



EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR
PROGRAMOZÁSELMÉLET ÉS SZOFTVERTECHNOLÓGIAI
TANSZÉK

**Felületrekonstrukció 3D-s LiDAR
ponthalmaz és kamerakép alapján**

Témavezető:
Tóth Tekla
egyetemi tanársegéd

Szerző:
Gyarmati András
programtervező informatikus BSc

Budapest, 2023

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:

Név: Gyarmati András

Neptun kód: UY18CX

Képzési adatak:

Szak: programtervező informatikus, alapképzés (BA/BSc/BProf)

Tagozat : Esti

Belső témavezetővel rendelkezem

Témavezető neve: Tóth Tekla

munkahelyének neve, tanszéke: ELTE IK, Algoritmusok és Alkalmazásaiak Tanszék

munkahelyének címe: 1117, Budapest, Pázmány Péter sétány 1/C.

beosztás és iskolai végzettsége: Egyetemi tanársegéd, ELTE IK programtervező informatikus MSc

A szakdolgozat címe: Felületrekonstrukció 3D-s LiDAR ponthalmaz és kamerakép alapján

A szakdolgozat témája:

(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását)

A dolgozatom célja egy olyan alkalmazás készítése, ami egy tesztautóra rögzített 16 sugarú 360 fokban körbeforgó LiDAR (Light Detection and Ranging) szenzor és 4-6 darab digitális Hikvision MV-CA020-20GC kamerából érkező adatokat fűz össze. A LiDAR szenzor kimenete egy 3D pontfelhő amelyet először különálló színezett pontokként majd később 3D felülettől összefűzve jeleníték meg, a kamerák pedig képeket rögzítenek, amelyeket a színezéshez használók fel.

A témát érintő főbb informatikai területek az adatvizualizáció és a számítógépes grafika. Valós szenzoradatokat használva lehetőségem van vizuálisan jól validálható feladatot megvalósítanom. A feladat jól skálázható és a már létező hasonló megoldásokkal összehasonlítható.

A megjelenítéskor feltételezem, hogy az adatok időben szinkronizáltak, és a szenzorok előre kalibráltak, azaz a kamerák és a LiDAR koordinátarendszere közti projekció ismert. Ez alapján a színezett pontfelhő elkészítése a következőképpen törénik: a 3D-s pontokhoz színérték rendelhető a megfelelő kamerakép megfelelő pixele alapján.

Ezt az egyszerűbb megjelenítést szeretném felületrekonstrukcióval kiegészíteni, azaz a 3D-s pontokból meshekkel gyártani, és több textúrainformációt felhasználni a képekről. Ez azért fontos, mert a jelenleg felhasznált 16 sugar felbontású LiDAR vertikálisan ritkás pontfelhőt ad, így a képi információ nagy része elveszik a 3D-s pontfelhő nézetben. A mesh építéshez Delaunay háromszögelési algoritmust veszem alapul, majd hatékonyabb megoldásokat is kipróbálok. Az algoritmust úgy fogom megtervezni hogy figyelembe veszem a pontok vertikális ritkásságát, valamit hogy horizontálisan viszont sürűn helyezkednek el. A kamera és a LiDAR különböző nézetéből adódó takarásbeli különbségeit is megpróbálom kezelní a színezés során.

A dolgozatot C++ nyelven írom, grafikus megjelenítésre az OpenGL programkönyvtárat, a képek feldolgozására pedig az Eigen, OpenCV keretrendszeret használom.

Budapest, 2022. 12. 01.

Tartalomjegyzék

1. Bevezetés	3
2. Felhasználói dokumentáció	5
2.1. A megoldott probléma rövid megfogalmazása	5
2.2. A felhasznált módszerek rövid leírása	5
2.3. A program használatához szükséges információk	6
2.3.1. Telepítési útmutató	6
2.3.2. A program indítása	6
2.3.3. Funkciók	8
3. Fejlesztői dokumentáció	18
3.1. A probléma részletes specifikációja	18
3.1.1. A LiDAR szenzor jellemzői, és az adatgenerálás	18
3.1.2. A kamerák jellemzői	19
3.1.3. Kamera paraméterek betöltése	19
3.1.4. A LiDAR szenzor és a kamera közötti transzformáció	21
3.1.5. A megjelenítési formák rövid leírása.	21
3.2. A felhasznált módszerek leírása, a használt fogalmak definíciója	22
3.2.1. A fájlok beolvasása	22
3.2.2. A pontok renderelése	22
3.2.3. Delaunay háromszögelés és tetraéderhálózás	22
3.2.4. Octree	23
3.2.5. Mesh építés az adatok struktúráját figyelembe véve	23
3.3. A szoftver szerkezete	24
3.3.1. Osztály diagram	24
3.3.2. main.cpp	24
3.3.3. application.h/cpp	25
3.3.4. file_loader.h/cpp	30

3.3.5. octree.h	30
3.3.6. delaunay_3d.h	31
3.3.7. A LiDAR ponthalmaz struktúráját felhasználó mesh generálás	34
3.4. Tesztelés	34
3.4.1. Textúra vetítés ellenőrzése gömb ponthalmazzal	34
3.4.2. Delaunay tetraéderháló kocka csúcsponjtaira	35
4. Összegzés	36
Köszönetnyilvánítás	37
Irodalomjegyzék	37
Ábrajegyzék	40
Algoritmusjegyzék	42
Forráskódjegyzék	43

1. fejezet

Bevezetés

A dolgozatom célja egy olyan alkalmazás készítése, ami egy tesztautóra rögzített 16 sugarú 360 fokban körbe forgó LiDAR [1] (Light Detection and Ranging) szenzor és 3 darab digitális Hikvision MV-CA020-20GC kamerából érkező adatokat fűz össze. Az eszközöket az 1.1 ábrán láthatjuk. A LiDAR szenzor kimenete egy 3D pontfelhő [2], amelyet először különálló színezett pontokként, majd később 3D felületté összefűzve jelenítek meg, a kamerák pedig képeket rögzítenek, amelyeket a színezéshez használok fel.

A témát érintő főbb informatikai területek az adatvizualizáció, és a számítógépes látás. Valós szenzor adatokat használva lehetőségem van vizuálisan jól validálható feladatot megvalósítanom. A feladat jól skálázható és a már létező hasonló megoldásokkal összehasonlítható.

A megjelenítéskor feltételezem, hogy az adatok időben szinkronizáltak, és a szenzorok előre kalibráltak, azaz a kamerák és a LiDAR koordinátarendszere közti projekció ismert. Ez alapján a színezett pontfelhő elkészítése a következőképpen történik: a 3D-s pontokhoz színértéket rendelek a megfelelő kamerakép megfelelő pixele alapján.

Ezt az egyszerűbb megjelenítést egészíttem ki felületrekonstrukcióval, azaz a 3D-s pontkból meshekkel gyártok, amiken már az egész képet - nem csak elszórtan mintavételezett pontjait - hasznosítom textúraként. Ez azért fontos, mert a jelenleg felhasznált 16 sugár felbontású LiDAR vertikálisan ritkás pontfelhőt ad, így a képi információ nagy része elveszik a 3D-s pontfelhő nézetben. A mesh építéshez Delaunay háromszögelési algoritmust [3] veszem alapul, majd egy bemeneti adatstruktúrát figyelembe vevő hatékonyabb megoldást is alkalmazok.

Az algoritmust úgy tervezem meg hogy figyelembe veszem a pontok vertikális ritkásságát, valamit, hogy horizontálisan viszont sűrűn helyezkednek el.

A dolgozatot C++ [4] nyelven írom, grafikus megjelenítésre az OpenGL [5] programkönyvtárat használom. Projektem kiindulási alapjaként a számítógépes grafika tárgyakon használt OGLBase [6] projektet használom.

Az alkalmazásom fő felhasználási területe segédprogram az egyetemi projektekben. A számítógépes grafikát oktató, tanuló és kutató kollégák már ismerik a felhasznált megoldások nagy részét így könnyen üzembe helyezhetik, továbbfejleszthetik és használhatják a projektet.



1.1. ábra. A LiDAR szenzor és a felhasznált kamerák kiemelése

2. fejezet

Felhasználói dokumentáció

2.1. A megoldott probléma rövid megfogalmazása

A dolgozatomban olyan pontfelhőkből készíték 3D-s felületeket, amelyeket az egyetem által használt LiDAR szenzor rögzített. A felületek színezéséhez a szenzor mellet elhelyezett kamerák képeit használom.

A pontfelhőket és a képeket fájlokból töltöm be majd az adatokat előkészítem a megjelenítésre. Beolvasást követően az OpenGL támogatja a pontok natív megjelenítését. A felület kirajzolásához a pontokat már össze kell kötnöm valamilyen formában, amire a háromszögek indexekkel való felsorolását választom.

Megjeleníthető továbbá a pontok octree [7] struktúrája is, valamit a Delaunay tetraéderhálózás egy részének vizualizációja.

2.2. A felhasznált módszerek rövid leírása

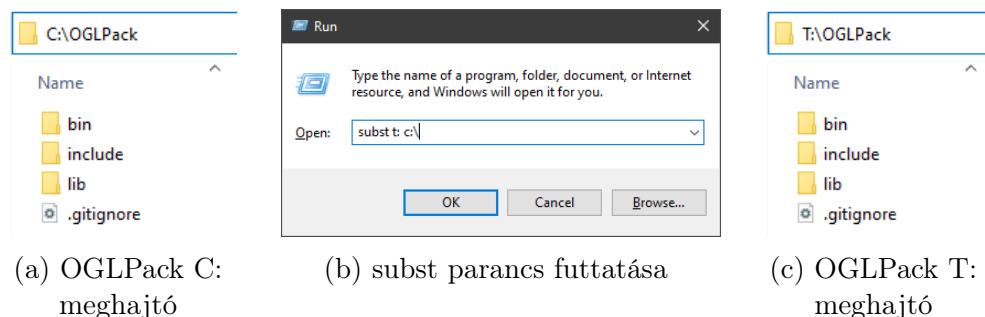
Az alkalmazásom egy C++17 program, amely SDL-t [8] (Simple DirectMedia Layer) használ az ablak megjelenítésére és a felhasználói interakció kezelésére. Az OpenGL kiegészítők betöltését a GLEW [9] (OpenGL Extension Wrangler) kezeli. Az SDL ablakon belül OpenGL 4.6 [10] könyvtárral jelenítem meg a grafikai elemeket, ami a felhasználói interfésből és a grafikusan megjelenített szenzoradatokból áll.

A grafikus felhasználói felületet (GUI - Graphical User Interface) az ImGui [11] csomag segítségével rajzolom ki.

2.3. A program használatához szükséges információk

2.3.1. Telepítési útmutató

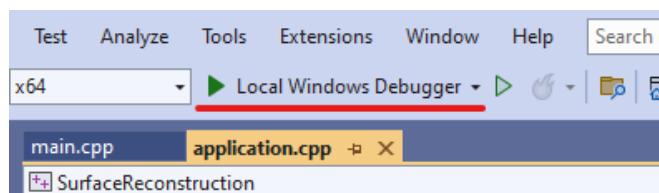
A programot a Visual Studio [12] fejlesztői környezetben lehet futtatni, továbbá szükség van az OGLPack [6] csomagra a rendszer gyökérkönyvtárában. A csomag elérhető az ELTE grafikai tárgyainak weboldalán a cg.elte.hu-n. A meghajtót ezután virtuálisan klónozni kell a `subst t: c:\` paranccsal, ahogyan a 2.1 ábrán látható. Erre azért van szükség, hogy a Visual Studio-ban könnyen beállíthassunk hivatkozásokat külső könyvtárakra.



