
**Eötvös Loránd Tudományegyetem**

**Informatikai Kar**

**Információs Rendszerek Tanszék**

# Moduláris feature-alapú 3D pontfelhő regisztráció

**Témavezető:**

Varga Dániel

**Szerző:**

Kovács Levente

Programtervező informatikus BSc.

Budapest, 2020

# Tartalomjegyzék

# *Bevezetés*

Már jó pár éve feszegette érdeklődésem határait az önvezető járművek, illetve az autonóm robotok működései, hogy hogyan ismerik fel a különböző tárgyakat, hogyan közlekednek, hisz napjainkban ezek a területek egyre elterjedtebbek és népszerűbbek. Amikor témát kerestem a szakdolgozatomhoz, egyből megakadt a szemem a 3D pontfelhő rendszerek regisztrálásán a témavezetőm kutatási irányai között. A 3D pontfelhők fontos szerepet töltenek be az imént említett kutatási területekben. A 3D-s pontfelhők 3D-s szkennerek által előállított térbeli pontok halmaza, de előállíthatók különböző objektumok „számítógépes látás” [1] algoritmusokkal is, mint egy háromszög vagy henger. (Persze mi is készíthetünk ilyen pontokat manuálisan). Ezek a pontfelhők lényegében X, Y, Z pontok halmaza egyéb speciális tulajdonságokkal felszerelve az adott pontfelhő típusától függően.  A területen belül egy jelentős probléma a 3D pontfelhők regisztrálása, aminek célja a különböző szögből felvett pontfelhők egyesítése. Vagyis egy olyan térbeli transzformáció keresése, amivel a két pontfelhőt az egyikből a másikba transzformálhatunk, összeigazíthatjuk. Ilyen transzformáció lehet például a térbeli forgatás vagy eltolás.

Ez a probléma már a 90-es évek elejétől foglalkoztatja a kutatókat. Számos elterjedt módszere van, talán a legelterjedtebb az *Iterative Closest Point* (ICP) módszere és ennek különféle továbbfejlesztései, módosításai. Az elmúlt években elkezdtek elterjedni a mély tanulást igénylő és felhasználó regisztrációk. Ezek a megoldások nagyban támaszkodnak az úgynevezett feature-alapú regisztrációs folyamatokra. Ezek a folyamatok több lépésből állnak: (1) kulcspont (olyan számunkra érdekes pontok, amik valamilyen speciális jellemzővel rendelkeznek az objektumon, mint például egy könyv sarka) detektálás, (2) feature vektor számolás (a kulcspontokhoz speciális értékeket tárolunk el annak környezetéből, szomszédságából), (3) párosítások becslése (a feature vektorok alapján összetartozó pontok keresése), (4) párosítások visszautasítása (mivel nem biztos, hogy az összes párosítás megfelelő és ezek negatív hatással lehetnek a transzformációra, így ezeket valamilyen algoritmus alapján ki kell szűrni), (5) transzformáció becslése a kapott értékek alapján (az így kapott párosításokból transzformációs mátrix számítása egyik felhőből a másikba).

Ehhez a problémához készítettem olyan programot, asztali alkalmazást, amellyel képesek vagyunk egy olyan moduláris feature-alapú regisztrációs folyamatot építeni, ami képes két felhőt előfeldolgozni, regisztrálni, és ezek eredményét eltárolni grafikus felhasználói felülettel ellátva.

## A dolgozat felépítése

A dolgozat első felében, a felhasználói dokumentációban az elkészített alkalmazás telepítése, használati lehetőségei lesznek bemutatva, a hozzájuk szükséges elméleti háttérrel kiegészítve. Ebben a részben a felhasználó végig lesz vezetve, hogyan is használhatja megfelelően a programot, hogyan képes betölteni a felhőket, használni az előfeldolgozást, illetve a regisztrálást, valamint, hogy ezek eredményét milyen formában képes eltárolni az alkalmazás jóvoltából. A dokumentáció ezen része segít megérteni, hogy az előfeldolgozás, illetve a különböző regisztrációs folyamatok esetén, milyen paraméterekkel járulhat hozzá a felhasználó azoknak finomításáért annak érdekében, hogy a várt végeredmény jelenjen meg a képernyőn az adott művelet elvégzése után.

A dolgozat második felében, a fejlesztői dokumentációban az alkalmazás megvalósításának módja lesz részletezve. Ki lesz fejtve többek között a fejlesztői környezet, a használt technológiák, eszközök, illetve az egyes osztályok megvalósításának módszerei, köztük a felület megvalósításának módjával. Ezen kívül az alkalmazás teszteléséről is olvashatunk a fejlesztői részben, valamint az esetleges későbbi fejlesztésekről.
A dolgozatot az irodalomjegyzék zárja, ahol a dolgozathoz igénybe vett források, cikkek lesznek megemlítve.

# *Felhasználói dokumentáció*

Ez a fejezet bemutatja a program általános, helyes használatát, célját, illetve használati feltételeit, telepítését. Részletesen kifejti a program által nyújtott lehetőségeket, szolgáltatásokat, felhasználói eseteket.

## A program célja és rövid leírása

A program célja a fentebb említett pontfelhő regisztrálási problémának megoldása és egy megfelelően használható szemléltető eszköz biztosítása a felhasználó számára moduláris módszerekkel. Moduláris a program, tehát a bevezetésben említett 5 pont (kulcspont detektálás, feature vektor számolás, párosítások becslése, párosítások visszautasítása, transzformáció becslése) legtöbbjére több megoldóalgoritmus is rendelkezésünkre áll, kicserélhetőek, paraméterezhetőek. Az alkalmazás egy grafikus felhasználói felületet is biztosít, és megfelelő vizuális eszközökkel bemutatja a regisztráció végeredményét, illetve a folyamatok során kapott értékeket is tudomásunkra hozza, így ez az alkalmazás egy megfelelően használható szemléltető eszközként is funkcionál. A felhők regisztrálásán kívül lehetőségünk van egy előfeldolgozásra is, ahol is a zajos adatokat vagyunk képesek kisimítani, illetve a túl nagy felhőket leritkítani, vagy a kiugró, „nem oda illő” pontokat eltörölni a megadott paraméterek alapján. A program az egyes folyamatait, illetve a közölni kívánt információkat angolul juttatja a felhasználóhoz, hogy az idegennyelvű felhasználók is kényelmesen használhassák nyelvi nehézségekbe ütközések nélkül.

# Rendszerkövetelmények, hardver igény

Operációs rendszer: A szoftver Ubuntu 18.04.3-as típusú operációs rendszeren lett fejlesztve, illetve tesztelve, más operációs rendszeren más telepítési folyamat lehet szükséges, illetve lehetséges, hogy más eredményt is kapunk az egyes műveletek során.

## Hardverigény

A szoftver által használt algoritmusok nagy erőforrásigénye miatt célszerű minél erősebb gépen futtatni az alkalmazást.

Ajánlott memóriaigény: legalább 8 Gb RAM.

Ajánlott CPU magszám: 8.

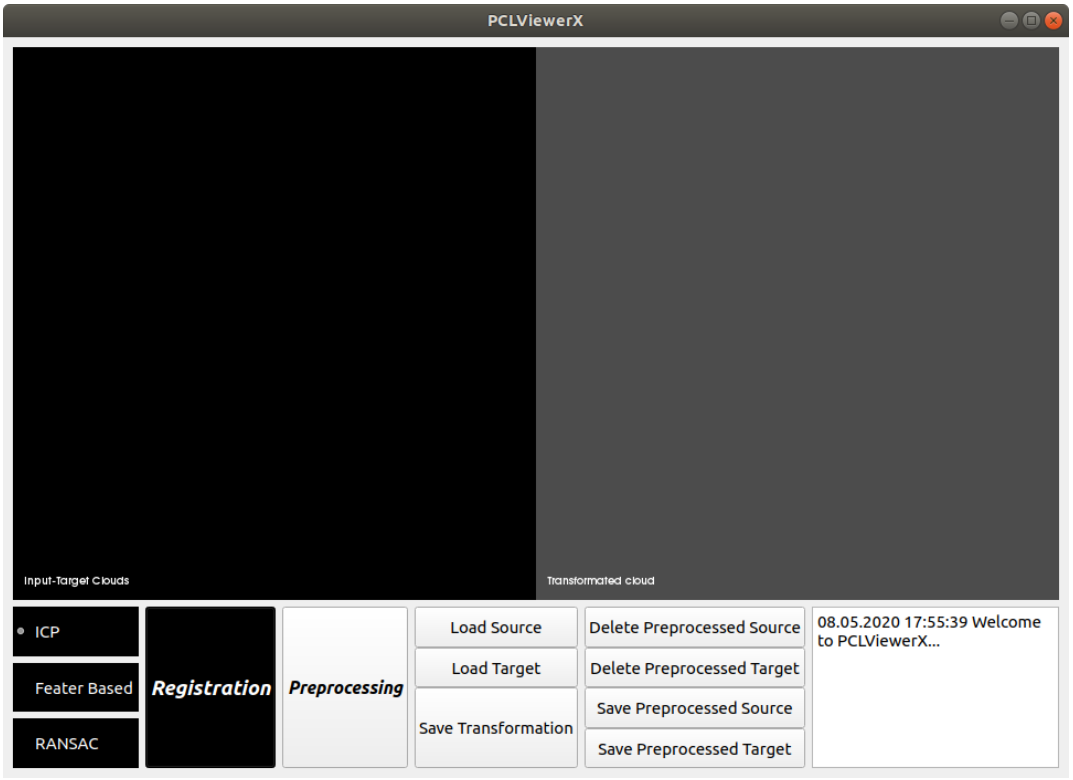
## Telepítés

Az alkalmazás futtatásához szükséges telepíteni a PCL (Point Cloud Library) nyitott projekt 1.8-as előre felépített verzióját, valamint a Qt keretrendszer 5.14-es verzióját. Ezekre a részekre a fejlesztői környezet alábbi fejezeteinél térünk ki:

- *Qt Creator telepítési útmutatója Ubuntu 18.04-es operációs rendszeren*
- *A PCL 1.8-as, előre felépített verziójának telepítése Ubuntu 18.04 operációs rendszerhez*

# A program használata

A telepítést követően, ha a felhasználó futtatja az alkalmazást, akkor a kezdőképernyő fog megjelenni a képernyőjén.




*1. ábra Kezdőképernyő*

## A felhasználó felület részei

### *(1) Felhőket megjelenítő panel*

Az ábrán látható módon az ablaknak nagy részét egy panel tölti ki, ami két részre van osztva és mind a két része üres kezdetben. Ez a panel fog felelni a pontfelhők, illetve a párosítások, kulcspontok megjelenítéséért.

### *(2) Regisztrációs rádiógombok*

A képernyő bal oldalán 3 rádiógomb található, ezek segítségével választhatjuk ki, hogy milyen regisztrálási algoritmust szeretnénk alkalmazni a felhőkre. Ezek mindegyike a következő témakörökben lesznek kifejtve.

### *(3) Mentés/Törlés/Betöltés gombok*

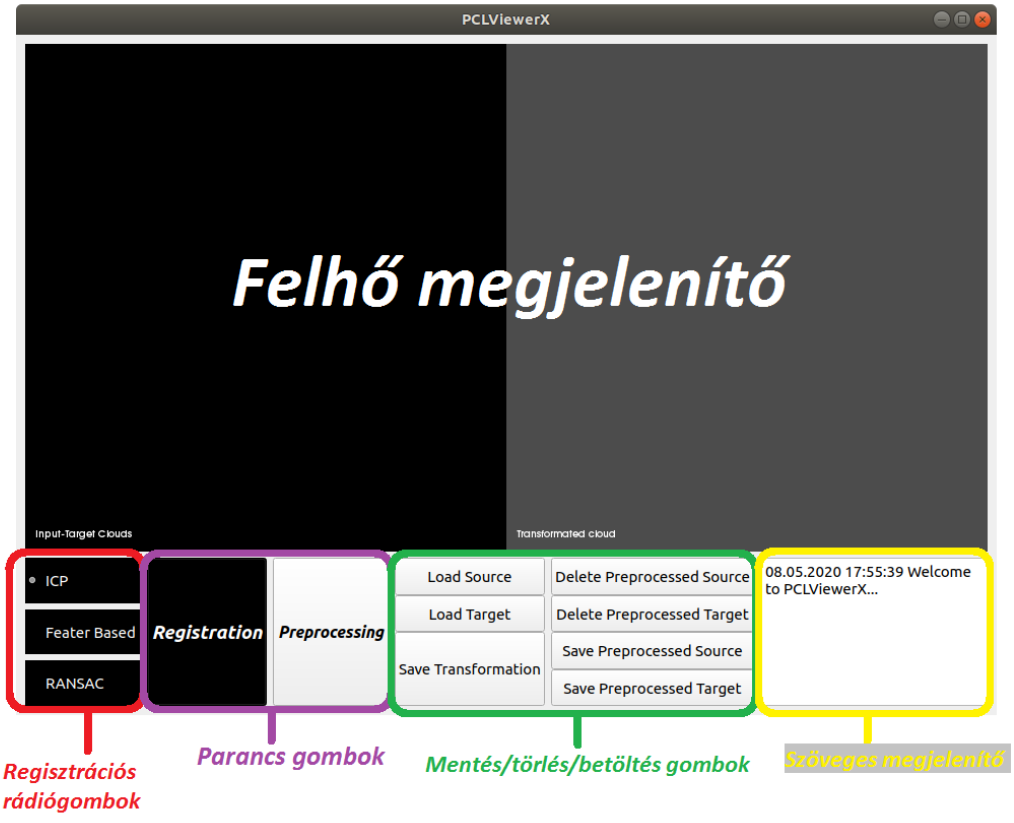
Az ábrán megjelölt módon, a középső részben helyezkednek el a felhők betöltésére, törlésére, illetve a transzformáció mentésére szolgáló funkciógombok.

### *(4) Parancs gombok*

Az ábrán megfelelően megjelölt részben az alkalmazás bal alsó sarkában találhatók a felhőkön végezhető műveleteket indító gombok csoportja.

### *(5) Szöveges megjelenítő*

A jobb alsó sarokban egy szöveges megjelenítő található, ami a szükséges információkat közli a felhasználóval. Minden információt visszanézhet a felhasználó, amit a program indításának kezdetétől kapott.

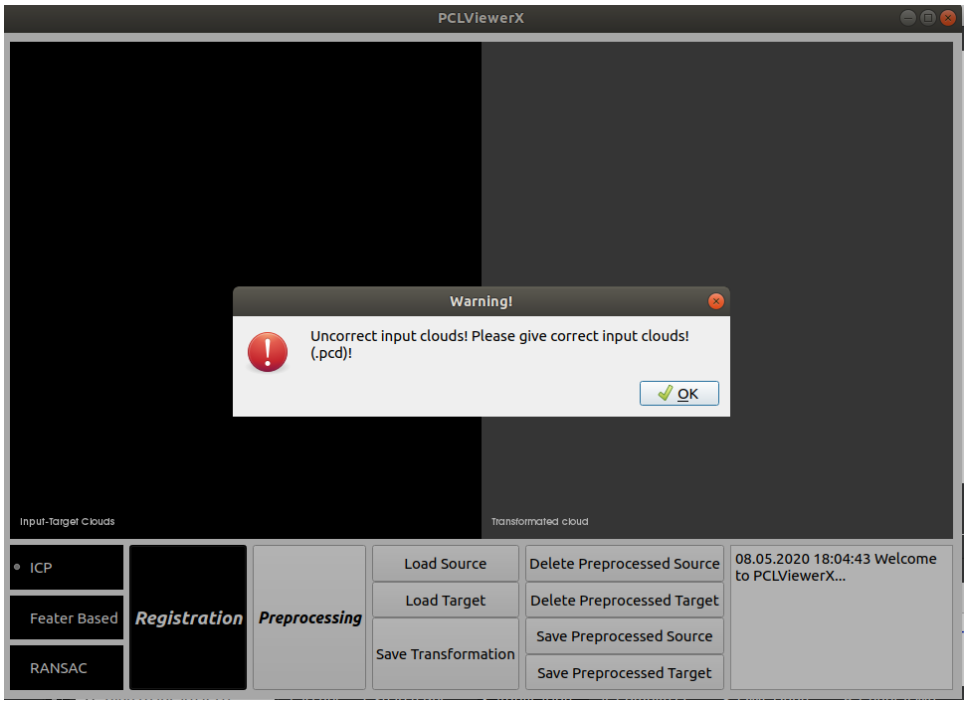



2. *ábra A felhasználó felület részei*

## Első lépések

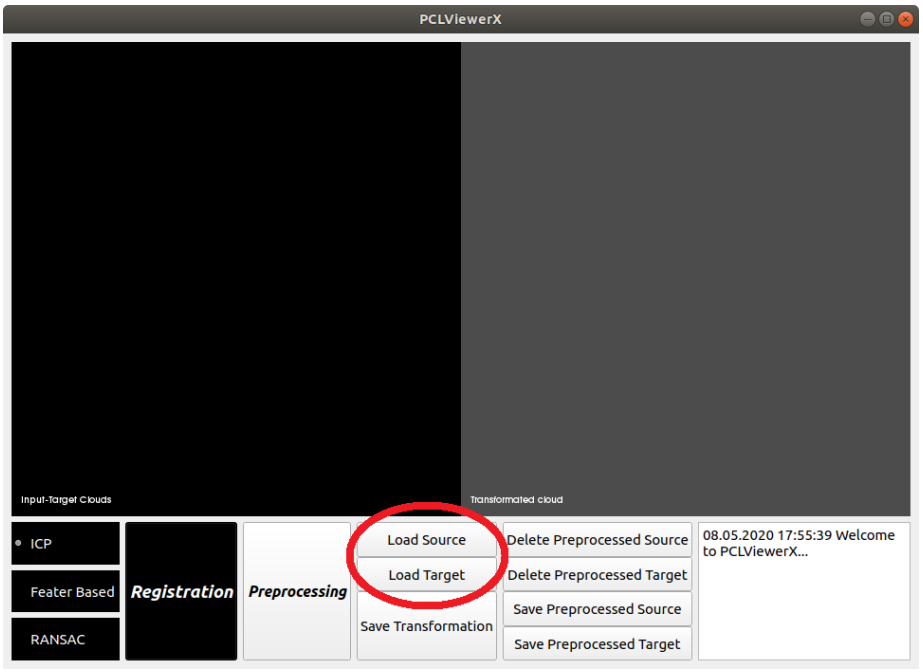
Mielőtt a felhasználó bármelyik műveletet is végre szeretné hajtani, előtte be kell töltenie a kívánt pontfelhőket, amin dolgozni szeretne.

Ha ezt nem tesszük meg, az alábbi hibaüzenetet kapjuk bármilyen olyan funkcióra rákattintva, amihez bemeneti felhő szükséges.



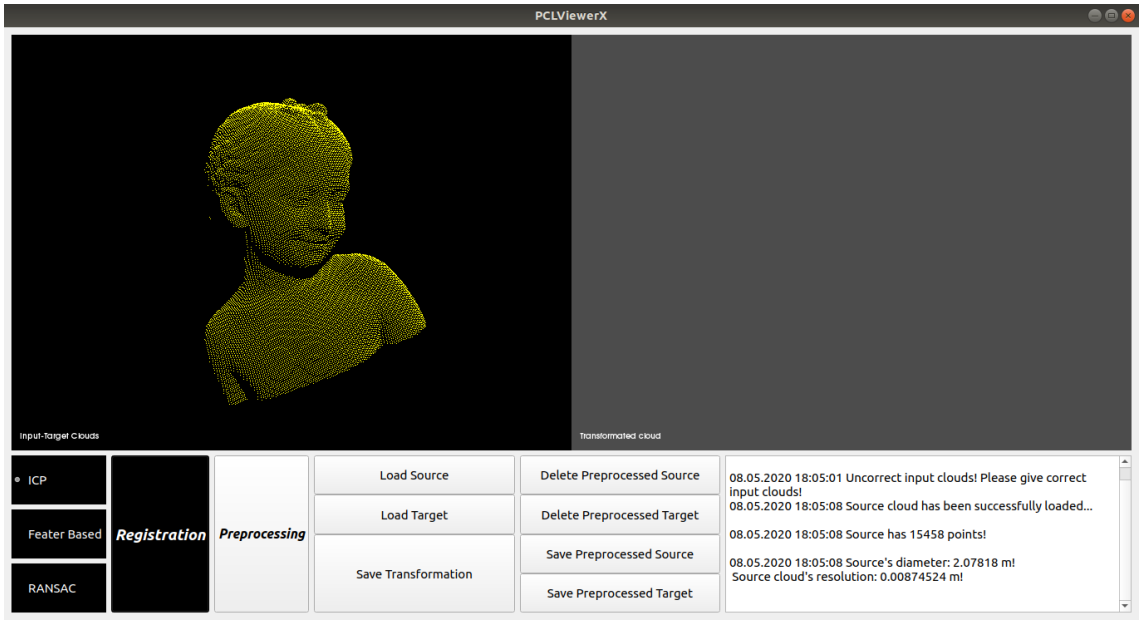

*3. ábra Művelet végzés felhők betöltése előtt*

A felhők betöltését a „*Load Source*”, illetve a „*Load Target*” feliratú nyomógombok (lásd 4. ábra) segítségével teheti meg. Ezekkel a funkciókkal a regisztrációhoz, illetve az előfeldolgozáshoz használt bemeneti felhőket adjuk meg. Regisztráció során a „Source” felhőt fogja a „Target” felhőre illeszteni.

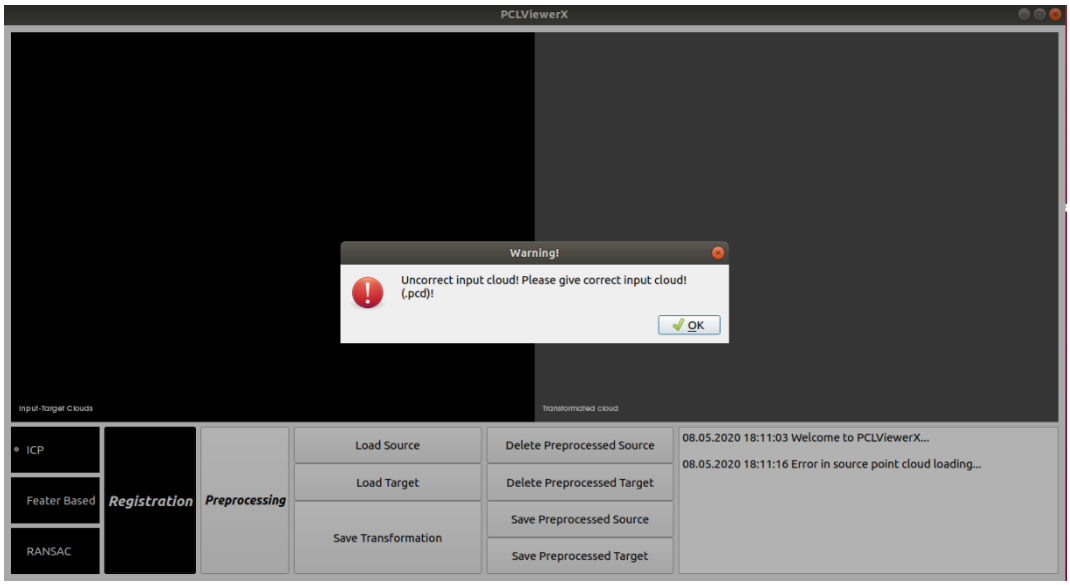



*4. ábra Felhők betöltésére szolgáló funkciógombok*

A „*Load Source*” gomb lenyomásával egy dialógus ablak jelenik meg a képernyőn, ennek segítségével kiválaszthatjuk a fájljaink közül a kívánt bemeneti forrásfelhőt. Rossz felhő, esetleg hibás fájl beolvasása esetén a program hibát jelez nekünk. Helyes beolvasást követően a képernyő bal oldalán jelenik meg a kívánt pontfelhő, és a jobb alsó sarokban lévő panelra kiíródnak a pontfelhő általános adatai.

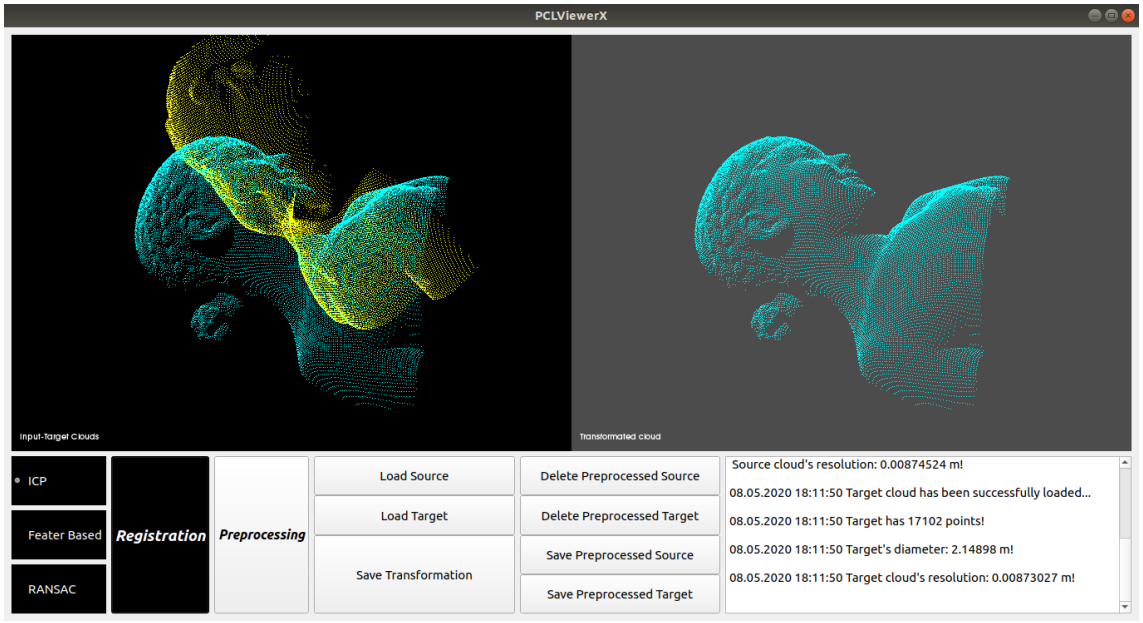



*5. ábra Forrásfelhő betöltése helyes bemenet esetén*




*6. ábra Forrásfelhő betöltése rossz bemenet esetén*

A „*Load Target*” gomb segítségével beolvashatjuk az előbbihez hasonlóan a célfelhőt is, helyes beolvasás esetén a felhőket megjelenítő panel mindkét részében meg fog jelenni a pontfelhő, mivel a regisztrálást követően azt akarjuk vizsgálni, hogy a forrásfelhőn történő transzformálás után mennyire illesztette össze a célfelhőt és a forrást. Továbbá a „*Load Source*” -hoz hasonlóan a célfelhő adatai is megjelennek az üzenetpanelban.




