



M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Balogh Tamás

3D MODELL MEGJELENÍTŐ SZOFTVERKOMPONENS KÉSZÍTÉSE ANDROID PLATFORMRA

KONZULENS

Fehér Marcell

BUDAPEST, 2012

Tartalomjegyzék

Összefoglaló	7
Abstract	8
1 Bevezető	9
1.1 A platform bemutatása	9
1.2 SDK és NDK	9
1.3 Fejlesztői eszközök	11
1.4 Kooperáció	12
1.5 A dolgozat vázlatos felépítése	13
2 A feladat újraértelmezése	14
2.1 Kapcsolódás hallgatótársam komponenseihez	14
2.2 A grafikus megjelenítő egység tervezése	15
2.3 Felhasználói felület	16
2.4 Szenzorkezelés	16
2.5 Kompatibilitás	16
2.6 Modellek	17
3 Irodalomkutatás	18
3.1 Min3D	18
3.2 A libGDX	18
3.3 Következtetések	19
4 Az alkalmazáskeret	20
4.1 Előzetes tervek	20
4.2 A project felépítése	20
4.3 A felület megvalósítása	21
4.4 Építő elemek	22
4.5 A főképernyő	23
4.6 A játékképernyő	24
4.7 Kapcsolódás a maghoz	25
4.8 Szenzorkezelés	25
5 A grafikus komponens	27
5.1 Bevezető, kapcsolat az alkalmazáskerettel	27
5.1.1 Az alkalmazásmag felépítése	27

5.1.2 Fordítási és linkelési beállítások	27
5.1.3 Esemény napló	28
5.1.4 Native.cpp.....	28
5.2 A grafikus könyvtár	31
5.2.1 A grafikus könyvtár felépítése.....	31
5.2.2 A motor	31
5.2.3 A motor funkciói.....	33
5.2.4 Hibaellenőrzés	35
5.3 Mátrixok és vektorok.....	37
5.3.1 Kiegészítő funkciók	37
5.3.2 Vektor funkciók	38
5.3.3 Alap mátrix funkciók	38
5.3.4 Transzformációs funkciók.....	38
5.3.5 Nézeti mátrixok	38
5.3.6 A mátrix verem	41
5.3.7 A mátrix állapotgép.....	42
5.4 Az árnyalás.....	43
5.4.1 Az OpenGL ES 1.1 csővezetéke.....	43
5.4.2 Az OpenGL ES 2.0 csővezetéke.....	44
5.4.3 Háromszög árnyaló (<i>Vertex Shader</i>).....	44
5.4.4 A képpont árnyaló (<i>Fragment Shader</i>)	45
5.4.5 Az árnyalás matematikai modellje.....	46
5.4.6 Interpolációs típusok	46
5.4.7 Fénytípusok	47
5.4.8 A fényvisszaverődések.....	47
5.5 Az árnyalók megvalósítása	49
5.5.1 A háromszög árnyaló megvalósítása	49
5.5.2 A képpont árnyaló megvalósítása	50
5.6 Textúrázás	53
5.6.1 Textúra térképezés	53
5.6.2 Textúra szűrés.....	54
5.6.3 Megvalósítás - Létrehozás.....	55
5.6.4 Megvalósítás - Megjelenítés.....	56
5.7 A grafikus objektum	58

5.7.1 Felépítése.....	58
5.7.2 A modellek betöltése.....	59
5.7.3 Elemek feltöltése a grafikus hardverre.....	60
5.7.4 <i>Uniform</i> változók feltöltése	60
5.7.5 <i>Attribute</i> változók feltöltése	60
5.7.6 A kirajzolás.....	61
5.8 Kamerák	62
5.8.1 A komponens felépítése	62
5.8.2 Kamera típusok.....	62
5.8.3 Kameramozgató funkciók	63
5.9 A felhasználói felület (HUD)	65
5.9.1 Felületi elemek.....	65
5.9.2 A HUD	66
5.10 Idő- és Érintéskezelés	67
5.10.1 Az időkezelő osztály	67
5.10.2 Az érintéskezelés	68
6 A játékalkalmazás	70
6.1 Entitások.....	70
6.1.1 Az <i>Item</i> osztály	70
6.1.2 A <i>Car</i> osztály.....	71
6.2 A játékalkalmazás főprogramja.....	73
6.2.1 Az inicializáció	73
6.2.2 Az érintéskezelés	75
6.2.3 Animáció	75
6.2.4 A kirajzolás.....	76
7 Értékelés és továbbfejlesztési lehetőségek	78
7.1 Tesztelés és értékelés	78
7.2 Továbbfejlesztési lehetőségek.....	81
7.3 Tervek	81
8 Köszönetnyilvánítás.....	82
Irodalomjegyzék.....	83
Ábrajegyzék.....	87
„A” Függelék	90
1. lépés	90

2. lépés	90
„B” Függelék	91

HALLGATÓI NYILATKOZAT

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

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2012. 12. 06.

.....
Balogh Tamás

Összefoglaló

Az elmúlt pár évben a mobiltelefonok rohamos fejlődésen mentek keresztül, és az okostelefonok megjelenésével bárki számára hozzáférhetővé váltak ezek a nagy teljesítményű ám tenyérben elférő eszközök. Már a korai okostelefonok piacra lépésével egy időben megjelentek az ezeken futó játékszoftverek, és a piaci igény ezek iránt folyamatosan nő. Később a telefonok fejlődésével a fejlesztők előtt teljes arzenál nyílt meg.

Bár az igény nagy rá, mégis kevés igazi háromdimenziós alkalmazás van a mobil alkalmazásboltokban, ezért is döntöttem úgy hogy a szakdolgozatom témája egy 3D modell megjelenítő rendszer fejlesztése lesz, mely lehetővé teszi háromdimenziós elemek valósidejű megjelenítését. Elterjedtsége, nyíltsága, valamint a natív fejlesztési lehetőségek miatt döntöttem az Android platform mellett.

Bár az Android platformot futtató telefonok is egyre gyorsabbak, mégis egy ilyen alkalmazás készítésénél egyik legfontosabb szempont a gyorsaság. A folyamatos képi élmény eléréséhez másodpercenként minimum 24 képernyőképet kell láttatnunk a felhasználóval. Ezért is döntöttem úgy, hogy bár az alkalmazást körülölelő keretet a megszokott Android alkalmazásfejlesztés eszközeivel készítem el, a megjelenítésért felelős magot viszont natív Android könyvtárak felhasználásával fejlesztem, kihasználva annak előnyeit.

A megjelenítő komponens az OpenGL ES 2.0-ás grafikus könyvtárat használja, mely már a 2.2-es Android verziótól elérhető, valamint az ehhez tartozó GLSL ES árnyaló nyelvet. Bár e grafikus könyvtár rendelkezik PC-kre szánt változattal, az ott elérhető funkcióknak csak egy része került megvalósításra mobil platformokon, így sok, PC-n már elérhető eszközt is én valósítok meg Android platformon.

Abstract

In the past few years, cell phones have been making considerable progress, and with the appearance of the smartphones, these high performance, but palm sized devices are available for everyone. As the early types of smart phones have entered the market, game software developed for these devices has also appeared, and the market-needs have ever been growing since.

Although there is growing demand in the market, there exist only few real 3D applications in the application markets. That is why I decided to choose the development of a 3D model renderer system as my thesis subject. This system provides functionality to render three-dimensional objects real-time. The reason I chose the Android platform is its popularity, openness and the possibilities of native development.

Fast as the smartphones may, during the development of this application one of the most important aspects is performance. To achieve motion picture, more than 30 frames must be seen per second. So I decided to however I make the outer-frame of the application using the usual way of Android development, the application core (responsible for rendering) is developed with the usage of Android Native Development Kit.

The renderer software component using OpenGL ES 2.0 graphics library (which is available since Android version 2.2), and GLSL ES, a shading language belongs to this library. However this library has a PC version, only some parts of that are available in the embedded version, so a lot of tools available for PCs are implemented by me for Android platform.

1 Bevezető

1.1 A platform bemutatása

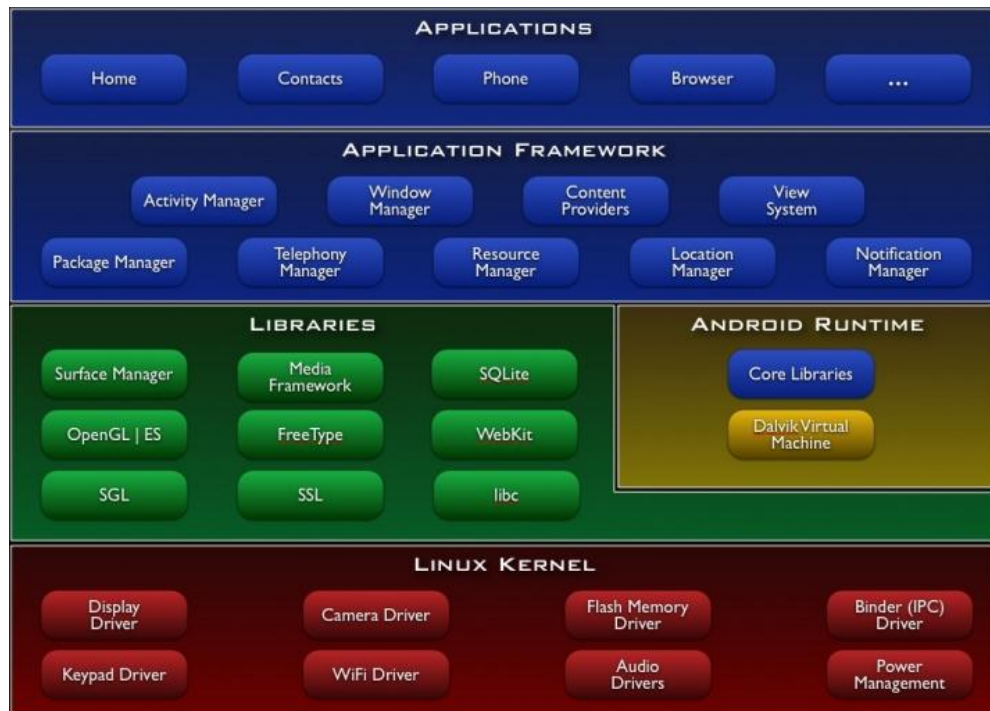
Napjaink egyik legelterjedtebb okostelefon platformja az Android. Ha máshonnan nem, a telekommunikációs cégek hirdetési kapcsán biztosan mindenki hallott már róla. Ez egy Linux kernelre épült, nyílt forráskódú, folyamatosan fejlődő okos telefon operációs rendszer és platform. Az Open Handset Alliance fejleszti, mely egy szoftverfejlesztő, operátor és hardware fejlesztő cégekből álló szervezet, de a fejlesztés jelentős hányadát a Google végzi, ezért hivatkoznak sokan az Androidra úgy, mint a Google okostelefon platformja.

Bár sok más okostelefon platform létezik a piacon, mégis ma már az eladott okos telefonok közel fele Android-alapú [1]. Az ok, amiért ennyire elterjedőben van a platform, a nyílt licence (Apache 2.0 és GNU-GPL) mellett az egyre bővülő alkalmazás bolt. Ezért egyre több okos telefonfejlesztő választja ezt a platformot, és így egyre több alkalmazás készül el.

A mai Androidos telefonok szinte mindegyike rendelkezik a processzoron kívül külön grafikus chip-el. Ezek sokkal kisebbek, mint a PC-kben használtak, és fogyasztásuk is a mobiltelefonok igényire van szabva. Mégis ezek az áramkörök teljesítményben már utoléri, az közel 10 évvel ezelőtti asztali számítógépek grafikus kártyáit, valamint várhatóan 2014-re utoléri a játékkonzolokban használt grafikus kártyákat is [2].

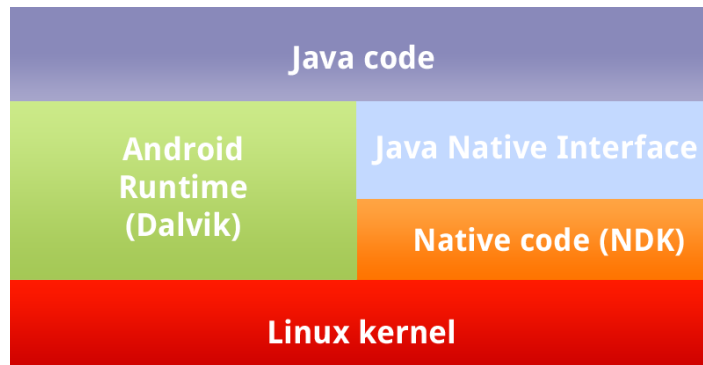
1.2 SDK és NDK

A fejlesztők egy dokumentált, és jól átgondolt fejlesztői csomagot kapnak (*Software Development Kit*, SDK [3]) mely segítségével Java nyelven fejleszthetnek szinte minden operációs rendszeren (Windows, Linux, Mac). Ez a fejlesztői csomag nagyon sok beépített osztálykönyvtárt tartalmaz (ablakozó rendszer, felhasználói felületek, hálózati kommunikáció, szenzorok... stb.), többek között OpenGL ES 2.0-ás grafikus könyvtárat valamint EGL könyvtárat is (OpenGL és az Android ablakozó rendszere közötti interface), melyek a háromdimenziós megjelenítéshez elengedhetetlenek (1.1. ábra).



1.1. ábra: Az Android Platform felépítése [7]

A fejlesztők kaptak továbbá egy eszközcsoportot *Native Development Kit* (NDK) [4] néven, mely C++ nyelven teszi lehetővé alkalmazások fejlesztését natív Android környezetben. Ezek már nem az Android rendszer Java virtuális gépén (*Dalvik*) futnak, hanem magán a Linux rendszerre épített Androidon. A fejlesztő eszköz fordítóprogramja nem a Dalvik által értelmezhető kódra fordít, hanem bináris futtatható állományokra (1.2. ábra). Az ilyen jellegű fejlesztésnek megvannak a maga előnyei és hátrányai is. Előnye hogy a futtatott kódot nem kell egy másik interpreternek értelmezni, így sokkal hatékonyabb programok írhatók így, és magas számításigényű szoftvereknél ez elengedhetetlen. Hátránya viszont hogy ennek a fejlesztő eszköznek a támogatottsága közel sem olyan jó, mint az SDK-nak, elveszítjük a felügyelt kód előnyeit, és sokkal-sokkal nehezebb, hosszabb a fejlesztés.



1.2. ábra: A Java és a Natív kód kapcsolata

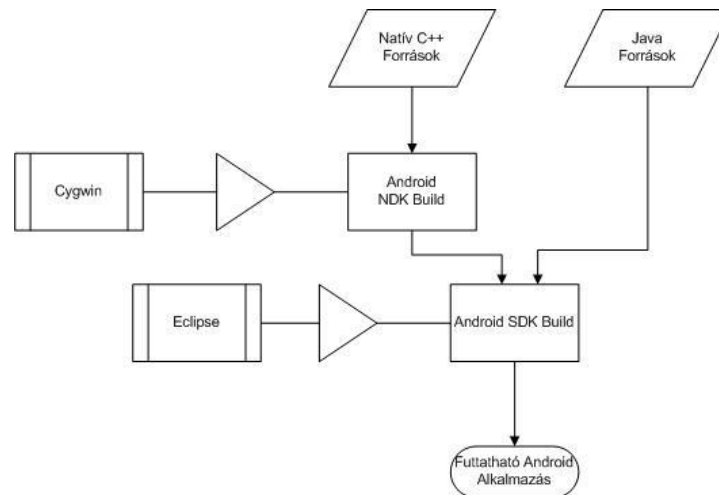
Szerencsére a két eszköz által fejlesztett komponensek csatlakozhatnak egymáshoz, jobban kihasználva mindkét oldal előnyeit. Az ablakozó rendszert, a szenzorkezelést általában az SDK-val fejlesztik, és csak a magas számítás igényű komponensek íródnak natív kódban.

1.3 Fejlesztői eszközök

Az Android fejlesztés, úgynevezett kereszt-platformos (*cross-platform*) fejlesztés, ami annyit jelent, hogy a fejlesztő eszköz (PC), illetve a fejlesztett eszköz (Telefon, Táblagép) különböző. Az Android fejlesztői eszközei alatt nem csak a kapott osztálykönyvtárakat értjük, hanem azokat a konkrét szoftvereket, melyekkel az alkalmazásokat fejlesztik. Alapvetően a fejlesztés az Eclipse nevű univerzális integrált fejlesztői környezettel (*Integrated Development Environment*, IDE) [8] történik. Ehhez a már meglévő eszközhöz készítettek az Android fejlesztést segítő kiegészítőket (*plugin*) melyek az SDK letöltése után könnyedén telepíthetők az Eclipse-be. Nem csak a fejlesztést, de a tesztelést is elősegíti az eszköz, emulálhatunk Android eszközöket számítógépünkön, vagy USB-kábellel csatlakoztatott telefonon is futtathatjuk az alkalmazásunkat. A futó alkalmazások paraméterei is megfigyelhetők, mint memória és hálózat használat, kivételek fája, valamint az alkalmazások által generált naplóbejegyzések, melyek hibakeresésnél nagyon hasznosak. Sajnos az Android emulátor nem támogatja az OpenGL ES 2.0-ás osztálykönyvtárat, így ilyen jellegű alkalmazás fejlesztésénél mindenképp szükség van egy legalább Android 2.2-es verziót futtató készülékre.

Egyelőre (2012. november) sajnos csak az SDK eszközei kerültek beépítésre az Eclipse IDE-be azért az NDK-val fejlesztett komponenseket külön fordítóval kell fordítanunk. Ez a fordítóprogram (*ndk-build*) sajnos csak Linux rendszerekre lett

fejlesztve (a futtatható állányok más platformra is le vannak fordítva), ezért Windows-on történő fejlesztésnél, szükségünk van egy újabb eszközre. Ez a *Cygwin* [9], mely képes Linux API környezetet szimulálni a rajta futó programoknak. A Cygwin terminál felületén hívható az NDK fordító programja, és azután a lefordított kód már tesztelhető az Eclipse segítségével.(1.3. ábra).



1.3. ábra: A Natív alkalmazás fordítás folyamata (Folyamatábra)

Java és az Android fejlett kivételkezelő rendszere sem használható az NDK komponensekben (bár a C++ nyelvi szinten támogatja a kivételkezelést), így egy futáskor bekövetkező hibánál csak annyit tudunk meg, hogy hiba történt, és az alkalmazás leáll. A hiba helyéről, jellegéről nem kapunk információkat, egyedül az általunk kiírt fejlesztői napló üzenetek melyek segítségével kideríthetők ezek a hibák.

1.4 Kooperáció

Mint az a témakiírásban is olvasható, munkám során felhasználok hallgatótársam Tóth Dániel szakdolgozatának („3D modell betöltő komponens készítése Android platformra”) komponenseit. A Dániel által írt modell betöltő komponens által betöltött modelleket jeleníti meg az én általam fejlesztett megjelenítő, valamint több kiegészítő funkciót is Dániel valósított meg. Kettőnk munkájaként egy olyan grafikus megjelenítő rendszer áll össze, mely képes a 3D-s tervező programok által készített modellek betöltésére, és megjelenítésére. Dolgozatomban csakis az én általam fejlesztett komponensek kerülnek bemutatásra, valamint a Dániel komponenseihez való csatlakozás módjai. Mindazonáltal a fejlesztés sokán szükségünk volt a szoros együttműködésre, a programkódok, és azok verzióinak megfelelő tárolására mely egy

ilyen összetett alkalmazásnál nem egyszerű feladat. Az általunk használt verziókövető rendszer a nyílt Subversion (SVN) [10] volt. Ez a rendszer lehetővé tette a megfelelő programverziók követését, a fájlokhoz való hozzáférés szabályozását. Subversion kliensként az ingyenes TortoiseSVN [11], míg szerverként a VisualSVN [12] szolgált, melyhez a hardware erőforrásokat a Tanszék biztosította. A különböző verziók összefésülésére a WinMerge [13] ingyenes eltérésjelző szoftvert használtuk.

1.5 A dolgozat vázlatos felépítése

Dolgozatomban a feladat részletes pontosítása után a már meglévő megoldásokat értékelem, és az azokból levonható tapasztalatokat összegzem. Majd a tervezés, és megvalósítás bemutatása során először az SDK felhasználásával készült keretalkalmazást, majd az NDK-val készített alkalmazásmag tervezését és megvalósítását írom le.

Az alkalmazásmagon belül az OpenGL alapvető eszközeitől a kész megjelenítő komponensig vezetem el az olvasót, nagy hangsúlyt fektetve a háromdimenziós képkalkotás matematikai modelljére. Végül a komponens képességeit bemutató játékal alkalmazás, és annak elemei kerülnek bemutatásra.

2 A feladat újraértelmezése

Feladatom, hogy a dolgozat első oldalán található „Szakdolgozat feladatkiírás” alapján leírt szoftvert kell terveznem. Ehhez viszont nem elegendő az a leírás, mely ott található az elkészülő szoftverre vonatkozóan, ezért tervezés előtt pontosítva újra kell specifikálnom az elkészítendő szoftverre vonatkozó követelményeket.

2.1 Kapcsolódás hallgatótársam komponenseihez

Mivel a hallgatótársam komponense által betöltött modelleket kell tudnia a szoftvernek megjeleníteni, először annak formátumát kell specifikálnom. Hallgatótársam komponense a betöltött geometriai modellt egy háromszöghálóként adja át, az én komponensemnek, mely tartalmazza a háromszögháló háromszögeinek pontjait (*vertex*), ezeket három lebegőpontos számként tárolva (x , y , z descartes koordináták). Minden ilyen ponthoz tartozik egy normál vektor mely az adott háromszög árnyalásánál játszik majd elengedhetetlen szerepet, szintén ez is lebegőpontos (*float*) számhármassal. Valamint még tartozik minden *vertex*-hez egy textúra koordináta, mely szintén az árnyalásánál kerül felhasználásra, ez viszont csak két koordinátával rendelkezik.

Azért, hogy ne kelljen egy pontot és a hozzá tartozó többi értéket többször is tárolni (egy pont több háromszög építőeleme is lehet), a háromszögek indexek segítségével épülnek fel, melyek, mind egy vertex-normál-textúra koordináta hármassal mutatnak. Ezeken felül, minden modell egy csoporttömbből áll, mely mindegyike egy háromszögcsoport (*group*) információit tartalmazza. Egy csoport egy tömböt, benne a fentebb leírt indexekkel, valamint a csoportra vonatkozó anyag jellemzőket tartalmaz. Csoporttömbnek nem lehet eleme újabb csoporttömb (nem tartalmazhat fa-szerű hierarchiában elemeket). Minden csoport csak egy anyagjellemzőt tartalmaz. Az anyagjellemzők tartalmazzák az adott csoport diffúz színét (matt szín), esetleg diffúz színtérképét (textúra), valamint spekuláris színét (megcsillanó fény színe). Textúra esetén a bittérképet képpontonként három darab 0.0 és 1.0 közötti lebegőpontos számként tartalmazza.