2.1. ábra. Az OGLPack és a subst parancs

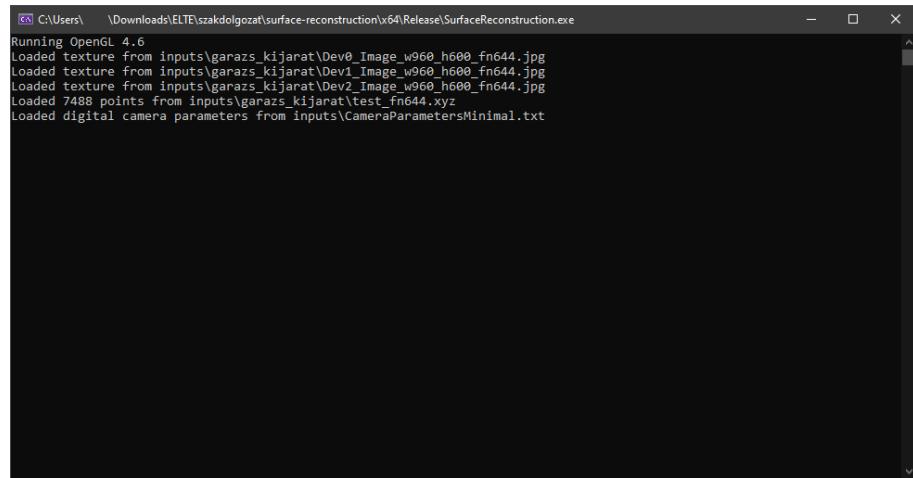
2.3.2. A program indítása

A program könyvtárban a `.sln` fájl megnyitásával tölthetjük be az alkalmazást a Visual Studio-ba, ahol a zöld háromszögű kattintva futtatható a program.



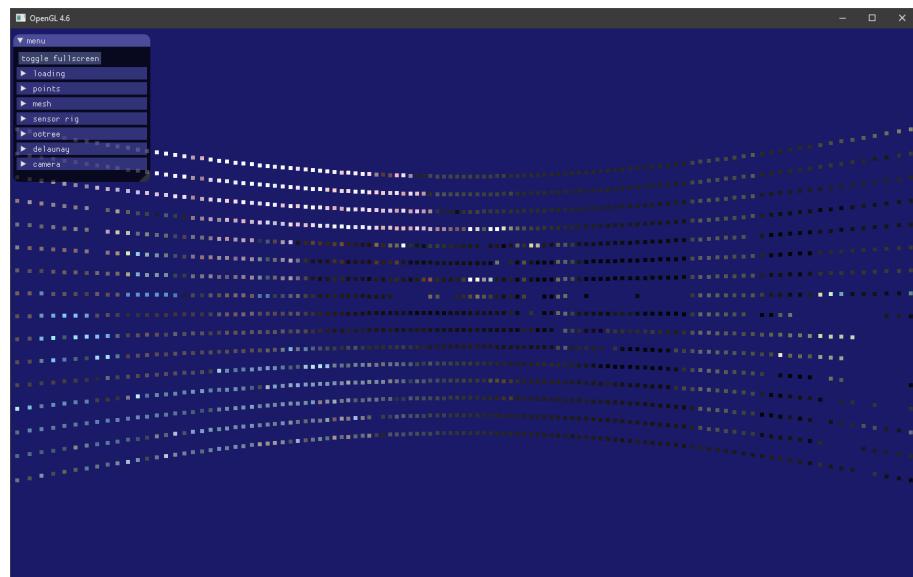
2.2. ábra. A program indítása Visual Studio-ból

A program futtatása után megjelenik a konzol (2.3 ábra), ahol a futás közbeni esetleges hibák, valamint az általunk kiírt üzenetek jelenhetnek meg. Megjelenik továbbá a grafikus ablak (2.4 ábra) is, ahol a GUI-ról választhatunk a különböző megjelenítési módok közül.



2.3. ábra. CMD ablak

A program indításakor az előre betöltött egyik ponthalmaz látható különálló pontozott megjelenítési módban.

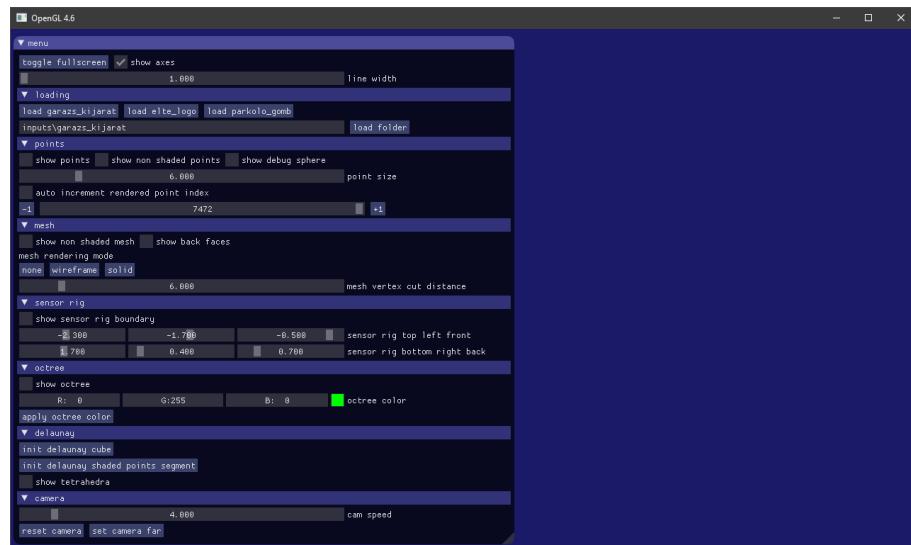


2.4. ábra. A grafikus ablak indítás után

2.3.3. Funkciók

A menüben a következő funkciók érhetőek el:

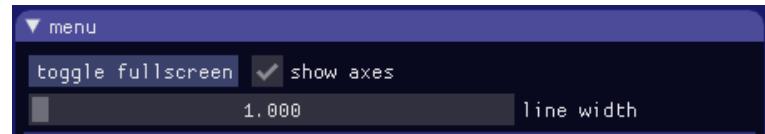
- Teljes képernyős mód, tengelyek mutatása, vonalvastagság
- Mappa betöltés
- Pontok megjelenítése
- Felület megjelenítése
- A szenzor körül terület beállítása
- Octree megjelenítése
- Delaunay tetraéderháló megjelenítése
- Kamera beállítások



2.5. ábra. A teljesen kinyitott menü

Teljes képernyős mód, tengelyek mutatása, vonalvastagság

Az ablak a megszokott módon a tálca minimalizálható, a teljes képernyőt kitölten - de még mindig ablakos módba - maximalizálható, bezárható, valamint az élek és sarkok megfogásával és húzásával tetszőleges méretre skálázható. Az ablak nélküli teljes képernyős módhoz külön gombot helyeztem el a menüben. A gomb mellett a tengelyek láthatóságát kapcsoló checkbox található. Ezek alatt a vonalvastagságot állíthatjuk egy csúszkával.



2.6. ábra. Teljes képernyős mód gomb, tengelyek mutatása checkbox, vonalvastagság csúszka

Mappa betöltés

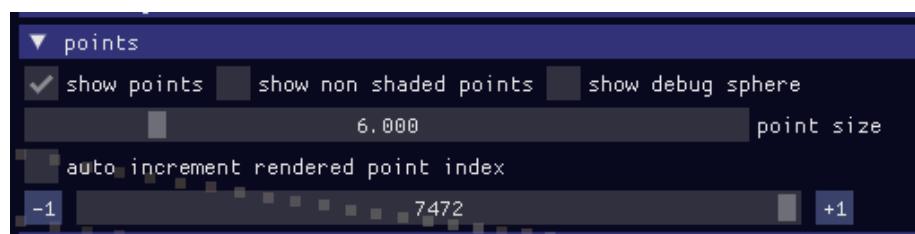
A menüben három előre definiált mappát tölthetünk be, amelyekhez gombokat hoztam létre. Az elérési út beillesztésével és a "load folder" gomb megnyomásával további mappák is betölthetők.



2.7. ábra. Mappa betöltés menü

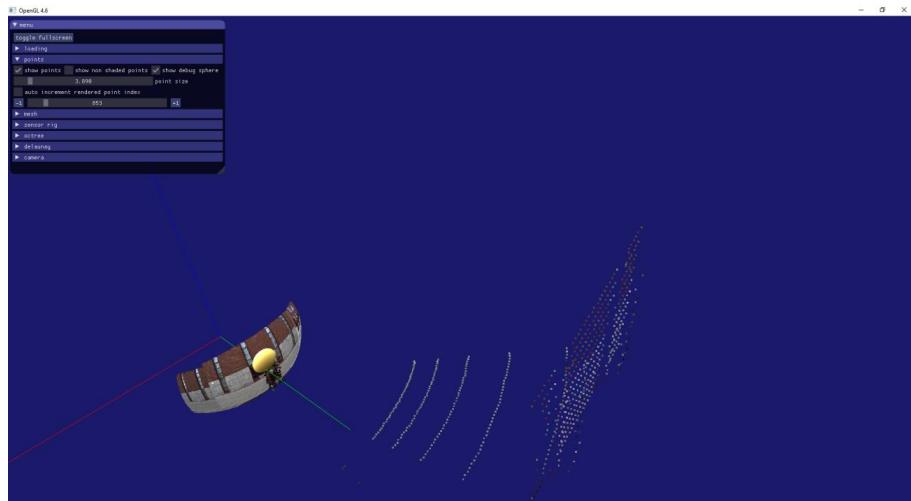
Pontok

A pontok megjelenítése ki- és bekapcsolható. Lehetőség van a nem textúrázott pontok láthatóvá tételere is. A következő opcióval egy gömbfelületet rajzolhatunk ki pontokból, amelyen minden irányban látszik a pontokra vetített textúra. A pontok méretét egy csúszkával állíthatjuk be. Ez után látható egy checkbox, ami lehetővé teszi a pontok szekvenciális megjelenítését, majd ezt követően további csúszka és plusz-mínusz gombok a renderelt pontok számának beállításához.

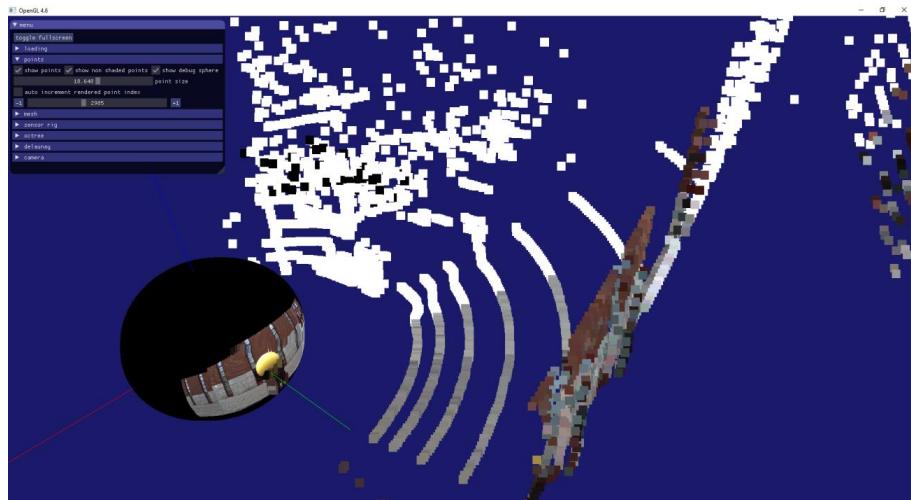


2.8. ábra. Pontok menü

A 2.9 ábrán alapállapotban hagytam a beállításokat csak a gömböt kapcsoltam be, amiből jelenleg csak a textúrázott rész jelenik meg. A 2.10 ábrán minden értéket megváltoztattam a különbségek érzékeltetése miatt.