*7. ábra A forrás, illetve a célfelhő helyes betöltése*

Ez a két kezdő lépés természetesen fordított sorrendben is alkalmazható (először a cél-, utána a forrásfelhő). Miután megfelelően betöltöttük a felhőket, megkezdhetjük a munkát.

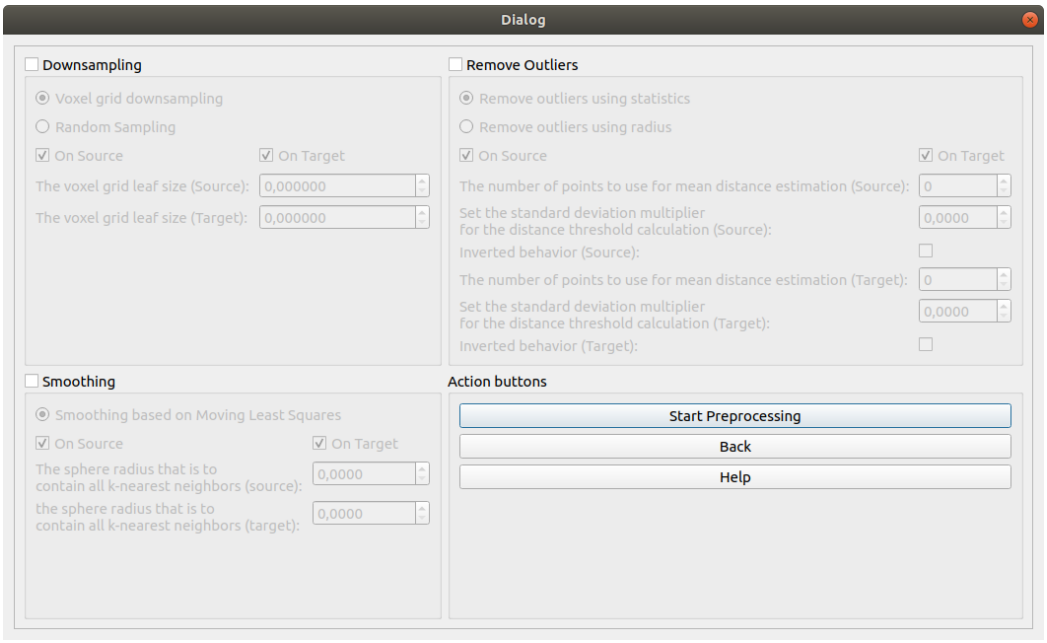
## Előfeldolgozás

### *Miért szükséges az előfeldolgozás? Elméleti bevezető*

Mielőtt hozzáfognánk a pontfelhőink regisztrálásához, hasznos lehet, ha előtte előfeldolgozást végzünk rajtuk. Ez nem kötelező lépés, szimplán csak ajánlott abban az esetben, ha lehetséges, hogy a felhőink nem optimálisak a regisztráláshoz. A valós pontfelhők általában hatalmas adatmennyiséget tartalmaznak a scannerek részletes beolvasása miatt, és ezen adatok közül rengeteg fölösleges a regisztráláshoz. További probléma azontúl, hogy ezáltal pontatlan lehet a regisztrálás is (rossz kulcspontokat talál a túl sok adat miatt), hogy jelentősen terheli a processzort és a memóriát a feldolgozás. Ha megritkítjuk a felhőt, a fölösleges, nem odaillő adatokat eltávolítjuk, valamint korrigáljuk az esetleges zajok által bekövetkezett hibákat (kiugró pontokat), akkor nagyban képesek vagyunk javítani nem csak a sebességén a regisztrációnak, hanem a pontosságán is. Az előfeldolgozásra különböző algoritmusok is rendelkezésünkre állnak a programban.  A következő részekben ezeket mutatjuk be részletesen.

## Előfeldolgozás bemutatása

Ha a felhasználó rámegy a „*Preprocessing*” gombra, akkor megjelenik előtte egy dialógus ablak, ahol némi korlátokkal beállíthatja a felhasználó a kedve szerint az előfeldolgozás részleteit.




*8. ábra Előfeldolgozás dialógus ablaka*

A dialógus ablakban lévő előfeldolgozó folyamatok három fő részre oszthatók: „*Downsampling*”, „*Outliers removal*”, „*Smoothing”*, vagyis csökkentett mintavételezés, kiugró részek eltávolítása, valamint a simítás. A felhasználó kedve szerint kiválaszthatja, hogy a három feldolgozás közül melyiket szeretné végrehajtani. Az alkalmazás megengedi, hogy egyszerre többet is végrehajtsunk ezek közül, ilyenkor nincs más dolgunk, csak bejelölni az adott folyamathoz tartozó jelölőnégyzetet, és kitölteni a paramétermezőket, majd rámenni az „*Start preprocessing*” gombra. Ha meggondoltuk magunkat és mégse szeretnénk előfeldolgozást végezni, akkor a „*Back*” gomb segítségével térhetünk vissza a kezdőképernyőre. Ha némi eligazításra van szükség az előfeldolgozást közben, a „*Help*” gombra kattintva hasznos tanácsokat olvashat a megjelenő ablakon a felhasználó. A következőkben az előfeldolgozás műveleteit részletezzük, paramétereit magyarázzuk:

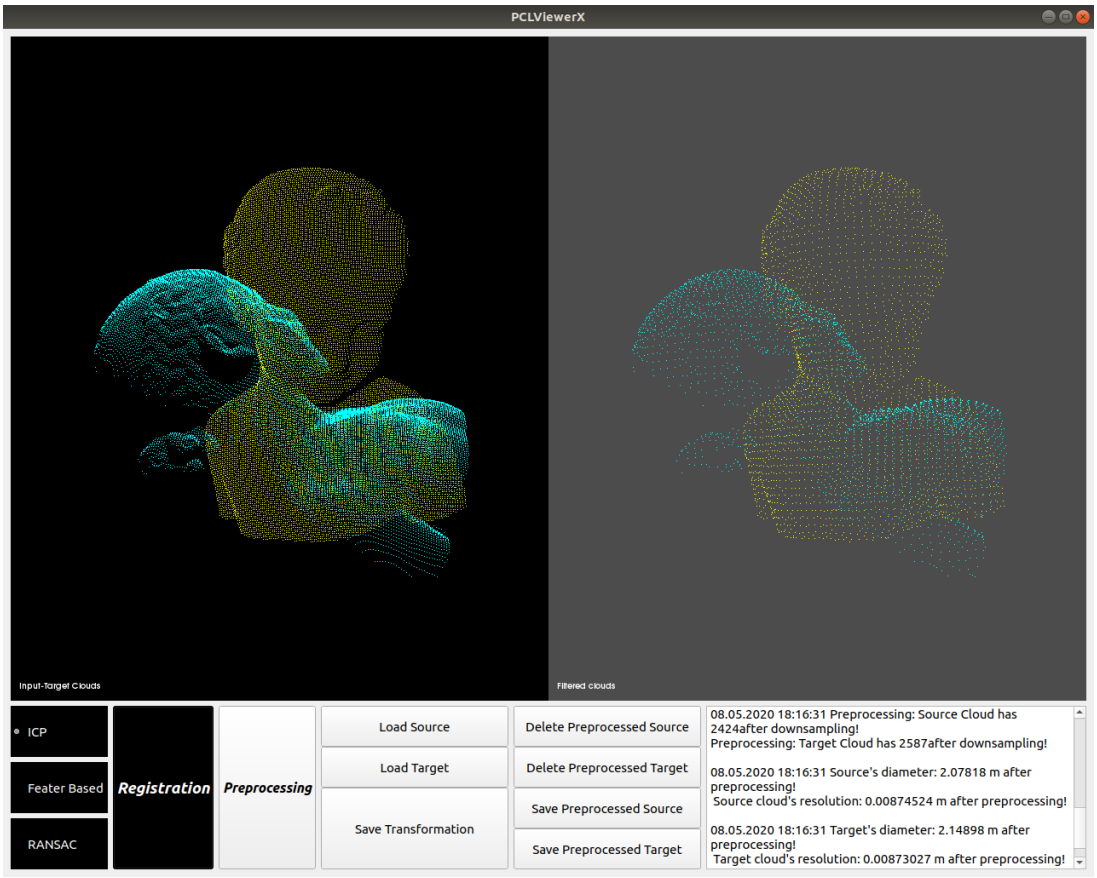
### Csökkentett mintavételezés, másnéven „*Downsampling*”

Mint ahogy fentebb említésre került, a csökkentett mintavételezés azt a célt szolgálja, hogy ritkítsuk a felhőt, ezzel javítva a regisztráció pontosságát, és sebességét. A program erre a célra két algoritmust kínál fel, de ezeken kívül számos egyéb eljárás létezik.

### *(a) Az első módszer a „voxelizált ráccsal való mintavételezés” [2]*

Ez az algoritmus összeállít egy helyi 3D-s rácsot (voxelizált rács) a megadott pontfelhőn keresztül és ennek segítségével végzi a mintavételezést, szűrést, így csökkentve a pontok számát a bemenő adathalmaznak. A voxelizált rácsra érdemes úgy tekinteni, mint 3 dimenziós dobozok halmazára a térben. Ezt követően az algoritmus minden egyes *voxelben* (3D-s dobozban) a benne lévő pontokat a doboz súlypontjával fogja reprezentálni, így ritkítva a felhőt. Ez a megközelítés lassabb, mintha a doboz középpontjait venné az algoritmus, de ezzel a módszerrel sokkal pontosabb lesz az így kapott felület cserébe.

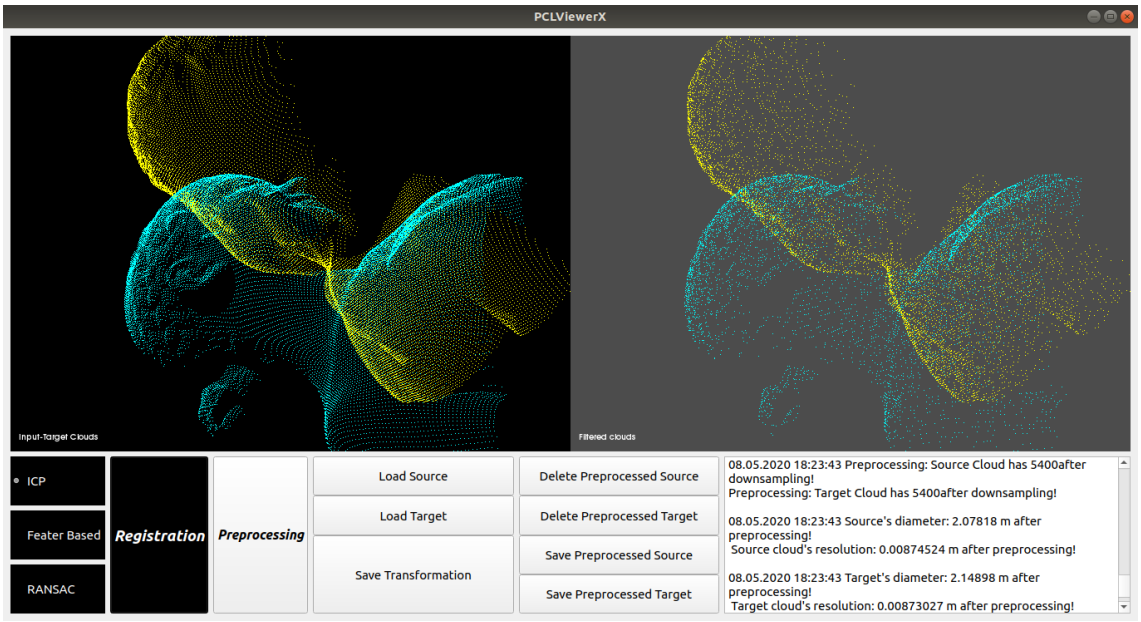
A dialógus ablakon látható paraméterekkel szabályozhatjuk a ritkítást mind a forrásfelhőn, mind pedig a célfelhőn. Az algoritmushoz tartozó paraméterek az egyes voxelek (apró dobozok,kockák) oldal méretét adják meg a forrásfelhőhöz, illetve a célfelhőhöz. Ajánlatos kis számot adni, maximum az átmérőnek a hatodát (a program kiírja az üzenetpanelre a felhő betöltésekor az átmérőt), hisz a túl nagy szám nagyon erős ritkítást eredményezhet, valamint a program hibát fog jelezni, ha ennél nagyobb számot ad meg a felhasználó.




*9. ábra Előfeldolgozás: voxelizált ráccsal történő mintavételezés utáni felhők*

### *(b) A második módszer a „Véletlenszerű mintavételezés” módszer, másnéven „Random Sampling”*

A Random Sampling, vagyis véletlenszerű mintavételezés a nevéből adódóan egységes valószínűséggel véletlenszerű mintavételezést alkalmaz. Alapját a *Jeffrey Scott Vitter* féle „**A”** algoritmus adja a „*Faster Methods for Random Sampling*” [3] publikációjából. Ez az algoritmus szekvenciálisan kiválaszt „*n*” pontot az „N” elemet tartalmazó felhőből. A kiválasztandó pontok száma nem lehet nagyobb tehát a felhő tényleges méreténél, és nagyobbnak kell lennie 0-nál is az algoritmus szerint. A program tovább szigorítja ezt a szabályt, hogy ne menjen a regisztráció kárára a mintavételezés. Így a pontok legalább 5%-át kötelező megtartania a felhasználónak, ennél nagyobb paraméterérték esetén, a folyamat hibajelzést követően leáll és visszakerül a kezdőképernyőre.




*10. ábra Előfeldolgozás: felhők véletlenszerű mintavételezés után*

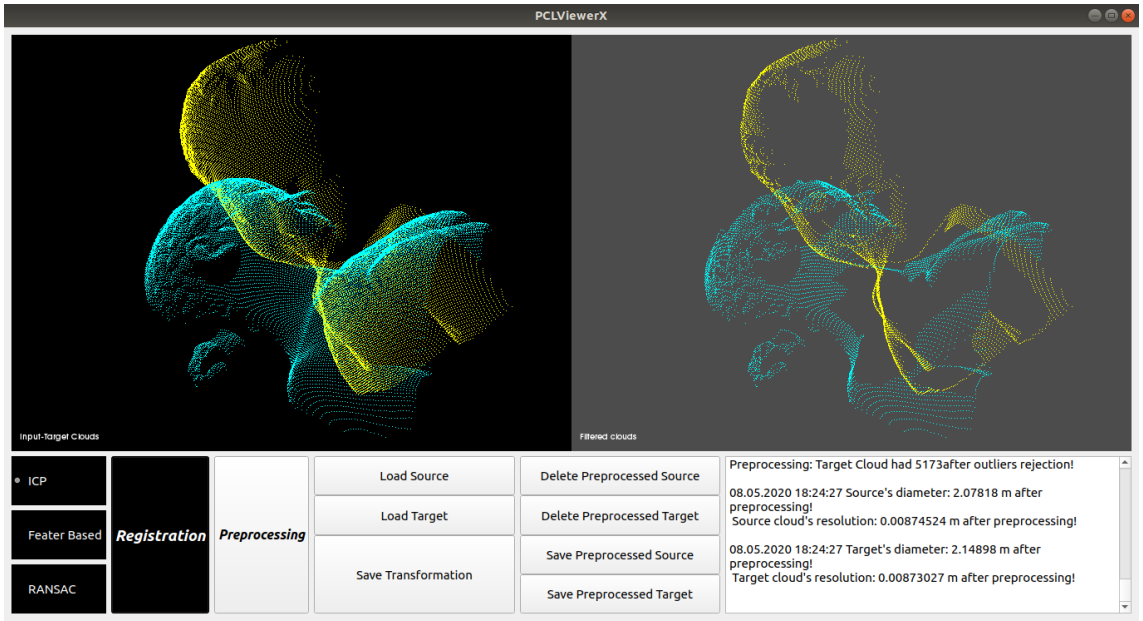
## Kiugró pontok eltávolítása vagy másnéven *„Outliers removal”* [4]

A lézeres szkennelés tipikusan változó sűrűségű felhőket generál. Az ebből adódó mérési hibák kiugró adatokat eredményeznek, amik nem egyértelműen meghatározhatóak, hogy a vizsgált objektum melyik részéhez tartoznak. Ezek a kiugró pontok jelentősen befolyásolhatják az eredményt. Bonyolítják a pontfelhő helyi tulajdonságainak becsléseit, mint például a felületi normák kiszámításait vagy a görbületek változásait, így téves értékekhez vezetve a regisztráció során, esetleg hibát is okozva. Ennek a hibának a kiküszöbölésére két módszert biztosít a program.

### *(a) Az egyik ilyen módszer a „Statisztikailag kiugró részek eltávolítása”* [4]

A folyamat szomszédsági statisztikát használ, hogy kiugró pontokat szűrjön. A megadható paraméterek magyarázatához érdemes az algoritmus működését röviden összefoglalni. Ez az algoritmus kétszer megy végig a teljes bemeneti felhőn. Először minden ponthoz a bemeneti felhőből kiszámolja, hogy átlagosan milyen távolságra van az adott ponttól a ***legközelebbi K darab*** pont. Az egyik bemeneti paraméterrel ezt a K értéket képes szabályozni a felhasználó. Ezt követően az algoritmus ezeknek a kiszámolt értékeknek veszi a szórását és az átlagát, hogy kiszámoljon egy távolsági küszöböt. Ez a küszöb egyenlő lesz az: átlag + szórás * ***szorzó*** képlet eredményével. A másik paraméter, amit a felhasználó megadhat ez a fentebb említett szorzó érték. A következő iterálás

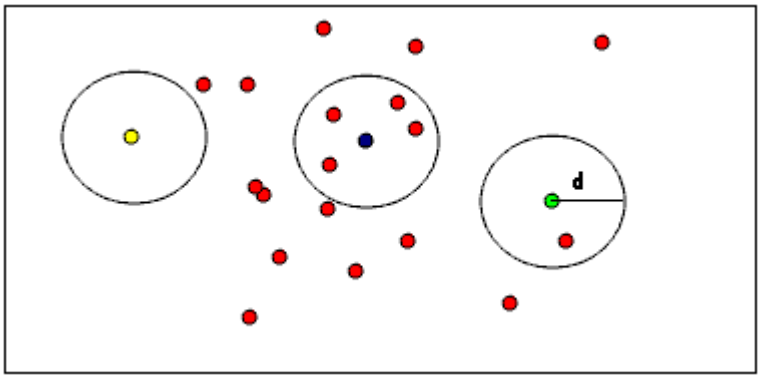
során a pontok be lesznek osztályozva aszerint, hogy valami kiugró pont vagy sem. Ez függ attól, hogy az átlagos szomszédsági távolságuk alatta vagy felette van-e a küszöb értéknek, amit az előző iterálás során kiszámolt az algoritmus. Így tehát ami felette van, kiugró pontnak lesz nyilvánítva és el lesz távolítva a felhőből, ami pedig nem, az értelemszerűen nem lesz. A szorzó növelésével így szigoríthatunk a kritériumon. Az utolsó paraméterként a felhasználó kérheti, hogy az algoritmus által kapott felhő negáltját kapja meg, vagyis minden olyan értéket, amit kiszűrt.



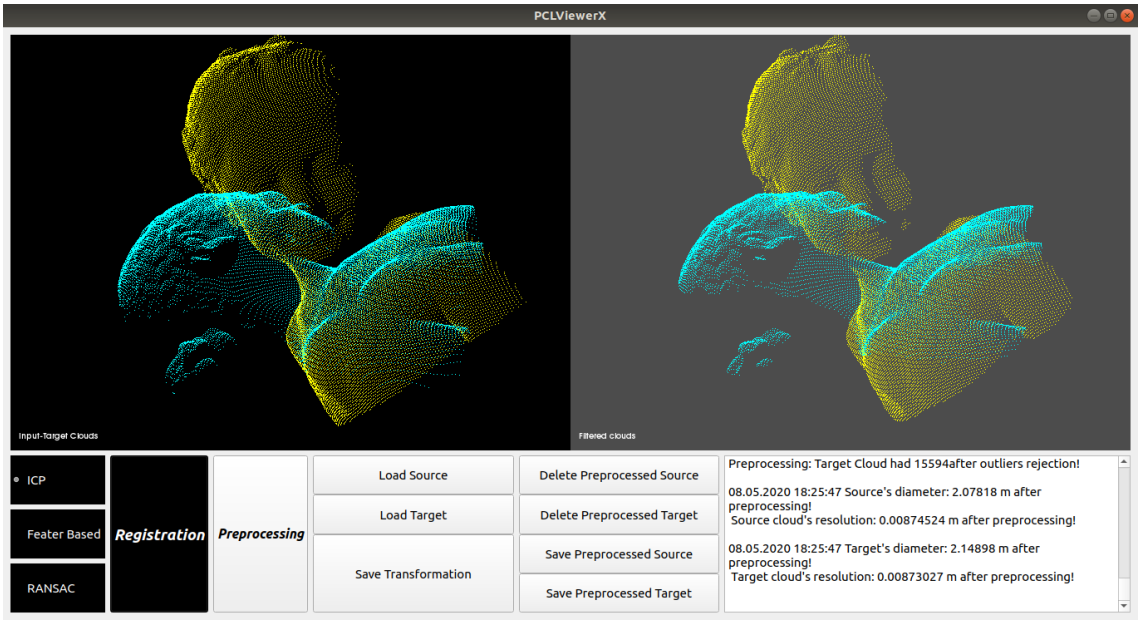

*11. ábra Előfeldolgozás: Statisztikailag kiugró pontok eltávolítása utáni felhők*

### *(b) A másik módszer a „kiugró pontok szűrése sugár alapján” [5]*

A pontok a szomszédok száma alapján lesznek szűrve egy adott sugár szerint. Az algoritmus végigmegy az egész bemeneti felhőn és visszatér a szomszédok számával, amiket egy bizonyos sugárban talált. Ezt a sugarat adhatja meg a felhasználó, finomítva az eltávolítást. Egy pont el lesz távolítva, ha túl kevés szomszédja van ebben a sugárban. Azt, hogy mekkora legyen ez a minimum szomszéd szám, szintén a felhasználó adhatja meg.




*12. ábra „Kiugró" pontok eltávolítása sugár alapján [5]*

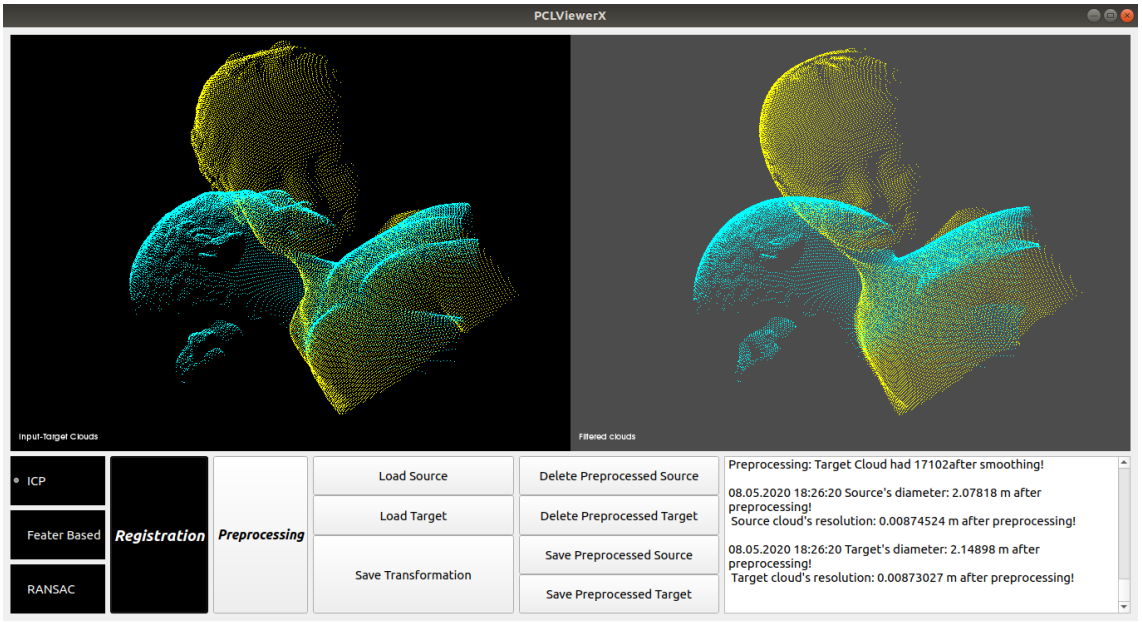



*13. ábra Előfeldolgozás: A sugár alapú kiugró pontok szűrése utáni felhők*

## *(6) Simítás vagy más néven „Smoothing" [6]*

Számos olyan szabálytalan adatot, kiugró részt (amik mérési hibák által keletkeztek) nagyon nehéz eltávolítani statisztikai vizsgálatokkal. Ezekben a helyzetekben egy megoldást nyújthatnak az újbóli mintavételezésen alapuló algoritmusok, amik megkísérlik helyre hozni a hiányos részeket, illetve elsimítani a nem oda illő adatokat. Egy ilyen újra mintavételező algoritmust ajánl fel a program is, ami elsimítja a kiugró pontokat és beleolvassza a felhőbe, alapja pedig a „*Moving Least Squares*" [6] algoritmus. A paraméterrel megadhatjuk, hogy az algoritmus során az egyes pontoknak mekkora sugarában keresse a szomszédokat, amelyeket felhasznál majd az újra mintázáshoz. Minél nagyobb értéket adunk meg, annál távolabbi elemeket leszünk képesek bevonni

a simításba az egyes pontokhoz képest, így növelve a simítás mértékét, viszont ezáltal ez jelentősebb erőforrást is igényel, így célszerű kisebb értékekkel próbálkozni először.




