Általános információk

A diplomaterv szerkezete:

1. Diplomaterv feladatkiírás
2. Címoldal
3. Tartalomjegyzék
4. A diplomatervező nyilatkozata az önálló munkáról és az elektronikus adatok kezeléséről
5. Tartalmi összefoglaló magyarul és angolul
6. Bevezetés: a feladat értelmezése, a tervezés célja, a feladat indokoltsága, a diplomaterv felépítésének rövid összefoglalása
7. A feladatkiírás pontosítása és részletes elemzése
8. Előzmények (irodalomkutatás, hasonló alkotások), az ezekből levonható következtetések
9. A tervezés részletes leírása, a döntési lehetőségek értékelése és a választott megoldások indoklása
10. A megtervezett műszaki alkotás értékelése, kritikai elemzése, továbbfejlesztési lehetőségek
11. Esetleges köszönetnyilvánítások
12. Részletesés pontos irodalomjegyzék
13. Függelék(ek)

Felhasználható a következő oldaltól kezdődő Diplomaterv sablon dokumentum tartalma. Ügyeljen a tanszék, a hallgató, a konzulens nevét és a beadás évét jelölő szövegdobozokra, mert azokra külön ki kell adni a frissítést. A mezők tartalma a sablonban a dokumentum adatlapja alapján automatikusan kerül kitöltésre (Fájl/Információ/Tulajdonságok/Speciális tulajdonságok).

A diplomaterv szabványos méretű A4-es lapokra kerüljön. Az oldalak tükörmargóval készüljenek (mindenhol 2.5cm, baloldalon 1cm-es kötéssel). Az alapértelmezett betűkészlet a 12 pontos Times New Roman, másfeles sorközzel.

Minden oldalon - az első négy szerkezeti elem kivételével - szerepelnie kell az oldalszámnak.

A fejezeteket decimális beosztással kell ellátni. Az ábrákat a megfelelő helyre be kell illeszteni, fejezetenként decimális számmal és kifejező címmel kell ellátni. A fejezeteket decimális aláosztással számozzuk, maximálisan 3 aláosztás mélységben (pl. 2.3.4.1.). Az ábrákat, táblázatokat és képleteket célszerű fejezetenként külön számozni (pl. 2.4. ábra, 4.2 táblázat vagy képletnél (3.2)). A fejezetcímeket igazítsuk balra, a normál szövegnél viszont használjunk sorkiegyenlítést. Az ábrákat, táblázatokat és a hozzájuk tartozó címet igazítsuk középre. A cím a jelölt rész alatt helyezkedjen el.

A képeket lehetőleg rajzoló programmal készítsék el, az egyenleteket egyenlet-szerkesztő segítségével írják le.

Az irodalomjegyzék szövegközi hivatkozása történhet a Harvard-rendszerben (a szerző és az évszám megadásával) vagy sorszámozva. A teljes lista névsor szerinti sorrendben a szöveg végén szerepeljen (sorszámozott irodalmi hivatkozások esetén hivatkozási sorrendben). A szakirodalmi források címeit azonban mindig az eredeti nyelven kell megadni, esetleg zárójelben a fordítással. A listában szereplő valamennyi publikációra hivatkozni kell a szövegben. Minden publikáció a szerzők után a következő adatok szerepelnek: folyóirat cikkeknél a pontos cím, a folyóirat címe, évfolyam, szám, oldalszám tól-ig. A folyóirat címeket csak akkor rövidítsük, ha azok nagyon közismertek vagy nagyon hosszúak. Internet hivatkozások megadásakor fontos, hogy az elérési út előtt megadjuk az oldal tulajdonosát és tartalmát (mivel a link egy idő után akár elérhetetlenné is válhat), valamint az elérés időpontját.

Fontos:

* a szakdolgozat készítő/diplomatervező nyilatkozata (a jelen sablonban szereplő szövegtartalommal) kötelező előírás Karunkon, ennek hiányában a szakdolgozat/diplomaterv nem bírálható és nem védhető!
* mind a dolgozat, mind a melléklet maximálisan 15 MB méretű lehet!

Jó munkát, sikeres szakdolgozat készítést ill. diplomatervezést kívánunk!