2.9. ábra. Pontok menü alapállapot



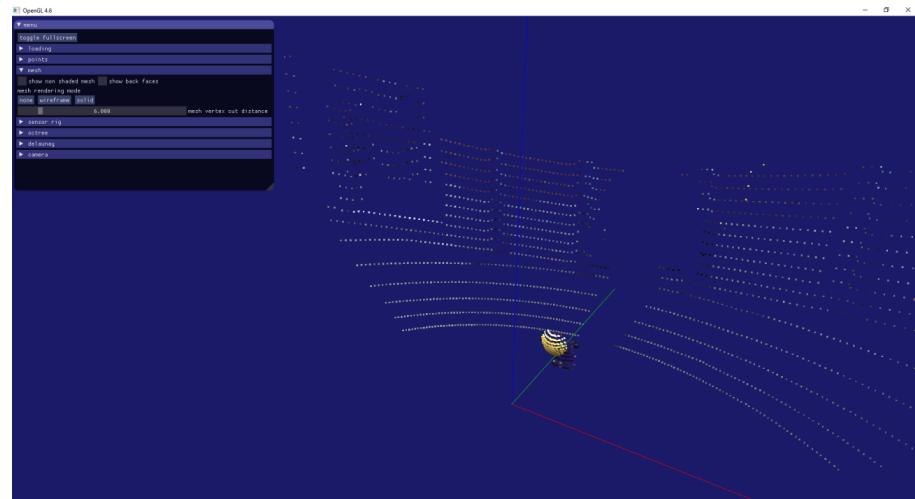
2.10. ábra. Pontok menü megváltoztatott értékekkel

Felület

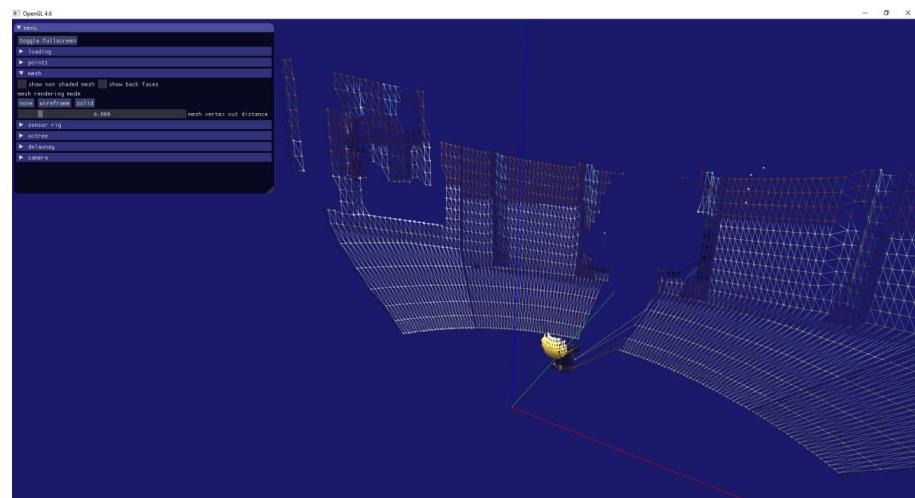
A menü ezen részén álltható felület azon területeinek megjelenítése, amelyek nincsenek textúrázva. A meshek általában zártak ezért a hátoldalakat nem mutatjuk, itt viszont nyitott felületünk van, ezért kapcsolható a hátoldalak megjelenítése. A mesh kirajzolási módot is állíthatjuk, ami lehet kikapcsolt, vonalas vagy teljesen színezett állapotban. Meghatározhatjuk a háromszögek oldalhosszának maximális értékét; ha ezt a határt túllépik, a háromszögeket nem jelenítjük meg. Ez a beállítás lehetővé teszi a nagyobb háromszögek elrejtését a megjelenítés során.



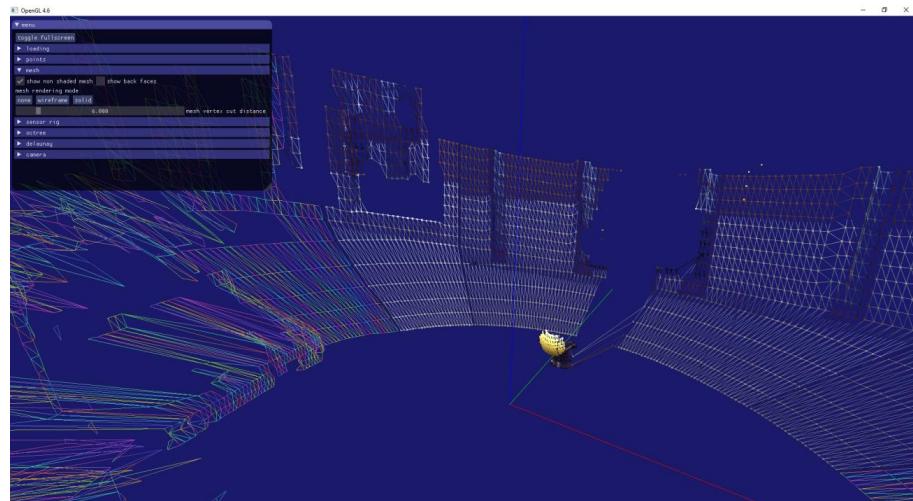
2.11. ábra. Mesh menü



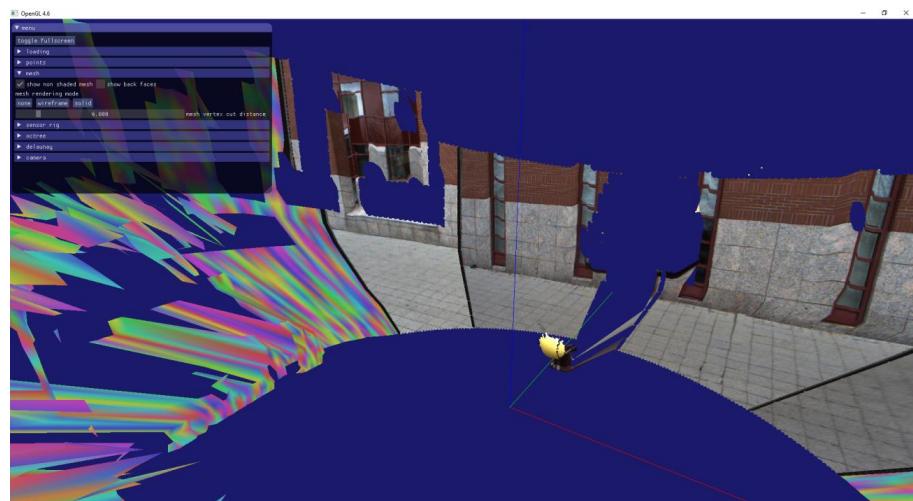
2.12. ábra. Mesh megjelenítés kikapcsolt mód



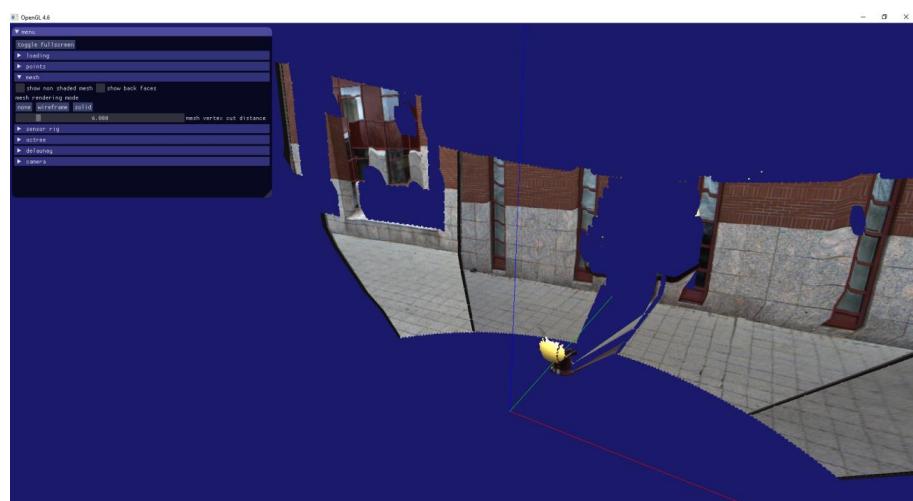
2.13. ábra. Mesh megjelenítés wireframe mód



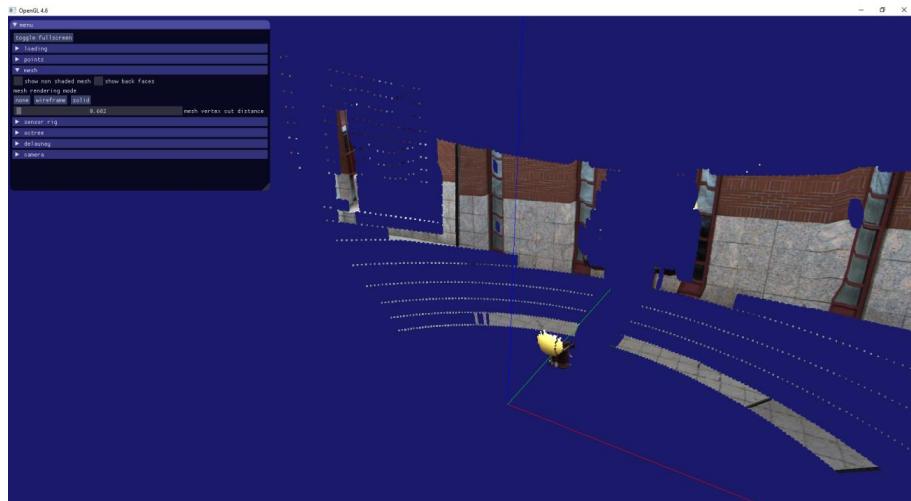
2.14. ábra. Mesh megjelenítés wireframe mód a nem textúrázott felületekkel



2.15. ábra. Mesh teljes megjelenítési mód a nem textúrázott felületekkel



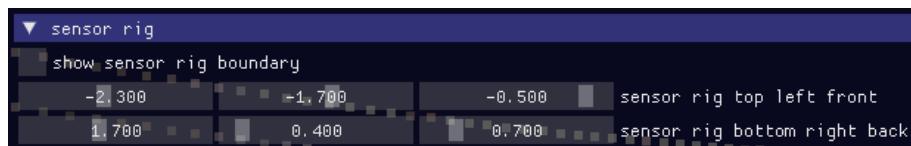
2.16. ábra. Mesh teljes megjelenítési mód



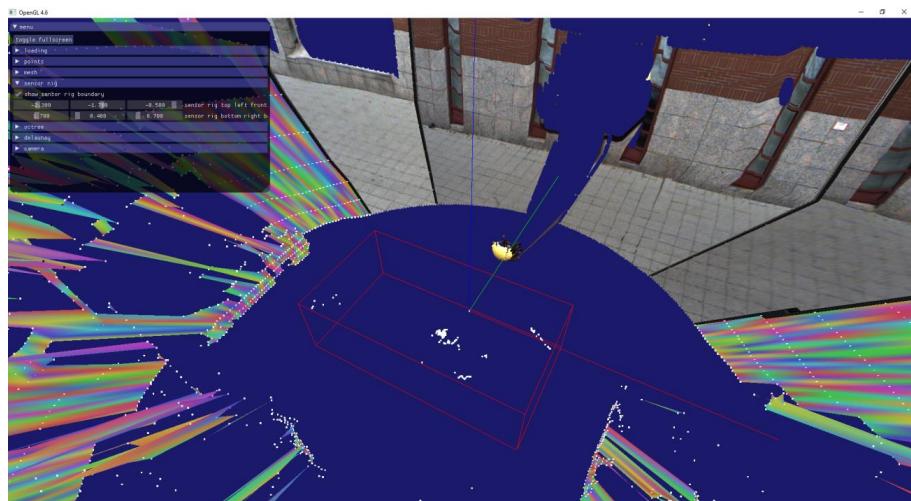
2.17. ábra. Mesh teljes megjelenítési mód a vágótávolság változtatásával

A szenzor közvetlen környezete

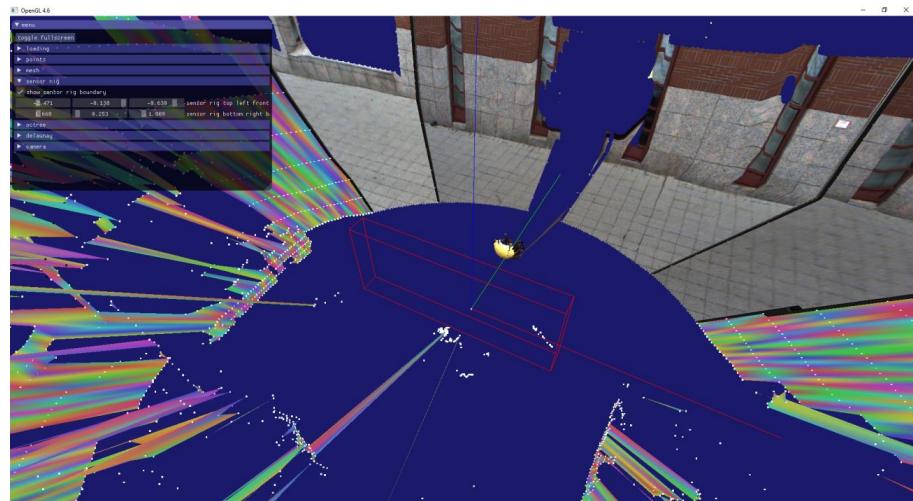
A megjelenítés során olyan felületrészek is képződtek, amelyek hibás pontok miatt jöttek létre. Ezek kiszűrése érdekében elérhető egy opció a szenzor környezetének egy téglalakkal való közelítése, amelynek területén belül eső pontokat nem használjuk fel a felületszámításhoz. A téglalék két átlós sarkának pozícióját változtathatjuk, amivel tetszőleges méretű területet vehetünk fel.



