

# Uvod u PCL – filtriranje i vizualizacija oblaka točaka

*Tara Knežević, mag.ing.mech.*  
*Doc.dr.sc. Filip Šuligoj*

---



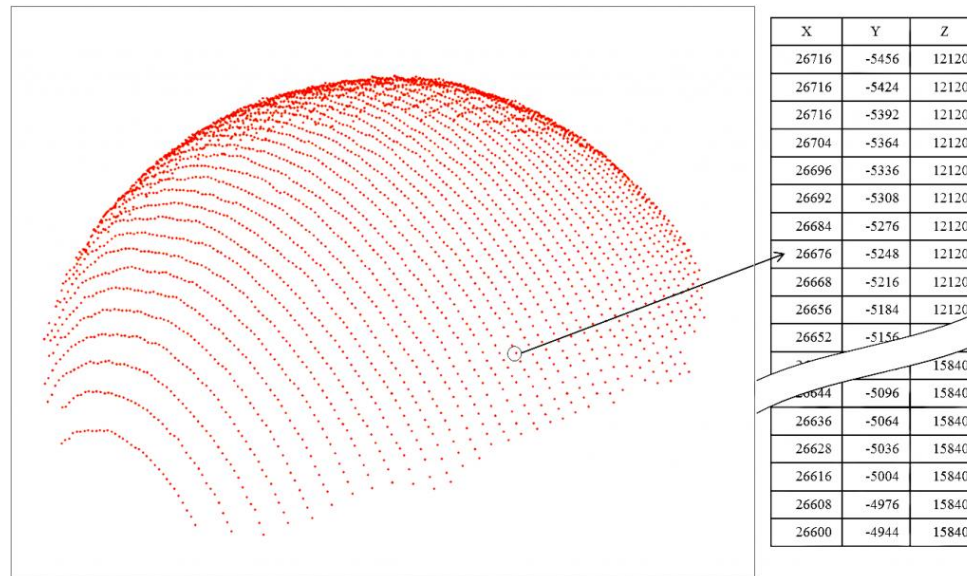
University of  
Zagreb



Faculty of mechanical  
engineering and naval  
architecture

# PC (Point Cloud) – oblak točaka

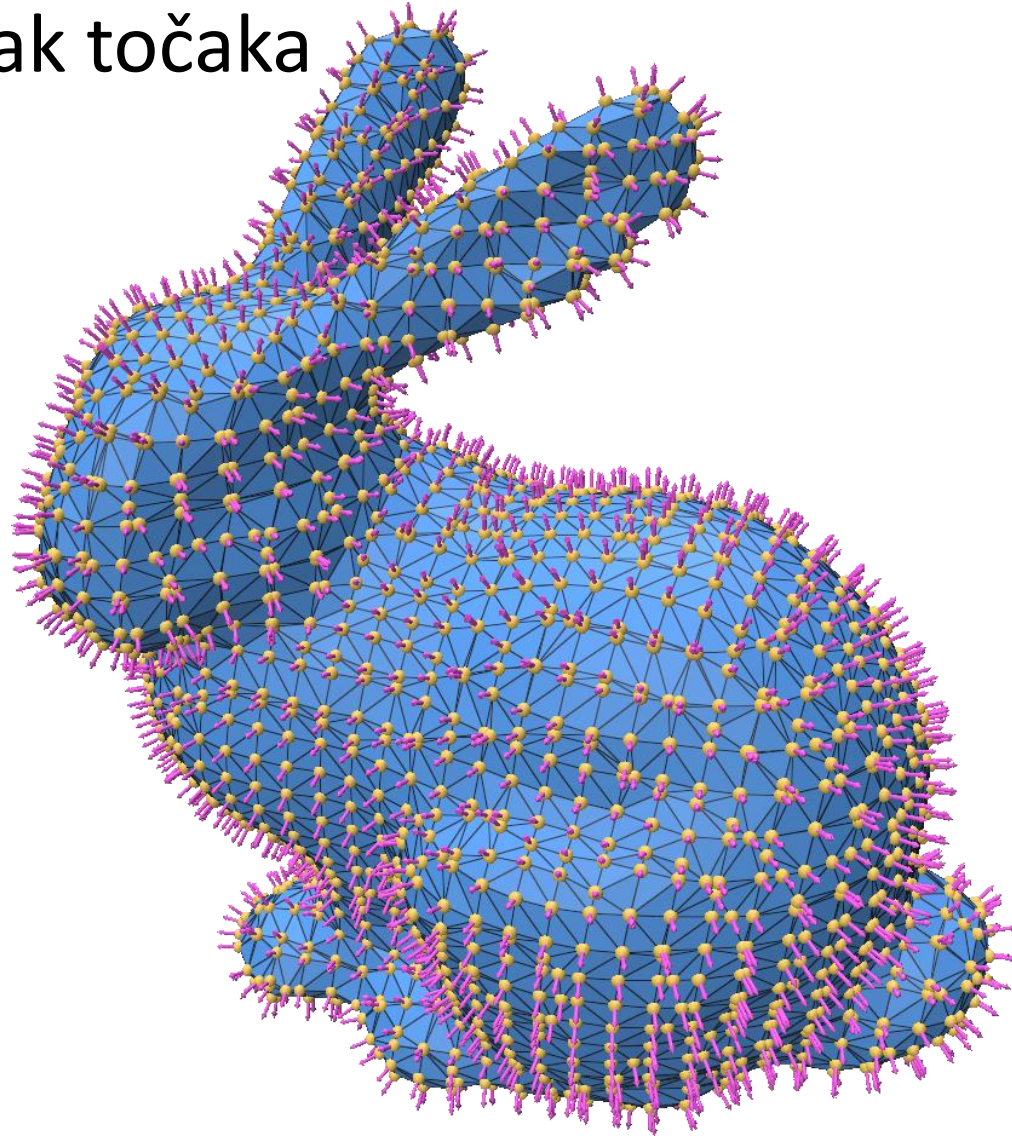
- Skup trodimenzionalnih točaka koje predstavljaju oblik ili površinu nekog objekta ili scene
- Svaka točka ima minimalno x, y i z koordinatu u prostoru



# PC (Point Cloud) – oblak točaka

Uz tri koordinate, oblak točaka može imati:

- Boju (RGB)
- Normale (pravci površine)
- Intenzitet (snaga refleksije signala)





