NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK Hallgató neve: **Horváth Levente Péter**
2021 Hallgató törzskönyvi száma: **T/004475/FFI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Horváth Levente Péter**
Törzskönyvi száma: T/004475/FFI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

SRTM adatok 3D-s megjelenítése és a megjelenítés optimalizálási lehetőségei

3D Visualization of SRTM data and the optimization opportunities of the visualization

Intézményi konzulens: Dr. habil Szénási Sándor
Külső konzulens:

Beadási határidő: 2022. május15.

A záróvizsga tárgyai: Számítógép architektúrák
Szoftvertervezés és –fejlesztés
specializáció

A feladat

Tervezzen meg és implementáljon egy olyan alkalmazást, ami képes az SRTM (Shuttle Radar Topology Mission) során kinyert magassági adatokból három-dimenziós megjelenítésre! Ehhez vizsgálja meg a hasonló fejlesztéseket, illetve a megjelenítés során a szakirodalomban használatos algoritmusokat! Ezek alapján tervezze meg az elkészítendő alkalmazást, válassza ki a használandó módszereket!
Valósítsa meg a megtervezett rendszert a választott programozási nyelv és keretrendszerek segítségével! A rendszer legyen testreszabható (textúrák), illetve optimalizálja a megjelenítést level-of-detail és occlusion-detection technikák segítségével! Tesztekkel igazolja a program működését, ezek alapján értékelje az elkészült munkát!

A dolgozatnak tartalmaznia kell:

- a feladat leírását,
- a meglévő rendszerek bemutatását,
- az alkalmazott eszközök, technológiák és eljárások bemutatását,
- a megvalósítandó feladat tervét,
- a tesztelés folyamatát és a teszteredményeket,
- eredmények bemutatását és értékelését,
- az architektúrában rejlő továbbfejlesztési lehetőségeket,
- a dokumentációt, valamint a rendszert bemutató prezentációt CD mellékleten.

Ph.

.....
Dr. Vámossy Zoltán
intézetigazgató

A szakdolgozat elévülésének határideje: **2024. május 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat / diplomamunka saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban / diplomamunkámban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2022 május 11.

hallgató aláírása

Tartalomjegyzék

Célmeghatározás

Szakdolgozatom célja 3-dimenziós vizualizáció alkalmazása a Nasa Shuttle Radar Topography Mission (SRTM) projektje során keletkezett magassági adatainak felhasználásával. A rendelkezésre álló adatokat az Endeavour űrsikló gyűjtötte be 2000 februárjában és a kezdeti 90 méteres mintavételezésű adatokat a sokkal részletesebb 30 méteres,1 ívmásodpercű magassági koordinátákkal tették szabadon felhasználhatóvá 2015-ben más forrásokból kitöltve, pótolva az esetleges hézagokat, hibás vagy adatokat nem tartalmazó területeket. A megjelenítés során alkalmazott textúrákat az SRTM letöltés során a projekthez rendelkezésre bocsátott földrajzi színskálával ellátott képekből, valamint egyéb műholdas felvételekből összeállított forrásból állítom össze.

A forrásadatok feldolgozása során az SRTM-ből kinyert, kiegészített adatmennyiség a textúrákkal együtt nagyméretűvé válik, amit különböző optimalizálási módszerekkel lehet és kell a számítógép erőforrásai számára - memória, grafikus kártya - majdnem ideális mennyiségben átadni.

Jelenleg az SRTM adatokat számos helyen használják, a térkép szoftverektől (OpenStreetMap) a számítógépes játékokig és ahogy elterjedtek a 3D-s technikák úgy lett igény a domborzati, magassági adatok felhasználására és azok minél pontosabb begyűjtésére. Emiatt is bocsátották útjára a lézeres radar magasságmérési projektet, hogy minél élethűbb megjelenítést tegyenek lehetővé és megbízhatóbb, pontosabb adatokra tudjanak támaszkodni. Szakdolgozatomban ezeket a pontosabb adatokat használom fel a lokalizált tartalom megjelenítéséhez.

Szakdolgozatomban ennek a feldolgozásnak és optimalizálásnak a felhasználásával szeretném elérni, hogy az adott lehetőségekhez képest csak a szükséges erőforrásokat felhasználva jelenjenek meg a térképrészletek a kiválasztott nézettel együtt. A vizualizáció során nagy mennyiségű adatot kell a grafikus kártyának, operációs rendszernek kezelnie. A folyamatos, gördülékeny megjelenítéshez a domborzati elemek közötti navigálás során biztosítani kell az optimális adatmennyiséget, így tartható fenn a folyamatosság látványa, akadozásmentesség bárhova közelít a térképen a felhasználó.

Nasa SRTM misszió

A NASA (National Aeronautics and Space Administration) SRTM missziója nemzetközi összefogás keretében valósult meg 2000 februárjában (ESA Earth Online, 2020). Több ország űrügynöksége, térinformatikai szervezete dolgozott együtt. A Nasa szolgáltatta az Endeavour űrsiklót, amivel a radarberendezéseket juttatták föld körüli pályára, valamint a duális, kétfajta radar közül az SIR-C mérőeszközt, a német és olasz ügynökség pedig az X-SAR-t. A két radar az űrsikló rakodóterében, illetve az abból kinyúló 60 méterre kitolt antennaárbócon helyezkedtek el (SRTM, 2020). A radarok az interferometria elvén működtek, a feltérképezendő területről két képet készítettek különböző nézőpontból és az eltérésekből számították ki az adott magasságot.
A 2000-es misszió egy már 1994-ben történő újramérést végzett, azzal a főbb különbséggel, hogy ekkor már egy ívmásodpercű felbontást rögzítettek. Az összegyűjtött adatok lefedték a föld felületének majdnem 80 %-át, ezzel az első, majdnem teljes globális magassági adatbázist hoztak létre. A cél egy valósághű digitális magassági modell - angolul DEM (Digital Elevation Model) – létrehozása. A modern földtantudomány megkövetelte azt a gyakorlati szempontból nagyon fontos feltételt, hogy valós topográfiai adatokkal dolgozzanak a hidrológia során árvizek, földcsuszamlások vizsgálatánál, de a polgári, katonai felhasználás is előtérbe került, mint a repülés közben használatos navigációs rendszerek minél hatékonyabb támogatása, szárazföldi katonai műveletek pontosítása, időjárás előrejelzés, klíma zónák feltérképezése. Az addig létező térkép adatok nem mindig voltak megbízhatóak, minőségük megkérdőjelezhető volt és eltérő skálázást is mutattak. Egy egységes léptékű és felbontású adatbázis létrehozása elég költséges lett volna, repülőgéppel történő feltérképezés mind politikai mind földrajzi okokból nem volt kivitelezhető. Az 1990-es években kifejlesztett synthetic aperture radar technológiai demonstrációja volt ez a misszió. A mért adatok 1 négyzet ívmásodperc részletességűek, ami egy 30x30 méteres felületnek felel meg (Facility, 2014). A magassági hibahatár 6 méter alatt maradt. A nem amerikai területekhez 3 ívmásodperc részletességű adatok elérhetőek. A teljes adatmennyiség elérte a 12,3 TB-ot, az adatok rögzítése többszörös sebességgel történt, mint az űrsikló központtal való átviteli sebessége így a küldetés alatt egy-egy területet csak ellenőrzés céljából töltöttek le (Center, 2011).

SRTM adatok

A szakdolgozatomban felhasznált SRTM adatokat a publikusan elérhető és a magas minőségű és felbontású adatokat elérhetőségét elősegítő OpenTopography szervezet által rendelkezésre bocsátott forrásokból építem fel. (OpenTopography, 2020)

Az adatokat több formában is le lehet tölteni GeoTIFF, IMG, szakdolgozatomban és a projektmunkám során az Arc ASCII formátumút használom. Ez egy térképszoftver használata nélkül is feldolgozható szöveges file, amiben a magassági adatok sorokban és oszlopokban szerepelnek, rendkívül megkönnyítve ezzel a háromdimenziós képalkotáshoz felépíthető koordináta rendszert és annak értékeit. Magyarország határait befoglaló régió kiválasztásával egy körülbelül 8530 oszlopos és 3611 soros mátrixban tölthetjük le a magassági értékeket, szélességi és magassági adatok nélkül. A háromdimenziós megjelenítéshez háromszögeket kell kirajzolnia a vizualizációs programnak. A teljes Magyarországot lefedő SRTM adatokkal több mint 61.5 millió ilyen háromszöget lehetne megjeleníteni, minden magassági értékből a hozzá szomszédos magassági értékhez köthető és kialakítható egy háromszög struktúra, amit a grafikus program tud kezelni és a képernyőre vetíteni.

Számítógépes 3D-s vizualizáció

A számítógépes képi és mozgó megjelenítés alap technológiája a vizualizált objektum modellje és annak textúrája. Egy tárgy elhelyezkedését meg lehet adni pontjainak koordinátáival, ha veszünk egy kockát annak csúcsaival és térbeli távolságaival, elhelyezhető egy adott háromdimenziós térben. Egy összetetteb tárgynál például egy játékbábúnál vagy egy felszín esetében sokkal több csúcsra van szükség. A modell megalkotásához, a jellegzetesebb főbb pontokat kell kiválasztani és azokat összekötve háromszögekből kell felépíteni a modellt. Az így kapott háromszögekből fognak összeállni a bemeneti csúcsadatok, amit a számítógépes technológia tud kezelni. (Evanson, 2019)

Minden egyes háromszöghöz hozzárendelnek egy textúrát, vagy textúra részt, ezzel öltöztetik fel úgymond az adott részt a modellből és adnak a váznak egy tetszőleges kinézetet. A kép mozgatása, forgatása során matematikai számítással jelenítik meg a látható és előtérben lévő háromszögeket, a textúrák is aszerint változnak, amerre a kép mozdul. A modellhez hozzákapcsolt háromszögekhez többféle textúra tartozhat, aszerint, hogy milyen távolinak, közelinek látszódjon az objektum, illetve effektus információkat is tartalmazhat például a fény visszaverődéséét az adott felületről. A háromdimenziós megjelenítés világában minden egy textúra és koordináták gyűjteményéből áll (Evanson, How 3D Game Rendering Works: Vertex Processing, 2019). Bufferek, tárolók segítségével kötik őket össze egymáshoz a programban. A vertex-buffer tartalmazza a csúcskoordinátákat, az index-buffer-ben helyezkedik el az a sorrend, ami szerint a térbeli pontok összekapcsolódnak és háromszögeket alkotnak. A megjelenítendő forma, háromdimenziós képek függvényében és ezzel együtt az adatmennyiségtől függően vagy az egész jelenet adatait tárolja a program a memóriában, vagy nagy területeket lefedő, nyílt-világú szoftverek esetében annak csak bizonyos részleteit, aszerint, hogy hol helyezkedik el a kamera által látott helyszín. Minden egyes jelenet során számos matematika mátrix

transzformáció kerül végrehajtásra az objektumokon a képen szereplő tárgyakon, így követi le a grafikus motor a távolságot, kamera pozíció váltást (Tomas Akenine-Möller, 2020).

3D technológiák

A következő alfejezetekben különböző application programming interface-ket, specifikációkat fogok röviden bemutatni, hogy mire használható a feladat szempontjából. Előnyöket és hátrányokat is leírok a szakmai felhasználásuk során.

OpenGL

Az OpenGL egy nyílt specifikáció, leírja és tartalmazza az általános grafikus parancsokat különböző alkalmazásokhoz. Az OpenGL megjelenése óta a legszélesebb körben használt grafikus fejlesztői könyvtár mind a kétdimenziós, mind a háromdimenziós alkalmazások terén. Számos platformra teszi lehetővé a fejlesztést. Az OpenGL könyvtárak összegyűjtik a renderelési, textúra mapping, speciális effektusok és egyéb vizuális funkciókat. A fejlesztőknek hatékonyan segít a népszerű asztali és akár munkaállomás platformokra történő szoftverek létrehozásában. Ezekbe beletartoznak egészségügyi képalkotó eljárások, 3D animációk, építészeti ArchiCAD szimulációk, de a játékipar és a manapság egyre népszerűbb virtuális valóságot használó alkalmazások is képezik a gerincét az OpenGL felhasználásának. Az OpenGL előnyei közé tartoznak a következők:

- Ipari szabvány: egy független konzorcium irányítja a specifikációt, az OpenGL Architecture Review Board, valamint az OpenGL egy szabad, szállító semleges multiplatform szabvány, ami iparilag széleskörűen támogatott
- Stabilitás: több éve működnek különböző platformokon és visszamenőlegesen kompatibilis az előző verziókkal, így a már lefejlesztett alkalmazások nem avulnak el.
- Megbízhatóság és portolhatóság: egységes vizuális teljesítményt nyújt bármelyik OpenGL API-t futtatni képes hardveren, operációs rendszertől függetlenül.
- Fejlődő: a folyamatos hardver fejlesztésekhez is elérhetővé válnak az API-k, lehetővé téve, hogy a szoftver és hardver fejlesztők implementálhassák megoldásaikat a termék fejlesztési ciklusaikban.
- Skálázhatóság: az OpenGL-t használó alkalmazások sokféle rendszeren képesek futni a laptopokon át a szuperszámítógépekig így bármelyik géposztályra skálázhatóak a fejlesztők választása alapján.
- Könnyű kezelhetőség: jól strukturált a maga logikai környezetében és a parancsok is egyértelműek, kevesebb sornyi kód írása szükséges az alkalmazásokhoz.

