



Pázmány Péter Katolikus Egyetem

Információs Technológiai és Bionikai Kar

## Szakdolgozat

# Régészeti térinformatikai alkalmazások fejlesztése

Bacsu Attila

Mérnökinformatikus BSc

2017

Témavezető:

Dr. Benedek Csaba

Témabejelentő 1. oldal

Témamejelentő második oldal

# Hallgatói nyilatkozat

Alulírott **Bacsu Attila**, a Pázmány Péter Katolikus Egyetem Információs Technológiai és Bionikai Karának hallgatója kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, és a szakdolgozatban csak a megadott forrásokat használtam fel. minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettettem, egyértelműen a forrás megadásával megjelöltem. Ezt a Szakdolgozatot más szakon még nem nyújtottam be.

.....  
aláírás

## Tartalomjegyzék

1.)	Bevezetés .....	9
2.)	Előzmények .....	10
2.1.)	Háromdimenziós adatgyűjtő eszközök .....	10
2.2.)	Főbb tulajdonságok: .....	10
2.2.1.)	FARO Focus3D X 330 .....	12
2.2.2.)	Lucida 3D Scanner.....	13
2.2.3.)	Kreon Zephyr 50 3D Scanner.....	14
2.2.4.)	Nub 3D SIDIO Scanner.....	15
2.2.5.)	Breuckmann Smart Scan 3D .....	16
2.3.)	Általános megállapítások, irodalomkutatás .....	17
2.3.1.)	Lehetséges megoldások .....	19
3.)	Eszközök.....	22
3.1.)	OpenGL Extension Wrangler Library (GLEW) .....	22
3.2.)	GLFW.....	22
3.3.)	OpenGL Mathematics (GLM) .....	22
3.4.)	Point Cloud Library (PCL).....	22
3.5.)	LASTools.....	23
4.)	Fejlesztés .....	24
4.1.)	Fejlesztési előzmények .....	24
4.2.)	Osztályokat implementáló állományok (diagram).....	25
4.3.)	Implementáció kifejtése, fontosabb kód részek elemzése .....	26
4.3.1.)	Navigáció a pontfelhőben.....	26
4.3.2.)	Android kitekintés .....	33
4.3.3.)	Objektum detektálás .....	34
4.3.4.)	MATLAB használata.....	36
4.3.5.)	Progresszív Morfológiai Szűrő .....	37
4.3.5.1.)	Matematikai háttér.....	37

4.3.6.) Megvalósított talajszeparálás .....	45
4.3.7.) Annotálást segítő eszközök implementációja.....	45
5.) Összefoglalás, értékelés.....	47
6.) Továbblépési lehetőségek, értékelés.....	48
7.) Köszönetnyilvánítás .....	49
8.) Irodalomjegyzék .....	50
Függelék .....	52

# Tartalmi összefoglaló

A szakdolgozat bevezető részében egy általános áttekintést próbálok nyújtani a háromdimenziós képalkotás technológiáról és annak felhasználhatóságáról különböző területeken. Ezen belül bővebben kitérek a régészeti alkalmazásra, az eszközök tekintetében pedig a lézerszkennerekre fektetem a hangsúlyt.

Az előzményekben bemutatom a háromdimenziós adatgyűjtő eszközök általánosságban, majd kifejtem a főbb tulajdonságaikat és bemutatom 5 különböző lézerszkennert is. A fellelhető szakirodalom áttekintésével bemutatom az eredmények egy szegmensét, ahol különböző megoldásokat említek meg.

Felsorolom a (C++-ban történő) fejlesztés során felhasznált eszközöket, majd kifejtem a fejlesztés előzményeit a korábbi évek munkája alapján. Az implementációt és az egyes megoldások kialakulását részletezem és kitérek olyan területekre is, mint a MATLAB használata a fejlesztésben, valamint bemutatom a PCL könyvtár által megvalósított progresszív morfológiai szűrőt. Ismertetem az implementált talajszeparációs algoritmust, valamint az annotálást segítő eszközök megvalósításáról is szó esik.

Végül a munkát összefoglalom, értékelem és továbblépési lehetőségeket fogalmazok meg.

# **Abstract**

The introduction part of the thesis provides a general overview of current 3D imaging technologies and the application of them in different fields. The thesis touches upon the archaeological applications more briefly, and regard of devices, focuses mainly on laser scanners.

In the history part of the thesis 3D data collection tools/devices are presented in general, then five of these devices are shown in more detail with all of their features. Afterwards different solutions are presented which have been found as part of the bibliography research.

A list of tools used during the development, and the process of the development follows. The implementation and the individual solutions are shown in detail, then the usage of MATLAB, and I demonstrate the progressive morphological filter implemented by PCL library. I present the implemented ground separation algorithm and I describe the methods of all annotation tools.

Finally, I summarize my work, make my own assessment, and I try to mention some possibilities to move forward.

# 1.) Bevezetés

A kulturális örökség védelmében egyre nagyobb szerepet kapnak a digitális eszközök alkalmazásai. A lézeres és fotogrammetriai módszerek alkalmazása ma már alapvető a régészeti lelőhelyek felderítésében, a feltáráskor és az ott előkerült leletek, vagy akár egy műemlék épület és annak kutatása során. Viszont a legtöbb esetben ezek még csak egyfajta látványos 3D nyers adatként jelennek meg. Az egyik legnagyobb kihívást az jelenti, hogy a felmérések során létrehozott pontfelhőből a lehető legtöbb adatot tudunk elemezhetővé, vizsgálhatóvá tenni. Mind a régészeti, mind a műemléki kutatás szempontjából kiemelten fontos feladat az, hogy a felmérésekből származó adatokat automatizált módon lehessen értelmezni. Első lépésként meg kell ismerkednünk a lézerszkenneléssel nyert pontfelhőkkel, azok kezelésével, megjelenítésével és átalakításával.

A kezdeti cél pontosan ez volt. Meg kellett ismerni a különbségeket abban a környezetben, melyet hagyományos poligonhálós modellek helyett pontfelhőkkel írunk le és jelenítünk meg. Ezeket megfelelő struktúrába kellett rendezni, majd különböző animációkat megvalósítani: ütközésvizsgálat, ütközésválasz, különböző árnyalás modellek beépítése, interaktív fizikai animációk. Ezen felül az objektumdetektálás is fontos része volt a feladatnak. Ezeknek régészeti relevanciája is jelentős, hiszen a cél mindenképpen a régészeti munka egyszerűsítése és gyorsítása lenne.

A szakirodalomból kiolvasható, hogy mennyire is fontos az új 3D-s képalkotási technológiák alkalmazása a régészek számára, hiszen amellett, hogy jelentősen lerövidül egy-egy munkafolyamat, az elemzésre felvett adatok bármikor és bárhol elérhetők. Így, ha nincsenek is kint a terepen, ugyanazt a munkát távolról fizikai kontaktus nélkül elvégezhető. Ez a kényelmes mivolta mellett még biztonságosabb is, hiszen a tárgyaknak így nem eshet baja, ráadásul ezen feladatok elvégzésekor a régészek nincsenek napszakhoz kötve, és az időjárás viszontagságainak sincsenek kitéve.

A szakdolgozatban a lézerszkenneléssel nyert pontfelhőkre fektetem a hangsúlyt, ugyanakkor fontos tudni, hogy ez csak egy a számos 3D-s képalkotási technológia közül, melyek használata elterjedt a szakmában. Ilyen például a geodézia, vagyis a földméréstan, amely bár a terepi munka és a feldolgozás hossza miatt nagyon időigényes, viszont még mindig kiegészíti a többi módszert. Ennek oka pedig nem más, mint a nagy pontossága más eljárásokhoz képest. Ugyanakkor – szintén a többi módszerhez viszonyítva – a feldolgozható pontok száma kicsi. A különböző fotogrammetriai módszerek esetén – vagyis amikor fényképek

felhasználásával alkotunk 3D-s képet a tárgyakról – a terepen a mérés gyorsasága kiváló, viszont a feldolgozás már viszonylag lassú folyamat, arról nem is beszélve, hogy általában rosszabb a pontosság, mint a geodéziai, vagy a lézerszkenneres felmérések végzésénél. (Bár ez nagyban függ a fényképezőgéptől és feldolgozástól is.) Előnye, hogy jóval több pont 3D-s helyzetének meghatározására alkalmas, közvetett módon. Ezzel szemben a lézerszkenneléssel a gyorsaság mellett megmarad –, vagy még nő is – a felmérhető pontok száma, a pontosság pedig a felbontástól függ. A működése azon alapszik, hogy egy irányított fény sugár segítségével meg tudjuk mérni egy pont és a műszer távolságát. A fény sugár paramétereit a műszer rögzíti, és annak visszaverődése alapján alkot képet az adott pont térbeli helyzetéről.

## 2.) Előzmények

### 2.1.) Hárromdimenziós adatgyűjtő eszközök

Számos különböző lézerszkennelési technológia létezik. A lényeg, hogy ezek minden megfelelően legyenek kiválasztva az adott feladathoz, és ez egy nagy kihívás a régészeknek is.

A gyakorlatban a lézerszkennert az alapján választják ki, hogy mit akarnak felvenni vele. Ez alapján két fő csoportra tudjuk osztani a szkennereket. Ezek a *földi lézerszkennerek* (*Long-Medium Range 3D Scanners*) és a *strukturált fényű szkennerek* (*Close Range 3D Scanners*). A régészeti minden kettőt, egymással kiegészítve használják.

### 2.2.) Főbb tulajdonságok:

A szkennér által használt technikák szerint megkülönböztetünk lézer impulzust használó eszközöket, melyek a visszaverődő fény sugár elindulásának és visszaérkezésének időkülönbsége alapján határozzák meg a pontok háromdimenziós helyét. Emellett vannak olyan eszközök is melyek valamelyen háromszögelési technikát alkalmaznak, továbbá meg kell említenünk a strukturált fényű eszközöket is.

A mérési távolság tekintetében elég széles a skála. Vannak olyan eszközök, melyek akár 300 méternél távolabb is képesek mérni. A közelű tárgyak felvételéhez viszont jobban illeszkednek azok a szkennerek, melyek viszonylag kicsi területet fednek le, vagyis a mérési távolság körülbelül a [0,100] centiméteres tartományba esik. Ezek közül a legkisebbek mérési

távolsága körülbelül 50 milliméter.

Fontos tulajdonság a felbontás, ami minden egyes szkenner esetében különböző lehet, valamint egy lézerszkenner által felvett adatok között is lehet különbség a felbontásban a pontok távolságától függően. Általában pont / cm<sup>2</sup> -ben mérlik.

A mérési idő is változó lehet különböző eszközök esetében. Ez a szám megmutatja, hogy egységes nagyságú területet mennyi idő alatt tud felvenni a szkenner.

A feldolgozási idő ezzel szemben a feldolgozás idejére ad egy becslést, ugyanis ez az idő pontosan nem megállapítható. Erősen függ a számítógép sebességtől, valamint a feldolgozást végző személy tapasztalataitól.

Megkülönböztetjük az eszközöket az szerint, hogy milyen fájlformátumban képesek visszaadni kimeneti adatként. A videó fájlokotól a szokványos pontfelhő formátumokon át a képfájlokig minden megtalálható.

Az egyes szkennerekhez járó felszerelés is változó lehet. A szkenner mellé kaphatunk három lábú állványt (*tripod*), különböző kalibrációs eszközöket, mozgató kart, laptopot, vagy akár egy erősebb számítógépet is, valamint ide sorolhatjuk a feldolgozást végző szoftvereket is, amiket a gyártók mellékelni szoktak az eszközeik mellé.

Szintén fontos szempont, hogy az eszközt milyen területen tudjuk alkalmazni. A régészeti felhasználás mellett alkalmazható az építőimárban, az önjáró eszközök vezérlésének fejlesztésére, homlokzatok, szobrok felvételére, falak freskók, kéziratok, érmék szkennelésére, stb.

Ezeket megerősítendő szeretném bemutatni 5 konkrét lézerszkennert:

## 2.2.1.) FARO Focus3D X 330 [8]

<i>Technika</i>	Lézer impulzus / időkülönbség alapú 3D szkenner
<i>Mérési távolság</i>	Földi lézerszkenner (0.6 m – 330 m)
<i>Felbontás</i>	A részletességi szinttől és a tárgy közelségétől függ. 1 millió – 750 millió pont letapogatásonként.
<i>Mérési idő</i>	Felbontás- és minőségfüggő. 30 másodperc – 2 óra
<i>Feldolgozási idő</i>	Adat- és kívánt kimenetfüggő, de általában nagyon hosszú.
<i>Fájl formátumok</i>	Az alapértelmezett formátum .FLS; pontfelhő formátumban
<i>Felszerelés</i>	LiDAR szkenner, háromlábú állvány, kamera és panorámakép készítő, referenciátárgyak
<i>Működés</i>	1 személy, Képességek: mérési és térinformatikai háttér/tapasztalat szükséges
<i>Feldolgozás</i>	Képességeket és érzéket is igényel. Képességek: Térinformatikai tapasztalat, valamint digitális képi megjelenítésben való jártasság
<i>Előnyök</i>	<ul style="list-style-type: none"> <li>• Kicsi, könnyű felszerelés</li> <li>• Nagy hatótávolságú mérés</li> <li>• 70 megapixeles kamera a színek felvételéhez és a fényképek készítéséhez a minél több adat kinyerése érdekében</li> <li>• Egyszerű, intuitív interfész</li> <li>• Nem szükséges számítógép a méréshez</li> <li>• Beépített cserélhető akku</li> </ul>
<i>Hátrányok</i>	<ul style="list-style-type: none"> <li>• A felbontás a távolsággal arányosan változik</li> <li>• A feldolgozáshoz szakértői tudásra van szükség</li> <li>• Áttetsző, tükröződő és sima felületekről nehéz felvételt készíteni</li> <li>• Nem lesz nagy felbontású adatunk az objektum textúrájáról és felületéről</li> </ul>
<i>Mihez kezdhetünk az adattal?</i>	3D bejárása a környezeteknek és az épületeknek, 3D nyomtatás, stb

## 2.2.2.) Lucida 3D Scanner [8]

<i>Technika</i>	Háromszögelés alapú 3D lézerszkenner
<i>Mérési távolság</i>	Közeli (8-10 cm)
<i>Felbontás</i>	100 mikron (10 000 pont / cm <sup>2</sup> )
<i>Mérési idő</i>	4 óra / m <sup>2</sup>
<i>Feldolgozási idő</i>	4-6 óra / m <sup>2</sup> (függ a számítógép gyorsaságától és az üzemeltető tapasztalatától)
<i>Fájl formátumok</i>	AVI (videó fájl), RIS, 32 bites TIFF, 8 bites TIFF
<i>Felszerelés</i>	Könnyű CNC keret és vezérlő, szkennelő fej, laptop
<i>Működés</i>	1 személy, Képességek: alap
<i>Feldolgozás</i>	1 személy, Képességek: alap
<i>Alkalmazások</i>	Falak, freskók, festmények, térképek, kéziratok, érmék, stb.
<i>Előnyök</i>	<ul style="list-style-type: none"> <li>• Más szkennereknek problémás lehet sokféle felület felvétele (csillagó, nagy kontraszt, stb.)</li> <li>• Tömörítetlen videóba menti az adatot</li> <li>• Kombinálni lehet egyéb adatokkal a kimenetet (UV, infravörös, szín, stb.)</li> <li>• Hordozható, könnyű összeszerelés és működtetés, olcsó</li> <li>• Hálózatról és akkáról is működtethető</li> </ul>
<i>Hátrányok</i>	<ul style="list-style-type: none"> <li>• Érzékeny a vibrációkra</li> <li>• Érzékeny az erős, irányított fényre</li> <li>• Lassú szkennelés</li> <li>• Átlátszó és áttetsző felületeket nem tud felvenni</li> <li>• Limitált mélységélesség (2,5 cm)</li> </ul>
<i>Mihez kezdhetünk az adattal?</i>	Képalkotás a különböző adatrétegekkel együtt (pl. szín)

### 2.2.3.) Kreon Zephyr 50 3D Scanner [8]

<i>Technika</i>	Háromszögelés alapú 3D lézerszkenner
<i>Mérési távolság</i>	Közeli, a szenzor mélységélessége 50 mm, a kar hatósugara kb. 2 m
<i>Felbontás</i>	Függ a szkennelés sebességtől, maximum 50 mikron (0,05 mm)
<i>Mérési idő</i>	30 000 pont / másodperc
<i>Feldolgozási idő</i>	30 000 pont / másodperc
<i>Fájl formátumok</i>	Bármilyen ismert pontfelhő-, vagy mesh-formátumban való exportálás támogatott (Polygonia szoftver)
<i>Felszerelés</i>	Zephyr 50 szkenner, háromlábú állvány, kar, laptop
<i>Működés</i>	1 személy, Képességek: közepes
<i>Feldolgozás</i>	1 személy, Képességek: közepes
<i>Maximum méret</i>	A kar hatótávolsága, nagyjából 2 m <sup>2</sup>
<i>Alkalmazások</i>	Önjáró eszközök vezérlése, repülés- és építőipar, visszafejtés, szobrok felvétele, stb.
<i>Előnyök</i>	<ul style="list-style-type: none"> <li>• Hordozható, könnyű, egyszerű működtetés</li> <li>• Nehezen elérhető helyek szkennelése</li> <li>• Szkennelés különböző körülmények között</li> </ul>
<i>Hátrányok</i>	<ul style="list-style-type: none"> <li>• Lassú szkennelés, újra kell szkennelni az objektumot a nagyfelbontású adathoz</li> <li>• Korlátozott szkennelési terület (kar mérete)</li> <li>• Csak hálózatról működtethető</li> <li>• Nagyon érzékeny a vibrációra, a háromlábú állványnak nagyon stabil alap kell</li> </ul>
<i>Mihez kezdhetünk az adattal?</i>	Pl. 3D nyomtatás

## 2.2.4.) Nub 3D SIDIO Scanner [8]

<i>Technika</i>	Strukturált fényű szkenner
<i>Mérési távolság</i>	Közeli, 30 cm, 60 cm, 1 m (különböző területenként)
<i>Felbontás</i>	75, 130, vagy 250 mikron
<i>Mérési idő</i>	Számos tényezőtől függ, mint például a záridő, a tárgy karakterisztikája, de minden egybevetve elég gyors (pár másodperc letapogatásonként)
<i>Feldolgozási idő</i>	A szkennelt objektum komplexitásától függ.
<i>Fájl formátumok</i>	Pontfelhő formátumok. A natív formátum TRI, de ezt PIF formátumban kell exportálni ahhoz, hogy olyan feldolgozó szoftverekkel meg lehessen nyitni, mint például a Polyworks
<i>Felszerelés</i>	3 db három lábú állvány, szkenner, 2 külső projektor, egy erős asztali számítógép, valamint a kalibráláshoz szükséges eszközök
<i>Működés</i>	Lehetséges egyedül is üzemeltetni, de javasolt 2 fő. Szint: közepes
<i>Feldolgozás</i>	Szint: szakértő
<i>Maximum méret</i>	Szkennelési szakaszonként 50 cm <sup>2</sup>
<i>Alkalmazások</i>	Bármire, amihez nagy részletességű és nagy pontosságú szkennelés szükséges. Pl.: homlokzatok, szobrok, stb
<i>Előnyök</i>	<ul style="list-style-type: none"> <li>• Automatikus elő-igazítás a vonatkoztatási pontok segítségével</li> <li>• Pontos és magas minőségű mérés</li> <li>• A kimenő adat egy tiszta és rendezett pontfelhő</li> <li>• Nehezen sérül</li> <li>• Kis vállalat, ahol a rendszert tervező és építő emberek adják el az eszközt</li> <li>• Ellenőrzés az utómunkálatok és a mesh generálás felett</li> </ul>
<i>Hátrányok</i>	<ul style="list-style-type: none"> <li>• Nehéz, sok felszerelés, sok területre van szüksége a működéshez. (A későbbi verziókban ezt valamelyest javították.)</li> <li>• Nem szkennelhető vele semmilyen tárgy, ami túl sötét, áttetsző, visszaverődő, vagy csillogó, fényes felület.</li> <li>• A halogén fény jelentős hőt termel</li> <li>• Érzékeny a vibrációra</li> </ul>

	<ul style="list-style-type: none"> <li>Folyamatos áramellátást igényel a hálózatról</li> <li>Drága rendező szoftver</li> <li>Komplex kalibrációs rendszer</li> <li>Csak egy kamerával rendelkezik</li> <li>Drága</li> </ul>
<i>Mihez kezdhetünk az adattal?</i>	3D nyomtatás, visszafejtés, különböző képernyő alapú alkalmazások, a nagy felbontású adat lehetővé teszi az alapos vizsgálatot.