2.18. ábra. A szenzor környezete menü



2.19. ábra. A szenzor környezetének egy téglalakkal való közelítése



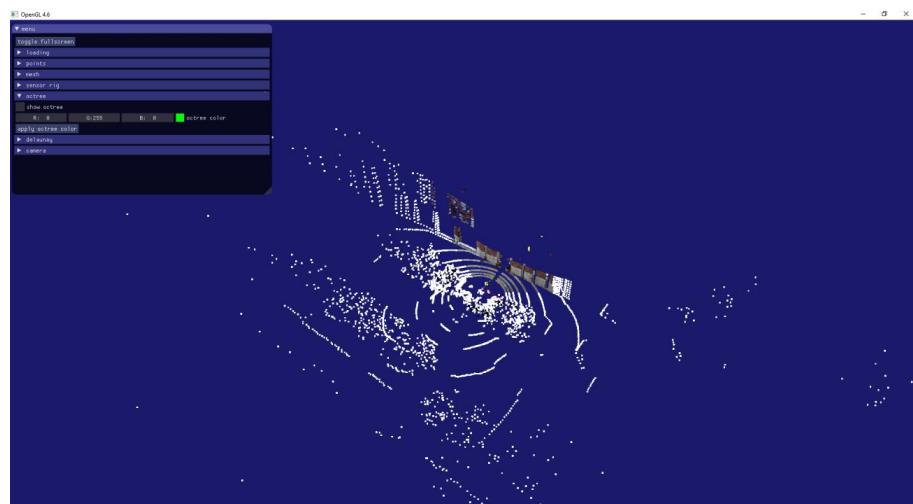
2.20. ábra. A szenzor környezetének átméretezése

Octree

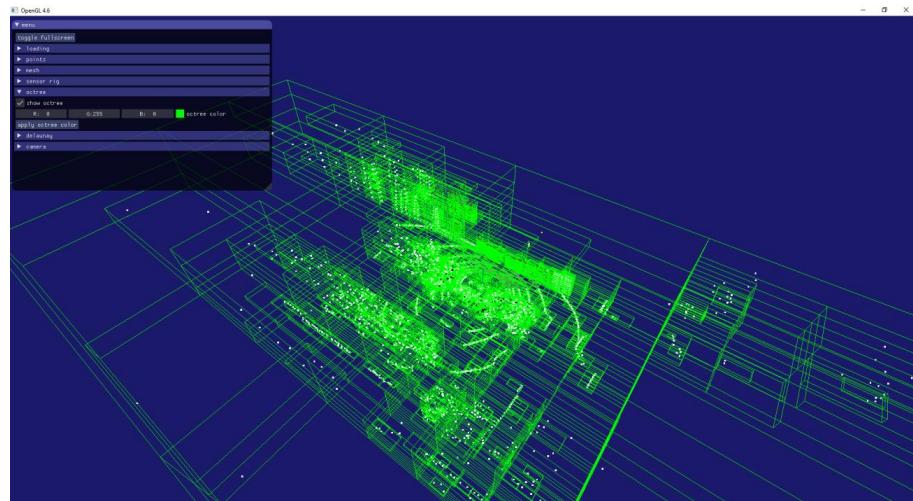
Az octree adatstruktúra megjeleníthető wireframe módban, lehetőség van továbbá a vonalak színének beállítására.



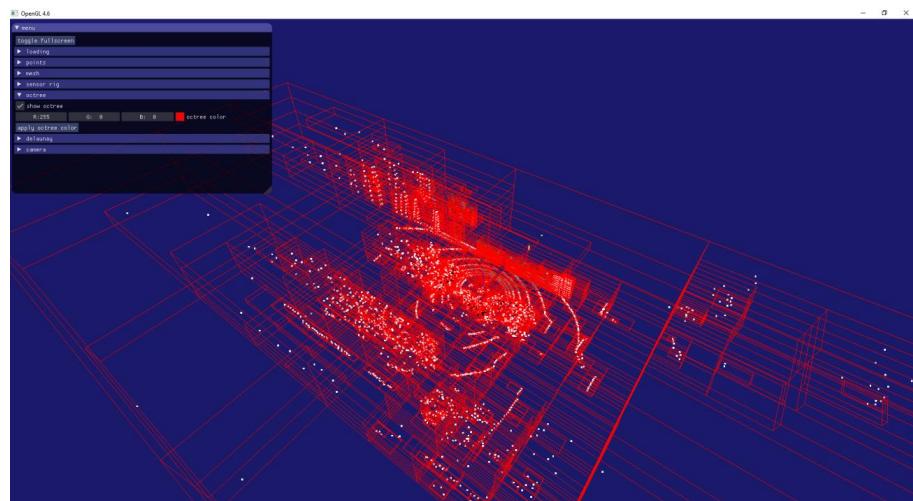
2.21. ábra. Octree menü



2.22. ábra. Octree kikapcsolt megjelenítéssel



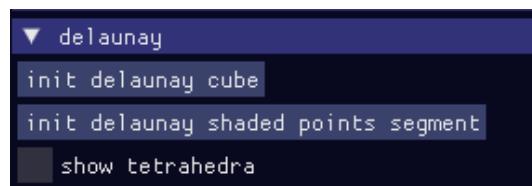
2.23. ábra. Octree wireframe megjelenítéssel



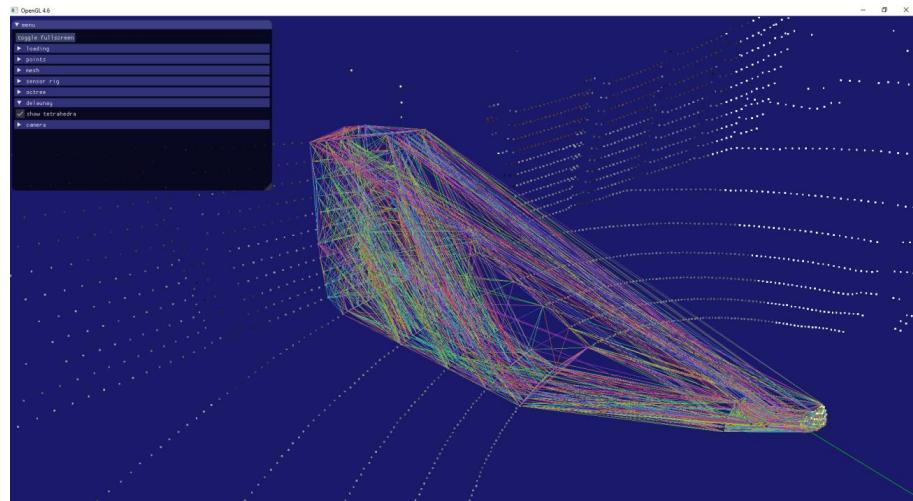
2.24. ábra. Octree átszínezése

Delaunay

A gombokkal válthatunk a teszt kocka és a pontok egy részéből generált tetraéderháló között. A checkbox-al megjeleníthető vagy elrejthető a háló.



2.25. ábra. Delaunay menü



2.26. ábra. Delaunay tetraéderháló wireframe megjelenítéssel

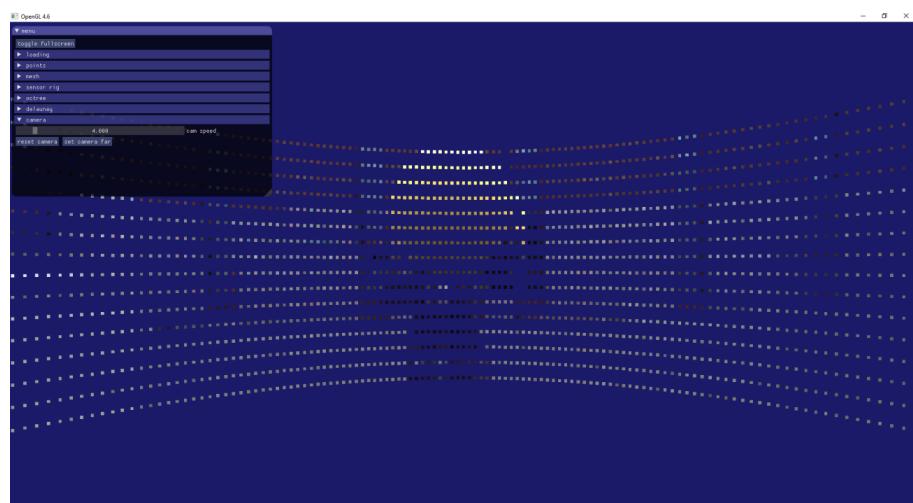
Kamera

Itt található egy csúszka a kamera sebességének beállításához, gombok a kamera pozíciójának alaphelyzetbe állításához, valamint egy távoli nézetbe mozgatásához.

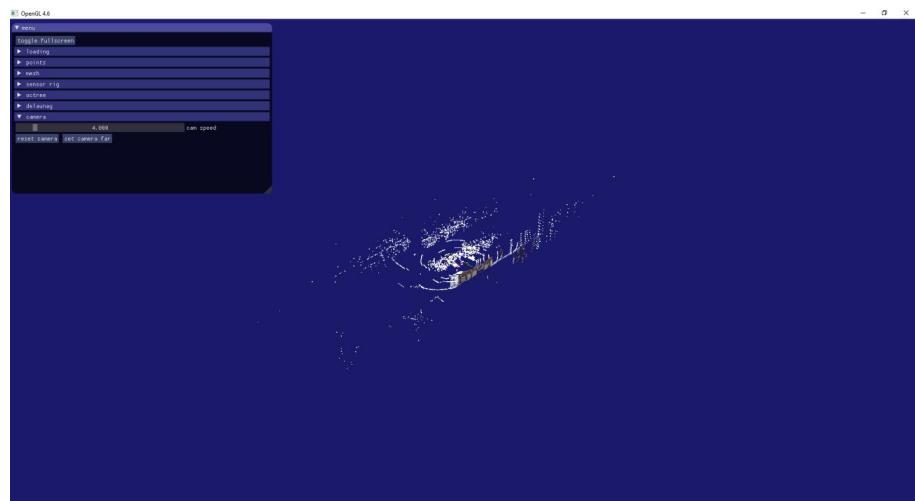


2.27. ábra. Kamera menü

Az alapértelmezett kamerapozícióban minden pont egyenlő távolságra látszik mivel a pontok mérete nem függ a kamerától vett távolságuktól, így a perspektivikus torzítás sem érvényes rájuk. Amint elkezdünk mozogni a kamerával a térbeli helyzetük jobban megmutatkozik az egymáshoz viszonyított pozíciójuk miatt.