Hardver információkkal is rendelkezik így felszabadítja a fejlesztőt a hardver specifikus jellegzetességektől, amik plusz munkát jelentenének.
- Dokumentáció: gazdag könyvtár listával rendelkezik a technológia leírása, példakódok is rendelkezésre állnak az információk könnyű megértéséhez.

Az OpenGL rutinok leegyszerűsítik a grafikus szoftver fejlesztését, a rendereléstől, a grafikus primitívek – pont, vonal, poligon – használatán át a fényeffektusokig és textúrák kezeléséig. Az OpenGL programnyelvi megkötései, hogy C, C++, Fortran, Java nyelven érhetőek el. Az összes OpenGL implementáció specifikációs és nyelvi dokumentációval kell, hogy rendelkezzen, amiknek alkalmazási teszteken is meg kell felelniük.

A gyártók külön OpenGL implementációt készíthetnek saját hardvereikre, így az asztali számítógépektől kezdve a munkaállomásokon át a szuperszámítógépekig terjedhet a grafikus gyorsítás hardveres lehetősége. A kiterjesztések, angolul extension-ök használatával differenciálhatják termékeiket, így további teljesítménybeli és technológiai innovációkat tudnak használni a szoftverfejlesztők. 2006-tól az OpenGL Architecture Review Board a Khronos Group konzorcium alá tartozó OpenGL Working Group-pá alakult. (The Khronos Group, 2020)

OpenGL szakmai felhasználása

Mint a legelső grafikus könyvtár széleskörben alkalmazható szinte bármilyen vizuális munkára, platform függetlensége miatt iPhone-ra megírt szoftvert könnyen át lehet Androidra portolni például, Windows és Unix operációs rendszerek közötti átmenet is lehetséges. A netes böngészőkben futtatható WebGL (Web Graphics Library) is kezd egyre elterjedni - ami JavaScript program interface-en keresztül működik egyéb plug-inek nélkül - a HTML5 kódban megírt weblapoknál, ami a felhasználó hardverét felhasználva alkalmaz gyorsítást a tartalom böngészésekor (Mozilla, 2020).

OpenGL funkciók

Az OpenGL-nek van egy magja, az alapokat ez tartalmazza, ezek a sima gl-el kezdődő függvények, library komponensek. A következőkben néhány elemi példa a funkciólistából (The Kronos Group, 2020):
- glVertex{234}{sifd}{v}(…) – vertexeket, csúcsokat specifikál, 2,3,4 dimenzióban, egész vagy lebegőpontos számokkal
- glBegin(GLenum mode) – körülhatárolja a primitív objektum vagy objektum csoport csúcsait, csúcspontjait. A GLenum módjai határozzák meg, miként kezelje a csúcsokat, mint például GL_LINES, GL_TRIANGLES, GL_TRIANGLE_STRIP, 10 darabot lehet ezekből megadni.

A glu-ban az u-betű mint utility suffix, utótag jelenik meg, a gyakran használt OpenGL eljárások, függvények kombinációját takarja, ami az alapokat magába foglalva egy magasabb szintű kirajzoló rutint nyújt az alap OpenGL-nél (The Kronos Group, 2020)

- gluLookat(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble aimx, GLdouble aimy, GLdouble aimz, GLdouble upx, GLdouble upy, GLdouble upz) – a view matrix-ot határozza meg, a kezdő látószöget itt adhatjuk meg, ahonnan a szemünk elhelyezkedne (eyex, eyey, eyez), hozzáadva az "ahova nézünk" referencia koordinátákat (aimx, aimy, aimz), és a nézőpont függőleges vektorát (upx, upy, upz).

A glut az ablakkezelést és egyéb más működéseket foglalja magába. Az OpenGL Utility Toolkit (GLUT) egy egyszerű ablakkezelő API ami operációs rendszer független, létrehozható a segítségével több ablak, méretezhetően, tulajdonságaikat módosíthatja (The Kronos Group, 2020). Általános 3D-s objektumokat az alapvető primitívekkel, wireframe-eket tudunk rutinszerűen létrehozni vele, de a billentyűzetről, egérről jövő inputokat is képes feldolgozni. Használatával sokkal egyszerűbb kódot lehet írni, nem kell ismerni az operációs rendszer specifikus ablakkezelő API-kat

A GLUT toolketben az összes függvény „glut" prefixel kezdődik, pár beszédes és gyakran használt kezelő funkciók

glutReshapeFunc(void (*func)(int width, int height)) – ablak méretezésénél meghívott függvény

glutKeyboardFunc(void (*func)(unsigned char key, int x, int y) – billentyűzet leütés eseményt kezeli, a char key az ASCII érték a paraméter listában, az x és y az egér aktuális helyzetét fogadja.

glutMouseFunc(void (*func)(int button, int state, int x, int y)) - egér eseményeket kezeli, a gomb paraméter lehet GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON vagy a GLUT_RIGHT_BUTTON .Az állapot, a state lehet GLUT_UP vagy GLUT_DOWN ami a kattintás folyamatát adja meg, elkezdve és lent tartva, vagy eleresztve. Az x és y az egér aktuális helyzetét fogadja.

Vulkan

A Vulkan egy újgenerációs grafikus interface, ami hatékonyabb és keresztplatformos hozzáférést nyújt a modern grafikus kártyákhoz az asztali számítógépektől vagy konzoloktól a mobiltelefonokig vagy beágyazott rendszerekig (The Kronos Group, 2020). A Khronos Group hozta létre és fejleszti tovább. Az API többszálúság központú így jobban ki tudja használni a modernebb többmagos processzorokat és a párhuzamos programozáson keresztül nyújt hozzáférést a grafikus kártyákhoz. 2016-ban jelent meg az 1.0-ás

verzió és az AMD Mantle API-ját vették át elég sok esetben egy az egyben. Az első játék a The Talos Principle volt, ahol az API debutált, A Vulkan-t fejlesztették tovább és szánták a következő generációs OpenGL-nek (The Kronos Group, 2016).

Vulkan szakmai felhasználása.

A párhuzamosítás és a CPU/GPU közötti egyensúly megtalálása komoly teljesítménybeli előnyöket hozott a Vulkan alkalmazásakor. Szoftver és hardver független, az Apple Metal és a Windows DirectX API-jához képest ez egy multiplatformos grafikus függvénytár, Androidon,Windowson,Linuxon is fut Nvidia, Amd, Intel és Qualcomm GPU-kon. Viszont a hardver központúbb fejlesztés miatt a fejlesztőknek többletmunkát ad a korábban az APIk által automatikusan elvégzett feladatokhoz. Kevesebb kézen fogva vezetést ad a használata és sokkal komplexebb. Ellenben a jobb teljesítmény miatt az alkalmazások sokkal gyorsabbak lettek, a teljes kontroll az alacsony hardverközeliség miatt az API felett a fejlesztőnél marad, csökkenti a CPU és GPU driver overheadet. Játékoknál még magasabb frame-per-second értékeket kaptak, mint korábban. A Vulkan tartalmazza már az Nvidia ray tracing technológiáját, a sugárkövetést, ami rendkívül látványos eredményt produkál belső és külső terek megvilágításához, tükröződések valósághű megjelenítésében, árnyékok követésében (Overvoorde, 2019).

Unity

A Unity egy grafikus motor és keretrendszer nem csak kettő és három-dimenziós alkalmazások készítéséhez, hanem egyéb művészi animációkat, szimulációkat, tervezési (autóipari, építészeti) munkákat is képesek vagyunk a segítségével elvégezni. Nem csak kód szinten tudunk kommunikálni a projektünkkel, hanem user-interface-n keresztül, látványelemeket kiválasztva is tudunk szerkeszteni és fejleszteni. Több mint 20 platformra a Unity is képes exportálni. Az Asset Store-ból tudunk komponenseket telepíteni, modelleket, hanghatásokat, de akár scripteket is amit egyszerűen drag-and-drop módszerrel hozzáadhatunk.

Unity szakmai felhasználása

Unityt tudunk C#, JavaScript nyelven is programozni, ebben írjuk meg a játék logikát és vezéreljük az objektumokat, jeleneteket. Visual Studio integrálással is rendelkezik de MonoDevelop-ban fejleszthetünk vele, ez az alap IDE (Petty, 2020). Maga a motor C++ alapú. A Unity keretrendszeren belül tesztelhetjük, futtathatjuk a projektünket, ekkor a kód Mono-n, NET Frameworkon fog futni. Unity alatt majdnem minden tárgy a jelenetben egy GameObject ősosztályból származik le, ez az alapja a három-dimenziós modellnek, a fényeknek. (Murphy, 2020) A Unity keretrendszer jellege miatt nem egy három-

dimenziós forma, modell generáló szoftver (Game-Ace Creative Studio, 2020). Habár sok mintát le lehet tölteni a modellekhez, ezek létrehozásához más szoftvert kell használni. Létezik harmadik féltől származó plugin, amivel ez lehetséges a Unity-n belül, de nem része az alap szoftvernek, kivételt a beépített felszíni, terület, tájkép kialakítási funkció jelent. A Unityt fejlett beépített grafikus motort automatikusan kezelő funkcióval látták el így a memória kezelés, garbage collection a C++-al ellentétben nem a fejlesztő feladata. Könnyebben lehet prototípust alkotni, kevesebb buggal de kevesebb kontrollt is ad a fejlesztő kezébe.

OpenTK

Az OpenTK egy alacsonyszintű, fejlett, gyors, C#-ban programozható OpenGL, OpenCL és OpenAL-hez hozzáférést nyújtó grafikus könyvtár (FrederikJA, 2020). A C++-ban megírt OpenGL-nek nincs közvetlen C# nyelvi támogatottsága, ezért fejlesztették az OpenTK-t. A Tao Framework-öt váltotta fel és helyettesíti, ami egy korábbi OpenGL-hez kötődő C# könyvtár .NET és Mono fejlesztőknek. Megfelel játékok, tudományos alkalmazások és más projektek számára, amiknek három-dimenziós, audió és számítási feladatokra van szüksége.

OpenTK szakmai felhasználása

Ezt a grafikus könyvtárat is számos dologra lehet használni, matematikai, lineáris algebrai csomaggal is el van látva, ablakkezelővel és perifériákról jövő bemeneti információkat is tud kezelni (FrederikJA, Projects using OpenTK, dátum nélk.). Az OpenTK nem egy játékfejlesztői motor, hanem egy könyvtárat nyújt ami a fejlesztő döntésére bízza, hogy hogyan használja fel őket. Könnyen integrálható .NET-es alkalmazásokkal és szabadon felhasználható, nincsen jogdíjhoz kötve. Fejlesztés során az OpenGL natív C-API-jához képest saját enumerációt határoz meg. A C-API-ban nincsenek obejktumok, csak funkciók és a memória. Minden állapot, parameter egy nagy enumeráció, aminek a függvények, funkciók használatakor a leírásra kell hagyatkozni. Nincs indikáció, melyik állapot a megfelelő, és az intellisense sem nyújt segítséget. Ezt a problémát hivatott az OpenTK saját enumerációja orvosolni, az OpenGL-ben a “void glBegin(int primitiveState);” függvényben a primitiveState-nek például az “enum PrimitiveType { GL_TRIANGLE, GL_QUAD, GL_FAN };” a megfelelője, ezt lehet használni az OpenTK “void glBegin(PrimitiveType primitiveState);” funkcióban. A Visual Studio intellisense felismeri, hogy itt a függvény argumentuma egy PrimitiveType típus és eszerint adja meg a releváns felugró lehetőséget.

Az OpenTK technológiai működése a három-dimenziós vizualizáció során

Ebben a fejezetben a választott grafikus megjelenítőnek a működéséről próbálok meg átadni minél több alapinformációt a nyilvános segédanyagok (FrederikJA, LearnOpenTK, 2020) felhasználásával, annak általános felépítését írom le, pár lényeges funkcionalitását részletezem a projektemmel kapcsolatban. A tanulási folyamatomban megszerzett tudást kapcsolom a tulajdonságok leírásához.

OpenTK és OpenGL

Az OpenTK mint korábban említettem az OpenGL application interface-re épül, ami számos funkciót magában foglal, hogy vizuális és kepi elemeket manipuláljunk. Maga az OpenGL önmagában viszont nem egy API, inkább egy specifikáció, ami konkretizálja, hogy a funkcióknak mi az elvárt kimenete és hogy hogyan kellene működnie. Ez a specifikáció viszont nem ad implementációs részleteket, így minden fejlesztés, ami eleget tesz a leírásoknak megengedett. A legtöbb esetben a videókártya gyártók maguk fejlesztik az OpenGL könyvtárakat, a hardver folyamatos fejlődése miatt. A korábbi immediate-modeban az OpenGL funkcionalitások el voltak rejtve, de idővel a fejlesztők egyre több kontrollt akartak szerezni ennek működése fölött. Ezt a módot könnyeb lehetett használni, de nagyon nem hatékony. Emiatt vezették ki a 3.2-es verzióból a sokat elvonatkoztatott immediate-mode-ot aminek háttérműködését nehéz volt megérteni és léptették a helyébe a core-profile módot ami a modern megközelítés. Ez viszont megköveteli, hogy valóban értsük a grafikus programozást és habár ezt is nehéz tanulni, mégis több rugalmasságot nyújt, a teljesítményre is nagyobb hatással lehet lenni és nem utolsó sorban jobban érthetővé teszi a programozást. Szakdolgozatomban az OpenTK a 3.3-as verziójú OpenGL core-profile szabványt használja. A jelenlegi OpenGL ennél jóval magasabb verziójú 4.5, viszont ennek alapjaira épültek a további extra kiegészítések, anélkül, hogy megváltoztatták volna azt. Az újabb verziók hasznosabb és hatékonyabb eljárásokat vezettek be ugyanazokhoz a feladatokhoz. A legújabb OpenGL funkciókat csak a legújabb videókártyagyártók eszközeivel tudjuk kihasználni, ezért érdemes egy alacsonyabb szintű verziót használni és feltételesen használni a magasabb verziójú funkciókat. A kiegészítésekkel a grafikával foglalkozó cégek esetleges új eljárásait és technikáit implementálják az OpenGL-be, ha azok nagyon népszerűek és hasznosnak bizonyulnak. A kiegészítéseket a 3.3-as verzióban ritkán használjuk, de ha ezzel hatékonyabban tudunk megvalósítani dolgokat akkor ahhoz megfelelő leírás található.