FELADATKIÍRÁS

A hallgató feladatának a következőkre kell kiterjednie:

* Különböző zajfüggvények megismerése és az adott feladatra alkalmas zaj kiválasztása
* Terepadat generálása és tárolása textúrában a kiválasztott zajfüggvény kiértékelésével
* A terep háromdimenziós háromszöghálós megjelenítése
* Dinamikus LoD (Level of Detail) rendszer bevezetése
* A pálya feltextúrázása különböző szabályok szerint (pl: meredek rész – kő, dombos rész – fű, adott magasság felett hó)
* Animált, hullámzó víz megjelenítése
* Ha a feladat megkívánja, fák és vagy kövek megjelenítése
* Mindezen programkomponensek vezérlésére alkalmas UI megjelenítése
* A program teljesítményének analizálása



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Irányítástechnika és Informatika Tanszék

Pecze Dávid

Procedurális terep generálása és megjelenítése

Konzulens

Dr. Szécsi László

BUDAPEST, 2022

Tartalomjegyzék

[Összefoglaló 6](#_Toc119758809)

[Abstract 7](#_Toc119758810)

[1 Bevezetés 8](#_Toc119758811)

[1.1 Formázási tudnivalók 8](#_Toc119758812)

[1.1.1 Címsorok 8](#_Toc119758813)

[1.1.2 Képek 8](#_Toc119758814)

[1.1.3 Kódrészletek 8](#_Toc119758815)

[1.1.4 Irodalomjegyzék 8](#_Toc119758816)

[2 Utolsó simítások 10](#_Toc119758817)

[Irodalomjegyzék 11](#_Toc119758818)

[Függelék 12](#_Toc119758819)

Hallgatói nyilatkozat

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

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) 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, 2022. 11. 21.

...…………………………………………….

Összefoglaló

Virtuális terep generálása és megjelenítése régóta és rendszeresen jelentkező feladat, főként 3D számítógépes játékoknál. Bejárható játéktér nélkül, igazából nincs számítógépes játék.

Pályát egyrészt lehet úgy alkotni, hogy egy modellező kézzel megtervezi és elkészíti a 3D modellt egy arra alkalmas programban, majd importálja az adott játékmotorba. Ennek előnye, hogy pontosan és precízen lehet kialakítani a virtuális teret az adott játékhoz, viszont idő- és erőforrásigényes.

A megoldás másik módja, hogy algoritmusok segítségével generáljuk, nem feltétlenül az egész világot, de legalább a terepet, amit bejárhat a játékos. Előnye, hogy ha egyszer meg van írva a generáló program, akkor végtelenféle világot lehet vele generálni és nincs szükség egy vagy több grafikus kreátorra, akik hosszas időn át modelleznék le.

A szakdolgozat célja az utóbbi módszer felhasználásával terep generálása és megjelenítése hatékony módon. Probléma ott merülhet fel, ha nagyobb pályáról beszélünk, ahol több ezer egységnyire lévő dombokat is lehet látni. Ilyenkor sok feleslegesen nagy csúcspontsűrűségű geometriát jelenítünk meg, erőforrást pazarolva, amit el lehetne használni máshol, például a közelebb lévő elemek részletesebb megjelenítésére. Erre megoldást ad egy dinamikus Level of Detail módszer, ami által, a terep háromszöghálójának a felbontása a virtuális kamerától való távolságtól függ. A GPU csővezeték tartalmaz erre használatos elemeket, mint például a tesszellációs árnyaló. Sikeres implementáció azt eredményezi, hogy a távolban lévő geometriákat a GPU akár jelentősen kevesebb csúcspont felhasználásával jeleníti meg, több FPS-t eredményezve, melynek köszönhetően gyengébb hardveren is el tud futni a program.

Abstract

The generation and rendering of virtual terrain has been a common occurrence in the past decades, especially in 3D computer games. Without a traversable map, there is no computer game.

A map can be made by a 3D modeler (a person), by hand, in a specific application, then exported into the game engine. The advantage of this approach is, that the person doing this can customize the model to a very specific, low level, but that usually takes a long time and consumes a lot more resources.

The other way of doing this is by using an algorithm to generate the geometry. Not necessarily the whole map with every object, but maybe the ground terrain. Even that can speed up the creation process by a significant amount, assuming the algorithm is written. That is the advantage of this approach. If it is written and is moderately customizable with parameters, then it can be used as many times as desirable to achieve the wanted look and feel of the world.