Az ilyen struktúrákban tárolt modelleket kell tudnia a komponensnek a fizikai valóságnak legmegfelelőbben megjeleníteni. Ehhez viszont szem előtt kell tartani a telefonok számítási korlátait. Ebből kifolyólag az úgynevezett sugárkövető módszerek,

ahol minden egyes képpontra a fizikai világ törvényeinek megfelelően indítunk sugarakat, és azok útját követjük, nem használhatóak, mert azok annyira számítás igényesek, hogy nem kapnánk mozgóképet a mai telefonok hardvereivel. Ehelyett a háromszögeket transzformáljuk úgy a képernyő kétdimenziós koordinátarendszerébe, hogy azok a háromdimenzió hatását keltsék, majd a kapott normál vektorok segítségével árnyaljuk őket.

2.2 A grafikus megjelenítő egység tervezése

Ezen a platformon a támogatott grafikus könyvtár, az OpenGL ES 2.0, valamint az OpenGL ES 1.1 [5] melyek a fent említett módszerrel való megjelenítéshez ideálisak. Az OpenGL ES 1.1 még fix grafikus csővezetékkel dolgozik, tehát a grafikus processzorra már rátöltött elemek viselkedését már nem befolyásolhatjuk. A 2.0-ás változat azonban már programozhatóvá teszi a csővezeték egyes elemeit, így nagyobb szabadságot biztosítva a fejlesztőknek, emellett számos fejlesztést is tartalmaz. Ezért döntöttem a 2.0-ás verzió mellett, bár a nagyobb fejlesztői szabadság miatt ez később többletmunkával is járt.

Még azelőtt, hogy a háromszögek adatai rátöltődnének a grafikus hardverre, sok matematikai számítást kell végezni (transzformációk, árnyalási paraméterek számítása) és mind ezt a processzoron. Ezért döntöttem úgy, hogy bár a keretalkalmazást a megszokott Android fejlesztői eszközök segítségével, Java nyelven, az SDK-val fejlesztem, addig a grafikus megjelenítő komponens magját az NDK-val, C++ nyelven.

A fent említett grafikus megjelenítési módszerhez nagyon sok transzformációs és vektoralgebrai eszközre lesz szükségem (vektorok és szorzataik, mátrixok, inverz, forgatási és mozgatási transzformációk, ezek transzformáltjaik) hogy ezeket felhasználva tudjon a komponens megjeleníteni, valamint pozicionálni elemeket a háromdimenziós térben. Mivel sajnos nem áll rendelkezésemre olyan osztálykönyvtár Android környezetben, mely ezeket elvégezné, (hasonló, mint PC-n OpenGL környezetben a GLUT [6]) ezért ezeket is implementálnom kell a feladat megoldása során. A rendszernek egyszerre több (mind ugyanolyan, mind különböző) modellt is meg kell tudnia jeleníteni.

2.3 Felhasználói felület

A grafikus felhasználó felület, mely az alkalmazás irányításáért felelős nagyjából a keretalkalmazásban kerül megvalósításra, de az alkalmazásmag is lehetőséget nyújt úgynevezett HUD elemek rajzolására, melyek kétdimenziós bittérképek, a háromdimenziós kamera által látott kép elé rajzolva. Ehhez a rendszert fel kell készíteni arra, hogy egyszerre több kamera képét is meg tudja jeleníteni, valamint több típusú kamerát kell készíteni (A HUD elemek nem szenvedhetnek perspektivikus torzítást, míg a többi elem igen).

2.4 Szenzorkezelés

A szenzorkezelés elengedhetetlen, hogy a felhasználó kapcsolatot tarthasson az alkalmazással, valamint a felhasználótól érkező eseményekre a komponens megfelelő módon módosítsa a háromdimenziós világ és a kamera állapotát. Bár a mai okos telefonok fejlett szenzorokkal rendelkeznek (gyorsulás mérő, GPS, fényerőmérő...), az alkalmazás csak az érintőképernyő érintést detektáló szenzorját használja fel. Ennek fő része az alkalmazáskeretben kerül megvalósításra, majd a megnyomott képernyő pozíció koordinátájával hív egy megfelelő natív függvényt. A nyomva tartott érintések kezelése is az alkalmazás feladata (a nyomva tartás ideje alatt is változnia kell a látott képnek).

2.5 Kompatibilitás

Bár hivatalosan az Android rendszer jelenleg (2012. november) a 4.2-es (*Ice Cream Sandwich*) verzióval tart, a piacon fogalomban lévő eszközök nagy része még közel nem rendelkezik a legújabb verzióval, sőt sokuk még 2.3 vagy az előtti verziókat futtatnak [14]. Feladatom, hogy az alkalmazás visszafelé is kompatibilis legyen, egészen a 2.2-es Android verzióig. Ezért a keretalkalmazásban nem használok olyan elemeket, melyek a 3.0-as Andoriddal jelentek meg, és kompatibilitásuk nem megoldott az régebbi verziókon. Az alkalmazásmagban használt könyvtárak (OpenGL ES, STL) mindegyike kompatibilis a 2.2-verzióig visszamenőleg, viszont attól régebbi verziókon nem támogatott az OpenGL ES 2.0, mely a komponens szerves része, így csak addig oldható meg a kompatibilitás.

2.6 Modellek

Az megjelenítő komponens képességeinek demonstrálására létrehozott játékkalkalmazás természetesen igényel modelleket, melyeket megjelenít. Az egyszerűbb geometriát tartalmazó modelleket, mint befoglaló doboz (*skybox*), út én készítettem el. Mivel ezek sokkal inkább művészibb, mint mérnöki munkák, ezek készítésének lépéseire nem szeretnék szakdolgozatomban kitérni. A bonyolultabb geometriákat tartalmazó modellek, a járművek, pedig szabadon felhasználható modellek, melyek forrás a függelékben található.

3 Irodalomkutatás

Az alkalmazással szemben támasztott követelmények részletes megfogalmazása után a hasonló, már meglévő munkákat kutattam fel. Ehhez a *Mobile Game Engines* [15], telefonos játékfejlesztést segítő szoftverekkel foglalkozó weboldal által nyújtott információkat használtam fel. Alapvetően csak olyan játékmotorokat kerestem, melyek Androidon elérhetőek. Fontos keresési kritérium volt továbbá, hogy az eszköz mindenképp támogasson valódi háromdimenziós megjelenítést, valamint hogy ingyenes legyen. Az általam kipróbált megoldások a *libGDX* [16], valamint a *min3D* [17] voltak. Azért ezekre esett a választásom, mert mindkettő nyílt forrású, valamint nagy felhasználó bázisú, jól dokumentált keretrendszerek.

3.1 Min3D

A min3D csak Androidra elérhető megjelenítő komponens. Ez az eszköz nem nyújt teljes körű beállításokat a felhasználónak, így jóval egyszerűbb a kezelése, a háromdimenziós alakzatok megjelenítése nem igényel hozzáértést a grafikus könyvtárakhoz. Csak grafikus eszközöket nyújt (nincs hang, szenzorkezelés stb.). A rendszer teljes egészében Java nyelven írt, és a felhasználója is Java nyelven éri el az eszközöket. Még az OpenGL 1.1-et használja.

3.2 A libGDX

A libGDX segy keresztplatformos fejlesztői eszköz, mely an Android mellett Windows, Linux, és MacOS rendszerekre is elérhető. Ahol elérhető ott OpenGL ES 2.0 –át használ, ahol nem ott OpenGL ES 1.1-et, valamint a teljesítmény igényes kódrészletek natív kódban kerültek megvalósításra. A felhasználó Java nyelven használhatja a komponenst. A grafikai megjelenítésen kívül még számos egyéb funkcióval rendelkezik (hangkezelés, fizikai motor, stb.). A grafikai megjelenítő eszköz többek között keppes háromszöghálók megjelenítésére, ezek anyagjellemzőinek, textúráinak megjelenítéséhez, valamint árnyalásra. Viszont széles eszközkészletének hátránya, hogy nagy hozzáértést igényel a felhasználótól, már egyszerű háromdimenziós alakzatok megjelenítéséhez is szükséges a grafikai osztálykönyvtárak (OpenGL) ismerete.

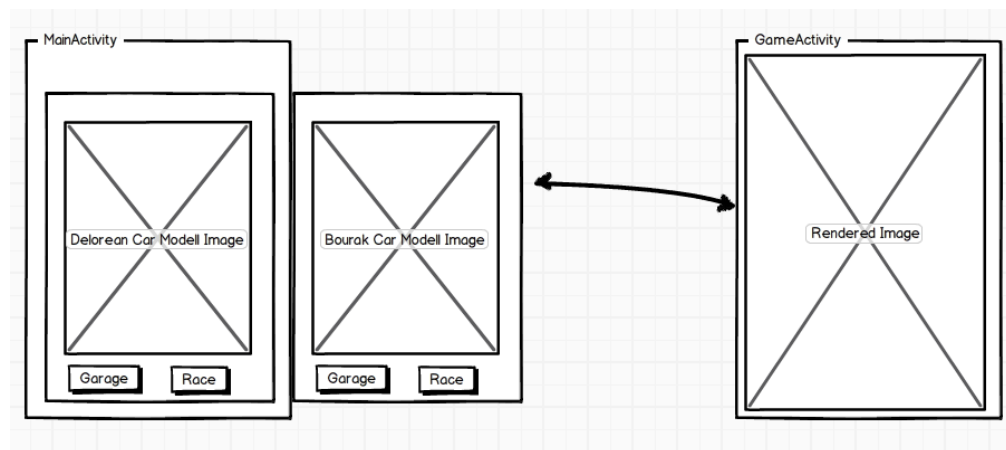
3.3 Következtetések

Az irodalomkutatás során végig szem előtt tartottam az általam fejlesztendő komponens feladatait, és a specifikációban kitűzött célokat. Ezért döntöttem úgy hogy a komponens nem enged beleszólást a felhasználónak az árnyalás, és grafikai megjelenítés megvalósításának részleteibe, mivel ez mély hozzáértést igényelne. Viszont cél, hogy a rendszer kihasználja az OpenGL ES 2.0 nyújtotta előnyöket, valamint teljesítmény szempontból a nagyrészt natív kód hatásai is érezhető legyen. A fentebb ismertetett eszközöktől eltérően teljesítmény szempontokból a felhasználó C++ nyelven tudja felhasználni a komponenst.

4 Az alkalmazáskeret

4.1 Előzetes tervek

A keretprogram magában foglalja az egész alkalmazást, így az alkalmazásmag működéséhez elengedhetetlen, hogy a keret megfelelően legyen megtervezve. A konkrét kód/osztály szinten történő tervezés előtt az alkalmazás képernyőit, és az ezek közötti navigációt az úgynevezett *Screen Flow*-t terveztem meg (4.1. ábra). Ehhez a *Balsamiq Mockups* [18] drótváz tervező alkalmazást használtam. Majd a szükséges képeket, és gombokat rajzoltam meg, az *Adobe Photoshop* [19] szoftver segítségével. Az autó modellekhez mellékelt képeket is felhasználtam.

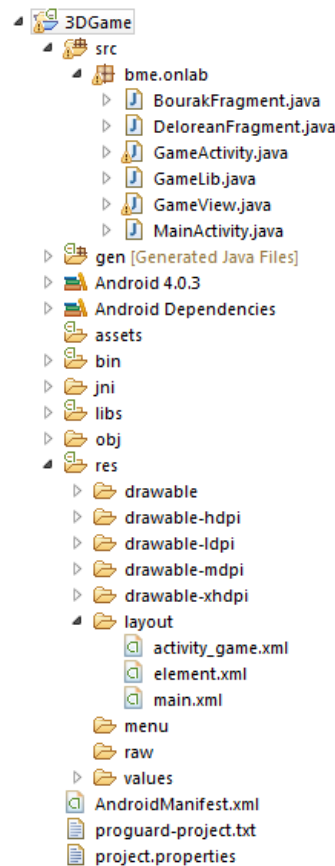


4.1. ábra: A megtervezett „Screen Flow”

4.2 A project felépítése

A project felépítésének szemléltetésére, az Eclipse szoftver *Project Explorer* eszköze által készített projekt fát fogom használni (4.2. ábra). Az *src* mappában a forrás, azaz *source* fájlok találhatóak. Ezek csak a keretprogram forrásai. Ezen mappában belül a Java-ban megszokott módon *package*-ekbe rendezve találjuk a forrásállományokat, egyetlen *package* van, a „bme.onlab”. A második fontos könyvtár a *res*, azaz *resources* könyvtár, mely a keretprogramhoz szükséges erőforrás fájlokat, mint képek, felület definíciós fájlok, szöveg fájlok tartalmaz. A *jni* könyvtár tartalmazza a natív komponenseket, azaz az alkalmazásmagot. Ezt részletesen a további fejezetekben lesz kifejtve.

A többi könyvtár a fordításhoz szükséges Android könyvtárakat, illetve a lefordított, illetve generált fájlokat tartalmazza. Ezekre részletesen nem szeretnék kitérni, mivel ezek tartalma az alkalmazás tervezése szempontjából irreleváns. A gyökérkönyvtárban található fájlok a fordítási, illetve Android specifikus beállításokat tartalmazzák.

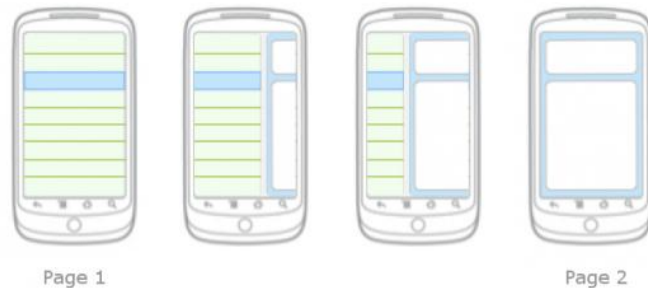


4.2. ábra: A Projekt felépítése

4.3 A felület megvalósítása

A felületeket (*UI, User Interface*) az Android SDK-nak, és az Android szoftverfejlesztési gyakorlatnak megfelelően XML-ben készítettem el. Ebben a UI elemeket fa hierarchiában hoztam létre, majd a statikus tulajdonságaikat (például a képek forrása) is deklaráltam. A felületek elemek többsége az általános Android UI elemek (*Widget*) közül kerültek ki, mint Kép (*Image*), Képgomb (*ImageView*). Egyetlen elem van mely nem a beépített *Widget*ek között van, ez pedig a *ViewPager*. Ez az elem az úgynevezett Android Támogató Csomagban (*Android Support Package*) [20], annak is a v4-es verziójában érhető el. Ez a csomag szolgál arra, hogy az újabb Android verziókban bevezetett funkciók, a régebbi rendszereken is elérhetőek legyenek. (Ennek

használatához importálnom kellett ezt az Eclipse Project-be). A *ViewPager* lehetővé teszi több oldal megjelenítését, az oldalak között 'söpréssel' (*swipe*) válthatunk (1.1. ábra).



4.3. ábra: A *ViewPager* elem működése [21]

4.4 Építő elemek

Egy Android alkalmazás alapvetően négy típusú építő elemből állhat. Ezek az *Activity*, *Service*, *Broadcast Receiver*, *Content Provider* [22]. Az *Activity* képernyőlapként funkcionál. Általában a különböző lapok egy alkalmazásban, különböző *Activity*-k is. A *Service*, mint neve is mutatja egy szolgáltatás, mely a háttérben fut. A *Broadcast Receiver* a rendszerben lévő körüzenetekre tud feliratkozni, a *Content Provider* pedig adatokat tud kiejánlani más alkalmazások részére. Utóbbi hármat nem használtam fel az alkalmazásban csak az *Activity*-t.

A 3.0-ás Android verzióban az SDK egy olyan eszközzel is bővült, mely lehetővé teszi, hogy egy *Activity* több Lapot, Lap darabot tartalmazzon. Ez *Fragment* (*Fragment*) [23]. Ezeket cserélhetjük, pozícionálhatjuk akár futás közben is. Bár csak a 3.0-ás verzióban jelent meg ez az eszköz, a már említett *Support Package* segítségével az 1.6-os Android verzióig visszamenőleg használható.

Az alkalmazás kerete kép képernyőből áll. Ezek a Főképernyő (*MainActivity*), valamint a Játékképernyő (*GameActivity*). A *MainActivity* további két lapot tartalmaz, ezek a *DeloreanFragment*, valamint a *BourakFragment*. A már említett *ViewPager*-ben ezek a *Fragment*ek lesznek a különböző lapok, melyek között váltani lehet.

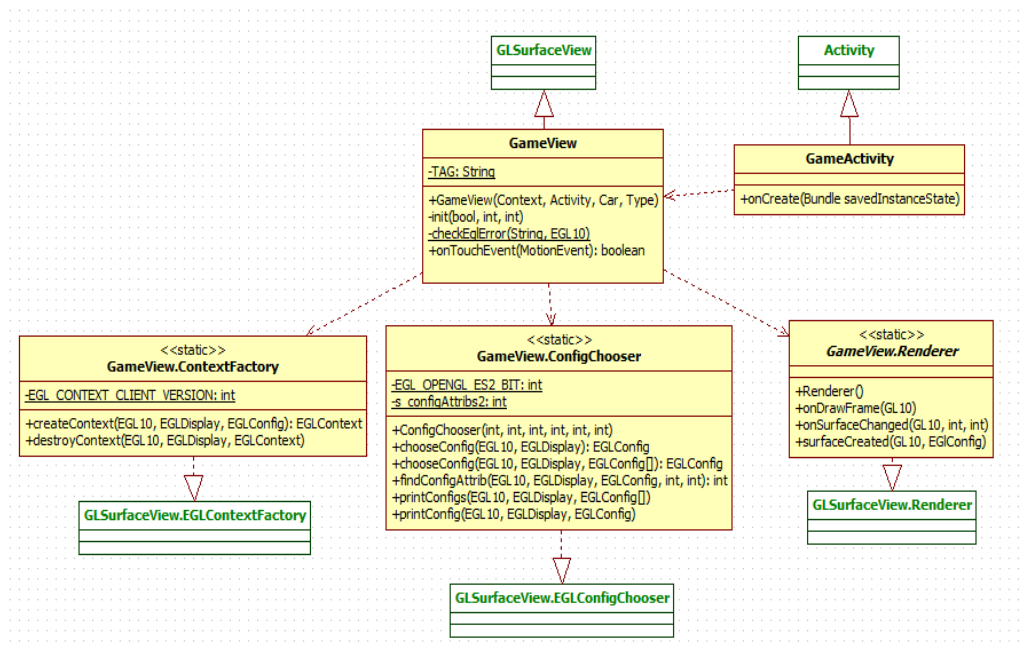
A Főképernyő lehetővé teszi, hogy a felhasználó söpréssel válaszson a két autó közül, hogy melyiket szeretné a játékban látni/irányítani. A *Fragment*ek tartalmazzák, az adott autó képét, valamint két gombot, *Garage* illetve *Race* felirattal. Ezekkel a

4.5 A főképernyő



Mint már fentebb említettem, a Főképernyő tartalmaz egy *ViewPager* elemet melyben két *Fregment*et jelenít meg. Ahhoz hogy ezeket a *Fregment*eket át tudjuk adni a *ViewPager*-nek szükség van egy *FragmentPagerAdapter*-ből származó adapter osztályra, ez a *MyPagerAdapter*. Itt vannak a tárolva a *Fregment*ek példányai (*cars*), és a *ViewPager* ettől az osztálytól kéri el az aktuális elemet (*getItem(..)*), valamint hogy hány elem van összesen (*getCount()*). A Főképernyő, valamint a *Fregment*ek *onCreate(..)* illetve *onCreateView(..)* függvényei az Android rendszer által hívott függvények, melyek az elemek létrejöttkor hívódnak, és az adott elem felépítésének lépéseit tartalmazzák.

4.6 A játékképernyő



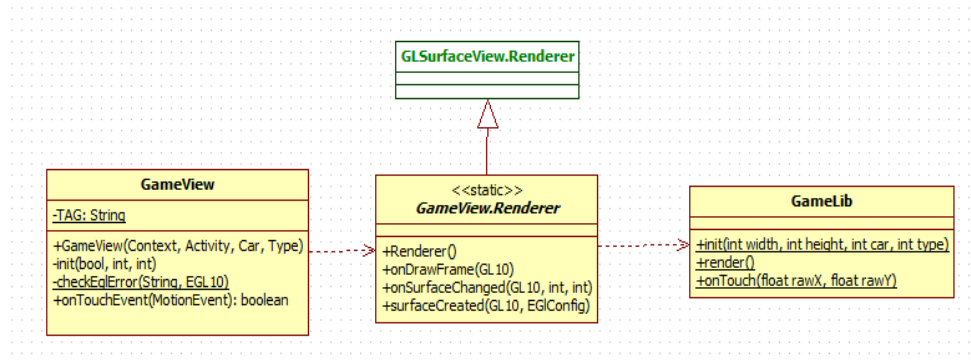
4.5. ábra: A Játékképernyő és környezete (UML Osztály Diagram)

A fenti diagram (4.5. ábra) tartalmazza a Játékképernyőt (*GameActivity*) és hozzákapcsolódó osztályok hierarchiáját. A jelölési konvenciók az előző diagrammal megegyezők. A Főképernyőhöz, valamint az alkalmazásmaghoz való kapcsolatok nincsenek jelölve. A Játékképernyő megjeleníti az alkalmazásmag által küldött képet, és ezen belül van megvalósítva az maghoz való kapcsolódás is. A kapott képet tartalmazó osztály/felhasználói felületi építőelem a *GameView*. Ez az elem a *GLSurfaceView*-ből származik, melynek feladata, hogy a grafikus rendszer által küldött képet tartalmazza, valamint azt megfelelően felépítse (*init(..)*), kezelje az eseményeit (*onTouchEvent(..)*), valamint futás közbeni hibáit (*checkEglError(..)*). A grafikus rendszer kontextusának felépítéséért a *ContextFactory* belső osztályfelelős. Az éppen aktuális készüléken elérhető konfigurációs beállítások megtalálásáért a *ConfigChooser* fele. Itt kerül beállításra a színmélység, a többszörös mintavételezés (*Multisampling*), ha lehetséges, illetve a mélységtár (*Depth Buffer*) is. A belső mag függvényeihez való kapcsolódást a *Renderer* osztály biztosítja.