Az OpenGL működése állapotgépnek tekinthető, a kontextust változtatjuk a funkciók meghívásával, ha háromszög helyett vonalat akarunk kirajzoltatni akkor a kontextus változókat módosítjuk és aszerint jelennek meg az adott objektumok. Az állapot használó funkciók segítségével pedig különböző műveleteket hajtunk végre az állapotnak

megfelelően. Maga az OpenGL egy nagy állapotgépnek tekinthető, eszerint sokkal könnyebb megérteni a működését. A C nyelven megírt könyvtáraknak a sokféle magas-szintű programnyelvre való származtatása nem a leghatékonyabb, számos elvonatkoztatással fejlesztették ezeket, egyik közülük ilyen az OpenGL objektumok. Ezek nem a klasszikus objektum orientált nyelvekben megfeleltethető objektumok, mivel ezek a grafikus kártyán találhatóak és számokkal vannak képviselve, amiket az objektumot használó függvényeknek átadunk. Általánosan először egy objektumot hozunk létre és egy hivatkozást tárolunk el róla, a valódi objektum így a háttérben marad, majd ezt az objektumot a cél kontextushoz kötjük és amikor nincs már rá szükségünk eltávolítjuk a kontextusból, azzal, hogy az őt reprezentáló objektum ID-t nullára állítjuk. Az objektumon beállított tulajdonságok ettől még megmaradnak és vissza tudjuk állítani amint a célkontextusnak a korábban kinullázott ID helyére megadjuk. Így az objektumok egy beállítások kollekciójaként is felfogható, az OpenGL állapotának egy részhalmaza. Számos objektumból épülhet fel így az alkalmazásunk, állapotait mi magunk állíthatjuk be és köthetjük a kontextushoz. Három-dimenziós tárgyak modell adatait definiálhatjuk, például egy papírrepülőét és amikor fel akarjuk használni a megjelenítéshez, egyszerűen csak hozzákötjük a kirajzoláshoz.

Az OpenTK-t Visual Studioban a NuGet csomagkezelőből tudjuk letölteni. Jelenleg ez a 4.0.0-ás verzió. A program kódban az OpenTK saját ablakkezelőjével tudunk dolgozni, az OpenTK alkönyvtárait a “using OpenTK” után tudjuk megadni, többek között a Graphics, Input, Graphics.OpenGL4-et. Maga a Graphics névtér az OpenGL implementáció, a Graphics.ES30 pedig az OpenGL Embedded System 3.0-ás verzióját tartalmazza. Meg tudjuk hívni az OpenGL 4.0-ás metódusokat a Graphics. OpenGL4-el, a billentyűzetről bejövő adatokat az Input osztályai kezelik. A hanghatásokárt értelemszerűen az OpenAL mint audio library felel. Az OpenTK saját ablakkezelőjének segítségével be tudjuk annak méretét állítani, címét a fejlécben és a futtatásához használatos “Run()” függvényen megadható az elérni kívánt frame per second érték. Ehhez a GameWindow osztályból kell azt leszármaztatni. Üresen hagyva a hardver által maximálisan elérhető képkocka fog kirajzolódni. A programot így lebuildelve egy üres ablakot fogunk még csak kapni, semmiféle objektumot még nem definiáltunk benne. A GameWindow osztály metódusaiból az összeset felül tudjuk írni, ha azt akarjuk, hogy az escape billentyű megnyomására befejeződjön az alkalmazásunk akkor a képkocka frissítés metódusba (void OnUpdateFrame(FrameEventArgs e)) kell egy billentyűzet állapot lekérést beépíteni (KeyboardState input = Keyboard.GetState();) és amint egy ilyen esemény bekövetkezik akkor az Exit() vagy Close() metódus lekapcsolja az OpenTK-t.

Grafikus pipeline

Az OpenGL-ben a három-dimenziós térben igazából mindennek két-dimenziósan kell a képernyő miatt megjelennie, így az OpenGL legfőbb munkája ezeknek a

transzformációknak az elvégzése. Ezt nevezik grafikus pipeline-nak aminek a két fő része a három-dimenziós-s objektumok két-dimenzióba átalakítása valamint a két-dimenziós koordináták az aktuális pixeleknek való megfeleltetése. Ezeknek a lépéseknek több közbenső fázisa van és mindegyiknek sorban szüksége van az előző kimenetére. Mindegyiknek különböző funkciója van és nagyon specializáltak és párhuzamosan is elvégezhetőek. Emiatt a grafikus kártyákba egy-két ezer grafikus mag van beépítve és egyéb más dedikált feladatot ellátó egység, amik elvégzik ezeket a számítási feladatokat, ezeket a programokat hívják shadereknek. Ezek a shader programok a GPU-n futnak így jelentős CPU időt tudunk velük megtakarítani, valamint helyettesíthetőek a fejlesztő által megadott más konfigurációval, a pipeline egyes részei felett így kontrollt gyakorolhatunk, a shadereket az OpenGL Shading nyelvben írjuk meg (GLSL). A grafikus pipeline hat lépésből álló elvonatkoztatását a következő részekből állíthatjuk fel:

- Vertex shader: ebben a szakaszban a koordináta-rendszerbeli csúcsok kerülnek feldolgozásra
- Shape assembly: a csúcsokból itt épülnek fel háromszögek
- Geometry shader: ez egy opcionális lépés, további finom-hangolásokat tudunk itt elvégezni
- Rasterization: a háromszögeket fragmentekké (a pixelek kirajzolásához szükséges adat) konvertálja át ez a lépés
- Fragment shader: a pixelek színeit, textúrákat, fény effekteket ebben a lépésben lehet formálni.
- Tests and blending: a fragment shader kimenetét integrálja a jelenet maradék, többi részével.

Az ablak megjelenése, inicializálása előtt egy egyszer lefutó OnLoad() funkciót kell meghívni amiben a törlést kell elhelyezni, a GL.ClearColor négy lebegő pontos argumentumot fogad, ez állítja be a színét a képkockák kirajzolása között. Az OnRenderFrame-ben a a képernyőtörlés GL.Clear szerepel először, kirajzolásnál az OnLoad funkcióban megadott értéket állítja be, majd a Context.SwapBuffers fut le. A kontextus ebben az esetben dupla-bufferelléssel történik, egyik esetben megjelenít, a másikban kirajzol. A csere meghívásánál ez a kettő kicserélődik és felváltja egymást. Létezik szimpla bufferelés is de ekkor hibák léphetnek fel, a kép szakadozhat. Képernyő átméretezésénél az OnResize fut le, ezt felülírva a Viewport metódusban – ami a kamera részletezésénél lesz megemlítve – kerül feldolgozásra a képernyő aktuális mérete a normalizált eszköz koordinátákhoz. A képernyőn való megjelenítéshez először bemeneti csúcs adatokat kell átadni az objektumról, x, y, z koordináták szerint
. Az OpenGL-nek csak a -1.0 és 1.0 közötti értékeket lehet átadni, az ebben az intervallumban megadott objektumot fogja feldolgozni és kirajzolni. Ezek a koordináták az úgynevezett normalizált eszköz koordináták, (Normalized Device Coordinates (NDC)). Egy háromszögnek a koordinátái ezek szerint ebben a formában lehet megadni: float[] vertices = {-0.8f, -0.8f, 0.0f,0.8f, -0.8f, 0.0f,0.0f, 0.8f, 0.0f}; . A háromdimenziós koordináták

csúcsai sorrendben: bal alsó, jobb alsó és felső csúcs. A mélységi koordinátákat most nem használjuk, viszont meg kell adni ebben az esetben is őket. Egy egyszerű papírrepülőgép három-dimenziós koordinátáit ez a tömb alkotja: readonly float[] paper_plane_vertices={-1.0f,0.0f,0.0f,0.8f,0.0f,-0.3f,0.8f,0.2f,0.0f,0.8f,- 0.2f,0.0f1.0f,1.0f,0.0f,1.0f,-1.0f,0.0f};. Ha felülről képzeljük el a modellt akkor a repülőgép orrát, farok részét, a törzs hátsó jobb és bal részét, ahol a szárnyak becsatlakoznak, valamint a jobb és bal szárnyvéget tudjuk hozzájuk kapcsolni. A koordinátákat a -1.0 és +1.0 intervallumban kell tehát megadni az objektumok elhelyezkedéséhez és nem a szokványos képernyő koordináták szerint, ami .NET, C#-ban a bal felső saroktól indul. Ezek az NDC koordinátákkal leírt objektumok fognak transzformálódni az adott jelenetet vizulizáló tér-koordinátákhoz. A koordináta rendszerekről később lesz említés.

Tárolók

Az NDC adatok ezután a grafikus folyamat első lépcsőjébe kerülnek, a vertex shaderek dolgozzák fel GPU memóriájában. Ezt a memóriát vertex buffer object-eknek nevezzük (VBO), kihasználjuk, hogy nagy adatmennyiségnyi koordinátákat küldünk egyszerre a hardver felé és nem kell az esetleg CPU által feldolgozott adatokat továbbítani ebbe a memóriába, ami elég költséges. Ez a vertex buffer object az első OpenGL objektum, amit egy egész szám típusú azonosítóval tudunk generálni: int VertexBufferObject; VertexBufferObject = GL.GenBuffer(); a buffertípusa pedig BufferTarget.Arraybuffer. A tárolót ezután a tároló tömbbhöz kötjük: GL.BindBuffer (BufferTarget.ArrayBuffer, VertexBufferObject); utána pedig a csúcsokat tartalmazó adatokat töltjük a tárolóba a BufferData funkcióval, ami a felhasználó által definiált adatokat másolja be az éppen aktuálisan kötődő tárolóba: GL.BufferData(BufferTarget.ArrayBuffer, vertices.Length * sizeof(float), vertices, BufferUsageHint.StaticDraw). A funkció első argumentuma a tároló típusa, a második az adat mérete, a harmadik maga az adat. Az utolsó paraméterrel megadhatjuk, hogy az adott információ hogyan fog viselkedni a GPU-n. A viselkedését, statikusnak adhatjuk meg, ekkor nagy valószínűség szerint nem fog változni, dinamikusnak tűntethetjük fel, vagyis sokat fog változni, illetve streamként is megadhatjuk, ebben az esetben minden alkalommal fog változni amikor kirajzolódik. Az utolsó két esetben az adatot memóriába helyezi el a grafikus kártya a gyors írás miatt. A program végén a tárolókat ki kell üríteni, ekkor nullát kötünk hozzájuk GL.BindBuffer(BufferTarget.ArrayBuffer, 0); és az objektumot töröljük: GL.DeleteBuffer(VertexBufferObject);.

A grafikus folyamat kezdetén így már rendelkezünk a modell adatiaval így a pipeline első szakaszában létre kell hoznunk egy vertex és egy fragment shadert. Ez a kettő programozható a fejlesztő által. Egy shader.vert filenak az első sorában meg kell adni annak verzióját, csakúgy mint a C nyelvben. A bemeneti változót annak helyével adjuk meg és az in kulcsszóval adjuk meg annak típusát, mint egy három-dimenziós vector és utána elnevezzük: layout (location = 0) in vec3 aPosition;. A shader fileban határozzuk meg a

main függvényt, a program belépési pontját, itt adhatjuk meg a feldolgozó műveleteket. A vektor végső helyét egy beépített gl_Position változóval adjuk meg, ennek a négydimenziós típusnak feleltetjük meg egy konverzió segítségével. gl_Position = vec4(aPosition, 1.0);. Ez az egyik legegyszerűbb vertex shader mivel semmiféle feldolgozást nem definiálunk benne. Ami akkor lenne szükséges, ha az adatok nem NDC-ben lennének. A fragment shader a másik shader file (shader.frag) amit létre kell hozni a végső kirajzoláshoz, itt számolja ki a program a pixel kimeneti színét. Ez a szín egy négy értéket tartalmazó vektor, a vörös-zöld-kék és áttetsző komponensekből áll össze. Ezt a filet is a verzióval kell kezdeni, #version 330 core és a main metódusban a FragColor-nak elnevezett négy értékű kimeneti vektornak ad megfelelő RGB és áttetszőség értéket: FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);. A shaderek futási időben fordulnak mivel elég sok tényezőtől függenek, elő feldolgozásra nincs lehetőség. A shader fileok feldolgozására külön shader osztályt definiál az OpenTK itt adható meg a konstruktorukban a file elérési útvonala, amit StreamReaderrel beolvasva adunk át az őket tartalmazó vertex vagy fragment shader source szöveges változónak (string VertexShaderSource;). A grafikus könyvtár segítségével definiáljuk az alkalmazásban, de előtte létre kell őket hozni és utána átadni nekik: VertexShader = GL.CreateShader(ShaderType.VertexShader); GL.ShaderSource(VertexShader, VertexShaderSource); A fordítást a CompileShader funkció végzi el GL.CompileShader(VertexShader); A statikusan most létrehozott objektumok a GPU-n való használatukhoz egy programot kell létrehozni, amit egy kezelőnek elnevezve és ahhoz hozzácsatolva tudunk implementálni: Handle = GL.CreateProgram(); ,GL.AttachShader(Handle, VertexShader);,GL.LinkProgram(Handle);. A handle ezután válik használható shader programmá. A fordulás után viszont a linkelés miatt már az eredeti shaderekre nincs szükség ezeket leválasztjuk a handle programról a detachshader paranccsal és töröljük is őket: GL.DetachShader (Handle, VertexShader);GL.DeleteShader(VertexShader);. A shader osztályunkban a kész handle-t kell használni, ekkor a UseProgram hívódik meg: GL.UseProgram(Handle); . Az osztályra nincs szükségünk annak megszűnése után így ezt a GL.DeleteProgram(handle)-el tudjuk kitisztítani a Dispose funkcióba ágyazva. A shader osztály működését felépítve az eredeti osztályban példányosítjuk és az OnLoad eljárásba elhelyezve fog működni.