### 2.2.5.) Breuckmann Smart Scan 3D [8]

<i>Technika</i>	Strukturált fényű szkenner
<i>Mérési távolság</i>	Körülbelül 1 m
<i>Felbontás</i>	Két beállítása létezik a szkennernek. Az M-410, ami körülbelül 140 mikron felbontást tesz lehetővé egy 30x30 cm területen, valamint az M-810, ami pedig 250 mikron felbontást nyújt egy 60x60 cm-es területre.
<i>Mérési idő</i>	Függ a felbontástól, de összességében gyorsabb, mint a SIDIO
<i>Feldolgozási idő</i>	Gyorsabb, mint a SIDIO szkenner esetében, a rendszer 3D mesh egyszerű generálására lett tervezve.
<i>Fájl formátumok</i>	3D mesh formátumok, pl OBJ és STL
<i>Felszerelés</i>	Szkenner, három lábú állvány, laptop erős videókártyával és legalább 32 GB memóriával, mivel az OPTOCAD szoftver a sok memóriát igényel a 3D mesh feldolgozáshoz; különböző kalibrációs eszközök
<i>Működés</i>	1 személy is lehetséges, de 2 javasolt. Szint: közepes
<i>Feldolgozás</i>	1 személy. Szint: közepes
<i>Maximum méret</i>	Szkennelési szakaszonként 1 m <sup>2</sup>
<i>Alkalmazások</i>	Homlokzatok, szobrok, tárgyak, stb.
<i>Előnyök</i>	<ul style="list-style-type: none"> <li>Könnyebb, gyorsabb, valamint kevesebb felszerelés (a SIDIO-hoz képest)</li> </ul>

	<ul style="list-style-type: none"> <li>• Színek felvétele (lehetséges felbontások: 0,8 – 2,0 – 8.0 megapixel)</li> <li>• Vibráció esetén szünetelteti a szkennelést, majd újrakezdi</li> <li>• Nincs szükség fix tárgyakra a szkennelések összefésüléséhez</li> <li>• Két kamerával rendelkezik, melyed adatait felhasználja a szkenneléskor</li> <li>• Képes kombinálni a fotogrammetriai és a strukturált fényű mérés eredményét, ezzel rendkívül alapos vizsgálatot lehetővé téve</li> <li>• LED égőt használ, ami nem termel hőt</li> <li>• A szoftver azonnal rendezi az adatot, arról pedig egy előképet mutat, ezzel egyszerűvé téve az adattal való munkát</li> <li>• A kimenet egyszerű, könnyen feldolgozható</li> </ul>
Hátrányok	<ul style="list-style-type: none"> <li>• Nem lehet hozzáférni a pontfelhőhöz, mivel a rendszer 3D mesh megjelenítésre lett tervezve az OPTOCAD szoftveren keresztül (a poligonhálós modellből a pontokat ki lehet emelni, de ez nem olyan pontos, mint az eredeti pontfelhő)</li> <li>• Nagy a kalibrációs felszerelés</li> <li>• Csak hálózatról működtethető</li> <li>• Drága</li> </ul>
Mihez kezdhetünk az adattal?	Képernyő alapú alkalmazások, gyors prototípus gyártás, stb.

### 2.3.) Általános megállapítások, irodalomkutatás

A szkennerek ma már több millió pontot tudnak felvenni másodpercenként. A pontok sűrűsége teljesen feladatfüggő dolog, és gyakori az is, hogy egy nagy ásatási terület vagy épület felmérésekor nem elég egyetlen helyről elvégezni a mérést, így a szkennert több pozíció beállítva, majd ezután az adatok egyesítésével tudjuk kinyerni a teljes pontfelhőt. Az egyesítés módja eszközönként változhat. Vannak olyan gyártók, melyek külön programot biztosítanak ehhez a felhasználók számára. Az így létrejött pontfelhővel lehet dolgozni. Ám ahhoz, hogy ez hatékony legyen, különböző módszerekkel el kell távolítani a pontfelhőről felesleges objektumokat, mint a növényzet (bokrok, fák), talaj (bizonos esetekben) és egyéb nem releváns

objektumok. Egy átlagos felmérésnél, ezek után még mindig túl nagy a pontfelhő, ami miatt átlagos teljesítményű PC-n és laptopon gyakorlatilag kezelhetetlen lesz, hiszen ehhez rengeteg erőforrásra van szükség. Emiatt szükség van még ezek ritkítására. Így nyilvánvalóan csökkenni fog a részletesség, viszont kezelhetővé válik a pontfelhő. A ritkítás mellett a másik lehetséges megoldás a részletek kivágása a pontfelhőből, így ezeket a részleteket külön-külön lehet elemezni. Persze célszerű megtartani az eredeti állományt is, hisz ebből tudjuk majd kivágni, illetve ritkítani a feldolgozásra szánt részletet. Annak tudatában, hogy milyen feladatot akarunk elvégezni az adatokon, meghatározható a szükséges felbontás és a méret. Ráadásul azért is érdemes az eredeti pontfelhőt megtartani, mert akár kézi, akár fél-automatikus, esetleg automatikus módon tisztítjuk meg a pontfelhőt a nem releváns objektumoktól, ebbe a folyamatba minden bekerülhet egy hiba. Ezt pedig csak az eredeti adatokból tudjuk helyreállítani. [18]

Szakmai berkeken belül is vitára ad okot a pontfelhők *polygonmodell*é alakítása. Amit mindenkiéppen kijelenthetünk az az, hogy az átalakítás számításigényessége és kézi munkát is követelő mivolta mellett szükségszerűen csökkenti a részletgazdagságot egy sűrű bemeneti pontfelhőt feltételezve. Ezért is nagyon fontos a pontfelhő alapú, illetve a sűrű *voxel-modell* alapú téreprezentáció, ami feldolgozási oldalról számos kihívást tartogat. Ugyanakkor teljesen megalapozottnak tűnik a pontfelhőből a hagyományos 2D-s képek előállítása is. Ezen a területen is különösen fontos, hogy mi az, amit éppen mérünk, és eszközönként is nagyon sok különbség lehet, hiszen például egy tárgysz kennelést nem kell olyan mértékben megtervezni, mint óriási (ásatási) területek felvételét. A nagy adatmennyiségek szinte lehetetlenné - vagy legalábbis durván erőforrásigényessé - is tenné a poligonmodellé alakítást a nagy pontfelhőknél, ugyanakkor nem feledkezhetünk meg arról sem, hogy régészeti szempontból néha kis objektumok, tárgyak felvétele is releváns lehet. Léteznek olyan lézerszkennerek, melyek pont erre vannak gyártva és már önmagukban, működés közben képesek a poligonháló előállítására.

A legtöbb lézerszkennerbe egy fényképezőgép is bele van építve, ami alapja lehet a színes pontfelhő elkészítésének. A gyakorlatban, ha nincs fényképezőgép az eszközben, vagy azzal nem lehet megfelelő felbontású és minőségű képet készíteni, gyakran használnak külön kamerát. Ezt persze pontosan kell illeszteni a szkenner helyére ahhoz, hogy a kinyert adatok felhasználhatóak legyenek a pontfelhő elkészítésekor. [18]

Fontos, hogy eszköztől függetlenül is kerülhetnek a felmérésbe hibák, hiszen nem tudunk 100%-osan olyan mérési pontokat kijelölni, ahonnan minden jól látható. Ahhoz, hogy ezeket kiküszöböljük, a mérést nagyon precízen kell megtervezni, és a megvalósításban is

nagyon pontosnak kell lenni. Akarva-akaratlanul ilyen hibák persze előfordulhatnak, ezeket a feldolgozás során kell orvosolni. Emellett ki kell szűrni azokat a zajokat is, melyeket a több mérési pontból – különböző felbontással – felvett redundáns pontok adnak. Ez főleg a színeknél okozhat gondot, hiszen könnyen lehet, hogy különböző mérési pontkból a fény másképp török meg adott felületen, vagy egész egyszeren, az egyik felvétel napos időben, a másik pedig borús időben készült. Ezért is nagyon fontos a mérés megtervezése, és a fennmaradó hibák szoftveres orvoslása.

### 2.3.1.) Lehetséges megoldások

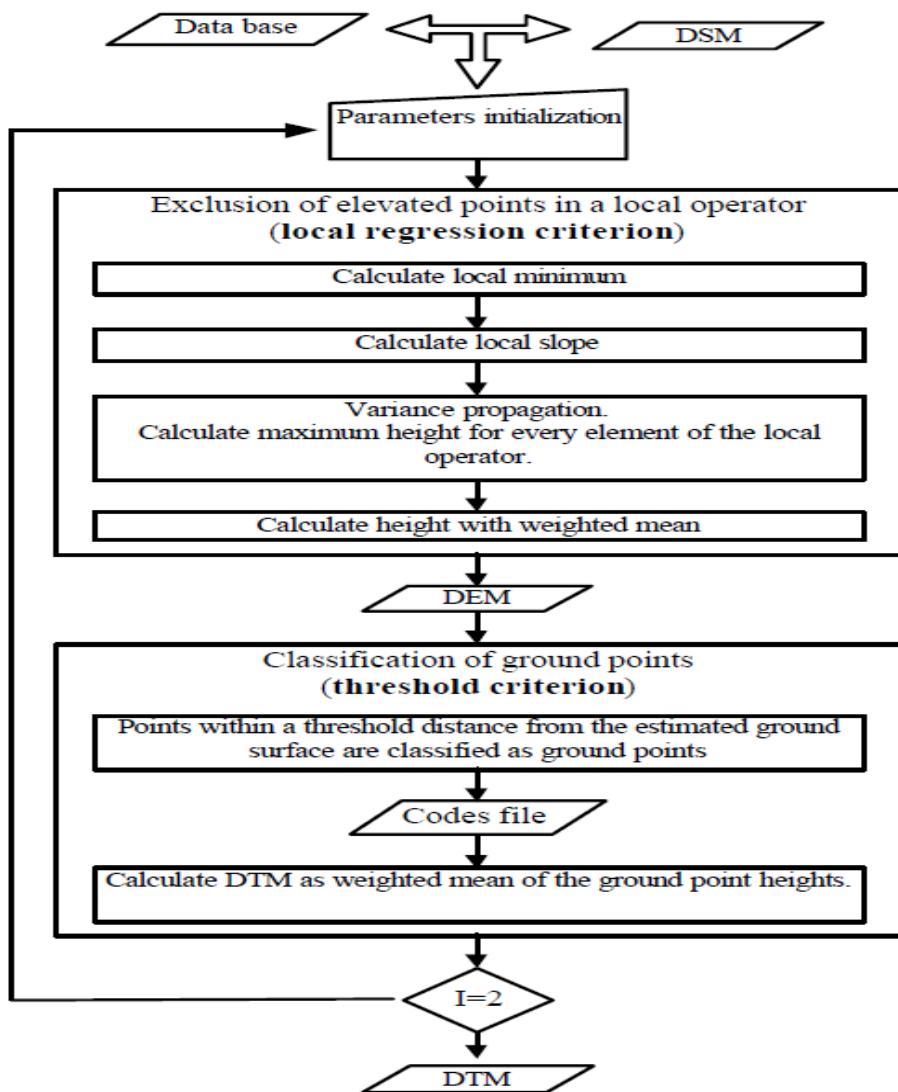
A nagy felbontású digitális terepmodellek tehát a régészet számára is nagyon fontosak az analízishez és a vizualizációhoz is. Ahogy azt bemutattam, azt ezt előállító rendszerek kimenetén egy pontfelhőt kapunk a horizontális ( $x,y$ ) koordinátaikkal és a magasságukkal ( $y$ ), térbeli pontok rendezetlen halmazát kapjuk meg. Szintén volt szó arról a korábbiakban, hogy ezek a szkennelések tartalmazhatnak épületeket, járműveket, növényzetet és a talajt. Ahhoz, hogy digitális terepmodellt hozhassunk létre, ezeket a részeket azonosítani és osztályozni kell. Sok kihívást tartogat tehát azon pontok eltávolítás a pontfelhőből, melyek nem tartoznak a talajhoz. *K. Kraus* és *N. Pfeifer* több mint húsz éve publikált eljárása a *Gauss* által kidolgozott, úgynevezett *legkisebb négyzetek módszerén* alapult. Ennek segítségével erdős területeken lehetett a fák eltávolításával digitális terepmodelleket készíteni.[16] Ezt később *Pfeifer* kiterjesztette az épületek szűrésével is, ami városi felmérések során hasznos, ám sajnos olyan tulajdonságoknak kell teljesülni ezeknek a működéséhez, ami a LIDAR méréseknél legtöbbször nem adott, így nem garantált a jó eredmény. [12]

*Vosselman* ezzel szemben egy olyan szűrőt mutatott be, amely az egyes 3D-s pontok között meghatározza a dőlést/lejtést. [20] Egy adott pontot a talaj pontjaként klasszifikál akkor, ha egy körön belül bármely két pont közötti dőlés maximum értéke egy előre meghatározott küszöbérték alatt van. Minél alacsonyabb ez a küszöbérték, annál több objektum kerül eltávolításra. A küszöbérték lehet konstans érték, de lehet a távolságnak is egy függvénye. [12] Ahhoz, hogy ezt a paramétert jól be tudjuk állítani, előzetes ismeretek szükségesek az adott pontfelhőről.

A LIDAR mérések klasszifikálása esetén két fő hibát különböztünk meg, akármilyen szűrőt is használunk. Az egyik az, amikor az algoritmus talajpontként klasszifikál egy pontot, ami nem tartozik a talajhoz, a másik pedig ennek pontosan az ellenkezője, amikor a talajhoz

tartozó pontokat is kiszűri az eljárás. [17] A dőlésen alapuló klasszifikáció lényege, hogy meg tudunk határozni egy olyan – közel optimális – küszöbértéket, amivel minimalizálni tudjuk az említett két hibát. Ennek a meghatározása a pontfelhő környezetének ismeretétől függően meglehetősen szubjektív. *Vosselman* azt próbálta demonstrálni, hogy jó eredményeket lehet kapni, ha különböző tanulóhalmazok alapján határozzuk meg a küszöbértéket. [20]

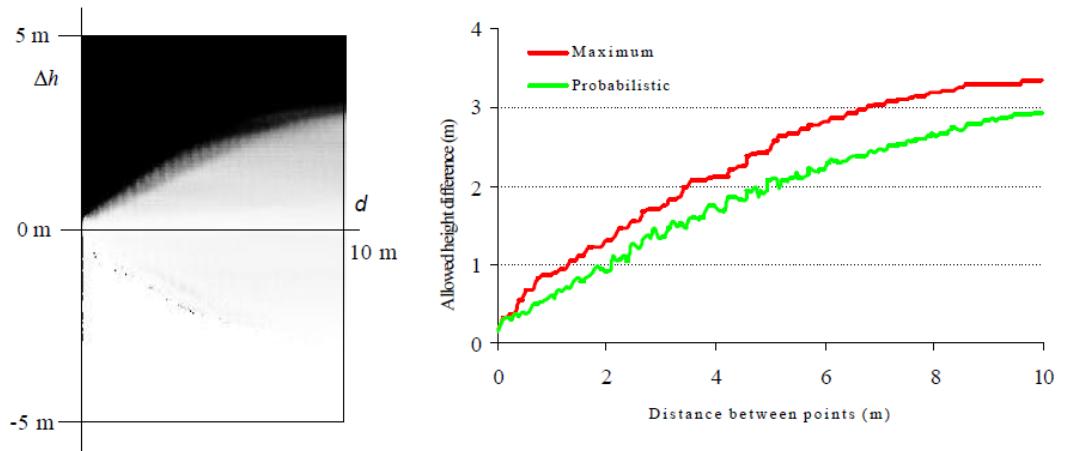
Az 1. ábrán láthatjuk az általános folyamatát annak, hogy miként generálunk *digitális terepmodellt(DTM)* és *digitális domborzatmodellt(DEM)* a *digitális felületmodellből(DSM)* olyan módon, hogy felhasználjuk a pontok közötti lejtést (*slope*) is. [13]



1. ábra [13]

Az alap feltevésünk ennél a szűrőnél az, hogy jelentős különbség van a terep pontok és a nem terep pontok (épületek, fák) dőlése, lejtése között. Ez városi környezetben általában igaz is, és jó eredményt adhat. Ugyanakkor, ha a felmérést például hegyi környezetben végezzük, sűrű növényzettel, ahol nagy az eltérés a talaj lejtésében, akkor egyáltalán nem garantált a jó eredmény, és mindenki említett hiba nagy arányban fog előfordulni. Éppen ezért ennek a módszernek a használata nem lenne célravezető egy olyan régészeti felmérésnél, ami például egy domboldalon levő vár romjairól készült. [12]

A 2. ábra Vosselman eredményeit mutatja egy 43000 pontból álló pontfelhőre, melyen 2 külön szűrőt tesztel (maximum, valószínűségi). Az ábra bal oldalán annak a valószínűségét láthatjuk, hogy  $p_i$  pont eleme a digitális felszínmodellnek (DEM) feltéve, hogy  $p_j$  is eleme, ha  $d$  a két pont távolsága és  $h$  a két pont magasságbeli differenciája ( minden egyes  $p_i, p_j$  pontpár esetén). A fekete szín jelzi a 0 valószínűségi értéket, a fehér pedig az 1-et. A jobb oldali részen láthatjuk pirossal a talajpontok közötti maximális magasság különbséget, zölddel pedig azon pontpárok közötti magasság különbséget, melyek esetében 50% annak a valószínűsége, hogy mindenki pont a digitális felszínmodell része. [20]



2. ábra [20]

Egy másik gyakran használt eljárás a nem talaj objektumok eltávolítására a *matematikai morfológián* alapul, és szürkeárnyalatos képeken használatos. [12] A növényzet, az autók, valamint az épületek magassága általában nagyobb, mint az öket körülvevő terepé. Ha a LIDAR felmérés eredményét egy szürkeárnyalatos képpé konvertáljuk (például egy pontfelhőmegjelenítőből felülnézetből kivágjuk a megfelelő 2D-s képet), akkor az épületek, az autók és a fák alakja azonosítható az alapján, hogy sötétebb vagy világosabb az árnyalatuk.

Köztudott, hogy a matematikai morfológia műveleteinek használatával különböző objektumokat tudunk azonosítani szürkeárnyalatos képeken, éppen ezért ez használható lézerszkennelt pontfelhők szűrésére is. [12]

## 3.) Eszközök

### 3.1.) OpenGL Extension Wrangler Library (GLEW)

Multiplatform függvénykönyvtár. Lehetőséget ad *shaderek*, *bufferek* és egyéb hasznos dolgok elkészítésére. Futásidőben hatékonyan tudja eldönteni, hogy mely *OpenGL* kiegészítések támogatottak az adott platformon.