A megfelelő beállítások megtalálására, valamint a mag (OpenGL-t használva) és a Grafikus rendszer kapcsolatát az EGL könyvtár biztosítja. Az EGL könyvtár egy Beágyazott-rendszer Grafikus Könyvtár (*Embedded-Systems Graphics Library*), mely az OpenGL könyvtár és az Android ablakozó rendszere közötti kapcsolatot hivatott

kezelni [24]. A GLSurfaceView, illetve belső osztályai az EGL-hez kapcsolódnak, és az EGL kapcsolódik majd a belső maghoz. (A forráskód részben az Android NDK példaalkalmazásainak felhasználásával lett írva [27]).

4.7 Kapcsolódás a maghoz



4.6. ábra: Az alkalmazásmaghoz való kapcsolódás Java oldalról

A fenti diagram (4.6. ábra) mutatja be a maghoz való kapcsolódás részleteit. Az előző diagramokhoz hasonlóan, ez a diagram sem teljes, csak a kapcsolódáshoz fontos részleteket mutatja be, és a már megszokott jelöléseket használja. A már ismertetett *Renderer* osztály, a *GameLib* osztály statikus függvényit hívja. A *Renderer* osztály *onDrawFrame(..)* függvénye ütemezetten hívódik, amikor a *GameView* újrarajzolást kér (amikor csak lehet mindig), és ez hívja a *GameLib* *render()* függvényét. A *Renderer* *onSurfaceChanged(..)* függvénye akkor hívódik meg, amikor a rajzolási felület megváltozott, ezért azt újra létre kell hozni. Ez a függvény a *GameLib* *init(..)* függvényét hívja. A *GameView* *onTouchEvent(..)* függvénye, pedig a *GameLib* *onTouch(..)* függvényét hívja meg. A *GameLib* függvényeinek megvalósítása már az alkalmazásmagban történik, ezért a *GameLib* osztály a konkrét kapcsolódási pont a natív komponenshez. A majd később kifejtett *native.cpp* fájlban található *init(..)*, *render()*, ill *onTouch()* függvények a *GameLib* függvényeinek natív implementációi.

4.8 Szenzorkezelés

A szenzorkezelés egy része is a keretalkalmazásban történik, és az itt kapott értékek kerülnek továbbításra az alkalmazásmag felé. A *GameView* a *GLSurfaceView*-on keresztül a *View* Android osztályt is megvalósítja. A *GameView* osztály feldefiniálja a *View* *onTouchEvent(MotionEvent event)* függvényét. Ez a függvény akkor hívódik meg, hogyha az adott felületi elem egy pontját megérintették az érintőképernyőn. A

MotionEvent típusú *event* változó tartalmaz információkat a megnyomásról, például hogy hol történt, illetve hogy felengedte-e az újat a felhasználó, vagy épp akkor tette-e le. (Mind a felengedés, mind a lenyomáskor történik esemény). Ez a függvény hívja meg a *GameLib onTouch* függvényét. Ezen hívás során átadja a megnyomott pont X és Y koordinátáját a bal felső saroktól számolva pixelekben. Ha felengedés történt, akkor a -1, -1 értékeket adja át. A natív komponensben ezután ellenőrizhető hogy lenyomás vagy felengedés történt-e (ha negatív, akkor felengedés, ha nem lenyomás). Mivel a képernyőnek nincs negatív számú pixele ezért ez a módszer használható.

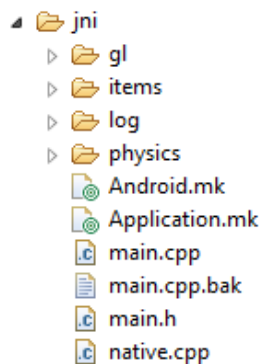
5 A grafikus komponens

5.1 Bevezető, kapcsolat az alkalmazáskerettel

5.1.1 Az alkalmazásmag felépítése

Az alkalmazásmag teljes egészében a projekten belüli *jni* (*Java Native Interface*, Java Natív Kapcsolódási pont [25]) mappán belül található. Ezen belül funkciójukat tekintve vannak csoportosítva az egyes fontosabb alkomponensek külön alkönyvtárakban (5.1. ábra). A gyökérkönyvtárban a Főprogram (*main.cpp*, *main.h*), a natív kapcsolódási felület (*native.cpp*) valamint a program fordítási és linkelési beállításai (*Application.mk*, *Android.mk*) találhatóak (valamint a fordítás során generált .bak fájlok).

A *gl* (*Graphics Library*) könyvtár tartalmazza a grafikus megjelenítő komponenst. Az *items* mappa tartalmazza a megjelenítendő modelleket körülvevő osztályokat, ezek a grafikus játék-alkalmazás entitásai. A *log* mappa az eseménynaplóba való naplózáshoz szükséges eszközöket tartalmazza. A *physics* mappában az alkalmazásban használt fizikai szimulációs eszközök vannak, melyeket Dániel készített el.



5.1. ábra: Az alkalmazásmag felépítése

5.1.2 Fordítási és linkelési beállítások

Az *Android.mk* fordítási és linkelési beállításokat tartalmazó file felelős a megfelelő rendszerkönyvtárak, valamint az alkalmazásmag forrásfájljainak betöltésért. A ' *LOCAL_SRC_FILES :=...* ' változó tartalmazza a forrásfájlok felsorolását (csak a *.cpp állományok, a fejlécfájlok (*headers*) itt nincsenek felsorolva). A

'LOCAL_LDLIBS:=...' tartalmazza a betöltendő könyvtárak. A '-llog' az eseménynaplóhoz szükséges eszközök, a '-lGLESv2' az OpenGL ES 2.0 könyvtárainak, a '-ldl', '-stdc++' pedig a C++ fordításhoz szükséges könyvtárak kapcsolói. A 'LOCAL_MODULE:= ' a modul nevét adja meg (az alkalmazás keretben, a *GameLib* osztályban való betöltésnél ezt kell megadni.). C++ nyelv része a STL (*Standard Template Library*) [26], mely hasznos generikus osztályokat, adatstruktúrákat (dinamikusan növekvő vektor, halom... stb.) tartalmaz. Az *Application.mk* file tartalmazza a STL könyvtárak betöltéséhez szükséges fordítási és linkelési beállításokat.

5.1.3 Esemény napló

Az hibakeresés valamint tesztelés szempontjából fontos, hogy a futás közben bekövetkező eseményekről információkat kaphassunk. Mivel natív környezetben nem támogatottak a különböző hibakeresési (*Debug*) funkciók, mint például a töréspontok, hívási fa... stb., valamint az OpenGL ES 2.0 sem támogat karakter megjelenítést a készülék képernyőjén, így az eseménynapló az egyetlen hibakeresési lehetőség. A telefonon bekövetkező események valós időben megfigyelhetők a fejlesztőeszközben (DDMS nézet, *Logcat* fül), valamint alkalmazás (*PID*, *TID*, *Level*), idő, illetve címke szerint szűrhetők, kimenthetők az események. Java környezetben az *android.util.Log* osztály segítségével küldhetők ilyen üzenetek. Az alkalmazásmagban ez némileg bonyolultabb. A *log/log.h* fejlécfájlban található egy olyan előre definiált „inline” függvény (*LOG(...)*), mely a paraméterül kapott karakterfüzért (mely a C beépített *printf(...)*-hez hasonlóan paraméterezhető) vár, és azt az eseménynaplóba bejegyzi. Itt van a szűrést segítő '*3DEngine*' címke is definiálva.

5.1.4 Native.cpp

A Java oldali *GameLib* osztály függvényei a *Native.cpp* osztályban vannak megvalósítva. Itt elsőnek az '*extern C{...}*' blokkon belül fel van sorolva minden csatolt függvény deklaráció ezzel jelezve, hogy ezek C függvények (bár C++ kódot is tartalmazhatnak, de a *jni* kötéshez C függvényekre van szükség), majd a függvények definíciója következik. A függvények elnevezése pontos szabályokat követ. A '*JNIEXPORT*' kulcsszó jelzi, hogy ez egy exportált függvény (Java oldal felé). Majd visszatérési érték és '*JNICALL*' kulcsszó. A függvénye neve szigorúan '*Java_[package]_[class]_[function]*' formában van megadva. A függvény paramétereire is vonatkoznak megkötések. Elsőnek egy *JNIEnv** típusu objektumot kap,

majd a Java oldalon deklarált paraméterek, melyek típusa előtt egy 'j' karakter szerepel. Például az inicializáció függvény Java és C++ oldali megfelelői:

Java (*GameLib.java*):

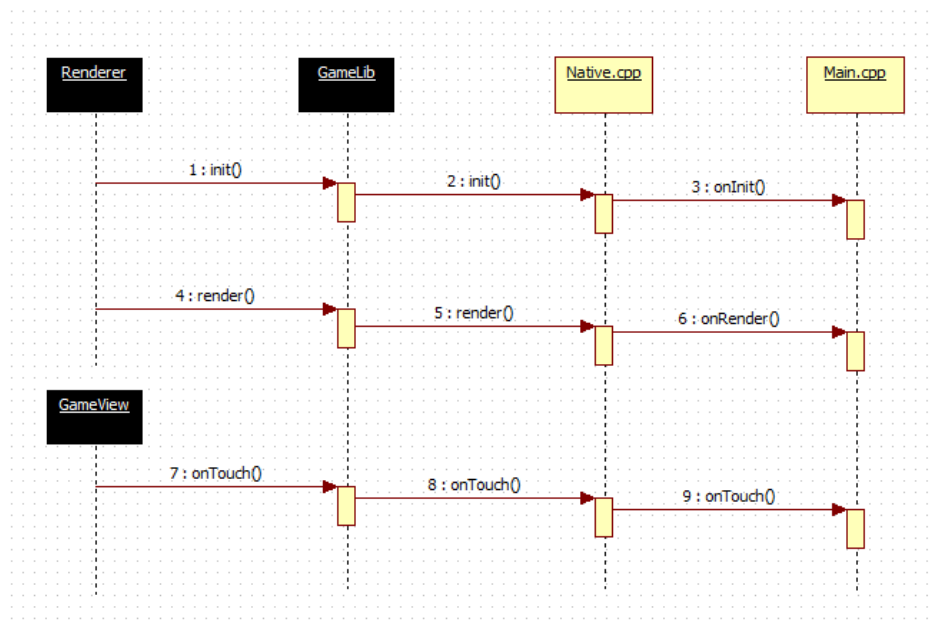
```
public static native void init(int width,int height,int car, int
type);
```

C++ (*Native.cpp*):

```
extern "C"
{
    JNIEXPORT void JNICALL Java_bme_onlab_GameLib_init(JNIEnv *
env,jobject obj, jint width, jint height,jint car, jint type);
...
}

JNIEXPORT void JNICALL Java_bme_onlab_GameLib_init(JNIEnv *
env,jobject obj, jint width, jint height,jint car, jint type)
{
    onInit(width,height,car,type);
}
```

Azért hogy a natív kapcsolat paraméterei ne zavarják a későbbi fejlesztést, az egész kapcsolódási paramétereket tartalmazó kódrész a *Native.cpp*-ben került implementálásra, majd a *Main.cpp* megfelelő metódusait hívja meg minden metódus a *Native.cpp*-ben.



5.2. ábra: A Natív kód kapcsolata a Java oldallal (UML Szekvencia Diagram)

A fenti szekvencia diagramon (5.2. ábra) a sötét háttérrel jelzett osztályok a Java oldali osztályok, az többi pedig C++ oldali file. Minden olyan esetben ahol a diagramon

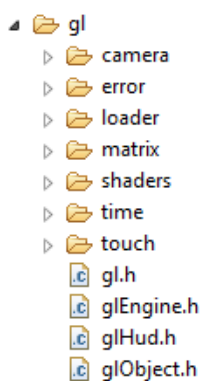
az osztály nevében szerepel a `.cpp` vagy a `.h` karaktersorozat, ott az adott osztály valójában egy C++ fájlként, nem konkrét osztályként van megvalósítva, és a diagramon jelzett metódusai a fájlban deklarált függvények.

A *Main.cpp* logikailag az egész mag központi fájlja, a játéklogika logika fő része itt van megvalósítva. Ezt a fájlt az egyes alkomponensek bemutatása után fogom bemutatni.

5.2 A grafikus könyvtár

5.2.1 A grafikus könyvtár felépítése

A grafikus megjelenítő komponens a */jni/gl* könyvtárban található. A fő funkciók külön könyvtárban találhatóak (5.3. ábra). A *camera* könyvtár a kamerák, valamint kamerakezeléshez szükséges funkciókat tartalmazza. Az *error* könyvtár a különböző grafikus hibadetektálási függvényeket tartalmazza. A *loader* könyvtár tartalmazza a Tóth Dániel által elkészített betöltő komponenst. A *matrix* könyvtárban a mátrix, valamint vektor műveletekhez szükséges funkciók találhatóak. A *shaders* könyvtár az árnyaló programokat tartalmazza. A *time* könyvtárban az időkezeléshez a *touch*-ban pedig az érintéskezeléshez kapcsolódó funkciók találhatóak, melyet részben Dániel készített el. Bár az idő- és érintéskezelés nem tartozik szorosan a grafikus megjelenítéshez, azonban felhasználásához szinte elengedhetetlen, valamint az OpenGL asztali verziójához tartozó kiegészítő könyvtárak is tartalmazznak ilyen funkciókat (pl. GLU, GLUT [6]) ezért is találhatóak meg itt.

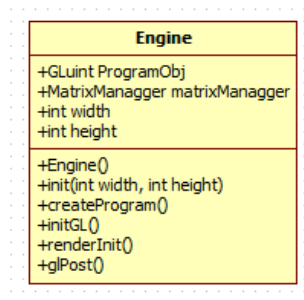


5.3. ábra: A grafikus komponens felépítése

A gyökérkönyvtárban található *gl.h* a grafikus könyvtár komponenseinek betöltését (*includes*) tartalmazza. A *glEngine.h* a grafikus motor főosztálya, a *glObject.h* a grafikus egységobjektum (nem azonos a játékkalkalmazás entitásával, hanem része). A *glHud.h* a HUD (a játéktérrel készített kép elé rajzolt kétdimenziós felhasználói felület) kezelésért felelős.

5.2.2 A motor

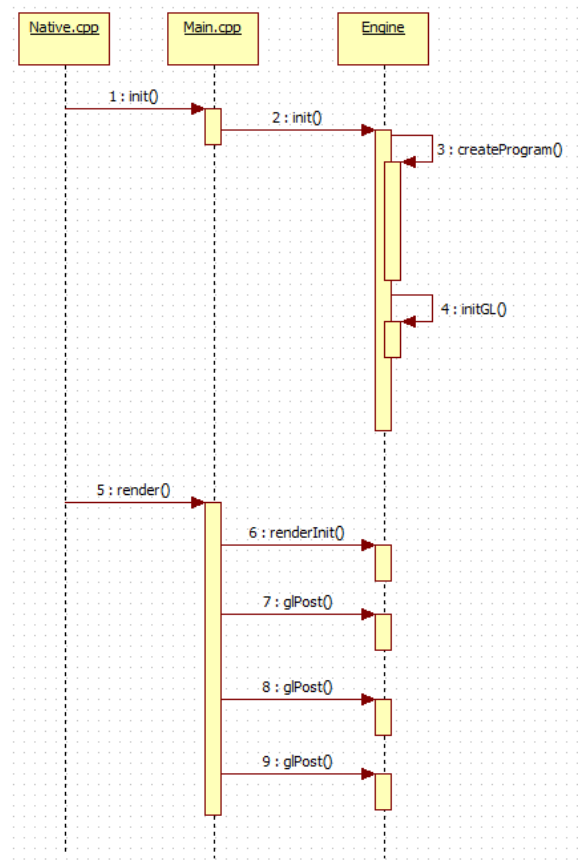
A Motor (*Engine*) osztály a *glEngine.h* fejlécfájlban található. Ez az osztály felelős az egész grafikus megjelenítés felépítésért, és menedzselésért (5.4. ábra). A *ProgramObj* egy *GLuint* típusú változó. Az OpenGL saját típusokat is használ, melyek általában a C++-os típus GL karakterekkel az elején. A különböző erőforrásokhoz, mint pl. Program, Textúrák, stb.. az OpenGL egy előjel nélküli egész számot rendel (*GLuint*) mellyel az később azonosítható. A *ProgramObj* az OpenGL Program azonosítóját tartalmazza.



5.4. ábra: Az Engine osztály (UML Osztály Diagram)

Az OpenGL Program egy olyan elem, mely egy megjelenítő kontextust azonosít. Ehhez a kontextushoz vannak csatolva az árnyaló programok. Az OpenGL ES 2.0 két árnyaló programmal dolgozik, ezek a Háromszög árnyaló (*Vertex Shader*), valamint Képpont árnyaló (*Fragment Shader*). Az árnyaló programok GLSL ES (OpenGL ES Shading Language) [29] nyelven írt programok, melyek a grafikus hardveren futnak. Ezeket minden kontextus felépítésnél le kell fordítani, majd rátölteni a grafikus hardverre. Ezután a programoknak adott bementeket is előbb fel kell tölteni a grafikus kártyára, majd azok lefutnak a bemenetekre, és így kapunk grafikus képet. Ezek az *Engine* osztály fő feladatai. Az árnyaló programokat egy későbbi fejezetben fogom bemutatni.

5.2.3 A motor funkciói



5.5. ábra: Az Engine függvényeinek hívási szekvenciája (UML Szekvencia Diagram)

Az *init(int width,int height)* függvény a grafikus kontextus fő inicializációs függvénye. Paraméterül a felépítendő grafikus felület méreteit kapja. Ez a függvény hívja tovább a *createProgram()* illetve *initGL()* függvényeket (5.5. ábra). A *createProgram()* függvény felelős az OpenGL program felépítésért, az *initGL()* pedig az OpenGL specifikus beállítások beállításáért.

A *createProgram()* elsőként létrehoz egy azonosítót (*GLuint*) a készítendő árnyaló programnak, majd - egy még üres - *Vertex Shader*-t ezzel az azonosítóval. Ehhez kell csatolni a bementi programot (egyenlőre a program forrása karakterláncként). Majd le kell fordítani a már felcsatolt *shader*t. Ezután ellenőrizni kell, hogy történt-e fordítási hiba. A *Vertex Shader* létrehozása, és fordítása:

```
GLuint VertexShaderObj;
VertexShaderObj = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(VertexShaderObj, 1, &VertexShaderFile, NULL);

glCompileShader(VertexShaderObj);
testShaderCompileError(VertexShaderObj);
```

Természetesen ezután a *Fragment Shader* is ugyan ilyen módon kell létrehozni, és fordítani. Majd magát az OpenGL Programot kell létrehozni, és ehhez csatolni kell a már lefordított *shader*eket. A csatolási hibák ellenőrzése után, aktiválni kell, azaz használni kell az OpenGL Programot.

```
ProgramObj = glCreateProgram();

glAttachShader(ProgramObj, VertexShaderObj);
glAttachShader(ProgramObj, FragmentShaderObj);

glLinkProgram(ProgramObj);
testProgramLinkError(ProgramObj);

glUseProgram(ProgramObj);
testGLError();
```

Megjegyzés: Az OpenGL ES 2.0ban lehetőség van egyszerre több lefordított, és csatolt OpenGL program használatára is, ha például ha más-más elemeket más-más technikával szeretnénk árnyalni. Ekkor az inicializálásnál több ilyen ProgramObj-ot kell létrehozni, és a különböző elemek kirajzolása előtt a hozzájuk tartozó ProgramObj-ot használni. Jelen esetben csak egy ilyen programot használunk, mivel hasonló módon árnyalunk minden elemet.

Az *createProgram()* lefutása után már rendelkezünk egy aktív OpenGL Programmal, majd az *initGL()* függvény állítja be a megfelelő OpenGL beállításokat. Itt történik meg a látótér paraméterezése (*glViewport(..)*), valamint a Színkeverés (*Blending*) és Mélységteszt (*Depthtest*) funkciók beállítása. Ezen funkciónak akkor van szerepe, amikor egymást elfedő elemeket szeretnénk megjeleníteni.

A *Depth Test* funkció arra szolgál, hogy eldöntsük, egy elem takarja-e a másikat. Ehhez szükség van egy úgynevezett Mélységtárolóra (*Depth Buffer*). Ez egy akkora tároló, mint a képernyő képpontjainak a száma, és minden képponthoz tartalmaz egy értéket, ami azt jelzi, hogy az adott képpont milyen közel van a kamerához. Amikor Képernyő tárolójába (*FrameBuffer*) íródik egy képpont színe, előtte ez a funkció megnézi, hogy az adott ponton lévő Depthbuffer érték kisebb-e mint a képpont távolsága a kamerától. Így eldönthető, hogy az újabb pont takarja-e az előzőt.

Megjegyzés: A Depth Test paraméterinél lehetőség van rá, hogy ne a kisebb, hanem a nagyobb érték esetén írja felül a képpont színét, így mindig a legtávolabbik elem látszik, ezzel különleges grafikai hatásokat elérve (pl. robbanó motor belső felépítésnek szemléltetése, stb..).

A *Blend* azt állítja be, hogy a már meglévő képpont (takart) színe, az új képpont (takaró) szín függvényében hogyan módosuljon. Jelen esetben ez egyből felülíródik az új képpont színével, tehát teljesen eltakarja.

Megjegyzés: Ezzel az eszközzel van lehetőség arra, hogy átlátszó objektumokat is megjelenítsünk (*Alpha Blending*). Jelen esetben nincsenek átlátszó elemeik, így ezt a funkciót nem használjuk fel.

A *renderInit()*, valamint *glPost()* függvények a transzformációkkal állnak kapcsolatba. A *renderInit()* minden képernyőfrissítés elején alaphelyzetbe állítja a transzformációs beállításokat, a *glPost()* pedig az éppen aktuális transzformációs beállításokat tölti a grafikus kártyára. Mivel általában minden elem máshogyan helyezkedik el a grafikus világtérben, ezért azok kirajzolása előtt a transzformációs beállítások is megváltoznak, így minden háromszögháló rajzolása előtt meg kell hívni a *glPost()* függvényt (ez a *glObject* feladata). A transzformációkról, valamint ezek grafikus egységre történő feltöltéséről egy későbbi fejezetben esik szó.

5.2.4 Hibaellenőrzés

Mint azt már fentebb említettem, az egyes OpenGL funkciók után hibaellenőrzés szükséges. Ezekhez a hibaellenőrzésekhez az *error* könyvtárban lévő *error.cpp* valamint *error.h*-ban található hibakezelő függvények használatosak. Az OpenGL tartalmaz beépített függvényeket, melyekkel az esetleges belső hibaüzenetek, valamint fordítási, linkelési, csatolási hibák elérhetőek. Ezeket használják fel az *error.cpp* függvényei a hibakeresésre, és az esetleges hibákat, a már említett *log* segítségével az alkalmazás eseménynaplójába írják.