A vertex shaderbe manuálisan kell megadni, hogy melyik bemeneti adat melyik vertex attributumhoz megy, direkt kell megmondani, hogyan kezelje a vertex adatot. Ehhez a GL.VertexAttribPointer funkció fog segíteni melynek bemeneti paraméterei sorrendben az,hogy melyik vertex attribútumot akarom konfigurálni, a shader fileban a lokáció során megadtuk,h a bemeneti vector az első, nulladik helyen legyen, ezért szerepel ott nulla. A második a vertex mérete, amit a típus követ. Negyedik helyen a normalizálásra vonatkozó utasítást adhatunk, ha egész számokkal reprezentált vertexeket adunk át akkor true, valós paraméterrel a koordináták az NDC tartomány (-1.0;1.0) szerint fognak működni. Az ötödik bemeneti érték a vertex attribútumok közötti lépték mértéke, három-dimenziós koordináták esetén ez három , mivel a sorvektorban minden x,y,z koordináta egyes tagjai

három egységre találhatóak egymástól. Az utolsó, amit meg kell adni, hogy a tárolóban hol kezdődik az adat. Egy lehetséges paraméterezés eszerint épül fel GL.VertexAttribPointer (0, 3, VertexAttribPointerType.Float, false, 3 * sizeof(float), 0); . Miután így megadtuk,hogyan kellene az OpenGL-nek fordítani a vertex adatokat engedélyezni kell a vertex attribútumot, úgy, hogy megadjuk annak helyét a GL.EnableVertexAttribArray(0); függvénnyel. Ebben az állapotban minden rendelkezésre áll a kirajzolásra, a tárolóban lévő csúcs adatokat inicializáltuk, megszerkesztettük a shadereket és utasítottuk az OpenGL-t,hogyan kösse össze a vertex adatokat a vertex attribútumokkal. A shadert ezután már csak használni kell és kirajzoltatni.
Viszont ezt mindaddig meg kéne ismételni ahány objektumot ki akarunk rajzolni, nem ritka azonban a több száz vagy ezer objektumos vizualizációk jelenleg így ez hatalmas hátrány lenne. Ezt áthidalni a vertex array object hivatott, ekkor az attribútum pointer konfigurálásánál csak egyszer kell azokat a hívásokat megtenni és bármikor az objektum kirajzolásakor a megfelelő VAO-hoz fogjuk kötni, ezzel lesz könnyebb a vertex adatok és attribútum konfigurációjuk közötti kapcsolat. A VAO-ban tároljuk el az állapotokat és ennek használatát az OpenGL megköveteli. A VAO tárolja a vertex attribútum tömb hívásokat, a pointer konfigurációkat és a hozzájuk érkező vertex buffer objektumokat és vertex attribútumokat. Ezt a VAO objektumot is létre kell hozni, szintén egy azonosítót fog kapni és hozzá kell csatolni a programhoz: int VertexArrayObject; VertexArrayObject = GL.GenVertexArray(); és GL.BindVertexArray(VertexArrayObject);. Maga a VAO tárolja a vertex attribútum konfigurációt és hogy melyik tárolót kell használni. Amikor több objektumot akarunk kirajzolni először ennek megfelelően az összes VAO-t kell létrehozni és a későbbi használatig eltárolni őket. A kirajzolás pillanatában a megfelelő VAO-t kapcsoljuk a megjelenítéshez és amint már nincs rá szükségünk leválasztjuk. Az OpenGL a DrawArrays funkcióval rajzolja ki a megadott primitíveket az aktuális shader segítségével. GL.DrawElements(PrimitiveType.Triangles, _Visegrad_indices.Length, DrawElementsType.UnsignedInt, 0);.

Technológiák összehasonlítása és a technológia kiválasztása

OpenTK melletti döntés a C# program nyelv és az alacsony szintű OpenGL könyvtárak konfigurálásának lehetősége miatt történt. Ezen a szinten tudom az optimalizálási algoritmusokat, számítási feladatokat lefejleszteni és nem egy grafikus UI-os interfaccel rendelkező keretrendszerben csúnyán mondva összekattintgatni, bár a produktivitás és az eredmény elérése szempontjából ez sokkal gyorsabb lenne. OpenGL-t a C++ nyelv nehézsége miatt vetettem el. A Unity annak modell alkotási hiányossága miatt nem volt szimpatikus, amit, habár pluginekkel lehetne pótolni, de elég magas szintű funkciókat nyújt és nem tudok a rendszerbe belenyúlni. Az egyetemen a C# a tárgyak fő

programnyelve így adta magát, hogy a C# szintű OpenGL átiratban fejlesszek. Így az alapjaitól tudok egy grafikus rendszerbe belefejleszteni és nem egy kész grafikus játékfejlesztői szoftverrel dolgozni.

Optimalizálás

A vizualizálás során több lehetőségünk van a rendszer teljesítményének optimalizálására, a minél hatékonyabb programkód írására, hogy ezáltal kevesebb erőforrást használjon fel a számítógép, ennek a két lehetséges módja a level of detail és az occlusion detection, ami beépíthető a grafikus alkalmazásba, ennek a rövid részletezése következik.

Level of Detail – LOD

A level of detail technika a kirajzoláskor szükséges megjelenítendő háromszögeket határozza meg, úgy, hogy a távolabb lévő objektumok komplexitását, a modelleket megalkotó háromszögek számát csökkenti rajtuk. Így a textúrák száma is csökken és a távoli képek minősége, térhatása is jobbá válik. A sziluettjük így változik, és közelről összehasonlítva a kevesebb háromszögből felépített modellt, egy távoli pontba elhelyezkedve ugyanolyannak fognak tűnni. A textúrák homályosításával is még jobban rá lehet játszani erre a hatásra. Lapos felületeknél egyértelműen sokkal kevesebb kirajzolás szükséges így. Az optimalizáláskor így a nagyon kis számú mikro háromszögeket tudjuk semlegesíteni. A LOD jelentős hatással van így a teljesítményre mivel kevesebb háromszöggel kell a programnak dolgozni, viszont jelentős memória overhead-del rendelkezik, mivel minden egyes szinthez meg kell adni a háromszögeket. Ökölszabályként a háromszögeket a felére szokták lecsökkenteni az egyes szinteken. Alkalmazásakor a megjeleníteni kívánt tárgy mérete és fontossága az adott jelenetben határozza meg milyen szintű LOD-okat alakítsunk ki. A túl kevés szint miatt nem jelentős a performancia, abban az esetben, ha nincs túl nagy különbség a polygon csökkentésben, illetve az átmenet az egyes szintek között észrevehető, fel vagy beugró, változó részletességű objektumok jelennek meg, autóverseny játékban például egy kanyarban a domborzatelemek, sziklák, fák csak hozzá közeledve jelenik meg. A túl sok szint viszont a CPU-ra rak nagyobb terhet, mivel sokat kell számolni, hogy mikor melyik legyen használatban, illetve a fileok tárolása is problémát jelenthet a megnövekedett részletesség miatt (Arm Limited, 2020).

Occlusion detection

A három-dimenziós térben több objektum esetén, ha fedésben vannak egymással a jelenetben, amit persze egy objektum esetén annak saját részei is megtehetnek, vagyis saját magát takarja el, akkor a nem látható részekre nincs szükség. Így felesleges erőforrást

vesz el a rendszertől. A modellek a térben a nézőpont szerint jelennek meg, a geometriai transzformációk és megjelenítési lépések során alakul ki a képernyőn feltűnő jelenet. Az occlusion detection ezekben a lépések között kell, hogy megtörténjen, kiszámolásra kerüljön. Amikor az objektumok fedésben vannak egymással vagy valamilyen arányban egymás részeivel aszerint kell a folyamatban limitálni a megjelenítendő objektumokat és csak a látható részek háromszögeit feldolgozni. Az occlusion detection ezt a számolási, detektálási folyamatot jelenti és az occlusion culling az elemek leválasztását. (Chandel & Vatta, 2020)

Tervezés

A munkám fejlesztése során Magyarország egy földrajzi területét három-dimenzióban megjelenítő alkalmazás a cél és annak minél optimálisabb működése. A felhasználó a forrás magassági SRTM adatokat adja meg a programnak, ez alapján készül el a három-dimenziós modell koordinátái, ebből építi fel az objektumot a grafikus szoftver. A felhasználó egy textúrát is megadhat, hogy az adott terület milyen felszínnel, megjelenéssel kerüljön a képernyőre. Az SRTM forrás letöltő oldalon lekérhető a magassági színezés szerinti textúra, ezt is lehet használni, de akár más kép is "ráhúzható", alkalmazható az adott domborzatra. A GoogleMaps-ből vagy OpenStreetMap-ből kivágott, nyilván a szélességi-hosszúsági adatokkal megegyező felülnézeti tájkép fotót alkalmazva valós látványt kapunk. Az eddig rendelkezésre álló funkciók tehát az SRTM adatokból három-dimenziós koordinátákat képező eljárás, a forrás SRTM input mellé lekérhető térképrészlet helyett valós textúra alkalmazása a megjelenítéshez. A program futása során a területet körbejárhatjuk, bármelyik oldalról, megnézhetjük, különböző kameraállásokból szemlélhetjük és aszerint változnak a méretbeli arányok, a domborzati viszonyoknak megfelelően a térkép egyes részei nem láthatóak, ez a jelenlegi működés röviden.

Jelenlegi funkciólista

- SRTM adatok feldolgozása három-dimenziós modell kialakításhoz
- SRTM forráshoz letölthető textúra helyettesítése valós térképpel
- Terület körbejárhatósága
- A kamera és az objektum távolsága szerinti arányok folyamatos változása a navigálás során, távoli és közeli látkép megjelenítése pozíciótól függően

Tervezett további funkciók

- File kiválasztással történő SRTM feldolgozás
- Textúra választás fileból
- A navigálás módosítása, a kamera lefelé való elmozdításának implementálása

- Level-of-detail optimalizálás fejlesztése
- Occlusion detection optimalizálás fejlesztése
- Saját domborzati textúra generálása a magassági koordináták szerint, az alacsony zöld és magas barna színek felcserélésének lehetősége
- Frame-per-second megjelenítése a képernyőn
- Memória használat kijelzése
- Terhelés mérése adat skálázódás hatására, SRTM input adatok növelésének hatása a performanciára

Elkészült példaalkalmazás a szakdolgozat I bemutató alkalmával

A második fő része az alkalmazásnak az SRTM nyers adatokat feldolgozó komponens mellett a grafikus megjelenítésért felelős osztályok a Game, Camera, Shader és Texture. Ezek az OpenTK keretrendszerből származnak.

Fejlesztési ütemterv

A fejlesztés további menete fontossági sorrendben az optimalizáló eljárások implementálása és a teljesítmény méréshez szükséges funkciók beépítése. A félév első egyharmadát ezzel tervezem eltölteni, 2021 január, február során.
A teljesítmény mérés beépítését és tesztelését márciusra datálnám, ekkor a már optimalizált rendszert lehet vizsgálni és összehasonlítani a jelenlegi működési és futási tulajdonságokkal.
A teljes rendszer tesztelését és a tesztesetek megírását áprilisra ütemezném
Amennyiben ezek működnek ezután fogok az angolosan “nice-to-have” vagyis “jó-havan” funkciókat fogom kidolgozni, mint a file kiválasztás, saját színű térkép generálás, ezekre, ha jut időm áprilisra, májusra datálnám ezt a munkafázist.

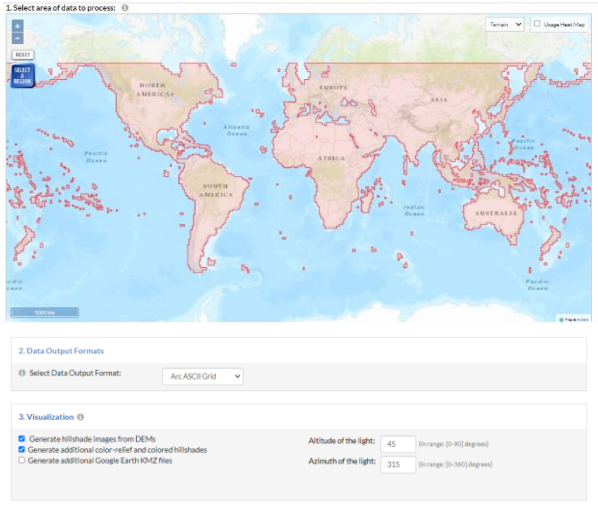
Elkészítendő alkalmazás komponensei

A jelenlegi alkalmazást kettő részre bontanám. Az egyik az SRTM szöveges fileokat feldolgozó osztály, az SRTM_Data_Process, ez a saját fejlesztés, három-dimenziós koordináták itt kerülnek kiszámolásra és adható át a grafikus keretrendszernek mint vertex adat. A másik az OpenTK ami a grafikus könyvtárakat tartalmazza a megjelenítéshez és a navigálás továbbfejelsztése és optimalizálás helye.