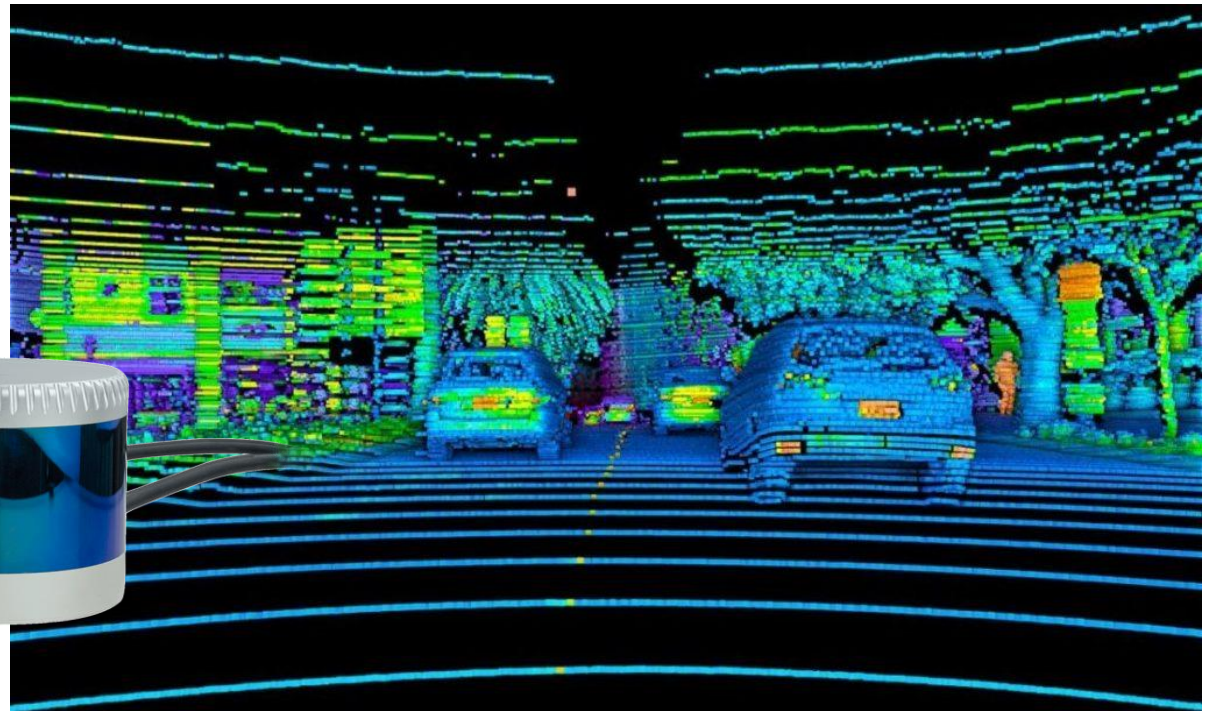
# PC (Point Cloud) – oblak točaka

Oblak točaka nastaje korištenjem 3D senzora

- 3D LIDAR



Leishen CX16



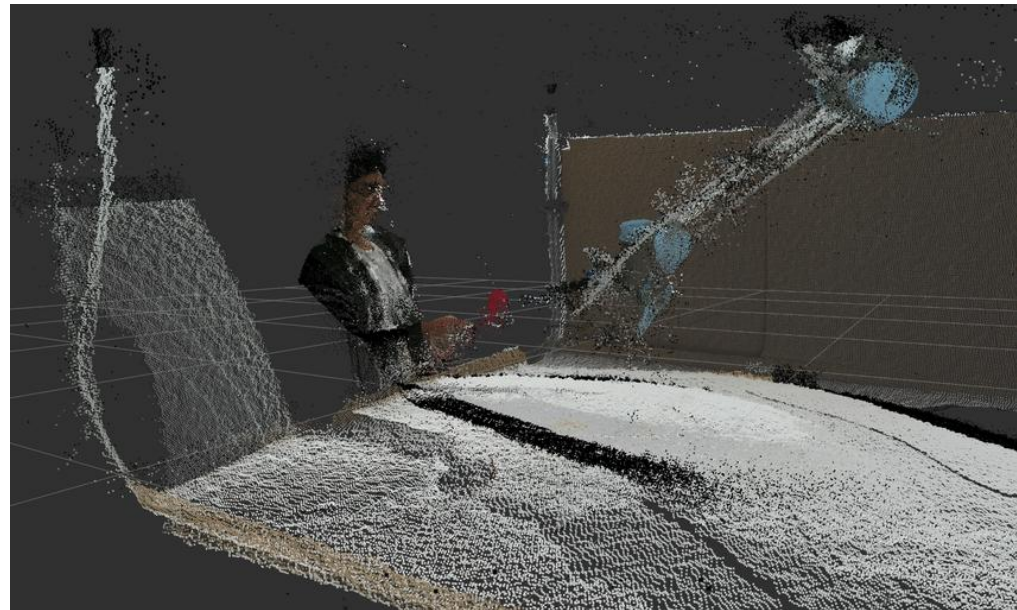
# PC (Point Cloud) – oblak točaka

Oblak točaka nastaje korištenjem 3D senzora

- 3D LIDAR
- RGB-D kamere



Microsoft Kinect



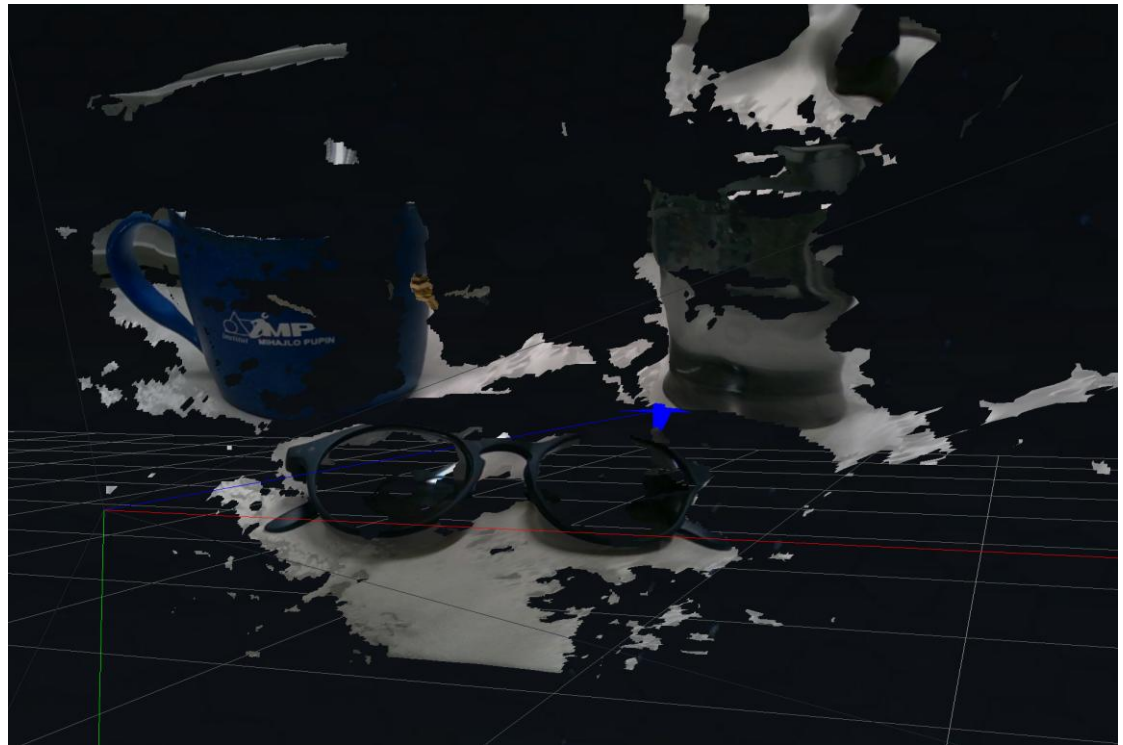
# PC (Point Cloud) – oblak točaka

Oblak točaka nastaje korištenjem 3D senzora

- 3D LIDAR
- RGB-D kamere
- Stereo kamere



Intel RealSense D405

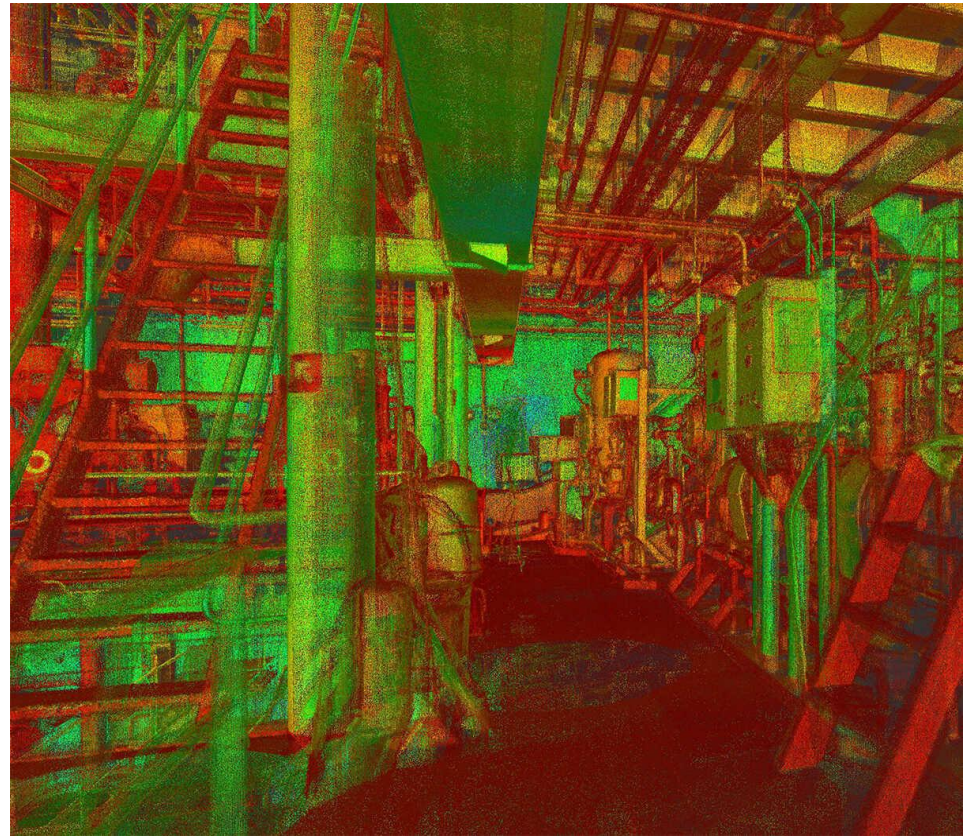




# PC (Point Cloud) – oblak točaka

Oblak točaka nastaje korištenjem 3D senzora

- 3D LIDAR
- RGB-D kamere
- Stereo kamere
- 3D skeneri



# PCL (Point Cloud Library)



- Open source biblioteka za obradu oblaka točaka
- Sadrži module za
  - Filtriranje
  - Segmentaciju
  - Grupiranje
  - Registraciju
  - Vizualizaciju
- Razvijen je kako bi standardizirao i olakšao rad s kompleksnim 3D podacima



# Point cloud u PCL-u

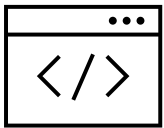
PCL nudi različite tipove točaka (point types) za različite aplikacije:

Tip	Opis
<code>pcl::PointXYZ</code>	Samo prostorne koordinate (x, y, z).
<code>pcl::PointXYZRGB</code>	Prostorne koordinate + boja (r, g, b).
<code>pcl::PointNormal</code>	Koordinate + normale (smjer površine).
<code>pcl::PointXYZRGBNormal</code>	Koordinate + boja + normale.
<code>pcl::PointXYZI</code>	Koordinate + intenzitet.