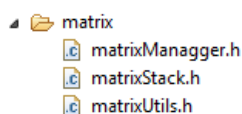
Az *error.cpp* alapvetően három típusú hiba detektálásához nyújt funkciókat. A *testGLError()*, az általános belső OpenGL hibákat írja az eseménynaplóba, például hívási verem betelte, vagy olyan funkció hívása, mely az adott állapotban tilos. Ez egyszerűen a *glGetError()* függvény-t hívja addig, míg az 0-át nem ad vissza. A visszaadott szám a hiba kódja [28].

A másik két függvény a programfordítási és csatolási hibákat ellenőrzi. A *testShaderCompileError(...)* az árnyaló programok fordításával kapcsolatos hibákat jelzi. A *testProgramLinkError(...)* pedig az árnyaló programok OpenGL programhoz való csatolásakor felléphető hibákat. Ezek elsőként a *glGetShaderiv(...)* illetve a *glGetProgramiv(...)* függvények segítségével eldöntik, hogy sikerült-e a fordítás, és

csatolás. A *glGetShaderInfoLog(...)*, valamint a *glGetProgramInfoLog(...)* segítségével érik el a konkrét hibaüzeneteket, majd kiírják az eseménynaplóba.

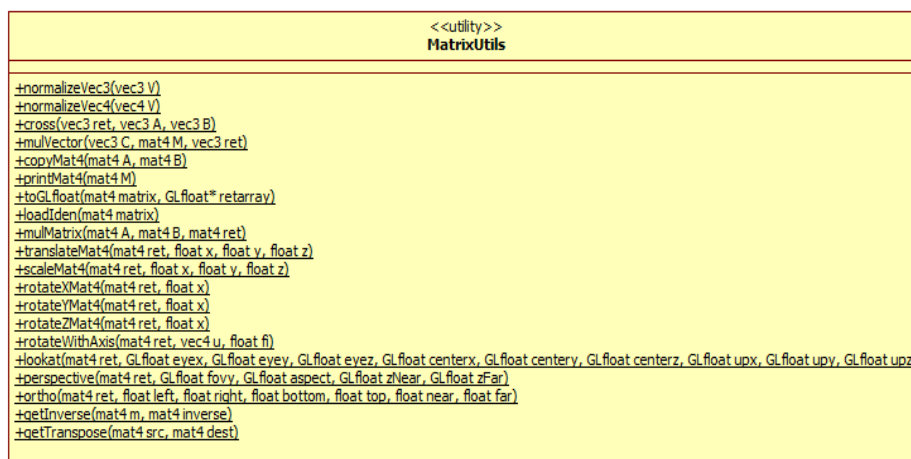
5.3 Mátrixok és vektorok

A Mátrix és Vektor műveleteket megvalósító funkciók a *matrix* könyvtárban találhatóak (5.6. ábra). Az OpenGL asztali változatával ellentétben, az OpenGL ES 2.0 natív környezetben nem nyújt támogatást mátrixok és vektorok kezelésére. Bár grafikus a hardveren futó árnyaló programok esetében a GLSL erre beépítve tartalmaz funkciókat, a kirajzolás előtti (központi feldolgozó egységen futó) kódrészek esetében nincs ilyen támogatás. Ezért a különböző vektor és mátrix funkciókat meg kellett valósítanom, mely során az asztali változat logikáját, felhasználási módját követtem.



5.6. ábra: A Mátrix alkomponens felépítése

5.3.1 Kiegészítő funkciók



5.7. ábra: A MatrixUtils osztály (UML Osztálydiagram)

Elsőnek magát a mátrix, valamint vektor fogalmát kellett definiálni. Minden vektort descartes koordinátákban definiált. Ez háromdimenziós esetben három darab lebegőpontos szám. Ahhoz, hogy egy háromdimenziós vektort transzformálni (mozgatni, forgatni, skálázni... stb.) lehessen a 3D térben 4x4- es mátrixokra van szükség, mivel az elmozgatást csak így lehet reprezentálni. Ehhez minden vektort ki kell egészíteni egy 4. lebegőpontos számmal melynek értéke mindig 1. Viszont kényelmi célból a 3 elemű vektor is használható, így beszélhetünk *vec3*, *vec4*, valamint *mat4* típusokról, melyek rendszerint 3,4 és 16 elemű lebegőpontos számtömbök. A

transzformációs mátrixok összeszorzásával a transzformációk összevonhatóak, az összevonási sorrend mindig a balról-jobbra irány a mátrixok szorzásánál.

Mivel az asztali környezet logikáját igyekeztem követni, ezért nem külön vektor és mátrix osztályokat hoztam létre, hanem a kiegészítő funkciók mind a *matrixUtils.h*-ban található *MatrixUtils* osztály statikus függvényei (5.7. ábra).

5.3.2 Vektor funkciók

A *mulVector(...)* vektort szoroz mátrixszal jobbról. A *normalizeVec[3|4](...)* pedig normalizálja (egység-hosszú alakra hozza) a kapott vektort. A *cross(...)* függvény két vektor keresztszorzatát (vektoriális szorzatát) állítja elő.

5.3.3 Alap mátrix funkciók

Az alap mátrix funkciók körébe tartozik a *copyMat4(...)*, illetve *printMat4(...)* függvények, melyek 4x4-es mátrixok másolását, illetve eseménynaplóba írását teszik lehetővé. A *toGLfloat(...)* függvény a lebegőpontos mátrixokat az OpenGL belső formátumára konvertálja. A *loadIden(...)* identitás(egység) mátrixot tölt a paraméterül kapott mátrixba. A *mulMatrix(...)* mátrixok szorzását teszi lehetővé, a *getInverse(...)* valamint *getTranspose(...)* pedig a paraméterül kapott mátrix inverzét illetve transzponáltját állítja elő. Ezek megvalósítása a megszokott lineáris algebrai módszereket követik, az inverz számolásánál az Adjungált-mátrix/Determináns módszert [30], a determináns számolásánál pedig a Kifejtési tételt [31] használja fel.

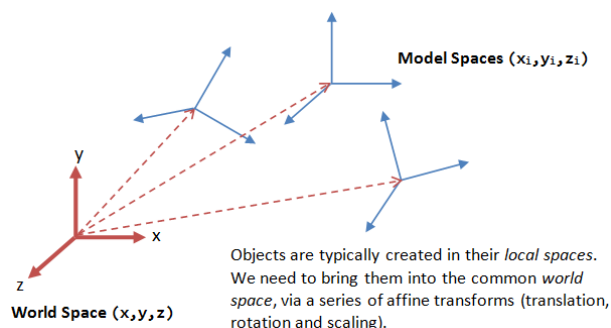
5.3.4 Transzformációs funkciók

A transzformációs mátrix függvények mindegyike valamilyen transzformáció mátrixával szorozzák meg a paraméterül kapott mátrixot jobbról. A *translateMat4(...)* az elmozgatási mátrixszal, *scaleMat(...)* pedig a skálázási mátrixszal szoroz, az x,y,z paraméterek függvényében. A *rotate[X|Y|Z]Mat4(..)* pedig a X|Y|Z tengelyek közül való forgatás mátrixával. A *rotateWithAxis(...)* pedig egy paraméterül kapott vektor által megadott tengely körüli forgatás mátrixával szoroz.

5.3.5 Nézeti mátrixok

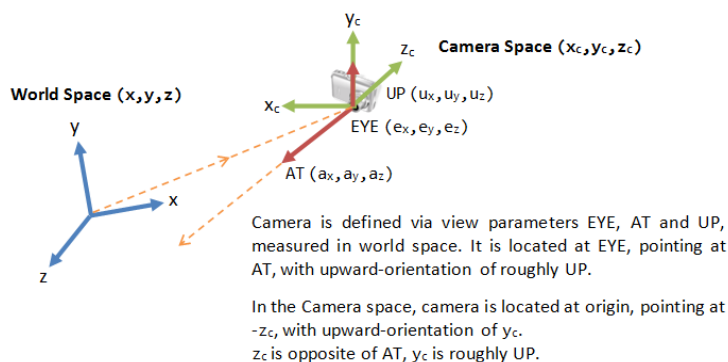
A háromdimenziós térben négy kitüntetett teret különböztetünk meg. A Modelltér (*Modell Space*), melynek középpontja az adott háromszögháló középpontja. A Világtér (*World Space*) a háromdimenziós világ tere, melynek középpontja az origó,

minden elem ebben a térben kerül elhelyezésre. A Kameratér (*Eye Space*) melynek középpontja a kamera középpontja, és a kamera nézetének megfelelően van transzformálva. A Vetített tér (*Projective Space*) pedig az a tér mely a kamerán keresztül a képernyőre vetül.



5.8. ábra: A Világtér és Modelltér kapcsolata [32]

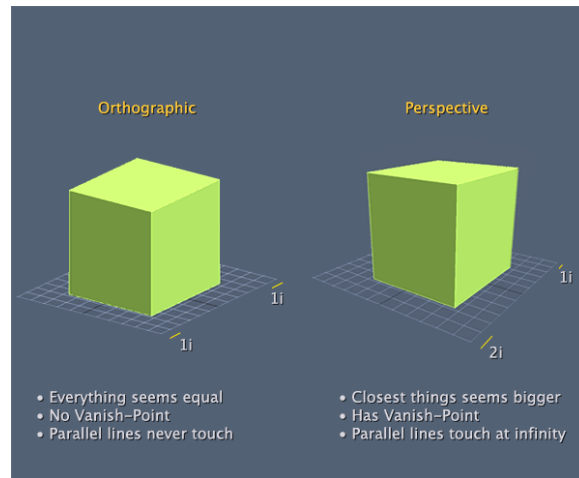
A háromszöghálót elsőnek a modelltérből a világtérbe transzformálni. Ez általában csak a modell pozíciójába való eltolást, vagy esetleg forgatást tartalmaz, ez a *Modell* transzformáció, mely minden elemre más, és más (5.8. ábra). Ahhoz hogy az eszköz kijelzőjén a Világtér vetülete jelenjen meg, még két transzformáció szükséges. Elsőnek a Világtérből a Kameratérbe kell transzformálni ez a *View* transzformáció (5.9. ábra). Majd a Kameratérből a Vetített térbe kell vetíteni, ez a *Projection* transzformáció. A három transzformáció segítségével minden elem helyzete megadható a Vetített térben.



5.9. ábra: A Világtér és a Kameratér kapcsolata [32]

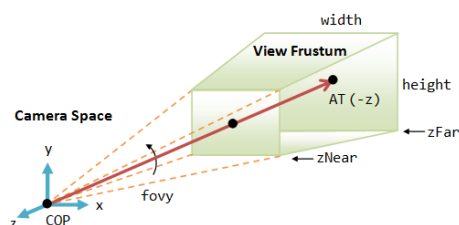
A *lookat(...)* funkció a Világtérből a Kameratérbe transzformáló mátrixszal szoroz. Ennek paraméterei a szem (*eyex*, *eyey*, *eyez*) helyzete, a nézeti irány (*looktx*, *lookty*, *looktz*), valamint a kameratérbeli felfelé irány (*upx*, *upy*, *upz*). Ezen adatok

mindegyike a világtérben értendő, és ezek segítségével előállítja a Kameratér bázisvektorait, majd a világtérből ebbe a térbe transzformál.



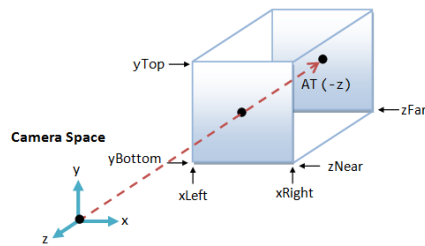
5.10. ábra: A Perspektívus és Ortografikus vetítés [35]

A *perspective()* a perspektív projekció mátrixával, az *ortho(..)* pedig az ortografikus projekció mátrixával szoroz. A perspektív projekció során úgy vetítünk, hogy a perspektíva szabályainak megfelelően a párhuzamos vonalak egy pontba tartanak, és ami távolabb van, nagyobbak látszik (5.10. ábra). A *perspective()* esetében a paraméterek a kamera látószöge, a magasság és szélesség aránya, valamint a közeli és távoli vágósík (5.11. ábra).



5.11. ábra: A Perspektívus vetítés paraméterei [32]

Az ortografikus projekció során nincs perspektíva, hanem a párhuzamos, vonalak párhuzamosak maradnak (5.10. ábra). Az *ortho()* esetében, a bal, jobb, fent, lent vektorok, valamint itt is a két vágósík a paraméterek (5.12. ábra).

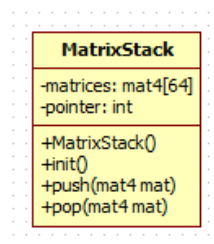


5.12. ábra: Az Ortografikus vetítés paramétereit [32]

A vágósíkok azt jelentik, hogy azok az elemek melyek a közeli vágósíktól közelebb vannak, illetve, melyek a távoli vágósíktól távolabb vannak, nem jelennek meg a vetített képen, ezzel kiszűrve a túl közeli és túl távoli elemeket. A már említett mélység tesztnél (*DepthTest*), a mélység értékek e síkok távolsága között mozoghatnak. (Éppen ezért a nem megfelelő vágó-síkválasztás mélységteszt hibákat, úgynevezett *Z-fight*-okat eredményezhet). A projekciós mátrixok számítása az asztali változattal megegyező módon történnek [33], [34].

5.3.6 A mátrix verem

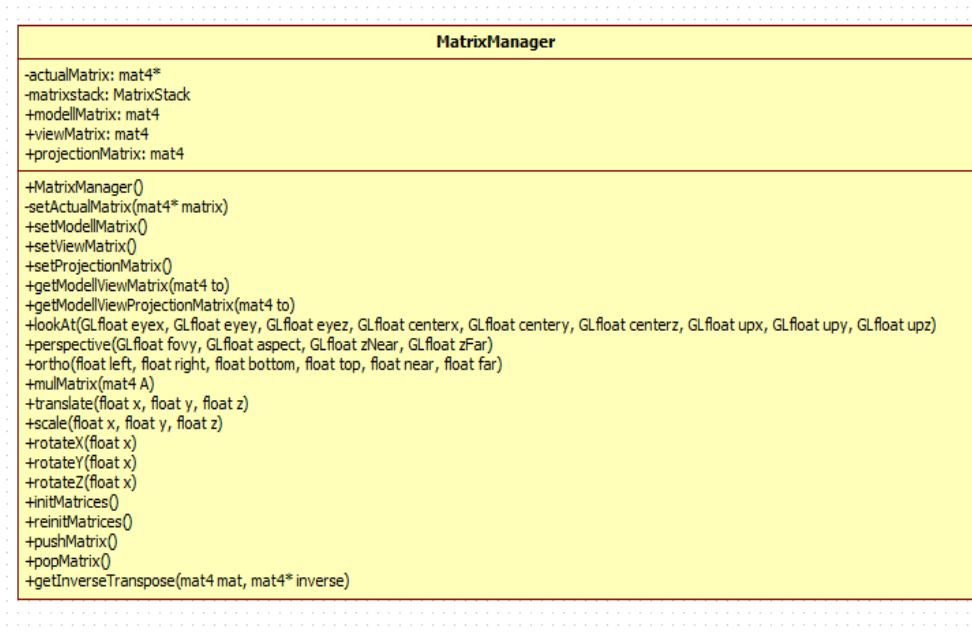
Az OpenGL asztali változatában lehetőség van arra, hogy bizonyos pontokon elmentsük a transzformációs mátrixok állapotát, majd visszatérjük oda. Ez akkor nagyon jó, akkor, amikor több transzformációt egyidejűleg használunk, és egy korábbi transzformált állapotba szeretnénk visszalépni. Például az egy autó kerekének kirajzolása után már nem a kerek koordinátarendszerébe szeretnénk transzformálni, hanem visszalépni az autó modell terébe. Az asztali verzióban ez veremszerűen működött, tehát mindig a legutoljára kimentett állapotba tértünk vissza.



5.13. ábra: A MatrixStack osztály (UML Osztálydiagram)

Azért, hogy ez a funkció mobil környezetben is használható legyen, a *MatrixStack* osztályt hoztam létre (*matrixStack.h*) (5.13. ábra), mely egy verem, mely 64 mátrix tárolását teszi lehetővé, és a c++ programozási technikákból már megszokott, *push(..)*, illetve *pop(..)* függvényekkel lehet elemeket betenni illetve kivenni a veremből. Az *init()* függvény kiüríti a vermet.

5.3.7 A mátrix állapotgép



5.14. ábra: A Matrix Manager osztály (UML Osztálydiagram)

Az asztali verzió felhasználói módszereit követve, a mátrixok kezelését állapotgéppel oldottam meg. A *MatrixManager* osztály (*matrixManager.h*) hivatott ezt az állapotgépet megvalósítani (5.14. ábra).

Az éppen aktív mátrix a *set[Modell|View|Projection]Matrix()* függvénnyel választható ki. (A *setActualMatrix()*-al más mátrix is beállítható, de ezt csak az osztályon belül használható). A *getModellMatrix(...)*, *getModellViewMatrix(...)*, és a *getModellViewProjectionMatrix(...)* a transzformációs mátrixok éppen aktuális állapotának szorzatait adják meg. A mátrixok egység mátrixszá alakításához, valamint verem törléshez használatos az *initMatrices()*, illetve a *reInitMatrices()*. A *reInitMatrices()* esetében a verem nem törlődik.

A *translate(...)*, *scale(...)*, *rotate[X|Y|Z](...)*, *lookat(...)*, *ortho(...)*, és *perspective(...)* mindegyike a megadott paraméterek szerint előállított mátrixszal szorozza meg az éppen aktuálisan kiválasztott mátrixot jobbról. A *pushMatrix()* az éppen aktuális mátrix értékét teszi a mátrix verem tetejére, a *popMatrix()* pedig a verem tetején lévő mátrixot másolja az aktuális mátrixba.

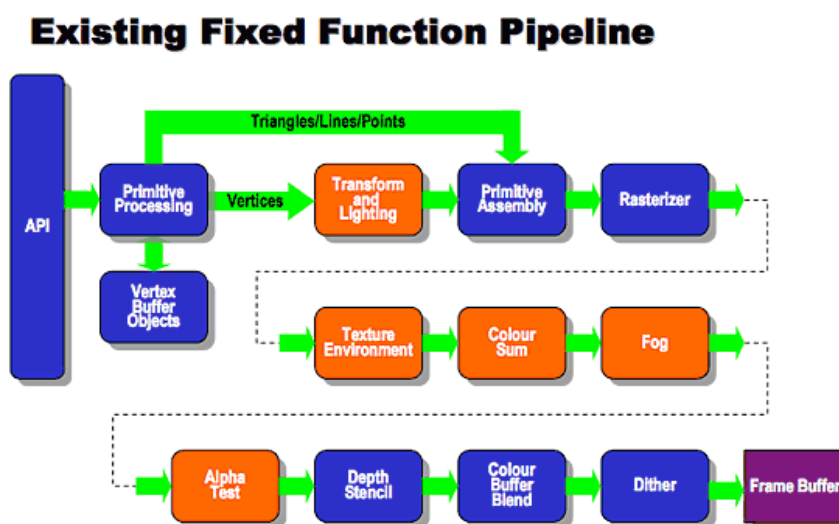
A *getInverseTranspose(...)* egy mátrix inverz-transzponáltját adja meg. Ez nem az állapotgép működéséhez kapcsolódik, csak a *MatrixUtils* inverz, és transzponált funkcióinak kivezetése.

5.4 Az árnyalás

Az OpenGL ES 2.0 egyik legfontosabb újítása elődjéhez az 1.1-es verzióhoz képpent, hogy programozhatóvá teszi a grafikus csővezetékét. Az OpenGL grafikus megjelenítő rendszere alapvetően egy csővezeték (*pipeline*). A vezeték egyes elemei adatokat kapnak, azokat átalakítják valamilyen módon, végül továbbadják a csővezeték soron következő elemének. Minden háromszögháló végigfut a csővezeték minden elemén a kirajzolás során.

A *pipeline* minden elemének programja a grafikus hardveregységen fut. A régebbi OpenGL ES verziókban, a kirajzolás előtt a csővezeték állomásainak kódja előre be volt drótozva, a fejlesztőnek mindössze bizonyos elemek paraméterezésére volt lehetősége. A kirajzolni kívánt háromszögeket átadták a csővezeték első elemének, a folyamat végén az utolsó elem pedig előállította a háromszögháló megjelenítendő képét. Az így kapott képek összessége adta ki a végső képet mely a felhasználó számára megjelent. A 2.0-ás verzióban megjelenő programozható csővezetéknél már lehetőségünk van rá, hogy a vezeték egyes elemeinek programját mi írjuk meg.

5.4.1 Az OpenGL ES 1.1 csővezetéke

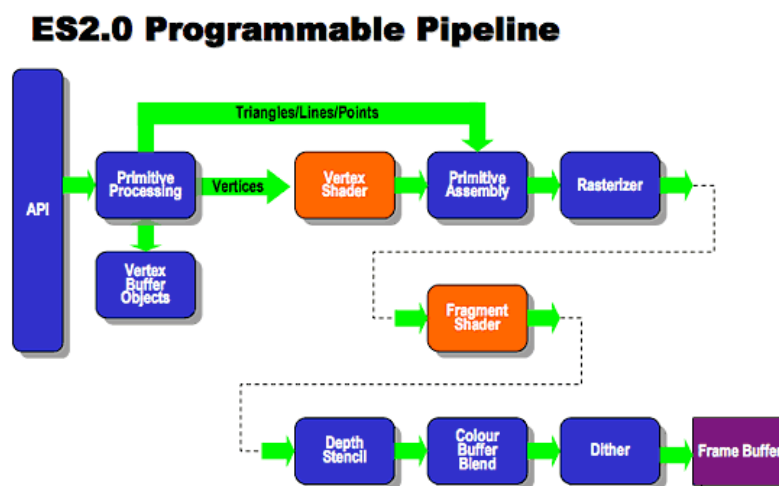


5.15. ábra: Az OpenGL ES 1.1 csővezetéke [5]

Az 5.15. ábra narancssárga színnel jelölt elemeit van lehetőség a rajzolás előtt paraméterezni, a kék színnel jelölt elemeken viszont csak alapbeállításokat lehet végezni, vagy még azokat sem nem. A zöld nyilak jelölik a csővezeték adatainak folyásirányát. A csővezeték végén található *Frame Buffer* a képernyő képpontjainak

tárolója. Az ábráról leolvasható, hogy a transzformációk és fények még háromszög (*Vertex*) szinten történtek feldolgozásra, és csak az ez utáni egységek dolgoznak képpont adatokkal.