2.28. ábra. Kamera pozíció alaphelyzetbe állítva



2.29. ábra. Kamera pozíció távoli pontba állítva

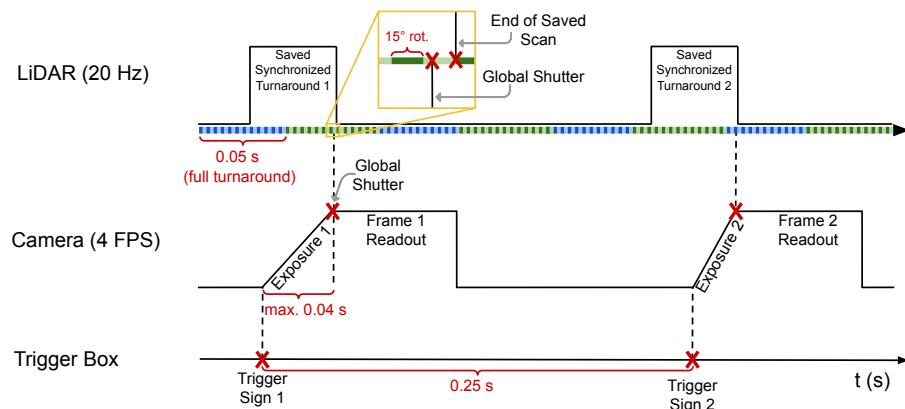
3. fejezet

Fejlesztői dokumentáció

3.1. A probléma részletes specifikációja

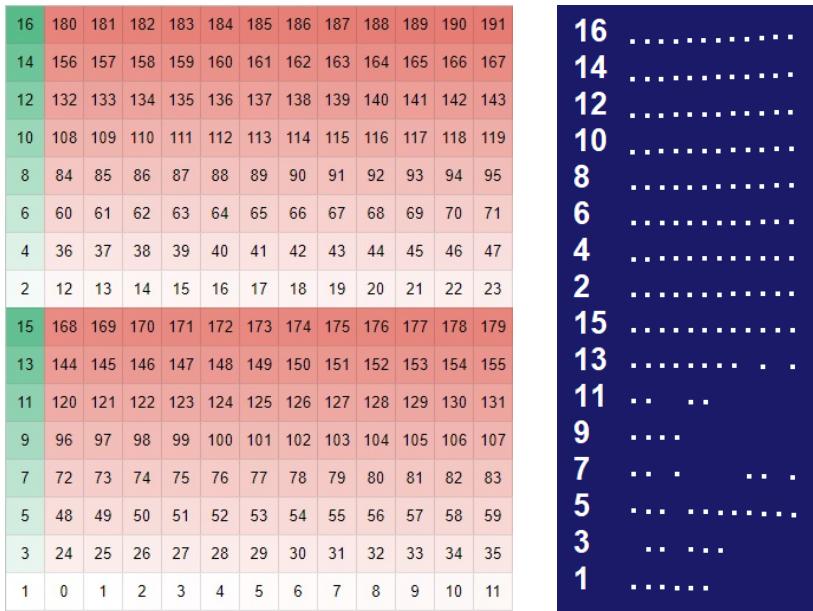
3.1.1. A LiDAR szenzor jellemzői, és az adatgenerálás

Az adatokat egy 16 sugarú 360 fokban körbe forgó LiDAR szenzorból kapom. Feltételezem, hogy az adatok időben szinkronizáltak, amit egy Arduino típusú kontroller áramkör biztosít. A LiDAR, a kamera és a kontroller működésének sematikus felépítését a 3.1 ábrán mutatom be.



3.1. ábra. Az eszközök szinkronizációja

A szenzor beállításai alapján jelenleg minden második pont duplikátum, amiket eldobok. A maradék pontok blokkosítva helyezkednek el. Térbeli helyzetüket a 3.2 ábrával illusztrálom. Zöld színnel jelöltetem a sorok számát, pirossal pedig a pontok sorszámát.



(a) A pontok eredeti sorrendje

(b) A pontok térbeli elhelyezkedése a programban

3.2. ábra. Pontok sorrendje és térbeli elhelyezkedése

Ahol az eszköz nem talál el felületet a lézersugárral, ott nem tudunk pontot megjeleníteni. Az adatstruktúrában viszont ilyenkor is létezik bejegyzés alapértelmezett értékekkel, ami a pontok sorrendjének meghatározásában nagy hasznunkra lesz. Ahol ezek a nem definiált pontok zavarnának a számításban, ott kiszűrjük őket.

A fent említett két pont-blokk alkot egy oszlopot, amelyet további oszlopok követnek az előzőtől balra lévő sorrendben egy körív mentén.

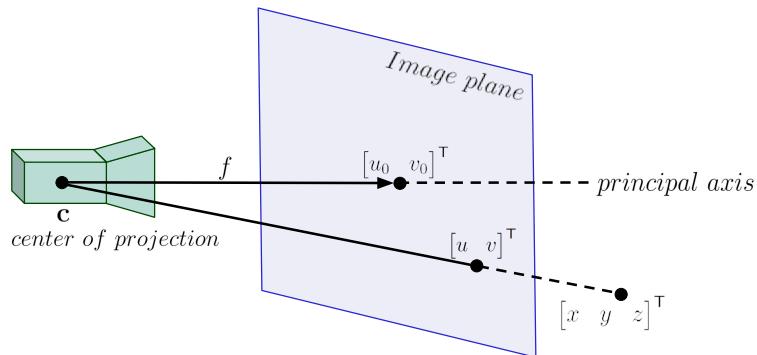
3.1.2. A kamerák jellemzői

A 3 darab digitális Hikvision MV-CA020-20GC kamera mindegyike egy 960x600 felbontású képet készít, amelyeket a pontok színezésére, valamint a generált felület textúrázására használunk fel. A szenzor és a kamerák az 1.1 ábrán láthatóak.

3.1.3. Kamera paraméterek betöltése

A LiDAR és a kamera koordinátarendszere közötti transzformációhoz szükséges paramétereit egy szöveges fájlból olvasom be. A fájl tartalmazza a kamera belső paramétereit (fókuszpont, uv paraméterek a 3.3 ábrán láthatóan), valamint a külső

paramétereket, mint a forgatás és pozíció. A kapott fájlt manuálisan átírtam a C++ által könnyebben beolvasható formátumra, ami a 3.4, 3.5 ábrákon látható.



3.3. ábra. Kamera modell

```

1 Belso parameterek (Dev0, Dev1, Dev2):
2   fu = 625;
3   u0 = 480;
4   fv = 625;
5   v0 = 300;
6
7 Kulso parameterek:
8 -Dev0:
9   R = [ 0.9230    -0.0066    -0.3847
10      0.3848     0.0203     0.9228
11      0.0018    -0.9998    0.0213];
12
13  t = [-0.0397    0.1842    -0.0944];
14
15 -Dev1:
16   R = [ 0.9999    0.0086    -0.0094
17      0.0095    -0.0110     0.9999
18      0.0085    -0.9999    -0.0111];
19
20  t = [-0.0463    0.0752    0.0932];
21
22 -Dev2:
23   R = [ 0.9543    0.0319    0.2971
24      -0.2969   -0.0113     0.9548
25      0.0338   -0.9994   -0.0014];
26
27  t = [0.1000    0.1962   -0.0663];

```

3.4. ábra. CameraParameters.txt

```

1 625 480 625 300
2 3
3 Dev0
4 0.9230 -0.0066 -0.3847 0.3848 0.0203 0.9228 0.0018 -0.9998 0.0213
5 -0.0397 0.1842 -0.0944
6 Dev1
7 0.9999 0.0086 -0.0094 0.0095 -0.0110 0.9999 0.0085 -0.9999 -0.0111
8 -0.0463 0.0752 0.0932
9 Dev2
10 0.9543 0.0319 0.2971 -0.2969 -0.0113 0.9548 0.0338 -0.9994 -0.0014
11 0.1000 0.1962 -0.0663

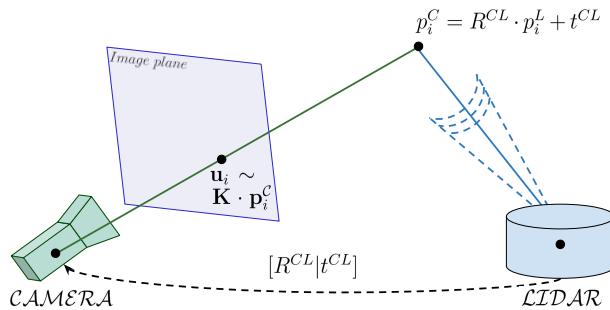
```

3.5. ábra. CameraParametersMinimal.txt

3.1.4. A LiDAR szenzor és a kamera közötti transzformáció

A LiDAR-kamera transzformációt a pontok megjelenítéséért felelős shader [13] programban számolom ki. A LiDAR-ról kamerára történő átalakítás a következőképpen írható le: $p_i^C = R^{CL} \cdot p_i^L + t^{CL}$. Az egyenletben a p_i^C és p_i^L a kamera és a LiDAR 3D koordinátái.

A kamera koordinátarendszerébe áttranszformált pontok pozícióiból ezután a K mátrixxal számolhatunk UV koordinátákat. Ezeket a normalizálás után közvetlenül használhatjuk a textúrából való színinformáció olvasására. A 3.6 ábrán látható módon az $u_i \sim K \cdot p_i^C$, vagyis az UV koordináta a K mátrix és az i indexű kamera pont szorzata.



3.6. ábra. A LiDAR szenzor és a kamera közötti transzformáció

3.1.5. A megjelenítési formák rövid leírása.

A programon belül lehetőség van a pontfelhő és a generált felület egymástól független megjelenítésére, valamint a textúrázatlan pontok és felületrészletek elrejtésére, amit a következőkben részletesebben tárgyalok.

3.2. A felhasznált módszerek leírása, a használt fogalmak definíciója

3.2.1. A fájlok beolvasása

A ponthalmazt és a képeket, valamint a kamera paramétereit az `inputs` mappából olvasom be, ami a projekt könyvtárában található. minden ponthalmaz külön mappában van elhelyezve, ami tartalmazza a pontok helyzeti és színinformációt tároló `.xyz` kiterjesztésű szöveges állományt, valamint a kamerák által készített képeket. A képek `DevX_Image` prefixtel vannak megjelölve, ahol az X karakter a kamera indexét jelöli, és jelen esetben a számozás nullától kezdődik.

A `.xyz` fájlt egy vertex tömb struktúrában tárolom az alkalmazáson belül, a képeket pedig egy textúra tömbben. Ezekből az adatokból készítem elő a megjelenítéshez szükséges adatstruktúrákat, amivel az OpenGL és a grafikus kártya dolgozni tud.

3.2.2. A pontok renderelése

Az előkészített szenzor és képi adatokat, valamint a rendereléshez szükséges egyéb változókat elküldöm a shader programnak amit a GPU-n futtatok.

A shader-ek a program indításakor töltődnek be, majd a rendereléskor a megfelelő shader aktiválása után felhasználhatjuk őket a rajzoláshoz. A `particle.vert` shader program lefut minden egyes pontra, és kiszámolja a szükséges transzformációkat, majd tovább küldi az adatokat a `particle.frag` shader-nek, ami kiolvassa a megfelelő helyről a textúrainformációt a pontok színezéséhez. A fragment shader kimenete határozza meg az adott pont színét.

3.2.3. Delaunay háromszögelés és tetraéderhálózás

A pontkból történő felület rekonstrukcióhoz először a Delaunay háromszögelést [3] vettem alapul, azon belül is a Bowyer–Watson [14] [15] inkrementális algoritmust. Ezt alakítottam át síkból térbeli megoldássá, ami háromszögek helyett tetraédereket alkalmaz. Az algoritmus kipróbálásakor észrevettem, hogy a pontfelhő felépítése miatt jelentős mennyiségű felesleges kapcsolatot is létrehoz a módszer, ami esetünkben nem előnyös. Az algoritmust

jelen formájában elvettem, mint lehetséges jelölt a felületrekonstrukcióra, viszont kiegészítő megoldásnak még jó lehet. A létrehozott tetraédereket wireframe, vagyis a pontok közötti vonalakkal ábrázolva jelenítem meg.