*14. ábra Előfeldolgozás: Simítás utáni felhők*

## Előfeldolgozás eredménye

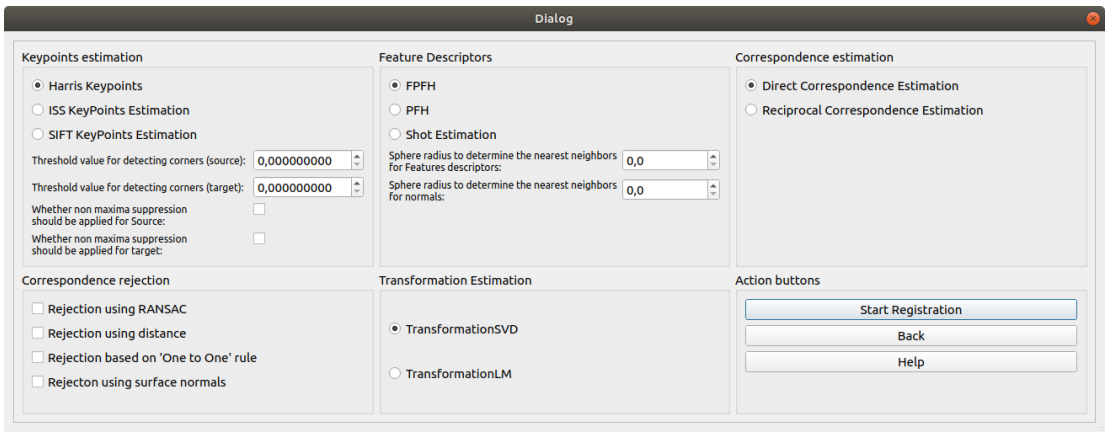
Miután rámentünk a „*Start preprocessing*” gombra, várjuk meg, hogy az alkalmazás lefuttassa az előfeldolgozást és végrehajtsa a műveleteket. Ha ez megtörtént, jelezni fog nekünk a program és a képernyőn megjelenik az előfeldolgozás eredménye. A felhőket megjelenítő panel bal oldalán a régi felhők fognak megjelenni, míg a jobb oldalán pedig az eredményül kapott felhők, így képesek vagyunk összehasonlítani őket. Az üzenet panelen láthatjuk az újonnan kapott felhők adatait: az átmérőjüket (a legtávolabbi pontok közötti távolságot), a felhők felbontását (vagyis, hogy átlagosan milyen távol helyezkednek el a felhőpontok egymástól), valamint, hogy hány pontot tartalmaznak az egyes felhők. Az előfeldolgozás során kapott felhőket a felhasználó kedve szerint eltárolhatja vagy el is vetheti, ha az eredmény nem a vártnak megfelelő lett. Ezeket rendre a „*Save preprocessed source*”, „*Save preprocessed target*”, illetve a „*Delete preprocessed source*”, „*Delete preprocessed target*” gombokkal tehetjük meg.

Ha valamelyik mentés gombra kattint a felhasználó, akkor megjelenik előtte egy dialógusablak, amivel kedve szerint eldöntheti milyen néven és hova szeretné menteni

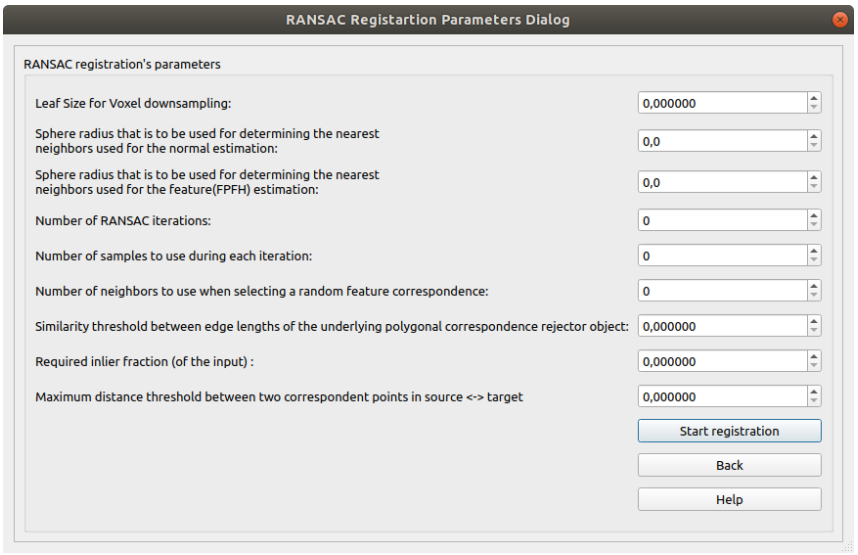
a felhőt. Ha a mentés nem sikerül valami miatt, a program hibát jelez és kiírja a képernyőre ezt nekünk. Ha a mentés sikeres volt, úgyszintén.

## Regisztrálás

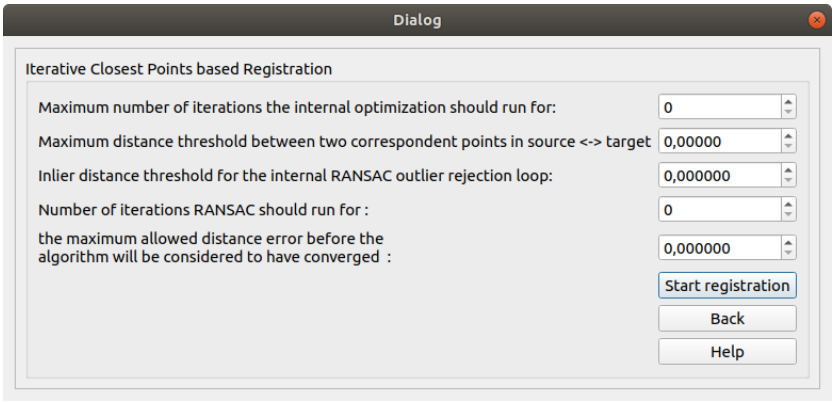
Miután a felhasználó betöltötte a használni kívánt forrás-, illetve célfelhőt is, valamint az esetleges előfeldolgozás véget ért, megkezdheti a pontfelhők regisztrálását. Erre az alkalmazás három módszert is felkínál. Az első módszer **Iterative Closest Points (ICP)** algoritmust használ a regisztráláshoz, a második egy előre felépített **RANSAC** alapú regisztráció, míg a harmadik esetben a felhasználónak lehetősége van egy saját **feature-alapú** regisztráló módszert kiépíteni. Azt, hogy a felhasználó melyik regisztrációs módszert választja, a megadott rádiógombok segítségével választhatja ki (alapértelmezett az ICP alapú regisztráció van kiválasztva). Miután kiválasztottuk, hogy melyik regisztrációs formulát szeretnénk használni, a „*Registration*” gomb lenyomásával megkezdhetjük a paraméterek megadását. A felületről lehetőségünk van visszalépni, illetve egy rövid leírást kérni az adott regisztrációról rendre a „*Back*”, illetve a „*Help*” gombok segítségével. A regisztrációt magát a „*Start Registration*” gomb segítségével indíthatjuk el. A következőkben a fent említett három regisztrációs módszer működtetése lesz kifejtve részleteseben.




*15. ábra Feature-alapú regisztrációhoz tartozó dialógusablak*




*16. ábra RANSAC-alapú regisztrációhoz tartozó dialógusablak*




*17. ábra ICP-alapú regisztrációhoz tartozó dialógusablak*

## Feature-alapú regisztráció [7] [8]

Ahogy a bevezetőben is ki lett fejtve a feature-alapú regisztráció az alábbi fő lépésekből áll:

- ***Kulcspont detektálás,***
- ***Feature vektor számolás,***
- ***Párosítások becslése,***
- ***Párosítások visszautasítása,***
- ***Transzformáció becslése a kapott értékek alapján.***

A program ezt a folyamatláncot valósítja meg különböző módszerekkel a regisztráció során. Ezt a „*Registration*” gombra való kattintással tudjuk elérni. Ekkor megjelenik a felhasználó előtt egy regisztrációs panel, ami öt fő részre van bontva a regisztrációs

lépések szerint. Az egyes lépések mindegyikénél különböző megoldási módszerek közül választhatunk, amikhez más-más paraméterek szükségesek. Ezeknek a módszerek megadására, illetve a paraméterek megadására szolgál a regisztrációs dialógus ablak. A következőkben az egyes lépések módszereit, paramétereit fogjuk áttekinteni.

### *(1) Kulcspont detektálás*

Annak érdekében, hogy ne a bemeneti felhők összes pontját használjuk az algoritmusok során, hanem azoknak csupán egy részhalmazát, különböző kulcspont detektáló algoritmusok állnak rendelkezésünkre. Erre azért van szükségünk, mert ha a teljes pontfelhőt vesszük alapul, akkor minden pont esetén ki kell számolni normákat, feature leírókat, illetve párosításokat, és ez rengeteg erőforrást igényel. A megfelelő kulcspontok kiválasztásával töredékére csökkenthetjük a regisztrálás idejét, miközben a regisztráció minőségén nem rontunk. Számos detektáló algoritmus létezik, van, ami véletlenszerűen választ pontokat, viszont van, ami egységesen valamilyen elméletet követve, törekedve a „fontosabb" pontok megkeresésére valamilyen elv szerint. A feature-alapú regisztrálásunk során a dialógus ablak legelső halmazában a kulcspont detektálás módszerét választhatjuk ki, erre a program három lehetőséget kínál fel.

#### *(a) Kulcspont detektálás Harris módszerrel*

Ez a módszer a Harris féle kulcspont detektálást alkalmazza, ami sarkokon és éleken alapuló algoritmus. A Harris algoritmus a bemenet minden valós pontjához kiszámítja az általa meghatározott intenzitást, és egy bizonyos határérték fogja eldönteni, hogy egy pontnak azt a vizsgálatát, hogy sarok-e (vagy él) vagy sem, folytassuk vagy sem. Ezt a küszöböt a felhasználó adhatja meg. Ha a pont intenzitása alacsonyabb, mint a határérték, akkor nem folytatjuk. Ha nagyobb, akkor az algoritmus keresőfa algoritmussal tovább vizsgálja a pontot, hogy az egy adott sugárban a legnagyobb intenzitású-e. Amikor teljesül a feltétel, tehát a sugárban a legnagyobb intenzitású az adott pont, akkor saroknak lesz nyilvánítva, tehát bekerül a pontfelhő halmazunkba. A felhasználó megadhat még egy paramétert egy jelölőnégyzet formájában, ami eldönti, hogy egyáltalán meg akarja-e vizsgálni a felhőt a fenti szempontból vagy egyszerűen adja vissza az összes pontot a bemeneti felhőben. Ha bejelöli a mezőt, akkor megvizsgálja a pontokat, ha nem, akkor pedig visszaadja a teljes bemeneti felhőt. (Tehát ha a

felhasználó minden bemeneti értéket kulcspontnak akar, érdemes a Harris módszert választani és ezt a paramétert kikapcsolni).

### *(b) Kulcspont detektálás ISS módszerrel [9]*

Ez a módszer az ISS (*Intrinsic Shape Signatures*) kulcspont detektálást használja, ami az egyes pontok „fontosságát” vizsgálja a felhőben a sajátértékek és a szomszédok alapján. Először kovarianciamátrixot számol az egyes pontokhoz, valamint kiszámolja hozzájuk az első, második, illetve harmadik sajátértéket. Ezt követően veszi a második, illetve az első sajátérték hányadosát, majd pedig a harmadik és a második sajátértékét. Ha ezek az értékek egy bizonyos alsó küszöböt átlépnek, akkor tovább folytatja a vizsgálatot.
A hányadosokra vett küszöböket a felhasználó adhatja meg, ezzel szigorítva vagy enyhítve a kritériumon. Ezt követően azt vizsgálja az algoritmus, hogy az előző kritériumnak megfelelt pontoknak elegendő szomszédja van-e egy bizonyos sugáron belül. Ezt a sugarat a felhasználó adja meg. Utolsó lépésben megnézi, hogy lokális maximum vagy minimum-e az adott pont egy gömb sugarán belül. Ennek a gömbnek a sugarát is a felhasználó adhatja meg. Ha mindezen kritériumoknak megfelelt a pont, akkor kulcspontnak nyilvánítja azt az algoritmus.

### *(c) Kulcspont detektálás SIFT módszerrel [10]*

A SIFT (*Scale Invariant Feature Transform*) az egyik leggyakrabban használt kulcspont detektáló algoritmus. Az algoritmus oktávokra bontja a detektálást, és minden oktávhoz keres kulcspontot. Ezekben az oktávokban először leritkítja a felhőt *Voxel rács* segítségével (a fentebb említett módszer alapján), aminek kezdeti paraméterét a felhasználó adhatja meg, majd minden oktávban megduplázza ennek a paraméternek az értékét. Így, ha minél több kulcspontot szeretne, érdemesebb ezt az értéket minél kisebbre állítani. Ezekben az oktávokban kiszámolja a kulcspontokat a ritkítással kapott felhőkből. Minden oktávon belül az algoritmus Gauss-függvényekből kapott skálák közötti különbségeket számol. Azt, hogy hány ilyen skálát számoljon ki, szintén a felhasználó adhatja meg, ezzel pontosítva a detektálást. Ezt követően a felhasználó azt is megadhatja, hogy ezeknél az értékeknél mi legyen az alsó korlát ahhoz, hogy kulcspont lehessen az adott pont. Ha ennek megfelel, valamint az adott pont lokális minimum vagy lokális maximum, akkor az algoritmus kulcsponttá nyilvánítja.

### *(2) Feature vektorok kiszámítása [11]*

Következő lépésben a feature vektorok meghatározását adhatja meg a felhasználó, hogy milyen úton, algoritmus alapján történjen meg a kiszámítása, valamint az ehhez szükséges paramétereket adhatja meg. Ebben a folyamatrészben az történik, hogy a talált kulcspontokhoz a program információkat gyűjt egy adott sugárbéli szomszédsága alapján az eredeti felhőből, és ezeket összegyűjti egy vektorba ahhoz, hogy később ezeket a vektorokat össze tudja hasonlítani. Erre azért van szükség, mert az egyes pontokat nem elég csak a koordinátájuk alapján összehasonlítani, hisz nem csak a környezete, de az objektum pozíciója is változhatott. A körülvevő szomszédok által lefedett felületi geometria levezethető és rögzíthető ezekben a feature vektorokban. Ideális esetben az összepárosítani kívánt feature vektorok között sok a hasonlóság, és azok között a leírók között, amiket nem szeretnénk összepárosítani pedig nagy az eltérés. Egy ideális feature leíró megkülönbözteti magát egy rossztól azáltal, hogy képes megragadni a felület által adott tulajdonságokat, attól függetlenül, hogy a két minta sűrűsége eltérő, vagy hogy a felhőkön esetleges enyhe zaj lép fel. A feature vektorok számításának egy elengedhetetlen, de korántsem elég (mivel nem nyújt elegendő információt a szomszédságról) megelőző lépése, hogy felszíni normákat számoljon az eredeti felhőkhöz, ezzel segítve a jellemzők számítását. A felszíni normák számításában, illetve a feature vektorok számításához is fakereső algoritmust használ az algoritmus annak érdekében, hogy az egyes felhőkben hatékonyan tudjon keresni. Azt, hogy egy adott ponthoz milyen sugárban keressen szomszédokat a számításokhoz, a felhasználó adhatja meg paraméterként. Ezeknek a paramétereknek a megadásán múlik később a regisztráció pontosságának jelentős része, hiszen, ha túl nagy sugarat ad meg a felhasználó, akkor hibás felületi normál értékeket kaphat, hasonlóan a feature vektorok kiszámítása esetén, mivel a feature vektorokhoz rendelt túl nagy keresést meghatározó sugár azt eredményezi, hogy túl sok hasonló tulajdonságú feature leíró lesz, ezzel később rossz megfeleltetéseket hozva létre. A program három módszert kínál fel a feature vektorok meghatározására.

### *(a) Feature vektor számítás PFH módszerrel [12]*

A PFH (Point Feature Histogram) algoritmus célja, hogy kódolja az egyes pontok és azok szomszédságának geometriai tulajdonságait, és ezeket valamilyen értékek formában eltárolja egy többdimenziós tárolóban. Ezeket a tárolókat feature hisztogramoknak nevezzük. A feature hisztogram előállítása úgy történik, hogy az algoritmus megpróbálja a lehető legjobban megragadni az adott mintában szereplő felszínt, figyelembe véve a különböző szomszédok közötti kapcsolatokat. Ez az algoritmus nagyban függ attól, hogy a felszíni normákat mennyire pontosan sikerült kiszámolni a korábbiakban. A PFH algoritmus hatékonyan képes lekezelni a különböző sűrűségű mintákat, illetve a zajszinteket is egyaránt. Hátránya, hogy a szomszédságok közötti összes kapcsolatot kiszámolja hisztogramként, így számítási szempontból ez $O(k^2)$ számítást igényel, ahol $k$ a talált szomszédok száma.

### *(b) Feature vektor számítás FPFH módszerrel [13]*

Az FPFH (Fast Point Feature Histograms) leíró algoritmus egy egyszerűsítése, fejlesztése a PFH formulának. Csökkenti a számítás összetettségét az algoritmusnak, miközben megtartja a PFH előnyeit. Ezt úgy éri el, hogy egy súlyozást használ az egyes hisztogramokra, ennek a súlyozásnak köszönhetően nem vesz minden egyes kapcsolatot a szomszédok között, így lehetővé teszi a sokkal gyorsabb folyamat végrehajtását a valós idejű alkalmazásokhoz. További különbség a PFH-hoz képest, hogy míg a PFH precízen betartja, hogy csak egy bizonyos sugárban keresi a szomszédokat és a kapcsolatokat, addig az FPFH hajlamos a sugáron kívüli kapcsolatokat is belevenni.

### *(c) Feature vektor számítás SHOT módszerrel [14]*

Ez a módszer rendkívül összetett, de cserébe nagyon hatékony a zajokra és a rendezetlenségre. A módszer elméleti alapja a vizsgált terület régiókra való felbontása. A SHOT kiszámító algoritmus lekódolja a szomszédság által lefedett felületet egy speciális gömb struktúrában, aminek sugarát a felhasználó határozhatja meg. Ez a gömb 32 részre van osztva különböző szabályok szerint és minden ilyen tartományhoz az algoritmus kiszámol egy egy-dimenziós helyi hisztogramot. Ha minden hisztogram kiszámolásra kerül, akkor összegyűjti őket egy végső leíróba.

### *(3) Párosítások keresése*

A következő lépés a regisztráció során a kulcspontokhoz kiszámolt feature leírók összepárosítása, annak érdekében, hogy megtaláljuk az átfedést a regisztrálni kívánt felhők között. Az alkalmazás ehhez a lépéshez kétféle kiszámítási módot ajánlj fel. Az első módszer a „*közvetlen párosítások meghatározása*” (*Direct Correspondences*), amikor is a forrásfelhőből kiszámolt minden kulcsponthoz hozzárendel egy célfelhőbeli kulcspontot a feature vektorok alapján. A második módszer a „*kölcsönös párosítások meghatározása*” (*Reciprocal correspondences*). Ekkor az algoritmus párosításokat keres a forrásfelhőből a célba, majd a célból a forrásba és ezeknek a párosításhalmazoknak veszi a metszetét. Tehát a második módszer jóval szigorúbb és kevesebb párosítást eredményez valószínűleg, de értelemszerűen így jóval hatékonyabb párosításokat kaphatunk az időigényért cserébe (viszont fennáll az esélye, hogy túl keveset).

### *(4) Helytelen párosítások elutasítása*

Mivel az előző lépésben a legtöbb esetben a különböző zajok, rossz paraméterek, hibás számítás, vagy a nem teljesen átfedő bemeneti felhők miatt keletkeznek helytelen párosítások, amik ronthatják a regisztráció végeredményét, ezért érdemes ezeket eltávolítani valamely algoritmus alapján. A rossz párosítások elutasítására megírt algoritmusok a megkapott párosítások egy részhalmazát veszik csak. Hogy mik tartoznak ebbe a részhalmazba, azt az egyes algoritmusok speciális kritériumai határozzák meg. Ekkor ezen kritériumok szerint értelemszerűen elutasítják a szerintük helytelennek vélt párosításokat. A program erre négy lehetőséget ajánl fel. A legelső a legelterjedtebb módszer, egy RANSAC alapú algoritmus, a második egy távolság alapú elutasító, a harmadik algoritmus az „Egyhez csak egyet” elvet használja, míg a negyedik a felületi normák újraszámítását használja. A felhasználó bármelyiket kiválaszthatja, akár egyszerre többet is, viszont ekkor fennáll az esély, hogy túl sok párosítást töröl el, sokszor akár helyes párosításokat is, így ügyelni kell a megadott feltételekre, paraméterekre. A következőkben a négy algoritmus elmélete és paraméterezési lehetőségei lesznek kifejtve.

### *(a) RANSAC alapú párosítás elutasító [15]*

A RANSAC egy iteratív folyamat, ami egy matematikai modell paramétereit határozza egy olyan bemeneti adathalmazból, ami „külső" adatokat tartalmaz. A *RANSAC* [14] algoritmus feltételezi, hogy minden adat, amit vizsgálunk, belső és külső adatokból áll. A belső adatokat le lehet írni valamilyen paraméterű matematikai modellel, míg a külső adatok nem illenek bele semmilyen körülmények között egyik modellbe sem.

A RANSAC alapú párosítás elutasító ilyen *Random Sample Consensus* (*RANSAC*) algoritmust használ, hogy megbecsüljön egy transzformációt az adott párosítások egy részhalmazához és eltávolítsa a külső párosításokat. A külső párosításokat az alapján határozza meg, hogy a kiszámolt transzformációt alkalmazza a forrásfelhőre és veszi a pontok közötti euklédeszi távolságot. A felhasználó megadhatja azt, hogy mi legyen az a szuprémum határérték, aminél kisebbnek kell lenni az euklédeszi távolságoknak ahhoz, hogy ne utasítsa el az algoritmus a párosítást, valamint azt is, hogy az algoritmus hány iterálást használjon, tehát hogy hányszor hajtsa végre a fenti folyamatot, amik közül a legjobb transzformációt veszi, és az ezekhez megfeleltetett párosításokat. Ez az egyik legösszetettebb, illetve leghatékonyabb elutasító algoritmus.

### *(b) Távolság alapú párosítás elutasító*

Ez az algoritmus kiszűri az olyan pontpárokat, amelyek közötti távolság nagyobb, mint a felhasználó által megadott határérték.

### *(c) „Egyhez csak egy" algoritmus alapú elutasító*

Az olyan párosítások kiszűrése, amelyek esetén egy forrásponthoz több célfelhőbeli pont is hozzá van rendelve vagy fordítva. Általában a forrásfelhő mindegyik kulcspontja megfeleltetést kap a célfelhőben. Ennél fogva előfordulhat, hogy a célfelhőben egy ponthoz több forráspont is hozzá lett rendelve. Ez az algoritmus ezek közül a párok közül a minimális távolságút választja.

### *(d) Párosítások elutasítása felületi normák kompatibilitása alapján*

Ez az algoritmus a pontok normáinak információit használja, hogy elutasítsa azokat a pontokat, amelyeknek túlságosan eltérő normái vannak. A mi esetünkben akkor utasít el egy párosítást, ha a pontok normái közötti szög nagyobb, mint a felhasználó által

megadott küszöb. Ez az algoritmus képes elutasítani azokat a párosításokat, amik helyesnek tűnnek a pontok közötti távolságok vizsgálata során. A felhasználó megadhatja, hogy az egyes pontok esetén az algoritmus mekkora sugarat használjon a normál értékek kiszámításához (ami alapján keresi a legközelebbi szomszédokat, ezzel meghatározva az általuk lefedett felületet).

### *(5) Transzformáció kiszámítása [16]*

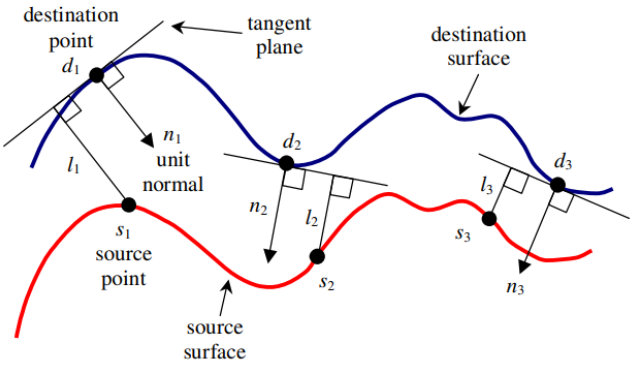
Az utolsó lépés, hogy igazából kiszámoljuk a transzformációt a párosítások alapján, amivel megkaphatjuk a forrásfelhő ráillesztését a célfelhőre. Számos matematikai megközelítés létezik, ami arra szolgál, hogy minimalizálják a pont párok közötti hibaértéket, ami a pont transzformálás során fellép. A transzformálás egy fordításból és egy eltolásból áll össze. A program erre a lépésre kétfajta módszert ajánl fel.