5.4.2 Az OpenGL ES 2.0 csővezetéke

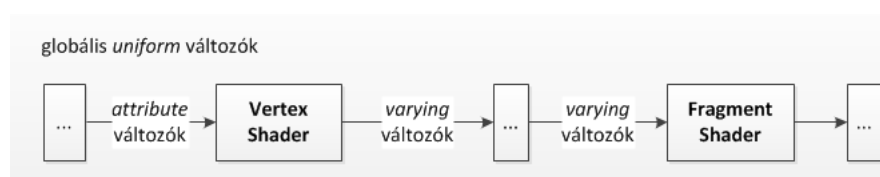


5.16. ábra: Az OpenGL ES 2.0 csővezetéke [5]

Hasonlóan az előző ábrához, az 5.16. ábra a kék színnel jelölt csővezeték elemei csak korlátozottan állíthatóak, viszont a narancssárgával jelölt elemeket a programozó teljes egészében maga valósíthatja meg úgynevezett árnyaló nyelvek segítségével. Jelen esetben az árnyaló nyelv *GLSL ES (OpenGL ES Shading Language)* nyelv [29]. Az ilyen nyelven írt *Vertex Shader* és *Fragment Shader* programok fordításáról, és csatolásáról már volt szó az előző fejezetekben. Az árnyaló programok a bemenet minden egyes elemére külön lefutnak. Az előző ábra narancssárga elemeit hivatottak felváltani ezen ábra narancssárga elemei.

5.4.3 Háromszög árnyaló (*Vertex Shader*)

A *Vertex Shader*, mint nevében is benne van, háromszögekkel dolgozik. Bementként is háromszögeket kap és kimenetként is háromszögetek ad ki.



5.17. ábra: Shader változó típusok

A GLSL nyelv három változótípus definiálást teszi lehetővé, ahogy az 5.17. ábra szemlélteti:

- Az *uniform* változók értéke nem változik meg a csővezeték futása alatt, tehát háromszöghálónként konstansok, ilyenek például a transzformációs beállítások, vagy a szem helyzete, melyek egy háromszögháló kirajzolása alatt végig azonosak. Általában az egész háromszöghálóra kiterjedő beállításokat tartalmaznak.
- Vannak úgynevezett *attribute* változók, melyek a *Vertex Shader* bemenetei, és az adott változó értéke mindig egy elem a bementi elemek tömbjéből. A *Vertex Shader* program minden háromszög minden egyes pontjára lefut, és az *attribute* változók mindig egy háromszög pontjai, vagy azokhoz kapcsolódó értékek.
- A *varying* változók olyan értékeket tartalmaznak, melyek a *Vertex Shader* lefutása után átadásra kerülnek a *Fragment Shader*nek, ezek a változók hivatottak a két típusú *shader* közötti kommunikációra.

A *Vertex Shader*ben az olyan jellegű feladatokat kell megvalósítani melyekhez a háromszögek adatai szükségesek, viszont nincs szükség olyan adatokra, melyek képpontokra vonatkoznak. Itt kell elvégezni a különböző transzformációs számításokat, melyek során a világ koordinátarendszerében megadott háromszögeket, normálvektorokat és fény vektorokat a kamera szemszögébe transzformáljuk. A *Vertex Shader* után a csővezeték úgynevezett raszterizációt végez, ami a háromszögek képpontokká alakítását jelenti.

5.4.4 A képpont árnyaló (*Fragment Shader*)

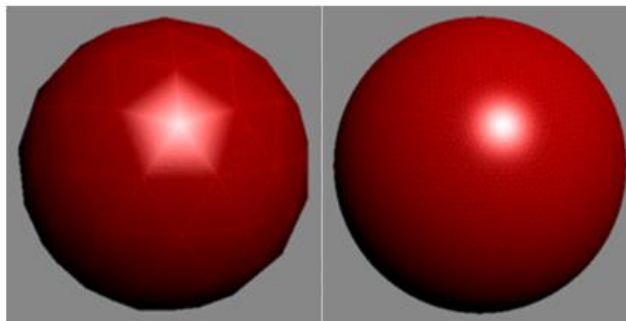
A képpont árnyaló is tartalmaz *uniform* változókat, ezek értékei megegyeznek a *Vertex Shader*-nél megadott értékekkel. A *varying* típusú változók a *Fragment Shader* bemenetei. A *Fragment Shader* mindig képpontokkal dolgozik, minden egyes képpontra külön fut le. Itt kell azokat a számításokat elvégezni, melyek képpontokra vonatkoznak, például a megvilágított tárgyak színe, illetve textúrázás. Ha a fények által kibocsátott intenzitást a *Vertex Shader*ben számoljuk, akkor háromszögenként történik az árnyalás (a régi OpenGL verziókban csak erre volt lehetőségünk), viszont ha a *Fragment Shader*ben, akkor képpont szinten történik, ezzel valóságosabb hatást keltve.

5.4.5 Az árnyalás matematikai modellje

Az árnyalás során a feladat az, hogy minden képpontra meghatározzuk azt a fény mennyiséget, ami a kamerába érkezik. A fényeknek mindig a 3 színsatorna (RGB) szerinti értékét határozzuk meg, 0 és 1 közötti lebegőpontos számként. Egy képpont színe attól függ, hogy milyen színt ver vissza az ott látható háromszög adott pontja. Egy háromszög minden csúcsához hozzá van rendelve egy normálvektor, mely a háromszögre eső fények törésének tükörirányát határozzák meg (5.22. ábra).

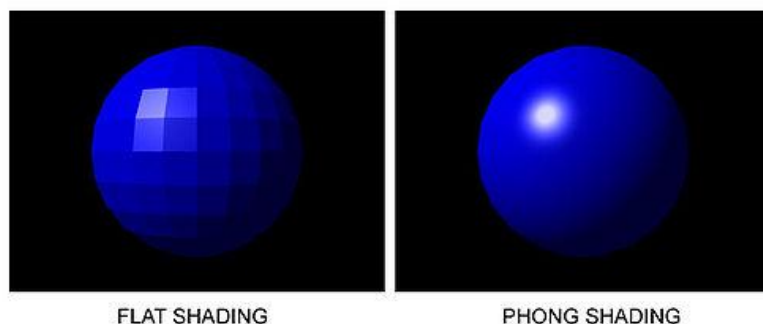
5.4.6 Interpolációs típusok

A megvilágítási értékek meghatározásakor kezdetben csak a háromszögek csúcspontjainál található értékek állnak rendelkezésre. Miután megtörténik a raszterizáció, már pontokkal dolgozhatunk. A régebbi OpenGL ES verziókban található fix csővezetékrendszernél a raszterizáció előtt történt a megvilágítás számolása, így a háromszög három pontjára történt az árnyalási értékek számítása, majd a raszterizáció során a megvilágítás értéke interpolálódott. Ez a *Gouraud árnyalás* [40], mely során a kevesebb háromszögből álló elemek a valóságtól nagyon eltérő képet mutattak, és a háromszögszám, illetve részletesség növelésével, a valóságérzet is nőtt (5.18. ábra).



5.18. ábra: A Gouraud árnyalás [40]

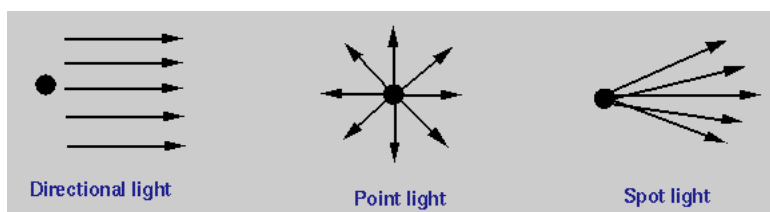
Az OpenGL ES 2.0-ban a normálvektorok értéke interpolálódik a háromszög sarkai között (*Phong árnyalás* [41]), így pixel szinten számolhatóak megvilágítási értékek, sokkal realisztikusabb hatást keltve a kisebb háromszög számú modellek árnyalásánál is (5.19. ábra).



5.19. ábra: A Phong árnyalás [41]

5.4.7 Fénytípusok

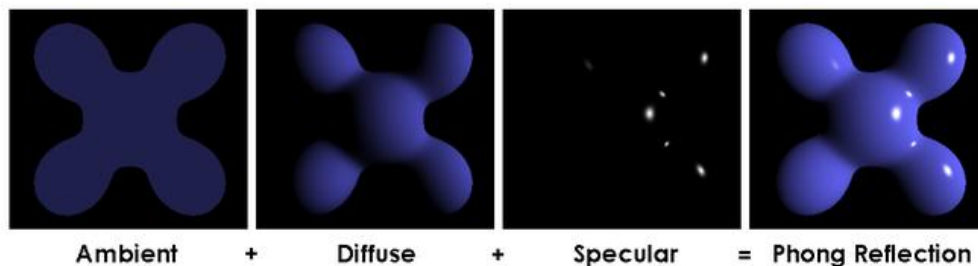
Alapvetően három féle fénytípust különböztetünk meg a háromdimenziós grafikában. Az *irányfényt* (például napfény), a *pontfényt* (például villanykörte), és a *spotlámpafényt* (1.1. ábra). Az elkészülő grafikus rendszer csak irányfényforrást tartalmaz, ezért a visszaverődések számításánál csak erre a fénytípusra vonatkozó részeket ismertetem.



5.20. ábra: Fénytípusok [42]

5.4.8 A fényvisszaverődések

A fényvisszaverődések számolásánál, a Blinn-Phong visszaverődési modellt [39] (mely a Phong visszaverődési modell [38] egy módosított változata) alkalmazom. Ez a modell 3 típusú visszaverődést definiál. Az ambiens (*Ambient*), diffúz (*Diffuse*) és spekuláris (*Specular*) (5.21. ábra).



5.21. ábra: A Phong árnyalás típusai [38]

Az ambiens fény számolásánál, az anyag színe a fény ambiens színével szorozódik. Így az ambiens fény a következő képlet szerint számolható:

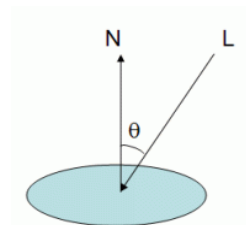
$$I_a = k_a * L_a$$

Ahol az ambiens fény (I_a) értéke az anyag ambiens színének (k_a), és az fény ambiens színének (L_a) szorzata.

A diffúz szín számolásánál, már szükség van az adott pont normálvektorára, valamint a fény helyzetére. Ha a fények pozíciójának megadása világtérben történik, akkor a normálvektort árnyalás előtt be kell transzformálni a világtérbe, mivel azok mindig modell térben kerülnek megadására (meg kell szorozni a Modell mátrix inverz transzponáltjával). Azután a diffúz fény visszaverődés értéke az alábbi képlet szerint számítható.

$$I_d = k_d * L_d * (L * N)$$

Ahol I_d a diffúz szín visszaverődés. K_d az anyag diffúz színe, L_d a fény diffúz színe, az L a fény vektor N pedig a normálvektor. Az L és N közötti szorzás vektorszorzás, tehát két vektor között bezárt szög koszinuszát adja meg (5.22. ábra).



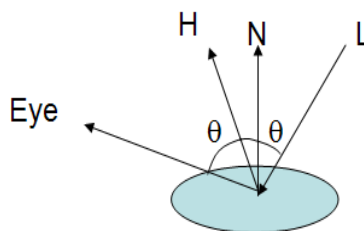
5.22. ábra: A diffúz fényvisszaverődés vektorai [37]

A spekuláris fény számolásánál ezeken felül szükség van még a kamera helyzetére is, mivel az anyag a spekuláris fényt más-más irányba más-más intenzitással veri vissza. A spekuláris fény az alábbi képlet szerint számolandó:

$$I_s = k_s * L_s * (N * H)^{Shins}$$

ahol

$$H = E - L$$



5.23. ábra: A spekuláris fényvisszaverődés vektorai [37]

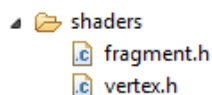
A I_s a spekuláris fényvisszaverődés értéke, a k_s az anyag spekuláris színe, az L_s a fény spekuláris színe, A *Shine* a fényességi faktor, H pedig a fél vektor (*Half vector*). A képletben az $N \cdot H$ szorzás vektorszorzás. Az L a fénypozíció vektora, az E pedig a kamera pozíciója az adott háromszöghöz képest (5.23. ábra).

A diffúz valamint spekuláris fényvisszaverődéseket minden fényforrásra ki kell számolni, így az árnyalás végső képlete:

$$I = I_a + \sum_{\text{Fényforrás}0}^{\text{Fényforrás}N} (I_d + I_s)$$

5.5 Az árnyalók megvalósítása

5.5.1 A háromszög árnyaló megvalósítása



5.24. ábra: Az *shaders* alkomponens

A Grafikus komponens egy háromszög, és egy képpont árnyalót használ. Ezek a *shaders* mappán belül találhatóak (1.1. ábra). A háromszög árnyaló a *vertex.h* fájlban található meg. Azért hogy az alkalmazás ne tudja fordítani az árnyaló programokat, azok karakterfüzérként vannak tárolva, melyekben a sortörések, és egyéb nem látható karakterek is jelölve vannak. A programok ismertetése során ezeket a kiegészítő karaktereket nem jelölöm. Az árnyaló programok bemeneteinek kiszámolását, valamint feltöltését más komponensek végzik, melyeket későbbi fejezetekben mutatok be.

A *Vertex Shader* elején a fentebb említett 3 típusú változók deklarációja történik.

```
uniform vec3 colorVec;
```

```

uniform vec3 eyeVec;
uniform int isShaded;
uniform mat4 modellMat;
uniform mat4 modellMatIT;
uniform mat4 modellViewMat;
uniform mat4 modellViewMatIT;
uniform mat4 modellViewProjectionMat;
uniform mat4 modellViewProjectionMatIT;

attribute vec4 vertexVec;
attribute vec3 normalVec;
attribute vec2 textureVec;

varying vec3 normalPos;
varying vec3 colorVecA;
varying vec3 lightPosVecA;
varying vec3 eyeDir;
varying vec2 textureVecA;
const vec3 lightPosVec=vec3(1.0,1.0,-1.0);

```

A *colorVec* egy színvektor, az anyag ambiens és diffúz színe. Az *eyeVec* a kamera pozíciója világtérben megadva. Az *isShaded* azt jelzi, hogy árnyalva legyen-e az adott elem. A mátrixok a már ismertetett transzformációs mátrixok szorzatai, valamint azok inverz transzponáltjai, hogy vissza is lehessen transzformálni vektorokat. A *vertexVec* a háromszög egy pontjának koordinátája, a *normalVec* az a ponthoz tartozó normálvektor, a *textureVec* pedig a textúra vektor. A *colorVecA* az *eyeDir* valamint a *textureVecA* a *FragmentShader*-nek továbbítják a már említett értékeket. A *normalPos* a normálvektor világtérbe transzformált változata. A *lightPosVecA* a fény pozíciója a világtérben. A *gl_Position* egy előredefiniált változó a pont végső helyzetének, ennek értéke a Modell térben megadott pont Vetített térbe transzformált változata. Ezen megvalósításban csak 1 fény van definiálva, és az is fix értékkel (1, 1, -1 irány), de ez is elegendő az alapszintű árnyaláshoz.

```

void main()
{
colorVecA=colorVec;
lightPosVec=vec3(vec4(normalize(lightPosVec),0.0));
    vec3 vertexW=vec3(modellMat*vertexVec);
    eyeDir=vec3(vec4(eye-vertexW,0.0));
    normalPos=vec3(modellMatIT*vec4(normalVec,0.0));

    vec3 v=-1.0*lightPosVecA;"
    textureVecA=vec2(textureVec);
    gl_Position = modellViewProjectionMat*vertexVec;
}

```

5.5.2 A képpont árnyaló megvalósítása

A képpont árnyalót a *shaders* mappán belül található *fragment.h* fájl tartalmazza. Itt történik meg a képpontok színének meghatározása, valamint a textúrázás

is. Elsőnek a precizitás kerül beállításra, magas precizitású lebegőpontos értéként. Majd a különböző *Vertex Shader*-től kapott *varying* változók deklarálása. Egy *sampler2D* változó is deklarálásra kerül, mely a textúra képpontjait tartalmazza.

```
precision highp float;

varying vec3 eyeDir;
varying vec3 normalPos;
varying vec3 colorVecA;
varying vec3 lightPosVecA;
varying vec2 textureVecA;
varying vec3 lightReflVec;

uniform sampler2D texture;
uniform vec3 isShaded;
```

A fény diffúz színe valamint az anyag spekuláris színe is konstans fehéreként van deklarálva, valamint a fényesség értéke 150.

```
const vec3 material=vec3(0.0,0.0,1.0);
const vec3 lightColor=vec3(1.0,1.0,1.0);
const vec3 shineColor=vec3(1.0,1.0,1.0);
const float shininess=150.0;
```

A program során a már ismertetett árnyalási egyenleteket használom, azzal a különbséggel, hogy egy ambiens faktorral szorzom az ambiens fényvisszaverődést, valamint 1-ambiens faktorral a diffúz színt, így a sötét részek is jobban láthatóak. Elsőnek normalizálom a vektorokat, majd kiszámolom a szükséges vektoriális szorzatokat (*dot(..)* függvény). Aztán az anyag alap színét határozom meg, ez az anyag diffúz színe összeadva a textúra pont színével. Ezután az ambiens, diffúz valamint spekuláris visszaverődési értékek kerülnek meghatározásra. Majd ellenőrzöm, hogy kell-e árnyalni az anyagot (*isShaded* nagyobb-e mint 0.5), ha kell árnyalni, akkor a visszaverődő fények összege lesz a képpont színe, ha nem kell, akkor pedig az anyag alap színe. Végül ezt a színt adom a *gl_FragColor* előredefiniált változónak mely a képpont színét jelzi.

```
void main()
{
    vec3 normal=normalize(normalPos);
    vec3 lightPos=normalize(lightPosVecA);
    float n_dot_l=max(dot(normal,lightPos),0.0);

    vec3 halfVec=normalize(eyeDir-(-1.0*lightPos));
    float n_dot_h=max(dot(normal,halfVec),0.0);
    vec3 specular = pow(n_dot_h, shininess) * shineColor * lightColor;

    vec3 color=(colorVecA+vec3(texture2D(texture,textureVecA)));
    vec3 ambient= color*ambient_factor;
    vec3 diffuse = color*(n_dot_l)*(1.0-ambient_factor);
```

```
vec3 unshaded = color;
vec3 shadedColor=vec3(1.0,1.0,1.0);

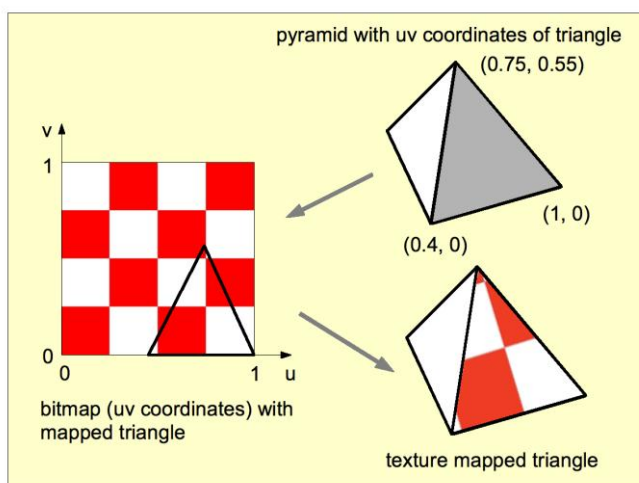
if( isShaded.x < 0.5)
shadedColor = unshaded;
else
shadedColor = ambient+diffuse+specular;

gl_FragColor = vec4(shadedColor,1.0);
}
```

5.6 Textúrázás

5.6.1 Textúra térképezés

Ahhoz, hogy a megjelenített háromszöghálók még valóságosabbak legyenek, a háromszögekre egy 2D-s bittérképet feszítenek rá a modell készítői, ezt a technikát textúra térképezésnek, vagy UV térképezésnek nevezik [43]. Ez matematikailag azt jelenti, hogy a háromszög minden pontjához tartozik egy 2D vektor, vagy koordináta, mely azt adja meg, hogy az adott 3D-s pont a bittérképen hol helyezkedik el. A háromszög megjelenítésekor, ezen koordináták alapján, a háromszögre kell feszíteni ezt a képet (5.25. ábra).



5.25. ábra: A textúra térképezés [43]

A modell betöltése után betöltésre kerülnek az ahhoz tartozó bittérképek is, melyek az anyagok jellemzői közt találhatóak meg. A textúra térkép gyakorlatilag az anyag (ambiens, diffúz, spekuláris) színének a térképe, így az anyag színe akár képpontról képpontra is változhat.

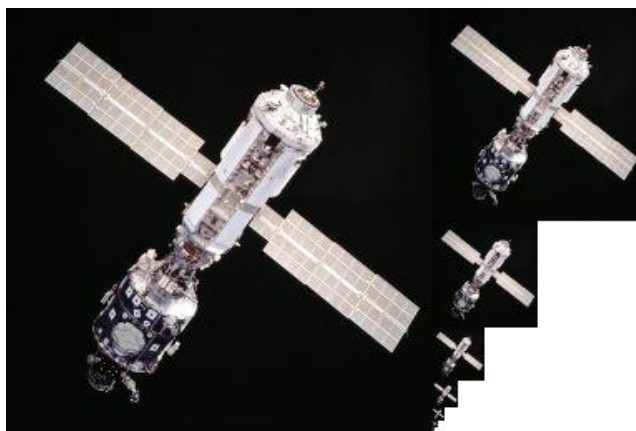
Megjegyzés: Léteznek úgynevezett 3D textúrák is, melyek mélység információkat is tárolnak, ezeket a modell részletességének javításához használják. Ilyen technika például a BumbMapping vagy Displacement Mapping [43]. Mivel ezeket a hatásokat nem valósítottam meg az alkalmazás fejlesztése során, így ezek nem kerülnek bemutatásra.

5.6.2 Textúra szűrés

Mivel csak a háromszögek koordinátája van megadva a textúra térképen, ezért a közöttük lévő pontokra meg kell határozni a textúra képpontok (*texel*) értékét, ez a textúra szűrés. Itt a feladat az, hogy a *texel* értékeket pixel értékekké képezzük le oly módon, hogy az az emberi szem számára a legvalóságghűbben hasson. Ezek a folyamatok elég számítás igényesek, így fontos a megfelelő textúra szűrési módszer megválasztása.

A manapság elterjedt szűrési módszerek asztali környezetben a *bilinear* (kettős-lineáris), *trilinear* (hármass-lineáris) és az *anisotropic* (nem izotropikus). A *bilinear* szűrésnél [45] a vetített kép hosszában és magasságában is 2-2 *texel*-ben történik az átlagolás (1 pixel értéke az oda eső 4 *texel* átlaga).

A *trilinear* szűrés [46] a *bilinear* javított változata, ahol vetítési mélység irányában is átlagolás történik, azaz a távolabb lévő pontok elmosódottabbak. Ezt úgy érik el, hogy *bilinear* szűrést alkalmaznak, de minél távolabb van a pont a kamerától attól függően egy kisebb felbontású textúrát használnak.