### 3.2.) GLFW

Nyílt forráskódú, multiplatform függvénykönyvtár *OpenGL-hez*, ami egy nagyon egyszerű és letisztult API-t kínál olyan funkciókhoz, mint például *OpenGL* ablak létrehozása, egér- és billentyűzet input fogadása, kezelése, stb.

### 3.3.) OpenGL Mathematics (GLM)

Matematikai függvénykönyvtár azon grafikai szoftverekhez, melyek az *OpenGL Shading Language (GLSL)* specifikáción alapszanak. Azonos implementációkat és azonos névkonvenciókat tartalmaz, mint a *GLSL*, tehát aki tudja a *GLSL*-t használni, az könnyen el tud boldogulni a *GLM* függvénykönyvtárral *C++-ban*.

### 3.4.) Point Cloud Library (PCL)

Nyílt forráskódú nagy függvénykönyvtár 2D/3D kép és pontfelhő feldolgozáshoz.

### 3.5.) LASTools

A *lasground* és a *lasclassify* eszközök a *LASTools* programcsokorba tartoznak. Ezt a *rapidlasso GmbH* fejleszti, és a célja, hogy egyszerűen használható eszközöket adjon a felhasználó kezébe a *LAS* formátumú pontfelhő fájlok feldolgozásához. A *LASTools* honlapján minden egyes modulhoz található külön dokumentáció rövidebb (használatot bemutató oldal) és hosszabb (*readme.txt*) változatban is.

Ezen eszközök közül a *las2las* nevű modul volt segítségemre kezdetben, ugyanis ezzel daraboltam fel a pontfelhőt, illetve a feldarabolás mellett képes az állományok ritkítására is. A feldarabolást és a pontfelhőből való kivágást később a – Dr. Jankó Zsolt által fejlesztett – pontfelhő manipulátor programmal csináltam meg.

A *lasground* modul segítségünkre lehet, ha szeretnénk definiálni egy ilyen fájlban a talaj pontjait. Az eszköz bemeneti paramétereként beállítható, hogy a definiált talajmagassághoz képest mennyivel magasabban levő elemeket szeparáljon le, valamint külön definiálhatjuk a sík magassági értékét. Ha elvégeztük a talaj elkülönítését, akkor a *lasheight* modullal minden egyes pontnak ki tudjuk számítani a talajszint feletti magasságát.

A *lasclassify* segítségével lehetőséget kapunk arra, hogy az előző két modul (*lasground* és *lasheight*) kimenetének felhasználásával klasszifikálni tudjuk a pontokat három csoportba: talaj-pontok, növényzet-pontok, épület-pontok. A talaj pontokat már egy az egyben megkapja bemenetként a modul, a növényzetet és az épületeket pedig úgy különíti el egymástól, hogy megvizsgálja a talaj feletti egyes pontok által létrehozott felület simaságát.

Sajnos a nagyobb pontfelhőket nem kezelik jól ezek a modulok, így ilyen esetekben ezekre nem számíthatunk jó eredményre.

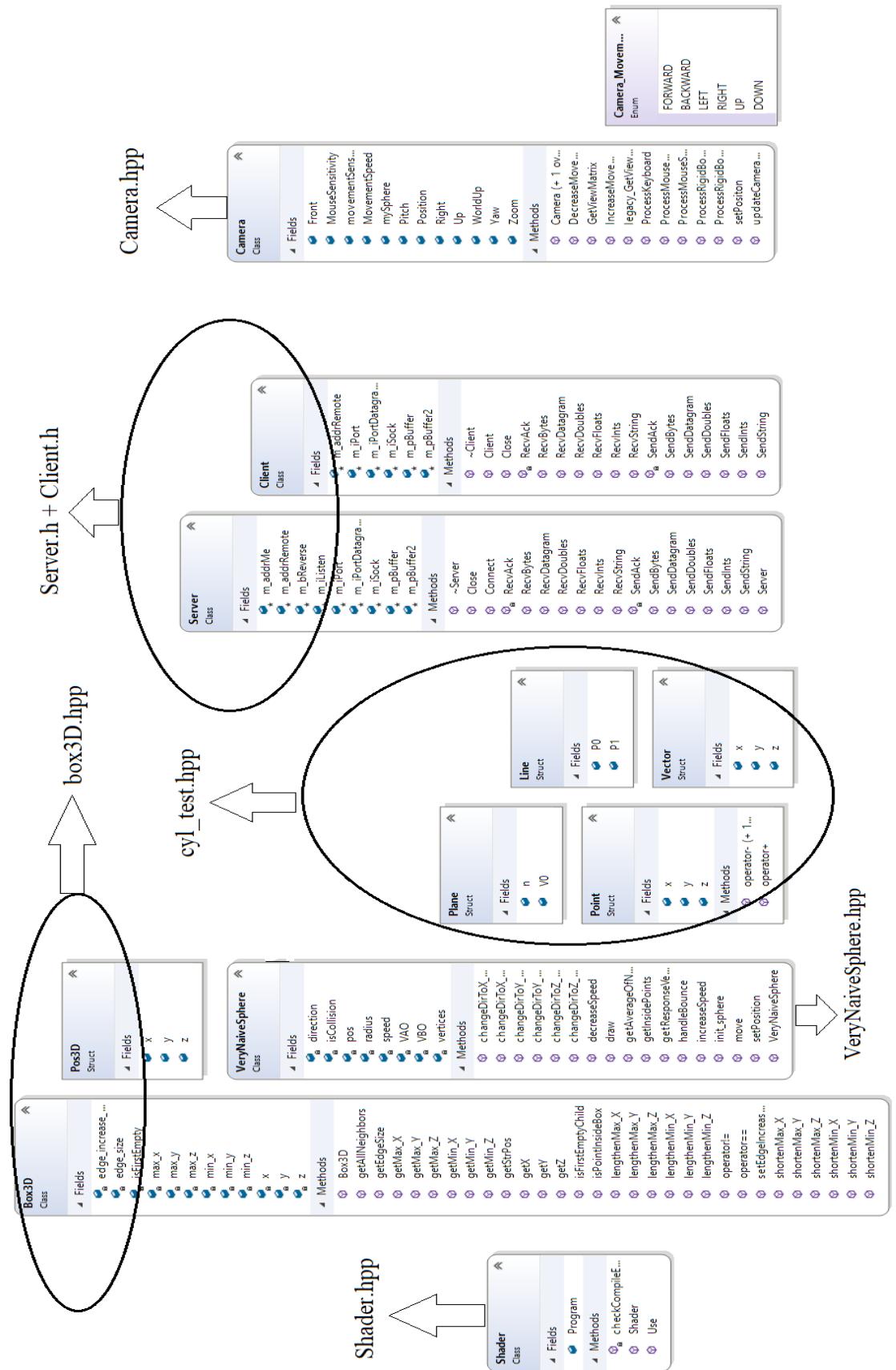
# 4.) Fejlesztés

## 4.1.) Fejlesztési előzmények

A *PCD* és *LAS* kiterjesztésű fájlokból a *Point Cloud Library(PCL)* segítségével olvastam ki az adatokat és ennek segítségével rendeztem *Octree-be* is, hogy később egyszerűbben tudjam kezelní.

Ugyanakkor a megjelenítést teljesen külön választottam az összes többi művelettől. A pontfelhő megjelenítését *OpenGL* segítségével valósítottam meg. Modern *OpenGL*-ben már nem érdemes a *C++* kódon keresztül közvetlenül elvégezni a megjelenítéshez szükséges számításokat, műveleteket, transzformációkat. Ehelyett ún. *Vertex Buffer Objecteket (VBO)* használunk. Ezek olyan módon teszik hatékonytá a megjelenítést, hogy az összes megjelenítendő pont – melyeket egy függvényen keresztül átadok a VBO-nak – a videókártya memoriájába kerül át, így a *GPU*-nak rendkívül gyors, közvetlen hozzáférése lesz a megjelenítendő adathoz. Innentől kezdve a megjelenítést a *shader* fájlokban keresztül lehet manipulálni. A transzformációs mátrixokat a *GLM* függvénykönyvtár segítségével lehet megcsinálni és ezeket a mátrixokat *C++* kódon keresztül hozzá lehet kötni a *shader* fájlokban definiált mátrixokkal.

#### **4.2.) Osztályokat implementáló állományok (diagram)**



## 4.3.) Implementáció kifejtése, fontosabb kódrészek elemzése

### 4.3.1.) Navigáció a pontfelhőben

Ennek megvalósításához egy úgynevezett „*free-fly*” kamerát implementáltam, melynek előre definiált mozgási irányai (előre, hátra) és fordulási irányai (fel, le, jobbra, balra) vannak. Ezekben kívül a kamerát megvalósító *header fájl*ban kerülnek inicializálásra a kezdeti szögek – melyek meghatározzák a vertikális, valamint a horizontális síkon történő elfordulást -, a kamera mozgási sebessége és az egérmozgásra való érzékenysége. Utóbbi két paraméter a futás során tetszőlegesen állítható dinamikusan, míg az előbbi két paraméter az egér érzékenység alapján változik az egér bizonyos mozgásairól.

Ez a fájl több függvényt is megvalósít:

A konstruktorban kezdőértéket adunk az összes paraméternek. Megadjuk a kamera kezdeti pozíóját, a horizontális és a vertikális síkon történő elfordulás szögét, illetve a koordinátarendszer felfelé mutató vektorának irányát.

A *GetViewMatrix()* függvényben a *glm::lookAt()* segítségével egy olyan transzformációt hajtunk végre, mely a modelltér koordináta-rendszerét új pozícióba transzformálja. Fontos, hogy ez a transzformáció utolsóként kerül végrehajtásra, vagyis a modellmátrix transzformációs sorának a végén helyezkedik el. A transzformációt a függvény három paraméter segítségével végzi el, ez a kamera jelenlegi pozíciója, egy – a kamera nézési irányában levő térbeli pont, valamint a kamera felső oldalának az irányába.

A *legacy\_GetViewMatrix()* függvény ugyanezt a funkciót valósítja meg, csak a modern *OpenGL* eszközei nélkül, a *gluLookAt()* metódust felhasználva. Ennek a függvénynek a használata ma már nem javasolt.

A *ProcessKeyboard()* függvény minden olyan információt fogad, ami a billentyűzetről jön be és ez alapján végez el több dolgot. A bemeneti paraméterek között van a kamera mozgási iránya (előre vagy hátra), illetve a teljes pontfelhő és az ebből alkotott *Octree* is. Ezekre azért

van szükség, hogy lehetőségünk legyen olyan módban is navigálni a pontfelhőben, hogy a kamera ne „mehessen át” a pontokon, vagyis animálni tudjuk a kamera és a pontfelhő fizikai interakcióját. Kezdetnek az algoritmus a kamera pozícióját egy gömb középpontjaként értelmezi, majd ezen gömb sugarán belüli pontokat keresi meg, hiszen ezekkel a pontokkal történik vélhetőleg a kamera ütközése. A kamerából egy vektorral rámutatunk ezekre a pontokra és átlagoljuk őket, vagyis összeadjuk az össze vektort, majd az eredmény vektor leosztjuk a saját hosszával, ezzel egy egységvektort kapunk. Előre definiált sebességvektorok mellett így megkapjuk az ütközés irányú sebességkomponenst is. Ez után rávetítjük ezt a sebességkomponenst a megfelelő irányú egységvektorra, amit a normalizálással kaptunk meg. Ha az ütközés irányú sebességkomponens és az ütközés irányú vektor bezárt szöge kisebb, mint 90 fok, akkor az eredetiből kivonom ezt a komponenst. Ellenkező esetben nem közeledtem a ponthoz, hanem távolodtam tőle, ezért nem kell kivonni.

Ez után már csak a billentyűzettől kapott irányítási információ szerint mozgatja az algoritmus a kamerát a megfelelő vektorral.

A *setPosition()* függvény segítségével közvetlenül állíthatjuk be a kamera pozícióját.

A *ProcessRigidBodyOrientation()* és a *ProcessRigidBodyMovement()* függvények által a szoftver fel van készítve a különböző *motion tracking* rendszerekkel való kommunikációra, hogy ezen adatok segítségével opcionálisan billentyűzet és egér helyett bármilyen – *markerekkel* felszerelt – tárgy mozgatásával legyen irányítható a kamera. Ezt a szoftver-funkciót a SZTAKI-ban kiépített rendszer segítségével valósítottam meg.

Ehhez meg kellett ismerkednem az *Optitrack Motive* nevű programjával, ami a kamerarendszer adatait dolgozta fel és jelenítette meg, illetve lehetőség volt streamelni az adatokat: *yaw*, *pitch*, *roll*, és elmozdulás értékek. Használnom kellett a *NatNet SDK-t* (*Optitrack* honlapján elérhető), ami segítséget nyújt azoknak a fejlesztőknek, akik saját programban szeretnék felhasználni a *Motive* szoftver által streamelt adatokat. Így sikerült implementálnom a kliens osztályt a megjelenítőm mellé, ami fogadta a streamelt adatok értékeinek a változását és megfelelő formában át is adta ezeket a kamera osztályom megfelelő függvényeinek. Mivel a *pitch* értékek az *Optitrack* konvenciója szerint csak 0-180 fokig mentek, ezért végül nem a *Motive* által kiszámolt értékeket használtam, hanem magam számoltam ki az orientációt: A merev testen(*RigidBody*) hat marker volt és ezek egy síkon helyezhetők el 3D-ben. Ebből háromat választottam ki, amivel meghatározható volt a *RigidBody* síkja, és így annak a *normálvektora* is. A háromból kettő markert úgy választottam, hogy az egyikből másikba mutató vektor jobbra mutat. Az előre mutató és a jobbra mutató vektor keresztszorzataként pedig

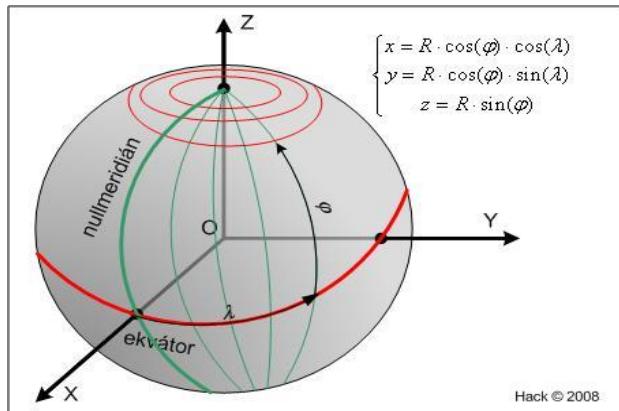
megkaptam a felfelé mutató vektort, és pont erre a három paraméterre volt szüksége a kamera osztályomnak a „kamera” aktuális orientációjának meghatározásához, így ezeket az értékeket adtam át a megfelelő függvényeknek. A kamera térbeli elmozdulását továbbra is a *Motive* által számolt és streamelt adatok alapján határoztam meg. A kész projektből ezután fordítottam egy külön könyvtárat, hogy tetszőleges megjelenítővel könnyen lehessen használni, illetve megírtam hozzá egy dokumentációt, ami egy rövid használati útmutatót és egy példafájlt tartalmazott a felhasználáshoz. (mellékletben)

Az *IncreaseMovementSensitivity()* és a *DecreaseMovementSensitivity()* függvények biztosítják a lehetőséget, hogy az egér érzékenységét dinamikusan állíthassuk a program futása közben.

A *ProcessMouseMovement()* függvény fogadja az adatokat az egéről és annak mozgása alapján mozgatja a kamerát a pontfelhőben. Ez úgy történik, hogy az egér x és y tengelyen történő elmozdulásait (*xoffset* és *yoffset*) megszorozzuk a beállított érzékenységi tényezővel, majd ezeket az értékeket hozzáadjuk az aktuális vertikális és horizontális elfordulási szögekhez (*Yaw*, *Pitch*). Ezen kívül az algoritmusba be lett építve egy védelem, mely megakadályozza, hogy felfelé vagy lefelé forgatva a kamerát túl tudunk fordulni a meghatározott határokon. (Ezek általában a lefelé és felfelé mutató vektorok.)

A *ProcessMouseScroll()* függvény az egérgörgő mozgatása által kiváltott eseményekre reagál, utolsó verziójában a megjelenítési részletességet változtatja a görgetés hatására.

Az *updateCameraVectors()* függvény a kamera új elfordulási szögeiből számolja ki a kamera jelenlegi vektorait. Ezt a  **gömbi polárkoordinátákról** való transzformációval hajtja végre az algoritmus. Ezt a függvényt a *ProcessKeyboard()* metódus hívja meg a futása végén.



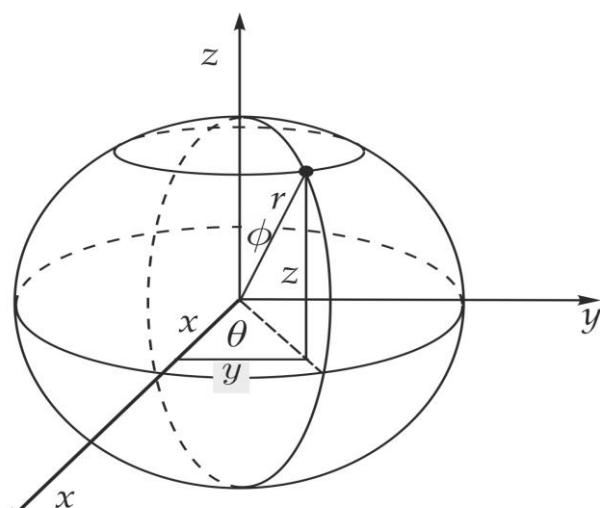
3. ábra [11]

Az algoritmusban az  $R$  tényező nincs jelen, ugyanis egység hosszúságú vektorokról beszélünk, így ez elhagyható ebben az esetben.

Egy  $P$  pontot három koordináta  $(r, \vartheta, \Phi)$  határozza meg, ahol [9]

- $r$  az origótól mért távolság, azaz a gömb sugara; [9]
- $\vartheta$  a  $P$  pont és a  $\Phi = 0$  tengely által meghatározott sík, valamint  $\vartheta = 0$  sík közötti hajlásszög az  $(x, y)$  koordinátásíkban,  $\vartheta$ -t „azimut” szögnéknél nevezik; [9]
- $\Phi$  a pontot és az origót összekötő egyenes hajlásszöge a  $\Phi = 0$  irányhoz, azaz a  $z$  tengellyel bezárt szög. [9]

A  $\vartheta$ -t a földrajzi hálózatban hosszúsági foknak (*longitude*) nevezik, továbbá  $\Phi = 90^\circ - \delta$ , ahol  $\delta$ -t pedig szélességi foknak (*latitude*) hívják. [9]



4. ábra [9]

A kód **PCL LOADING PART** részében első körben inicializáljuk az *Octree-t*, amiben a pontfelhő később tárolásra kerül. Ez után bekérjük a fájlnak a nevét, majd a *PCL* könyvtár *loadPCDFile()* metódusa segítségével beolvassuk a pontfelhőt. Ha nem található azonos nevű fájl, akkor a program futása hibával leáll. A pontfelhő szélessége, hosszúsága és magassága globális változókban kerül eltárolásra, a későbbiek során ezekre az adatokra többször is hivatkozni fog a kód. Ez után minden a három koordinátának a minimumát megkeresem, majd az egész pontfelhőt a koordinátarendszerének az origójába transzformálom, egyszerű tengelyenkénti eltolással. Effektív jelentősége ennek nincsen, minden össze az adatok könnyebb kezelése remélhető ettől (absztrakciós szinten). Ha ezek megvannak, már csak a pontfelhőt határoló téglatest átmérőjének kiszámítása marad erre a részre. Az egyes koordinátatengelyeken felvett minimum és maximum értékek ismeretében ez minden össze ez egész egyszerűen a 3D-s pontok távolságának számítására való képlettel kiszámítható. Ezen a ponton tudjuk inicializálni a előbbiekből bemutatott kamera mozgási érzékenységét olyan módon, hogy ezt az átmérőt leosztjuk egy fix számmal, így a pontfelhő mérete alapján arányosan fog változni a kamera mozgás sebessége is.