#### *(a) SVD alapú transzformálás*

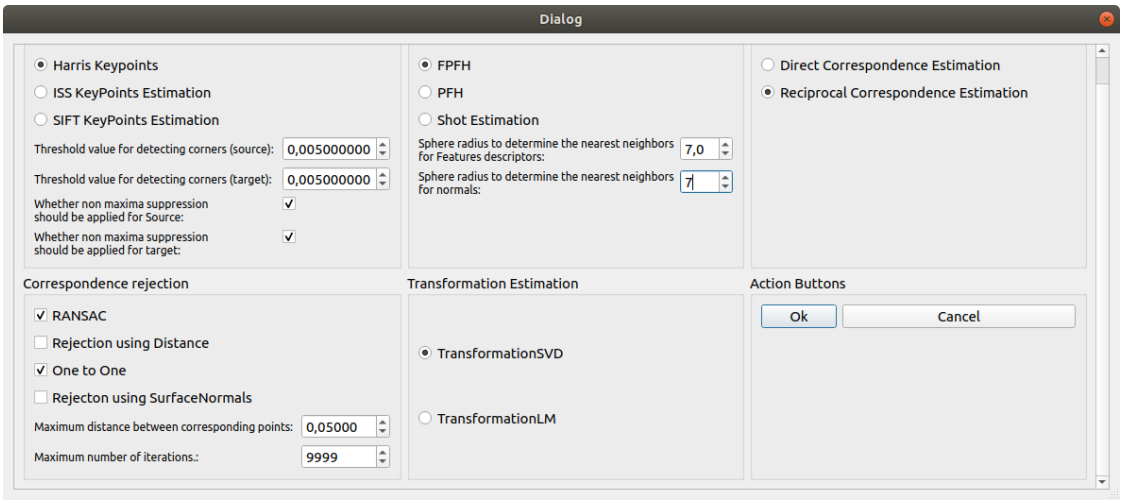
Az *SVD* (*Singular Value Decomposition*) algoritmus szabványos *pont-pont metrikát* használ, célja pedig, hogy minimalizálja az összes párosított pont esetén a forráspontok és a hozzájuk tartozó célpontok közötti távolságokat.

#### *(b) LM alapú transzformálás*

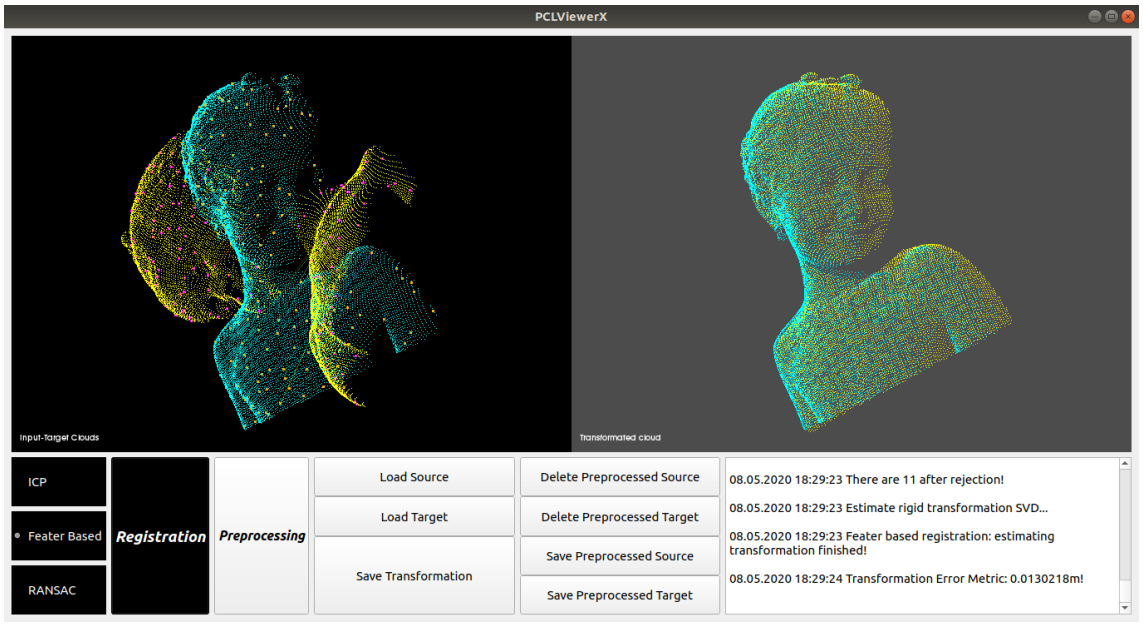
Az LM (Levenberg-Marquadt) féle algoritmus pont-sík hibametrikát használ. A ponttól pontig módszerrel szemben nem a két pont távolságának minimalizálására törekszik, hanem ennek az algoritmusnak a célja, hogy minimalizálja az összes párosított pont esetén a forráspontok és a hozzájuk tartozó célpontok érintő síkja közötti távolság négyzetes összegét.



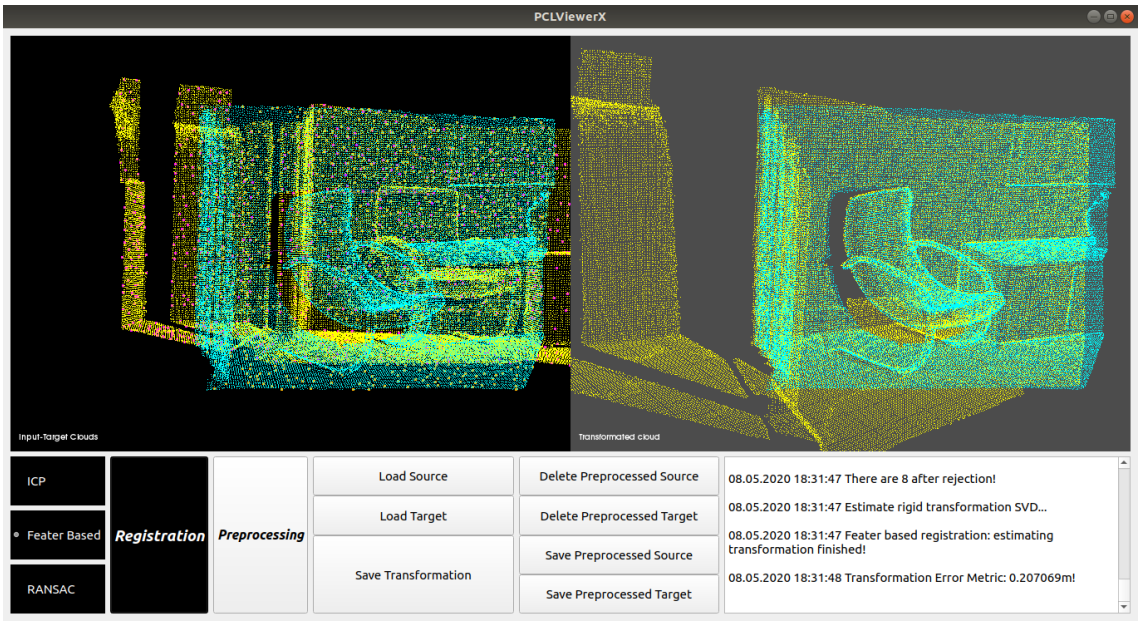

*18. ábra Pont-sík metrika szemléltetése [15]*




*19. ábra Példa egy lehetséges beállításra*




*20. ábra Feature-alapú regisztráció végeredménye normál méretű felhők esetén*

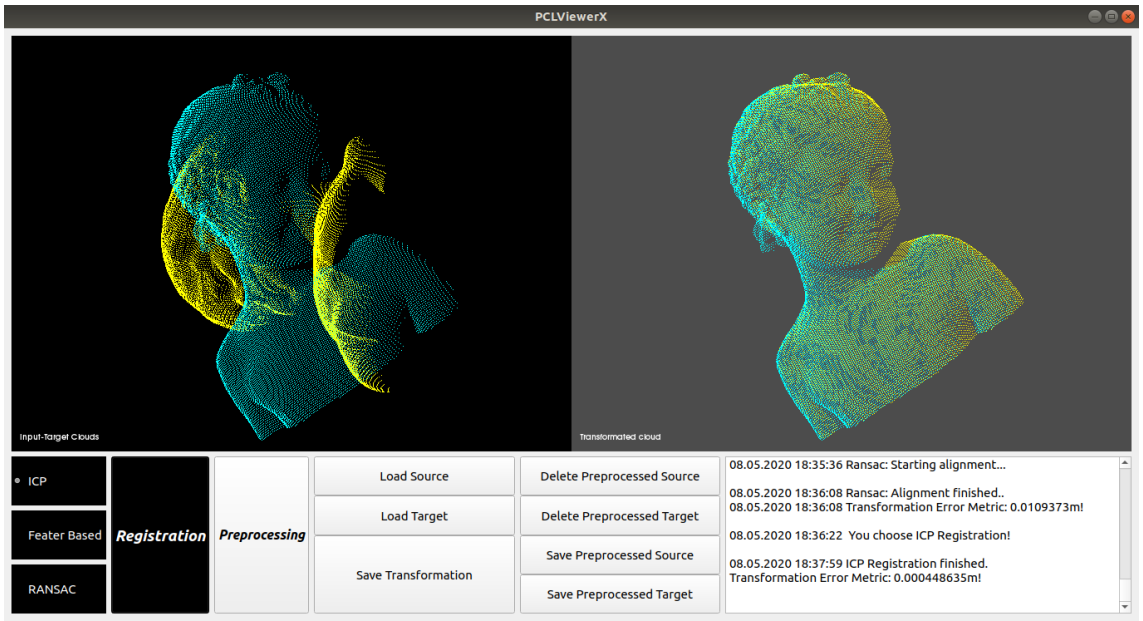



*21. ábra Feature-alapú regisztráció végeredménye nagyméretű felhők esetén*

## Legközelebbi pontok iteratív regisztrációja, vagyis az ICP algoritmus [8] [17]

A feature-alapú regisztrációkkal szemben, az **Iterative Closest Point (ICP)** alapú regiszráló algoritmusok nem hasonlítják össze a kulcspontok alapján megalkotott feature leírókat annak érdekében, hogy párosításokat hozzanak létre a forrás- és célfelhőből. Ehelyett először legközelebbi pontokat keresnek úgy, hogy minden forrásbeli ponthoz megkeresik a célfelhőben lévő hozzá legközelebbi pontot, majd ezeket a párokat rendezik össze (egy transzformáció segítségével). Ezt a két lépést ismétlik addig, amíg az eredmény hibametrikája nem konvergál, vagy pedig el nem ér egy másik terminálási kritériumot (maximális iterálási szám, határérték a hibametrikához), így finomítva iteratív módon a forrásfelhő célhoz való igazítását. Optimális esetben a rendezés konvergál a globális minimumhoz bizonyos feltételek mellett (olyan feltételek minthogy a rendezni kívánt felhők tökéletesen átfedik egymást). Az iteratív regisztrálás legfőbb hátránya, hogy az algoritmusok megragadnak egy helyi minimumban, ezáltal nem konvergálva a globális minimum felé, ha bizonyos feltételezések nem teljesülnek. Tehát ha például a pontfelhők csak részben fedik egymást. Ebben az esetben a párosítások negatív hatással lehetnek a regisztráció eredményére. Ilyenkor a feature-alapú regisztrációnál is említett elutasító folyamatokat alkalmazhatunk ezek kiszűrésére, ezzel segítve a konvergálást. Mivel az ICP egy iteratív módszer és a teljes bemeneti felhőkön dolgozik, ezért ennél a regisztrációs

algoritmusnál különösen ajánlott egy előfeldolgozási módszer, mert ellenkező esetben rendkívül idő- és erőforrásigényes lehet a folyamat. Az ICP algoritmussal való regisztráláshoz a program három paramétert ajánl fel, amivel finomítani, pontosítani képes a felhasználó az eredeti algoritmust. Első paraméterként megadható az ICP iterálásnak maximum száma, tehát hogy legfeljebb hány illesztést próbáljon meg, ha nem konvergál a globális minimumhoz a hiba-metrika. A globális minimumot is meghatározhatja a felhasználó. Ezt a hibaküszöböt elérve terminál az algoritmus. Továbbá a felhasználó megadhatja a helytelen párosítások elutasításához használt RANSAC alapú algoritmus paramétereit, vagyis annak maximum iterálási számát, illetve a határértéket (lásd feature-alapú regisztrálás esetén a párosítás elutasító algoritmusoknál). Ha ezeket az értékeket 0-n hagyja a felhasználó, akkor a program ezeket figyelmen kívül hagyva automatikusan az alapértelmezett értékekkel számol majd (kivéve a konvergálási határérték esetén, mert ott lehet 0 az érték).




*22. ábra ICP alapú regisztráció végeredménye*

## RANSAC (RANdom SAmple Consensus) alapú regisztráció [15] [18]

Ez egy előre megírt feature-alapú rendező algoritmus, ami a fentebb említett RANSAC [14] modellt használja a regisztrációhoz. A kulcspontokra nem speciális kulcspont detektáló algoritmusokat használ a mintavételezéshez, hanem egy egyszerű Voxel rácsot. A feature leírókhoz FPFH alapú algoritmust használ a folyamat. A transzformáció

becslés és a rendezés egy előre elutasító (*prerejective*) RANSAC algoritmus segítségével történik. Ez az osztály beilleszt egy egyszerű, mégis hatékony „előzetes visszautasítási" lépést a RANSAC által biztosított iterálási ciklusába, a valószínűleg téves hipotézisek elkerülése érdekében. Ezt a feature leírók összehasonlításával végzi az algoritmus.
A modellek hatékony regisztrálása érdekében ez a folyamat nem az illesztési hibát minimalizálja, ahogy láttuk például az ICP algoritmusnál, hanem inkább a belső párosítások arányát próbálja maximalizálni (tehát, hogy minél több olyan párosítás legyen, amely esetén a megadott küszöbön belül van a hibaérték).

*A felhasználó különböző paraméterekkel járulhat hozzá a regisztráció finomításához [19]:*

- *Megadhatja a felhasznált minták számát, amit egy iteráció során felhasznál az algoritmus: ez a forrás és a cél közötti mintavételi pontok számát jelöli a párosításokhoz. Egy iterálás során ennyi feature leírót fog megvizsgálni. Legalább három pont szükséges ehhez.*
- *Az iterálás során kiválasztott minták mindegyikén végigmegy, és veszi az adott leíróhoz az N darab, hozzá legjobban hasonlító feature leírót, majd ezek közül választ véletlenül egyet. A véletlenszerűség fokozását ennek az N értéknek a növelésével érhetjük el. Ezt adhatja meg a felhasználó, módosítva az egyes RANSAC iterációk hasznosságát.*
- *Lehetőségünk van egy hasonlósági küszöb beállítására a [0,1) intervallumban, amivel az algoritmus előzetes elutasításának mértékét állíthatjuk, ahol 1 esetén maximálisan elutasító lesz az algoritmus, 0 esetén pedig ez az előre elutasító lépés úgymond ki lesz kapcsolva.*
- *Sok gyakorlati esetben a megfigyelt tárgy, amit igazítani akarunk a célfelhőre, nem látszik teljesen a célfelhőben különböző zajok vagy mérési hibák miatt. Az ilyen esetekben nekünk engedélyeznünk kell olyan hipotézisek használatát a rendezések során, amelyek nem illesztik rá minden pontját a forrásnak a célfelhőre. Megadhatunk egy olyan paramétert, ami azt szabályozza, hogy mikor fogadjunk el egy hipotézist. Ez a határérték azt mondja meg, hogy mi legyen a helyesen rendezett pontok minimális aránya a teljes forrásfelhőhöz képest ahhoz, hogy elfogadja az algoritmus a hipotézist.*

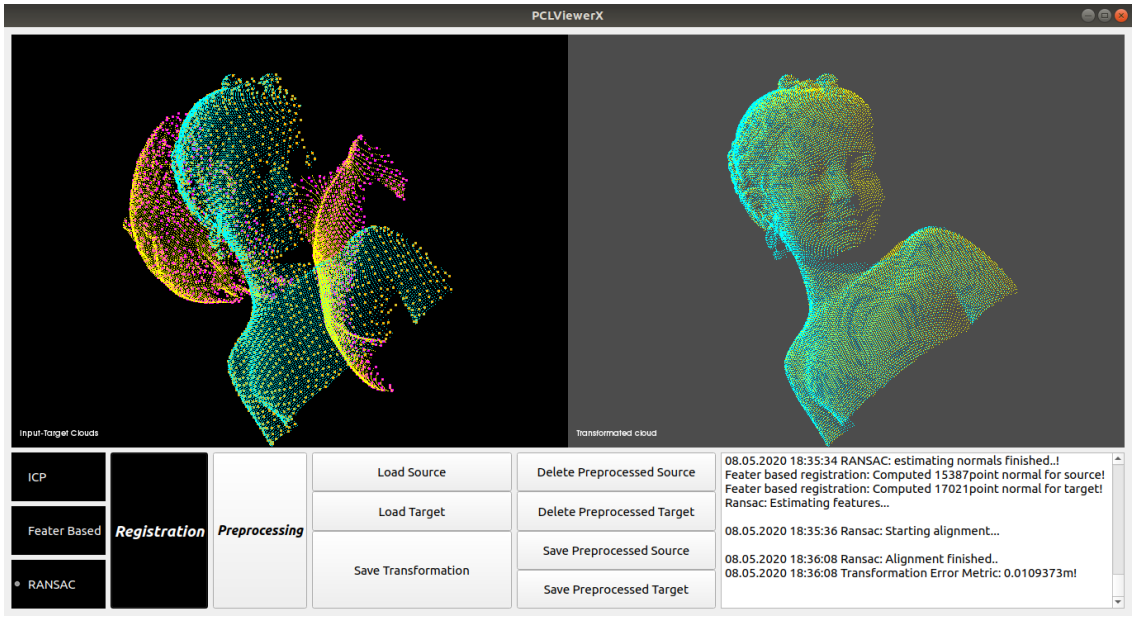
❖ *Megadhatjuk azt a határértéket, ami meghatározza, hogy az egyes iterálások során a párosításokon végzett transzformációk esetén elfogadunk-e egy párosítást vagy sem (lásd **RANSAC-alapú rossz párosítások elutasítása**).*

❖ *Az FPFH feature leíróhoz használt szomszédsági kereső sugár hosszát (lásd **FPFH módszerrel történő feature vektor kiszámításának leírásánál**).*

❖ *A normálszámításhoz használt kereső sugár nagyságát.*

❖ *A voxel rácshoz használt paramétert, ami megadja, hogy egy rács „doboz” oldal hossza mekkora legyen (lásd **voxel ráccsal történő mintavételezés**).*

❖ *Ezen kívül megadhatjuk a RANSAC által használt iterálások számát.*




*23. ábra RANSAC alapú regisztráció végeredménye*

## A regisztrálás után

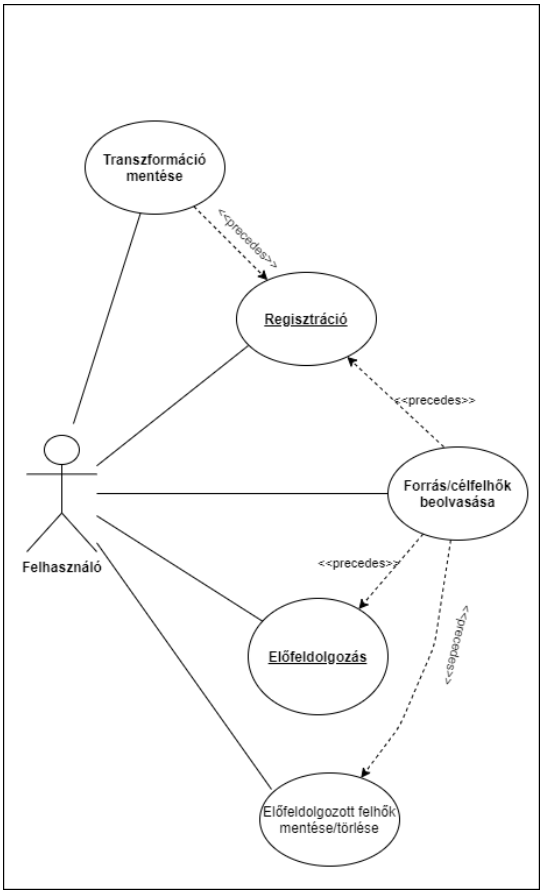
Miután a regisztráció befejeződött, az egyes részfolyamatok eredményeinek adatait láthatjuk a szöveges megjelenítő panelen. Minden esetben, ha a regisztrálás hiba nélkül végig futott, akkor annak pontosságát egy speciális hibametrikával ellenőrizhetjük. Ez a hibametrika végigmegy a transzformált felhő minden pontján, és egy keresőfa algoritmussal megméri az egyes pontok és a pontoktól számított legközelebbi célfelhőbeli szomszédsági pontok távolságát, majd ezt átlagolja. Ezzel megkapjuk, hogy átlagosan a pontokhoz viszonyítva mekkora az eltérés.

Ha elégedettek vagyunk a regisztrációval a „*Save Transformation*” gomb segítségével képesek vagyunk elmenteni a kapott transzformációs mátrixot. Ha rámegyünk a

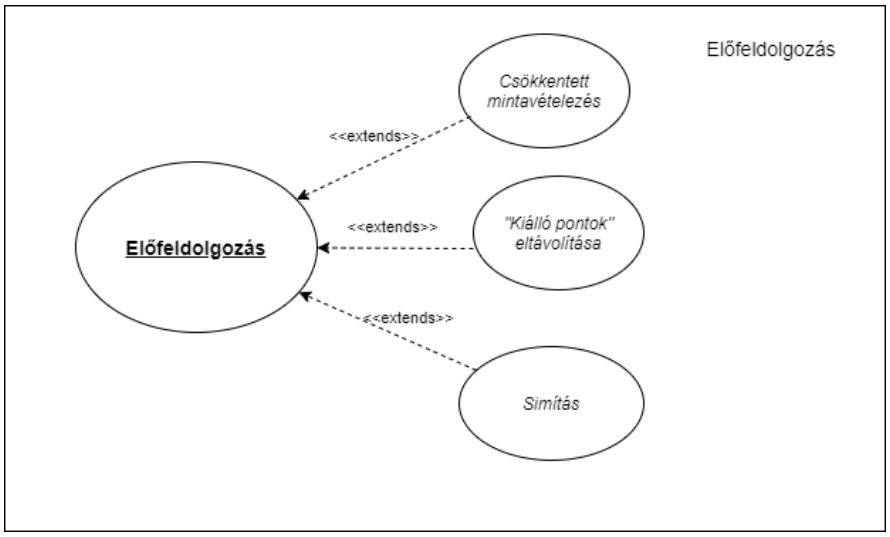
gombra, egy dialógus ablak jelenik meg, ami a mentéshez segít nekünk. Miután megadtuk a fájl nevét és azt, hogy hova szeretnénk menteni fájlt (ami tartalmazni fogja a regisztrációs transzformációt), és elfogadtuk a dialógus ablakot, a program szöveges formátumban beleírja a transzformációs mátrix tartalmát a fájlba. Ha hiba történt a fájl megnyitása során, akkor a program jelzi ezt nekünk üzenet formájában. Ha regisztrálás elött mentjük el a transzformációs mátrixot, akkor a 4x4-es egység mátrixot találhatunk meg a fájlunkban.

## Use - Case diagram

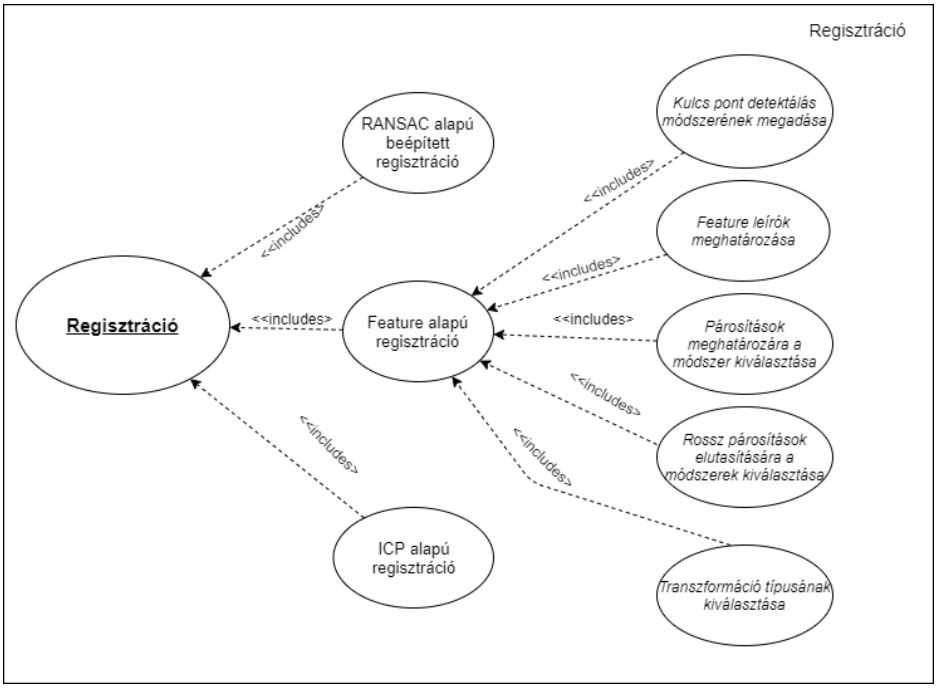
Az alábbi diagram bemutatja a felhasználónak biztosított lehetőségeket a program jóvoltából. Ezt követően a Regisztrációhoz, előfeldolgozáshoz tartozó részletező diagramokat láthatjuk. Ahogy a fentebb leírt használati útmutatóban is említésre került, láthatjuk, hogy ha bármilyen folyamatot akarunk végezni, először szükséges a bemeneti felhők beolvasása. Ezt követően végezhetjük el az egyes műveleteket.