3.2.4. Octree

A pontokhoz továbbá egy térbeli gyorsítószerkezetet is implementáltam, ami a program továbbfejlesztésekor hasznos lehet. Az octree [7] adatstruktúrát választottam, mert ez jelentősen lecsökkenti a szomszédos pontok lekérdezését térbeli elhelyezkedés alapján. Az octree minden dimenzió mentén feldarabolja a teret hierarchikus módon szegmensekre és alszegmensekre, ami egy fa struktúrát hoz létre. Az így létrejött adatszerkezet a fákra jellemző előnyökkel rendelkezik.

3.2.5. Mesh építés az adatok struktúráját figyelembe véve

Az előző módszer után, ami rendezetlen pontfelhőt feltételez, megpróbáltam figyelembe venni az kapott pontok struktúráját. A könnyebb feldolgozás érdekében a pontok sorrendjét a fent említett blokkos elrendezésből folytonos struktúrába rendeztem át, ahol könnyű a sorrend alapján az indexek felhasználásával háromszögeket képezni a pontkból. Az átsorrendezés megoldotta az indexekben való nagy ugrásokat a blokkok határain, valamint a soros haladás helyett az oszlopos haladási irány miatt nagy ugrások nélkül folytathatjuk a háromszögek létrehozását.

16	180	181	182	183	184	185	186	187	188	189	190	191		11	10	9	8	7	6	5	4	3	2	1	0
14	156	157	158	159	160	161	162	163	164	165	166	167		191	175	159	143	127	111	95	79	63	47	31	15
12	132	133	134	135	136	137	138	139	140	141	142	143		190	174	158	142	126	110	94	78	62	46	30	14
10	108	109	110	111	112	113	114	115	116	117	118	119		189	173	157	141	125	109	93	77	61	45	29	13
8	84	85	86	87	88	89	90	91	92	93	94	95		188	172	156	140	124	108	92	76	60	44	28	12
6	60	61	62	63	64	65	66	67	68	69	70	71		187	171	155	139	123	107	91	75	59	43	27	11
4	36	37	38	39	40	41	42	43	44	45	46	47		186	170	154	138	122	106	90	74	58	42	26	10
2	12	13	14	15	16	17	18	19	20	21	22	23		185	169	153	137	121	105	89	73	57	41	25	9
15	168	169	170	171	172	173	174	175	176	177	178	179		184	168	152	136	120	104	88	72	56	40	24	8
13	144	145	146	147	148	149	150	151	152	153	154	155		183	167	151	135	119	103	87	71	55	39	23	7
11	120	121	122	123	124	125	126	127	128	129	130	131		182	166	150	134	118	102	86	70	54	38	22	6
9	96	97	98	99	100	101	102	103	104	105	106	107		181	165	149	133	117	101	85	69	53	37	21	5
7	72	73	74	75	76	77	78	79	80	81	82	83		180	164	148	132	116	100	84	68	52	36	20	4
5	48	49	50	51	52	53	54	55	56	57	58	59		179	163	147	131	115	99	83	67	51	35	19	3
3	24	25	26	27	28	29	30	31	32	33	34	35		178	162	146	130	114	98	82	66	50	34	18	2
1	0	1	2	3	4	5	6	7	8	9	10	11		177	161	145	129	113	97	81	65	49	33	17	1
														176	160	144	128	112	96	80	64	48	32	16	0

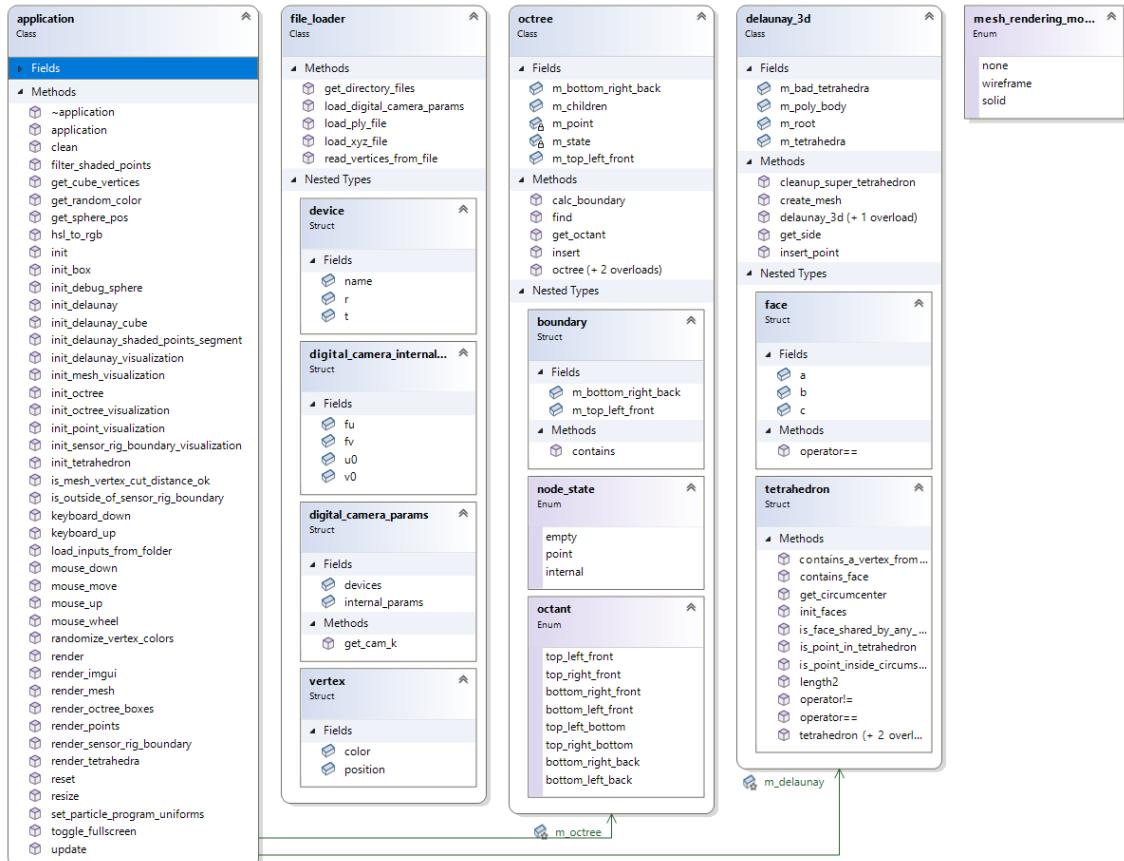
(a) A pontok eredeti sorrendje

(b) A pontok átrendezett sorrendje

3.7. ábra. A pontok indexelése

3.3. A szoftver szerkezete

3.3.1. Osztály diagram



3.8. ábra. Osztály diagram

3.3.2. main.cpp

Felelős az SDL ablak és az OpenGL-es környezet létrehozásáért, valamint a felhasználói interakció továbbításáért az OpenGL alkalmazásba. Az osztály vázlatos működését az 1. algoritmussal mutatom be.

1. algoritmus Main

```

1: procedure MAIN
2:   Initialize system dependencies (SDL, OpenGL, GLEW, ImGui)
3:   if any initialization failed then
4:     Print error and exit program
5:   end if
6:   Create window and OpenGL context
7:   if creation failed then
8:     Print error and exit program
9:   end if
10:  Set debug callback if in debug context
11:  Initialize and start application
12:  Start application loop:
13:  while not quitting do
14:    Start event loop:
15:    for each event do
16:      Process event with ImGui and application
17:    end for
18:    Update the application
19:    Render the application
20:  end while
21:  Clean up application
22:  Shut down system dependencies, delete OpenGL context, destroy window
23: end procedure

```

3.3.3. application.h/cpp

A header fájlban deklarálom a függvényeket és a változókat, amelyeket a pontok és a mesh megjelenítéséhez használok. Az alkalmazás három legfontosabb eleme az előkészítő, a frissítő és a megjelenítő metódusok.

Az init függvény

Az `init` függvény állítja be a virtuális kamera paramétereit, a háttérszínt, betölti a shader programokat, betölt egy előre megadott input mappát, és előkészíti a teszteléshez a gömböt, valamint a szenzor környezetét jelző téglatestet.

```

1 bool application::init(SDL_Window* window) {
2   m_window = window;
3   m_virtual_camera.SetProj(glm::radians(60.0f), 1280.0f / 720.0f, 0.01f, 1000.0f);
4
5   glClearColor(0.1f, 0.1f, 0.41f, 1);
6   glEnable(GL_DEPTH_TEST);
7
8   m_axes_program.Init({
9     {GL_VERTEX_SHADER, "shaders/axes.vert"},
```

```

10     {GL_FRAGMENT_SHADER, "shaders/axes.frag"}
11 };
12 m_particle_program.Init({
13     {GL_VERTEX_SHADER, "shaders/particle.vert"},
14     {GL_FRAGMENT_SHADER, "shaders/particle.frag"}
15 },
16 {
17     {0, "vs_in_pos"}, {1, "vs_in_col"}, {2, "vs_in_tex"}
18 });
19 m_wireframe_program.Init({
20     {GL_VERTEX_SHADER, "shaders/wireframe.vert"},
21     {GL_FRAGMENT_SHADER, "shaders/wireframe.frag"}
22 },
23 {{0, "vs_in_pos"}, {1, "vs_in_col"},}});
24
25 load_inputs_from_folder("inputs\\garazs_kijarat");
26 init_debug_sphere();
27
28 init_sensor_rig_boundary_visualization();
29
30 return true;
31 }

```

3.1. forráskód. Az init függvény

Az update függvény

Az update függvény frissíti a vonalvastagságot, a hátlapok eldobását, a virtuális kamerát és a pontok animált, egyesével megjelenítésének számolását.

```

1 void application::update() {
2     glLineWidth(m_line_width);
3
4     if (m_show_back_faces) {
5         glDisable(GL_CULL_FACE);
6     } else {
7         glEnable(GL_CULL_FACE);
8         glCullFace(GL_BACK);
9     }
10    static Uint32 last_time = SDL_GetTicks();
11    const float delta_time = (float)(SDL_GetTicks() - last_time) / 1000.0f;
12    m_virtual_camera.Update(delta_time);
13    last_time = SDL_GetTicks();
14
15    if (m_auto_increment_rendered_point_index && m_render_points_up_to_index < m_vertices.size()) {
16        m_render_points_up_to_index += 1;
17    }
18 }

```

3.2. forráskód. Az update függvény

A render függvény

A `render` függvény felelős a szín és mélység pufferek ürítéséért, a tengelyek, a pontok, a teszt gömb, az octree, a szenzor környezet téglatest, a mesh, a Delaunay tetraéderháló és a GUI kirajzolásáért.

```

1 void application::render() {
2     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
3
4     if (m_show_axes) {
5         m_axes_program.Use();
6         m_axes_program.SetUniform("mvp", m_virtual_camera.GetViewProj());
7         glDrawArrays(GL_LINES, 0, 6);
8     }
9
10    if (m_show_points)
11        render_points(m_particle_vao, m_render_points_up_to_index);
12
13    if (m_show_debug_sphere)
14        render_points(m_debug_sphere_vao, m_debug_sphere.size());
15
16    if (m_show_octree)
17        render_octree_boxes();
18
19    if (m_show_sensor_rig_boundary) {
20        init_sensor_rig_boundary_visualization();
21        render_sensor_rig_boundary();
22    }
23
24    if (m_mesh_rendering_mode != none) {
25        if (m_mesh_rendering_mode == solid) {
26            glPolygonMode(GL_FRONT, GL_FILL);
27        } else if (m_mesh_rendering_mode == wireframe) {
28            glPolygonMode(GL_FRONT, GL_LINE);
29        }
30        init_mesh_visualization();
31        render_mesh();
32    }
33
34    if (m_show_tetrahedra)
35        render_tetrahedra();
36
37    render_imgui();
38 }
```