Ez után következik a *GLFW* inicializálása. Első körben meg kell adnunk a használni kívánt *OpenGL* verzió számát, illetve azt is, hogy ha ez nem lehetséges, akkor melyik korábbi verziót próbálja meg használni a könyvtár. A fő ablak adatait is itt kell megadni. Ezek a szélesség, a hosszúság és az ablak neve. Ezen kívül opcionálisan megadhatunk még számos megszorítást is, többek között lehetővé teszük a az ablak átméretezhetőségét, vagy kikapcsolhatjuk az egér mutató láthatóságát a programunkon belül. Különösen fontos ezen a ponton az úgynevezett *callback függvények* regisztrálása az „ablakozó” könyvtár számára(*GLFW*). Külön *callback függvény* tartozik a billentyűzetről jövő információk kezeléséhez, az egér pozíció változásának kezeléséhez, az egér gombokhoz, valamint az egérrel történő kattintások figyeléséhez. Ennek a résznek a végén a *glEnable(GL\_DEPTH\_TEST)* utasítással engedélyezzük, hogy azok a pontok, melyek éppen nem látszanak a képernyőn, nem kerülnek rendelerésre az adott ciklus-iterációban. Ezek a pontok kívül esnek a képernyőn, vagy takarásban vannak más pontok által. Végül pedig beállítjuk a pontoknak a méretét a *glPointSize* parancssal.

A *VBO(Vertex Buffer Object)* betöltésének első részében beolvassuk a *shader* fájljainkat. Ez két külön fájlt jelent, a *vertex shader* és a *fragment shader*. Ezeknek a fájloknak a jelentőségről és funkciójáról már írtam az előzmények bevezető részében.

Ezek után a beolvasott pontokat felosztom annyi részre, ahány logikai szál kezelésére

alkalmas az aktuális eszközben lévő processzor, majd több szálon a *C++ std::vector* tárolójába teszem a beolvasott adatokat. Ez után a létrehozzuk a *VBO-t (Vertex Buffer Object)*, majd feltöljük a beolvasott adatokkal. A *glBufferData()* nevű metódus segítségével jelezzük, hogy egy tömb adatait fogjuk tárolni, valamint a *VBO* méretét is definiáljuk, ami a feltöltött tömbünk elemszáma megszorozva a benne tárolt elemek típusának méretével. A *glVertexAttribPointer()* függvényel pedig első meghívásra megadjuk a pozíció paraméterek számát, ami három lesz (*x,y,z*), a második meghívással pedig a szín paraméterek számát, ami szintén három (*r,g,b*). Ezen kívül még megadjuk azt is, hogy a *vertex blokkokban* ezek az adatok pontosan hol kezdődnek.

A következő nagy rész a **Main Loop**, amely az ablak bezárásáig folyamatosan fut. Ezen belül a *glfwPollEvents()* metódus összegyűti az ablakkezelő könyvtár által összegyűjtött eseményeket, melyeket később a már regisztrált *callback függvények* kezelnek le. A *glm::perspective()* függvény megfelelő paraméterekkel történő meghívásával létrehozzuk a vetítési mátrixot, ezt pedig a *glUniformMatrix4f()* segítségével átadjuk a *shadernek*, hogy az kezelni tudja a VBO-ban tárolt pontok *renderelését*, ami a *glDrawArrays()* metódus meghívásával történik meg. A függvény paramétere zére pontok megjelenítése esetén a mellékelt képen látható. Végül pedig a *glSwapBuffers()* metódust meghívva elérjük a pufferek cseréjét, ami azt jelenti, hogy a folyamatos futás érdekében mindenkor nem látható adatokon dolgozunk, melyek a második pufferben vannak tárolva, majd a következő iterációban a második pufferben levő adatokon dolgozunk, míg a másik tartalma van megjelenítve.

A függvény implementáció részben találhatóak a segédfüggvények és a *callback függvények* megvalósításai is. A *Do\_Camera\_movement()* függvény felel a kamera előre, hátra, jobbra, balra, fel és le irányokba történő mozgatásáért, mely a billentyűparancs alapján meghívja a *Camera osztály* fentebb részletezett függvényét a megfelelő paraméterezéssel.

A *deleteMarkedPoints()* függvényben került implementálásra a pontok törlésének mechanizmusa. A pontfelhő tisztítása, ritkítása és feldolgozása miatt ez egy különösen fontos opció. A felhasználó által megjelölt pontokat futás közben egy címkével látjuk el, és ezt vizualizáljuk is, vagyis minden megjelölt pont színe piros. Így, a törlést elindító billentyű megnyomására a címkék alapján ezek a pontok eltávolításra kerülnek a pontfelhőből. Fontos, hogy ez csak a memoriában történik meg. A program a futás befejezésekor megkérdezi, hogy szeretnénk-e elmenteni egy új fájlba a módosított pontfelhőt. Érdemes megjegyezni, hogy a szoftver optimális működését biztosítandó, a pontok a program elején felépített *Octree-ből* is törlődnek, és az egész fát újra kell építenie ennek a függvénynek.

A *callback függvények* (*key\_callback()*, *mouse\_callback()*, *mouse\_press\_callback()*)

hivatottak biztosítani a program megfelelő működését biztosítani a különböző bemenetekre. Az egér- és billentyűzet funkciók leírása a mellékletben található.

Az *optimize\_octree\_resolution()* függvény az igényeknek megfelelően optimalizálja a pontfelhő felbontását. Ha a felhasználó olyan paramétereket ad meg a program használata során, akkor a program ezt a funkciót önállóan felajánlja.

A *drawNormals()* metódus meghívásával ki tudjuk rajzolni azokhoz a pontokhoz tartozó felületi normálisokat, amelyekre már korábban meghívtuk a *PCL* könyvtár normális-számító algoritmusát.

A *CaptureScreen()* függvénytelével lehetőség nyílik a pontfelhőből kép kivágására az aktuális kameraállás szerint. A metódus úgy lett megvalósítva, hogy *BMP* fájlban mentse el a képet. Ezt később más programoknak (például *MATLAB*) tudjuk átadni további feldolgozásra.

A *box3D.hpp* fájlban különböző metódusokat implementáltam, melyekre szükség lehet egy olyan régióövelő algoritmus használatánál, ahol a kezdőpont nem egy gömb középpontja, hanem egy téglatesthez közepe. Ilyen metódusok a téglatest kiterjedésének a kis mértékű növelés és csökkentése, a téglatest méreteinek lekérdezése, valamint a téglatesten belüli pontok gyors megtalálása.

A *VeryNaiveSphere.hpp* fájlban a szoftver kezdetleges verziójához implementáltam olyan metódusokat, melyek segítségével **animációkon** keresztül lehetett bemutatni a program lehetőségeit.

Az egyik első animáció egy labda ütközésének és visszapattanásának a megvalósítása volt pontfelhőben. A „labdát” inicializáláskor adtam hozzá pontfelhőhöz, hogy majd később szabadon lehessen benne mozgatni. A *PCL* könyvtár *normális számító algoritmusának* segítségével lehetett megoldani az ütközésvizsgálatot. Ahogy már említettem, a megjelenéstől elválasztva, a pontokat a *PCL* által biztosított *Octree struktúrába* rendeztem. Az *Octree* tulajdonságait kihasználva nagyon gyorsan lehet keresni adott sugáron belüli szomszédokat (*kNN*), így adott „felület” *normálisát* ezek alapján meg tudja határozni a *PCL* beépített algoritmusa. A visszapattanás implementálása innen már egyszerű volt, hiszen a labda mozgási irányából és normálisból egyszerű vektoralgebraival meg lehet kapni a válasz vektort.

Ez után az előző feladatot összekötve a világítással és árnyalással, a labdát, mint fényforrást definiáltam, és így implementáltam a pontok megvilágítását. Mivel a megvilágítás a valóságban nagyon komplex, ezért az *OpenGL*-ben minden a valóság megközelítésén alapszik,

amihez leegyszerűsített modelleket használunk, melyekkel sokkal egyszerűbben tudunk dolgozni. Ezek a modellek a fény fizikáján alapulnak. Ilyen modell a *Phong-árnyalás*. Ez a modell három fény komponenst különbözet meg: *ambiens*, *diffúz*, és *spekuláris*. Ezeket a komponenseket itt most az *OpenGL* referencia szerint értelmezzük. A megvilágításhoz fehér fénnnyel dolgoztam. A *shader fájl*ban felbontottam a megvilágítást az említett komponensekre. Az *ambiens* – alap megvilágítás – összetevőt gyengére állítottam a látványosság kedvéért. A *diffúz* – szort fény – összetevőnél skaláris szorzatát kell venni a fény irányának és adott pont 5 normálisának. Ha 0-nál kisebb értéket kapok, akkor automatikusan 0 lesz, hiszen, ha a bezárt szög koszinusa kisebb mint 0, akkor a pont a sötét oldalon van, ellenkező esetben pedig a világoson. Végül a *spekuláris* összetevőnél a skaláris szorzatát kell venni a nézési irány vektorának és a visszaverődési vektornak. Ezt x-edik hatványra emelem, ahol x a fényességet jelenti. Minél nagyobb x, annál gyorsabb az elhalás. Adott pont színe végül a három komponens összege megszorozva az eredeti színnel.

#### 4.3.2.)      **Android kitekintés**

Régészeti szempontból releváns lehet még az is, hogy az adataik kezelését a lehető legkönyebbé tegyük, hogy akár terepen is képesek legyenek használni egy alkalmazást anélkül, hogy PC-hez ülnének, vagy le kellene ülni a laptop elé. Ehelyett a – PC-n feldolgozott – pontfelhőt valamilyen mobil eszközön nézhetné és kezelhetné a felhasználó. Így Androidra valósítottam meg egy olyan alkalmazást, amivel a felhasználó az Androidos eszköze képernyőjén láthatja a pontfelhőt, és úgy mozoghat benne, mintha az android készülék lenne az implementált kamera. Mivel a pozíció meghatározás külső szenzorok nélkül nehézkes, valamint a GPS-t használva nagyon pontatlan, ezért ez az ág redukálva lett arra, hogy az eszköz *belső szenzorainak* a segítségével az aktuális orientációt meg lehessen határozni, az eszköz érintőképernyőjén való kattintással ugyanúgy ki lehessen jelölni objektumokat, mintha a PC-n kattintanánk, valamint a számára releváns részeket meg tudja jelölni. Ehhez a szerver oldalt C++-ban kellett megírnom, és beépíteni az eddigi programba, a kliens oldalt pedig *Javában* valósítottam meg. Az androidos alkalmazásban külön szalon fut a kliens, ami folyamatosan küldi a szervernek az eszköz szenzorai által érzékelt adatokat, melyeket aztán a C++ kódban dolgozok fel, ezzel megvalósítva a kamera mozgatását.

### 4.3.3.) Objektum detektálás

Régészeti felhasználásban különösen is releváns a különböző objektumok látványos megjelenítése és annotációja. Ehhez közelebb vivő programfunkció a pontfelhőből való objektum kiválasztása, illetve ezek elszeparálása úgy, hogy az objektumon kívüli pontfelhő részek ne sérüljenek. A pontok kijelölését úgy implementáltam, hogy az egér kattintás helyét felhasználva a kamerából a kattintás irányába kibocsátott vektor egy henger középvonalaként értelmeztem. Így a hengeren belüli pontok közül a kamerához legközelebbi pontot kiválasztva kapjuk meg a 3D-s pontot, amire kattintottam.

A hengeren belüli pontok kiválasztásához szükséges algoritmusokat a *cyl\_test.hpp* fájlban implementáltam. A megvalósításban a *GLM* könyvtár beépített algoritmusait használtam a skaláris szorzat, illetve a vektorok hosszának számításához, valamint a normalizáláshoz is. Ezek után két külön metódust valósítottam meg, melyek elegendők annak meghatározásához, hogy a hengeren belül helyezkedik-e el egy pont, vagy sem. Ezek a függvények a *dist\_Point\_to\_Line()* és a *dist\_Point\_to\_Plane()*. Előbbi a pont és egyenes távolságát implementálja, míg az utóbbi a pont és sík távolságát.

Koordináta geometria kitekintés: [15]-ből

- Két pont távolsága: a  $P_1(x_1, y_1, z_1)$  és  $P_2(x_2, y_2, z_2)$  pontok távolságát a  $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$  képlettel számoljuk ki. Ez a két pont által meghatározott vektor hossza.
- Pont és egyenes távolsága: a  $P_1(x_1, y_1, z_1)$  pont és az  $r = r_0 + vt$  egyenes távolságának meghatározásához felírjuk a  $P_1$  ponton átmenő és az egyenesre merőleges sík egyenletét, ami  $v(r - r_1) = 0$ . Kiszámoljuk, a sík és egyenes  $P_2$  döfespontját, majd a  $P_1$  és  $P_2$  pontok távolságát.
- Pont és sík távolsága: a  $P_1(x_1, y_1, z_1)$  pont és az  $n_1(x - x_0) + n_2(y - y_0) + n_3(z - z_0) = 0$  sík előjeles távolsága  $d = \frac{n_1(x_1 - x_0) + n_2(y_1 - y_0) + n_3(z_1 - z_0)}{\sqrt{n_1^2 + n_2^2 + n_3^2}}$

A függvény a két pont távolságának számításával megkapjuk a kamerából kibocsátott egyenes hosszát (henger magassága), így megállapíthatjuk, hogy a pont a henger alapjai között helyezkedik-e el, vagy sem. A pont és egyenes távolságából pedig a henger „középvonalától” való távolságot kapjuk meg, így meg tudjuk mondani, hogy az adott pont a paláston belül van-e. Ha mindkettő igen, akkor a pont a hengerben van és releváns pontnak tekintjük a kattintás

helyének meghatározásakor.

Az algoritmus helyes működéséhez fontos, hogy a paramétereket jól válasszuk meg. A henger alapjának sugara ne legyen se túl kicsi és se túl nagy. Mivel a *PCL* könyvtár segítségével *Octree* struktúrába vannak rendezve a pontok, így a legkezdetlegesebb megoldás egy *kNN* algoritmus használata, amivel megkapjuk a kattintás helyéhez legközelebb levő *k* pontot. Ahhoz, hogy ez működőképes legyen talajon és falon levő objektumokra, tudnunk kell definiálni a talaj/fal síkját. A *PCL* könyvtár beépített normális számító algoritmusának használatával megkaphatjuk annak a résznek a felületi normálisát, ahova kattintottunk, ebből pedig meghatározható a talaj síkja is. Így a pont-sík távolság képletéből elhagyva az abszolútértéket az eredmény előjele alapján meghatározható, hogy adott pont a talaj felett van-e, illetve a falon van-e.

Az objektumok határait úgynevezett „régió növelő” algoritmussal lehet a legegyszerűbben megtalálni. Kezdetben egy téglatest alakú régiót növelte a korábban már bemutatott *box3D.hpp* fájlban implementált függvények segítségével, majd felváltottam egy hagyományos – a szomszédos pontok hozzáadásával növelő – algoritmusra. Ez egy rekurzív algoritmus iteratív implementációját jelenti. Az adott sugáron belüli pontok megtalálása *Octree*-ben nagyon gyors és jó eredményt ad. Így az algoritmus a kiinduló ponttól való adott távolságot belüli pontokat keresi meg, ezekre pedig rekurzívan újra megnézi ugyanezt. Ez redundanciát okoz, amit a megvalósításkor kezelnem kellett olyan módon, hogy felcímkezem azokat a pontokat, amelyek már egyszer meg lettek vizsgálva, így rekurzió következő szintjén ezeken a pontokon már egyszerűen átugrik az algoritmus. Ez a funkció még jobb eredményt ad a talajon levő objektumokra, ha előtte egy talajszeparációs algoritmust lefuttattunk, és a szeparált pontokra futtatjuk le a programot.

Ugyanakkor régészeti szempontból ez legtöbbször nem ad releváns eredményt. A sikeressége a talajszeparálástól függ, és attól, hogy az adott objektum a talajon áll-e, vagy sem. Így pl. egy teljes várfal kijelölhető a talajon, de arra már nem nyújt megoldás ez a funkció, ha olyan apróbb részeket szeretnénk kiválasztani és felcímkezni, melyek csak részei egy talajon álló objektumnak. Ilyenek például a téglák, amikből a fal áll. Ilyen esetben ugyan használható a régió növelő algoritmus, de teljesen más hasonlósági kritériumok ellenőrzését követeli meg. Erre egy megoldás lehet, hogy a kritériumok közé vesszük a szomszédos pontok átlagos távolságát, és a szomszédos pontkból számított felületi normálisok átlagos eltérését és erre adunk meg olyan határokat, amit nem léphetnek át ezek az értékek. Ez az algoritmus néhány esetben jól működik, ha megfelelően állítjuk be a paramétereket, ugyanakkor vannak olyan

helyzetek, amivel nem tud mit kezdeni. Ilyen például az, ha egy téglában - vagy más kijelölendő objektumban - van valamilyen fajta törés, illetve repedés. Ezekben az esetekben az a futás elakadhat a repedésekknél, és az azon túli részek kimaradnak az objektumból a kijelölést követően. Emellett sűrű pontfelhőt feltételezve még lassú is a lefuttatása a kritériumok ellenőrzésének, amit első sorban a felületi normális számítás erőforrásigénye okoz.

#### 4.3.4.) MATLAB használata

Egy másik lehetséges megoldás erre, ha 2D-be levetítjük a látott képet, majd ezt fekete-fehérré alakítva lefuttatunk rajta egy olyan algoritmust, ami valamilyen metódus alapján a megfelelő küszöbértéket választja az objektum és az azon kívüli részek elkülönítésére. Ilyen metódus a MATLAB *graythresh()* függvénye, ami *Otsu algoritmusát* használja erre. Ebből az *imbinarize()* függvénytel megkaphatjuk a kiszámolt küszöb alapján szegmentált képet.

Ahhoz, hogy ez működjön, C++-ból meg kell tudnunk hívni a MATLAB függvényeinket.

A MATLAB és a C++ interakciójára három módot különböztetünk meg:

1. Abban az esetben, ha a MATLAB C++ API-ját szeretnénk használni, az úgynevezett MEX függvények használata. Ez lehetővé teszi, hogy C, C++, vagy Fortran szubrutinokat hívunk meg a MATLAB parancssorból. Ezek a bináris MEX fájlok, dinamikusan linkelt szubrutinok, amit a MATLAB interpreter tölt be és futtat. Egy MEX fájl csak egy függvényt tartalmaz, aminek a neve maga a fájlnév. A programból a fájl nevével tudjuk meghívni a függvényt (kiterjesztés nélkül).
2. Abban az esetben, ha C++-ban szeretnénk a MATLAB függvényünket meghívni lehetőségünk van az úgynevezett MATLAB Coder használatára. Ez a kiegészítő képes olvasható és hordozható C vagy C++ kódot generálni a MATLAB kódból, így ezeket a felhasználó integrálni tudja a saját projektjébe statikus, vagy dinamikus könyvtárként, illetve forráskódként.
3. Másik megoldás a C++-ban történő futtatására a MATLAB szkripteknek a MATLAB Engine használata. Ennek segítségével különböző API-kon keresztül számos programozási nyelvből képesek lehetünk lefuttatni a MATLAB kódunkat. Ilyen API létezik többek között a C, C++, Java, Python, Fortan nyelvekhez.

