól nagyon kis méretű skybox beálítása mellett, ezzel bizonyosodva meg arról, hogy a Z-puffer írása helyesen van beállítva
- A laserglow effektust, annak különböző beállításai mellett teszteltük le optimális megjelenésre kihegyezve, miközben megbizonyosodtunk az egyes fázisok működéséről (a későbbi fázisokat üres shaderekkel helyettesítve)
- A játék összes grafikai megoldását teszteltük ezen kívül sebességre is a fejlesztés folyamán, amihez a fraps nevű program ingyenes változatát használtuk, a Frame Per Second (azaz, a másodpercenkénti képkockák) értékének a mérésére, minden effektus esetén annak a várható legrosszabb forgatókönyvét jelentő jelenetet véve alapul.

Az ütközési rendszer tesztelése pedig kis objektumok esetén a kész játék megfigyelésével zajlott, míg a **COctree** adatstrúktúrát a Visual C# belső debuggerével „kézzel” is leellenőriztük szúrópróba-szerűen. Így derült például fény a háromszögek éleinek és a dobozok éleinek a párhuzamosságából származó problémára, melyet ki is javítottunk.

A játék irányíthatóságának, pontozásának, telepítésének a tesztelésében a családom segítségét kértem, akik hajlandóak voltak erre fordítani az idejüket így a telepítés során

is találtunk egy problémát, ami abból származott, hogy a ClickOnce telepítővarázsló alapesetben nincs megfelelően paraméterezve egy XNA 4.0 projekt készre fordításához.

3.5.11 Pár szó az XBox-ra történő portoláshoz

A programot XBox-ra közvetlenül nem valósítottuk meg, de végig figyeltünk az architekturális sajátosságokra, illetve megszorításokra, az irányítási rendszer pedig fel van készítve a konzolhoz adott irányító eszköz osztályának az implementálásához és teszteléséhez. A portoláshoz ha minden igaz, így csupán az irányítási rendszer kiegészítésére van szükség és a részecskerendszer fejezetében említett blend állapot megfelelő átalakítására. Persze tesztelés hiányában előfordulhatnak nem várt problémák.

4. Összegzés

Az eddigieket összegezve eljutottunk tehát egy egyszerű, bár kiterjeszthető játék megvalósításáig, miközben implementálásra és bemutatásra kerültek az eredeti célkitűzésünkben szereplő játékfejlesztési megoldások.

A program előállítása során bemutattunk néhány manapság alapvetőnek számító modern grafikai effekturt és néhány saját megoldást is (ütközési rendszer, selective laserglow), miközben a fejlesztés során végig a bővíthetőséget és az átláthatóságot tartottuk szem előtt.

Külön örüm, hogy az elkészült játék ezen felül a tesztelések alapján viszonylag gyorsan fut a régebbi gépeken is, feltéve hogy azok rendelkeznek a szükséges **shader modell 2.0** támogatással.

Felhasznált irodalom

[1] Riemer Grootjans: XNA 3.0 Game Programming Recipes: A Problem-Solution Approach, Apress, 2009, 663, 978-1-4302-1855-5

[2] http://create.msdn.com/en-US/ , 2011.jan.10