3.3. forráskód. A render függvény

A mappa betöltő függvény

A `load_inputs_from_folder` függvény kiolvassa az összes fájlnevet a megadott elérési úton lévő mappából és ha talál benne megfelelő nevű képeket akkor betölti őket textúraként, valamint, ha talál pontfelhőt akkor azt is beolvassa egy vertex tömbbe. Beolvassa továbbá a kamera paramétereket az `inputs` mappából és ezután előkészíti a pontok az octree a Delaunay tetraéderháló és a mesh megjelenítését. Ezek minden függenek a kiválasztott ponthalmaztól tehát minden betöltéskor inicializálni kell őket.

```

1 void application::load_inputs_from_folder(const std::string& folder_name) {
2     const std::vector<std::string> file_paths = file_loader::get_directory_files(folder_name);
3     std::string xyz_file;
4
5     for (const std::string& file : file_paths) {
6         if (file.find("Dev0") != std::string::npos) {
7             m_digital_camera_textures[0].FromFile(file);
8             std::cout << "Loaded texture from " << file << std::endl;
9         } else if (file.find("Dev1") != std::string::npos) {
10            m_digital_camera_textures[1].FromFile(file);
11            std::cout << "Loaded texture from " << file << std::endl;
12        } else if (file.find("Dev2") != std::string::npos) {
13            m_digital_camera_textures[2].FromFile(file);
14            std::cout << "Loaded texture from " << file << std::endl;
15        } else if (file.find(".xyz") != std::string::npos) {
16            xyz_file = file;
17        }
18    }
19
20    m_vertices = file_loader::load_xyz_file(xyz_file);
21    std::cout << "Loaded " << m_vertices.size() << " points from " << xyz_file << std::endl;
22    m_render_points_up_to_index = m_vertices.size() - 16;
23    m_digital_camera_params =
24        file_loader::load_digital_camera_params("inputs\\CameraParametersMinimal.txt");
25    std::cout <<
26        "Loaded digital camera parameters from inputs\\CameraParametersMinimal.txt"
27        << std::endl;
28
29    init_point_visualization();
30    randomize_vertex_colors(m_vertices);
31    init_octree(m_vertices);
32    init_octree_visualization(&m_octree);
33    init_delaunay_shaded_points_segment();
34    init_mesh_visualization();
35 }
```

3.4. forráskód. A `load_inputs_from_folder` függvény

További függvények

Ezeken kívül a függvényeket két fő csoportba sorolhatjuk. Az első az inicializáló, vagyis előkészítő függvény, ezekkel álltom elő a második csoport, vagyis a renderelésért felelős függvények bemenetét. Az adatokat a GPU által feldolgozható struktúrákba kell alakítani. Az vertexeket (csúcspont) ArrayBuffer-be töltöm, opcionálisan használok IndexBuffer-t, ahol egy vertexet többször is fel akarok használni (pl mesh) majd a két buffert egy VertexArrayObject-be töltöm.

A renderelő függvények feladata az előkészített adatok megjelenítése. A rajzoláskor a VAO-t (VertexArrayObject) csatolom (`vao.Bind()`) majd a rajzolás után leválasztom (`vao.Unbind()`).

A rajzoláshoz a shader programot kell használni a `program.Use()` metódus meghívásával. Ha megadtunk a shaderben bemenő paramétereket azokat is itt kell leküldeni a GPU-nak a `program.SetUniform(name, value)`-el.

A rajzolás többféleképpen történhet, attól függően, hogy milyen típusú adatokat szeretnék megjeleníteni. Ha sorfolytonos adatokat renderelek (pl pontok) akkor `glDrawArrays()`-t ha pedig saját indexelést használok (pl vonalak, háromszögek) akkor a `glDrawElements()` metódust használom.

Példaként a pontok megjelenítésének előkészítését és renderelését hozom a következő forráskód részletekben.

```

1 void application::init_point_visualization() {
2     m_particle_buffer.BufferData(m_vertices);
3     m_particle_vao.Init({
4         {
5             AttributeData{
6                 0, 3, GL_FLOAT, GL_FALSE, sizeof(file_loader::vertex),
7                 (void*)offsetof(file_loader::vertex, position)
8             },
9             m_particle_buffer
10        },
11        {
12            AttributeData{
13                1, 3, GL_FLOAT, GL_FALSE, sizeof(file_loader::vertex),
14                (void*)offsetof(file_loader::vertex, color)
15            },
16            m_particle_buffer
17        }
18    });
19 }
```

3.5. forráskód. Az init_point_visualization függvény

```
1 void application::render_points(VertexArrayObject& vao, const size_t size) {
2     vao.Bind();
3     set_particle_program_uniforms(m_show_non_shaded_points);
4     glEnable(GL_PROGRAM_POINT_SIZE);
5     m_particle_program.SetUniform("point_size", m_point_size);
6     glDrawArrays(GL_POINTS, 0, size);
7     glDisable(GL_PROGRAM_POINT_SIZE);
8     vao.Unbind();
9 }
```

3.6. forráskód. A render_points függvény

3.3.4. file_loader.h/cpp

Tartalmazza a fájlból betölthető adattípusok (struct) deklarációját, valamint a betöltő függvényeket. A digitális kamera paramétereit a load_digital_camera_params(filename) függvény a pontokét pedig a load_xyz_file(file, vertex_count) valósítja meg.

3.3.5. octree.h

Az octree [16] adatszerkezet ebben az implementációban csak a levelekben tárol adatot és lelevelenként egy bejegyzést engedélyezünk. A csúcsok lehetnek üresek, belső csúcsok vagy pontot tartalmazó levelek. minden csúcsnak lehetnek gyerekei, amelyek az octree esetében 8 darab alszegmenst jelent ($3 \text{ dimenzió} = 3 \text{ felezés} = 2^3 = 8$). Üres állapotban még nincs se pont, se gyerek csúcs, belső állapotban csak gyerekei vannak, levél állapotban pedig csak pontot tartalmaz a csúcs.

Pont beszúrásakor megnézzük a gyökér csúcsot, ha a pozíció a területén belülre esik akkor megnézzük, hogy üres-e, ha igen akkor elmentjük a pontot, ha nem akkor megkeressük melyik gyerekébe esik a pont és oda szúrjuk be.

2. algoritmus Octree Insert Function

```

1: procedure INSERT(point_to_insert)
2:   if find(point_to_insert) then
3:     print "Point already exists in the tree"
4:     return
5:   end if
6:   if point_to_insert is out of bounds then
7:     print "Point is out of bounds."
8:     return
9:   end if
10:  mid  $\leftarrow$  average(m_top_left_front, m_bottom_right_back)
11:  octant  $\leftarrow$  get_octant(point_to_insert, mid)
12:  if m_children[octant].state is internal then
13:    m_children[octant].insert(point_to_insert)
14:    return
15:  else if m_children[octant].state is empty then
16:    delete m_children[octant]
17:    m_children[octant]  $\leftarrow$  new octree(point_to_insert)
18:    return
19:  else
20:    already_stored_point  $\leftarrow$  m_children[octant].point
21:    delete m_children[octant]
22:    m_children[octant]  $\leftarrow$  new octree based on octant
23:    m_children[octant].insert(already_stored_point)
24:    m_children[octant].insert(point_to_insert)
25:    m_children[octant].point  $\leftarrow$  null
26:    m_children[octant].state  $\leftarrow$  internal
27:  end if
28: end procedure

```

3.3.6. delaunay_3d.h

A Delaunay-háromszögelés egy adott P diszkrét pontkészletre vonatkozik, ahol $DT(P)$, és P-ben egyetlen pont sem található meg bármely $DT(P)$ -beli háromszög köré írt körben. Az algoritmus maximalizálja a háromszögek minimális szögét, így általában elkerüli a nagyon vékony háromszögeket. A módszert Boris Delaunay-ról nevezték el, aki 1934-ben dolgozott ezen a témaén.

Az egy vonalra eső pontkészletre nem értelmezett a Delaunay-háromszögelés. Négy vagy több, ugyanazon a körön lévő pont esetén (pl. egy négyszög csúcsain) a Delaunay-háromszögelés nem egyértelmű. Ebben az esetben két lehetséges háromszögelés is megfelel a "Delaunay-feltételnek", hogy minden háromszög köré írt köreinek belső területei üresek legyenek.

A körülírt gömböket figyelembe véve a Delaunay-háromszögelés fogalma kiterjed három és magasabb dimenziókra is.

Több módszer is létezik a háromszögelésre, amelyek közül a Bowyer–Watson algoritmust választottam, mert viszonylag egyszerű implementálni és inkrementálisan feldolgozható vele a ponthalmaz.

A Bowyer–Watson inkrementális algoritmus egyenként adja hozzá a pontokat a pontok egy részhalmazának már érvényes Delaunay-háromszögeléséhez. minden beszúrást követően törlődnek a háromszögek, melyek köré írt körök tartalmazzák az új pontot. Az így keletkezett sokszög alakú lyukat újraháromszögeljük az új pont használatával. minden a sokszöget alkotó él új háromszöget alkot az éppen beszúrt ponttal. A legelső háromszöget úgy vesszük fel, hogy minden pont a területére essen. Az utolsó pont beszúrása után törlünk minden háromszöget, aminek csúcspontjainak egyike megegyezik a legelső háromszög csúcspontjainak egyikével.

3. algoritmus BowyerWatson algorithm

Require: pointList is a set of coordinates defining the points to be triangulated

```
1: triangulation ← empty triangle mesh data structure
2: add super-triangle to triangulation
3: for each point in pointList do
4:   badTriangles ← empty set
5:   for each triangle in triangulation do
6:     if point is inside circumcircle of triangle then
7:       add triangle to badTriangles
8:     end if
9:   end for
10:  polygon ← empty set
11:  for each triangle in badTriangles do
12:    for each edge in triangle do
13:      if edge is not shared by any other triangles in badTriangles then
14:        add edge to polygon
15:      end if
16:    end for
17:    remove triangle from triangulation
18:  end for
19:  for each edge in polygon do
20:    newTri ← form a triangle from edge to point
21:    add newTri to triangulation
22:  end for
23: end for
24: for each triangle in triangulation do
25:   if triangle contains a vertex from original super-triangle then
26:     remove triangle from triangulation
27:   end if
28: end for
```

Ensure: triangulation

A fent leírt algoritmust alapul véve terjesztettem ki a megoldásomat a harmadik dimenzióba. A háromszögek helyett tetraédereket használtam, befoglaló körök helyett gömböket, valamint az élek helyett háromszögeket a törléskor keletkezett lyuk oldalainak definiálására.

A megoldás ígéretesnek mutatkozott a teszt kocka pontjainak összekötésekor, kis elemszám miatt viszont nem tudtam messze menő következetiséseket levonni. A szenzorból érkező ponthalmazra alkalmazva már jelentős növekedést láttam a csúcsok közötti kapcsolatok számában, amire nem találtam egyértelmű megoldást. Az algoritmust jelen formájában alkalmatlannak ítélem a probléma megoldására, viszont lokális háromszögelési problémák esetén az adathalmaz egy-egy kis részén felhasználható lehet az implementáció.

3.3.7. A LiDAR ponthalmaz struktúráját felhasználó mesh generálás

A következő megoldásomban a ponthalmaz struktúráját figyelembe véve készíték felületet háromszögek segítségével. Az eredeti sorrendet átrendezve a blokkosodás megszűntetése után sorfolytonosan dolgozom fel a pontokat.