4. Ráadásként pedig írhatunk egy *shell szkriptet*, amiben meghívjuk a MATLAB függvényünket, majd ezt integráljuk a C++ kódba.

A MATLAB *graythresh()* függvényének eredményét a függelékben mellékeltem. Az eredményesség nagyban függ a vetítési síktól, azaz, hogy milyen szemszögből nézem, ugyanakkor ha ezt jól sikerül megtalálni, akkor a jó eredmény mellett még gyorsan is lefut, nem úgy, mint a 4.3.3 fejezet végén felvázolt lehetséges megoldás, amit a sebessékgükönbség miatt nem is tartottam meg erre a konkrét esetre. (Növényzet félautomatikus interaktív kinyerése esetén viszont jól teljesített.)

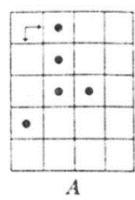
### 4.3.5.) Progresszív Morfológiai Szűrő

#### 4.3.5.1.) Matematikai háttér

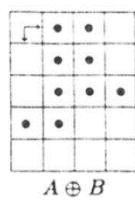
Ahogy az előzményekben már tárgyaltuk korábban, a matematikai morfológia különböző hasznos eszközöket ad a kezünkbe a képanalízishez, ezeket pedig halmazműveletekkel írjuk le. A két legfontosabb alapvető művelet a *dilatáció* (5. ábra) és az *erózió* (6. ábra). Előbbi növeli, míg utóbbi csökkenti a struktúra- és adatpontok számát. [6]

$$A = \{(0, 1), (1, 1), (2, 1), (2, 2), (3, 0)\}$$

$$B = \{(0, 0), (0, 1)\}$$



$$A \oplus B = \{(0, 1), (1, 1), (2, 1), (2, 2), (3, 0), (0, 2), (1, 2), (2, 2), (2, 3), (3, 1)\}$$

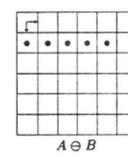
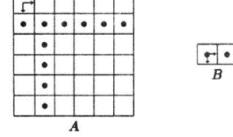
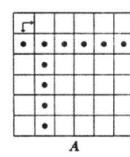


5. ábra [6]

$$A = \{(1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 1), (3, 1), (4, 1), (5, 1)\}$$

$$B = \{(0, 0), (0, 1)\}$$

$$A \ominus B = \{(1, 0), (1, 1), (1, 2), (1, 3), (1, 4)\}$$



6. ábra [6]

Ezeknek a kombinálásával valósíthatók meg a morfológiai nyitás és a morfológiai zárás műveletek, melyek közül előbbi az apró és világos objektumok eltávolítására használható,

utóbbi pedig az apró és sötét objektumokat tünteti el. Szürkeárnyalatos képek esetében a dilatáció és az erózió gyakorlatilag azt jelenti, hogy minden egyes pixel értékét helyettesítjük annak a csoportnak a minimumával (*erózió*), illetve a maximumával (*dilatáció*), melybe az adott *pixel* tartozik. Ez a csoport maga a *pixel*, és egy előre meghatározott szomszédsága a *pixel*nek.

[6]

Ez a koncepció természetesen kiterjeszthető 3D-s felületek analízisére is, vagyis lézerszkennelt pontfelhőkön is alkalmazható. Egy  $p(x,y,z)$  mérés esetében az  $x$  és  $y$  koordinátákhoz tartozó  $z$  magasság dilatációja így definiálható: [12]

$$d_p = \max_{(x_p, y_p) \in w} (z_p)$$

, ahol az  $(x_p, y_p, z_p)$  pontok a  $p$  pont szomszédjait reprezentálják egy  $w$  ablakon belül. Ez az „ablak” lehet egy-dimenziós (vonal), vagy két-dimenziós (négyzet, vagy egyéb alakok). A dilatáció kimenete a maximum magassági érték lesz  $p$  pont szomszédságában. Az erózió ennek pontosan az ellenkezője: [12]

$$e_p = \min_{(x_p, y_p) \in w} (z_p).$$

Az erózió és a dilatáció kombinációja a nyitás és a zárás, ami felhasználható lézerszkennelt pontfelhők szűrésére. A nyitásnál előbb az eróziót alkalmazzuk, ezt követően pedig dilatációt. A zárásnál ennek pontosan a fordította történik. Egy-dimenziós ablakot használva egy 2D-s képen a nyitás jól látható eredményt ad. Az erózió eltünteti azokat a fapontokat, melyek az adott ablakon belülre esnek, míg a dilatáció helyreállítja a nagy épületek pontjait. Az ablak méretének helye megválasztásával megvédi azokat az annál nagyobb struktúrák kiszűrését. [12]

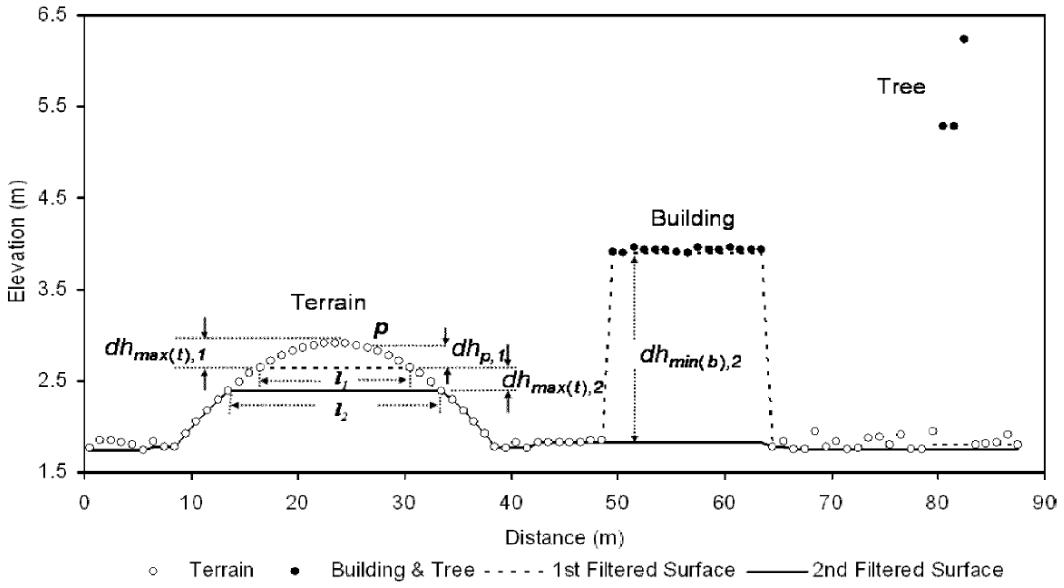
*Kilian* egy olyan metódust publikált a nem talaj objektumok eltüntetésére morfológiai szűrővel, melyben a megadott ablak méretben egy morfológiai nyitás után detektálta azt a pontot, melynek a legkisebb volt a magassága. [2] Ezután pedig a talaj részeként jelölte meg mindeneket a pontokat, melyek ezen az ablakon belül a legkisebb magassághoz képest ez előre meghatározott számon belül esnek. Ennek a sávnak a távolsága erősen függ a felmérés pontosságától, de általában 20-30 cm. Az összes talaj-pont úgy kerül azonosításra, hogy az egész adaton „végig mozgatjuk” az ablakot. Az ablak méretének a kiválasztása, valamint az épületek

és a fák eloszlása a tér egyes részein kritikus tényezők az algoritmus helyes működéséhez. Ha túl kicsi az ablak mérete, akkor a legtöbb talaj-pont meg lesz védve a szűrőtől, azonban csak a kis objektumok lesznek megfelelően eltávolítva, mint az autók és a fák. Azok a pontok viszont nem távolíthatók el, melyek a városi környezetben levő magas épületek tetején vannak. Nagy tehát az esélye annak a hibának, amikor a különböző objektumok pontjait is talaj-pontként azonosítja az algoritmus. A másik oldalról sem egyszerűbb a dolog, hiszen a túl nagy méretű ablak esetén pedig rengeteg pontot fog eltávolítani a szűrő a talaj pontjai közül is. (Például vízelvezető csatornák közelében a teljes útszakasz el lesz távolítva. Emellett levágásra kerülnek gyakran a homokdűnékhez hasonlító képződmények is egy lapos tengerparti területen.) Ideális esetben az ablak mérete pontosan akkora, hogy egyik hiba se fordulhasson elő, de ez a valós életben sajnos nem reprodukálható. [12]

Ennek a problémának a kiküszöbölésére azt a megoldást találták ki, hogy több különböző ablak méret is legyen használva, a legkisebbtől kezdve, és mindenki mérettel futtassák le az adatokon az algoritmust. [2] minden pont meg lesz jelölve egy súlyval, ami az alapján kerül meghatározásra, hogy milyen ablak mérettel lett a talaj pontjaként azonosítva. Minél nagyobb az ablak mérete, annál nagyobb a ponthoz rendelt súly. Ezután a talaj az egyes pontok súlyainak felhasználásával kerül véglegesen meghatározásra. A súlyok felhasználása ugyan lehetőséget nyújt a talaj felületének jobb meghatározására, a metódus nem jelent fejlőést a lézerszkennelt pontfelhők talajszeparációját tekintve.

Az előzményekben említett morfológiai szűrőkben számos probléma található, mint például az előre meghatározott ablak méret szükségessége. Emellett egy nagyrészt automatikusan működő szűrő jelentheti a jó megoldást, mely kijavítja az elődjei hibáit. Ennek fényében terveztek a *Progresszív Morfológiai Szűrőt* (PMF), melyet a *PCL* könyvtárban is implementáltak a talaj pontok szegmentálásához a *(./pcl/segmentation/progressive\_morphological\_filter.h)* fájlban.

Láthattuk, hogy ezek a szűrők jól használhatók arra, hogy épületeket és fákat lehessen a pontfelhőből kivágni, de a fix ablak méret miatt sajnos nem jók arra, hogy minden egyes objektumok detektáljanak, melyek nem a talaj részei. [2]



7. ábra [12]

A PMF használatakor megfigyelhető, hogy egy  $l_1$  széles ablakkal történő morfológiai nyitás után a nagyobb objektumok (például épületek) nem kerülnek eltávolításra, hiszen azok mérete nagyobb, mint  $l_1$ , míg a kisebb objektumok, mint az autók sikeresen kiszűrhetők, mivel a méretük kisebb, mint  $l_1$ , és ezeknek a pontoknak a magassági értéke helyettesítve lett az adott ablakon belüli legalacsonyabb magassági értékkal. A következő iterációban  $l_2$  emelkedik az ablak mérete és újra végig futtatjuk a morfológiai nyitást. A nagyobb ablak méret miatt eltávolításra kerülnek a nagyobb épületek is, és az  $l_1$  ablakmérettel való mérésből kapott talaj pontok magassági értékével kerülnek helyettesítésre az eredeti adatok. [12]

Azáltal tehát, hogy fokozatosan növekedő ablak méretet használunk, a PMF sikeresen el tudja távolítani a fákát és az épületeket a pontfelhőkből. Ennek ellenére olyan felületeket is visszakapunk, melyek a terep alatt fekszenek, ami ahhoz vezet, hogy a magasabban fekvő talaj részleteket is eltávolítja a szűrő. Még lapos talaj esetében a szűrő lefutása után általában kisebb lesz a felület, mint az eredeti mérésekben volt. Éppen ezért az eredeti felmérésből származó legtöbb pont eltávolításra kerül, és csak a szűrt pontfelhőt kapjuk vissza. Ez kiküszöbölnihető az által, hogy magassági különbségi küszöbértékeket határozunk meg a különböző méretű objektumok magassága alapján.

Általánosan megfigyelhetjük, hogy sokkal nagyobb magasságbeli különbség van egy épület teteje és alapja között, mint például a talaj különböző pontjainak magassága között, hiszen utóbbi esetben a magasság fokozatosan, lépcsőzetesen változik legtöbbször. Ez a megfigyelés is segít elkülöníteni egymástól a talaj és az épületek pontjait. Tegyük fel, hogy  $dh_{p,1}$

jelöli a magasságbeli különbséget egy az eredeti pontfelhő és a szűrt pontfelhő bármely p pontjában, valamint  $dh_{T,I}$  jelöli magasság változásának a küszöbértékét. Így p pont talajpontként lesz klasszifikálva, ha  $dh_{p,I} \leq dh_{T,I}$  és nem talaj pontként, ha  $dh_{p,I} > dh_{T,I}$ . Az eredeti pontfelhő talaja és a szűrt változat közötti maximum magassági különbséget  $dh_{max(t),I}$  jelöli. Abban az esetben, ha a küszöbérték úgy kerül kiválasztásra, hogy nagyobb ennél a magassági értéknél, akkor a pontfelhő talajának minden része „védett” lesz a szűréstől, ezért ezt a paramétert jól kell megválasztani, hogy ez teljesüljön. A küszöbérték legtöbbször az ablak méret függvénye. [12]

A második iterációban az eredeti jelölést megtartva jelöljük  $dh_{max(t),2}$ -vel az előző iteráció és a mostani iteráció talajpontjai közötti maximális magasság-különbség nagyságát. Ezen az értéken belül eső különbségek esetén a talajpontok véde lesznek a szűréstől, hiszen ez az érték kisebb, mint a második iterációhoz választott küszöbérték. Emellett  $dh_{min(b),2}$  reprezentálja a minimális magasság-különbség értéket az előző iterációban levő épület pontok, valamint a mostani iteráció pontjai között, ami általában közel van az épület magasságához. Éppen ezért az épületek el lesznek távolítva, hiszen ez a szám nagyobb lesz a küszöbértéknél. [12]

Általában a küszöbérték ( $dh_{T,k}$ ) úgy kerül meghatározásra a k. iterációban, hogy az a legkisebb épület magasságával legyen egyenlő. Ezt a küszöbértéket használva a k. iterációban p pontot talajpontként jelölünk, ha  $dh_{p,k} < dh_{T,k}$ . [12] Ellenkező esetben nem klasszifikáljuk talajpontként a pontot. Ezen módszer segítségével azonosíthatók a különböző méretű épületek úgy, hogy fokozatosan emeljük az ablak méretet, minden egyes esetben lefuttatunk egy morfológiai nyitást, és ezt addig ismételjük, amíg el nem érjük a maximum ablak méretet, ami már nagyobb, mint a legnagyobb épület mérete. Mivel a fák pontjai és a talajpontok között is jelentős hirtelen eltérés van a magassági értékekben, ezért ugyanolyan jól el lehet tüntetni a fákat is ezzel a módszerrel. Fontos, hogy a szűrő által eltávolított pontok nincsenek felhasználva a digitális terepmodell alkészítéséhez.

#### 4.3.5.2. Paraméterek [12]

A PMF használata esetén rendkívül fontos a paraméterek megfelelő beállítása ahhoz, hogy jó eredményeket kaphassunk. Ilyen például az ablak méret, illetve a küszöbérték meghatározása. Az ablak méretének kiválasztására ideális lehet a következő formula:  $w_k = 2kb + 1$ , ahol a k az adott iteráció száma ( $k=1,2,\dots,M - M$  darab iteráció esetén), b pedig az ablak kezdő mérete. Így a maximális mérete az ablaknak  $2Mb + 1$ . Ha így választjuk meg az

ablak méretét az biztosítja számunkra, hogy az szimmetrikus lesz a középpont körül. (Ez a morfológiai nyitás miatt fontos tulajdonság, egyszerűbbé teszi annak az implementációját.) Az ablak méretének lineáris növelésének köszönhetően a fokozatosan változó topográfiai tulajdonságok ellenére sem lesz kiszűrve talajpont, viszont sajnos a számítási idő jelentősen megnő azokon a területeken, ahol sok és nagy olyan objektum van, mely nem része a talajnak. Ehelyett az ablak méretét növelhetjük exponenciálisan is, ha le szeretnénk csökkenteni az iterációk számát:  $w_k = 2b^k + 1$ , ahol  $b$  az exponenciális függvény alapja (és az ablak kezdő mérete),  $k$  az aktuális iteráció száma ( $k=1, \dots, M$ ) és  $2b^M + 1$  a maximum ablak méret. [12]

A küszöbérték meghatározása erősen függ az adott területen levő talaj lejtésétől. A talaj lejtését a következőképpen határozzuk meg (feltéve, hogy konstans a lejtés mértéke): [12]

$$s = \frac{dh_{\max(t), k}}{(w_k - w_{k-1})}$$

Ebből a küszöbértéket így kapjuk meg: [12]

$$dh_{T,k} = \begin{cases} dh_0, & \text{if } w_k \leq 3 \\ s(w_k - w_{k-1})c + dh_0, & \text{if } w_k > 3 \\ dh_{\max}, & \text{if } dh_{T,k} > dh_{\max} \end{cases}$$

, ahol  $dh_0$  a kezdő küszöbérték,  $s$  a lejtés,  $c$  az adott cella mérete, illetve  $dh_{\max}$  pedig maximális magasságbeli eltérés küszöbértéke.

Városi környezetben az elsődleges objektumok, melyek nem a talaj részei, az autók, a fák és az épületek. Az egyes autók és fák általában kisebbek, mint egy épület, ezért a legtöbbjük az első pár iterációban már eltávolításra kerül, amíg a nagy épületek csak utolsóként kerülnek eltüntetésre. A maximális magasságbeli eltérés küszöbértéke ( $dh_{\max}$ ) meghatározható egy fix magasságként (például a legalacsonyabb épület magassága), ezzel biztosítva, hogy minden épület azonosítva legyen a pontfelhőben.

Ugyanakkor a régészeti felhasználás során legtöbbször nem városi környezetről van szó, így előfordulhatnak, egyes, dombos, sziklás vidékek, ahol főleg a növényzet elkülönítése a cél. A fák eltávolításához nem szükséges fix maximális magasságbeli eltérés küszöbértéket meghatározni globálisan. Ez az érték általában az adott területen található legnagyobb

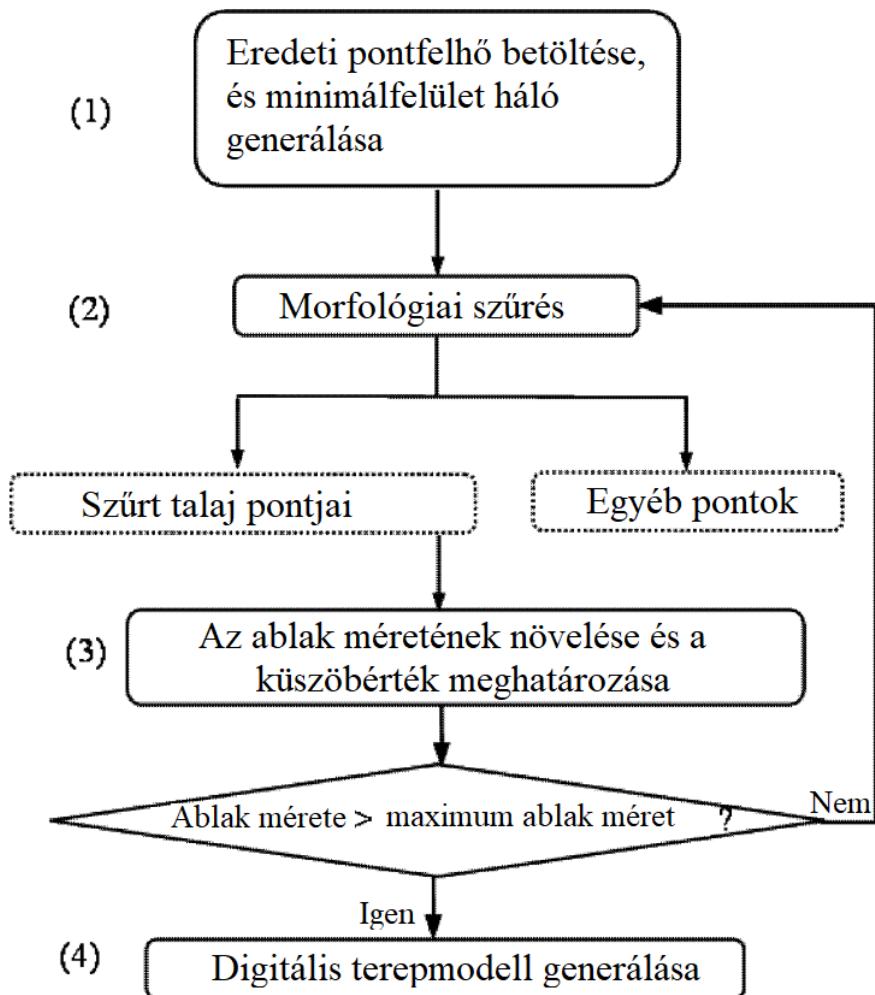
magasságbeli különbségre lesz beállítva, lokálisan.