Implementáció

A megvalósított rendszer rövid bemutatása a felhasználó szempontjából

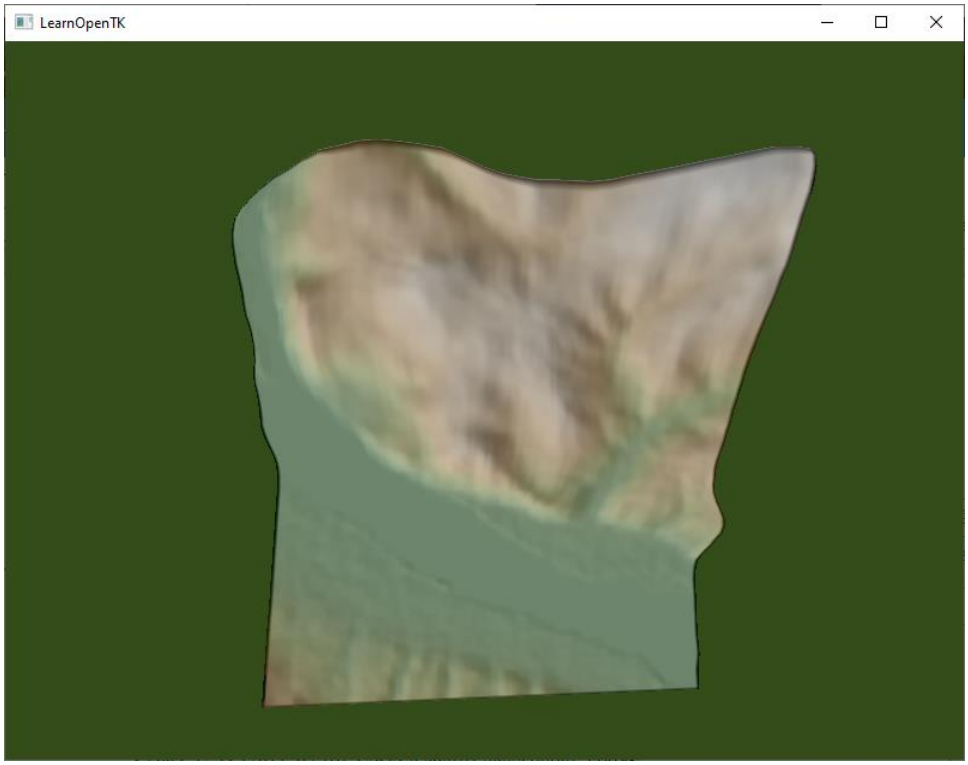
A programot a kiválasztás során értékelt OpenTK C# alapú grafikus megjelenítő keretrendszerben valósítottam meg. A program jelenleg egy előre letöltött magassági adatokat tartalmazó file-t dolgoz fel és jelenít meg, de tetszőlegesen letölthető hozzá bemeneti adat a kiválasztott területről az OpenTopography honlapjáról https://portal.opentopography.org/raster?opentopoID=OTSRTM.042013.4326.1
A kiválasztott régió mérete ne haladja meg a kb. 3x3 km-es méretet, az optimális jelenleg egy 64x31 méretű rács, amit letöltés után a filet- megnyitva le tudunk ellenőrizni. Az output típusánál az Arc ASCII Grid –et válasszuk ki és a vizualizációhoz bepipálhatjuk, hogy generáljon a rendszer a domborzatnak megfelelő színezetű textúrát. A program jelenleg nincs felkészítve ennél nagyobb adat átkonvertálására, feldolgozására, habár az ehhez szükséges függvények rendelkezésre állnak, ebben az esetben körülbelül 2000 háromszöget rajzol ki, ez a maximum jelenleg a betöltött egy objektumnál.



1. *Ábra, SRTM pubilkus adatok letöltési oldala az OpenTopography oldalról*

Az így kiválasztott és lementett magassági adatokat és textúrát a program bin\Debug almappájába kell másolni, valamint a keretrendszer Program.cs file Main metódusában kell rá hivatkozni az SRTM_Data_Process_OOP osztály konstruktorának megadva.

Az alkalmazás indítása után egy újabb ablakban történik a megjelenítés.



2. Ábra, A program induló képernyője az előre beállított adatok betöltése utána

A billentyűzet WSDA gombjaival tudjuk a térképadatokból felépített domborzatot bejárni előre, hátra, jobbra, balra mozdítva a perspektívát, illetve a bal shift és szóköz billentyűvel le vagy fel irányítható a kamera, az egér mozdulását is leköveti a rendszer, aszerint változik a kirajzolás.

Az implementált funkciók és eljárások bemutatása

A grafikus keretrendszer kezdő, bevezető használata mellé három hozzáadott funkciót implementáltam, amelyek mindegyike a megjelenítést befolyásolja. Ezeket a felhasználó egy adott billentyűkombinációval tud bekapcsolni és aszerint változik a kirajzolás. A változások, átmenetek szemléltetőbb megjelenítéséhez a ctrl+p paranccsal átváltott Polygon mode-ok közül a vonal nézetre célszerű átváltani és abban kipróbálni a módosításokat.

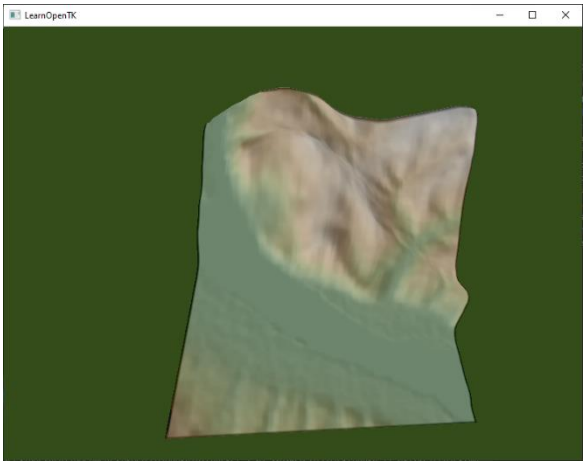
1. A domborzat mintavételezését, részletességét a a kamera pozíciójától függően dinamikusan változtató, adott területnek egy előre meghatározott gömb “térfogatán” belüli legrészletesebb adataiból felépített megjelenítés, a Level of Detail egyik implementációja. Aktiválása a Ctrl+o gombbal hívható elő.

2. A domborzat mintavételezését, részletességét a a kamera pozíciójától függően dinamikusan változtató, teljes térképrészletre alkalmazott legrészletesebb adatokból Lagrange interpolációval feldolgozott megjelenítés, a Level of Detail másik implementációja. Aktiválása a Ctrl+l gombbal hívható elő.

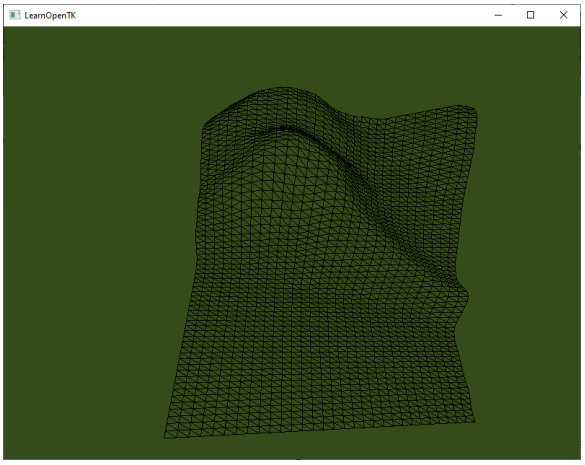
3. A domborzat mintavételezését, részletességét a kamera pozíciójától függően dinamikusan változtató, teljes térképrészletre alkalmazott legrészletesebb adatokból felépített megjelenítés oly módon, hogy a domborzat nem látható csúcsait elrejti. Ez egy kísérleti megoldás az occlusion detection egyik alternatív megoldására. Aktiválása a Ctrl+h gombbal hívható elő.

Az implementált funkciók és eljárások technikai bemutatása

Az első Level of Detail implementációt úgy sikerült leprogramoznom, hogy a keretrendszernek szükséges megadni egy csúcsokból álló float tömböt, amiben egy csúcshoz tartozó adatrészletnek tartalmaznia kell a három-dimenziós koordinátákat és a textúra adott ponthoz tartozó helyét, vagyis öt elem alkot ebben a formában egy csúcsot. Szükséges még emellé ezeknek a csúcsoknak, vagy a megjeleníteni kívánt csúcsoknak egy primitív típusát, egy triangle-strip-et kirajzoló, háromszögeket megadó sorrendjét megadni szintén egy tömbben. Az így megadott modell paraméterei kerülnek feldolgozásra a keretrendszerben és jelenik meg a képernyőn. Erre a modellre ezen az alacsony szinten a tömbök módosításával tudok változást generálni. A funkció aktiválásakor az előre beégetett kamera pozícióból figyeljük a jelenetet, a Level of Detail szemléletesebb bemutatására a ctrl+p billetnyűkombinációval átváltott PolygonMode,vonallal kitöltött megjelenítésével láthatjuk, hogy hogyan változik a kirenderelt objektum részletessége ahogy közelítünk a felszínhez.

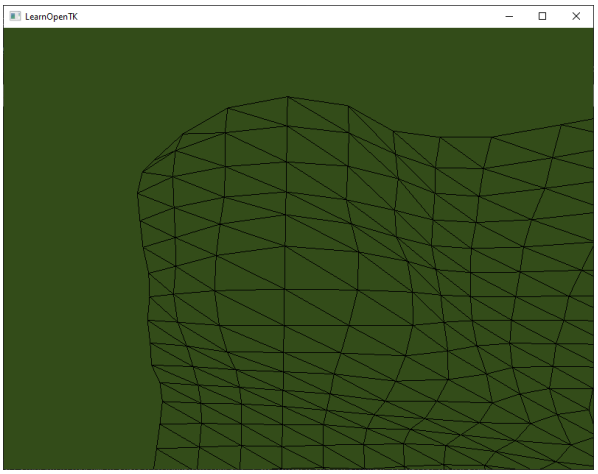


3. Ábra, Kis mértékű navigálás után a bejárás állapota

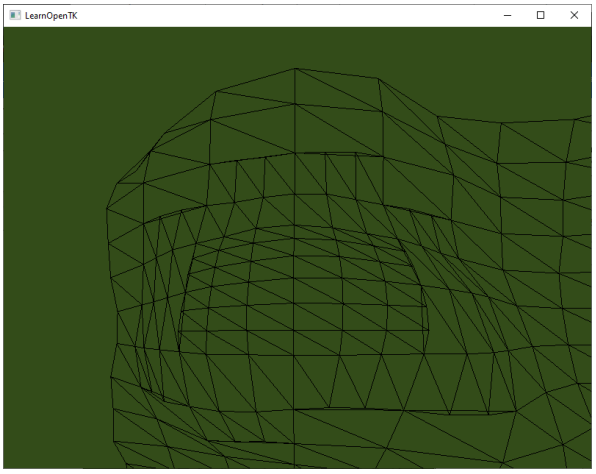


4. Ábra, Wireframe mode bekapcsolása utáni megjelenítés

Amint a ctr+o kombinációval átváltunk az első level of details implementációra, az irányító billentyűkkel a domborzathoz közelebb navigálva, a leütéskor a program kiszámol egy fixen beépített gömb térfogatot, aminek a középpontja a kamera pozíciója és ellenőrzi, hogy a kétdimenziós tömbben tárolt magasságok közül melyiket foglalja éppen magába. Ez egy segéd tömbbe íródik be és ez alapján kerülnek kiszámolásra az új megjelenítendő csúcspontok és a hozzájuk tartozó triangle-strip háromszögek sorrendjének létrehozása. 4x4-es mintavételezést állapítottam meg a kezdeti kirajzoláshoz, ez lesz egyre részletesebb. Ha a gömb térfogatán kívül eső csúcsokat talál a feldolgozás egy 4x4-es maszkban a kétdimenziós tömbben, és az összes 16 csúcsot kívül esőnek számolja akkor x és y irányba a negyedik csúcsot számolja, mint figyelembe veendő pont és kerül további lépésre a ciklus a következő 16 elemmel amíg a tömb végére nem ér. Ha a 16 elem mindegyike benne foglaltatik a térfogatban akkor az összes magassági csúcspontot hozzáfűzi a megjelenítendő pontokhoz és aszerint inkrementálja a háromszögek kirajzolásához szükséges sorrendet. Ha a 16 elem közül akár egy is de nem az összes már beleér a gömb térfogatába akkor 2x4-es méretű háromszögekből építi fel a jelenetet.