*24. ábra A program Use-Case diagramja*




*25. ábra Use-Case diagram - Előfeldolgozás kifejtése*




*26. ábra Use-Case diagram - Regisztráció kifejtése*

# *Fejlesztői dokumentáció*

## Feladat specifikációja

A feladat célja olyan X Y Z alapú pontfelhő megjelenítő alkalmazás készítése, amivel lehetőségünk van a felhők változtatható módszerű regisztrálására, a regisztráció szemléltetésére, valamint a regisztráció előtti előfeldolgozásra **Point Cloud Library (PCL)** open-project, illetve **Qt** keretrendszer felhasználásával.

### Az alkalmazástól elvárt funkciók

- Könnyen kezelhető felhasználói felület biztosítása felhő megjelenítő panellel és a megfelelő eszközökkel ellátva.
- Felhők beolvasása a megfelelő gombok használatával, ezek megjelenítése.
- A beolvasott felhőkön előfeldolgozás végzésének lehetősége, ami során alkalmasabbá tesszük a felhőt a regisztrációra. Ehhez paraméter ablak biztosítása.
- A feldolgozás során az egyes előfeldolgozási módszerek közüli választás lehetőségének biztosítása.
- Az egyes módszerekhez paraméterek leírása, ezeknek megadási lehetőségei.
- Az előfeldolgozott felhők megjelenítése, hogy össze lehessen vetni az eredeti felhővel.
- Az előfeldolgozott felhők mentésére, törlésére való lehetőség biztosítása a megfelelő eszközökkel.
- Több regisztrációs módszer kínálata, valamint ezek kiválasztásának lehetősége, az erre alkalmas rádiógombok segítségével (RANSAC, ICP, feature-alapú regisztráció).
- Regisztrációs ablak biztosítása, ahol az egyes regisztrációtól függő beállításokat, paraméterezéseket végezhetjük el.
- Az adott regisztrációs ablakban lévő paraméterek rövid magyarázatának biztosítása, valamint ezekhez tartozó értékek módosíthatósága.
- Feature-alapú regisztráció esetén több lehetőség biztosítása az egyes folyamatokhoz, hogy a felhasználó különféle módszereket is kipróbálhasson.
- Regisztráció eredményének megjelenítése a felhőket megjelenítő panelen.

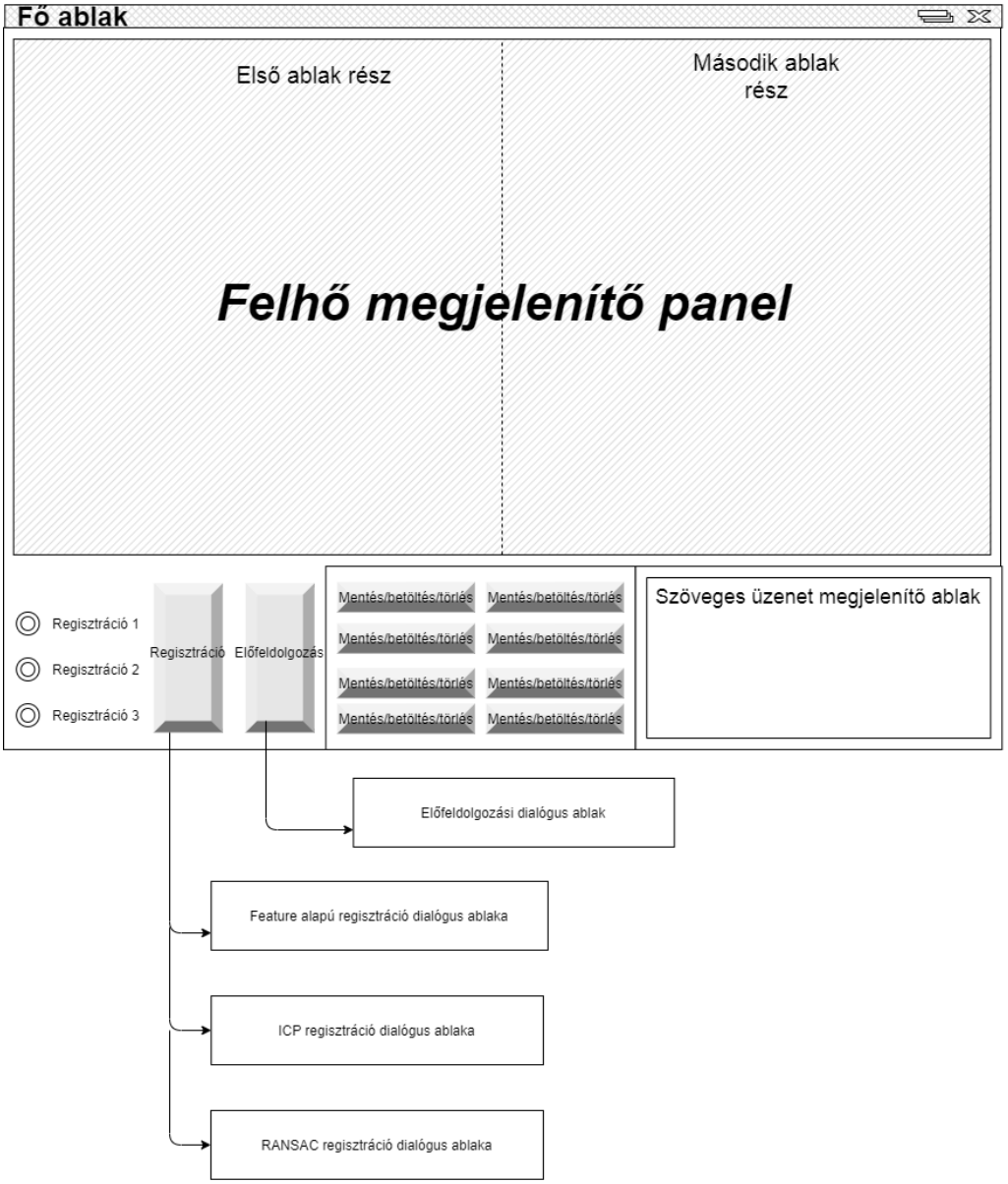
❖ Regisztrálást követően az eredmény mátrix elmentésére való lehetőség biztosítása.

❖ A regisztráció, illetve a teljes program működése során bekövetkező egyes történések részletes leírása az arra alkalmas szöveges megjelenítő panelen.

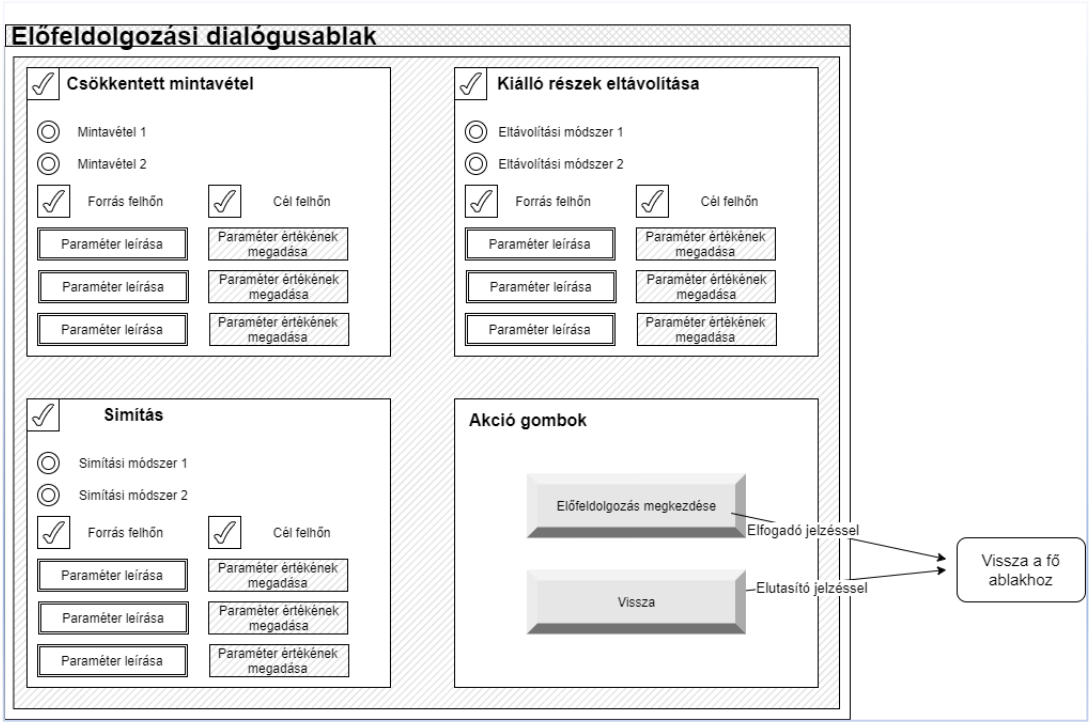
## Az alkalmazás felépítése

Az alkalmazás felépítése a *View – Model* sémára épül. A *View*-hoz társulnak további nézet osztályok, amik a dialógus ablak szerepet töltik be, így a program két fő részre bontható. Míg a *View* egyes részei (előfeldolgozáshoz, feature-alapú regisztráláshoz, *ICP* alapú regisztráláshoz, illetve *RANSAC* segítségével történő regisztráláshoz használt dialógus ablakok, valamint az alkalmazás fő megjelenítő ablaka) számos *Qt* specifikus elemeket tartalmaznak, és csak a fő ablak tartalmaz PCL specifikus technikákat, addig a modell is csak minimális Qt specifikus elemeket használ (események küldésére). Itt használjuk az egyes PCL által biztosított, speciális algoritmusokat a regisztrálást felépítő lépésekhez, kezeljük az egyes felhőket, valamint a regisztrálási módszereket is itt építjük fel. Az alkalmazás fő osztályát a *PCLViewerX* valósítja meg, ami adattagjaiban tartalmazza a további osztályok által megvalósított objektumokat. Ennek az osztálynak a segítségével jelenítjük meg az alkalmazás fő ablakát, és ez az osztály fogja meghívni a modellt az egyes funkciók végrehajtásához. A modellen kívül a dialógus ablakokat is ő hozza létre, majd jeleníti meg őket, ha szükséges. A modell szerepét a *PCLViewerXModel* osztály tölti be. Az *ICP* alapú regisztrációhoz szükséges paraméterek megadásának lehetőségét a *ICPRegistrationParamsDialog* osztály biztosítja egy dialógus ablak formájában. Hasonló módon a *RANSAC* regisztrációhoz szükséges paraméterek megadását biztosító, illetve az előfeldolgozáshoz, valamint a feature-alapú regisztrációhoz szükséges paraméterek megadását biztosító osztályok is dialógus ablakok formájában valósulnak meg.

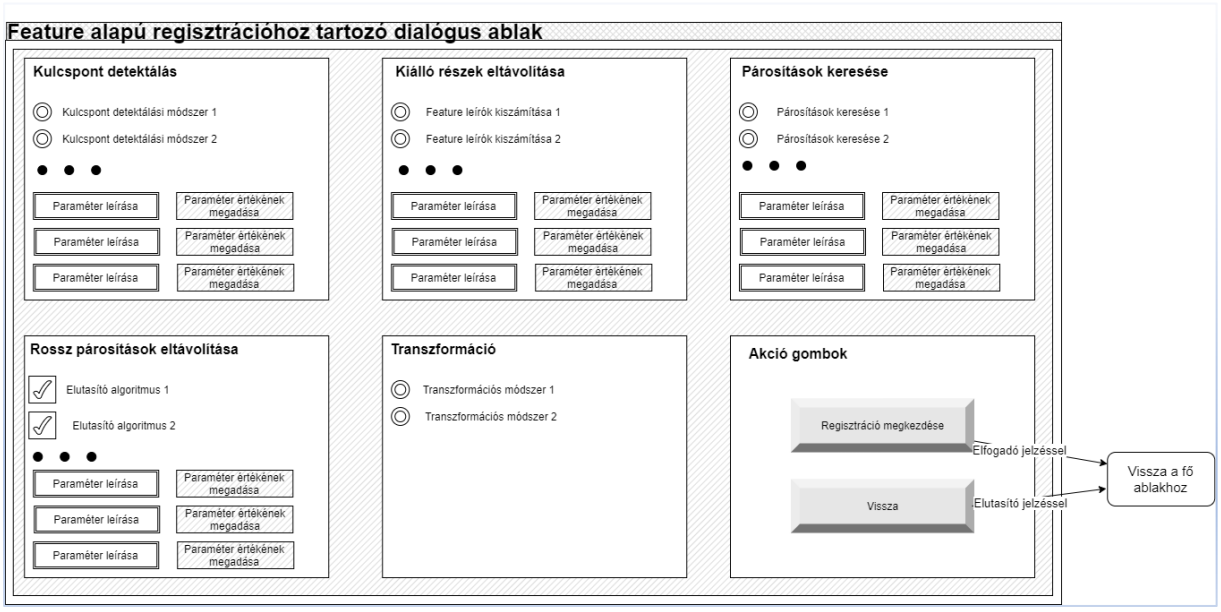
## Felhasználói felületi model




*27. ábra A főablak felületi modellje*

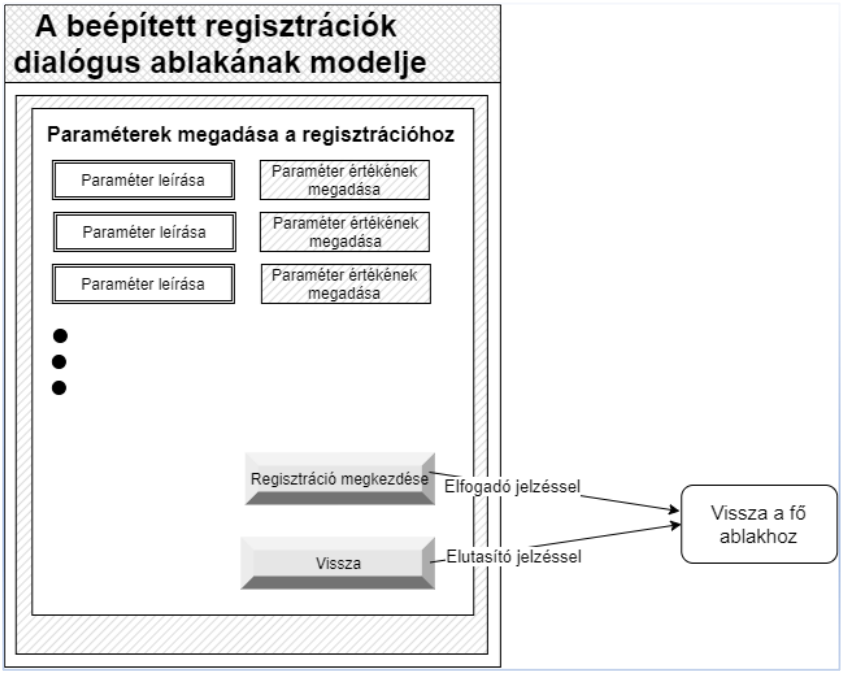



*28. ábra Az előfeldolgozás dialógus ablakának felületi modellje*



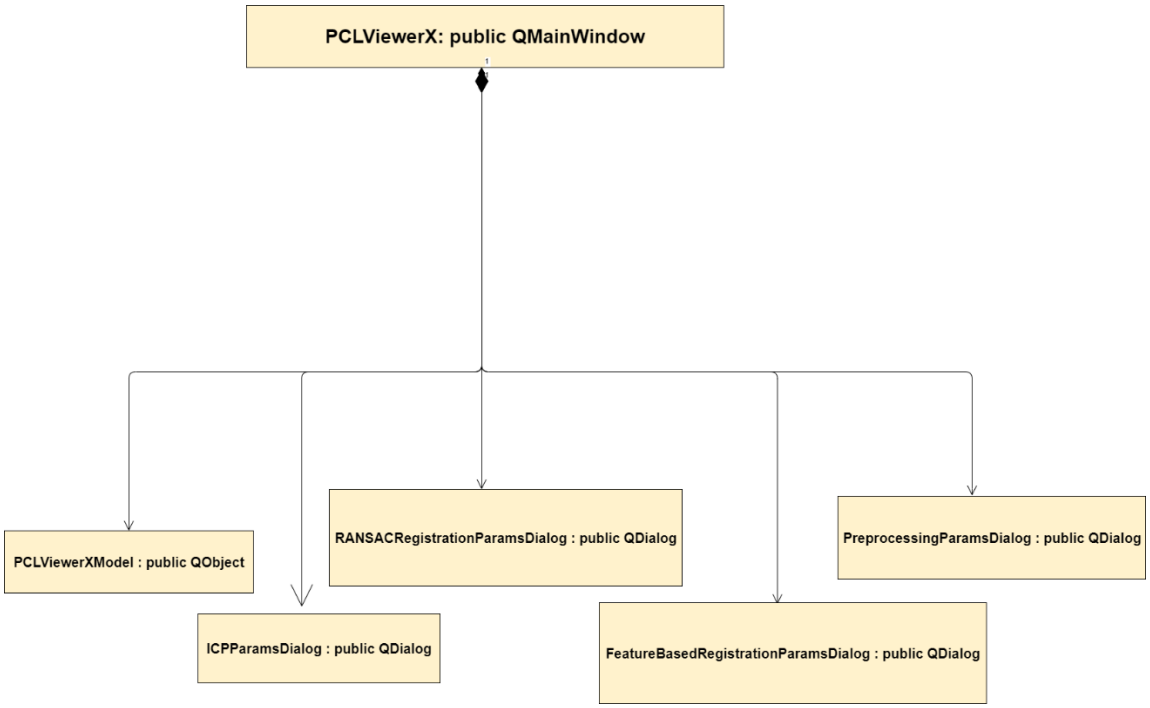

*29. ábra Feature-alapú regisztráció dialógus ablakának felületi modellje*

## ICP/RANSAC regisztrációkhoz tartozó dialógus ablak modellje



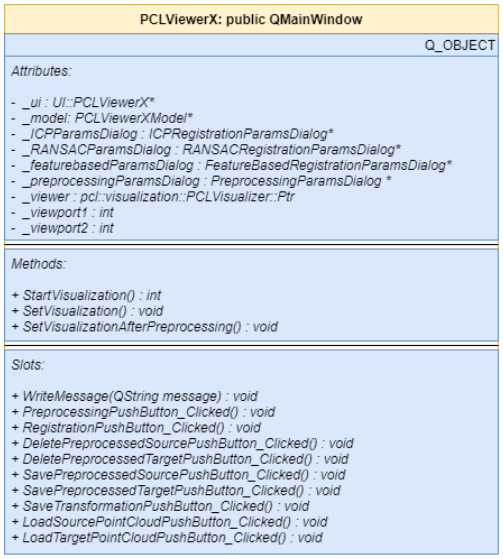

*30. ábra ICP/RANSAC regisztráció dialógus ablakának felületi modellje*

# Az alkalmazás osztálydiagramja

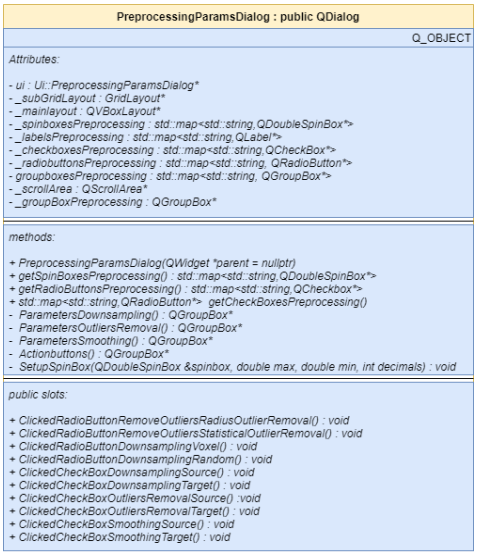



*31. ábra Az alkalmazás részletes osztálydiagramja*

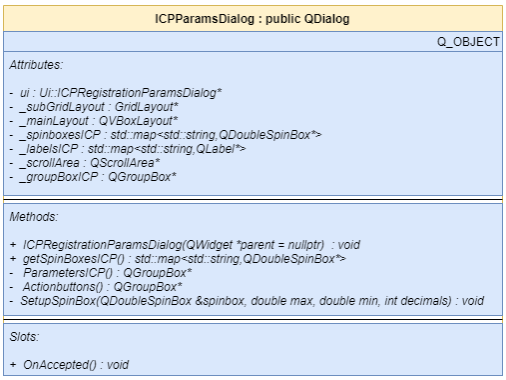
## Az egyes osztályok osztálydiagramjainak kifejtése



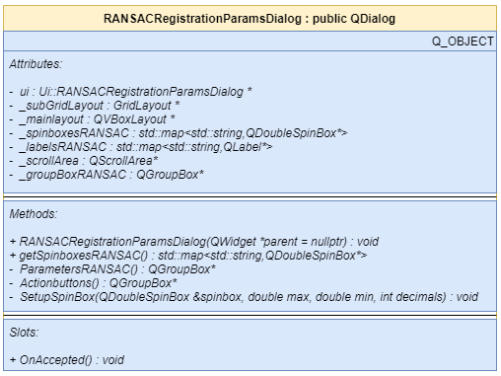

*32. ábra PCLViewerX osztálydiagramja*




*33. ábra PreprocessingParamsDialog osztálydiagramja*




*34. ábra ICPParamsDialog osztálydiagramja*




*35. ábra RANSACRegistrationParamsDialog osztálydiagramja*


*36. ábra FeatureBasedParamsDialog osztálydiagramja*


*37. ábra PCLViewerXModel osztálydiagramja*

# Fejlesztői környezet, a megvalósításhoz alkalmazott technológiák ismertetése

A program fejlesztése C++ nyelven, Qt szoftver fejlesztői keretrendszer segítségével, Qt Creator integrált fejlesztői környezet használata mellett történt. A program maga egy Qt típusú, modulokon alapuló asztali alkalmazás, amely az egyes felhők megjelenítésére, regisztrálására, előfeldolgozására a Point Cloud Library 1.8-as verziójú open project-et használja. A következőkben ezek lesznek részletesebben kifejtve.

## Qt keretrendszer és a Qt Creator rövid összefoglalója [20]

A Qt szoftver fejlesztői keretrendszer egy modul gyűjteményt biztosít a felhasználóknak a grafikai felhasználói felületek (GUI) elkészítéséhez, de olyan GUI nélküli alkalmazások elkészítésére is lehetőséget ad, mint például parancssori eszközök és szerverekhez szükséges konzolos programok. Mivel a Qt segítségével hatékony cross-platform alkalmazásokat tudunk fejleszteni, ezért az egyik legelterjedtebb fejlesztői eszköz lett. Számos operációs rendszerrel képes futni, mint Linux, Windows, macOS vagy Android. A Qt keretrendszer használatával történő fejlesztés a Qt Creator segítségével valósul meg. Ez egy szemléletes fejlesztői környezet beépített eszközökkel a fejlesztői felület tervezéséhez, kódszerkesztéshez. A buildelés kétféleképpen is történhet. Az első módszer, hogy a Qt saját eszközét használjuk, vagyis qmake-et. A másik módszer, hogy CMake segítségével végezzük el a buildelési folyamatot. A fejlesztés során a CMake buildelési módszert használtunk, mivel az egyszerűbb számunkra a PCL használata miatt. A CMake a CMakeLists.txt fájlt használja a program felépítéséhez. Ebben írjuk le a különböző függőségeket, illetve adjuk hozzá az általunk használt csomagokat, könyvtárakat.

### *Az eszközök közötti kommunikáció, események és eseménykezelők [21]*

Az ablakok, illetve azok objektumai közötti kommunikációja *Signal-ok* és *Slot-ok* segítségével valósul meg, vagyis eseményekkel és eseménykezelőkkel. Ezek az események és eseménykezelők a Qt keretrendszer egyik fő sajátosságai. Ezt a

kommunikáció fajtát használjuk többek között akkor, amikor a felhasználó valamilyen nyomógombot lenyom, vagy valamilyen rádiógombot, jelölőnégyzetet bejelöl. Ha ezeket az eseményeket „elkaptuk”, akkor az eseménykezelőkkel megadhatjuk, hogy mit csináljon ilyen esetekben a program. Ennek segítségével nem csak jelzéseket küldhetünk, de például adatot is küldhetünk az egyes ablakok között. Ezt egyéni eseményekkel valósíthatjuk meg. Az egyes eseményeket és az eseménykezelőket a *connect* függvény segítségével párosíthatjuk össze. A dialógus ablakok elfogadása, illetve elutasítása esetén ilyen eseményekkel jelzünk a fő ablaknak (*accept*() elfogadás esetén, és *reject*() az elutasítás esetén). Egy egyéni esemény megtörténését egy *emit* paranccsal vagyunk képesek jelezni.

### *Qt Creator telepítési útmutatója Ubuntu 18.04-es operációs rendszeren*

A Qt Creator telepítésének két módja is van ezen az operációs rendszeren.