Ezeket a paramétereket a PCL könyvtárban a következőképpen állíthatjuk be:

```
pcl::ProgressiveMorphologicalFilter<pcl::PointXYZ> pmf;  
pmf.setInputCloud (cloud); // bemeneti pontfelhő  
pmf.setMaxWindowSize (20); // maximális ablak méret  
pmf.setSlope (1.0f); // talaj dölésének beállítása  
pmf.setInitialDistance (0.5f); // Kezdő magasság beállítása(talaj  
felett)  
pmf.setMaxDistance (3.0f); // Max magasság beállítása
```

#### 4.3.5.3. A PMF lépései [12]

1. Betöltésre kerülnek a rendezetlen pontfelhő egyes pontjai (x, y, z) koordinátákkal. Egy minimál felület rendezett hálót generál az algoritmus az által, hogy minden egyes cellában kiválasztja a minimum magasságot. A pontok koordinátái (x, y, z) az egyes cellákon belül vannak eltárolva. Ha egy cellában nincsenek pontok, akkor az koordinátáinak az értékét egyenlővé tesszük a hozzá legközelebb eső pont koordinátáinak értékével.
2. A *PMF*, melynek fő komponense a morfológiai nyitás művelete, alkalmazásra kerül a hálóra. Az első iterációban a minimál felület háló, valamint a kezdeti ablak méret együttes határozzák meg a szűrő bemenetét. minden későbbi iterációban a már szűrt adatból és a 3. lépésből kapott megnövel ablakméretből kapjuk a bemeneti paramétereket. Ennek a lépésnek a kimenete a következőket tartalmazza:
  - a. a szűrő által lesimított felület
  - b. a detektált pontok, melyek nem részei a talajnak a meghatározott küszöbérték alapján
3. Az ablak méretét megnöveljük és kiszámítjuk az új küszöbértéket. Ez után a 2. és a 3. lépést addig ismételgetjük, amíg az ablak mérete el nem éri az előre meghatározott maximum értéket. Ezt az értéket a legnagyobb épület méreténél kicsit nagyobbra szokás beállítani.
4. Az utolsó lépés a digitális terepmodell legenerálása az után, hogy az összes objektum el lett távolítva a szűrő által, és csak a talaj lett meghagyva.



8. ábra [12]

A teljes algoritmus és pszeudokód megtalálható a mellékletekben az 53. oldalon. Az eredmény képek kis pontfelhőre való futtatásra a függelékben találhatóak. (62. oldal)

Amint láthatjuk, a *PMF* bizonyos esetekben megfelelő megoldás lehet, ugyanakkor számos hibalehetőséget is tartogat magában, illetve a *PCL* könyvtárban implementált változat nagy adatokra rendkívül lassú lefutást produkál, vagy le se fut. Utóbbi esetben valószínűleg a kellő mennyiségű memória lefoglalása okozhat gondot, ugyanis ezekben az esetekben kivétel nélkül azt tapasztaltam, hogy felszabadításra kerül a beolvasott pontfelhő számára lefoglalt terület, miközben a két kimeneti pontfelhő (1. talaj, 2. egyéb objektumok) írása még nem kezdődött el, és a beolvasott pontfelhő feldolgozása még nem fejeződött be.

#### **4.3.6.) Megvalósított talajszeparálás**

A szoftverben jelenleg futtatott megoldás bizonyos mértékben tartalmaz hasonló megfontolásokat, ugyanakkor nagy pontfelhőre is viszonylag gyorsan lefut. Kezdetben ebben az algoritmusban is az x és y koordináták szerinti globális minimum és maximum megkeresése történik, majd ugyanúgy legenerálásra kerül négyzetes háló, melyen belül a lokális minimum pontokat keressük a magasság ( $z$  koordináta) szerint. A szűrés után viszont a *PCL::StatisticalOutlierRemoval* segítségével eltávolítjuk a különálló pontfelhő részleteket, vagyis eltakarítjuk a talajszeparálás után visszamaradt „szemetet”. [19]

A talaj detektálása mellett a növények eltüntetését is segíti ez az implementáció. Az metódus többszöri lefuttatása után azt tapasztaljuk, hogy bár egyre több talaj-pont tűnik el, emellett egyre több hiba is csúszik a szeparálásba, vagyis sok pontot veszítünk el a régészeti szempontból releváns objektumokból is, például rom, várfal. Ugyanakkor a fákat és a bokrokat, szinte teljes egészében meghagyja az algoritmus, így sokadik futtatás után kiválóan használható a korábbiakban bemutatott funkció, mellyel különböző objektumokat lehet kijelölni. A szeparált pontfelhőből így egyszerűen csak a fákat befoglaló alakzatokra van szükség, majd az eredeti pontfelhő megjelenítésekor egyszerűen kihagyjuk ezeket a területeket, így a talaj eltüntetése mellett megmaradnak a releváns objektumok és eltüntetésre kerül a növényzet – egy része – is.

#### **4.3.7.) Annotálást segítő eszközök implementációja**

Ahhoz, hogy a pontfelhőn a felhasználó értékes munkát tudjon végezni, nagyon fontos, hogy az egyes elemeket fel tudja címkezni, és később ugyanott tudja folytatni a munkát ugyanazzal az állománnal, ahol előtte befejezte. vagyis, ha például sikerül detektálni a történetet, akkor ezt az eredményt érdemes valamilyen formában eltárolni, hogy a következő használat során már kelljen újból lefuttatni a klasszifikáló algoritmust. Pontosan ezért egy ilyen funkció is megvalósításra került a programomban.

A legkézenfekvőbb megoldás egyszerűen az adatpontok megjelölése még egy attribútum segítségével. Ez viszont akár az egy negyedével is megnövelhetné a pontfelhő méretét. Így helytakarékkosság céljából az implementációm azt a megoldást tartalmazza, melyben az objektumokat befoglaló téglatest paramétereit tárolom egy külső fájlban, majd

később ebből a fájlból töltöm be újra az adatokat. Ez a legtöbb esetben jól működik. Például egy fa lombkoronája esetében nem gond, hogy ha ezt az információt csak a talajszeparált pontfelhő alapján tudjuk kinyerni, viszont, ha egy talajon levő kisebb objektum is a befoglaló téglatesten belül lesz, akkor az a pont tévesen lesz felcímkézve.

Ennek elkerülésére az implementáció lehetőséget ad ezeknek a hibáknak az utólagos javítására. A különböző címkekhez tartozó adatokat külön fájlokba menthetjük el, a későbbi használatkor pedig több fájlból együttesen tölthetjük vissza az adatokat, így azokat a fájlokat használhatja a felhasználó, amelynek tartalmára éppen szüksége van. Vagyis, ha egy előző használat során a felhasználó sikeresen felcímkézte azokat az objektumokat, amelyre neki szüksége, elegendő kiolvasni azokhoz a címkekhez tartozó fájlok adatait, melyekre szükség van, így felesleges adat nem kerül betöltésre és megjelenítésre. Több különböző címkehez tartozó objektumok elkülönítésére azok színének megváltoztatásával ad lehetőséget a szoftver.

## 5.) Összefoglalás, értékelés

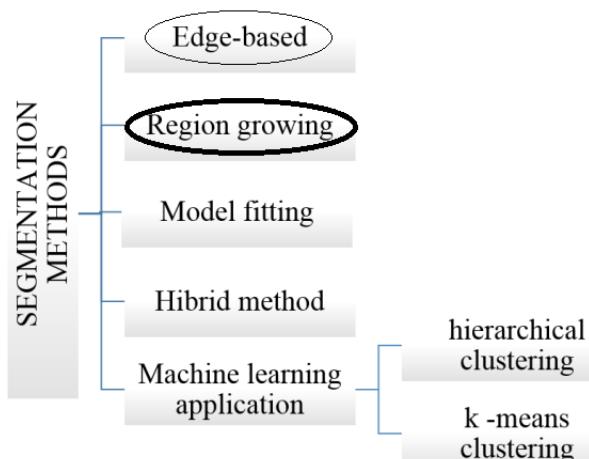
A dolgozatban bemutattam a háromdimenziós adatgyűjtő eszközök főbb típusait, működési elvét, és egy általanosságban összehasonlítottam azok felhasználhatóságát és a főbb paramétereiket. Az egyes eszközök által használt technikák mellett kirétem a mérési távolságra és a felbontás fontosságára is. Ezen kívül foglalkoztam a mérési- illetve feldolgozási időbeli különbségekkel is, valamint azzal, hogy ezen részfeladatok elvégzése milyen képességeket igényel, hiszen ez is eszközönként változó. Megemlítettem még az eszközökhöz járó hardveres és szoftveres felszerelést, illetve megmutattam az alkalmazásbeli különbségek nyújtotta lehetőségeket is. Ezen kívül bemutattam öt konkrét eszközt, ismertetve a paramétereiket, valamint egyenként összegyűjtve kifejtettem az eszközök előnyeit és hátrányait is.

A szakirodalomkutatás keretében olyan eredményeket gyűjtöttem össze és elemeztem, melyek a régészeti pontfelhők manuális és félautomatikus elemzésével kapcsolatban relevánsak, vagyis minimális emberi interakciót igénylő klasszifikációs módszereket írnak le, mellyel lehetőség nyílik elkülöníteni a talaj, a növényzet, illetve a mesterséges objektumok régióit. Az eredményeken kívül feltüntettem azok felhasználhatóságát, hibáit és gyengeségeit az általános és régészeti felhasználás tekintetében is.

A szoftverben a modern *OpenGL* paradigmát követve effektív eljárást implementáltam azok megjelenítésére és annotációjára. Mélyebb betekintést adtam a fejlesztésbe is. Az implementáció fő részeit bemutatva fejtettem ki a programfunkciók létrejöttének részleteit, és azok változásait.

## 6.) Továbblépési lehetőségek, értékelés

Minden szoftver esetében elmondható, hogy soha nem lehet rá azt mondani, hogy teljesen elkészült. Ez természetesen itt is igaz. A funkciók terén a bővítési lehetőségek száma szinte végtelen, de fő szempont lehet a szoftver kezelésének átalakítása – konzultációk útján – úgy, hogy az a régészeti felhasználás során nyújtson több lehetőséget, még akkor is, ha ezzel rontunk a más területen való felhasználhatóságon. Amellett, hogy a jelenleg implementált algoritmusokban és megoldásokban is van lehetőség a fejlesztésre és optimalizációra, kézenfekvő a szoftver bővítési lehetősége olyan metódusok irányába, melyek alternatívát is nyújtanak a felhasználónak. Az implementáció során leginkább a *régiónövelés* alapú algoritmusok kerültek előtérbe, valamint érintve voltak az „*edge-based*” megoldások is, ugyanakkor a megvalósítás nem tartalmaz *modell illesztés*en alapuló metódusokat és a *gépi tanulást* is nélkülözi. [10] Ezek pedig egyértelműen olyan területek, ahonnan számos biztató eredmény jön, így a szoftver bővítésének alapjául szolgálhatnak.



9. ábra [10]

## **7.) Köszönetnyilvánítás**

Köszönnettől tartozom témavezetőmnek, Dr. Benedek Csabának, aki az útmutatás mellett mindenkorral ellátott a megfelelő információkkal.

Szeretném megköszönni Dr. Jankó Zsolt segítségét is, akinek a felmerülő problémákhoz való profi hozzáállása segített megerősíteni a benne kialakulóban lévő mérnöki szemléletmódot.

Emellett szeretnék köszönetet mondani az MTA SZAKI Gépi Érzékelés Kutatólaboratóriumának, ami a munkához az infrastruktúrát – részben – biztosította.

A szakdolgozat a PPKE ITK EFOP- 3.6.2-16-2017-00013 azonosító számú „Innovatív informatikai és infokommunikációs megoldásokat megalapozó tematikus kutatási együttműködések” című projekt támogatásával jött létre.

## 8.) Irodalomjegyzék

- [1] L. Szirmay-Kalos, **Számítogépes grafika**. Budapest: ComputerBooks, 2001.
- [2] J. Kilian, N. Haala, and M. Englich, “**Capture and evaluation of airborne laser scanner data,**” *Int. Arch. Photogramm. Remote Sens.*, vol. 31, pp. 383–388, 1996.
- [3] R. B. Rusu and S. Cousins, “**3D is here: Point Cloud Library (PCL),**” 2011 *IEEE International Conference on Robotics and Automation*, Shanghai, 2011, pp. 1-4.
- [4] Niloy J. Mitra and An Nguyen. 2003. **Estimating surface normals in noisy point cloud data.** In *Proceedings of the nineteenth annual symposium on Computational geometry (SCG '03)*. ACM, New York, NY, USA, 322-328.
- [5] J. Kammerl, N. Blodow, R. B. Rusu, S. Gedikli, M. Beetz and E. Steinbach, “**Real-time compression of point cloud streams,**” 2012 *IEEE International Conference on Robotics and Automation*, Saint Paul, MN, 2012, pp. 778-785.
- [6] R. M. Haralick, S. R. Sternberg, and X. Zhuang, “**Image analysis using mathematical morphology,**” *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-9, pp. 523–550, 1987.
- [7] M. Miknis, R. Davies, P. Plassmann and A. Ware, “**Near real-time point cloud processing using the PCL,**” 2015 *International Conference on Systems, Signals and Image Processing (IWSSIP)*, London, 2015, pp. 153-156.
- [8] Factum Arte, S. (2017). *Factum Arte :: 3D Scanning for Cultural Heritage Conservation.* [online] Factum-arte.com. Available at: <http://www.factum-arte.com/pag/701/3D-Scanning-for-Cultural-Heritage-Conservation> [Accessed 19 Dec. 2017].
- [9] Aries.ektf.hu. (2017). *3. fejezet - Koordináta-rendszerek (Coordinate system).* [online] Available at: <http://aries.ektf.hu/~hz/pdf-tamop/pdf-01/html/ch03.html> [Accessed 21 Dec. 2017].
- [10] Grilli, E., Menna, F. and Remondino, F. (2017). **A REVIEW OF POINT CLOUDS SEGMENTATION AND CLASSIFICATION ALGORITHMS.** *ISPRS - International*

*Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-2/W3, pp.339-344.

- [11] Hu.wikipedia.org. (2017). *Gömbi koordináták*. [online] Available at: [https://hu.wikipedia.org/wiki/G%C3%B6mbi\\_koordin%C3%A1t%C3%A1k](https://hu.wikipedia.org/wiki/G%C3%B6mbi_koordin%C3%A1t%C3%A1k) [Accessed 19 Dec. 2017].
- [12] Keqi Zhang, Shu-Ching Chen, Whitman, D., Mei-Ling Shyu, Jianhua Yan and Chengcui Zhang (2003). **A progressive morphological filter for removing nonground measurements from airborne LIDAR data.** *IEEE Transactions on Geoscience and Remote Sensing*, 41(4), pp.872-882.
- [13] M. Roggero, “**Airborne laser scanning: Clustering in raw data,**” *Int. Arch. Photogramm. Remote Sens.*, pt. 3/W4, vol. 34, pp. 227–232, 2001.
- [14] Remondino, F. (2011). **Heritage Recording and 3D Modeling with Photogrammetry and 3D Scanning.** *Remote Sensing*, 3(12), pp.1104-1138.
- [15] Math.bme.hu. Ádám Katalin. (2014). „**Koordinátageometria**” [online] Available at: [http://math.bme.hu/~adamk/PDF/2014\\_koordinatageometria\\_elmelet.pdf](http://math.bme.hu/~adamk/PDF/2014_koordinatageometria_elmelet.pdf) [Accessed 21 Dec. 2017].
- [16] N. Pfeifer, P. Stadler, and C. Briese, “**Derivation of digital terrain models in the SCOP++ environment,**” in *Proc. OEEPE Workshop on Airborne Laserscanning and Interferometric SAR for Digital Elevation Models*, Stockholm, Sweden, 2001.
- [17] R. G. Congalton, “**A review of assessing the accuracy of classifications of remotely sensed data,**” *Remote Sens. Environ.*, vol. 37, pp. 35–46, 1991.
- [18] Magyarregeszet.hu. (2017). „**3D szkennerek alkalmazása a régészethez.**” [online] Available at: [http://www.magyarregeszet.hu/wp-content/uploads/2013/07/Feher\\_13Ny1.pdf](http://www.magyarregeszet.hu/wp-content/uploads/2013/07/Feher_13Ny1.pdf) [Accessed 21 Dec. 2017].
- [19] Leichner Dávid, „**Objektumfelismerés és változás követés Lidar pontfelhősorozatokon**”, szakdolgozat, PPKE-ITK, 2016.
- [20] G. Vosselman, “**Slope based filtering of laser altimetry data,**” *Int. Arch. Photogramm. Remote Sens.*, pt. B4, vol. 33, pp. 958–964, 2000.

# Függelék

## A. Szoftver használata

- irányítás: w (előre), s (hátra), a (balra), d (jobbra), szóköz (felfelé), valamint egérrel az irány változtatása
- megnyitáskor két fájlnevet kér, első az eredeti pontfelhő (.pcd formátumban), második pedig a talajszeparált pontfelhő, ha a másodikat üresen hagyjuk, akkor a szoftver automatikusan generálja a talajszeparált pontfelhőt
- objektumok kijelölése: középső egérgomb (floodfill algoritmus), jobb egérgomb (befoglaló téglatest + region growing)
- kijelölt objektumok törlése: e
- kijelölés megszüntetése: r
- kijelölt objektumok felcímkézése (fájl generálása): p
- előző címkézés betöltése: l
- eredeti pontfelhő megjelenítése: v
- talajszeparált pontfelhő megjelenítése: c
- bezáráskor a szoftver megkérdezi, hogy szeretnénk-e elmenteni a szerkesztett pontfelhőt, és igen ('y') válasz esetén meg kell adni az új pontfelhő fájl nevét (.pcd kiterjesztéssel)

*A teljes forráskód megtalálható az alábbi linken:*

*<https://github.com/bacat/thesis>*

## B. Progresszív morfológiai szűrő [12]

[12]-ből átemelve a PCL által implementált algoritmus pontos ismertetése végett

Dilation( $Z, w_k$ ):

1. for  $j = 1$  to  $n$
2.  $Z_f[j] = \max_{j-[w_k/2] \leq l \leq j+[w_k/2]} (Z[l])$
3. return  $Z_f$

Erosion( $Z, w_k$ ):

1. for  $j = 1$  to  $n$
2.  $Z_f[j] = \min_{j-[w_k/2] \leq l \leq j+[w_k/2]} (Z[l])$
3. return  $Z_f$

**Algorithm 1:** The progressive morphological filtering algorithm

**Input:**

- o A set of points representing LIDAR measurements. Each point has three components ( $x$ ,  $y$ , and  $z$ ) to represent horizontal coordinates and elevation of a LIDAR measurement.
  - o Cell size  $c$ .
  - o Parameter  $b$  in (4) or (5).
  - o Maximum window size.
  - o Terrain slope  $s$ .
  - o Initial elevation difference threshold  $dh_0$ .
  - o Maximum elevation difference  $dh_{\max}$ .
- Output:**
- o Two sets of the classified points representing ground and nonground measurements.
  - 1. Determine the minimum and maximum  $x$  and  $y$  values.