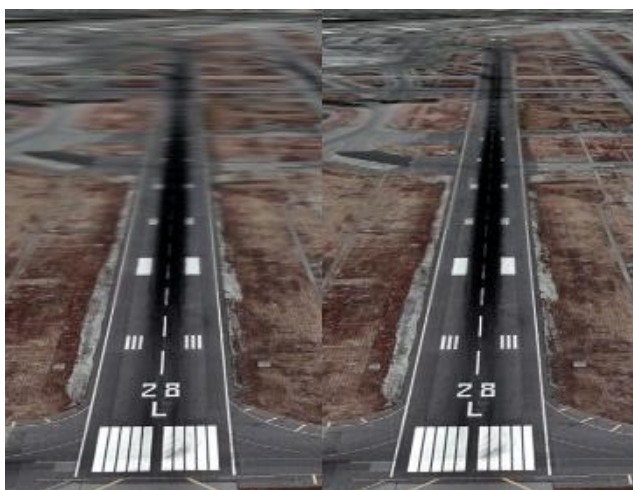
5.26. ábra: A *Mipmapping* technika által készített kép [46]

Ennek a technikának a kiegészítése a *Mipmapping* [46], ahol a textúrák betöltésekor minden textúrához generálódik több kisebb felbontású változat is (5.26. ábra). A textúra szűrésnél a távolabb lévő pixelekhez a kisebb felbontású képeket használja.



5.27. ábra: A *Ripmapping* technika által létrehozott kép [47]

Létezik még egy fejlettebb szűrési módszer is, az *anisotropic* szűrés [47], ami a *trilinear* szűrés továbbfejlesztése (5.28. ábra). A *trilinear* szűrésnél a *Mipmap* szintek négyzetesen generálódnak, így a távolabbi pontokban magasság és szélesség koordináták mentén is történik átlagolás a textúráknál, emiatt sok esetben, magasság irányban is átlagolódik a kép, holott csak szélességben kellene (vagy fordítva). Ezt úgy oldja meg az *anisotropic* textúra szűrő, hogy vagy több *Mipmap* szintet használ egy pont számolására, vagy nem csak négyzetes *Mipmap* szinteket generál (*Ripmap* [47]) (5.27. ábra). Ez a szűrési módszer nagyon számításigényes, és az OpenGL ES 2.0-ban még nem támogatott.



5.28. ábra: A Trilinear (bal), és az Anisotropic (jobb) szűrés közötti látványos különbség

5.6.3 Megvalósítás - Létrehozás

A Dániel által fejlesztett komponenstől (*loader*) az agyaghoz csatolt textúra azonosítója (*GLuint*) érhető el. Ez egy textúrát definiál, a textúra kép maga már a

grafikus kártyára töltődött a modell beolvasásakor. Ezen beolvasás során történik a textúrák létrehozása a bittérképből, valamint azok szűrési beállítása (*loader/converter.h* - *loadFrom[Obj]3DS(..)*). Ezt a rész közösen fejlesztettük Dániellel, a textúra létrehozási, valamint szűrési beállítások implementálása az én feladatomból volt.

Az OpenGL lehetővé teszi egyszerre több textúra használatát (*multitexturing*)[43], viszont jelen esetben egy anyaghoz csak egy textúra tartozik, így csak az 0. textúra-szintet kell bekapcsolni.

```
glActiveTexture (GL_TEXTURE0);
```

Minden egyes anyaghoz mely textúrával rendelkezik, elsőként létre kell hozni egy textúra azonosítót, mellyel később majd a textúrára hivatkozni lehet, ezen azonosítót egy *GLuint* tömb tárolja (*texture_id*), melybe annyi számú azonosítót generálunk ahány anyag van.

```
glGenTextures(materialFile.materialCntr, texture_id);
```

Minden egyes textúrát először be kell kapcsolni, beállítani a hozzá tartozó bittérképet, majd legenerálni a különböző textúra szinteket (*Mipmapping*).

```
glBindTexture(GL_TEXTURE_2D, texture_id[j]);  
glTexImage2D(GL_TEXTURE_2D, 0,  
GL_RGB,materialFile.materials.at(j).width,  
materialFile.materials.at(j).height,0,GL_RGB,GL_UNSIGNED_BYTE,  
materialFile.materials.at(j).texture);  
glGenerateMipmap(GL_TEXTURE_2D);
```

Majd a *trilinear* textúra szűrési beállításai következnek az adott anyagra. (*További beállítási lehetőségek* [44])

```
glTexParameterf(GL_TEXTURE_2D,  
GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,GL_LINEAR);
```

Ezután a megfelelő textúra azonosító kötésével (mely az anyagjellemzők közt van tárolva) aktiválható az adott textúra az modellelem kirajzolása előtt.

5.6.4 Megvalósítás - Megjelenítés

A textúra megjelenítésének megvalósítása az árnyalás során történik, a *Fragment Shaderben*. Miután az anyaghoz tartozó textúra be lett kapcsolva, a grafikus csővezeték hívása előtt, az adott azonosítóhoz tartozó textúra értékeit a *Fragment Shaderben* el lehet érni. A *Fragment Shaderben* definiálva van egy *texture* nevű

uniform változó melynek *sampler2D* a típusa. Ez a textúra bittérkép értékeinek a tárolója.

```
uniform sampler2D texture;
```

Valamint a *Vertex Shaderből* kapott (de ott nem módosított) textúra vektor (UV vektor), mely *vec2* típusú mivel a 2D-s bittérkép egy pontját definiálja.

```
varying vec2 textureVecA;
```

A képpont színének számításánál az arra a képpontra eső textúra értéke úgy számolható ki, hogy a *texture2D(..)* GLSL függvény segítségével a *texture*-ből kiolvastva az adott textúra koordinátaához tartozó képpont színéhez hozzá kell adni az anyag alap színét (*colorVecA*). Ez az anyag árnyalatlan színé az adott képponton.

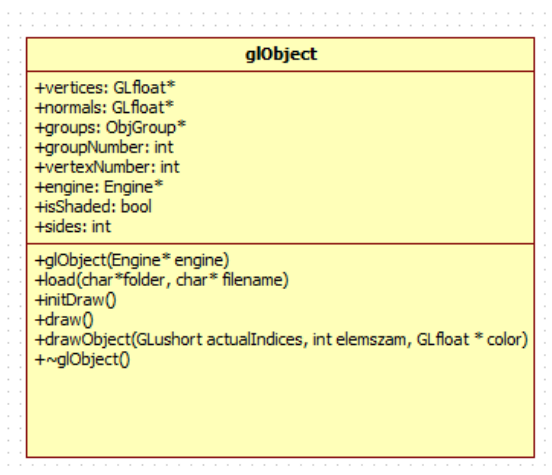
```
vec3 color=(colorVecA+vec3(texture2D(texture,textureVecA))); \n"
```

Az árnyalás során ezt a színt mint diffúz színt felhasználva árnyal az árnyaló program.

5.7 A grafikus objektum

5.7.1 Felépítése

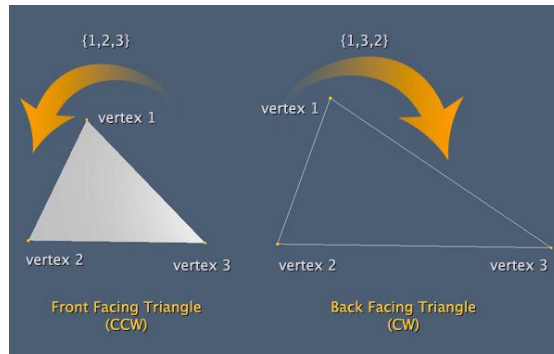
A grafikus objektum (*glObject*), a grafikus világban lévő modellek betöltésért, tárolásáért és megjelenítésért felelős osztály. A játék-alkalmazás minden olyan eleme, mely rendelkezik 3D-s megjelenéssel, tartalmaz egy ilyen objektumot. Nem azonos a játékal alkalmazás entitásával, mivel az entitás a grafikai megjelenésen kívül egyéb adatokat is tartalmaz.



5.29. ábra: A *glObject* osztály (UML Osztálydiagram)

A *glObject* a *gl* könyvtár gyökérkönyvtárában található, a *glObject.h* fájlban. Az osztály *vertices* és *normals* tagváltozói *GLfloat* típusú pointererek (5.29. ábra). Ezek mutatnak a Dániel komponense által lefoglalt adatokra. A *groups* egy *ObjGroup* pointer, ez a modell alcsoportjainak adatait tartalmazza. A *groupNumber* ezen alcsoportok száma, a *verticesNumber* pedig a háromszögek pontjainak száma.

Az *engine* egy mutató a grafikus motor objektumára, mely az grafikus objektum létrejöttékor kerül beállításra. Egy *glObject* objektum csak úgy jöhet létre, ha már létezik egy létrehozott *Engine* változó. (Nincs üres konstruktora a *glObject*-nek). Az *isShaded* változó azt állítja be, hogy egy elem legyen-e árnyalva, mivel vannak elemek, amiket nem szeretnénk árnyalni (pl. HUD elemek, égdoboz ...). A *sides* azt határozza meg, hogy egy elem hány oldalas megjelenítéssel kell kirajzolni, érte 1 és 2 lehet.



5.30. ábra: A hátsólap eldobás szemléltetése [49]

Azért hogy olyan elemeket melyek biztosan takarásban vannak, ne kelljen feleslegesen kirajzolni az OpenGL rendelkezik egy olyan opcióval, hogy a nekünk háttal lévő háromszögeket már a rajzolás előtt eldobja, ennek neve hátsólap-eldobás (*Backface Culling*) [49]. A modellezés során a háromszögekben a pontokat mindig meghatározott sorrend szerint adják meg a tervező eszközök. Ha háromszög velünk szembe néz, akkor a pontok felsorolási sorrendje óramutató járásával ellentétes. Ha egy háromszöget olyan helyzetben nézünk, hogy a pontjai óramutató járásával megegyezően vannak, akkor az a háromszög nekünk háttal van (5.30. ábra). Az valószínűleg takarásban lesz, mert elfedi, egy olyan háromszög mely előtte van. Mindazonáltal, léteznek olyan modellek, ahol láthatóak háttal fordított háromszögek is, még akkor is, ha az nincs takarásban. Ezért van a *sides* beállítás mely meghatározza, hogy hány oldalát kell az elemek kirajzolni. Ha ez az érték 1, akkor a fentebb ismertetett szabály szerint eldobásra kerülnek az adott háromszögek, ha 2, akkor nem kerülnek eldobásra. Az alapértelmezett érték 1.

5.7.2 A modellek betöltése

A modellek betöltése a *load(..)* függvény feladata, mely paraméterül egy mappa elérési útvonalat, valamint a modell fájl nevét kapja. A modell nevének végződése alapján megvizsgálja a file formátumát, és az ahhoz tartozó betöltő függvényt hívja meg a *converter* objektumban. Majd ha ez lefutott a *converter* objektumba betöltött értékeket másolja át a megfelelő tagváltozóiba.

```
void load(char* folder, char* filename)
{
    Converter c;

    if (strstr(filename, ".obj"))
    {
        c.convertFromObj(folder, filename);
    }
}
```

```

    }
    if (strstr(filename, ".3ds"))
    {
        c.convertFrom3ds(folder, filename);
    }

    this->vertices = c.vertices;
    this->normals = c.normals;
    this->textures = c.textures;
    this->groups = c.groups;
    this->groupNumber = c.groupNumber;
    this->vertexNumber = c.verticesNumber;
}

```

Ezután már a modelladatok a *glObject* objektumban tárolódnak.

5.7.3 Elemek feltöltése a grafikus hardverre

A már bemutatott *Vertex Shader* bemenetei a rajzolás előtt a grafikus hardverre kell tölteni. Ehhez először azonosítani kell (*GLuint* azonosítóval) a *Vertex Shader*-ben létrehozott bementi értéket majd erre az azonosítóra kell feltölteni az értékeket. A *Vertex Shader* bemeneti a *uniform* és az *attribute* változók.

5.7.4 Uniform változók feltöltése

Elsőként a *Vertex Shader*-ben használt változó azonosítójának lekérését kell elvégezni, itt meg kell adni az OpenGL Programot, valamint a változó nevét:

```
H_ModellMatIT = glGetUniformLocation(ProgramObj, "modellMatIT");
```

Majd a megfelelő típusú függvénnyel feltölteni az adott azonosítóval a grafikus hardverre a *GLfloat* tömböt, valamint az elemek darabszámát és hogy kell-e transzponálni a mátrixot.

```
glUniformMatrix4fv(H_ModellMat, 1, GL_FALSE, modellMatrixF);
```

Természetesen a más típusú uniform változókhoz más függvények szükségesek [50].

5.7.5 Attribute változók feltöltése

Itt is le kell kérdeni a *Vertex Shader*-beli változóhoz tartozó azonosítót.

```
GLuint textureVec = glGetAttribLocation(engine->ProgramObj, "textureVec");
```

Majd feltölteni az értékeket, ahol meg kell adni az azonosítót, az elemek méretét, típusát, azt hogy kell-e normalizálni a vektorokat, valamint az egyes háromszög elemek közötti távolságot *byte*-ban, és természetesen az értékeket tömbjét [51].

```
glVertexAttribPointer(vertexVec, 3, GL_FLOAT, GL_TRUE, 0, this->vertices);
```

Majd engedélyezi kell az attribútomtömböt.

```
glEnableVertexAttribArray(vertexVec);
```

5.7.6 A kirajzolás

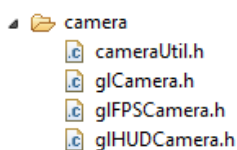
A kirajzolás folyamata három függvényből áll. Elsőnek az *draw()* függvény hívása után az *initDraw()* függvény fut le, mely a grafikus megjelentető hardverre tölti azokat az információkat, melyek az egész modellre vonatkoznak. Ezek a háromszögek pontjai, és az azokhoz tartozó normál, valamint textúra vektorok. Az alcsoportok kirajzolása során csak indexekkel hivatkozunk a most feltöltött *vertex*, normál vektor, textúra vektor hármaskra. Itt csak az *attribute* változók kerülnek feltöltésre, az uniform változók (transzformációs mátrixok) feltöltése az *Engine* feladata. Itt történik a *sides* értéktől függően a hátsólap-eldobás bekapcsolása is a *glEnable/Disable(GL_CULL_FACE)* függvénnyel.

Az *initDraw()* után a *draw()* függvény minden alcsoporton végighalad, beállítja a hozzájuk tartozó színt, textúrát, valamint a transzformációs beállítások, és azokat a grafikus kártyára tölti (*engine.glPost()*). Majd az alcsoport indexeivel meghívja a *drawObject(...)* metódust, mely egy alcsoport kirajzolásáért felelős. A *drawObject(...)*-ben a szín, valamint az *isShaded* értéktől függő csupa 1, illetve csupa 0 vektor feltöltése után az indexek töltődnek a kártyára, majd azok kirajzolása történik (a beállított OpenGL csővezetéken keresztül).

5.8 Kamerák

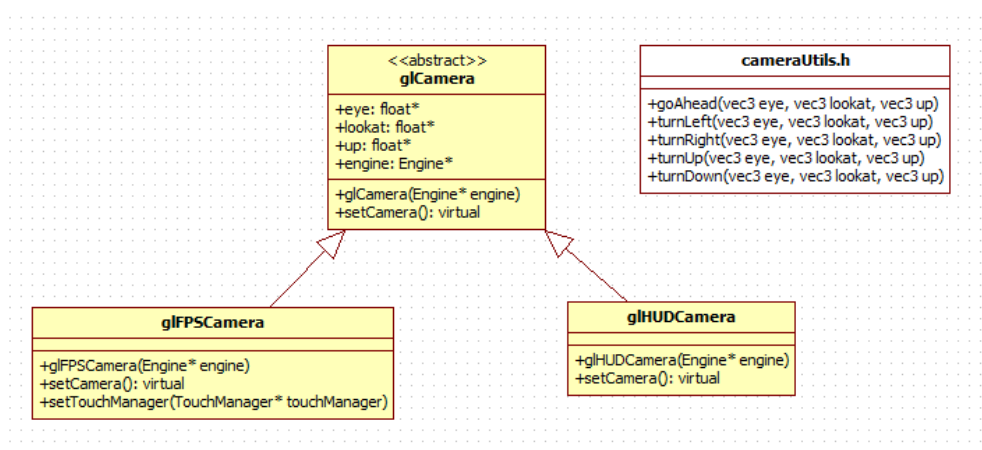
5.8.1 A komponens felépítése

A háromdimenziós transzformációs mátrixok (*Modell*, *View*, *Projection* mátrixok) kezelése összetett feladat, ezért egy külön alkomponens felel érte. Ez a Kamera komponens, mely a *camera* mappában található (5.31. ábra). Itt található meg a fő kamera osztály (*glCamera.h*), a két kamera (*glFPSCamera.h*, és a *glHUDCamera.h*), valamint a kamerakezelés segítő osztály (*cameraUtils.h*).



5.31. ábra: A *camera* alkomponens felépítése

A komponens főosztálya a *glCamera*. Ez az osztály a kamera osztályok közös őse, valamint absztrakt osztály, így csak az azt megvalósító osztályok példányosíthatóak (5.32. ábra). Ez az osztály tartalmazza a kamera fő adatait, mint a pozíció, nézeti irány, és a kameratérbeli felfelé irány, valamint egy pointer a grafikus motor főosztályára (*Engine*). Ezt az osztályt örökli a *glFPSCamera* és a *glHUDCamera* osztály. Mindkét osztály egy-egy típusú kamerát definiál.



5.32. ábra: A *camera* alkomponens osztályai (UML Osztálydiagram)

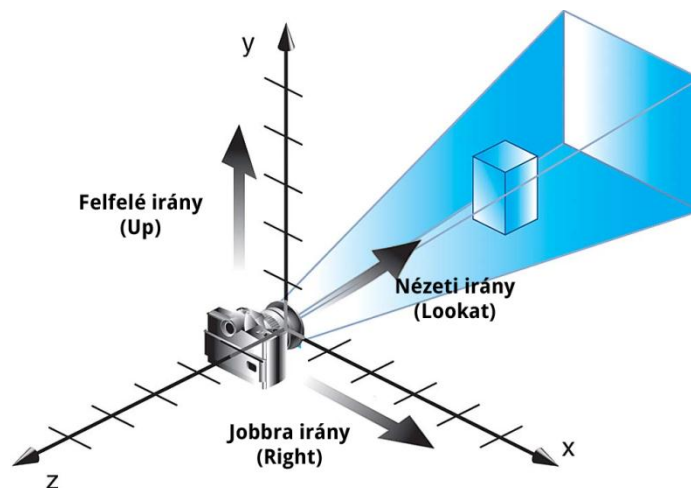
5.8.2 Kamera típusok

A *glFPSCamera* a belsőnézetes játékokban megszokott kamerák viselkedését valósítja meg. Perspektív projekcióval vetít, 45° látószöggel, valamint a grafikus

motorban beállított magasság és szélesség arányokkal. A közeli vágósík 100, a távoli 500000 értékek. Ezeket az értékeket a *setCamera(...)* függvény állítja be, és ezek alapján módosítja a grafikus motor transzformációs mátrixait, az *Engine* osztályban lévő *matrixManager* segítségével. (Ezeket rajzolás előtt még fel kell tölteni a grafikus kártyára). Ebben a függvényben történik meg a kamera pozíciójának feltöltése a grafikus hardverre, melyre a *Fragment Shader*-ben, a spekuláris fényvisszaverődés kiszámításakor van szükség.

A *glFPSCamera* rendelkezik még egy *setTouchManager(..)* funkcióval, mely a kamera vezérlésekor kerül felhasználásra. A külső vezérlő osztály a *TouchManager*, mely a képernyő érintések alapján vezérli az objektumok és a kamera helyzetet. Ebben a függvényben a *TouchManager* által beállított pozíció, nézeti irány és felfelé irány beállítása történik. A vezérlő osztály egy másik fejezetben kerül bemutatásra.

A *glHUDDCamera* a HUD elemek vetítésére használatos. Abban különbözik a *glFPSCamera*-tól, hogy ortografikus vetítést alkalmaz a teljes képernyő szélességében, valamint 0.00...1 és 100 értékek a közeli illetve távoli vágósík. Itt is megtörténik a kamera helyzetének feltöltése a grafikus hardverre. Ez az osztály nem tartalmaz vezérlő függvényt, mivel helyzete mindig állandó.



5.33. ábra: Kamera vektorok [52]

5.8.3 Kameramozgató funkciók

A kameramozgatást segítő funkciók a *cameraUtils.h*-ban találhatóak és a külső vezérlő osztályok használják fel. Működésének elvei azonban szorosan kapcsolódnak a

kamerákhoz, így ebben az alkomponensben került megvalósításra, bár programlogikailag nem kapcsolódik hozzá. A 5.33. ábra szemlélteti a kamerakezelésnél használatos vektorokat.

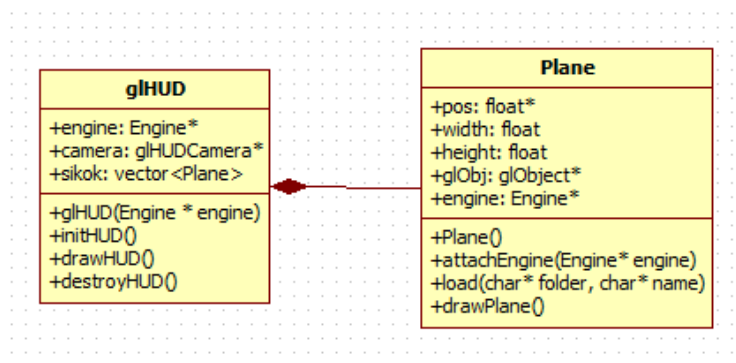
A *turnLeft(...)* illetve *turnRight(...)* funkciók a kapott kamera értékeket úgy állatják be, hogy a nézeti irány vektorát, a felfelé irány vektor körül forgatják el adott szöggel, így a kamerát jobbra, illetve balra fordítva.

A *turnUp(...)* illetve *turnDown(...)* funkciók a felfelé, illetve lefele nézést valósítják meg. Ehhez elsőnek egy jobb irány vektort (*right*) kell kiszámolni, mely a nézeti irány és a felfelé irány vektoriális szorzata. Majd ekörül a vektor körül elforgatni mind a felfelé mind a nézeti irányt adott szöggel, így fordítva fel, illetve le a kamerát.

A *goAhead(...)* függvény a térben való mozgást megvalósító funkció. A kamera pozíciójához hozzáadja a nézeti irány vektort (50 szerezére skálázva), így a kamerából nézve mindig arra haladunk előre amerre nézünk.