The goal of my thesis is to implement the latter variation and also write a little engine that can render it efficiently. It can be efficient in a way so that it renders the farther part of the terrain in less detail, this way we can spare resources. It means, that the map of the game can be way bigger. The reference point (the origin) is the camera position. This process is called dynamic Level of Detail. The modern GPU pipeline contains an element, that is specifically made for this, called the tessellation shader. After a successful implementation we should see that the hills, that are far away are being rendered by using a lot less vertices, compared to the ones next to the camera, resulting in more FPS. This means that the game can run on older hardware.

# Bevezetés

Mindig is szerettem játszani különböző RPG (szerepjáték) játékokkal, melyek nagy hangsúlyt helyeztek a pálya kinézetére és annak immerzivitására, melyet sokszor úgy értek el a készítők, hogy az abban a korban készült játékokhoz képet jelentősen nagyobb pályákat alkottak meg ezzel is emulálva a való világ méreteit. Egy idő után viszont nem éri meg kézzel megtervezni mindent, mert az túl sok idő és erőforrásba telne. Számos játék a procedurális generálást választotta megoldásként erre problémára, ilyen például a Minecraft, Terraria, Astroneer, Starbound, Binding of Isaac és még sok hasonló.

## Motiváció

Kifejezetten érdekel, hogy valójában mi módon oldják meg a pályák generálását. A szép az egészben számomra az, hogy pár paraméter átírásával jelentősen más pályát lehet generálni ugyanazzal a kóddal. Ez a változatosság az, ami életben tarthatja a játékot, hiszen a manuálisan elkészített pályák nagyságára, számosságára erős időbeli határok vonatkoznak, mindegy milyen nagy csapat dolgozik rajta. Egy idő után pedig repetitív lesz ugyanaz a tartalom, mindegy milyen mekkora. Ha viszont futás közben, mondjuk akárhányszor új szintre lép át a játékos egy teljesen új pályán játszhat, akkor az a változatosság lehet az a faktor, ami miatt folytatni fogja a játékot.

Erre nagyon jó példa a Binding of Isaac nevű játék. Ott minden szint más az előzőnél, általánosságban hasonlítanak a pályák, de a szobák elosztása és a benne található szörnyek mások. Ami közös mondjuk, hogy minden szinten van egy kincses szoba és egy főellenség, akit le kell győzni a következő szinthez való átlépéshez, de ez az ellenség minden szinten mondjuk 10 közül bármi lehet, a kincses szobában több tucat felhasználható tárgy közül fog egy darab várni a játékosra. Ennek köszönhetően emberi szemmel nézve végtelen féle játékmenetünk lehet, nem lesz 2 ugyanolyan, melyhez nagyban hozzájárul a folyton változó pálya.

## Saját procedurális világ

Amennyire szép is jó, az előbb említett játék 2 dimenziós, felülnézetes. Úgy gondoltam látványosabb, ha 3 dimenziós tereppel dolgozok. Itt tud értelmet nyerni a Level of Detail rendszer. Ha felülnézetes a játék nem láthatunk el a távolba, viszont a jól megszokott 3 dimenziós, belső nézetes játékokban igen. Az én alkalmazásom nem a játék részére fókuszál, hanem kizárólag a pályagenerálásra és annak megjelenítésére. Nem implementáltam fizikai szimulációt szóval úgymond sétálni nem lehet a generált terepen, de átrepülni rajta azt lehet. Célom az volt, hogy gombnyomásra tudjak generálni különböző hegyes vagy dombos tájakat, állítható paraméterektől függően. Mindezt grafikus kezelőfelületen át, futásidő közben. A dinamikus Level of Detail módszer alkalmazásával el lehet érni, hogy gyengébb képeken is jól fusson a program nagy világgal. Véleményem szerint a világ nagynak számít, ha mondjuk minden irányban 3 kilométerre el lehet látni és el is lehet menni.

# Felhasznált technológiák

A projekt elején nagy dilemmát okozott, hogy használjak-e egy meglévő játékmotort vagy írjak egy sajátot Hosszas mérlegelés után úgy döntöttem, hogy írok egy sajátot, mivel ezzel sokkal többet fogok tanulni, funkciók szempontjából, pedig közel sem kell elérnem egy létező játékmotor szintjét, sőt inkább a megjelenítés részre kell fókuszálnom. Inkább megjelenítő motornak hívnám a programom, mint játékmotornak. Ha használtam volna egy létezőt, akkor a **Unity**-re esett volna a választás, mivel azzal már dolgoztam korábban és a tény, hogy C#-ban lehet programot írni benne, biztos segítette volna a fejlesztést. Ám ehelyett saját motorra esett a választás, amit C++-ban írtam meg.