# Primjer 01 – generiranje PC

- Za početak rada potrebno je **generirati ili učitati** point cloud
- U ovom primjeru generiramo nasumičnih **500 točaka** u 3D prostoru
- Svaka točka ima (x, y, z) koordinate u **opsegu od 0 do 1024**
- Točke su zatim prikazane pomoću **PCLVisualizer** alata





# Terminator

Preporuka koristiti ga:

- Terminator nudi puno više opcija za uređivanje izgleda, tipki prečaca i ponašanja nego obični Terminal
- Terminator omogućuje **horizontalno i vertikalno dijeljenje prozora** na više terminala unutar jednog

## Splitting

Ctrl+Shift+O	Split terminals Horizontally.
Ctrl+Shift+E	Split terminals Vertically.
Ctrl+Shift+W	Close the current terminal.

## Moving

Alt+Right	Move to the right terminal.
Alt+Left	Move to the left terminal.
Alt+Up	Move to the upper terminal.
Alt+Down	Move to the bottom terminal.

## Tabs

Ctrl+Shift+T	Open new tab
Ctrl+PageDown	Move to next Tab
Ctrl+PageUp	Move to previous Tab

## Font Size

Ctrl+Plus (+)	Increase font size.
Ctrl+Minus (-)	Decrease font size.
Ctrl+Zero (0)	Restore font size to original setting.



# Primjer 01 – generiranje PC

```
#include <iostream>
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <pcl/io/pcd_io.h>

int main() {
    // 1. Kreiranje praznog point clouda
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ>);

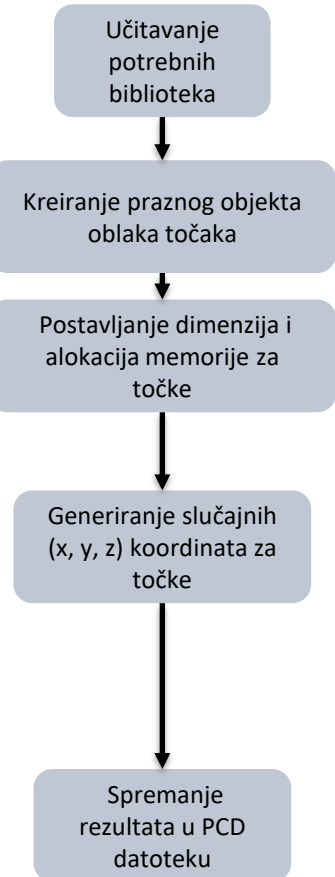
    // 2. Definiranje veličine
    cloud->width = 500;
    cloud->height = 5;
    cloud->is_dense = false;
    cloud->points.resize(cloud->width * cloud->height);

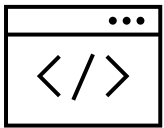
    // 3. Generiranje nasumičnih točaka
    for (auto& point : cloud->points) {
        point.x = 1024 * rand() / (RAND_MAX + 1.0f);
        point.y = 1024 * rand() / (RAND_MAX + 1.0f);
        point.z = 1024 * rand() / (RAND_MAX + 1.0f);
    }

    std::cout << "Generirano " << cloud->points.size() << " točaka." << std::endl;

    // 4. Spremanje u data folder
    std::string output_path = "/home/asvtara/pcl_ws/data/generated_cloud.pcd";
    pcl::io::savePCDFileASCII(output_path, *cloud);
    std::cout << "Point cloud spremljen u '" << output_path << "'." << std::endl;

    return 0;
}
```





# Predložak (Template)

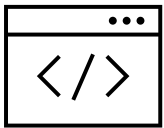
- Template se najčešće definira za funkcije i klase
- **Template klase** je generička definicija klase koja omogućava da se ista struktura koristi za različite tipove podataka, bez dupliciranja koda
- Kod je **kraći, čitljiviji, i lakši za održavanje**
- Tip podatka koji će se koristiti u klasi definira se **prilikom instanciranja klase**, a označava se **simbolički (npr. T) unutar deklaracije template** <typename T>

```
template <typename T>
class MyTemplateClass {
public:
    T value;

    MyTemplateClass(T v) : value(v) {}

    void print() {
        std::cout << value << std::endl;
    }
};
```

Generička definicija  
template klase



# Predložak (Template)

```
template <typename T>
class MyTemplateClass {
public:
    T value;

    MyTemplateClass(T v) : value(v) {}

    void print() {
        std::cout << value << std::endl;
    }
};
```

Generička definicija  
template klase

*Definira da je ovo **template klasa**  
koja radi s bilo kojim tipom *T**



Template je prepoznatljiv po tome što se tip podatka nalazi između znakova <>  
Puni naziv klase stvorene na osnovu template-a uključuje argumente koji se navode unutar znakova <>. Za stvaranje objekata klase potrebno je koristiti njen pun naziv



- 'good to know', savjet, informacija

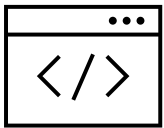


University of  
Zagreb



Faculty of mechanical  
engineering and naval  
architecture





# Predložak (Template)

```
template <typename T>
class MyTemplateClass {
public:
    T value;

    MyTemplateClass(T v) : value(v) {}

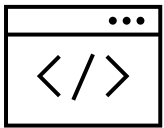
    void print() {
        std::cout << value << std::endl;
    }
};
```

Generička definicija  
template klase

*Definira klasu s imenom  
MyTemplateClass*



Klase i funkcije definiraju se unutar zagrada {}. Kod funkcija se pišu čiste zagrade, a kod klasa se na kraju definicije obavezno dodaje ;.



# Predložak (Template)

```
template <typename T>
class MyTemplateClass {
public:
    T value;

    MyTemplateClass(T v) : value(v) {}

    void print() {
        std::cout << value << std::endl;
    }
};
```

Generička definicija  
template klase

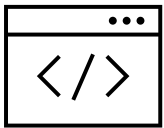
*Omogućuje da članovi klase (varijable, funkcije) budu dostupni izvana, tj. drugim funkcijama, objektima i klasama.*



U C++-u postoje tri modifikatora pristupa:

1. *public* - članovi klase su dostupni izvana – drugim funkcijama, klasama i objektima
2. *private* - članovi su dostupni samo unutar same klase, tj. skriveni su od ostatka programa
3. *protected* - omogućava pristup članovima unutar klase i u svim klasama koje nasljeđuju tu klasu

Postoji puno razloga zašto je dobro objekte dijeliti u te tri razine pristupa, a glavni je da se štite unutarnja stanja objekta od nepoželjnih promjena izvana.



# Predložak (Template)

```
template <typename T>
class MyTemplateClass {
public:
    T value;

    MyTemplateClass(T v) : value(v) {}

    void print() {
        std::cout << value << std::endl;
    }
};
```

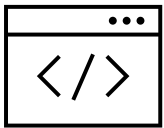
Generička definicija  
template klase

*Varijabla klase MyTemplateClass, tipa T*



Što će T biti, ovisi o tome kako će se klasa instancirati.  
Može biti bilo koji tip podatka (int, string, float...)





# Predložak (Template)

```
template <typename T>
class MyTemplateClass {
public:
    T value;

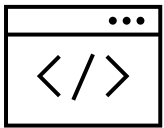
    MyTemplateClass(T v) : value(v) {}

    void print() {
        std::cout << value << std::endl;
    }
};
```

Generička definicija  
template klase

*Konstruktor klase koji prima argument v,  
tipa T koji inicijalizira varijablu value, s v*

- 💡 Konstruktor je **posebna metoda klase** koja se automatski poziva **prilikom stvaranja objekta**  
Služi za **inicijalizaciju podataka** unutar objekta i uvijek ima **isto ime kao i klasa**, ali **nema tip povratne vrijednosti** (čak ni void)  
Kad stvorimo objekt klase MyTemplateClass, konstruktor će se pozvati i inicijalizirati varijablu value na vrijednost v, koju smo unijeli pri stvaranju objekta



# Predložak (Template)

```
template <typename T>
class MyTemplateClass {
public:
    T value;

    MyTemplateClass(T v) : value(v) {}

    void print() {
        std::cout << value << std::endl;
    }
};
```

Generička definicija  
template klase

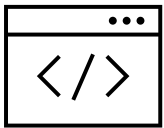
*Metoda klase koja ispisuje vrijednost  
člana value*



U srži, **funkcija i metoda su ista stvar** - obje su blokovi koda koji mogu primiti argumente i vraćati vrijednosti

**Funkcija** je **samostalan blok koda** koji može postojati izvan klase i poziva se neovisno o objektima

**Metoda** je **funkcija koja je definirana unutar klase i poziva se nad objektom** (ili klasom, ako je statična)

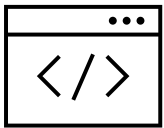


# Predložak PointCloud klase

- PCL koristi template kako bi mogli definirati kakav PC tip podatka želimo koristiti, bez da postoji više različitih klasa
- Dakle, s istom klasom PointCloud, može se raditi s točkama koje imaju samo koordinate, boju, normale, ili sve zajedno

PointCloud  
template klasa

```
template <typename PointT>
class pcl::PointCloud {
public:
    std::vector<PointT> points;
    uint32_t width;
    uint32_t height;
    bool is_dense;
    // Ostali članovi i metode...
};
```



# Predložak PointCloud klase

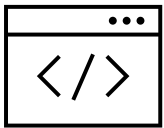
**Zadatak:** otvoriti datoteku '*point\_cloud.h*' te pronaći:

- Liniju u kojoj se deklarira klasa PointCloud
- Dio koda gdje je objašnjeno kako se definiraju širina i visina point clouda, kao i ukupan broj točaka

```
// Forward declarations
template <typename PointT> class PointCloud;
```

```
* The PointCloud class contains the following elements:
*   - \b width - specifies the width of the point cloud dataset in the number of points. WIDTH has two meanings:
*   - it can specify the total number of points in the cloud (equal with POINTS see below) for unorganized datasets;
*   - it can specify the width (total number of points in a row) of an organized point cloud dataset.
*   \a Mandatory.
*   - \b height - specifies the height of the point cloud dataset in the number of points. HEIGHT has two meanings:
*   - it can specify the height (total number of rows) of an organized point cloud dataset;
*   - it is set to 1 for unorganized datasets (thus used to check whether a dataset is organized or not).
*   \a Mandatory.
*   - \b points - the data array where all points of type <b>PointT</b> are stored. \a Mandatory.
```

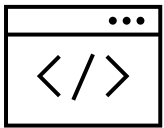




# Koje sve tipove podataka možemo koristiti s PointCloud klasom? Gdje su definirani?

- Definirani su kao **strukture** unutar datoteke *'point\_types.h'*
- Najčešće korišteni tipovi podataka:

Tip	Opis
pcl::PointXYZ	Samo prostorne koordinate (x, y, z).
pcl::PointXYZRGB	Prostorne koordinate + boja (r, g, b).
pcl::PointNormal	Koordinate + normale (smjer površine).
pcl::PointXYZRGBNormal	Koordinate + boja + normale.
pcl::PointXYZI	Koordinate + intenzitet.



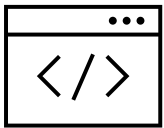
# Instanciranje template klase

- **Instanciranje** = postupak dodjeljivanja konkretnog tipa podataka umjesto generičkog (PointT) u trenutku korištenja (npr. deklaracije objekta te klase)

```
// 1. Kreiranje praznog point clouda  
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ>);
```



Što ovaj dio označava?



# Pointer

- **Pointer** je varijabla koja sadrži **memorijsku adresu nekog drugog objekta** (varijable, funkcije, objekta klase)
- **Primjer 02 – pointer**

```
#include <iostream>

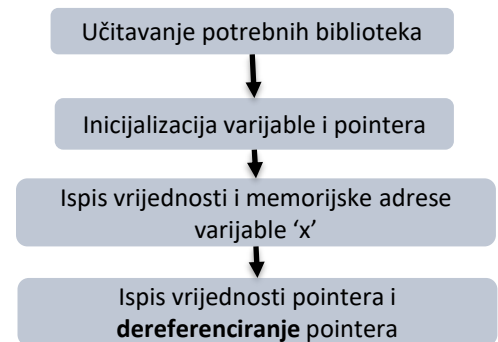
int main() {
    int x = 37;           // obična varijabla 'x'
    int* p = &x;          // pointer 'p' koji pokazuje na x

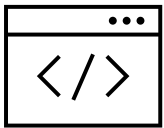
    std::cout << "Vrijednost x: " << x << std::endl;
    std::cout << "Adresa x (&x): " << &x << std::endl;

    std::cout << "\nVrijednost pointera p (adresa na koju pokazuje): " << p << std::endl;
    std::cout << "Vrijednost na koju pokazuje p (*p): " << *p << std::endl;

    std::cout << "\nPromjena x preko pointera (*p = 100):" << std::endl;
    *p = 100;
    std::cout << "Nova vrijednost x: " << x << std::endl;
    std::cout << "Nova vrijednost *p: " << *p << std::endl;

    return 0;
}
```





# Pointer

- Primjer 02 – pointer

```
#include <iostream>

int main() {
    int x = 37;           // obična varijabla 'x'
    int* p = &x;          // pointer 'p' koji pokazuje na x

    std::cout << "Vrijednost x: " << x << std::endl;
    std::cout << "Adresa x (&x): " << &x << std::endl;

    std::cout << "\nVrijednost pointera p (adresa na koju pokazuje): " << p << std::endl;
    std::cout << "Vrijednost na koju pokazuje p (*p): " << *p << std::endl;

    std::cout << "\nPromjena x preko pointera (*p = 100):" << std::endl;
    *p = 100;
    std::cout << "Nova vrijednost x: " << x << std::endl;
    std::cout << "Nova vrijednost *p: " << *p << std::endl;

    return 0;
}
```



Učitavanje potrebnih biblioteka



Inicijalizacija varijable i pointera



Ispis vrijednosti i memorijske adrese  
varijable 'x'

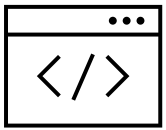


Ispis vrijednosti pointera i  
**dereferenciranje** pointera



Promjena vrijednosti varijable  
preko pointera i provjera nove  
vrijednosti

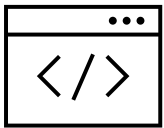
- Dereferenciranje pointera** = pristup stvarnoj vrijednosti na adresi memorije na koju pointer pokazuje, koristeći operator '\*'



# Zašto koristimo pointere?

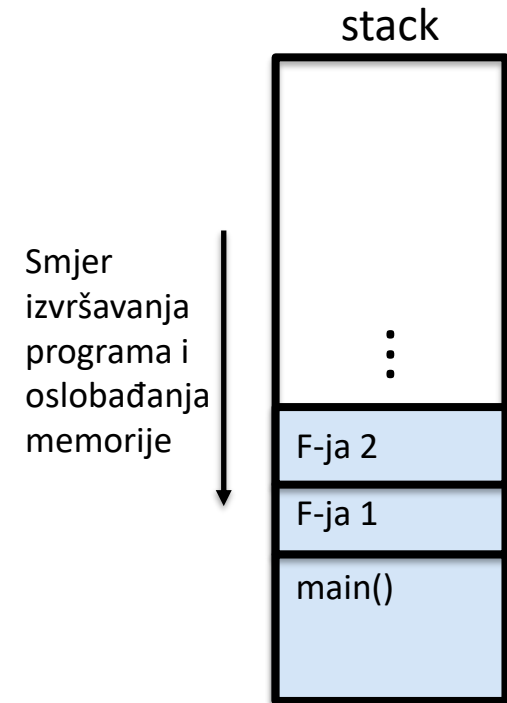
- Dijeljenje podataka između funkcija bez kopiranja
- Kada se funkciji pošalje neka vrijednost, ona dobije kopiju te varijable. Ako se pošalje pointer, funkcija pristupa originalnom podatku
- Efikasniji rad s velikim objektima (npr. pointcloud)
  - Kod velikih objekata nije praktično kopirati ih svaki put, a pointer omogućuje da radimo s originalnim objektom i time štedimo memoriju i vrijeme
- Potrebni za rad s nizovima, strukturama i klasama
  - Pogotovo kod PCL-a
- Pristup podacima u heap memoriji
  - Ova memorija traje duže od scope-a, ostaje živa dok ju ne oslobodiš

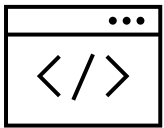




# Stack (stog) memorija

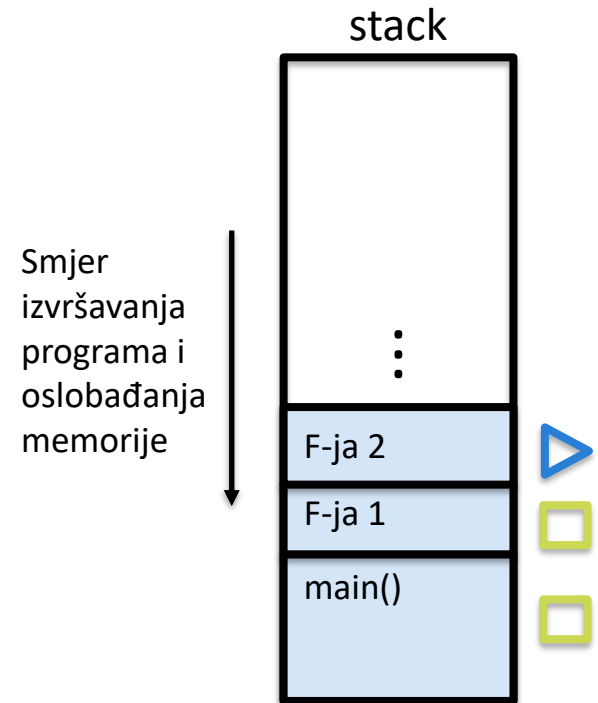
- **Stack** = dio memorije koji automatski upravlja lokalnim varijablama i parametrima funkcija
- Memorija se slijedno zauzima
- Svaka funkcija koja se pozove dobiva svoj "stack frame", a kada se funkcija završi, taj frame se automatski uklanja
- Brz je i jednostavan, ali ima ograničen kapacitet i varijable nestaju čim funkcija završi