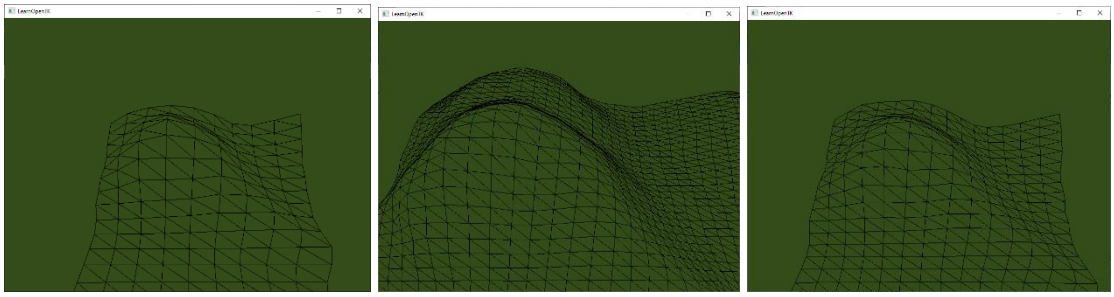
5. Ábra, Magas sorszámú LoD során kirajzolt triangle-strip



6. Ábra, Közelítésre részletesebb feldolgozás után kirajzolt triangle-strip

Az így kigenerált új adatok sűrűbben, részletesebben adják vissza az adott felszínt. A level of detail elve szerint így a háromdimenziós megjelenítésben egy adott objektumot vagy annak egy részét csak közelebbről a kamera pozíciójától függően szükséges teljes részletességgel megjeleníteni, távolról nézve elég, ha csak a főbb csúcsokat foglalja magában, megtartva az objektum jellegzetességét.

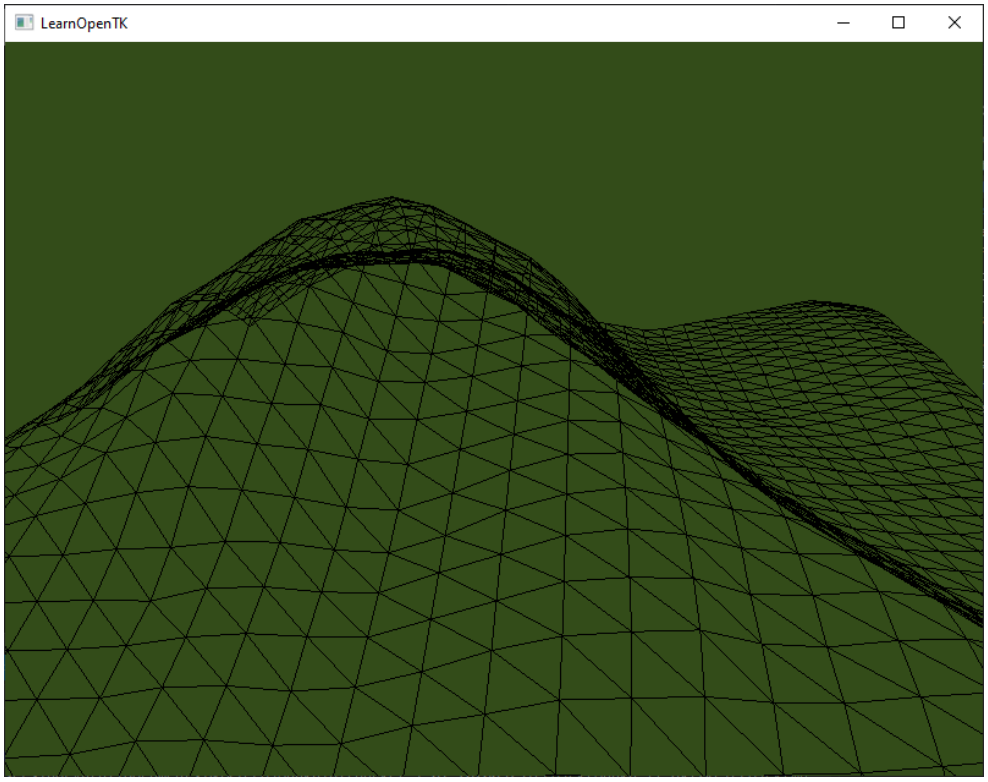
A második Level of Detail implementációnál nem egy adott geometriai alakzatot vesz alapul a program és ellenőrzi, hogy áthaladt-e már a térfogatán, hanem a kamera pozíciójától, magasságától függően a teljes betöltött térképrészletre számolja a kevesebb vagy több háromdimenziós koordinátákat a textúrával együtt. Ismét átváltva a polygon modera és a ctrl+l-el bekapcsolt második LoD működést prezentálva három szint lett meghatározva a konzulensemmel. A kezdő pozícióból indulva a legtávolabbi kameraállásból a térkép teljes részletességének csak minden harmadik magassági pontját számolja ki és jeleníti meg. Közelebb navigálva egy fixen beépített kamera magasság, vertikális pozíció alá érve minden második elemet vesz ki a magassági pontokból és azokat dolgozza fel, vetíti ki a képernyőre. Még közelebb érve már a teljes adatmennyiséget figyelembe veszi és adja át a megjelenítésnek a program.



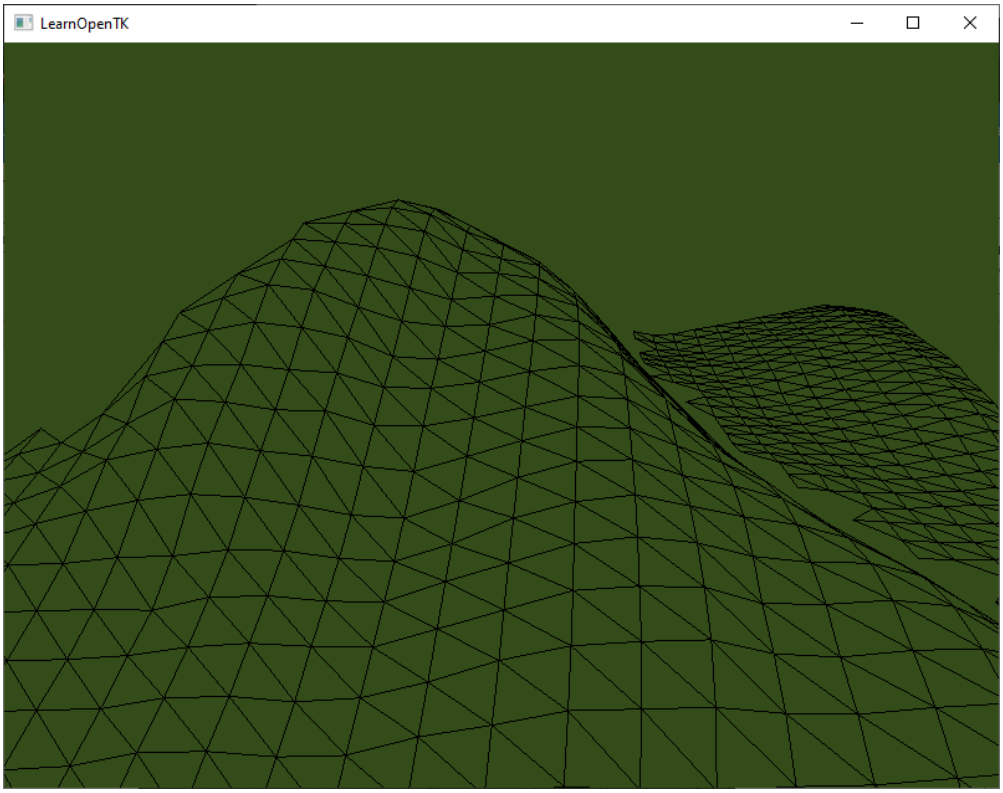
7. Ábra, Teljes térképrészlet részletességének változása a kamera közelítésének függvényében

Ezt a funkciót eredetileg Lagrange-interpoláció segítségével szerettem volna finomítani, kiszámolni, de az adott arányokhoz, az új sorokra és oszlopokra való felbontásához kiszámolt magassági pontok már torzuláshoz vezettek és a domborzat majdnem teljesen elvesztette a jellegzetességét a kisebb mértékek kiszámolásakor, a közelítéskor pedig a beégetett váltásokkor nagyon éles átmenetet és további alakzat deformálást okozott, így maradt végül a kisebb de eredeti mintából felépített virtuális tér. A funkció bekapcsolásakor az l betű A Lagrange számításra való célzást szolgálta volna, a kódban a függvények használatát meghagytam az esetleges későbbi bővítés, továbbfejlesztés lehetőségéhez, arra az esetre, ha több, az eredetinél még sűrűbb, részletesebb pontokat kellene meghatározni.

A harmadik implementáció szolgálná az occlusion detection fejlesztését, de az ipari, gyakorlati megoldásokkal szembe menve egy saját elven működő eljárást gondoltam ki egy objektumra, ami annak az egy objektumnak a kamera pozíciójából nem látható csúcspontjait hagyja figyelmen kívül. Az elgondolásomat arra építettem, hogy ha az összes diszkrét csúcsponthoz a kamera pozíciójából húzunk egy egyenest és az abba az irányba tartozó csúcsok közül akár egy is e vonal felett tartózkodik akkor az a pont nem látható. A vonal irányát a kamera és a vizsgált pont különbsége határozza meg, hogy felfelé vagy lefelé irányba “néz”-e a két végpont. Mivel a domborzat lehet hullámos így a vizsgált magasság és kamera között az összes ponthoz ezt ki kell számolni. A folyamatos irányítás miatt a kamera pozíciója, magassága változik így az összes csúcsra ki kell számolni a navigálás során a takarásban lévő pontokat. Mivel ez nem pixel szintű, hanem a modellt dolgozza fel így kapunk egy nem folytonos modellt, ami a megjelenítés során feltűnő hiányosságot eredményez a domborzatban, hegyek oldalánál a háttérben előkerülő csúcsok tűnnek fel viszont nincs átmenet a felszínben. Az eljárás három paramétert fogad a kamera x,y és z koordinátáját , ez alapján dől el, hogy mit tekint az adott logika szerint bejárt , a magassági adatokat két dimenziós tömbben tárolt csúcsok közül eltakartnak vagy sem. Lehetőségünk van tovább “játszani” a beállításokkal, az eljárás kamera paramétereit kis mértékben megváltoztatva, hogy a feldolgozást a megjelenítéshez jobban hozzáigazítva valósághűbb képet kapjunk, viszont ekkor az ellenkező irányba növelt vagy csökkentett oldalon fog nőni a folytonossági hiány.



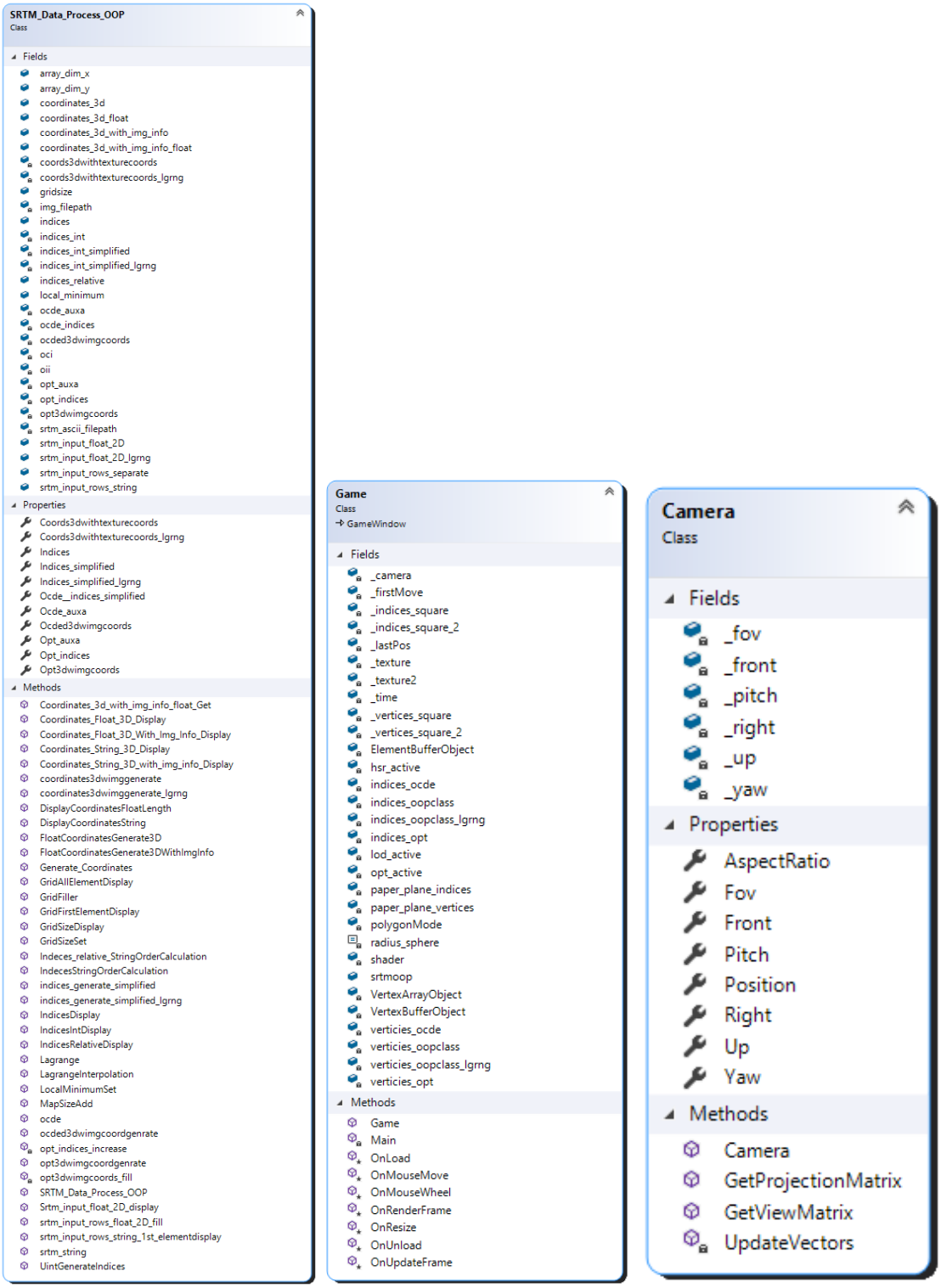
8. *Ábra, Az összes poligon kirajzolásának állapota, domborzat "háta" mögötti részletek megjelenítésével*