## Választott nyelv

Mint írtam a választott nyelv C++. Nincs benne garbage collection, amit fel tudok hozni érvnek és ellenérvnek is. Mivel nincs, gyorsabban fut egy C++-ban írt program, viszont nagyon oda kell figyelni a memóriaszivárgásokra. Emellett tudtommal a játék és grafikus iparban igencsak használt (és többnyire kedvelt) nyelv. Jó 2 éve programozok benne, szóval ismerem a nyelvet annyira, hogy magabiztosan meg tudjam benne írni a munkámat.

## Ablak és bemenet kezelése

Windows-ra fejlesztettem, szóval rendelkezésemre állt a Win32 API egésze. Utólag visszatekintve nem lett volna sokkal több munka nyers Win32 API hívások köré felépíteni saját osztályokat és keretrendszert, de nem ez volt a projekt célja így úgy döntöttem, hogy a **GLFW** nevű nyílt forráskódú könyvárat fogom használni az ablak és bemenet kezelésére. Van OpenGL és Vulkan implementációja is, könnyen megérthető, egyszerűen használható.

## Grafikus API

A számítógépes grafika tárgynak köszönhetően ismerem az **OpenGL**-t és az önálló laborra készült ruhaszimulációs programhoz is OpenGL-t használtam a geometria megjelenítéséhez. Már ismertem a **C++ API-ját,** sőt kipróbáltam a Java-s verziót is **(LWJGL 3)** is, úgy gondolom kellően ismerem. Kevesebb váratlan hiba jöhet elő fejlesztés közben más grafikus API-khoz képest, mint például a Vulkan vagy DirectX.

A lineáris algebra gyakori előfordulása miatt szükségem volt egy könyvtárra, amivel vektor és mátrix műveleteket tudok végezni. A **glm** tökéletes erre a célra, mert működése a **GLSL**-en alapul.

## Grafikus kezelőfelület

Ahhoz, hogy kényelmesen lehessen fejleszteni és használni a programot, szükség van valamiféle UI-hoz, hogy ne azt kelljen csinálni, hogy átírunk egy paramétert kódból, újra fordítunk, megnézzük az eredményt, bezárjuk, átírjuk kicsit másra és így tovább. Helyette futásidőben lehet kényelmesen változtatni vektorok, mátrixot vagy akármilyen egyéb változók értékét, plusz nem szükséges a forráskód a paraméterek átírásához.

Az **ImGui**-t számos játékfejlesztő cég használja belsős eszközök fejlesztésekor. Rendkívül gyorsan össze lehet rakni egy grafikus felületes, az immediate mode felépítse miatt. Ugyan minden frame-ben újra rajzolódik, erősen optimalizált, nem fogja vissza a program teljesítményét.

Amit emellett használtam még, inkább a 3D modellek betöltésének tesztelésekor, az az **ImGuizmo**, mely az ImGui-ra építve lehetővé teszi a transzformációs mátrixok interaktív változtatását, **gizmo**-k használatával, melyek kirajzolása kompatibilis az ImGui kódjával.

## Egyéb függőségek

A képek beolvasására az **stb** GitHub repository-ból használtam az stb\_image.h fájlt. Fejlesztés közben, a konzolra való formázott logging végett pedig az **spdlog** könyvtárat használtam, később pedig beleépítettem egy ImGui ablakba a kiírást. A zajt pedig, amíg nem saját kóddal generáltam, addig egy **PerlinNoise** nevű egy header-ből álló könyvtárat használtam. Ez nem használta a GPU-t, így lassú volt, de amíg a pálya megjelenítésével foglalkoztam, teljesen megfelelt.

A projekt fájlok legenerálásához **Premake**-et használtam.

Igyekeztem úgy megválasztani a felhasznált külső könyvtárakat, hogy ami nem a feladatom része, azt nem próbálom meg újra feltalálni és megírni.

# Terv megalkotása

Fontosnak tartottam, hogy ne gondolkodás nélkül essek neki a fejlesztésnek. Minden projekthez kell egy terv, ami alapján folyni fog a fejlesztés, ezt a szakmai gyakorlat alatt is megtapasztaltam. A kívánt végeredményt kell a szemünk elé helyezni, és visszafele haladva, kisebb feladatokra bontani.