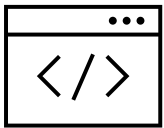




# Stack (stog) memorija

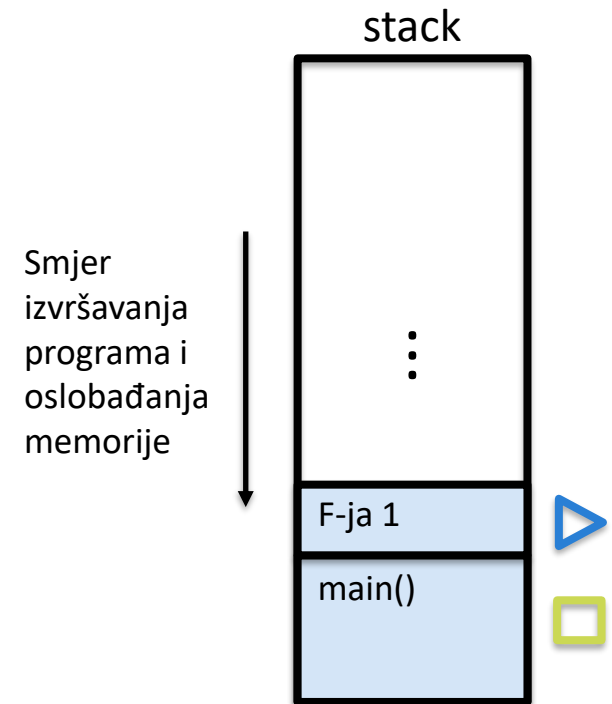
- **Stack** = dio memorije koji automatski upravlja lokalnim varijablama i parametrima funkcija
- Memorija se slijedno zauzima
- Svaka funkcija koja se pozove dobiva svoj "stack frame", a kada se funkcija završi, taj frame se automatski uklanja
- Brz je i jednostavan, ali ima ograničen kapacitet i varijable nestaju čim funkcija završi

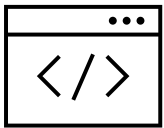




# Stack (stog) memorija

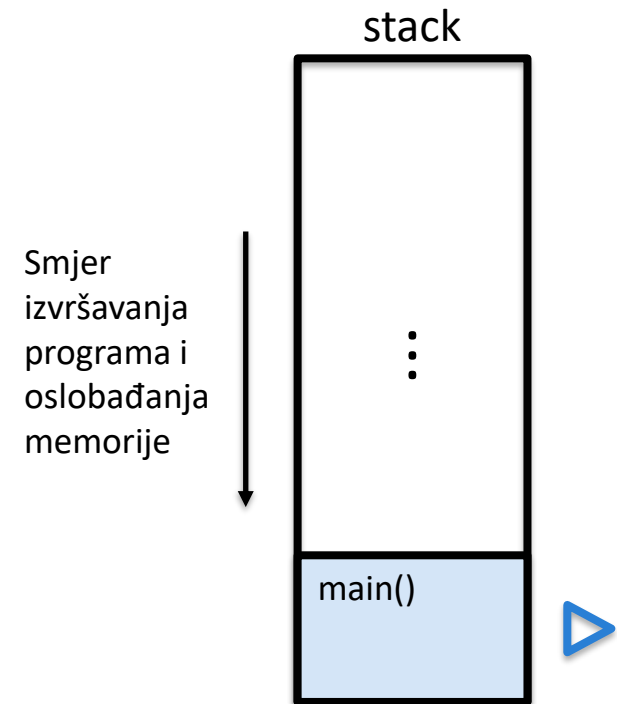
- **Stack** = dio memorije koji automatski upravlja lokalnim varijablama i parametrima funkcija
- Memorija se slijedno zauzima
- Svaka funkcija koja se pozove dobiva svoj "stack frame", a kada se funkcija završi, taj frame se automatski uklanja
- Brz je i jednostavan, ali ima ograničen kapacitet i varijable nestaju čim funkcija završi

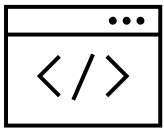




# Stack (stog) memorija

- **Stack** = dio memorije koji automatski upravlja lokalnim varijablama i parametrima funkcija
- Memorija se slijedno zauzima
- Svaka funkcija koja se pozove dobiva svoj "stack frame", a kada se funkcija završi, taj frame se automatski uklanja
- Brz je i jednostavan, ali ima ograničen kapacitet i varijable nestaju čim funkcija završi



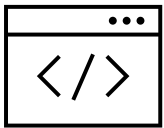


# Stack (stog) memorija

- Na kraju programa, sva zauzeta memorija se oslobađa i može se koristiti u drugom programu

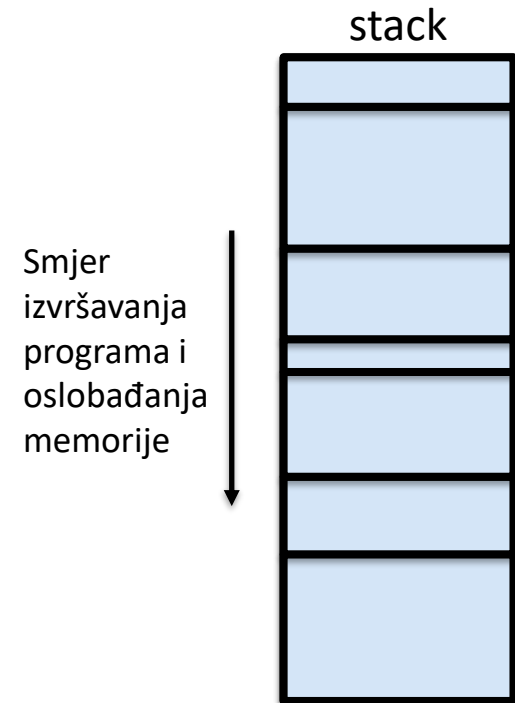


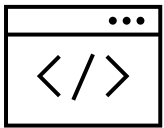




# Stack (stog) memorija

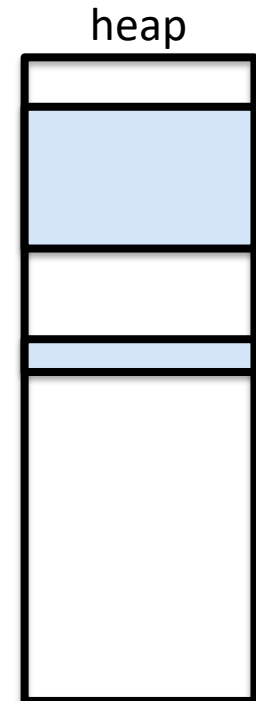
- Na kraju programa, sva alocirana memorija se oslobađa i može se koristiti u drugom programu
- Može li se stack memorija zapuniti?
- DA! – to se zove **stack overflow** i izaziva rušenje programa tijekom izvedbe
- Najčešći razlog – krivo programiranje (npr. rekurzija koja je ušla u beskonačni loop) ili preveliki objekt na stacku (npr. **pointcloud** s velikim brojem točaka)

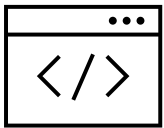




# Heap (gomila) memorija

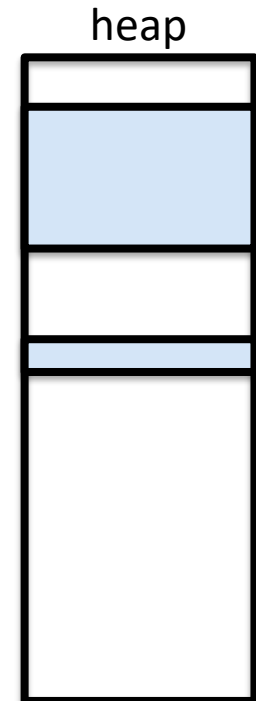
- **Heap** = dio memorije koji omogućuje dinamičku alokaciju memorije tijekom izvođenja programa – memoriju tražiš tijekom izvođenja programa pomoću *'new'*
- Drugi naziv – dinamička memorija
- Korištenje memorije dinamički naziva se **alokacijom memorije**
- Programer je odgovoran za **alokaciju** (*new*) i **dealokaciju** (*delete*) memorije

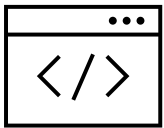




# Heap (gomila) memorija

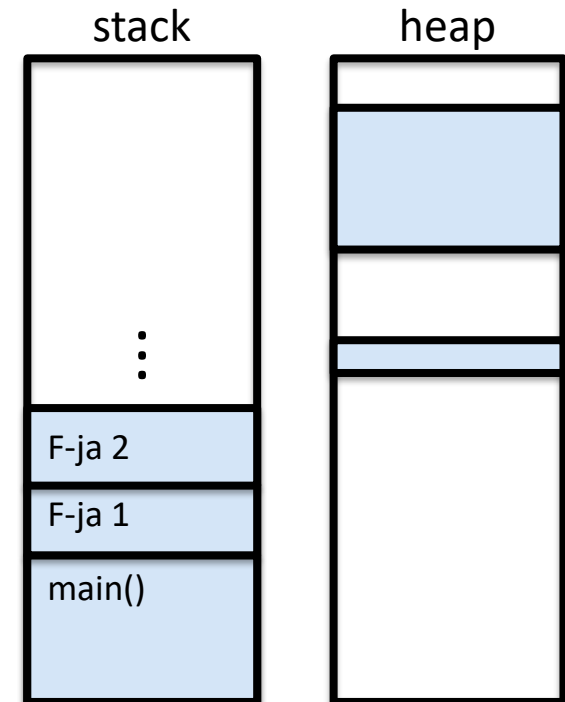
- Memorija se **ne popunjava slijedno**, nego se traži mjesto u memoriji gdje ima slobodnog mjesta
- Za razliku od stacka, **ti upravljaš trajanjem memorije** – ostaje zauzeta dok ju ručno ne oslobodiš
- Heap ima veći kapacitet, ali je pristup sporiji i zahtijeva pažljivo upravljanje da ne bi došlo do curenja memorije (**memory leak**)