5.9 A felhasználói felület (HUD)

A grafikus komponens lehetőséget nyújt felhasználói felületi elemek elhelyezésére. Ezek az elemek textúrázott síkok, melyek a háromdimenziós játéktérrel készült kép felé rajzolódnak ki. Minden ilyen elem egy külön háromdimenziós modell, egy sík, melyre különböző bitképek vannak textúráként ráfeszítve. A HUD kezelés megvalósítása a grafikus komponens főkönyvtárában található *glHUD.h*-ban van megvalósítva. Ebben találhatóak meg a *Plane* illetve *glHUD* osztályok (5.34. ábra).



5.34. ábra: A HUD osztályai (UML Osztálydiagram)

5.9.1 Felületi elemek

A *Plane* osztály a megjelenítendő sík, melynek modellje egység hosszú oldalakkal rendelkezik. A *pos* tárolja a pozícióját a képernyőn, a *width-height* pedig a méretét. A képernyő középpontja a 0,0 koordináta, bal alsó sarka a -1,-1 jobb felső a 1,1. A *Plane* osztály rendelkezik még egy *glObject*, valamint *Engine* pointerrel is, mely a kirajzoláshoz szükséges.

A *Plane* osztály *attachEngine(...)* függvénye hivatott a motor pointer beállítására, valamint a *glObj* létrehozására. A *load(...)* a modell betöltésére, a *drawPlane()* pedig a sík kirajzolására. Az osztály a *glObject* osztályt használja fel, azzal a különbséggel, hogy a *draw(...)* függvényét a transzformációk miatt nem használja, hanem saját transzformációs beállításokkal rajzol (pozícióba mozgatás és skálázás a megfelelő paraméterekkel).

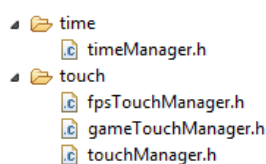
5.9.2 A HUD

A HUD felépítéséért a *glHUD* osztály felel. Ez az osztály rendelkezik egy *Engine* és egy *glHUDCamera* pointerrel, valamint egy dinamikus tömbbel, melyben a kirajzolandó síkok vannak. A konstruktorában létrejön a *glHUDCamera*, mely a felhasznált kamera lesz, és beállításra kerül úgy, hogy a $-z$ irányba nézzen az origóra. Itt a kirajzolt elemeknek nem szabad perspektivikus torzítás szenvednie, ezért kell az ortografikus kamerát használni. Az *initHUD()* függvény kimentti az éppen aktuális mátrix beállításokat, majd inicializálja őket, hogy a többi kamera beállításai ne változzanak, ill. ne legyenek hatással a HUD-ra. A *destroyHUD()*, az *initHUD()* elején kimentett értékeket tölti vissza a transzformációs mátrixokba.

A *drawHUD()* függvény elsőnek az *initHUD()* függvényt hívja, majd a síkokat sorban kirajzolja, és a *destroyHUD()* függvénnyel befejezi a HUD kirajzolását.

5.10 Idő- és Érintéskezelés

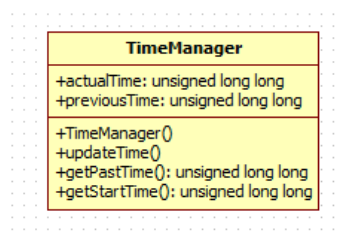
Bár a grafikus megjelenítéshez nem szükséges idő- és érintéskezelés, annak felhasználásához szinte elengedhetetlen, ezért található meg a grafikus komponensben időkezelő és érintéskezelő alkomponens. Az időkezelő alkomponens a *time* könyvtárban található, a *timeManager.h* állományban. Az érintés kezelő alkomponens pedig a *touch* könyvtárban található *touchManager.h*, *fpsTouchManager.h* illetve *gameTouchManager.h*-ban (5.35. ábra).



5.35. ábra: A *time* és a *touch* alkomponensek felépítése

5.10.1 Az időkezelő osztály

Az időkezelést megvalósító osztály a *TimeManager*. Ez az osztály beépített *posix* kéréseket használva kérdezi le a rendszer idejét. Ez a *posix* függvény [53] a *clock_gettime(CLOCK_MONOTONIC,&time_spec);* hívásával érhető el, és a rendszerindítás óta eltelt időt adja vissza egy *timespec* struktúrában. Csak a futási időt számolja, de amíg a telefon alvó állapotban van, addig nem számol. Ebből a *timespec* struktúrából elérhető az eltelt idő nanoszekundumban.

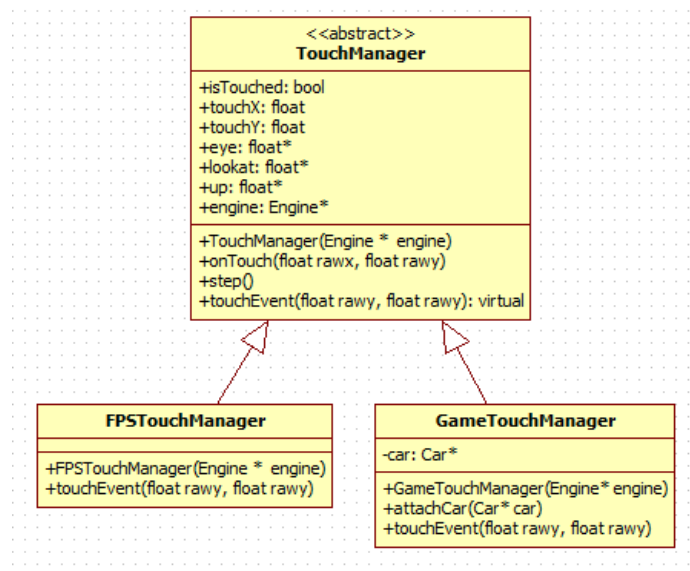


5.36. ábra: A *TimeManager* osztály (UML Osztálydiagram)

Az osztály (5.36. ábra) tárol egy *actualTime* és egy *previousTime* változót, melyek a kellő pontosság érdekében előjel nélküli *long long* változók (64 bit). Az *actualTime* az éppen kirajzolás alatt lévő kép előtt lévő idő. A *previousTime* pedig az előző képernyőkép rajzolása előtt lévő idő. A kettő különbségéből kiszámolható, a képernyőrajzolás alatt eltelt idő.

Az osztály rendelkezik egy `updateTime()` függvénnyel, mely frissíti, az `actualTime` és `previusTime` változókat. Ezeket a kirajzolás elején érdemes meghívni. A `getPastTime()`-al a rajzolás alatt eltelt időt lehet lekérni, a `getStartTime()`-al pedig az indítás óta eltelt időt.

5.10.2 Az érintéskezelés



5.37. ábra: A *touch* alkomponens osztályai (UML Osztálydiagram)

Az érintéskezelés megvalósítása a *touch* alkomponensben a *touchManager.h*, *fpsTouchManager.h*, valamint *gameTouchManager.h* fájlokban található. Az alkomponens főosztály az absztrakt *TouchManager* osztály (5.37. ábra). Ez az osztály tárolja a (később hozzácsatolt) kamera értékeit (*eye*, *lookat*, *up* vektorok), valamint a lenyomott képernyő koordinátáit (*touchX*, *touchY*), és egy pointert a grafikus motor főosztályára.

Az osztály hierarchia megtervezése során fontos szempont volt, hogy a lenyomás, nyomva tartás illetve felengedés kezelésért a főosztály feleljen, az öröklő osztályok pedig csak a rájuk jellemző működést valósítsák meg. A főosztály *onTouch(...)* függvénye akkor hívódik meg, amikor az alkalmazás keret meghívja a *main.cpp* *onTouch()* függvényét, tehát minden olyan esetben, amikor a felhasználó hozzáért, vagy levette az ujját az érintőképernyőről. Itt kerül elmentésre a lenyomott pont koordinátái, valamint hogy lenyomás történt vagy felengedés. A *step()* minden képernyőrajzolás előtt hívódik meg, és ebben a függvényben a *touchEvent(...)* függvény hívódik tovább, melyet a különböző öröklő osztályok valósítanak meg.

Az *FPSTouchManager* az FPS játékokban megszokott mozgásért felel, ami a játékkalkalmazás garázsnézetében került felhasználásra. Ehhez felhasználja a már ismert *cameraUtils* segédfunkciókat, illetve az *FPSCamera setTouchManager(..)* metódusával állítja be az adott kamerának a már kiszámolt értékeket. A szélessége illetve magassági koordináták alapján dönti el a fordulási irányt. A sarkok érintésekor nem alkalmaz változtatás, a képernyő közepén pedig az előre mozgást végzi el.

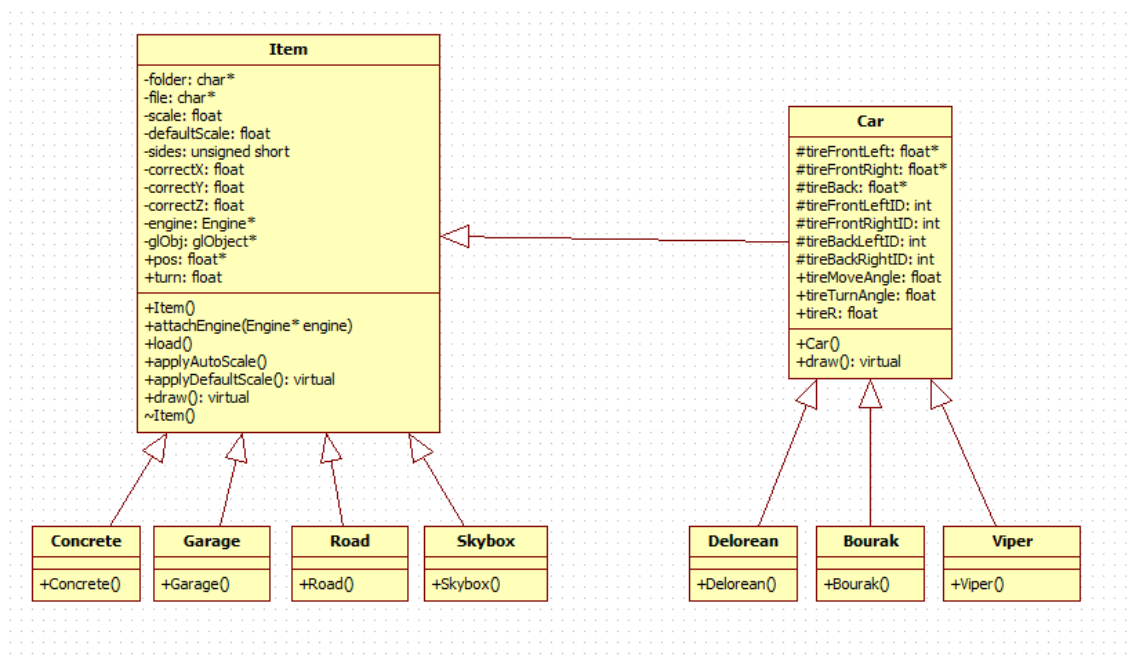
A *GameTouchManager* a játékkalkalmazás verseny módjában került felhasználásra. Az előzőekhez képp est annyival egészül ki az osztály, hogy egy pointert tárol arról az autóról, amihez hozzá van kötve. Jobbra, ill. balra fordulás eseménykor az autót forgatja jobbra illetve balra. Ennek az osztálynak az implementálását Dániel készítette el.

6 A játékkalkalmazás

A grafikus komponens funkcióinak demonstrálására, egy játékkalkalmazást készítettem, melyben bizonyos részeket Dániel implementált (ezeket minden esetben jelölöm).

6.1 Entitások

A játékkalkalmazás entitásai mindig egy entitás osztály példányai. Minden entitás típus egy külön osztály (6.1. ábra). Az entitás típusok alapvetően két csoportba sorolhatóak, és az alkalmazásmag *items* mappájában találhatóak. Alapvetően minden elem egy *Item* entitás típus leszármazottja, és vannak elemek melyen autóként is viselkednek, így egy *Car* entitás típus leszármazottai is. Mivel minden autó maga egy elem is, ezért a *Car* osztály az *Item* leszármazottja. Az entitás típusok mind, ezen két osztály valamelyikét öröklík, és csak a speciálisan arra a típusra jellemző adatokat definiálják.



6.1. ábra: A entitás típusok (UML Osztálydiagram)

6.1.1 Az *Item* osztály

Az *Item* osztály tartalmazza az entításokra általánosan vonatkozó információkat, és azok beállítását végző funkciókat. A *folder* és *file* a modellfájl elérési útja. A *scale*,

defaultScale az elem nagyítására vonatkozó információk, míg a *correctX*, *correctY*, *correctZ* a modell saját origójához való eltolását szabályozza. A *sides* az oldal információkat tartalmazza a hátsólap eldobáshoz. A publikus *pos* illetve *turn* az autó pozícióját illetve az Y tengelyen való elfordulási szögét állítja be a világtérben. Ezen kívül tartalmaz még egy *Engine* és egy *glObject* pointert, melyek a rajzoláshoz szükségesek.

Az *attachEngine(...)* függvénye a grafikus motor csatolásához, valamint a rajzolási objektum, a *glObj* létrehozásához használatos. A *load()* függvény betölti a modellt a fájl elérési út alapján. Az *applyAutoScale()* függvény az éppen megfelelően látható méretre nagyítja a tárgyat, míg a *applyDefaultScale()* visszaállítja az eredeti nagyítási beállításokat. Az automatikus skálázás függvényeit Dániel valósította meg. A *draw()* függvény kirajzolja az entitást a *glObj* felhasználásával, a megadott paraméterekkel transzformálva.

6.1.2 A *Car* osztály

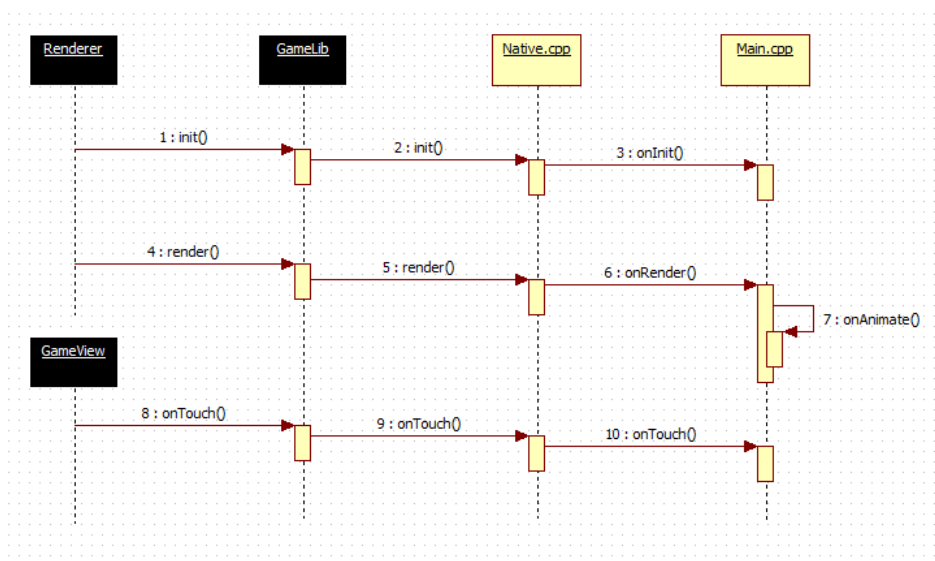
A *Car* osztály minden autó entitás típus öse. Az osztály az *Item* osztályból származik, és az autókra vonatkozó plusz információkat tárolja, illetve ezen paraméterektől függően rajzolja ki az entitást. A *tireFrontLeftID*, *tireFrontRightID*, *tireBackLeftID*, *tireBackRightID*, a 3D modellben a bal-jobb, első és hátsó kerekek azonosító számai. A *tireFrontLeft*, *tireFrontRight*, *tireBack* vektorok a bal első, jobb első, és hátsó kerekek elfordulási középpontját definiálják (*pivot point*), mivel mindkét hátsó kerék 1 tengely mentén fordulhat csak el, elég az egyik kerék értékei tárolni. A *tireR* a kerekek sugara. Ezen értékek mindegyike meghatározható a modell és egy 3D modellkészítő program segítségével (jelen esetben a *Blender*-t [54] használtam). Sajnos az általunk felhasznált modell formátumok ezeket az információkat nem vagy csak nehezen kiszámolhatóan tartalmazzák, így azokat a forráskódban kell tárolni. A *tireMoveAngle* a kerék mozgásából fakadó elfordulás szögét tartalmazza, a *tireTurnAngle* pedig a kormányzásból fakadó elfordulás szögét.

A *draw()* függvény, a *glObj* felhasználásával (amit az *Item*-től örökölt) a kerekre vonatkozó transzformációkat alkalmazva rajzolja ki az autót. Először minden kereket eltranszformál a forgási középpontjában, majd ott elforgatja, és visszatranszformálja a pozíciójában. Ezek után magát az autót is a megfelelő pozícióba transzformálja, és kirajzolja.

A többi, az ábrán szereplő osztály mind a konkrét entitás típusok, és csak a rájuk vonatkozó konkrét értékeket inicializálják a konstruktorukban (modell elérési út, az autóknál az azonosítók és a forgási középpontok... stb.).

6.2 A játékalkalmazás főprogramja

A játékalkalmazás főprogramja a *main.cpp*, és a hozzá tartozó *main.h*, melyben függvényinek definíciói vannak, és a *jni* könyvtár gyökerében találhatóak. Mint az már kifejtettem a *native.cpp*-n keresztül a *JNI* technológia segítségével csatlakozik a keretalkalmazás és az alkalmazásmag. A Main logikailag négy részre osztható: Inicializáció, Érintéskezelés, Animáció, Kirajzolás. A Main hívási szekvenciáit, a 6.2. ábra mutatja.



6.2. ábra: A Main hívási szekvenciái (UML Szekvencia Diagram)

6.2.1 Az inicializáció

Az inicializáció két részből épül fel, első a szükséges globális változók létrehozása, majd ezek beállítása az *onInit()* függvényben. Az szükséges könyvtárak fejlécfájlok betöltése után, elsőnek a grafikus motor, és a fő kamera jön létre, majd a két féle érintéskezelő osztály, valamint egy ezek valamelyikére mutató pointer, melynek az egységes irányításkezelésnél lesz szerepe.

```
Engine engine;

glFPSCamera cam(&engine);

FPSTouchManager fpsTouchManager(&engine);
GameTouchManager gameTouchManager(&engine);
TouchManager * touchManager;
```

Majd az időkezelő egység, és a felhasználói felületkezelő jön létre.

```
TimeManager timeManager;  
unsigned long long startTime;  
  
glHUD hud(&engine);
```

Ezután az aktuális autó és játéktípust tároló globális változók jönnek létre, majd az entitások jönnek létre.

```
int Gcar;  
int Gtype;  
  
Garage garage;  
Concrete concrete;  
  
Bourak bourak;  
Bourak enemyBourak;  
Delorean delorean;  
  
Car* playerCar;  
Car* enemyCar;  
  
Skybox skybox;  
Road road;
```

Végül az entitásokhoz felhasznált fizikai szimulációs osztályok, és konstansok jönnek létre.

```
RigidBody playerPhy;  
RigidBody enemyPhy;  
vec2 gearForce;
```

Ezután a grafikus felület létrejöttékor, vagy frissülésekor, az *onInit(...)* függvény fut le. Ennek a függvénynek a feladata a kezdőértékek, és kezdőbeállítások alkalmazása. Elsőnek a törlési szín kerül beállításra, valamint az elemek alapvető inicializálása. Ezután a kiválasztott autó kódjától függően valamelyik autó értéke kerül a játékos autójának pointerjébe, valamint ha a játékmód olyan, az ellenséges autó pointerje is beállításra kerül.

```
if (Gcar == 1){playerCar = &delorean;}  
if (Gcar == 2){playerCar = &bourak;}
```

Miután ezek a mutatók beállításra kerültek az általuk mutatott modellek, valamint a játéktér többi modelljének a betöltése kezdődik meg, majd ezeknek az entitásoknak az alapvető beállítása (pozíció, motorcsatolás... stb.) történik. A játékmódtól függően kerül beállításra az érintéskezelésért felelős osztályra mutató pointer is.

```
if (Gtype == 1)  
{  
    touchManager = &fpsTouchManager;  
    ...  
}  
if (Gtype == 2)  
{  
    gameTouchManager.attachCar(playerCar);
```

```
touchManager = &gameTouchManager;
...
}
```

Ezután a felhasználói felület elemeinek létrehozás, betöltése, és a felülethez hozzáadása történik.

```
Plane p;
p.pos= {-0.825f,0.75f};
p.height=0.175f; p.width=0.25f;
p.attachEngine(&engine);

if (Gtype == 1) p.load("GarageHUD","garage_HUD.obj");
if (Gtype == 2) p.load("RaceHUD","race_HUD.obj");

hud.planes.clear();
hud.planes.push_back(p);
```

6.2.2 Az érintéskezelés

A Main érintéskezelő függvénye az *onTouch(..)* függvény, mely a képernyő egy pontjának lenyomása, és felengedése esetén hívódik. Itt a már beállított érintéskezelő érintés függvénye hívódik meg (ami a fő érintéskezelő osztályban lett megvalósítva, de a virtuális függvények nyújtotta előnyöket kihasználva, a tovább hívott függvény az aktuálisan beállított érintéskezelőnek megfelelő).

```
touchManager->onTouch(rawx, rawy);
```

6.2.3 Animáció

Az animációs függvény alapvetően két részre oszlik, a játéktípustól függően. Ha a garázs játéktípus van beállítva, akkor csak a játékos autójának forgatása történik, és az autó kerekének forgatása, az idő függvényében.

```
float angle=time / (1000000 * 10);
playerCar->turn =angle;
playerCar->tireMoveAngle =angle;
```

A másik részben a versenymód animációi történnek. Itt a fizikai szimulációs elemek összekötése történik az entitásokkal. Itt történik az erők alkalmazása, forgatása, a kerekek forgatása a megfelelő szögekkel, valamint az autó mozgatása a fizikai szimulációtól függően.

```
...
playerPhy.force[0] = gearForce[0]* sin(playerCar->turn / 180.0f * 3.1415f)
+ gearForce[1] * cos(playerCar->turn / 180.0f * 3.1415f);
playerPhy.force[1] = 0.0f;
playerPhy.force[2] = gearForce[0]* cos(playerCar->turn / 180.0f * 3.1415f)
- gearForce[1] * sin(playerCar->turn / 180.0f * 3.1415f);
playerPhy.mass = 1000.0f;
```

```

...
playerPhy.animate(dt);
...
playerCar->pos[0] = playerPhy.pos[0];
playerCar->pos[2] = playerPhy.pos[2] + 800.0f;
...
playerCar->tireTurnAngle = 0.0f;
playerCar->tireMoveAngle = -playerPhy.getS() / (2.0f * playerCar->tireR) * 180.0f;
...