**TODO: valami vakítás még**

# Projekt kialakítása

Amit tudtam, hogy Windows-ra fogok fejleszteni C++-ban, Visual Studio-t használva. Szerettem volna úgy összerakni a projektet, hogy bárki, aki letölti GitHub-ról gond nélkül tudja fordítani és futtatni. Ezt nehéz úgy megcsinálni, hogy a solution fájlt előre legenerálom. Premake segítségével írhatok egy szkriptet **Lua**-ban, aminek futtatásával lehet generálni Visual Studio solution és projekt fájlokat, be lehet állítani a konfigurációkat, mint például Debug vagy Release és lehet adni build utána utasításokat, mondjuk, hogy build után másolja át az assets mappa tartalmát a build mappába. A Dist konfig ezt csinálja nálam. Kódrészlet a Lua szkriptből:

workspace "OpenGLWorkspace"

architecture "x64"

startproject "OpenGL-Engine"

configurations

{

"Debug",

"Release",

"Dist"

}

project "OpenGL-Engine"

location "OpenGL-Engine"

language "C++"

kind "ConsoleApp"

cppdialect "C++17"

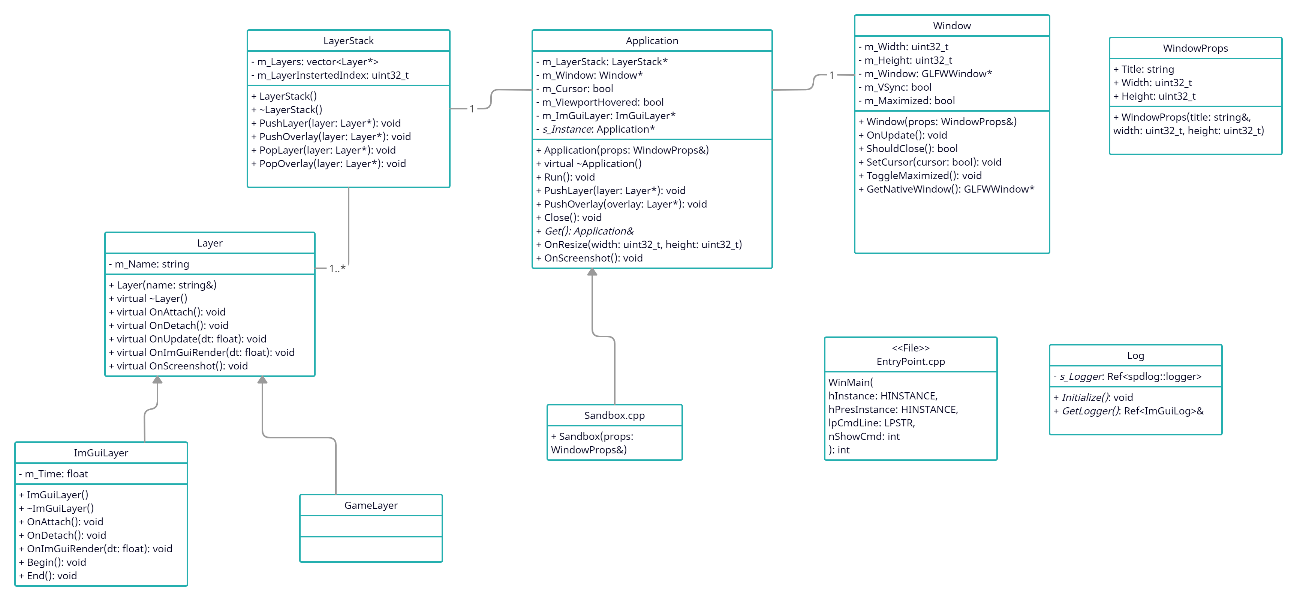
staticruntime "on"

entrypoint "WinMainCRTStartup"

Ez a kód része a szkriptnek ami létrehoz egy OpenGLWorkspace nevű solution fájlt, benne egy OpenGL-Engine nevű projektet és azt rakja alapértelmezett projektnek. Ugyanebben a fájlban lehet linkelést beállítani dll-ekhez, hozzáadni az include könyvtárakhoz.