**(1) Az első módszer, amikor a Qt *hivatalos oldaláról* töltjük le (ajánlott).** A letöltések menüfülnél kikeressük a „*Downloads for open source users*” részt, majd rámegyünk a *„Go open source”* gombra. Ezt követően egy új oldalra kerülünk, ahol, ha letekerünk teljesen az oldal aljára, akkor megtaláljuk *a „Download the Qt Online Installer”* feliratú gombot. Ha erre is rámegyünk, akkor szintén egy új oldalra kerülünk, ahol az oldal a saját operációs rendszerünk alapján ajánl egy letöltési verziót az online telepítőhöz. Miután ezt letöltöttük és futtattuk, akkor a hagyományos módon végig vezet minket a telepíteni kívánt modulokon, illetve a telepítés egyes beállításain.

**(2) A második módszer, amikor egyszerűen a következő parancsokat egymás után alkalmazzuk a parancssorban** (nem biztos, hogy a legújabb verziót fogja telepíteni, így nem ajánlott).

***(a)*** *sudo apt install build-essential*

***(b)*** *sudo apt install qtcreator*

### *CMake telepítési módja Ubuntu 18.04 verzió esetén*

A következőkben két telepítési fajtáját mutatjuk meg a CMake-nek, ha esetleg még nincs rajta a számítógépen.

**(1) Az első módszer**, hogy Ubuntu szoftverek alkalmazás tárolójából kikeressük a CMake alkalmazást és a hagyományos módon telepítjük.

**(2)** **Másik módszer,** amikor konzolból telepítjük a következő parancs segítésével.

*sudo snap install cmake*

## Point Cloud Library [22]

A Point Cloud Library egy egyedülálló, a pontfelhő feldolgozáshoz fejlesztett nagyszabású nyitott projekt, ami számos korszerű algoritmust biztosít a felhasználók számára. Ilyen fajta algoritmusok például a szűrő, feature meghatározó, felületi rekonstruáló, regisztrációs, illetve szegmentáló algoritmusok. Ezek az algoritmusok használhatók kulcspont keresésre, zajos adatok javítására, kiugró pontok eltávolítására, 3D pontfelhők regisztrálására, valamint az egyes kulcspontok által meghatározott feature vektorok kiszámítására annak érdekében, hogy az egyes objektumokat felismerjük a különböző geometriai közelítések segítségével, majd közös felületre vetítsük és vizualizáljuk őket. Mivel a PCL a cross-platform fejlesztést lehetővé teszi, így Linux, MacOS, Windows és Android operációs rendszereken is használható, de jelentősen eltérő telepítési mód szükséges hozzájuk.

### *A PCL 1.8-as, előre felépített verziójának telepítése Ubuntu 18.04 típusú operációs rendszerhez*

A *Point Cloud Library* telepítésére szintén több lehetőségünk is van. Ezek közül a legegyszerűbbet fogjuk bemutatni. Ehhez mindössze a parancssorban kell futtatni a következő parancsot*: „sudo apt install libpcl-dev”.*

Ekkor az operációs rendszer telepíti a PCL használatával történő fejlesztéshez szükséges fájlokat.

### *A Pontfelhő [23]*

A pontfelhő egy olyan adatstruktúra, ami arra lett kifejlesztve, hogy több dimenziós pontok gyűjteményét tárolja. Az egyes pontokat az X, Y, Z koordináták képviselik általában, de a különböző pontfelhők esetén ennél több dimenzióba is átnyúlhatnak az egyes pontok. A pontfelhőket reprezentáló, alapvető adattípus a PCL-ben a *pcl::PointCloud<pcl::PointT>*. A PointCloud egy C++ osztály, ami a következő mezőket tartalmazza:

**1)** ***width*** (int): Specifikálja egy pontfelhő adathalmaz szélességét a pontok számával. Két jelentése is lehet:

- Jelentheti az összes pontot a pontfelhőben rendezetlen adathalmaz esetén.
- Jelentheti egy rendezett adathalmaz „szélességét” (a pontok számának összege egy adott sorban). A rendezett felhők olyan pontfelhők, amik egy rendezett képre (vagy mátrixra) hasonlítanak, vagyis olyan struktúrák, ahol az adatok sorokra és oszlopokra vannak tagolva. ***A projekt során mi rendezetlen felhőkkel dolgozunk.***

**2)** ***height*** (int): Meghatározza a magasságát egy pontfelhő adathalmaznak pontokban mérve. A szélességhez hasonlóan ennek is két megadási módja lehet.

- Rendezett adathalmaz esetén a sorok számát jelöli.
- Rendezetlen adathalmaz esetén ennek értéke 1.

**3)** ***points*** (std::vector<PointT>): Ez tartalmazza az adattömböt, ahol az összes PointT típusú pont tárolva van. Például a mi esetünkben egy felhő points adatmezője, ami XYZ típusú adatokat tárol, egy pcl::PointXYZ elemekből álló vektort tartalmaz.

**4)** ***is_dense*** (bool): Ez a mező mondja meg, hogy egy felhő minden pont mezője véges érték-e. Ekkor az értéke a mezőnek *true*. Ha bizonyos pontok XYZ értéke *Inf* vagy *NaN* érték (Infinite, Not a Number), akkor az *is_dense* értéke false lesz.

### *A használt pontfelhő típusok [24]*

Egy PointT típus leírja, hogy milyen egyedi tulajdonságokat tárol az adott pontfelhő. A PCL nagy választékot biztosít a különböző ponttípusokhoz. Ezeket meg kell ismernie a fejlesztőnek, hogy milyen adatokat tárol az adott felhő, a kód egyszerűbb esetleges javításának, fejlesztésének érdekében.
A következőkben a program által használt alapvető struktúrák lesznek elemezve.

#### *(1) PointXYZ – Elemei: x, y, z*

Ez az egyik leggyakrabban használt adattípus a PCL-ben, mivel a háromdimenziós tér X, Y, Z koordinátáit tárolja csupán float adattípusokkal reprezentálva. Ilyen pont típussal tároljuk az alkalmazásban a legtöbb felhőt. Például a bemeneti felhőket, a kulcspontokat tartalmazó felhőket, az előfeldolgozott felhőket. A felhasználó ekkor egy adott felhő

adott pontjának x értékét a következőképpen érheti el: cloud.points[i].data[0] vagy pedig cloud.points[i].x.

```
union
{
  float data[4];
  struct
  {
    float x;
    float y;
    float z;
  };
};
```

*38. ábra A PointXYZ struktúra ábrázolása [24]*

### *(2) PointXYZI – Elemei: x, y, z, intenzitás*

Egyszerű XYZ pont típus intenzitással kiegészítve. Az „ideális” környezetnek a leírásához ezek a pontok szükségesek, hogy készítsünk egy egyszerű struktúrát. Ezt a struktúrát csak néhányszor használjuk a fejlesztés során, akkor is csak minimálisan. Gyakori típus a kulcspont detektálásoknál, a kulcspont kereső algoritmusok sokszor az intenzitást vizsgálják egy felhő esetén (ISS kulcspont detektáló implementálása, valamint *HarrisKeypoint3D* használata során is ilyen típusú felhőkbe számolja az értékeket). Az intenzitás adatot külön struktúrában tárolják a XYZ pontokhoz képest (de elérésük hasonló módon történik) annak érdekében, hogy az algoritmusok, amik nem használják ezeket a tulajdonságokat, ne írják felül az értékét. (A SSE rendezés miatt van benne a data_c tömb, mivel az további három float értéket is eltárolt az intenzitáshoz, és így kompatibilis maradt vele.)

```
union
{
  float data[4];
  struct
  {
    float x;
    float y;
    float z;
  };
};
union
{
  struct
  {
    float intensity;
  };
  float data_c[4];
};
```

*39. ábra A PointXYZI struktúra ábrázolása [24]*

### (1) PointWithScale – float x, y, z, scale

A PointXYZI-hez hasonló, azzal a különbséggel, hogy a *PointWithScale scale* mezője eltárol egy skálát, amin egy bizonyos pontot figyelembe vettek egy valamilyen geometriai művelethez (például egy gömb sugara a legközelebbi szomszédok kiszámolására, vagy az ablak mérete stb). Ezzel a ponttípussal az alkalmazás során szintén csak a kulcspont detektálásnál találkozunk, a SIFT kulcspont detektáló esetén, ami ilyen adattípusban tárolja el az általa kiszámolt skálákhoz tartozó mértékeket.

```
struct
{
  union
  {
    float data[4];
    struct
    {
      float x;
      float y;
      float z;
    };
  };
  float scale;
};
```

*40. ábra A PointWithScale struktúra ábrázolása [24]*

### (2) Normal – float normal[3] , görbület

Ez a másik leggyakrabban használt adat típus a PCL-ben. A *Normal* struktúra eltárolja a felületi normát egy adott pontban, valamint egy mérési görbületet (ezek kiszámításához több információt nyújt a PCL *NormalEstimation* osztályának dokumentációja [26]). Az előbbiekhez hasonlóan történik itt is az egyes adattagok elérése. Például x-hez tartozó normál értékét egy adott felhő adott pontjához a következőképpen érheti el a felhasználó: *cloud.points[i].data_n[0]* vagy *cloud.points[i].normal[0]* , valamint cloud.points[i].normal_x. A görbületet külön struktúrában tároljuk, hogy az egyes műveletek ne írják felül őket, ha nem használják.

```
union
{
  float data_n[4];
  float normal[3];
  struct
  {
    float normal_x;
    float normal_y;
    float normal_z;
  };
}
union
{
  struct
  {
    float curvature;
  };
  float data_c[4];
};
```

*41. ábra A Normal struktúra ábrázolása [24]*

### (3) PFHSignature125 – float histogram[125];

Az egyes feature leírók kiszámítására szolgáló algoritmusok külön adattípusokat használnak az általuk kiszámolt értékek eltárolására. Ez a pont típus PFH (Point Feature Histogram) értékeket tárol egy adott ponthoz. Ezt egy 125 méretű tömbbel reprezentálja. Egy adott ilyen leírókat tartalmazó felhő, egyik leírójának adatait a következőféleképpen érhetjük el: *cloud.points.histogram*[i] (az indexelésre figyelni kell természetesen). Az értékeket tartalmazó tömb méretét pedig a *descriptorSize*() tagfüggvénnyel érhetjük el. A programunkban ez a pont típus a feature vektorok meghatározása esetén játszik szerepet, pontosabban a PFH módszerrel történő kiszámításnál.

```
struct PFHSignature125
{
  float histogram[125];
  static int descriptorSize () { return 125; }

  friend std::ostream& operator << (std::ostream& os, const PFHSignature125& p);
};
```

*42. ábra PFHSignature125 struktúra*

### (4) FPFHSignature33 – float histogram[33]

Egy egyszerű ponttípus, ami az FPFH (Fast Point Feature Histogram) értékeit tárolja egy adott ponthoz. Ezt egy 33 hosszú tömbbel reprezentálja. Egy adott ilyen leírókat tartalmazó felhő egyik leírójának adatait a következőféleképpen érhetjük el: cloud.points.histogram[i]. A tömb méretét pedig a descriptorSize() tagfüggvénnyel érhetjük el. Ezt a pont típust szintén az FPFH módszerrel történő feature vektorok kiszámításához használjuk, megtalálható mind a RANSAC-alapú, mind a feature-alapú regisztráció implementálása esetén.

```
struct FPFHSignature33
{
  float histogram[33];
  static int descriptorSize () { return 33; }

  friend std::ostream& operator << (std::ostream& os, const FPFHSignature33& p);
};
```

*43. ábra FPFHSignature33 struktúra*

### *(5) SHOT352 – float descriptor[352] – float rf[9]*

Egy újabb ponttípus, ami a SHOT (Signature of Histograms of OrienTations) értékeit tárolja el egy adott ponthoz egy 352 hosszú tömb segítségével. Egy ilyen leírókat tartalmazó felhő egyik leírójának adattagjait a következőképpen érhetjük el: cloud.points.descriptor[i]. A tömb méretét pedig a *descriptorSize*() tagfüggvénnyel érhetjük el. Ezt a pont típust is a feature kiszámításnál használjuk, konkrétan a SHOT módszerrel történő kiszámítás esetén.

```
struct SHOT352
{
  float descriptor[352];
  float rf[9];
  static int descriptorSize () { return 352; }

  friend std::ostream& operator << (std::ostream& os, const SHOT352& p);
};
```

*44. ábra SHOT352 struktúra*

### *A PCD fájlformátum*

A pontfelhők mentésére egy PCD nevezetű fájlformátum áll rendelkezésre, ami eltárolja a fentebb említett mezőket, valamint egyéb speciális információkat megfelelően tagolva. Az egyéb információk közé tartozik például a *PCD* fájl verziója (*VERSION*). Az egyes pontok típusait a *FIELDS*, illetve a *TYPE* mezők adják meg. A fejlesztés során ilyen típusú fájlokat olvasunk be, illetve ilyen típusú fájlokba írjuk az esetleges képzett felhők adatait. Sokszor ezeket a fájlokat binárisan tároljuk.

```
1  # .PCD v.7 - Point Cloud Data file format
2  VERSION .7
3  FIELDS x y z
4  SIZE 4 4 4
5  TYPE F F F
6  COUNT 1 1 1
7  WIDTH 3400
8  HEIGHT 1
9  VIEWPOINT 0 0 0 1 0 0 0
10 POINTS 3400
11 DATA ascii
12 -18.23139954 -27.45949936 2.90733004
13 -18.07099915 -27.41930008 4.03047991
14 -18.02630043 -27.09539986 1.66534996
15 -18.03333282 -25.98942375 3.13369989
16 -17.98999977 -28.38290024 2.69094992
17 -17.92009926 -25.77630043 1.77848995
18 -17.80030060 -27.89410019 1.37961996
19 -17.79070091 -28.34110069 3.93017006
20 -17.53554535 -27.10744476 0.62723821
21 -17.58323860 -25.73995781 0.81091875
```

*45. ábra PCD fájl példa*

## *A fejlesztés során használt egyéb PCL osztályak és azok használata*

A következőkben végig vesszük az egyes regisztrációs és előfeldolgozási folyamatokhoz, illetve megjelenítéshez használt, PCL által biztosított, a programban szereplő egyéb osztályokat, és azoknak a működését, lehetőségeit, valamint a programban való használatukat vizsgáljuk.

### *Kulcspont detektáláshoz használt osztályok és azok használata [25]*

Az ide tartozó osztályok mind a PCL-nek a „*keypoints*” alkönyvtárában vannak implementálva.

**(1)** ***pcl::ISSKeypoint3D< PointInT, PointOutT >:*** Az ISS (Intrinsic Shape Signatures) módszerrel történő kulcspont detektáláshoz a fejlesztés során a PCL *ISSKeypoint3D* osztálya lett használva, ami a megfelelő paraméterekkel és bemeneti felhővel a számolást végző metódus paraméterében megadott kimeneti felhőbe másolja a bemeneti felhő azon pontjait, amelynek indexeit kiszámolta a kulcspontdetektáló algoritmus. A *PointInT,* illetve a *PointOutT* template paraméterek megadják a bemeneti és a kimeneti felhő pontjainak a típusait. (A mi esetünkben mind a kettő pcl::PointXYZ ).

**(2)** ***pcl::HarrisKeypoint3D< PointInT, PointOutT >***: A Harris módszerrel történő kulcspont detektáláshoz a PCL *HarrisKeypoint3D* osztályt használjuk. Mivel a Harris algoritmus intenzitásokat számol az egyes élek/sarkok megkereséséhez, ezért a kimeneti felhő típusa a mi esetünkben *PointXYZI* típusú lesz. Ennek típusát a második template paraméterben adjuk át. Az első template paraméterben pedig az előzőhöz hasonlóan a bemeneti felhő pontjainak típusát *pcl*::*PointXYZ* típusként adjuk meg. A kulcspont detektáláshoz szükséges bemeneti paraméterek a megfelelő *Setter* függvényekkel állítjuk be, valamint beállítunk neki egy kereső fa objektumot is a pontok közötti hatékony keresés érdekében. A bemeneti felhőt, illetve az eredmény (kimeneti) felhőt az osztály megfelelő metódusainak paramétereként adhatjuk át. Mivel a fejlesztés során a későbbi algoritmusok *PointXYZ* típusú bemeneti paramétereket várnak, ezért fontos, hogy a kimeneti felhőt átkonvertáljuk ilyen típusúra.

***(3)*** ***pcl::SIFTKeypoint< PointInT, PointOutT >:*** A SIFT algoritmus alapján történő kulcspont detektáláshoz a PCL *SIFTKeypoint* osztályát használjuk. A bemeneti template paraméter típus az ezt megelőzőekhez hasonlóan *pcl::PointXYZ* típusú, míg a kimeneti paraméter ebben az esetben *pcl::PointWithScale* típusú lesz, mivel ez az algoritmus különböző skála mértékekkel számol, és ezeknek eltárolásához ezt a ponttípust használja. Az algoritmus által az egyes számolásokhoz szükséges paramétereket a megfelelő *Setter* függvénnyel adhatjuk meg, valamint kereső fa objektumot is megadhatunk a fában történő hatékony keresés érdekében. A bemeneti, illetve a kimeneti felhőket a megfelelő metódus paramétereként adhatjuk át. A template paraméterből adódóan a kimeneti pontfelhő *PointWithScale* típusú pontokat fog tartalmazni, ezért figyelnünk kell ennek átkonvertálására *PointXYZ* típusra, hasonló módon, mint a *Harris módszernél.*

### *Feature leírókhoz (vektorokhoz) használt osztályok és azok használata [26]*

Ezek az osztályok a PCL-nek a *features* alkönyvtárában vannak implementálva.

**(1)** ***pcl::PFHEstimation< PointInT, PointNT, PointOutT >***: A PFH módszerrel történő feature vektor számításhoz a *PFHEstimation* osztályt használjuk. Három template paraméterrel rendelkezik, amelyek rendre a bemeneti felhő, a bemeneti normákat tartalmazó felhő, illetve a kimeneti felhő típusai. A mi esetünkben a bemeneti felhő pcl::PointXYZ típusú lesz, a normál felhő pcl::Normal típusú, illetve a feature vektorokat tartalmazó felhő pedig a PFHEstimation típussal kompatibilis *pcl::PFHSignature125* típusú lesz. A template paramétereknek megfelelő bemeneti, normál, illetve a kimeneti felhőket az objektum megfelelő *Setter* (pl *setInputCloud*) függvényeivel vagy a számításért felelős függvény (*compute*) paramétereként adhatjuk meg. Egyéb paramétereket szintén az adott paraméterhez tartozó Setter függvénnyel adhatjuk meg.

**(2)** ***pcl::FPFHEstimationOMP< PointInT, PointNT, PointOutT > :*** Az FPFH módszerrel történő feature vektor számításhoz az *FPFHEstimationOMP* osztályt használjuk. Mint ahogy *PFHEstimation* esetében láttuk, hasonló módon adjuk meg a bemeneti template paramétereket. Az első lesz a bemeneti felhő típusa, ami pcl::PointXYZ lesz, a második paraméter a normál felhő típusa, ami pcl::Normal lesz. A harmadik, vagyis a kimeneti

felhő pontjainak típusa az algoritmussal kompatibilis *pcl*::*FPFHSignature33* típus lesz. A PFH-hoz hasonló módon különböző paramétereket adunk meg az objektumnak a helyes működés érdekében. Az OMP jelölés azt jelenti, hogy az algoritmus egyszerre több szálon is futhat, ezzel megengedve a párhuzamos futtatást.

**(3)** ***pcl::SHOTEstimationOMP< PointInT, PointNT, PointOutT >***: A SHOT algoritmus alapján működő feature vektor kiszámításhoz a *SHOTEstimationOMP* osztályt használjuk. Az objektum felparaméterezése az előzőekhez hasonlóan működik, a template paraméterek itt is a bemeneti, a normál, illetve a kimeneti felhők típusait jelentik, ahol a bemeneti felhő *pcl::PointXYZ*, a normál pedig *pcl::Normal* típusú lesz. A kimeneti felhő a *SHOTEstimation* típussal kompatibilis *pcl::SHOT352* típusú leíró lesz. A számítási művelethez használt paramétereket a megfelelő metódus segítségével adjuk meg. Az OMP az osztály nevének végén azt jelöli, hogy a számítás során képes több szálon is műveletet végezni.

Mind a három feature vektor kiszámításra alkalmas osztály esetén az algoritmushoz használt egyes paraméterek megadásai hasonló módon történnek. A bemeneti felhőt a megfelelő *setInputCloud Setter* függvénnyel adhatjuk meg. Az algoritmushoz használt keresősugár nagyságának megadása a *setRadiusSearch* segítségével, a felhőben történő hatékony kereséshez használt algoritmus beállítása a *SetSearchMethod* segítségével történik. A bemeneti felhőn kívüli keresőfelületet a *setSearchSurface* metódus paramétereként adhatjuk meg, ami arra szolgál, hogy a feature leírók számításához ne csak a kulcspontokat vegye figyelembe a szomszédok keresése során. A tényleges számolás a *compute* metódus meghívásával történik, aminek paraméterében a kimeneti felhőt adjuk át. Ideális esetben miután a számolást befejezte az adott algoritmus, akkor a megfelelő feature leíró típusú felhőben, amit megadtunk paraméterül, megtalálhatjuk a kiszámolt leírókat. Az *FPFHEstimationOMP*, illetve a *SHOTEstimationOMP* több szálas működése miatt beállíthatjuk a *setNumberOfThreads Setter* segítségével az algoritmus által használni kívánt szálak számát.

### *A párosítások számításához használt osztályok és azok használata [27]*

***pcl::CorrespondenceEstimation< T , T >:*** Az adott feature leírók párosításához szolgáló algoritmus megvalósítása a *CorrespondenceEstimation* osztály által történik, ami a PCL-nek a *registration* alkönyvtárában van implementálva. Template paramétereknek azokat a feature vektor típusokat adjuk meg, amelyek között összetartozó pontokat szeretnénk találni. Az első template paraméter a forrás kulcspontjaihoz tartozó feature leíró típusa, a második pedig célfelhőhöz tartozó kulcspontokhoz kiszámolt feature leírók típusa. A fejlesztés során három fajtája lehet ezeknek a típusoknak: a felhő típusoknál említett *pcl::PFHSignature125*, *pcl::FPFHSignature33*, illetve a *pcl::SHOTEstimation* segítségével kiszámolt leírók típusa, a *pcl::SHOT352*. A template paramétereken kívül megadhatjuk a típuson belüli, általa felkínált két számítási módszerének egyikét. Az egyik a közvetlen számítás, a másik pedig a kölcsönös számítás. Attól függ, hogy melyiket számolja, hogy az osztály *determineCorrespondences* (közvetlen módszer) vagy a *determineReciprocalCorrespondences* (kölcsönös módszer) metódusát hívjuk meg a számításhoz. Ennek a két függvénynek meg kell adnunk azt az objektumot, amibe az egyes párosításokat szeretnénk eltárolni.

***pcl::CorrespondencesPtr (boost::shared ptr<Correspondences>):*** A párosítások eltárolásához a *CorrespondencePtr* osztályt használjuk, ami lényegében a Correspondences-hez tartozó *boost* könyvtárbeli *smart pointer* alias elnevezése. A *Correspondences* pedig a pcl::*Correspondence* típusú párosításokat tartalmazó vektor alias elnevezése.

A **pcl::Correspondence** elemek tartalmazzák a párosítani kívánt pontok forráshoz, illetve célhoz tartozó indexeit, valamint a közöttük lévő távolságot a 3D-s térben.

### *A rossz párosítás elutasításához használt osztályok és azok használata [27]*

Ezek az osztályok a PCL-nek a registration alkönyvtárában vannak implementálva.

**(1) pcl::registration::CorrespondenceRejectorSampleConsesnsus< PointT >** : A RANSAC algoritmus használatával történő rossz párosítások elutasításához a *CorrespondenceRejectorSampleConsensus* template osztályt használjuk. Template

paramétere megadja a párosítások által tárolt indexekhez tartozó felhők típusát, amelyek között szeretnénk kiválasztani a helyes párosításokat. További paraméterként az iterálások számát, a kiválasztási határértéket, illetve a bemeneti párosításokat a *setInlierThreshold*, *setMaximumIterations*, valamint a *setInputCorrespondences* segítségével adhatjuk meg.

**(2)** ***pcl::registration::CorrespondenceRejectorDistance***: A rossz párosítások elutasításához távolság alapú algoritmus használatával a PCL által biztosított *CorrespondenceRejectorDistance* osztályt használjuk. Itt nem kell megadni template paramétereket, mivel a párosítások tartalmazzák a közöttük lévő távolságokat. Paraméterként a *setMaximumDistance* segítségével megadhatjuk az algoritmus által használt távolság küszöböt, valamint a *setInputCorrespondences,* illetve a *getCorrespondences* segítségével a bemeneti párosításokat, illetve azt az objektumot, ahol szeretnénk, hogy a kimeneti párosítások legyenek a számítást követően.

**(3)** ***pcl::registration::CorrespondenceRejectorOneToOne:*** A rossz párosítok „Egyhez egy" elvet követő algoritmus felhasználásával történő elutasításához a *CorrespondenceRejectorOneToOne* osztályt használjuk. Használt metódusai megegyeznek a *CorrespondenceRejectorDistance* típusával. A kimeneti párosítások a bemeneti párosításokat fogják tartalmazni a szűrt párosítások kivételével.