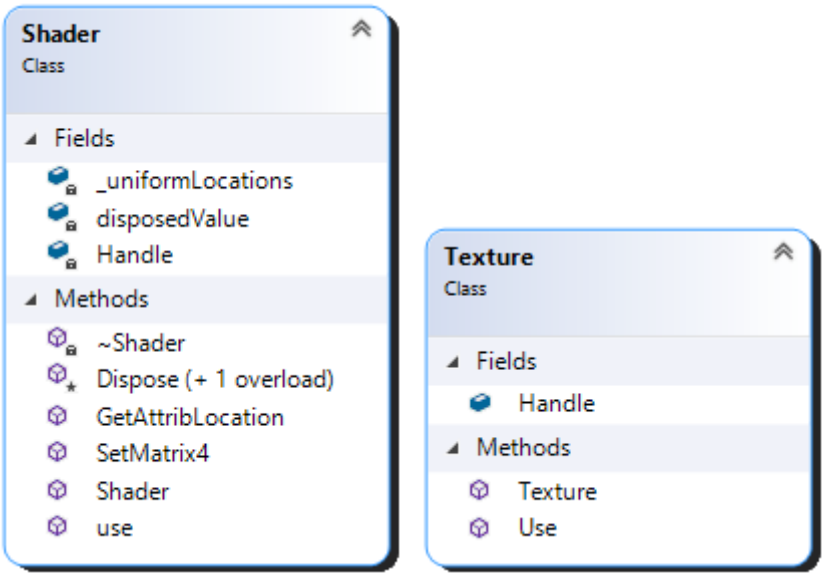


9. *Ábra, Az egyéni occlusion-detection algoritmus futása utáni megjelenítése*

Ennél a pontnál amikor ezt elértem és megjelent a képernyőn az eredmény nem tudtam tovább dolgozni a fejlesztés ezen részén, meghaladta a kapacitásaimat, hogy a nem általam választott grafikus megjelenítő rendszerben hogyan lehetne minél hatékonyabban felhasználni a rendelkezésre álló funkciókat, hogy az OpenGL-ben és más egyéb ipari megoldásokban használt depth vagy Z buffer elvű funkciókat bekapcsoljam és kiteszteljem. Ennek az a lényege, hogy minden egyes pixelhez a program hozzárendel egy mélységi számot és ha átfedés van más objektumok pixeljeivel akkor a kamerához legközelebb esőt jeleníti csak meg. A probléma nehézségét az adta inkább - és kezdem inkább a technikai oldalával a fejlesztésnek - hogy a grafikus keretrendszerben nem tudtam jól tájékozódni, nem állt rendelkezésre csak egy alapszintű tutorial a funkciók bemutatására. Maga az OpenTk egy grafikus library ami az OpenGL-t is magában foglaló .NET-es átirat. Ez magában hordozza, hogy a fejlesztő kezébe adja a döntést, hogy hogyan használja őket. Ez jelentette a nehézséget, hogy a bemutatott példákon túl nem voltak kapaszkodók, amivel még mélyebben megismerhettem volna a három-dimenziós megjelenítést. Csak próbálgatással tudtam elérni bizonyos dolgokat, de magas szintre nem tudtam fejleszteni a használatát. Például az egyik függvényben - GL.VertexAttribPointer - meg lehet adni, hogy a csúcsokat akarjuk-e normalizálni. Ekkor a tutorial oldal leírása alapján -1 és 1 intervallum közöttre kellett volna az egész számtípusokból megadott koordinátákat - integert használtam ennek kipróbálására - konvertálni és aszerint változott volna a megjelenítés, feltételezésem szerint sokkal kisebbnek kellett volna lennie, hogy beférjen az intervallumba, de a képernyőn azonban maradtak ugyanazok az arányok úgy, hogy a kamera pozíciója fixen be volt állítva. Nem tudtam eldönteni, hogy a program most melyik beállítással működik, az ezen intervallumon kívüli pontokat a leírás szerint el kellett volna hagynia, mégis megjelentek, illetve a paraméter állításra sem (true/false) történt változás azonos intervallumon kívüli adatok használatával. Nehézséget okozott, hogy C++-os OpenGL használata során nem kellett megadni a háromszögek sorrendjét, ezt alapból megoldja a rendszer - itt viszont meg kellett adni. Ez elsőre könnyű feladatnak tűnik, egy egyszerű algoritmust írni egy sima kétdimenziós tömbben tárolt magassági koordinátákat lefedni vagy inkább összekötni egy háromszöggel, kvázi mozaikszerűen megadni, de amint a level-of-detail implementálásakor a "felbontás" változott, vagyis a nem szomszédos csúcsokat kellett összekötni alaposan végig kellett gondolni, hogy milyen adatok vannak már meg és milyen eljárással tudom összefűzni egy tömbbe az indexeket. Redundánsan sikerült szerepeltetni ezeket a csúcspontsorrendeket de legalább működött és a wireframe módban line polygon mode-ban ez látszódik is. Problémát okozott az is, hogy csak egy objektummal tudtam dolgozni, de nem találtam példát arra, hogy hogyan lehetne több mindent hozzáadni a programhoz - nyilvánvalóan egy objektumlistára lett volna szükség - , ez is beárnyékolta a munkát. Egy modern game-enginben ilyenekre már nem is kell gondolni,Unity-ben, Unreal-ban összekattintgatja az ember a modelljét, széles palettáról tud választani és nagyobb produktivitással képes alap dolgokat elkészíteni, a fejlesztés során ez demotiválóan hatott, hogy ilyen problémákat kellett megoldani, hogy melyik háromszöget alkotó csúcsokat milyen sorrendben jelenítsen meg a rendszer. Nem egy

kifinomult kész mű született a kezeim közül csak egy megoldás, ami az alapokat használja és aminél létezik magasabb szintű sokkal professzionálisabb megvalósítás. Ezen tényeken túl a szakdolgozatban megjelölt probléma feladat nehézsége kihívást jelentő, az optimalizálás főleg valós igény. A modern és folyamatosan fejlődő megjelenítési iparágban - szoftveres, hardveres oldalon szintén - korlátozottan állnak rendelkezésre erőforrások, adott tranzisztorszámból kell a lehető legjobb és legtöbb munkát elvégeztetni, ami nem mehet az "élmény" rovására, akadozásra egy számítógépes játéknál, szimulátornál már nem lehet elfogadhatóan tekinteni, de kompromisszumok között erre van megoldás. A feladat összetett volt mert egy három-dimenziós modellt kellett létrehozni kiegészítve a textúra koordinátákkal és annak csak egy részét feldolgozni az optimalizálás során, megoldást kellett találni a csúcsok kirajzolásának sorrendjére, ez főleg a level-of-detail működése közben adott kihívást amit csak redundánsan tudtam megoldani. A végeredményt tekintve, hogy az optimalizálás miatt kevesebb terhelést kapjon a processzor a folyamatos kalkulálás miatt egy dolog miatt nem érte el bizonyítottan, még pedig amiatt, hogy idő szűkében nem file-ból/memóriából olvassa be a koordinátákat, hanem on-the-fly billentyűleütésre számolja ki. GPU szinten van egy kisebb nyereség.


SRTM_Data_Process_OOP
Class
Fields
array_dim_x
array_dim_y
coordinates_3d
coordinates_3d_float
coordinates_3d_with_img_info
coordinates_3d_with_img_info_float
coords3dwithtexturecoords
coords3dwithtexturecoords_lgrng
gridsize
img_filepath
indices
indices_int
indices_int_simplified
indices_int_simplified_lgrng
indices_relative
local_minimum
ocde_auxa
ocde_indices
ocded3dwimgcoords
oci
oii
opt_auxa
opt_indices
opt3dwimgcoords
srtm_ascii_filepath
srtm_input_float_2D
srtm_input_float_2D_lgrng
srtm_input_rows_separate
srtm_input_rows_string
Properties
Coords3dwithtexturecoords
Coords3dwithtexturecoords_lgrng
Indices
Indices_simplified
Indices_simplified_lgrng
Ocde__indices_simplified
Ocde_auxa
Ocded3dwimgcoords
Opt_auxa
Opt_indices
Opt3dwimgcoords
Methods
Coordinates_3d_with_img_info_float_Get
Coordinates_Float_3D_Display
Coordinates_Float_3D_With_Img_Info_Display
Coordinates_String_3D_Display
Coordinates_String_3D_with_img_info_Display
coordinates3dwimggenerate
coordinates3dwimggenerate_lgrng
DisplayCoordinatesFloatLength
DisplayCoordinatesString
FloatCoordinatesGenerate3D
FloatCoordinatesGenerate3DWithImgInfo
Generate_Coordinates
GridAllElementDisplay
GridFiller
GridFirstElementDisplay
GridSizeDisplay
GridSizeSet
Indeces_relative_StringOrderCalculation
IndecesStringOrderCalculation
indices_generate_simplified
indices_generate_simplified_lgrng
IndicesDisplay
IndicesIntDisplay
IndicesRelativeDisplay
Lagrange
LagrangeInterpolation
LocalMinimumSet
MapSizeAdd
ocde
ocded3dwimgcoordgenrate
opt_indices_increase
opt3dwimgcoordgenrate
opt3dwimgcoords_fill
SRTM_Data_Process_OOP
Srtm_input_float_2D_display
srtm_input_rows_float_2D_fill
srtm_input_rows_string_1st_elementdisplay
srtm_string
UintGenerateIndices
Game
Class
GameWindow
Fields
_camera
_firstMove
_indices_square
_indices_square_2
_lastPos
_texture
_texture2
_time
_vertices_square
_vertices_square_2
ElementBufferObject
hsr_active
indices_ocde
indices_oopclass
indices_oopclass_lgrng
indices_opt
lod_active
opt_active
paper_plane_indices
paper_plane_vertices
polygonMode
radius_sphere
shader
srtmoop
VertexArrayObject
VertexBufferObject
verticies_ocde
verticies_oopclass
verticies_oopclass_lgrng
verticies_opt
Methods
Game
Main
OnLoad
OnMouseMove
OnMouseWheel
OnRenderFrame
OnResize
OnUnload
OnUpdateFrame
Camera
Class
Fields
_fov
_front
_pitch
_right
_up
_yaw
Properties
AspectRatio
Fov
Front
Pitch
Position
Right
Up
Yaw
Methods
Camera
GetProjectionMatrix
GetViewMatrix
UpdateVectors



10. Az elkészült alkalmazás osztály diagrammjai (Az SRTM Process osztály a saját fejlesztés a többi a grafikus keretrendszer része)

Eredmények értékelése

A feladatot és az elért célt tekintve csak rész sikerekről tudok őszintén beszámolni. A program képes valósághű domborzati adatokat megjeleníteni, az adott grafikus keretrendszer alap működését sikerült megtanulnom és kiegészítettem egyéb funkciókkal, amik a témámhoz hozzátartoztak. A C# nyelv mellett azért döntöttem mert az egyetemi tanulmányok során is ebben programoztunk a legtöbbet, ebben éreztem magam a legbiztosabbnak, az itt megszerzett tudást akartam kamatoztatni. Tisztában vagyok vele, hogy az ipari sztenderdnek a három-dimenziós megjelenítések világában a C++ tekinthető. Az OpenTK keretrendszer is egy OpenGL alapú megjelenítésből lett átportolva.
Az optimális megjelenítést definiálva azt tekintettük elsődlegesnek, hogy nem memória hanem CPU használatot szeretnénk csökkenteni. A CPU-nál a grafikus mag végrehajtó egységeinek és a processzorban találhatók által végzett munkát külön vettem figyelembe. A mérést a beépített feladatkezelőre hagyatkozva végeztem, manuálisan navigálva a felszín felett úgy, hogy mind a sík mind a domborzat felett történjen a bejárás. A futó folyamatok közül a Visual Studioból futtatott projekt exe-t vizsgáltam a LearnOpenTk ablakot. A manuális teszt elvégzése közben igyekeztem kizárni az időszakos kilengéseket és nem vettem figyelembe a legmagasabb, legalacsonyabb értékeket, külön célszoftvert nem használtam az értékek rögzítésére.

	CPU				GPU			
	With Texture		PolygonMode : Line		With Texture		PolygonMode : Line	
	min	max	min	max	min	max	min	max
Default	1.9%	3.1%	1.5%	2.6%	4.2%	5.7%	3.8%	5.5%
Optimized	5.6%	8.6%	6.0%	8.7%	4.1%	4.6%	4.4%	4.9%

I. A mért kihasználtságok a különböző módok bekapcsolása során

A “default” értékeket a keretrendszer alapfunkciójának futtatásával rögzítettem, az eredeti textúra alkalmazásával, illetve a vonal polygonmode beállítás és sötét háttérrel való futtatással összevetve, CPU és GPU külön vizsgálva. Az “optimized” nézet esetben, amit a ctrl+o paranccsal lehet aktiválni, kijelenthető, hogy a kevesebb csúcspontból felépített megjelenítés a GPU-nak kisebb terhelést jelent, a kevesebb háromszög kisebb GPU kihasználtságot eredményez. Az ezek előállításához szükséges modell felépítéséhez a processzor terhelés, habár emelkedett, de nem annyira jelentősen. Figyelembe véve azt, hogy a kamera mozgását feldolgozva, annak pozíciójából számol és épít fel a közelítésre sokkal részletesebb domborzat felszínt a CPU által végzett munka alacsonyabb GPU terhelést jelentett.