# Az applikáció struktúrája

Próbáltam minél egyszerűbben felépíteni úgy, hogy közben az alap funkciók kényelmesen elérhetőek legyenek a kódból. Itt főként a bemenetre gondolok. Egy korábbi projektem mintájára a következő struktúrája jutottam:



1. ábra Kezdetleges UML diagram

https://app.creately.com/d/6O0WSGEV1aI/edit/s/keXbZrbnVzn

A lelke mindennek az Application osztály, az tartja számon az ablakot, a különböző rétegeket. Singleton mintát követ, egy egyed létezik belőle a program futása során, ezt bárhol elérjük kódban, amin át elérhetjük az ablakot is.

A Log statikus függvényeken keresztül, melyeket szintén bárhol használhatunk inicializálás után. A kód esztétikája miatt bevezettem C++ macro-kat, hogy ne kelljen minden esetben kiírni, hogy Log::GetLogger()->trace(…), helyette elég azt kiírni, hogy TRACE(…)

#define TRACE(...) Log::GetLogger()->trace(\_\_VA\_ARGS\_\_)

A főbb logika a GameLayer-ben valósul meg, ott tárolom a textúrákat, modelleket, árnyalókat.

Az ImGuiLayer felelős az ImGui inicializálásáért, kirajzolásáért. Logikai megkülönböztetést tettem overlay és layer között, de mind a kettő egy Layer osztály egyede. Az én feladatom egészében 2 darab ilyen réteg van jelen, a GameLayer és az ImGuiLayer. Nagyobb applikáció esetén jó ötlet, ha logika szerint rétegekre bontjuk az alkalmazást. A fenti ábrán nem az OpenGL részen van a hangsúly.

Igyekeztem betartani egy egységes konvenciót a kódra vonatkozóan, hogy jól olvasható legyen a kód. Konkrétan ezeket, először **The Cherno** videóiban láttam. Tetszettek, mivel nem túlzás egyik szabály sem, ám logikusak és gyorsabban lehet tőle kódot olvasni. Ezek a következők:

* Camel case használata mindenhol
* Függvények nevét nagy kezdőbetűvel írom láthatóságától függetlenül
* Kapcsos zárójel mindig új sorban van
* Osztályok adattagjainak neve mindig egy „m\_” prefix-el kezdődik, jelölve, hogy „member” változó, ha statikus, akkor „s\_”. Ez akkor jó, ha függvényen belül szerepel ez a változó, akkor egyből látszik, hogy nem egy lokális, hanem osztályváltozó
* Struct-ok adattagjainál pedig nem használok semmi prefixet, egyből nagy betűvel kezdem a változó nevét

# CPU zaj

Miután volt egy működő programom, amiben tudtam logolni, és alap geometriát megjeleníteni, azt tűztem ki célnak, hogy egy egyszerű négyzet csúcspontjainak a pozícióját manipuláljam CPU kódból generált zaj értékeivel.

# GPU zaj

# Dinamikus Level of Detail

# Hullámok szimulációja

# Az összeállt program

# Teljesítmény mérése

# Összegzés

# Utolsó simítások

Miután elkészültünk a dokumentációval, ne felejtsük el a következő lépéseket:

* Kereszthivatkozások frissítése: miután kijelöltük a teljes szöveget (Ctrl+A), nyomjuk meg az F9 billentyűt, és a Word frissíti az összes kereszthivatkozást. Ilyenkor ellenőrizzük, hogy nem jelent-e meg valahol a "Hiba! A könyvjelző nem létezik." szöveg.
* Dokumentum tulajdonságok megadása: a dokumentumhoz tartozó meta adatok kitöltése (szerző, cím, kulcsszavak stb.). Erre való a Dokumentum tulajdonságai panel, mely a Fájl / Információ / Tulajdonságok / Dokumentumpanel megjelenítése úton érhető el.
* Kinézet ellenőrzése PDF-ben: a legjobb teszt a végén, ha PDF-et készítünk a dokumentumból, és azt leellenőrizzük.

Irodalomjegyzék

1. Levendovszky, J., Jereb, L., Elek, Zs., Vesztergombi, Gy.: Adaptive statistical algorithms in network reliability analysis, Performance Evaluation - Elsevier, Vol. 48, 2002, pp. 225-236
2. National Istruments: LabVIEW grafikus fejlesztői környezet leírása, <http://www.ni.com/> (2010. nov.)
3. Fowler, M.: UML Distilled, 3rd edition, ISBN 0-321-19368-7, Addison-Wesley, 2004
4. Wikipedia: Evaluation strategy, <http://en.wikipedia.org/wiki/Evaluation_strategy> (revision 18:11, 31 July 2012)

Függelék