**(4)** ***pcl::registration::CorrespondenceRejectorSurfaceNormal:*** A rossz párosítások felületi normák vizsgálatával történő elutasításához a *CorrespondenceRejectorSurfaceNormal* osztályt használjuk. A megvalósított típushoz a paraméterek többségét template metódusokkal tudjuk megadni. Az algoritmus számítása közben az adatok tárolására szükséges típusokat az osztály *initializeDataContainer*<T1, T2> metódus template paramétereiben adhatjuk meg. A mi esetünkben ezek rendre *pcl::PointXYZ*, illetve *pcl::Normal* típusok lesznek. Az algoritmus implementálása miatt a legtöbb függvényt template paraméterrel kell meghívni ennél az osztálynál. Az adat tárolók inicializálására az *initializeDataContainer <PointIn, PointN >* metódus szolgál, a bemeneti felhők megadására a *setInputSource<PointIn>,* illetve a *setInputTarget*<PointIn> metódusok szolgálnak. A normál felhőket a *setInputNormals<PointIn, PointN>*, illetve *a setTargetNormals<PointIn, PointN>*

metódusok segítségével számolhatjuk ki. A bemeneti értékeket minden esetben paraméterként adjuk át. A párosításokat, amiket módosítani szeretnénk a *setInputCorrespondences* metódus segítségével adhatjuk át, a végeredményt pedig a *getCorrespondences* paraméterében megadott, ennek tárolására alkalmas objektumba kerülnek. Az utóbbi függvény kezdi meg a tényleges számolást. A fejlesztés során a *pcl::PointXYZ,* illetve *pcl::Normal* pont típusokat rendeltünk a PointIn, illetve a PointN *template* paraméterekhez.

### *A transzformáció megvalósításához használt osztályok és azok használata [27]*

**(1)** ***pcl::registration::TransformationEstimationSVD< PointSource, PointTarget >*** : Az SVD alapú, pont-pont hibametrikát használó transzformációs becsléshez a PCL által biztosított *TransformationEstimationSVD* osztályt használjuk. A template paraméterek rendre a bemeneti forrásfelhő típusa, amit transzformálni szeretnénk, és a kimeneti felhő típusa, amire szeretnénk transzformálni. A mi esetünkben mind a kettő pcl::PointXYZ típusú lesz. Az objektum létrehozása után nincs más dolgunk, mint meghívni az *estimateRigidTransformation* metódusát a bemeneti(kulcspontok) és a kimeneti felhővel, a párosításokkal, valamint a transzformációs mátrix-szal, amiben az eredményt szeretnénk tárolni.

**(2)** ***pcl::registration::TransformationEstimationLM<PointSource, PointTarget >***: Az LM alapú, pont-felület hibametrikát használó transzformációs becsléshez a *TransformationEstimationLM* osztályt használjuk. Template paraméterei, illetve a számításhoz használt függvény neve, valamint azok paraméterei megegyeznek az SVD algoritmus használatánál olvasottakkal.

A transzformációk becsléséhez, illetve a tényleges transzformáció elvégzéséhez használt transzformációs mátrix az Eigen könyvtárból használt Eigen::Matrix4f típusú mátrix, ami egy 4x4-es float típusú elemeket tartalmazó mátrixot valósít meg.

### *Ritkításhoz használt osztályok és azok használata [28]*

Ezek az osztályok a PCL-nek a *filters* alkönyvtárában vannak implementálva.

**(1)** ***pcl::VoxelGrid< PointT >:*** A Voxel ráccsal történő ritkításhoz a PCL által megvalósított *VoxelGrid* osztályt használjuk. Template paraméterében értelemszerűen a felhő típusa kerül, amelynek a pontjait ritkítani szeretnénk. A *setLeafSize* metódus segítségével megadjuk a ritkításhoz szükséges paramétert. A bemeneti felhő a *setInputCloud* függvény paramétereként adható át. A ritkítást a *filter* metódus fogja kezdeményezni, aminek paraméterében az eredményfelhőt adjuk meg.

**(2)** ***pcl::RandomSample< PointT >:*** A véletlenszerű mintázással történő ritkításhoz a PCL által biztosított *RandomSample* osztályt használjuk. Paraméterezése a VoxelGrid-hez hasonlóan történik, azzal a különbséggel, hogy a mintázáshoz szükséges paramétert a *setSample* metódussal adjuk meg.

### *Kiugró pontok eltávolításához használt osztályok és azok használata [28]*

Ezek az osztályok a PCL-nek szintén a *filters* alkönyvtárában vannak implementálva. Mindkét típus esetében a bemeneti felhő megadása, illetve az algoritmus elindítását biztosító függvények a *setInputCloud*, illetve a filter metódus, amelyeknek a paramétereiben megadjuk a forrás-, illetve a célfelhőt (ahova az eredményt várjuk).

***pcl::StatisticalOutlierRemoval<PointT>***: A statisztikailag kiugró pontok eltávolításához a PCL-ben implementált *StatisticalOutlierRemoval* osztályt használjuk, aminek template paraméterében a szűrni kívánt pontfelhő típusát adjuk meg (a mi esetünkben ez pcl::PointXYZ lesz). Különböző *Setter* függvényeinek átadjuk az algoritmushoz használt paraméterértékeket. Például a *setMeanK* segítségével a vizsgált szomszédok száma adható meg, a *setStddevMulThres* metódussal pedig az algoritmusban szereplő határérték szorzója.

***pcl::RadiusOutlierRemoval<PointT>:*** A sugár vizsgálatával történő kiugró pontok keresését és eltávolítását a *RadiusOutlierRemoval* osztállyal végezzük el. Template paraméterként itt is a bemeneti pontfelhő típusát adjuk meg (az előzőhöz hasonlóan is pcl::PointXYZ típusú felhőket vizsgálunk). A *setRadiusSearch*, illetve a *setMinNeighborsInRadius* paraméterek segítségével az algoritmushoz használt speciális paramétereket állítjuk be. Ilyen speciális paraméter például a kereső sugár nagysága, illetve az ezen sugárban lévő minimális szomszédsági számot megadó kritérium.

### *Simításhoz használt osztály és használata [29]*

***pcl::MovingLeastSquare< PointInT, PointOutT >:*** Az MLS algoritmussal történő pontfelhő simításához a PCL-nek a *surface* alkönyvtárában implementált **MovingLeastSquare** osztályt használjuk, aminek template paramétereit a bemeneti, illetve az eredmény felhő típusa alapján adjuk meg. Ezek rendre pcl::PointXYZ típusok lesznek. A bemeneti felhőt a *setInputCloud*, az eredmény felhőt pedig a tényleges számítást elindító függvény (*process*) paraméterében adjuk át. További paraméterként beállítható az, hogy az algoritmus számoljon-e normákat vagy sem (*setComputeNormals segítségével*). Azt, hogy a folyamat során polinom közelítéssel becsüljön-e az algoritmus, a *setPolynomialFit* metódus segítségével adjuk meg. A kereső algoritmust, illetve a kereséshez használt sugarat is megadjuk, rendre a *setSearchMethod*, valamint a *setSearchRadius* metódusok segítségével.

### *Regisztrációs algoritmusokhoz használt további PCL osztályok és azok használatának bemutatása [27]*

***pcl::IterativeClosestPoint< PointSource, PointTarget >*** : Az ICP alapú regisztráláshoz a PCL által biztosított registration alkönyvtárában implementált *IterativeClosestPoint* template osztályt használjuk. A template paraméterei rendre a regisztráláshoz szükséges forrásfelhő típusa, illetve a célfelhő típusa. A mi esetünkben ezek pcl::PointXYZ típusúak lesznek. Ezeket a bemeneti felhőket, amiknek a típusait megadtuk, a *setInputSource*, illetve a *setInputTarget* paramétereinek segítségével adhatjuk meg. Az osztály számos egyéni beállítást tesz lehetővé, amelyeket a különböző Setter függvények segítségével állíthatunk be. Megadjuk például a maximum ICP iterálási számot (*setMaxIterations*), a rossz párosítások elutasításához használt, RANSAC-alapú elutasító maximális iterálási számát (*setRANSACIterations*), és a RANSAC belső pont vizsgálatához tartozó határértéket (*setRANSACOutlierRejectionThreshold*). Ezeken kívül beállítjuk a megállási kritériumot befolyásoló határértéket (*setEuclideanFitnessEpsilon*) és a párosításokhoz használt maximum távolságot (*setMaxCorrespondenceDistance*) is.

***pcl::SampleConsensusPrerejective<PointSource, PointTarget, FeatureT>*** : Az előzetes elutasítással kibővített RANSAC alapú algoritmushoz a PCL registration alkönyvtárában

implementált *SampleConsensusPrerejective* template osztályt használjuk. A template paraméterek rendre a bemeneti forrásfelhő típusa, a célfelhő típusa, valamint a párosítani kívánt feature vektorok elemeinek típusa. A bemeneti felhőket, a feature vektorokat, valamint a regisztráció finomítására szolgáló paramétereket a megfelelő *Setter* függvényekkel adjuk meg. A bemeneti forrásfelhőt és célfelhőt a *setInputSource*, illetve a *setInputTarget* segítségével, az egyes feature vektorokat pedig a *setSourceFeatures*, illetve a *setTargetFeatures* segítségével adjuk át. Az *align* metódussal kezdhetjük meg a transzformáció kiszámítását, és ennek paraméterében a kimeneti felhőt adjuk meg. A regisztráció finomítására szolgáló egyéb paramétereket is beállítunk. Ezeket a paramétereket például a *setMaxIterations* (maximum RANSAC iterálások száma), *setInlierFraction* (a megkövetelt belső pontok aránya, amivel elfogad az algoritmus egy hipotézist) segítségével adjuk meg.

### *Megjelenítéshez használt PCL osztályok és azok használata [30]*

***pcl::visualization::PCLVisualizer***: A pontfelhők vizualizálásához a PCL visualization alkönyvtárában implementált *PCLVisualizer* osztályt használjuk. A megvalósított osztály segítségével jelenítjük meg az egyes felhők pontjait, kulcspontjait, valamint szemléltetjük a regisztráció, illetve az előfeldolgozás eredményeit. Lehetőségünk van a párosítások ábrázolására is. A felhő hozzáadását az *addPointCLoud<PointT>* template metódusával végezhetjük el, amely paraméterül a bemeneti felhőt, egy színkezelőt (amivel a hozzáadni kívánt felhő színét határozzuk meg), valamint egy azonosító szöveget kap, és a mi esetünkben egy megjelenítőn belüli, belső ablakrészhez tartozó azonosítót is megadunk. A *setPointCloudRenderingProperties* segítségével egyéb tulajdonságait határozhatjuk meg a megjelenített felhőnknek, például a pontok méretét (a kulcspontokat érdemes kiemelni).

***pcl::visualization::PointCloudColorHandlerCustom<PointT>:*** Amikor a vizualizációs panelhez pontfelhőt adunk meg, ennek az osztálynak a segítségével határozzuk meg, hogy milyen színű legyen az adott pontfelhő. Template paraméterében a felhő típusát adjuk meg, és konstruktorában a bemeneti felhőt (amit hozzá szeretnénk adni a vizualizációs elemhez), valamint az RGB színmodell által meghatározott három értéket sorrendben (Red, Green, Blue).

# A program egyes osztályainak megvalósításának módszerei

A következőkben az egyes osztályok megvalósításának módszerei lesznek kifejtve röviden. A fejezet kitér arra, hogy az egyes ablakok megjelenítése milyen osztályok, technikák segítségével történt.

## Fő ablak és az ahhoz tartozó PCLViewerX osztályának megvalósítása

A fő ablakot egy *QMainWindow*-ból leszármaztatott felhasználói felülettel (UI) ellátott osztállyal, a *PCLViewerX* segítségével valósítjuk meg. Ennek felhasználói felületén helyezzük el a különböző Qt által biztosított elemeket. Az elemek elrendezéséhez *QGridLayout* felületet használunk, aminek segítségével rácsos szerkezetet adhatunk a felületnek. Az egyes rádiógombokat, amiknek használatával kiválaszthatjuk a regisztráció típusát, *QRadioButton* osztállyal valósítjuk meg. A program egyes funkcióinak (regisztrációs ablak, előfeldolgozási ablak lehívása, egyes felhők beolvasása / törlése) meghívásának kiváltásáért felelős eszközöket *QPushButton* típusú, egyszerű nyomógombok használatával oldja meg a szoftver. A szöveges panelt, ami arra szolgál, hogy a felhasználóval folyamatosan kommunikáljon, az esetleges hibákról visszajelzést adjon, az eredményekről tájékoztasson, a *QPlainTextEdit* osztály példányosításával valósítottuk meg. Ez az osztály egyszerű használatot biztosít mind a felhasználónak, mind a fejlesztőnek, hisz az alkalmazás futásának végéig a felhasználónak biztosított üzenetek megmaradnak, és a fejlesztőnek sem kell törődnie azzal, hogy megtelik-e esetleg az ablak, hisz egy ekkor egy automatikus csúszka jelenik meg, ezzel keresést biztosítva az esetleges hosszú szövegekben. A felhőket megjelenítő panel elkészítéséhez a *QVTKWidget* típus jelent megoldást, hisz ennek segítségével a PCL-ben a felhők megjelenítésére biztosított *PCLVisualizer* típusú objektumot össze tudjuk kapcsolni a saját *QVTKWidget* objektumunkkal. Ez a *QVTKWidget* SetRenderWindow, illetve a *SetupInteractor* metódusainak használatával történt. A *QVTKWidget* a VTK nyílt forrású szoftver rendszeréből származik, ami különböző lehetőséget biztosít vizualizációkhoz, valamint egyéb grafikus megjelenítésekhez.

A PCLViewerX attribútumai között találjuk meg a többi dialógus ablakot megvalósító objektumot, illetve a modellt is. Feladatai közé tartoznak a felhők funkciótól függő megjelenítése, az egyes üzenetek folyamatos közlése a felhasználóval, valamint a

kommunikáció biztosítása a felhasználó és a modell között. Ezeken kívül a dialógusablakok kezeléséért is felelős az osztály. Meghívja őket, majd az eredményüket továbbítja a modell megfelelő metódusainak.

## Dialógusablakok, és a hozzájuk tartozó osztályok megvalósítása

Az egyes dialógus ablakok közé tartoznak a ICP-alapú regisztrációhoz, a RANSAC-alapú regisztrációhoz, a feature-alapú regisztrációhoz tartozó dialógus ablakok, valamint az előfeldolgozáshoz tartozó dialógus ablak is. Ezek egy-egy QDialog-ból leszármaztatott osztály segítségével lettek megvalósítva, az ICPRegistrationParamsDialog, a RANSACRegistrationParamsDialog, a FeatureBasedRegistrationParamsdialog, illetve a PreprocessingParamsDialog osztályokkal. Az egyes paraméterek megadására szolgáló elemeket a *QDoubleSpinBox* típus segítségével adjuk meg, ezekkel double típusú értékeket vagyunk képesek megadni és eltárolni a felhasználói felületen keresztül. A paraméterekhez tartozó definíciókat *QLabel* segítségével valósítjuk meg, ami egy egyszerű szövegmezőt reprezentál. Az egyes módszerek kiválasztását *QRadioButton* segítségével biztosítjuk, amit már a fő ablaknál láthattunk. Az ezeket az elemeket tartalmazó „csoportokat” (az ICP és RANSAC regisztrációkhoz tartozó dialógus ablakok esetén csak egy-egy ilyen csoport van) *QGroupBox* segítségével valósítjuk meg a rendszerezés miatt. A csoportokon belül *QFormLayout* segítségével helyezzük el az egyes elemeket az elegáns elrendezés érdekében. A felhasználóbarát felület megvalósítása miatt csúsztatható ablakban jelenítjük meg az egyes dialógusablakokat annak érdekében, hogy kis képernyő esetén is használható legyen az alkalmazás. Ezt a *QScrollArea* osztály segítségével valósítjuk meg. Az egyes osztályok esetén a különböző modulok eltárolására (*QLabel*, *DoubleSpinBox*, *Checkbox*, *RadioButton*), a *Standard Library* gyűjteménybeli *map* típusú tárolókat használjuk, amelyek segítségével könnyedén elérhetjük az egyes paraméterekhez tartozó azonosító alapján a megfelelő modulokat.

## A model, vagyis a PCLViewerXModel osztály megvalósítása

A PCLViewerXModel a QWidget-ből leszármaztatott osztályként lett megvalósítva. Adattagjai közé tartoznak azok a pontfelhő típusú objektumok, amikkel számolunk az egyes műveletek során. Ide tartozik még a transzformációs mátrix és a párosításokat

tartalmazó objektum is. Metódusai közül csak azok nyilvánosak, amelyekkel a nézet közvetlen kommunikál. Ezek a metódusok végzik az egyes PCL-alapú felhő műveleteket, számításokat, és ezek többségét az egyes regisztrációs, előfeldolgozó függvények hívják meg. A feature-alapú regisztrációs metódus (*FeatureBasedRegistration*), valamint az előfeldolgozó metódus (Preprocessing) tartoznak az osztály legösszetettebb függvényei közé. Ezek a függvények a View-tól megkapott paraméterek alapján választják ki, hogy melyik további metódusokat hívják meg, miközben folyamatos hiba ellenőrzést végeznek. A *FeatureBasedRegistration* fogja meghatározni a feature-alapú regisztrációs sémát, míg a *Preprocessing* metódus pedig az előfeldolgozás sémáját határozza meg a bejövő paraméterek alapján. A modell az egyes számítások esetén egyéni események segítségével jelzi a kapott eredményeket a View-nak, ami ezáltal képes folyamatosan megjeleníteni a kapott értékeket az erre megfelelő szöveges megjelenítőjének felületén, a hozzá tartozó eseménykezelője használatával.

## Későbbi esetleges fejlesztések

Számos fejlesztési lehetőség van még tervben a későbbiekben, amennyiben igény van rá. Ezek között van olyan, ami a felhasználóbarát tulajdonságát fejleszti a programnak, és van, ami plusz szolgáltatást nyújt és kiterjeszti az alkalmazás használati területeit. A fejezet során ilyen fejlesztéseket említünk meg.

- Felület továbbfejlesztése, modernizálása.
- További funkciók bevezetése (paraméter keresés, felhő optimalizáció, felhőkön való transzformációk elvégzése).
- További algoritmusok alkalmazása a regisztrálási folyamat egyes lépései esetén.
- További algoritmusok alkalmazása az előfeldolgozási folyamat lehetőségei esetén.
- Az egyes algoritmusokhoz további paraméterek megismerése és bevezetése a programba.

## Tesztelés

A **modell** tesztelése CLion integrált fejlesztői környezet, valamint Google Test tesztelő könyvtár segítségével valósult meg, egység tesztelő módszerrel.

A Google Test egy egység tesztelő könyvtár C++ programozási nyelvekhez, ami lehetőséget biztosít a tesztek párhuzamos futására is. Mivel a PCLViewerXModel által megvalósított modell metódusai közül többen is az osztály egyazon adattagján dolgoznak, ezért a párhuzamos tesztelés nem hatékony a mi esetünkben. Az egység tesztelést megvalósító PCLViewerXModelTesting osztályt a Google Test által implementált Test osztályából származtattuk le. Az osztályban szereplő egyes funkciókat kategóriák szerint soroltuk be és különböző szempontok szerint teszteltük, különböző bemenetekkel.

## A modell egység tesztjei

### *A bemeneti felhők betöltéséért felelős függvények tesztelése*

Ezeknek a függvényeknek a tesztelését a „child1.pcd”, „child2.pcd”, „cloud_bin_0.pcd”, illetve a „cloud_bin_1.pcd” felhők segítségével végeztük.

**LoadSrcCloud tesztelése:**

1. ***Teszt***:
   - **Bemeneti paraméter**: „” (üres szöveg)
   - **Kimenet**: -1. A függvény hibát jelez, ha nem létező fájlt szeretnénk beolvasni.
2. ***Teszt***:
   - **Bemeneti paraméter**: „child1.pcd” (létező felhő megadva)
   - **Kimenet**: 0. A függvény ezzel jelzi a helyes működést Ekkor a bemeneti forrásfelhő elemszámának vizsgálatával ellenőrizzük, hogy minden értéket helyesen beolvasott-e.
3. ***Teszt***:
   - **Bemeneti paraméter**: „cloud_bin_0.pcd” (nagy pontmennyiséget tartalmazó felhő)
   - **Kimenet**: 0. A függvény ezzel jelzi a helyes működést. Ekkor a bemeneti forrásfelhő elemszámának vizsgálatával ellenőrizzük, hogy minden értéket helyesen beolvasott-e.

**LoadTgtCloud tesztelése:**

4. ***Teszt***:

- **Bemeneti paraméter**: „” (üres szöveg)
- **Kimenet**: -1. A függvény hibát jelez, ha nem létező fájlt szeretnénk beolvasni.

5. ***Teszt***:
   - **Bemeneti paraméter**: „child1.pcd” (létező felhő megadva)
   - **Kimenet**: 0. A függvény ezzel jelzi a helyes működést. Ekkor a célfelhő elemszámának vizsgálatával ellenőrizzük, hogy minden értéket beolvasott-e.
6. ***Teszt***:
   - **Bemeneti paraméter**: „cloud_bin_1.pcd” (nagy pontmennyiséget tartalmazó felhő)
   - **Kimeneti**: 0. A függvény ezzel jelzi a helyes működést. Ekkor a célfelhő elemszámának vizsgálatával ellenőrizzük, hogy minden értéket helyesen beolvasott-e.

### *A normál számításért felelős függvény tesztelése*

Előre beolvasott felhőkhöz (child1.pcd, cloud_bin_0.pcd) számoljuk ki a pontokhoz tartozó normálértékeket.

**EstimateNormals függvény tesztelése már előre beolvasott felhőkön:**

1. ***Teszt***:
   - **Bemeneti paraméterek**: normál méretű forrásfelhő, normál méretű forrásfelhő, a normák tárolására alkalmas felhő, 32
   - **Kimeneti**: 0. A függvény hiba nélkül lefutott és az egyes normál felhők nem tartalmaznak NaN értékeket.
2. ***Teszt***:
   - **Bemeneti paraméterek**: nagyméretű forrásfelhő, nagyméretű forrásfelhő, normák tárolására alkalmas felhő, 32
   - **Kimeneti**: 0. A függvény hiba nélkül lefutott és az egyes normál felhők nem tartalmaznak NaN értékeket.

### A kulcspontok kiszámításáért felelős függvények tesztelése

Az előre beolvasott felhőkhöz (child1.pcd /cloud_bin_0.pcd) számítunk kulcspontokat különböző módszerekkel.

**EstimateKeypointsBasedOnHarrisKeyPoint3d függvény tesztelése**

1. ***Teszt***:
   - **Bemeneti paraméterek**: a beolvasott normál nagyságú felhő, a kulcspontok tárolására alkalmas felhő, 0.000001, false
   - **Kimenet**: 0. Ezt követően megvizsgáljuk, hogy a kulcspontokhoz megadott felhőben valóban szerepelnek-e értékek.
2. ***Teszt:***
   - **Bemeneti paraméterek**: a beolvasott sok pontot tartalmaz felhő, a kulcspontok tárolására alkalmas felhő, 0.000001, false
   - **Kimenet**: 0. Ezt követően megvizsgáljuk, hogy a kulcspontokhoz megadott felhőben valóban szerepelnek-e értékek.

**EstimateKeypointsBasedOnISS függvény tesztelése**

3. ***Teszt:***
   - **Bemeneti paraméterek**: a beolvasott normál nagyságú felhő, a kulcspontok tárolására alkalmas felhő, 0.98, 0.98, 2, 4, 6
   - **Kimeneti érték**: 0. Ezt követően megvizsgáljuk, hogy a kulcspontokhoz megadott felhőben valóban szerepelnek-e értékek.
4. ***Teszt***:
   - **Bemeneti paraméterek**: a beolvasott sok pontot tartalmaz felhő, a kulcspontok tárolására alkalmas felhő, 0.98, 0.98, 3,4,6)
   - **Kimeneti**: 0. Ezt követően megvizsgáljuk, hogy a kulcspontokhoz megadott felhőben valóban szerepelnek-e értékek.

**EstimateKeypointsBasedOnSIFT függvény tesztelése**

5. ***Teszt***:
   - **Bemeneti paraméterek**: a beolvasott normál nagyságú felhő, a kulcspontok tárolására alkalmas felhő, 0.001, 12, 2, 0.0001
   - **Kimenet**: 0. Ezt követően megvizsgáljuk, hogy a kulcspontokhoz megadott felhőben valóban szerepelnek-e értékek.
6. ***Teszt***

- **Bemeneti paraméterek**: a beolvasott sok pontot tartalmaz felhő, a kulcspontok tárolására alkalmas felhő, 0.001, 12, 2, 0.0001
- **Kimenet**: 0. Ezt követően megvizsgáljuk, hogy a kulcspontokhoz megadott felhőben valóban szerepelnek-e értékek.

### *A feature leírók becsléséért felelős függvények tesztelése*

Ekkor előre beolvasott felhőn (bunny.pcd, bunny2.pcd) számoljuk ki az egyes leírókat. Ehhez a felhőhöz a normák már ki vannak számítva.

**EstimateFPFH függvény tesztelése**

1. ***Teszt***:
   - **Bemeneti paraméterek**: a beolvasott felhő, a normákat tartalmazó felhő, az adott típusú feature leírók tárolására alkalmas felhő, 6
   - **Kimenet**: 0, vagyis nem történt hiba. Ezt követően megvizsgáljuk, hogy a felhő pontjainak száma, amihez kiszámoltuk a leírókat, valamint maga a feature leírókat tartalmazó felhő pontjainak a száma megegyezik-e.

**EstimatePFH függvény tesztelése**

2. ***Teszt***:
   - **Bemeneti paraméterek**: a beolvasott felhő, a normákat tartalmazó felhő, az adott típusú feature leírók tárolására alkalmas felhő, 6
   - **Kimenet**: 0, vagyis nem történt hiba. Ezt követően megvizsgáljuk, hogy a felhő pontjainak száma, amihez kiszámoltuk a leírókat, valamint maga a feature leírókat tartalmazó felhő pontjainak a száma megegyezik-e.