```

6.2.4 A kirajzolás

A kirajzolás függvény (*onRender()*) a keretalkalmazás által hívódik, amikor az csak lehetséges, a lehető legtöbb képet láttatva a felhasználóval. A kirajzolás elején a törlési beállítások illetve a transzformációs mátrixok alaphelyzetbe állítása történik.

```

engine.renderInit();

glClearColor(0.15, 0.15, 0.15, 0.0f);
glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);

```

Ezután az időkezelő értékeinek frissítése, valamint az érintéskezelő léptetése következik, valamint az elemek animációja (a főprogram *onAnimate(...)* hívása)

```

timeManager.updateTime();
touchManager->step();
onAnimate(timeManager.actualTime, timeManager.getPastTime());

```

A játéktípustól függően, ha garázmódban van a játék, akkor a kamerát az aktuális érintéskezelőhöz csatlakozik, majd a kamera értékei frissülnek (*setCamera(...)*), valamint az entitások rajzolása történik.

```

cam.setTouchManager(touchManager); cam.setCamera();
garage.draw(); ...

```

Ha verseny módban van a játékos, akkor is hasonlóan az előző módhoz beállításra kerül az érintéskezelő, és az elemek kirajzolódnak, ám előtte a kamerát mozgatni kell a játékos autójához, hogy az folyamatosan az irányított autóhoz „tapadjon”. Egy kamera offset kerül beállításra, hogy az irányított autó középpontjához képpest hol helyezkedik el a kamera, majd ez az offset az autó elfordulásával van elforgatva, hogy a kamera mindig a haladási irányba nézzen.

```

float turnRad = playerCar->turn / 180.0f * 3.1415f;
vec3 cameraOffset = {-2000.0f, 0.0f, 0.0f };
vec3 cameraOffsetRotated = {cameraOffset[0] * sin(turnRad), 0.0f,
cameraOffset[0] * (cos(turnRad))};

```

Majd a kamera helyzete és nézési iránya kerül beállításra (fontos hogy a nézeti vektort nem szabad az autó pozíciójára állítani, fontos azt előtte bemozgatni a szem pozíciójába, azaz a kamera pozíció vektorát kivonni belőle).

```
touchManager->eye      = {playerCar->pos[0]+cameraOffsetRotated[0],  
1000.0f+cameraOffsetRotated[1],      playerCar->  
>pos[2]+cameraOffsetRotated[2]};  
  
touchManager->lookat[0] = -touchManager->eye[0] + playerCar->pos[0];  
touchManager->lookat[1] = -touchManager->eye[1] + 200.0f;  
touchManager->lookat[2] = -touchManager->eye[2] + playerCar->pos[2];
```

Végül a felhasználói felület kirajzolása történik (HUD rajzolás), mely a játékkalkalmazásban a játékmódot jelző kép, a bal felső sarokban.

```
hud.drawHUD();
```

7 Értékelés és továbbfejlesztési lehetőségek

7.1 Tesztelés és értékelés

Az elkészült grafikus komponens mind a témakiírás, mind a pontosított feladat-specifikációban támasztott követelményeknek megfelel. A használhatóság és esztétikán kívül a teljesítmény is szempont volt, így azt csupán az alkalmazás használata során nem tudtam kellően tesztelni. Ezért az időkezelő és eseménynaplóba író komponenseket felhasználva megmértem a képernyő frissítések számát másodpercenként. A specifikációban az elfogadható képfrissítés számnak a 30-at definiáltam. Az általam használt tesztkészüléken (Samsung Galaxy S2) ez az érték mindkét játékmódban átlépte a 30-as határértéket. Részletesebb modellek (több háromszög) használatával ez a szám folyamatosan csökkent. Az alkalmazás tesztelése során készített képernyőképeket egy Samsung Galaxy S2 készülékkel készített, melyen 4.0.3-as verziójú Android rendszer futott (7.1. ábra, 7.2. ábra, 7.3. ábra, 7.4. ábra, 7.5. ábra, 7.6. ábra).



7.1. ábra: Az autóválasztó (az első autó kiválasztva)



7.2. ábra: Az autóválasztó (a második autó kiválasztva)



7.3. ábra: Garázs nézet az első autóval



7.4. ábra: Garázs nézet a második autóval



7.5. ábra: Verseny nézet az első autóval



7.6. ábra: Verseny nézet a második autóval

Több eszközön is kipróbáltam az alkalmazást (a fentebb említett tesztkörnyezet nélkül), és szinte mindegyiken élvezhető képfrissítést és megjelenést tapasztaltam, az alkalmazás megfelelően működött.

7.2 Továbbfejlesztési lehetőségek

Bár a követelményeknek megfelel, mégis több szempontból nem teljes az alkalmazás. Sok olyan grafikus megjelenítési módszert használnak a piacon lévő játékal alkalmazások, melyet a grafikus motor nem támogat. Ilyen például az árnyékok kezelése (*Shadow Mapping*)[55], a többmenetes rajzolás (*Deffered Rendering*)[56], a már említett *Displacement Mapping* [57], környezet tükröződés (*Environment Mapping*)[58], illetve különböző képrajzolási módszerek (*Cell Shading* [59], *HDR Rendering* [60]).

A felhasználói felület (HUD) is csak korlátozott funkciókkal bír (csak négyzet alakú elemek, nincs átlátszó elem). Az árnyalás során csak egy darab fix fény van, valamint sok változó értéke is konstans (pl. spekuláris visszaverődés színe). Sok játékmotor lehetővé teszi az entitások scriptnyelvekből való kezelését (pl. *Lua Script* [61], *JavaScript* [62]), ebben motorban csak a C++ forrásban lehet ezt megtenni.

7.3 Tervek

Az említett továbbfejlesztési lehetőségeket tervezem megvalósítani a későbbiekben, illetve tervezem az elkészült grafikus motor publikálását a weben.

8 Köszönetnyilvánítás

Szeretném megköszönni konzulensemnek, Fehér Marcellnek mind az alkalmazás fejlesztése közben, mind ezen dolgozat szerkesztése közben nyújtott segítségét. Szeretném továbbá megköszönni Tóth Dánielnek a közös munka során nyújtott támogatását, valamint Badics Alexnek a 3D képalkotás elméletének megértésében nyújtott segítségét.

Irodalomjegyzék

- [1] Mobile Thinking: *Latest Mobile Stats*,
<http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats/a#smartphoneos> (2012.nov.)
- [2] Android Authority: *Mobile GPU Performance Evolution*,
<http://www.androidauthority.com/mobile-gpu-performance-evolution-77655/>
(2012.nov.)
- [3] Android Developers: *Az Android SDK*,
<http://developer.android.com/sdk/index.html> (2012.dec.)
- [4] Android Developers: *Az Android NDK*,
<http://developer.android.com/tools/sdk/ndk/index.html> (2012.dec.)
- [5] Khronos Group: *Az OpenGL ES*,
<http://www.khronos.org/opengles/> (2012.dec.)
- [6] OpenGL.org: *GLUT*,
<http://www.opengl.org/resources/libraries/glut/> (2012.dec.)
- [7] Blogspot: *system-architecture.jpg*,
http://2.bp.blogspot.com/-4Ok-ecmUOCs/TfHsC-yd_SI/AAAAAAAAAiE/XqwMGqK4Xzw/s1600/system-architecture.jpg
(2012.dec)
- [8] Eclipse: *Eclipse Juno*, <http://www.eclipse.org/> (2012.dec.)
- [9] Cygwin: *Cygwin 1.7.17-1*, <http://www.cygwin.com/> (2012.nov.)
- [10] Subversion: *Subversion 1.7.7*, <http://subversion.apache.org/> (2012.nov.)
- [11] TortoiseSVN: *TortoiseSVN 1.7.10*, <http://tortoisesvn.net/> (2012.nov.)
- [12] VisualSVN: *VisualSVN Server 2.5.7*, <http://www.visualsvn.com/> (2012.nov.)
- [13] WinMerge: *WinMerge 2.12.4*, <http://winmerge.org/> (2012.nov.)
- [14] Android Developers: *Dashboards*,
<http://developer.android.com/about/dashboards/index.html> (2012.dec.)
- [15] Mobile Game Engines: *3D Android Game Engines*,
http://mobilegameengines.com/android/3d_game_engines (2012.dec.)
- [16] Google Code: *A libGDX Grafikus Motor*,
<http://code.google.com/p/libgdx/> (2012.dec.)
- [17] Google Code: *A min3d Grafikus Motor*,
<http://code.google.com/p/min3d/> (2012.dec.)

- [18] Balsamiq Mockups: *Balsamiq Mockups for Windows 2.2.3*,
<http://www.balsamiq.com/products/mockups> (2012. dec.)
- [19] Adobe Photoshop: *Adobe Photoshop CS6 Trial*,
<http://www.adobe.com/hu/products/photoshop.html> (2012.dec.)
- [20] Android Developers: *Support Library*,
<http://developer.android.com/tools/extras/support-library.html> (2012.dec.)
- [21] Stuermer Benjamin: *ViewPager fertig*,
http://www.stuermer-benjamin.de/v12_1/wp-content/themes/elemin/themify/img.php?src=http://www.stuermer-benjamin.de/v12_1/wp-content/uploads/2012/11/001_viewpager_fertig.png&w=548&h (2012.dec.)
- [22] Android Developers: *Fundamentals*,
<http://developer.android.com/guide/components/fundamentals.html> (2012. dec.)
- [23] Android Developers: *Fragments*,
<http://developer.android.com/guide/components/fragments.html> (2012. dec.)
- [24] Khronos Group: *EGL*, <http://www.khronos.org/egl> (2012.dec.)
- [25] Oracle: *Java Native Interface*,
<http://docs.oracle.com/javase/7/docs/technotes/guides/jni/index.html> (2012.dec.)
- [26] C++.com: *Standard Template Library*, <http://www.cplusplus.com/reference/stl/>
(2012.dec.)
- [27] Android Developers: *Android NDK Samples*,
<http://developer.android.com/tools/sdk/ndk/index.html#Samples> (2012.dec.)
- [28] OpenGL.org: *OpenGL Error Codes*,
http://www.opengl.org/wiki/GL_Error_Codes (2012.nov.)
- [29] Khronos Group: *GLSL ES Specification*,
http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf (2012.dec.)
- [30] Wikipedia: *Invertible Matrix*, http://en.wikipedia.org/wiki/Invertible_matrix
(revision: 17:36, 19 November 2012)
- [31] Wikipedia: *Kifejtési tétel*,
http://hu.wikipedia.org/wiki/Kifejt%C3%A9si_t%C3%A9tel (revision: 11:03, 5 October 2012)
- [32] NTU EDU: *3D Graphics with OpenGL*,
http://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html (2012. nov.)
- [33] Ugrad CS UBC: *glFrustum(...)*,
<http://www.ugrad.cs.ubc.ca/~cs314/opengl/glFrustum.html> (2012.dec.)

- [34] UNI-MUENSTER: *glOrtho(...)*,
<http://earth.uni-muenster.de/~joergs/opengl/glOrtho.html> (2012.dec.)
- [35] DB-IN: *Projection_example.gif*
<http://pages.cpsc.ucalgary.ca/~eharris/cpsc453/tut16/camera2.gif> (2012.dec.)
- [36] CPSC UCALGARY: *camera2.gif*,
<http://pages.cpsc.ucalgary.ca/~eharris/cpsc453/tut16/camera2.gif> (2012.dec.)
- [37] Lighthouse3D: *Directional Lights*,
<http://www.lighthouse3d.com/tutorials/glsl-tutorial/directional-lights-i/>
(2012.dec.)
- [38] Wikipedia: *Phong reflection model*
http://en.wikipedia.org/wiki/Phong_reflection_model
(revision: 14:17, 25 November 2012)
- [39] Wikipedia: *Blinn-Phong shading model*
http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model
(revision: 16:34, 29 November 2012)
- [40] Wikipedia: *Gouraud shading*,
http://en.wikipedia.org/wiki/Gouraud_shading
(revision: 22:05, 30 November 2012)
- [41] Wikipedia: *Phong shading*,
http://en.wikipedia.org/wiki/Phong_shading
(revision: 06:24, 1 November 2012)
- [42] UQ EDU: *litType.gif*
<http://www.vislab.uq.edu.au/users/manuals/cosmoworlds/images/litType.gif>
(2012.dec.)
- [43] Blog.tartiflop.com: *Texture Mapping*,
<http://blog.tartiflop.com/tag/texture-mapping/> (2012.dec.)
- [44] Greg Dolley's Webblog: *OpenGL Texture Parameters Explained*,
<http://gregs-blog.com/2008/01/17/opengl-texture-filter-parameters-explained/>
(2012.dec.)
- [45] Wikipedia: *Bilinear filtering*,
http://en.wikipedia.org/wiki/Bilinear_filtering (revision: 10:00, 12 September 2012)
- [46] Wikipedia: *Trilinear filtering*,
http://en.wikipedia.org/wiki/Trilinear_filtering (revision: 13:18, 16 November 2011)
- [47] Wikipedia: *Anti-aliasing filter*,
http://en.wikipedia.org/wiki/Anti-aliasing_filter (revision: 13:47, 2 December 2012)

- [48] Db-interactively: *All about OpenGL ES 2.x – (part 2/3)*,
<http://db-in.com/blog/2011/02/all-about-opengl-es-2-x-part-23/> (2012.dec.)
- [49] Wikipedia:*Back-face culling*,
http://en.wikipedia.org/wiki/Back-face_culling (revision: 22:15, 15 July 2011)
- [50] Khronos Group:*glUniform*,
<http://www.khronos.org/opengles/sdk/docs/man/xhtml/glUniform.xml> (2012.nov.)
- [51] OpenGL.org: *glVertexAttribPointer()*,
<http://www.opengl.org/sdk/docs/man/xhtml/glVertexAttribPointer.xml>,
(2012.nov.)
- [52] GameDev.net: *03fig11.jpg*,
<http://images.gamedev.net/features/programming/oglch3excerpt/03fig11.jpg>
(2012.dec.)
- [53] Linux-documentation.com: *POSIX Manual Page – clock_gettime*,
http://linux-documentation.com/en/man/man3p/clock_gettime.html (2012.dec.)
- [54] Blender:*Blender 2.64a*,
<http://www.blender.org/download/get-blender/> (2012.dec.)
- [55] Wikipedia:*Shadow Mapping*,
http://en.wikipedia.org/wiki/Shadow_mapping (revision: 14:21, 14 November 2012)
- [56] Wikipedia:*Deferred Shading*,
http://en.wikipedia.org/wiki/Deferred_shading (revision: 16:13, 26 August 2012)
- [57] Wikipedia:*Displacement Mapping*,
http://en.wikipedia.org/wiki/Displacement_mapping (revision: 02:33, 3 September 2012)
- [58] Wikipedia:*Reflection Mapping*,
http://en.wikipedia.org/wiki/Reflection_mapping (revision: 15:17, 15 August 2012)
- [59] Wikipedia:*Cell Shading*,
http://en.wikipedia.org/wiki/Cel_shading (revision: 00:52, 27 November 2012)
- [60] Wikipedia:*High Dynamic Range Rendering*,
http://en.wikipedia.org/wiki/High_dynamic_range_rendering (revision: 11 November 2012)
- [61] Lua.org:*The Programming Language Lua*,
<http://www.lua.org/> (2012.dec.)
- [62] Wikipedia:*JavaScript*,
<http://hu.wikipedia.org/wiki/JavaScript> (revision: 01:57, 1 November 2012)

Ábrajegyzék

1.1. ábra: Az Android Platform felépítése [7]	10
1.2. ábra: A Java és a Natív kód kapcsolata	11
1.3. ábra: A Natív alkalmazás fordítás folyamata (Folyamatábra)	12
4.1. ábra: A megtervezett „Screen Flow”	20
4.2. ábra: A Projekt felépítése	21
4.3. ábra: A ViewPager elem működése [21]	22
4.4. ábra: A Főképernyő és környezete (UML Osztály Diagram)	23
4.5. ábra: A Játékképernyő és környezete (UML Osztály Diagram)	24
4.6. ábra: Az alkalmazásmaghoz való kapcsolódás Java oldalról	25
5.1. ábra: Az alkalmazásmag felépítése	27
5.2. ábra: A Natív kód kapcsolata a Java oldallal (UML Szekvencia Diagram)	29
5.3. ábra: A grafikus komponens felépítése	31
5.4. ábra: Az Engine osztály (UML Osztály Diagram)	32
5.5. ábra: Az Engine függvényeinek hívási szekvenciája (UML Szekvencia Diagram)	33
5.6. ábra: A Mátrix alkomponens felépítése	37
5.7. ábra: A MatrixUtils osztály (UML Osztálydiagram)	37
5.8. ábra: A Világtér és Modelltér kapcsolata [32]	39
5.9. ábra: A Világtér és a Kameratér kapcsolata [32]	39
5.10. ábra: A Perspektívikus és Ortografikus vetítés [35]	40
5.11. ábra: A Perspektívikus vetítés paraméterei [32]	40
5.12. ábra: Az Ortografikus vetítés paraméterei [32]	41
5.13. ábra: A MatrixStack osztály (UML Osztálydiagram)	41
5.14. ábra: A Matrix Manager osztály (UML Osztálydiagram)	42

5.15. ábra: Az OpenGL ES 1.1 csővezetéke [5].....	43
5.16. ábra: Az OpenGL ES 2.0 csővezetéke [5].....	44
5.17. ábra: <i>Shader</i> változó típusok	44
5.18. ábra: A Goruad árnyalás [40]	46
5.19. ábra: A Phong árnyalás [41]	47
5.20. ábra: Fénytípusok [42]	47
5.21. ábra: A Phong árnyalás típusai [38]	47
5.22. ábra: A diffúz fényvisszaverődés vektorai [37]	48
5.23. ábra: A spekuláris fényvisszaverődés vektorai [37]	49
5.24. ábra: Az <i>shaders</i> alkomponens	49
5.25. ábra: A textúra térképezés [43].....	53
5.26. ábra: A <i>Mipmapping</i> technika által készített kép [46]	54
5.27. ábra: A <i>Ripmapping</i> technika által létrehozott kép [47]	55
5.28. ábra: A Trilinear (bal), és az Anisotropic (jobb) szűrés közötti látványos különbség.....	55
5.29. ábra: A <i>glObject</i> osztály (UML Osztálydiagram).....	58
5.30. ábra: A hátsólap eldobás szemléltetése [49].....	59
5.31. ábra: A <i>camera</i> alkomponens felépítése	62
5.32. ábra: A <i>camera</i> alkomponens osztályai (UML Osztálydiagram)	62
5.33. ábra: Kamera vektorok [52].....	63
5.34. ábra: A HUD osztályai (UML Osztálydiagram)	65
5.35. ábra: A <i>time</i> és a <i>touch</i> alkomponensek felépítése	67
5.36. ábra: A <i>TimeManager</i> osztály (UML Osztálydiagram)	67
5.37. ábra: A <i>touch</i> alkomponens osztályai (UML Osztálydiagram)	68
6.1. ábra: A entitás típusok (UML Osztálydiagram)	70
6.2. ábra: A Main hívási szekvenciái (UML Szekvencia Diagram)	73

7.1. ábra: Az autóválasztó (az első autó kiválasztva)	78
7.2. ábra: Az autóválasztó (a második autó kiválasztva)	79
7.3. ábra: Garázs nézet az első autóval	79
7.4. ábra: Garázs nézet a második autóval	80
7.5. ábra: Verseny nézet az első autóval	80
7.6. ábra: Verseny nézet a második autóval.....	81

„A” Függelék

Az „A” függelék tartalmazza a telepítési útmutatót. Az alkalmazás telepítője, valamint forráskódja is elérhető az alábbi címen vagy a mellékletként csatolt lemezen is:

<https://dl.dropbox.com/u/53134959/Installer.zip>

Az alkalmazás telepítő fájljai a melléklet „Telepítő” mappájában találhatóak.

1. lépés

Az alkalmazás telepítése előtt a „teszt” mappát, annak teljes tartalmával együtt a telefon belső tárterületének gyökérkönyvtárába kell másolni. Ez a könyvtár tartalmazza a szükséges modell fájlokat. Ezek nélkül az alkalmazás nem működőképes.

2. lépés

Az 3DGame.apk-t át kell másolni a telefon belső tárhelyére bárhova, majd azt a telefonon egy fájlböngésző segítségével megkeresni, és megnyitni. Ez az alkalmazás telepítője, a szükséges engedélyeket engedélyezni kell, majd az alkalmazás feltelepül. Ezután az alkalmazás használható.

„B” Függelék

A „B” függelék a felhasznált modell, és képfájlok forrását tartalmazza.

[1] Név: *Bourak*, Fájl: „./teszt/Bourak” teljes könyvtár,

Forrás: <http://www.turbosquid.com/FullPreview/Index.cfm/ID/270273>

Szerző: <http://www.turbosquid.com/Search/Artists/StepanProkop>

[2] Név: *Delorean*, Fájl: „./teszt/Delorean/” teljes könyvtár

Forrás: <http://www.turbosquid.com/3d-models/free-delorean-3d-model/214487>

Szerző: <http://www.turbosquid.com/Search/Artists/lkraan>

[3] Név: *Garázs beton textúra*, Fájl: „./teszt/Concrete/Texture/concrete.BMP”

Forrás: <http://onlyhdwallpapers.com/art/texture-concrete-cracked-by-ivelt-resources-high-resolution-desktop-hd-wallpaper-664581/>

Szerző: Ismeretlen

[4] Név: *Garázs plafon textúra*, Fájl: „./teszt/Garage/Texture/garage2.BMP”

Forrás: <http://ftextures.com/textures/Wood-wall-texture.jpg>

Szerző: Ismeretlen

[5] Név: *Garázs oldalsó textúra*, Fájl: „./teszt/Garage/Texture/garage2.BMP”

Forrás: http://fc09.deviantart.net/fs23/f/2008/003/7/d/Wall_Texture_by_shadowh3.jpg

Szerző: <http://shadowh3.deviantart.com/>

[6] Név: *Verseny fű textúra*, Fájl: „./teszt/Road/Texture/grass.BMP”

Forrás: http://www.dynamuse.com/wp-content/uploads/2010/03/images/Grass_texture%20167915.gif

Szerző: Ismeretlen

[7] Név: *Verseny út textúra*, Fájl: „./teszt/Road/Texture/road.BMP”

Forrás: <http://storage3d.com/storage/2007.11/dc537c3ea06bf3f721ff178a87a692dd.jpg>

Szerző: Ismeretlen

[8] Név: *Verseny skybox textúra*, Fájl: „./teszt/Skybox/Texture/skybox.BMP”

Forrás: <http://www.3delyvisions.com/sky8.zip>

Szerző: <http://creativecommons.org/licenses/by/3.0/>

[9] Név: *Viper*, Fájl: „./teszt/Viper/viper.3DS”

Forrás: <http://thefree3dmodels.com/stuff/vehicles/viper/13-1-0-14>

Szerző: <http://thefree3dmodels.com/index/8-1>