2. Determine the numbers of rows ( $m$ ) and columns ( $n$ ) using  $m = \text{floor}[(\max(y) - \min(y))/c] + 1$  and  $n = \text{floor}[(\max(x) - \min(x))/c] + 1$ .
3. Create a 2-D array  $A[m, n]$  for LIDAR points,  $p(x, y, z)$ . Traverse every point to determine the cell in which the point will fall according to its  $x$  and  $y$  coordinates. If more than one point falls in the same cell, select the one with minimum elevation.
4. Interpolate elevation of cells in  $A$  which do not contain any points using the nearest neighbor method. Set the  $x$  and  $y$  coordinates of those interpolated cells as zero to distinguish them from those cells that contain LIDAR points. Copy  $A$  to  $B$ . Initialize elements of a 2-D integer array  $\text{flag}[m, n]$  with 0.
5. Determine series of  $w_k$  using (4) or (5), where  $w_k \leq$  maximum window size.
6.  $dh_T = dh_0$
7. for each window size  $w_k$
8. for  $i = 1$  to  $m$
9.  $P_i = A[i, :]$  ( $A[i, :]$  represents a row of points at row  $i$  in  $A$  and  $P_i$  is a 1-D array)
10.  $Z \leftarrow P_i$  (Assign elevation values from  $P_i$  to a 1-D elevation array  $Z$ )
11.  $Z_f = \text{erosion}(Z, w_k)$
12.  $Z_f = \text{dilation}(Z_f, w_k)$
13.  $P_i \leftarrow Z_f$  (Replace  $z$  values of  $P_i$  with the values from  $Z_f$ )
14.  $A[i, :] = P_i$  (Put the filtered row of points  $P_i$  back to row  $i$  of array  $A$ )
15. for  $j = 1$  to  $n$
16. if  $Z[j] - Z_f[j] > dh_T$  then  $\text{flag}[i, j] = w_k$
17. end for  $j$  loop
18. end for  $i$  loop
19. if ( $dh_T > dh_{\max}$ )
20.  $dh_T = dh_{\max}$
21. else
22.  $dh_T = s(w_k - w_{k-1})c + dh_0$
23. end for window size loop
24. for  $i = 1$  to  $m$
25. for  $j = 1$  to  $n$
26. if ( $B[i, j](x) > 0$  and  $B[i, j](y) > 0$ )
27. if ( $\text{flag}[i, j] = 0$ )
28.  $B[i, j]$  is a ground point
29. else
30.  $B[i, j]$  is a nonground point
31. end for  $j$  loop
32. end for  $i$  loop

## C. Eredmény képek

Eredeti bemeneti pontfelhő, módosítások nélkül

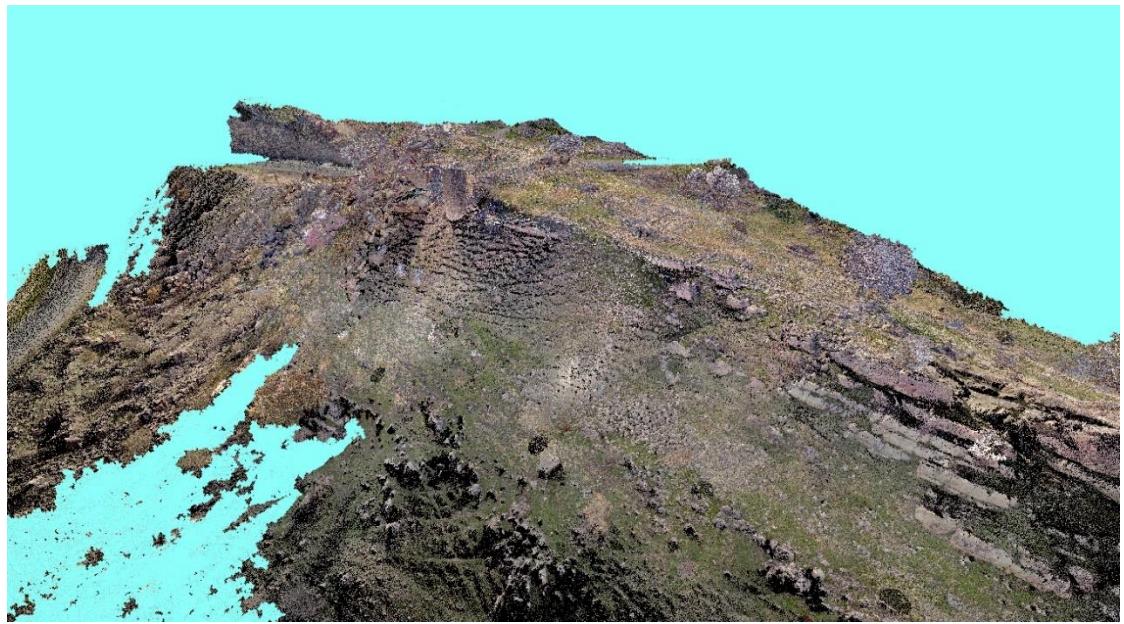
- lézerszkenneléssel nyert állomány egy ásatás helyszínéről – Dwin vára, Irak
  - az állomány 7-szeres ritkításon esett át ( minden hetedik pontot tart meg)



1. kép

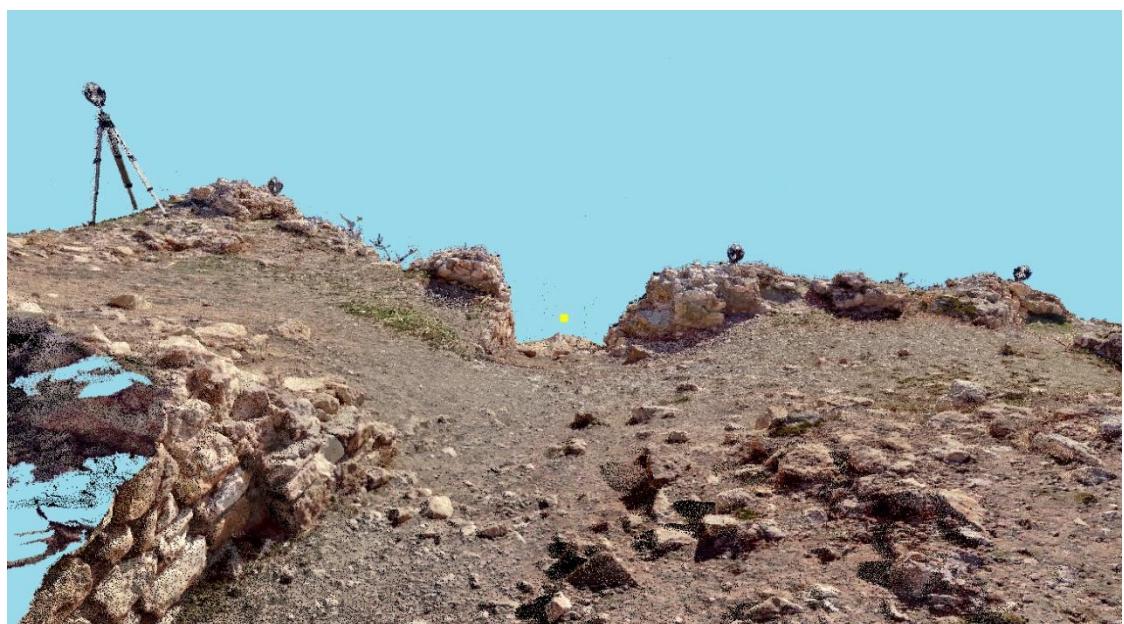


2. kép



3. kép

- Az utolsó három kép az eredeti pontfelhő egy kivágott részéről készült ritkítás nélkül!



4. kép



5. kép



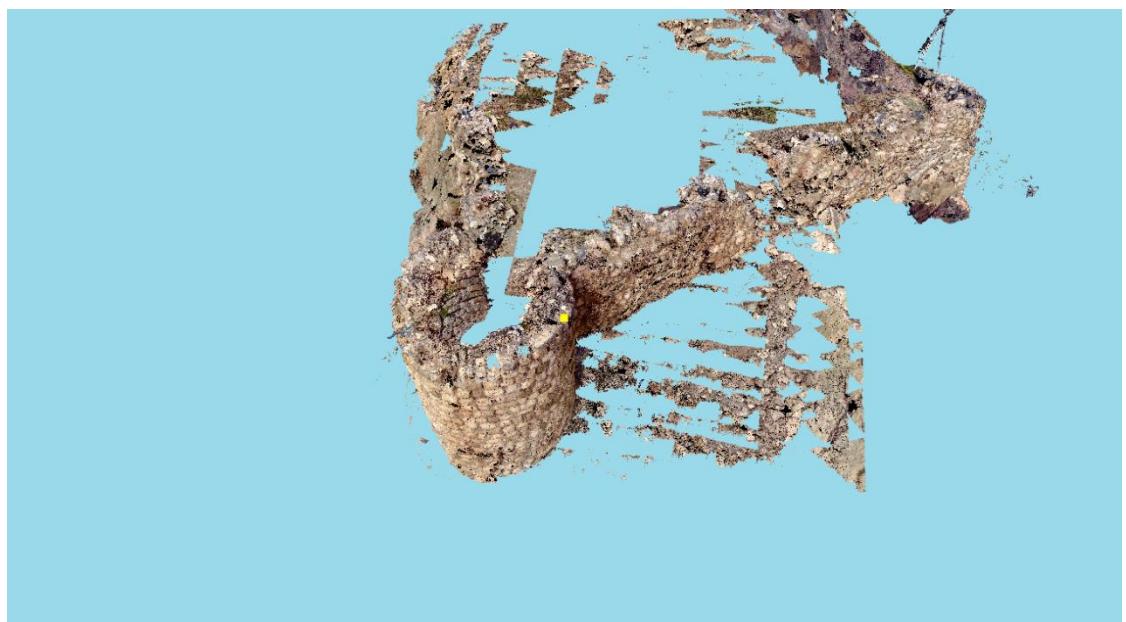
6. kép

## Talajszeparálás eredményei

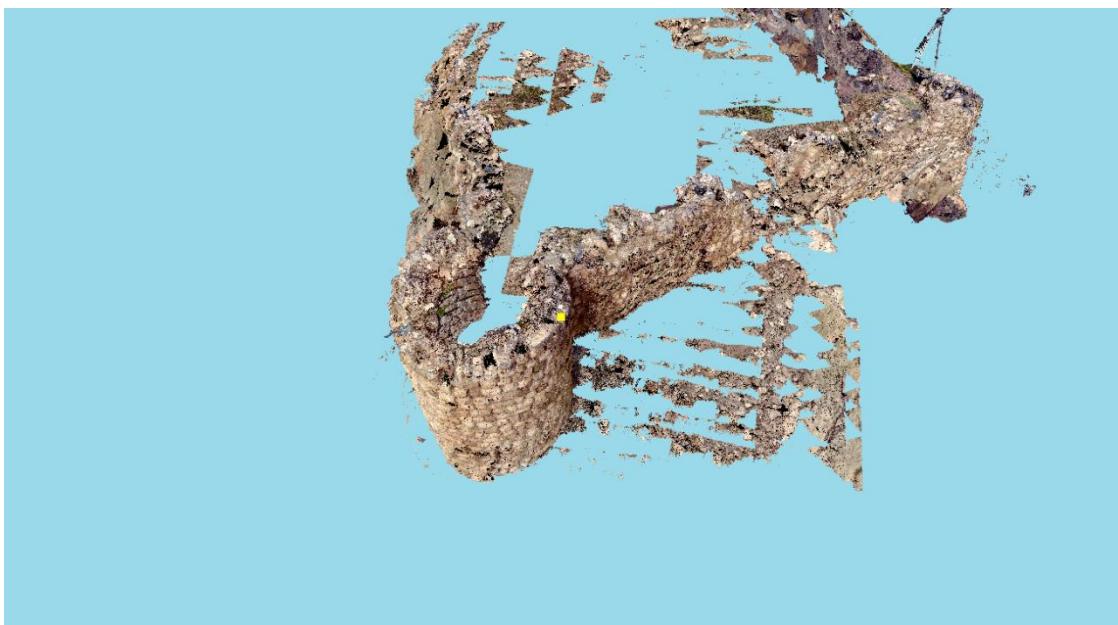
- (PMF-hez hasonló megfontolásokat tartalmazó eljárással készítve, a legtöbb kép a 3. iteráció(lefutás) utáni állapotban készült, bővebben lásd: 4.3.6 fejezet)



7. kép



8. kép



9. kép

- A 10. és 11. kép az első iteráció lefutása után készült, megfigyelhető a több maradvány a talajból (a többihez képest)



10. kép



11. kép



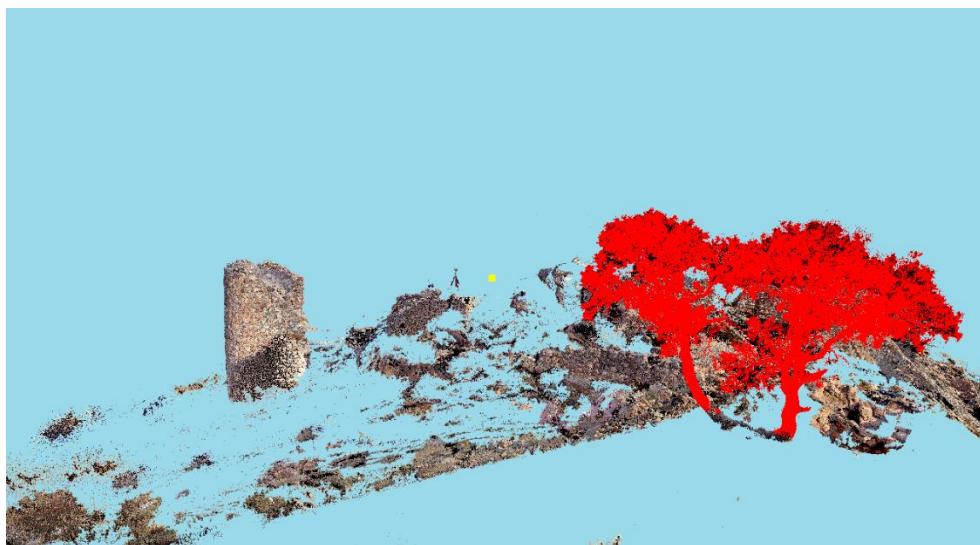
12. kép

## Növényzet félautomata interaktív kinyerésének eredményei

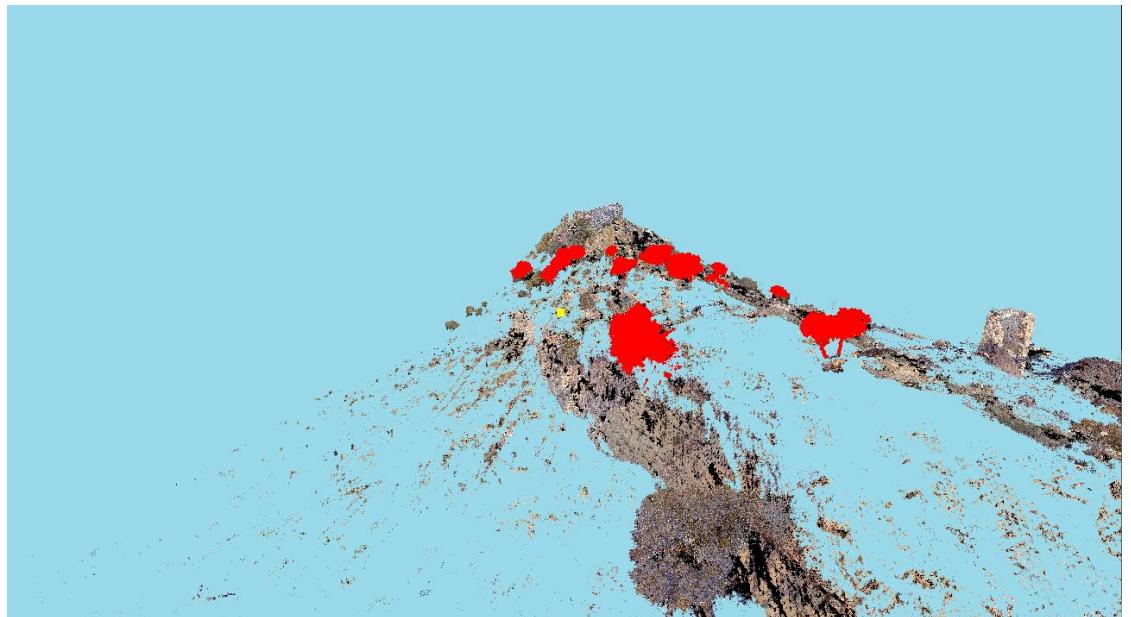
- (Első két kép a kijelölések utáni állapotot tartalmazza – lásd a 4.3.3 fejezet)
  - (A harmadik kép az annotálás információit eltároló fájl betöltése után készült – lásd 4.3.7 fejezet)
  - Mindegyik kép talajszeparáció után készült



13. kép



14. kép



15. kép

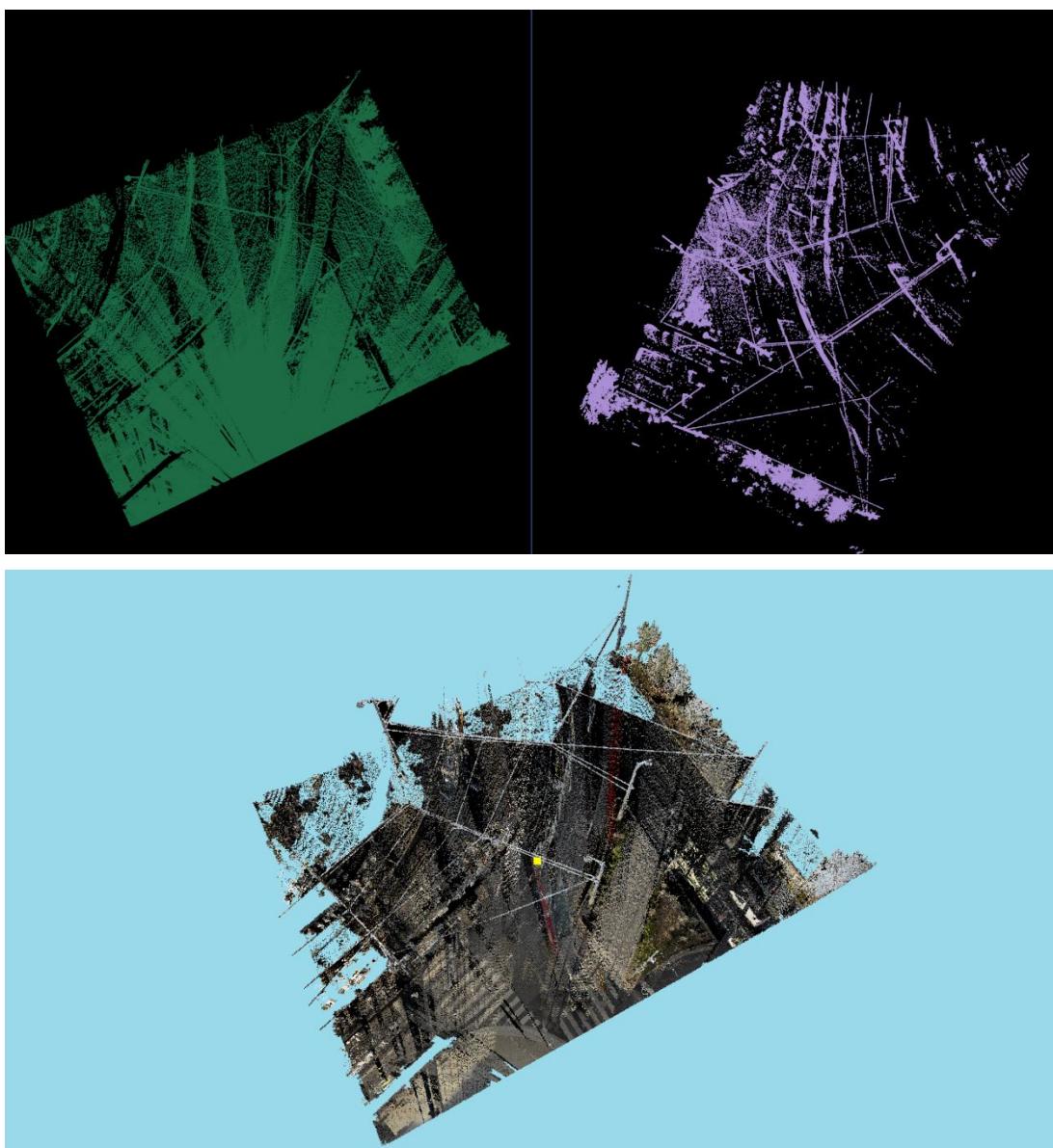
- Az utolsó kép a kijelölt pontok (fák) törlése utáni állapotot mutatja.