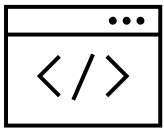




# Heap (gomila) memorija

- Popunjava se **uz stack memoriju**, sve što nije dinamički alocirano spremljeno je na stack-u



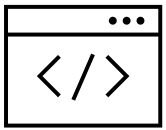


# Stack vs. heap pointer

- razlikuju se po tome **gdje** u memoriji **pokazuju** i kako se tim memorijama upravlja

	Stack pointer	Heap pointer
<b>Mjesto alokacije memorije</b>	Stack	Heap
<b>Memorijom upravlja</b>	Kompajler (automatski)	Programer (opasnost memory leak-a)
<b>Brzina pristupa</b>	Vrlo brz	Sporiji
<b>Kapacitet</b>	Ograničen	Velik
<b>Životni vijek objekata</b>	Dok traje funkcija	Dok ga ne izbrišeš
<b>Pokazuje na</b>	Lokalnu varijablu	Dinamički alociran objekt

- **Primjer 02 – pointer** – u ovom primjeru definiran je stack pointer

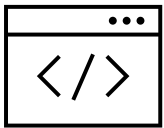


# Heap pointer

Postoje dva načina definicije ovih pointera

1. **Raw pointeri** (*'new' + 'delete'*) – ako zaboraviš pozvati *'delete'*, memorija ostaje zauzeta sve dok program traje, što uzrokuje memory leak
  2. **Pametni pointeri** - ne moraš pozivati *'delete'*, pametni pokazivač automatski briše objekt čime čini kod sigurnijim
- Preporuka je raditi s **pametnim pointerima**

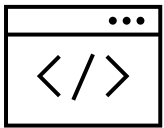




# Pametni pointeri

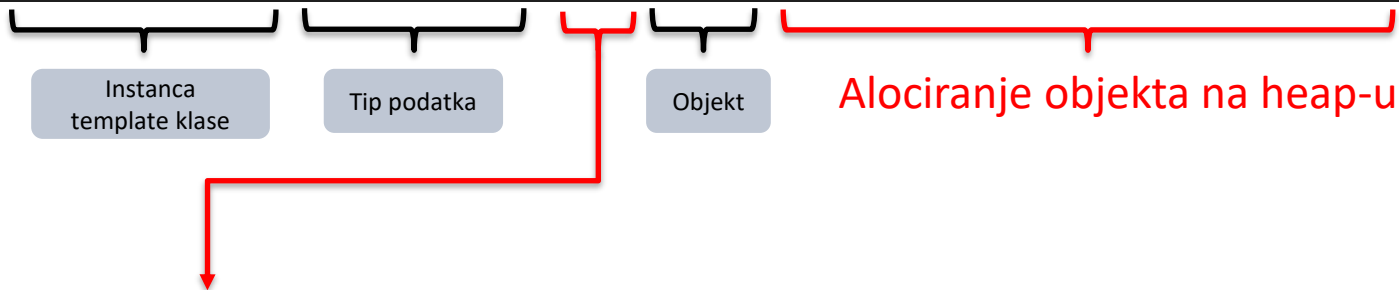
1. **`std::unique_ptr<T>`** - ima jednog vlasnika objekta, ne može se kopirati
2. **`std::shared_ptr<T>`** - više pointera može pokazivati na isti objekt, a interno se broji koliko ih je, objekt se briše kada broj referenci dođe na ‘
3. **`std::weak_ptr<T>`** - nema vlasništvo, ne produljuje životni vijek objekta

**Vlasnik** je pametni pokazivač koji odlučuje kad će se objekt automatski obrisati (ako nitko više nije vlasnik – objekt se briše iz memorije)



# Heap pointer – primjer 01 – generiranje PC

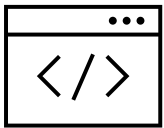
```
// 1. Kreiranje praznog point clouda  
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ>);
```



```
using Ptr = shared_ptr<PointCloud<PointT> >;
```

*Kod kuće pronaći ovu definiciju*

- Pointer je definiran kao `shared_pointer`, jer svaki pointcloud prođe više stadija manipulacije (filtriranje, grupiranje, stitching, registracija, segmentacija...)



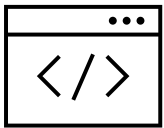
# Dereferenciranje pointera – primjer 01 – generiranje PC

- Pri uređivanju postavki pointclouda (pointer), potrebno je prvo **dereferencirati** pointer, kako bi se moglo pristupiti varijabli i vrijednostima iste

```
// 2. Definiranje veličine
cloud->width = 500;
(*cloud).height = 5;
cloud->points.resize(cloud->width * cloud->height);
```

Prikazana su dva načina dereferenciranja, ekvivalentna su!

- Prvi način: '*pointer->*'
- Drugi način: '*(\*pointer).*'



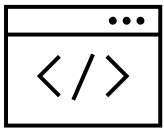
# 'Auto' ispred varijable – primjer 01 – generiranje PC

- Prepoznavanje tipa varijable s kojim se radi

```
// 3. Generiranje nasumičnih točaka
for (auto& point : cloud->points) {
    point.x = 1024 * rand() / (RAND_MAX + 1.0f);
    point.y = 1024 * rand() / (RAND_MAX + 1.0f);
    point.z = 1024 * rand() / (RAND_MAX + 1.0f);
}
```

Objekt 'point' automatski poprima tip podatka koji ima i 'cloud->points' (riječ je o `std::vector<pcl::PointXYZ>`)

- Jasno je da je ovo vrlo korisno koristiti jer štedi pisanje, ali i zato što ne moramo mijenjati kod ako promijenimo tip podataka s kojim radimo



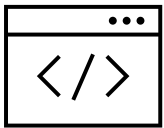
# Što znači ‘&’ u ‘auto& point’

- & označava referencu, tj. da se izravno pristupa originalnom point cloudu i da se utječe na pravi pointcloud

```
// 3. Generiranje nasumičnih točaka
for (auto& point : cloud->points) {
    point.x = 1024 * rand() / (RAND_MAX + 1.0f);
    point.y = 1024 * rand() / (RAND_MAX + 1.0f);
    point.z = 1024 * rand() / (RAND_MAX + 1.0f);
}
```

**Referenca** je alias, tj. drugo ime za postojeći objekt. Ne stvara novu kopiju, nego omogućuje **direktan pristup originalu**.

- Jasno je da je ovo vrlo korisno koristiti jer štedi memoriju i omogućuje promjene izravno u originalnom objektu



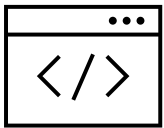
# Struktura c++ projekta

- Prikazana je preporučena hijerarhija projekta te glavni dijelovi istog (i njihova objašnjenja)

```
projekt_ime/
├─ CMakeLists.txt      ← Glavna CMake konfiguracija
├─ src/                ← Izvorni kod (.cpp datoteke)
│   └─ main.cpp        ← Ulazna točka (ili više .cpp modula)
├─ include/            ← Zaglavlja (.h/.hpp)
│   └─ my_header.hpp
├─ build/              ← Direktorij za izgradnju (generira se)
└─ data/               ← Pomoćni podaci (slike, PCD datoteke itd.)
```

- **Napomena:** nećemo raditi s *'h/.hpp'* datotekama u sklopu ovog kolegija, ali inače su ona bitan dio programiranja u C++ jeziku





# CMakeLists.txt

- Cmake je alat za automatiziranu izgradnju C++ projekata
- Datoteka 'CMakeLists.txt' opisuje kako da se projekt kompilira – definira izvore, zavisnosti i pravila buildanja

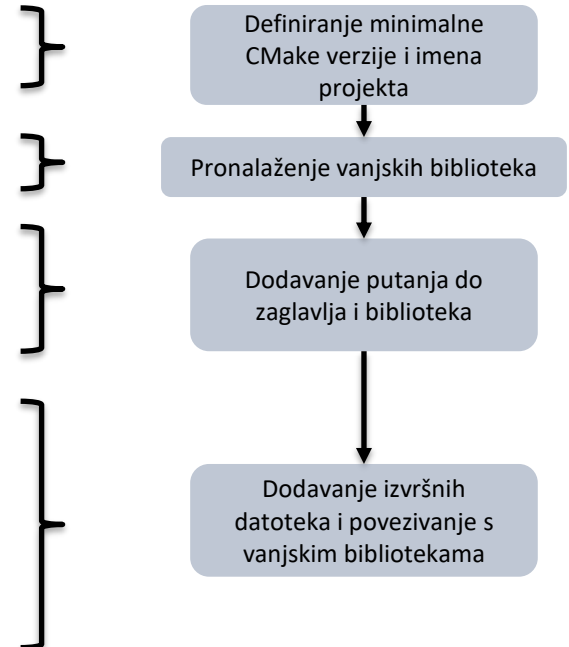
```
cmake_minimum_required(VERSION 3.5)
project(GeneratePointCloud)

# Nađi PCL biblioteku
find_package(PCL 1.10 REQUIRED)

# Uključi zaglavlja i biblioteke
include_directories(${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})

# Dodaj izvršnu datoteku i poveži s PCL-om
add_executable(generate_point_cloud src/01_generiranje_PC.cpp)
target_link_libraries(generate_point_cloud ${PCL_LIBRARIES})

add_executable(pointer src/02_pointer.cpp)
target_link_libraries(pointer ${PCL_LIBRARIES})
```