**EstimateSHOT függvény tesztelése**

3. ***Teszt***:
   - **Bemeneti paraméterek**: a beolvasott felhő, a normákat tartalmazó felhő, az adott típusú feature leírók tárolására alkalmas felhő, 6
   - Kimenet: 0. Ezt követően megvizsgáljuk, hogy a felhő pontjainak száma, amihez kiszámoltuk a leírókat, valamint maga a feature leírókat tartalmazó felhő pontjainak a száma megegyezik-e.

### *Párosítások keresésének tesztelése*

Ekkor előre beolvasott felhőkön (bunny.pcd, bunny2.pcd), előre kiszámolt leírókkal, normákkal teszteljük a függvényt.

**FindCorrespondences függvény tesztelése**

1. ***Teszt***:
   - **Bemeneti paraméter**: true
   - **Kimenet**: 0, tehát nem történt hiba és a párosítások száma nagyobb, mint 4. Ezt követően megvizsgáljuk, hogy a párosítások száma megegyezik-e a forrásfelhő pontjainak számával (az algoritmusból adódóan).
2. ***Teszt***:
   - **Bemeneti paraméter**: false
   - **Kimenet**: 0, tehát nem történt hiba és a párosítások száma nagyobb, mint 4. Ezt követően megvizsgáljuk, hogy a párosítások száma különbözik-e a forrásfelhő pontjainak számával (az algoritmusból adódóan).

### *Rossz párosítások elutasításának tesztelése*

Ekkor előre beolvasott felhőkön (bunny.pcd, bunny2.pcd), előre közvetlen kiszámolt párosításokon teszteljük az egyes függvényeket.

**RejectBadCorresespondencesRANSAC függvény tesztelése**

1. ***Teszt***:
   - **Bemeneti paraméterek**: 0.01, 500
   - **Kimenet**: 0. Tehát nem történt hiba a működés során és 4-nél több párosítás is maradt, ami szükséges a későbbi regisztrációs műveletekhez. Ezt követően megvizsgáltuk, hogy valóban kevesebb párosítást kaptunk-e, mint amennyi volt.

**RejectBadCorrespondencesDistance függvény tesztelése**

2. ***Teszt***:
   - **Bemeneti paraméterek**: 0.01
   - **Kimenet**: 0. Tehát nem történt hiba a működés során és 4-nél több párosítás is maradt, ami szükséges a későbbi regisztrációs műveletekhez. Ezt követően

megvizsgáltuk, hogy valóban kevesebb párosítást kaptunk-e, mint amennyi volt.

**RejectBadCorrespondencesOneToOne függvény tesztelése**

3. ***Teszt***:
   - **Bemeneti paraméterek**: -
   - **Kimenet**: 0. Tehát nem történt hiba a működés során és 4-nél több párosítás is maradt, ami szükséges a későbbi regisztrációs műveletekhez. Ezt követően megvizsgáltuk, hogy valóban kevesebb párosítást kaptunk-e, mint amennyi volt.

**RejectBadCorrespondencesSurfaceNormals függvény tesztelése**

4. ***Teszt***:
   - **Bemeneti paraméterek**: 0.9, 5
   - **Kimenet**: 0. Tehát nem történt hiba a működés során és 4-nél több párosítás is maradt, ami szükséges a későbbi regisztrációs műveletekhez. Ezt követően megvizsgáltuk, hogy valóban kevesebb párosítást kaptunk-e, mint amennyi volt.

### *Az előfeldolgozásért felelős függvények tesztelése*

Ekkor egy előre beolvasott felhőn teszteljük a felhőt (child1.pcd /cloud_bin_0).

**OutliersRemovalBasedOnStatistics függvény tesztelése**

1. ***Teszt***:
   - **Bemeneti paraméterek**: normál méretű bemeneti felhő, normál méretű bemeneti felhő, 25, 0.4, true
   - **Kimenet**: 0. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy valóban kevesebb pontot kaptunk-e, mint amennyit eredetileg tartalmazott a felhő.
2. ***Teszt***:
   - **Bemeneti paraméterek**: nagyméretű bemeneti felhő, nagyméretű bemeneti felhő, 25, 0.4, true

- **Kimenet**: 0. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy valóban kevesebb pontot kaptunk-e, mint amennyit eredetileg tartalmazott a felhő.

**OutliersRemovalBasedOnRadius függvény tesztelése**

3. ***Teszt***:
   - **Bemeneti paraméterek**: normál méretű bemeneti felhő, normál méretű bemeneti felhő, 80, 0.05
   - **Kimenet**: 0. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy valóban kevesebb pontot kaptunk-e, mint amennyit eredetileg tartalmazott a felhő.
4. ***Teszt***:
   - **Bemeneti paraméterek**: nagyméretű bemeneti felhő, nagyméretű bemeneti felhő, 25, 0.4, true
   - **Kimenet**: 0. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy valóban kevesebb pontot kaptunk-e, mint amennyit eredetileg tartalmazott a felhő.

**SmoothingBasedOnMovingLeastSquares függvény tesztelése**

5. ***Teszt***:
   - **Bemeneti paraméterek**: normál méretű bemeneti felhő, normál méretű bemeneti felhő, 80, 0.05
   - **Kimenet**: 0. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy változtak-e a pontok elhelyezkedései a felhőben.
6. ***Teszt***:
   - **Bemeneti paraméterek**: nagyméretűbemeneti felhő, nagyméretű bemeneti felhő, 80, 0.03
   - **Kimenet**: 0. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy változtak-e a pontok elhelyezkedései a felhőben.

**DownSamplingBasedOnVoxelGrid függvény tesztelése**

7. ***Teszt***:
   - **Bemeneti paraméterek**: bemeneti felhő, bemeneti felhő, 0.04
   - **Kimenet**: 0. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy valóban kevesebb pontot kaptunk-e az új felhőben, mint amennyi volt az eredetiben.
8. ***Teszt***:
   - **Bemeneti paraméterek**: nagyméretű bemeneti felhő, nagyméretű bemeneti felhő, 0.04
   - **Kimenet**: 0. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy valóban kevesebb pontot kaptunk-e az új felhőben, mint amennyi volt az eredetiben.

**DownSamplingBasedOnRandomRampling függvény tesztelése**

9. ***Teszt***:
   - **Bemeneti paraméterek**: normál méretű bemeneti felhő, normál méretű bemeneti felhő, 3000
   - **Kimenet**: 0. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy a ritkított felhőben 3000 pont található-e (az algoritmusból adódóan).
10. ***Teszt***:
    - **Bemeneti paraméterek**: nagyméretű bemeneti felhő, nagyméretű bemeneti felhő, 3000
    - **Kimenet**: 0. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy a ritkított felhőben 3000 pont található-e (az algoritmusból adódóan).

### *Pontfelhő tulajdonságait kiszámoló függvények tesztelése*

Ezek vizsgálata előre beolvasott felhőkön történnek (child1.pcd / cloud_bin_0.pcd).

**ComputeCloudDiameter függvény tesztelése**

1. ***Teszt***:
   - **Bemeneti paraméterek**: normál méretű bemeneti felhő

- **Kimenet**: a kiszámolt átmérő. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy ez az átmérő 2 és 2.5 között van-e (előre ismert adatok).

2. ***Teszt***:
   - **Bemeneti paraméterek**: nagyméretű bemeneti felhő
   - **Kimenet**: a kiszámolt átmérő. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy ez az átmérő 3 és 4 között van-e (előre ismert adatok).

**ComputeCloudResolution függvény tesztelése**

3. ***Teszt***:
   - **Bemeneti paraméterek**: normál méretű bemeneti felhő
   - **Kimenet**: a kiszámolt átmérő. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy ez az érték 0 és 0.01 között van-e (előre ismert adatok).
4. ***Teszt***:
   - **Bemeneti paraméterek**: nagyméretű bemeneti felhő
   - **Kimenet**: a kiszámolt átmérő. Tehát nem történt hiba a működés során. Ezt követően megvizsgáltuk, hogy ez az érték 0 és 0.008 között van-e (előre ismert adatok).

### *Regisztrációk végeredményét kiszámító függvény tesztelése*

**ComputeFinalScore függvény tesztelése**

1. ***Teszt***:
   - **Bemeneti paraméterek**: normál méretű bemeneti felhő, ugyanaz a normál méretű felhő
   - **Kimenet**: a kiszámolt pontok közötti átlagos távolság a két felhő között. Ezt követően megvizsgáltuk, hogy ez az érték 0-hoz közelít-e.
2. ***Teszt***:
   - **Bemeneti paraméterek**: nagyméretű bemeneti felhő, ugyanaz a nagyméretű felhő

- **Kimenet**: a kiszámolt pontok közötti átlagos távolság a két felhő között. Ezt követően megvizsgáltuk, hogy ez az érték 0-hoz közelít-e.

### *Az előfeldolgozott felhők törlését biztosító függvények tesztelése*

Ekkor előre beolvasott felhőkkel vizsgáljuk a függvényeket (child1.pcd, child2.pcd / cloud_bin_0.pcd, cloud_bin_1.pcd).

**DeletePreprocessedSource tesztelése**

1. ***Teszt***:
   - **Bemeneti paraméterek**: -

     A forrásfelhőhöz az egyik normál méretű felhőt olvassuk be, az előfeldolgozott forrásfelhőhöz pedig a másik normál méretű felhőt. A két felhő mérete nem egyezik meg.
   - **Kimenet**: -. A folyamat végén a két felhő mérete megegyezik.
2. ***Teszt***:
   - **Bemeneti paraméterek**: -

     A forrásfelhőhöz az egyik nagyméretű felhőt olvassuk be, az előfeldolgozott forrásfelhőhöz pedig a másik nagyméretű felhőt. A két felhő mérete nem egyezik meg.
   - **Kimenet**: -. A folyamat végén a két felhő mérete megegyezik.

**DeletePreprocessedTarget tesztelése**

3. ***Teszt***:
   - **Bemeneti paraméterek**: -

     A célfelhőhöz az egyik normál méretű felhőt olvassuk be, az előfeldolgozott célfelhőhöz pedig a másik normál méretű felhőt. A két felhő mérete nem egyezik meg.
   - **Kimenet**: -. A folyamat végén a két felhő mérete megegyezik.
4. ***Teszt***:
   - **Bemeneti paraméterek**: -

A célfelhőhöz az egyik nagyméretű felhőt olvassuk be, az előfeldolgozott célfelhőhöz pedig a másik nagyméretű felhőt. A két felhő mérete nem egyezik meg.

- **Kimenet**: -. A folyamat végén a két felhő mérete megegyezik.

### *Regisztrációs műveletek tesztelése*

A regisztrációs műveleteket létező, előre beolvasott felhők segítségével, valamint üres felhőkkel is teszteljük. A létező felhők esetén két felhő teljesen átfedi egymást. A célfelhőt úgy kaptuk meg, hogy a forráson alkalmaztunk egy bizonyos transzformációt. A tesztelés során azt vizsgáljuk, hogy visszakapjuk-e az adott transzformációs mátrixunkat.

**ICPRegistration függvény tesztelése**

1. ***Teszt*:** (normál méretű felhő esetén)
   - **Bemeneti paraméterek**: 2000, 0.07, 0.07, 2000, 0.00005
   - **Kimenet**: a regisztráció hibaértékét adja vissza. Megvizsgáljuk, hogy ez az érték 0 és 0.004 között van-e (kellően kis hiba). Ha ez teljesül, akkor az általa kiszámolt transzformációs mátrixot vizsgáljuk meg, hogy kisebb hibákkal, de megegyezik-e az ismert transzformációs mátrixszal, amit alkalmaztunk korábban a forrásfelhőn a célfelhőre.
2. ***Teszt***: (üres felhő esetén)
   - **Bemeneti paraméterek**: 2000, 0.07, 0.07, 2000, 0.00005
   - **Kimenet**: -1. A függvény jelzi, hogy hiba történt.

**RANSACRegistration függvény tesztelése**

3. ***Teszt***: (normál felhő mellett)
   - **Bemeneti paraméterek**: 0.01,3,3,50000,3,2,0.6,0.97,0.05
   - **Kimenet**: a regisztráció hibaértékét adja vissza. Megvizsgáljuk, hogy ez az érték 0 és 0.15 között van-e (kellően kis hiba). Ha ez teljesül, akkor az általa kiszámolt transzformációs mátrixot vizsgáljuk meg, hogy kisebb hibákkal, de megegyezik-e az ismert transzformációs mátrixszal, amit alkalmaztunk korábban a forrásfelhőn a célfelhőre.

4. ***Teszt***: (üres felhő esetén)
   - **Bemeneti paraméterek**: 0.01,3,3,50000,3,2,0.6,0.97,0.05
   - **Kimenet**: -1. A függvény jelzi, hogy hiba történt.

**FeatureBasedRegistration függvény tesztelése**

1. ***Teszt***: (normál felhő esetén)
   - **Bemeneti paraméterek**: true, false, false, false, true, true, false, true, false, true, 0.0005, 0.005, false, false, 0.9, 0.9, 0.9, 0.9, 3, 3, 3, 3, 3, 3, 12, 12, 4, 4, 0.01, 0.01, 0.001, 0.001, 4, 4, 0.07, 8000, 0.1, 0.6, 6
   - **Kimenet**: a regisztráció hibaértékét adja vissza. Megvizsgáljuk, hogy ez az érték 0 és 0.06 között van-e (kellően kis hiba). Ha ez teljesül, akkor az általa kiszámolt transzformációs mátrixot vizsgáljuk meg, hogy kisebb hibákkal, de megegyezik-e az ismert transzformációs mátrixszal, amit alkalmaztunk korábban a forrásfelhőn a célfelhőre.
2. ***Teszt***: (nagyméretű felhő esetén)
   - **Bemeneti paraméterek**: false, true, false, false, true, true, false, true, false, true, 0.0005, 0.005, false, false, 0.9, 0.9, 0.9, 0.9, 5, 5, 4, 4, 6, 6, 12, 12, 4, 4, 0.01, 0.01, 0.001, 0.001, 8, 8, 0.04, 8000, 0.1, 0.6, 6
   - **Kimenet**: a regisztráció hibaértékét adja vissza. Megvizsgáljuk, hogy ez az érték 0 és 0.06 között van-e (kellően kis hiba). Ha ez teljesül, akkor az általa kiszámolt transzformációs mátrixot vizsgáljuk meg, hogy kisebb hibákkal, de megegyezik-e az ismert transzformációs mátrixszal, amit alkalmaztunk korábban a forrásfelhőn a célfelhőre.
3. ***Teszt***: (üres felhő esetén)
   - **Bemeneti paraméterek**: true, false, false, false, true, true, false, true, false, true, 0.0005, 0.005, false, false, 0.9, 0.9, 0.9, 0.9, 3, 3, 3, 3, 3, 3, 12, 12, 4, 4, 0.01, 0.01, 0.001, 0.001, 4, 4, 0.07, 8000, 0.1, 0.6, 6
   - **Kimenet**: -7. A függvény jelzi, hogy hiba történt.

### *Előfeldolgozást végző művelet tesztelése*

Előre beolvasott felhőkön történik (child1.pcd, child2.pcd /cloud_bin_0.pcd, cloud_bin_1.pcd) kis és nagy felbontású felhők esetén. A teszt során vizsgáljuk, hogy

hogyan változik a felhők mérete. Üres felhő esetén történő vizsgálatnál pedig hibát várunk.

**Preprocessing függvény tesztelése:**

1. ***Teszt*** (normál méretű felhő beolvasása esetén):
   - **Bemeneti paraméterek**: true, true, false, true, true, true, false, false, false, true, false, 9000, 9000, 0.01, 0.01, 15, 0.01, false, 15, 0.1, false, 10, 0.1, 25, 0.06, 0.1, 0.08
   - **Kimenet**: 0. Tehát nem történt hiba a program működése során. Ezt követően megvizsgáljuk, hogy az így kapott feldolgozott felhő elemszáma különbözik-e a bemeneti felhőtől.
2. ***Teszt*** (nagyméretű felhő esetén):
   - **Bemeneti paraméterek**: true, true, false, true, true, true, false, false, false, true, false, 321, 4000, 0.01, 0.01, 15, 0.01, false, 15, 0.01, false, 20, 0.1, 25, 0.06, 0.1, 0.08
   - **Kimenet**: 0. Tehát nem történt hiba a program működése során. Ezt követően megvizsgáljuk, hogy az így kapott feldolgozott felhő elemszáma különbözik-e a bemeneti felhőtől.
3. ***Teszt*** (üres felhő esetén):
   - **Bemeneti paraméterek**: true, false, false, false, true, true, false, true, false, true, 0.0005, 0.005, false, false, 0.9, 0.9, 0.9, 0.9, 3, 3, 3, 3, 3, 3, 12, 12, 4, 4, 0.01, 0.01, 0.001, 0.001, 4, 4, 0.07, 8000, 0.1, 0.6, 6
   - **Kimenet**: -7. A függvény jelzi, hogy hiba történt.

## Következtetések

**A tesztelés miatt bekövetkezett módosítások**: kezdetben az egyes regisztrációs folyamatok nem ellenőrizték a bemeneti felhők helyességét, így üres felhő megadása esetén elszállt a program, így ezt ki kellett javítani a megvalósításban.

Egy regisztrációs folyamat helyességét a **ComputeFinalScore** függvény segítségével adhatjuk meg, ami a célfelhőt, és a regisztráció eredményéből keletkezett felhőket hasonlítja össze úgy, hogy az összes célon belüli ponthoz legközelebbi szomszédot keres, és az ezek közötti távolságot vizsgálja (teljesen átfedő felhők esetén pontos csak, de

összehasonlításnak megfelelő részben átfedő felhők esetén is), így a tesztelés során ez alapján vizsgáltuk meg, hogy egy-egy regisztráció mennyire helyes.

# *Irodalomjegyzék*


[1] *„**Point Set Registration - Wikipedia**” [Online]*

*Elérhető: https://en.wikipedia.org/wiki/Point_set_registration*

*[Hozzáférés dátuma: 2020.05.08]*

[2] *„**Downsampling a PointCloud using a VoxelGrid filter**” [Online]*

*Elérhető:*

*https://web.archive.org/web/20190908130915/http://docs.pointclouds.org/trunk/classpcl_1_1_approximate_voxel_grid.html#details [Hozzáférés dátuma: 2020.05.08]*

*[3] „**Vitter, Jeffrey. (1984). Faster Methods for Random Sampling.. Commun. ACM. 27. 703-718. 10.1145/358105.893.** " [Online]*

*Elérhető: http://www.ittc.ku.edu/~jsv/Papers/Vit84.sampling.pdf*

*[Hozzáférés dátuma: 2020.05.08]*

[4] **„Removing outliers using a StatisticalOutlierRemoval filter”** *[Online]*

*Elérhető:*

*https://web.archive.org/web/20191016180723/http://pointclouds.org/documentation/tutorials/statistical_outlier.php#statistical-outlier-removal*

*[Hozzáférés dátuma: 2020.05.08]*

[5] **„Removing outliers using a Conditional or RadiusOutlier removal”** *[Online]*

*Elérhető:*

*https://web.archive.org/web/20191102193048/http://pointclouds.org/documentation/tutorials/remove_outliers.php#remove-outliers [Hozzáférés dátuma: 2020.05.08]*

[6] **„Smoothing and normal estimation based on polynomial reconstruction”** *[Online]*

*Elérhető:*

*https://web.archive.org/web/20190915113842/http://www.pointclouds.org/documentation/tutorials/resampling.php#moving-least-squares*

*[Hozzáférés dátuma: 2020.05.08]*

[7] **„The PCL Registration API”** *[Online]*

*Elérhető:*
*https://web.archive.org/web/20191020171604/http://pointclouds.org/documentation/tutorials/registration_api.php#registration-api [Hozzáférés dátuma: 2020.05.08]*

[8] *„***D. Holz, A. E. Ichim, F. Tombari, R. B. Rusu and S. Behnke, "Registration with the Point Cloud Library: A Modular Framework for Aligning in 3-D," in *IEEE Robotics & Automation Magazine*, vol. 22, no. 4, pp. 110-124, Dec. 2015, doi: 10.1109/MRA.2015.2432331.***" [Online]***
*Elérhető (utoljára ellenőrizve: 2020.05.08)*:
*https://ieeexplore.ieee.org/document/7271006/*

[9] *„***How to use the ISS 3D keypoint detector**" *[Online]*
*Elérhető*: *http://www.pointclouds.org/blog/gsoc12/gballin/iss.php*
*[Hozzáférés dátuma: 2020.04.08]*

[10] *„***PCL Documentation**" *[Online]*
Elérhető:
*https://web.archive.org/web/20190908115616/http://docs.pointclouds.org/trunk/classpcl_1_1_s_i_f_t_keypoint.html#details [Hozzáférés dátuma: 2020.05.08]*

*[11] „**How 3D Features work in PCL**" [Online]*
*Elérhető:*
*https://web.archive.org/web/20191021105743/http://www.pointclouds.org/documentation/tutorials/how_features_work.php#how-3d-features-work*
*[Hozzáférés dátuma: 2020.05.08]*

[12] *„***Point Feature Histograms (PFH) descriptors**" *[Online]*
*Elérhető*:
*https://web.archive.org/web/20191005035458/http://pointclouds.org/documentation/tutorials/pfh_estimation.php#pfh-estimation [Hozzáférés dátuma: 2020.05.08]*

[13] *„***Fast Point Feature Histograms (FPFH) descriptors**" *[Online]*
*Elérhető:*

*https://web.archive.org/web/20191027172025/http://pointclouds.org/documentation/tutorials/fpfh_estimation.php#fpfh-estimation [Hozzáférés dátuma: 2020.05.08]*

[14] „**PCL/OpenNI tutorial 4: 3D object recognition (descriptors)**” *[Online]*
*Elérhető:*
*http://robotica.unileon.es/index.php/PCL/OpenNI_tutorial_4:_3D_object_recognition_(descriptors) [Hozzáférés dátuma: 2020.05.08]*

[15] „**PCL Documentation - How to use Random Sample Consensus model**” *[Online]*
*Elérhető*:
*https://web.archive.org/web/20191023224933/http://pointclouds.org/documentation/tutorials/random_sample_consensus.php#random-sample-consensus*
*[Hozzáférés dátuma: 2020.05.08]*

[16] „**Introduction to Mobile Robotics -Iterative Closest Point Algorithm**” *[Online]*
*Elérhetőség*:
*http://ais.informatik.uni-freiburg.de/teaching/ss12/robotics/slides/17-icp.pdf*
*[Hozzáférés dátuma: 2020.05.08]*

*[17] „**Iterative Closest Point – Wikipedia**” [Online]*
*Elérhetőség:*
*https://en.wikipedia.org/wiki/Iterative_closest_point [Hozzáférés dátuma: 2020.05.10]*

[18] „**PCL Documentation – Prerejective Ransom Sample Consensus**” *[Online]*
*Elérhetőség*:
*https://pointcloudlibrary.github.io/documentation/classpcl_1_1_sample_consensus_prerejective.html#details [Hozzáférés dátuma: 2020.05.08]*

*[19]* „**Robust pose estimation of rigid objects**” *[Online]*
*Elérhetőség:*
*https://pcl-tutorials.readthedocs.io/en/master/alignment_prerejective.html#alignment-prerejective [Hozzáférés dátuma: 2020.05.08]*

[20] „**Qt (softwarre – Wikipedia)**” *[Online]*
*Elérhetőség: https://en.wikipedia.org/wiki/Qt_(software)*
*[Hozzáférés dátuma: 2020.05.08]*

[21] „**Qt – Signals and Slots**” *[Online]*
*Elérhetőség*: *https://doc.qt.io/qt-5/signalsandslots.html*
*[Hozzáférés dátuma: 2020.05.08]*

[22] „**PCL – About**” *[Online]*
*Elérhetőség: https://pointcloudlibrary.github.io/about/*
*[Hozzáférés dátuma: 2020.05.08]*

*[23] „**PCL - Getting Started / Basic Structures**” [Online]*
*Elérhetőség***:**
*https://web.archive.org/web/20191029194912/http://www.pointclouds.org/documentation/tutorials/basic_structures.php#basic-structures*
*[Hozzáférés dátuma: 2020.05.08]*

[24] „**Adding your own custom PointTtype**” *[Online]*
*Elérhetőség*:
*https://web.archive.org/web/20191027172321/http://pointclouds.org/documentation/tutorials/adding_custom_ptype.php#adding-custom-ptype*
*[Hozzáférés dátuma: 2020.05.08]*

*[25] „**Module Keypoints – PCL**” [Online]*
*Elérhetőség:*
*https://pointcloudlibrary.github.io/documentation/group__keypoints.html*
*[Hozzáférés dátuma: 2020.05.08]*

*[26] „**Module Features – PCL**” [Online]*
*Elérhetőség: https://pointcloudlibrary.github.io/documentation/group__features.html*
*[Hozzáférés dátuma: 2020.05.08]*

*[27] „**Module Registration – PCL**” [Online]*

*Elérhetőség:*
*https://pointcloudlibrary.github.io/documentation/group__registration.html*
*[Hozzáférés dátuma: 2020.05.08]*

*[28] „**Module filters – PCL**” [Online]*
*Elérhetőség: https://pointcloudlibrary.github.io/documentation/group__filters.html*
*[Hozzáférés dátuma: 2020.05.08]*

*[29] „**Module surface – PCL**” [Online]*
*Elérhetőség: https://pointcloudlibrary.github.io/documentation/group__surface.html*
*[Hozzáférés dátuma: 2020.05.08]*

*[30] „**Module visualization – PCL**” [Online]*
*Elérhetőség:*
*https://pointcloudlibrary.github.io/documentation/group__visualization.html*
*[Hozzáférés dátuma: 2020.05.08]*