16. kép

## **PMF működése (PCL implementáció) nem régészeti vonatkozású pontfelhőn**

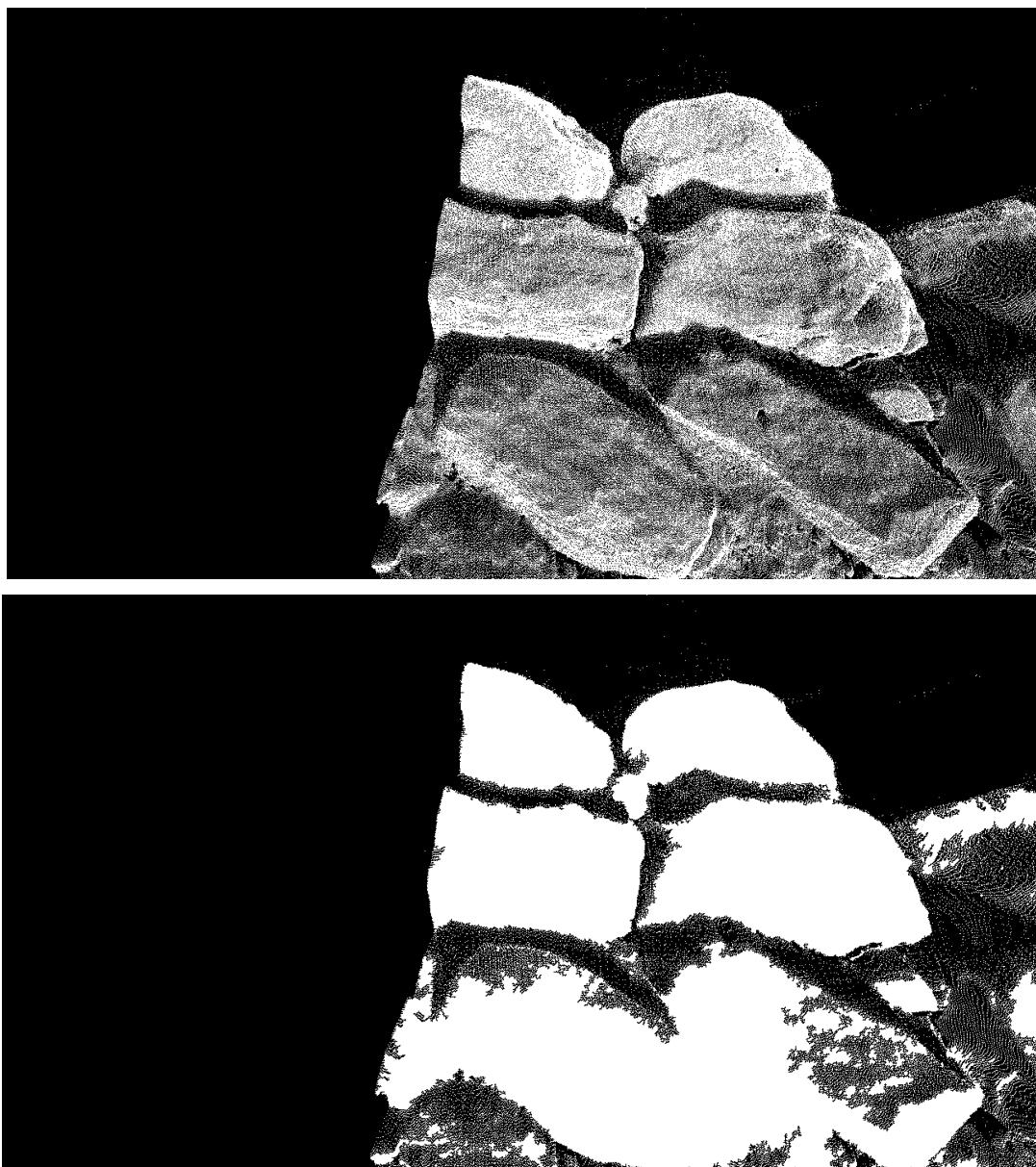
- **bal oldalt látható a talajként elkülönített rész, jobb oldalt pedig az egyéb objektumok**
  - alattuk látható az eredeti pontfelhő (bemenete egy lézerszkennerrel készült felvétel a Gellért téren – ennél jelentősen nagyobbakra a PCL implementációja nem fut le)



# MATLAB eredmények

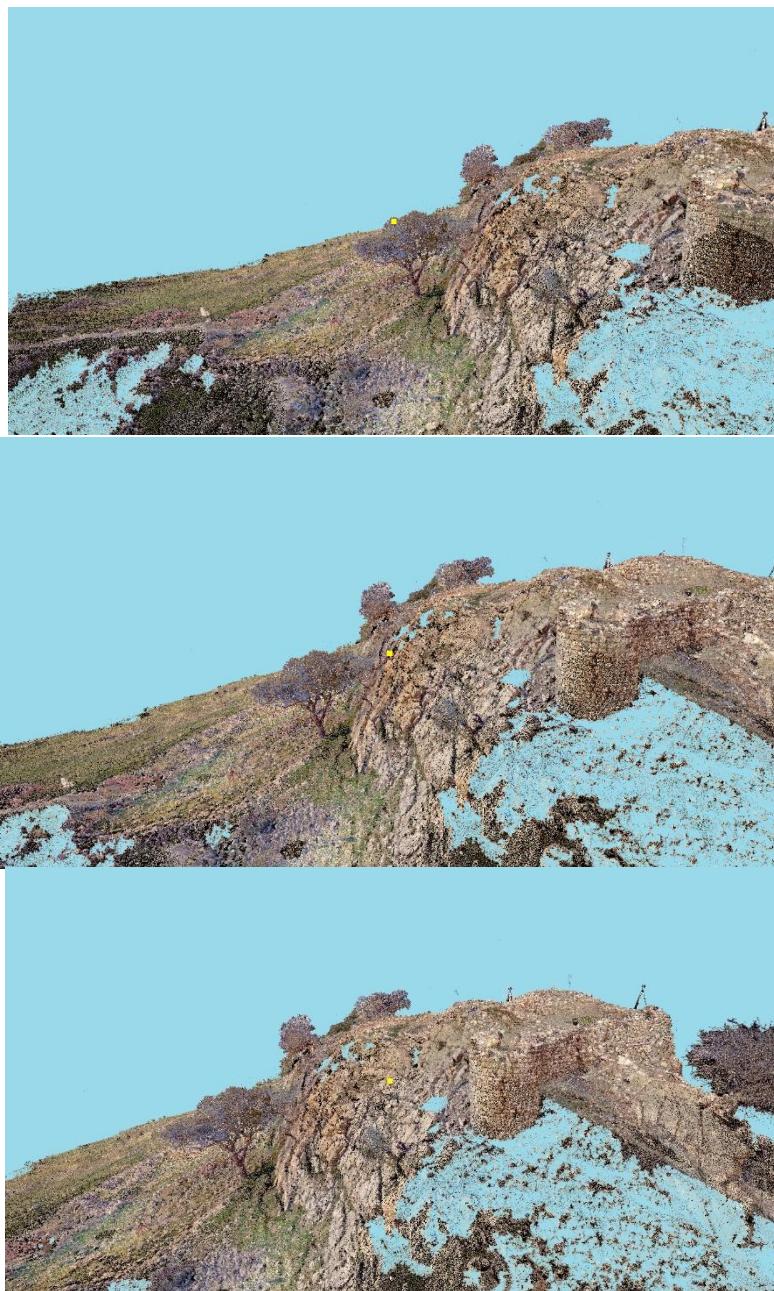
- *graythresh()* és *imbinarize()* függvények kimeneti eredmény
- a bemeneti pontfelhő az ásatás (Dwin vára, Irak) egy kivágott része volt, a Matlab számára a bemeneti képet a szoftver futás közben a **Capture()** függvény meghívásával biztosítja

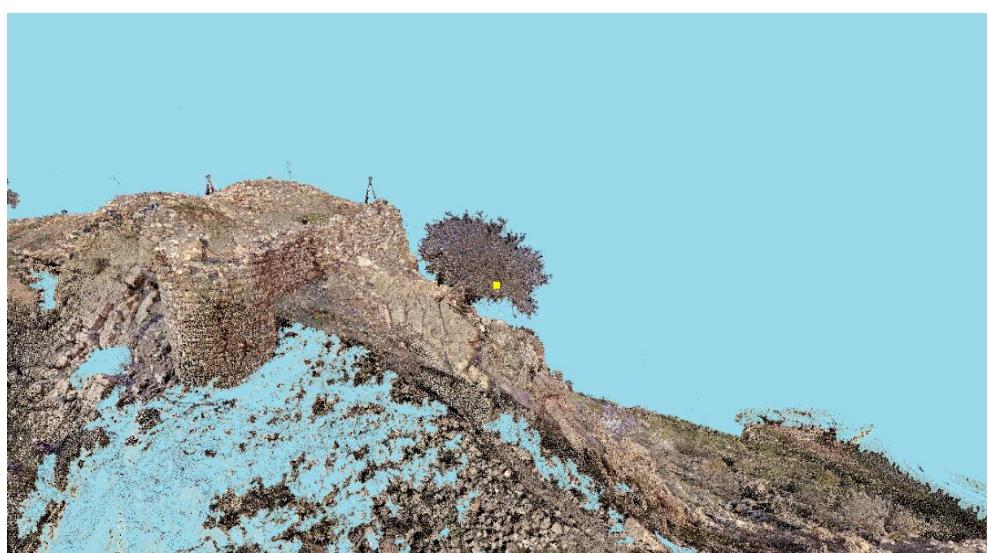
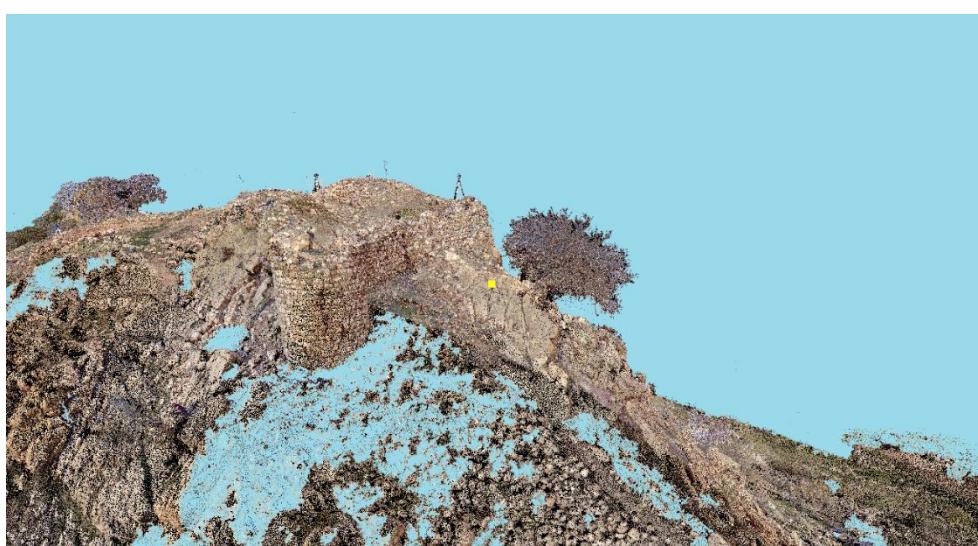
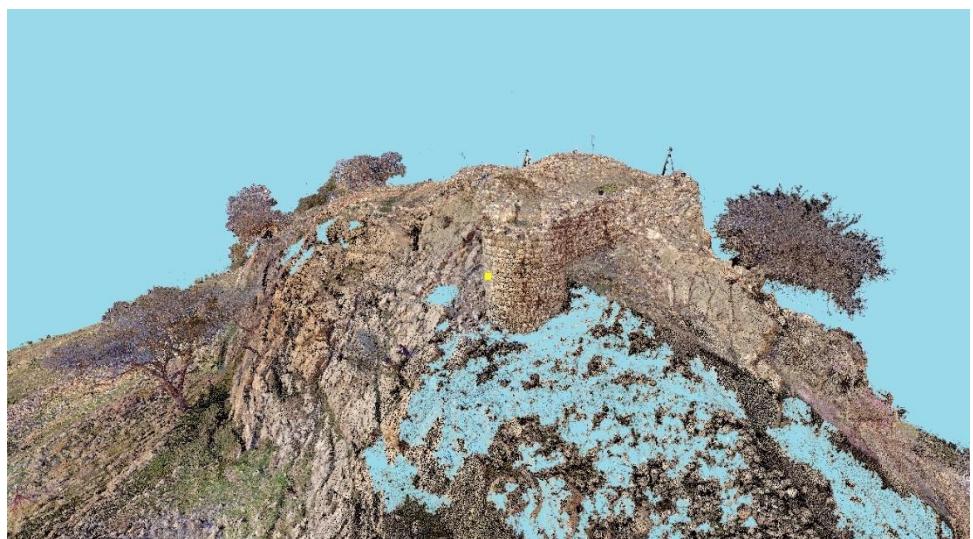
\**imbinarize()* függvény miatt itt csak a fekete háttér megoldható

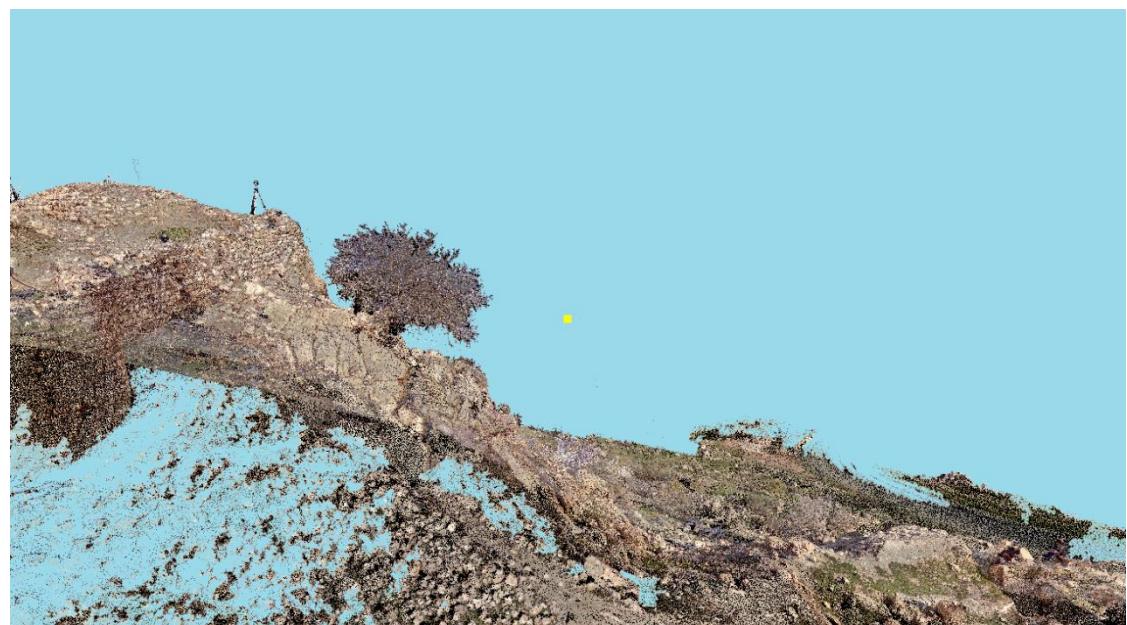
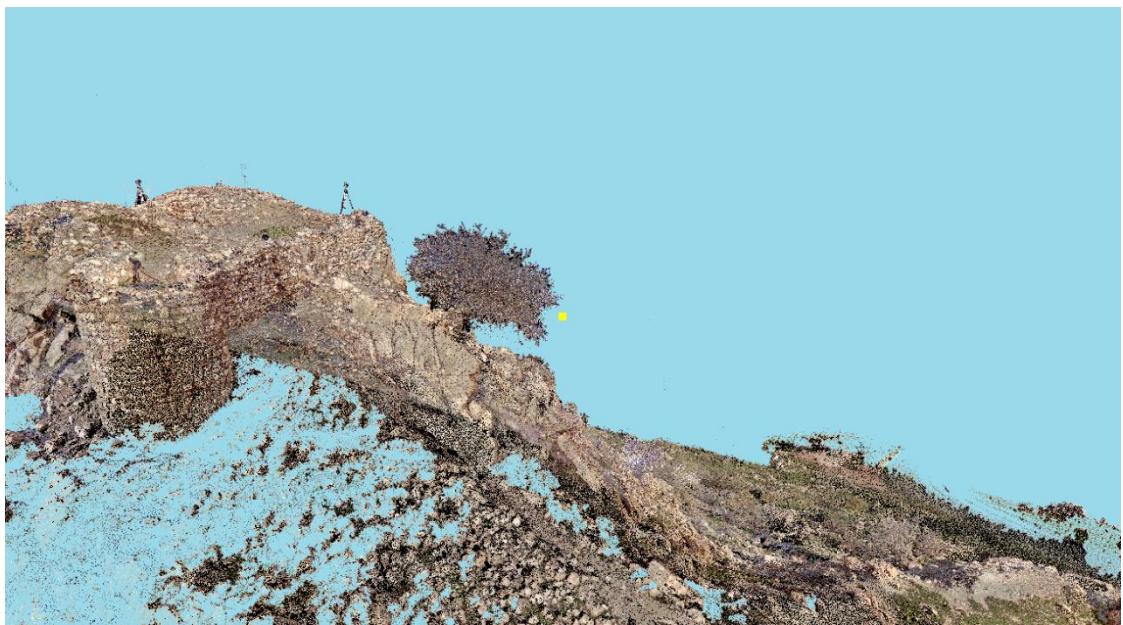


## D. Jellemző képsorozatok a szoftver segítségével történő pontfelhőbeli navigációról

- Itt az egér balról jobbra mozgatásával figyelhetjük meg a kamera fordulását







- Itt azt láthatjuk, ahogy előrefelé(w) végig megyünk a Dwin vára mellett levő kis útszakaszon.