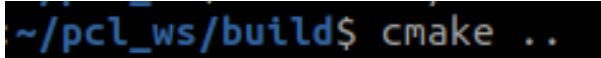
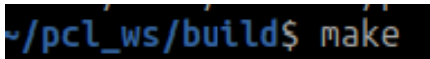


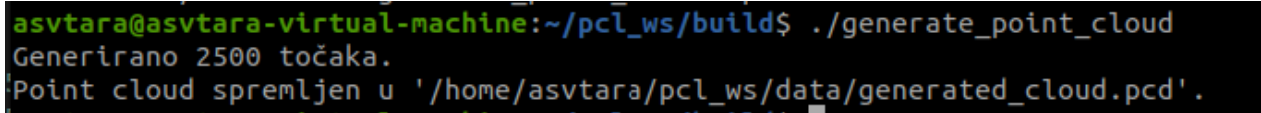
# Primjer 01 – generiranje PC - build

- Da bismo pokrenuli C++ program koji koristi vanjske biblioteke (npr. PCL), koristimo CMake – alat koji generira sve potrebne build datoteke
- Postupak:
  1. **Struktura projekta** mora biti organizirana (src/, build/, CMakeLists.txt, data/...)
  2. Uđemo u build/ mapu s naredbom '**cd build**'
  3. Pokrenemo '**cmake ..**' – stvara potrebne datoteke
  4. Pokrenemo '**make**' – kompajlira kod
- Nakon uspješnog builda, u build/ direktoriju pojavi se **izvršna datoteka** koju zatim **pokrećemo u terminalu**

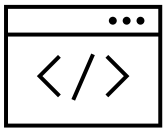
# Primjer 01 – generiranje PC

**Zadatak:** buildati i pokrenuti datoteku primjer 01

- ***cd build***
- ***cmake ..*** 
- ***make*** 
- Pokretanje datoteke: ***./executable\_datoteka***



```
asvtara@asvtara-virtual-machine:~/pcl_ws/build$ ./generate_point_cloud
Generirano 2500 točaka.
Point cloud spremljen u '/home/asvtara/pcl_ws/data/generated_cloud.pcd'.
```



# pcl\_viewer

- Alat za brzu vizualizaciju '*pcd*' datoteka
- Dolazi s PCL bibliotekom i podržava prikaz točaka, RGB boju i normale
- Korištenje: pokreće se iz terminala, unutar direktorija u kojem se nalazi datoteka koja se vizualizira
- '+/-' = zoom in/zoom out
- 'shift i +/ shift i -' = povećavanje/smanjivanje veličina točaka

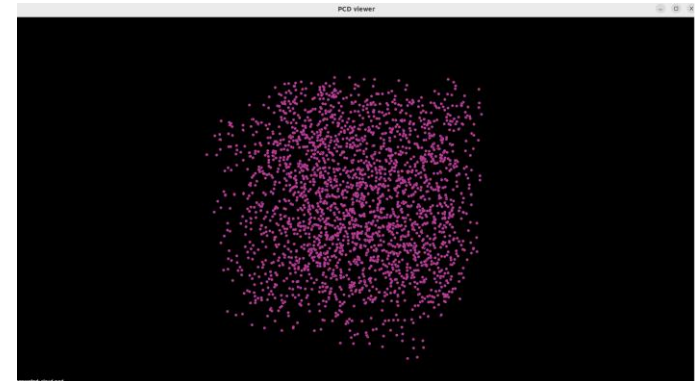
# Primjer 01 – generiranje PC

**Zadatak:** vizualizirati generirani PC koristeći pcl\_viewer

- ući u direktorij u kojem se nalazi pointcloud
- pokrenuti pcl\_viewer s imenom datoteke

```
asvtara@asvtara-virtual-machine:~/pcl_ws/data$ pcl_viewer generated_cloud.pcd
2025-05-05 22:00:23.918 ( 0.002s) [ 5EEFC880] vtkContextDevice2D.cxx:32   WARN| Error: no override found for 'vtkContextDevice2D'.
The viewer window provides interactive commands; for help, press 'h' or 'H' from within the window.
> Loading generated_cloud.pcd [PCLVisualizer::setUseVbos] Has no effect when OpenGL version is ≥ 2
[done, 180.936 ms : 2500 points]
Available dimensions: x y z
```

- rotirati, zoomirati pointcloud
- povećati veličinu točaka pointclouda



# Zadatak 01

**Zadatak:** iz koda saznati informaciju o veličini PC-a te ga vizualizirati koristeći `pcl_viewer`

- PC se zove '**cloud\_1.pcd**' i nalazi se u '`data`' folderu
- Kod bazirati na primjeru 01, kostur datoteke se zove '`03_citanje_pc.cpp`'
- Unutar koda potrebno je:
  1. Stvoriti PC objekt
  2. Učitati `.pcd` datoteku
  3. Ispisati ime PC-a koji je učitano
  4. Ispisati broj točaka PC-a
- Executable dodati u `CMakeLists.txt` te ga nazvati '`citanje_pc`'
- Buildati projekt (`make`)
- Pokrenuti kod i provjeriti rezultate, a nakon toga vizualizirati PC

---

\*\*\* u kodu označava mjesto gdje morate nešto dodati

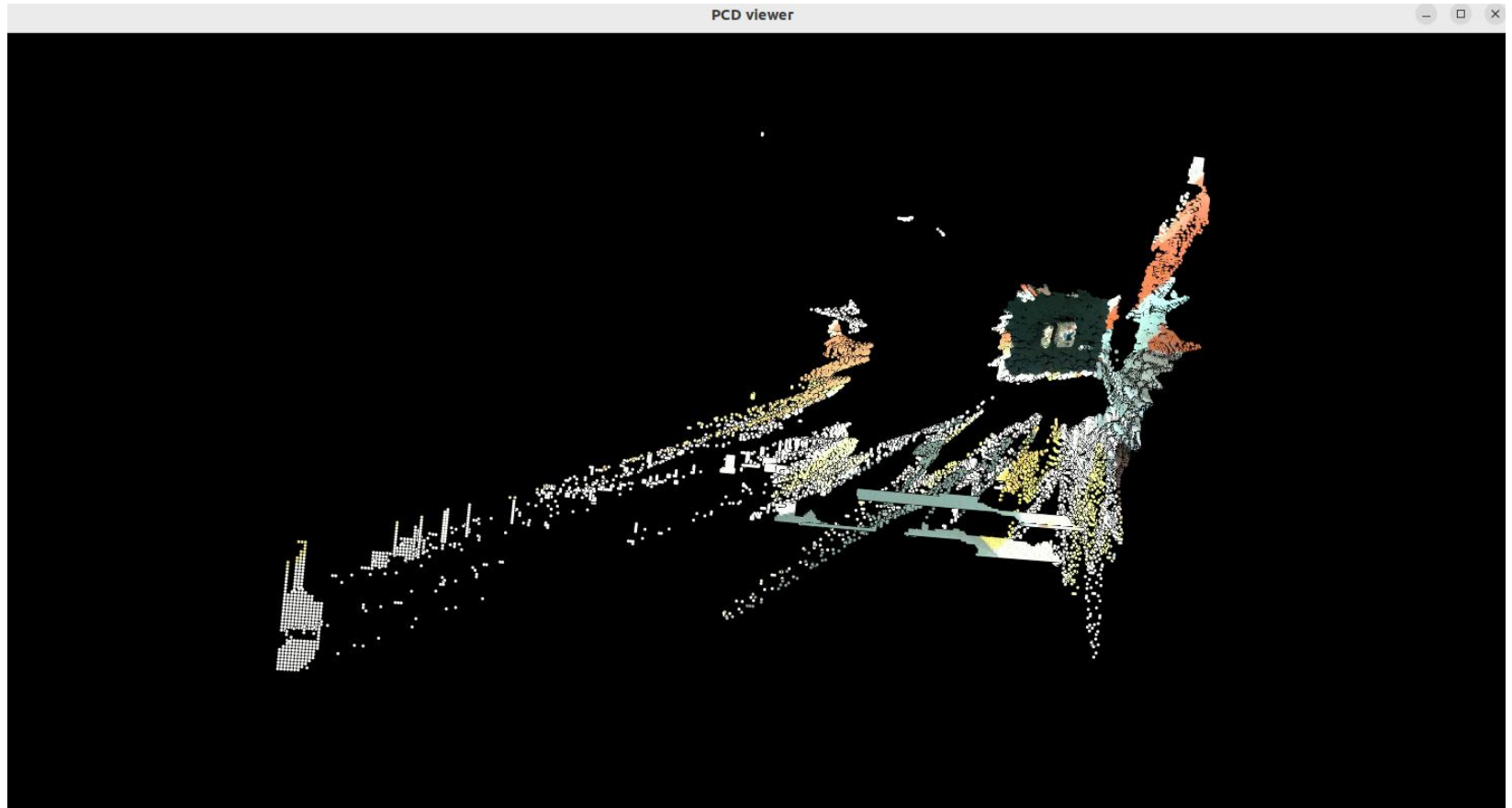


University of  
Zagreb



Faculty of mechanical  
engineering and naval  
architecture

# Zadatak 01 – PC - vizualiziran



# Filtriranje point clouda

- **Filtriranje** je proces **čišćenja, smanjivanja ili izdvajanja** dijela podataka iz oblaka točaka prema određenim kriterijima
- Zašto filtriramo PC?
  1. **Uklanjanje šuma:** senzori često vraćaju točke koje su netočne (npr. refleksije, praznine, artefakti)
  2. **Smanjenje količine podataka:** olakšava i ubrzava algoritme koji slijede (registracija, klasifikacija, grasping...)
  3. **Fokus na određeni dio scene:** npr. gledamo samo prostor ispred robota ili iznad stola