4. algoritmus Point order dependent mesh generation

```

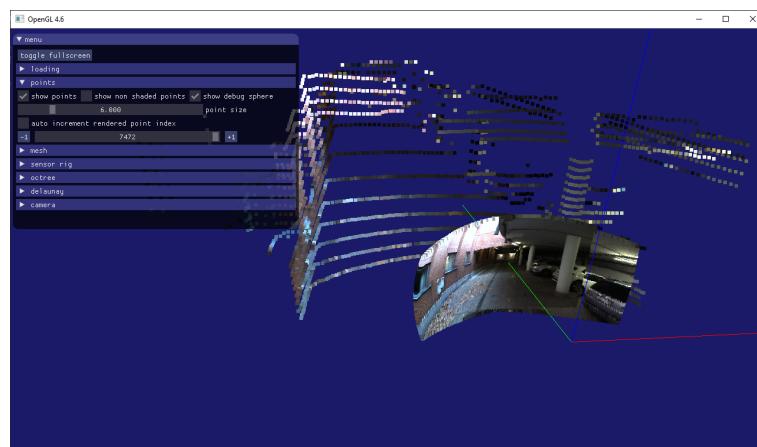
1: for i from 0 to max do
2:   if i is not in last column or last row then
3:     if check if triangle is not too long(i, i + 1, i + 17) is true then
4:       Add i to m_mesh_indices
5:       Add i + 1 to m_mesh_indices
6:       Add i + 17 to m_mesh_indices
7:     end if
8:     if check if triangle is not too long(i, i + 17, i + 16) is true then
9:       Add i to m_mesh_indices
10:      Add i + 17 to m_mesh_indices
11:      Add i + 16 to m_mesh_indices
12:    end if
13:  end if
14: end for

```

3.4. Tesztelés

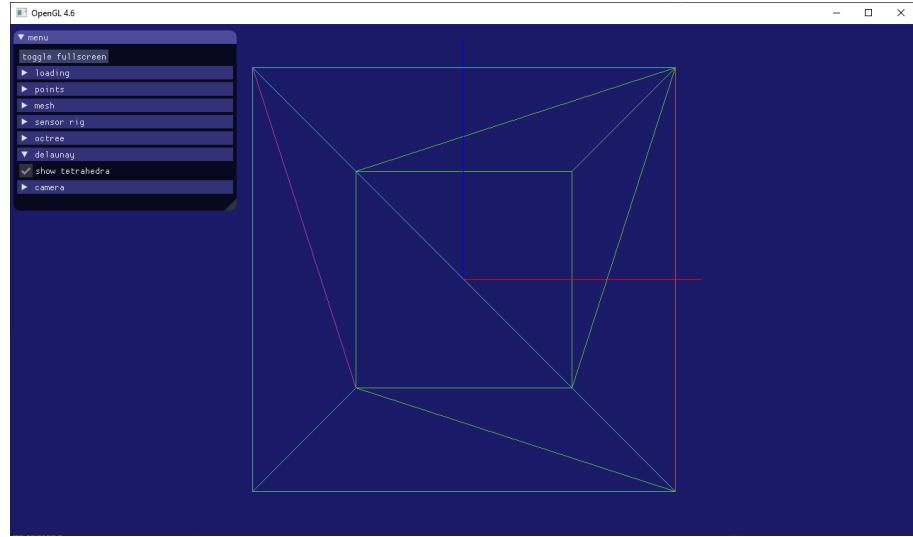
A program természete miatt főleg vizuálisan tesztelhetőek a megoldások. A helyes működés ellenőrzése érdekében az alábbi teszteket használom.

3.4.1. Textúra vetítés ellenőrzése gömb ponthalmazzal

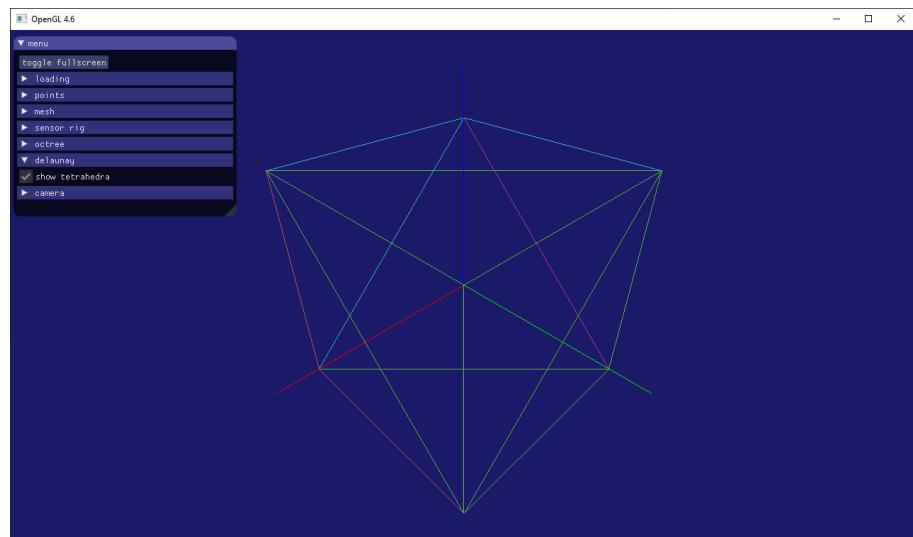


3.9. ábra. Teszt gömb

3.4.2. Delaunay tetraéderháló kocka csúcsponjtaira



3.10. ábra. Teszt Delaunay kocka - oldal nézet



3.11. ábra. Teszt Delaunay kocka - sarok nézet

4. fejezet

Összegzés

Dolgozatomban egy olyan alkalmazást valósítottam meg, amely a háromdimenziós térben jelenít meg valós helyszíneken bescannelt adatokat. A programban lehetőség van különböző mappák betöltésére, amelyek egy LiDAR szenzorból generált ponthalmazt és kamera képeket tartalmaznak. A menüben változatos szempontok alapján testreszabható az adatok megjelenítése. A virtuális kamerával szabadon mozoghatunk a térben a ponthalmaz vagy az abból generált felület körül.

A program kiindulási alapjának az számítógépes grafika tárgyakon használt projektet vettettem, mivel azt már kezdő szinten ismertem. A fő technológiák, amiket felhasználtam a C++ és az OpenGL, amelyeket a dolgozat elkészítése során kellett mélyebben megismernem.

A megvalósítás során megpróbáltam felhasználni a Delaunay háromszögelést, és az alapján tetraéderhálót generálni. Implementáltam az octree adatszerkezetet és mindeneknek vizuális megjelenítési lehetőséget készítettem. Ezek után egy hatékony felületgenerálási algoritmust írtam, ami figyelembe veszi a szenzoradatok struktúráját.

Köszönetnyilvánítás

Szeretném megköszönni a témavezetőmnek az egész féléves munkáját, a szakmai útmutatást és a hasznos tanácsokat a személyes konzultációk és az online alkalmak során.

Irodalomjegyzék

- [1] NOAA. *What is lidar?* URL: "<https://oceanservice.noaa.gov/facts/lidar.html>" (elérés dátuma 2023. 05. 26.).
- [2] Radu Bogdan Rusu és Steve Cousins. „3D is here: Point Cloud Library (PCL)”. *2011 IEEE International Conference on Robotics and Automation*. 2011. máj., 1–4. old. DOI: [10.1109/ICRA.2011.5980567](https://doi.org/10.1109/ICRA.2011.5980567).
- [3] Boris Delaunay és tsai. „Sur la sphere vide”. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk* 7.793-800 (1934), 1–2. old.
- [4] ISO/IEC. *Programming Languages — C++*. Draft International Standard N4660. 2017. márc. URL: <https://web.archive.org/web/20170325025026/http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4660.pdf>.
- [5] Mason Woo és tsai. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [6] ELTE. *Grafika BSc Gyakorlat anyagok*. URL: "<http://cg.elte.hu/index.php/grafika-bsc-gyakorlat-anyagok/>" (elérés dátuma 2023. 05. 26.).
- [7] Donald Meagher. „Geometric modeling using octree encoding”. *Computer graphics and image processing* 19.2 (1982), 129–147. old.
- [8] SDL Community. *SDL official website*. URL: "<https://www.libsdl.org/index.php>" (elérés dátuma 2023. 05. 26.).
- [9] Nigel Stewart. *GLEW Website*. URL: "<https://glew.sourceforge.net/>" (elérés dátuma 2023. 05. 26.).
- [10] Khronos Group. *Khronos Releases OpenGL 4.6 with SPIR-V Support*. URL: "<https://www.khronos.org/news/press/khronos-releases-opengl-4.6-with-spir-v-support>" (elérés dátuma 2023. 05. 26.).

- [11] Omar Cornut. *ImGui*. URL: "<https://github.com/ocornut/imgui>" (elérés dátuma 2023. 05. 26.).
- [12] Microsoft. *Visual Studio*. URL: "<https://visualstudio.microsoft.com/>" (elérés dátuma 2023. 05. 26.).
- [13] Joey de Vries. *Shaders*. URL: "<https://learnopengl.com/Getting-started/Shaders>" (elérés dátuma 2023. 05. 26.).
- [14] A. Bowyer. „Computing Dirichlet tessellations**”. *The Computer Journal* 24.2 (1981. jan.), 162–166. old. ISSN: 0010-4620. DOI: [10.1093/comjnl/24.2.162](https://doi.org/10.1093/comjnl/24.2.162). eprint: <https://academic.oup.com/comjnl/article-pdf/24/2/162/967239/240162.pdf>. URL: <https://doi.org/10.1093/comjnl/24.2.162>.
- [15] D. F. Watson. „Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes**”. *The Computer Journal* 24.2 (1981. jan.), 167–172. old. ISSN: 0010-4620. DOI: [10.1093/comjnl/24.2.167](https://doi.org/10.1093/comjnl/24.2.167). eprint: <https://academic.oup.com/comjnl/article-pdf/24/2/167/967258/240167.pdf>. URL: <https://doi.org/10.1093/comjnl/24.2.167>.
- [16] Parth Maniyar. *Octree / Insertion and Searching*. 2023. URL: "<https://www.geeksforgeeks.org/octree-insertion-and-searching/>" (elérés dátuma 2023. 05. 26.).

Ábrák jegyzéke

1.1.	A LiDAR szenzor és a felhasznált kamerák kiemelése	4
2.1.	Az OGLPack és a subst parancs	6
2.2.	A program indítása Visual Studio-ból	6
2.3.	CMD ablak	7
2.4.	A grafikus ablak indítás után	7
2.5.	A teljesen kinyitott menü	8
2.6.	Teljes képernyős mód gomb, tengelyek mutatása checkbox, vonalvastagság csúszka	9
2.7.	Mappa betöltés menü	9
2.8.	Pontok menü	9
2.9.	Pontok menü alapállapot	10
2.10.	Pontok menü megváltoztatott értékekkel	10
2.11.	Mesh menü	11
2.12.	Mesh megjelenítés kikapcsolt mód	11
2.13.	Mesh megjelenítés wireframe mód	11
2.14.	Mesh megjelenítés wireframe mód a nem textúrázott felületrészekkel .	12
2.15.	Mesh teljes megjeleníti mód a nem textúrázott felületrészekkel . .	12
2.16.	Mesh teljes megjeleníti mód	12
2.17.	Mesh teljes megjeleníti mód a vágótávolság változtatásával	13
2.18.	A szenzor környezete menü	13
2.19.	A szenzor környezetének egy téglatesttel való közelítése	13
2.20.	A szenzor környezetének átméretezése	14
2.21.	Octree menü	14
2.22.	Octree kikapcsolt megjelenítéssel	14
2.23.	Octree wireframe megjelenítéssel	15
2.24.	Octree átszínezése	15
2.25.	Delaunay menü	15

2.26. Delaunay tetraéderháló wireframe megjelenítéssel	16
2.27. Kamera menü	16
2.28. Kamera pozíció alaphelyzetbe állítva	16
2.29. Kamera pozíció távoli pontba állítva	17
 3.1. Az eszközök szinkronizációja	18
3.2. Pontok sorrendje és térbeli elhelyezkedése	19
3.3. Kamera modell	20
3.4. CameraParameters.txt	20
3.5. CameraParametersMinimal.txt	21
3.6. A LiDAR szenzor és a kamera közötti transzformáció	21
3.7. A pontok indexelése	23
3.8. Osztály diagram	24
3.9. Teszt gömb	34
3.10. Teszt Delaunay kocka - oldal nézet	35
3.11. Teszt Delaunay kocka - sarok nézet	35

Algoritmusjegyzék

1.	Main	25
2.	Octree Insert Function	31
3.	BowyerWatson algorithm	33
4.	Point order dependent mesh generation	34

Forráskódjegyzék

3.1.	Az init függvény	25
3.2.	Az update függvény	26
3.3.	A render függvény	27
3.4.	A load_inputs_from_folder függvény	28
3.5.	Az init_point_visualization függvény	29
3.6.	A render_points függvény	30