Az így felhasznált egyedi level of detail implementáció több CPU erőforrást igényelt, de ez a GPU-nál kevesebb kihasználtságot eredményezett. Az eredmények valószínűsíthetően sokat javultak volna CPU szinten, ha a kalkulációkat előre elvégzi egy külön erre a célra fejlesztett modul, így azt csak vagy memóriából vagy ssd tárolóból olvassa fel a program a kamera pozíciójától függően, így illesztve egy valószínűsíthető újabb téradatot a feldolgozandó csúcsokat tartalmazó objektumhoz és csak ez kerülne kiszámításra a kamera pozíciójától függően.

Összegzés

Szakdolgozatomban sikeresen implementáltam az elvégzendő megjelenítési feladatot egy alacsony szintű nem mainstream grafikus keretrendszer felhasználásával. A szabadon felhasználható valós domborzati adatokból a NASA SRTM publikusan elérhető és kiválasztható adatait valósághűen rajzolja ki a program. Bizonyos feltételekkel használható bármely 64x31-nél nem nagyobb méretű mátrix adataival az elérhető világ bármely felszínének kiválasztásával, ami nagyjából 9 km^2-nyi területnek felel meg. A fejlesztés során le kellett programoznom optimalizálási módszereket is, a konzulensemmel megbeszélt kettő opció közül egyet sikerült teljesen megvalósítani. Ez a Level-of-Detail eljárás, aminek a lényege, hogy a megjelenítés során a távol eső tárgyakat, objektumokat nem szükséges teljes részletességgel és azoknak a modelljét alkotó összes csúcspontot felhasználva átadni feldolgozásra, hanem a távolság függvényében elegendő csak kisebb poligonokból elvégezni a kirajzolást így kevesebb transzformációs műveletet kell végrehajtani. Az implementációt a legjobban úgy tudom leírni, hogy a térben egy gömböt kell elképzelni, aminek a térfogata állandó kiszámítással történik és abban az esetben, ha a domborzat felszínéhez navigálva a kamerát a két felszín keresztezi egymást akkor egyre sűrűbb, pontosabb modellt számol ki és jelenít meg a program. Így tudtam szimulálni, hogy a közelebb lévő pontok, területek a legvalósághűebbek legyenek, míg a távoli objektum részek veszítsenek feldolgozottságukból, de megtartsák jellegzetességüket. A Level-of-Detail egy másik megoldását is be tudtam fejezni, ekkor a teljes térképfelszínnek az összes csúcspontja változik a kamera magasságától függően, a legrészletesebb és maximális adatokkal való megjelenítésen túl felezi, illetve harmadolja a magassági adatokból való mintavételezést. Szerettem volna, ha ennek a részfeladatnak az elvégzése során tudok interpolációt használni, de az implementált Lagrange eljárás torzuláshoz vezetett a felszínben, nem tartotta meg a szintenkénti átmenetet és rendkívül éles képbeli váltáshoz vezetett, így ezt elvetettem. Egy másik eljárás az optimalizációra az occlusion detection lett volna, ami az objektumok eltakarását számította volna ki és az egymást elfedő alakzatok nem kerülnek kirajzolásra. Ezt nem tudtam leprogramozni, ehelyett egy hidden-surface-removal-hoz hasonlító egyedi implementációba kezdtem, ami a megjelenített egy objektumnak a nem látható csúcspontjait szűri ki, ez félig sikerült is de további logika és paraméterezés implementációjára lenne szükség, hogy a látható és nem látható csúcsok átmenete le legyen kezelve, hogy ne legyen folytonossági hiány a diszkrét pontok között. A nem általam választott grafikus rendszer képes megjeleníteni Magyarország valós domborzatának egy részét, de a teljes adathalmazból felépített modellt nem dolgoztam ki. Az ehhez szükséges függvények és eljárások rendelkezésre állnak így további munkával akár ezt is be lehet építeni. Ekkor akár nagyobb és más világrészeknek az adatainak a megjelenítésére is képes lenne a program, de jelenleg olyan kettőezer háromszöget képes kirajzolni egy üzleti laptopon. A megjelenítés a navigálás során folyamatos, tetszés szerint tudjuk az adott területet bejárni. Habár nem sikerült maximálisan teljesíteni a kiszabott feladatot csak részben,

érdekesnek találtam a számítástechnika ezen területét, eddig ilyen típusú technológiával nem dolgoztam, nem volt tantárgy az egyetemen. A nehézségek ellenére, amit leginkább a grafikus keretrendszer alacsony szintje okozott értékesnek tekintem a munkával töltött időt és sokat tanultam.

Summary

In my thesis I have implemented successfully the graphical rendering task with a low-level not mainstream graphical framework. I used the publicly available real elevation data, sourced from NASA SRTM mission , that is visualized realistic , following the characteristics of the real-world hill . This program is capable of processing of a 64x31 data matrix of elevation data of real-world , that is approximately 9 km^2 surface volume. During the implementation I had to develop some optimizatio methods as well. I could fulfill one of the two agreed method entirely, the other one succeeded only partially. The fully-implemented Level-of-Detail method is about objects in the distance need to be rendered only with less details and in this way only less of their valid vertices needs to be used in the three-dimension framework, only depending on the distance between object and camera, less or more vertices needs to be managed thus less or more transformation process need to run. To describe the implementation imagine a sphere whit a pre-fixed radius and its origo is the camera position. During navigation it is re-calculated in each position and in that case the volume of this sphere and the surface meet the more detailed of surface is generated, more real vertex and elevation data, points get into the calculation. In this way I could simulate that cases when the camera gets closer to the surface , it is desplayed more realistic and detailed, while zooming out less elevation is enough, also keeping the characteristics of the original object as well. Another Level-of-Detail implementation happened as well, in this case the whole map-data gets recalculated depending on the height of the camera, half or two-third of the original elevation gets processed. In this phase I tried out an interpolation technique, but using Lagrange and retrieving other points from the original data set deformed the relief lookout so I decided to omit. Another optimization would be the so called occlusion-detection, where occluded objects are not rendered out. This is what I could not develop, but I started another custom solution that might have been a similar process, mixture of occlusion-detection and hidden-surface-removal and which is based on the object vertices. Partially I could finish this function of the program but extended logic and extra parameters are needed to render absolutely the object details and handle the gaps between some elevation points and triangles. This program is capable to display a part of Hungary's real-world relief, but method of covering the whole dataset and building a manageable elevation object was not worked out. Despite of this the available functions and methods would make this program capable to generate this object, so it is open to extend in this direction as well.Also in this case other parts of the world – via downloading the srtm data site – could be rendered out, but currently approximately two-thousand triangle is being processed on an average business laptop. The navigation during the display is continuous, we can "fly" in any area of the object. Although I succeeded partially of fulfilling the tasks I found very interesting this work and I got insight of three-dimension visualization, so far I had no such experience. Despite

of the difficulties – for example; low level of the graphical framework – I spent valuable time learning the basics of this topic I could achieve some goals.

Irodalomjegyzék

Arm Limited. (2020. 12 01). *Geometry Best Practices for Artists*. Forrás: https://developer.arm.com/solutions/graphics-and-gaming/developer-guides/game-artist-guides/geometry-best-practices/single-page#Tri

Center, U. D. (2011. 05 16). *USGS scientists publications.* Forrás: https://pubs.usgs.gov/fs/2003/0071/report.pdf

Chandel, H., & Vatta, S. (2020. 12 01). *Researchgate.* Forrás: https://www.researchgate.net/publication/278729405_Occlusion_Detection_and_Handling_A_Review

ESA Earth Online. (2020. 12 01). Forrás: https://earth.esa.int/web/eoportal/satellite-missions/s/srtm

Evanson, N. (2019. 06 03). *3D Game Rendering 101*. Forrás: https://www.techspot.com/article/1851-3d-game-rendering-explained/

Evanson, N. (2019. 07 21). *How 3D Game Rendering Works: Vertex Processing*. Forrás: https://www.techspot.com/article/1857-how-to-3d-rendering-vertex-processing/

Facility, O. (2014. 04 08). *OpenTopography*. Forrás: https://opentopography.org/news/srtm-version-30-global-90m-and-united-states-30m-elevation-data-now-available

FrederikJA, P.-J. B. (2020. 12 01). *Introduction - OpenTK*. Forrás: https://opentk.net/

FrederikJA, P.-J. B. (2020. 12 01). *LearnOpenTK*. Forrás: https://opentk.net/learn/index.html

FrederikJA, P.-J. B. (dátum nélk.). *Projects using OpenTK*. Forrás: https://opentk.net/resources.html

Game-Ace Creative Studio. (2020. 12 01). *3D Modeling for Unity: The Complete Guide*. Forrás: https://game-ace.com/blog/3d-modeling-for-unity/

Group, T. K. (2020. 09 27). *OpenGL Overview.* Forrás: https://www.opengl.org/about/

Mozilla. (2020. 11 02). *WebGL: 2D and 3D graphics for the web - Web APIs | MDN*. Forrás: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

Murphy, K. (2020. 12 01). *Unity Game Engine Review*. Forrás: https://www.gamesparks.com/blog/unity-game-engine-review/

OpenTopography. (2020. 10 09). *OpenTopography - Shuttle Radar Topography Mission (SRTM GL1) Global 30m*. Forrás: https://portal.opentopography.org/raster?opentopoID=OTSRTM.082015.4326.1

Overvoorde, A. (2019. 10 15). *Introduction - Vulkan Tutorial.* Forrás: https://vulkan-tutorial.com/Overview

Petty, J. (2020. 12 01). *What is Unity 3D & What is it Used For?* Forrás: https://conceptartempire.com/what-is-unity/

SRTM. (2020. 11 27). Forrás: https://www2.jpl.nasa.gov/srtm/mission.htm

The Khronos Group. (2020. 09 30). *OpenGL Overview*. Forrás: https://www.opengl.org/about/

The Kronos Group. (2016. 02 15). *Videos, Presentations, & Supporting Materials Archives - The Khronos Group Inc.* Forrás: https://www.khronos.org/assets/uploads/developers/library/overview/Vk_20160 2_Overview_Feb16.pdf

The Kronos Group. (2020. 11 25). *FreeGLUT*. Forrás: http://freeglut.sourceforge.net/

The Kronos Group. (2020. 10 10). *gluLookAt*. Forrás: https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/gluLookAt.xml

The Kronos Group. (2020. 09 29). *OpenGL Functions*. Forrás: https://csciwww.etsu.edu/barrettm/4157/OpenGL%20Functions.htm

The Kronos Group. (2020. 10 15). *Vulkan Overview*. Forrás: https://www.khronos.org/vulkan/

Tomas Akenine-Möller, E. H. (2020). *Real-Time Rendering, Fourth Edition.* Forrás: Real-Time Rendering: https://www.realtimerendering.com/udacity/?load=demo/unit7-view-pipeline.js

Ábrajegyzék

Táblajegyzék

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Horváth Levente Péter..... GXI2KP..... esti.....

Telefon: Levelezési cím (pl: lakcím):

06204934292..... 1131 Bp., Keszkenő utca 8. 3/9.....

Szakdolgozat / Diplomamunka[1] címe magyarul:

SRTM adatok 3D-s megjelenítése és a megjelenítés optimalizálási lehetőségei.....

Szakdolgozat / Diplomamunka[2] címe angolul:

3D Visualization of SRTM data and the optimization opportunities of the visualization

Intézményi konzulens: Külső konzulens:

Dr. habil. Szénási Sándor..... .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2021-09-28	EDDIGI MUNKA ÁTNÉZÉSE KONZULENS CSERE MIATT	
2.	2021-10-26	LEVEL-OF-DETAIL IMPLEMENTÁCIÓ BEMUTATÁSA	
3.	2021-11-10	OCCLUSION DETECTION IMPLEMENTÁCIÓ BEMUTATÁSA	
3.	2021-12-06	DOLGOZAT FORMAI KÖVETELMÉNYÉNEK EGYEZTETÉSE	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplo-mamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2021-12-13

.....
intézményi konzulens

1 Megfelelő aláhúzandó!
2 Megfelelő aláhúzandó!
3 Megfelelő aláhúzandó!
4 Megfelelő aláhúzandó!

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Horváth Levente Péter..... GXI2KP..... esti.....

Telefon: Levelezési cím (pl: lakcím):

06204934292..... 1131 Bp., Keszkenő utca 8. 3/9.....

Szakdolgozat / Diplomamunka[1] címe magyarul:

SRTM adatok 3D-s megjelenítése és a megjelenítés optimalizálási lehetőségei.....

Szakdolgozat / Diplomamunka[2] címe angolul:

3D Visualization of SRTM data and the optimization opportunities of the visualization

Intézményi konzulens: Külső konzulens:

Dr. habil. Szénási Sándor..... .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2020-09-14	SRTM, TRIANGLE-STRIP	
2.	2020-10-21	VIZUALIZÁCIÓS TECNOLÓGIÁK ÁTTEKINTÉSE: OPENGL, VULKAN	
3.	2020-11-30	VIZUALIZÁCIÓS TECNOLÓGIÁK ÁTTEKINTÉSE: UNITY, OPENTK	
4.	2020-12-09	OPTIMALIZÁLÁS: LOD, OCCLUSION DETECTION	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplo-mamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2020-12-09

.....
intézményi konzulens

1 Megfelelő aláhúzandó!
2 Megfelelő aláhúzandó!
3 Megfelelő aláhúzandó!
4 Megfelelő aláhúzandó!