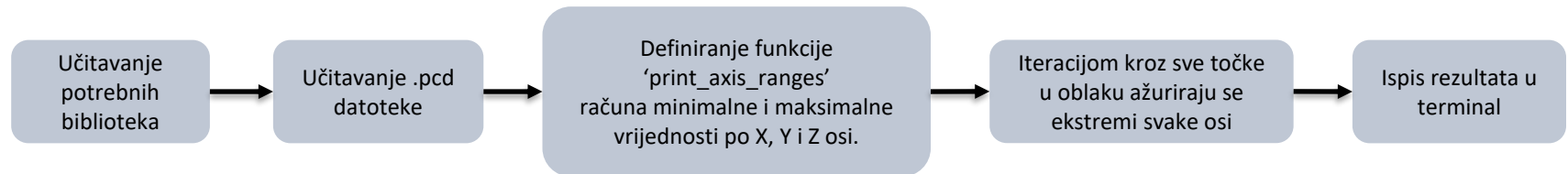
# Filtriranje point clouda

- Kada koristimo filtriranje?
  1. Prije vizualizacije (da bude preglednije)
  2. Prije segmentacije objekata
  3. Kao priprema za 3D rekonstrukciju, registraciju, grasp planning

## Primjer 03 – dimenzije unutar PC

- Filtriranjem PC-a mijenjamo njegovu veličinu (size), ali i prostor u kojem se nalazi
- Kako bi lakše odredili granice koje koristimo za filtriranje (to postavljamo ručno), korisno je moći provjeriti dimenzije (**prostorne**)

# Primjer 03 – dimenzije unutar PC



```
Učitano točaka: 76573
```

```
[Ulazni oblak] Raspon koordinata:
```

```
X: [-2.39593, 4.94801]
```

```
Y: [-3.68209, 1.55603]
```

```
Z: [0.0894, 6.5535]
```

Vrijednosti za  
*'cloud\_1.pcd'*

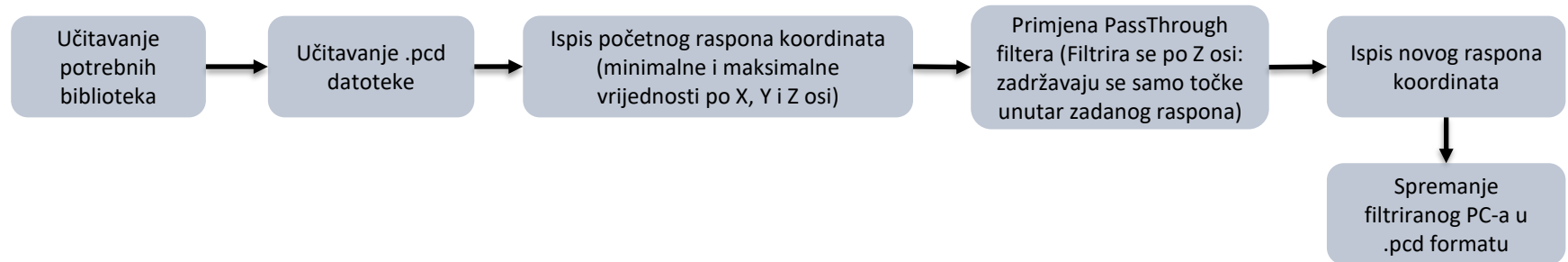
# PassThrough filter

- **PassThrough filter** = alat u PCL-u koji omogućuje filtriranje točaka prema vrijednostima uzduž jedne dimenzije (osi)
- Kako radi?
  - Filtrira sve točke koje **ne pripadaju** zadanoj granici
  - Čuva samo one točke koje zadovoljavaju kriterij
  - Ostale odbacuje → ne pohranjuju se u izlazni oblak točaka

# PassThrough filter

- Česte primjene:
  - Uklanjanje poda ili stropa iz scene
  - Ograničavanje scene na određeni volumen prostora (npr. zona hvatanja robota)
  - Izdvajanje objekata na određenoj visini

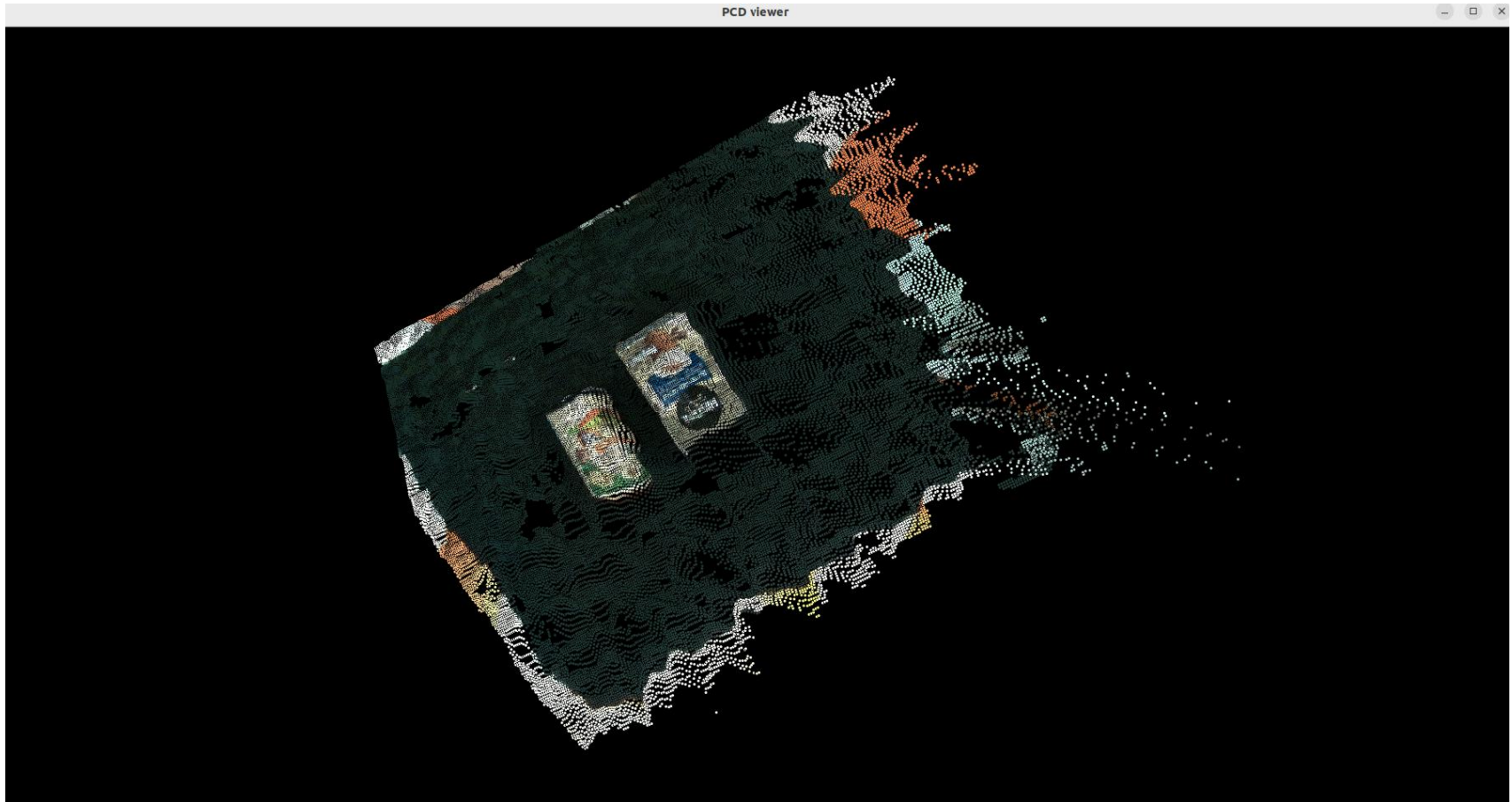
# Primjer 04 – PassThrough filter



**Zadatak:** pokrenuti datoteku i vizualizirati rješenje

- Učitati datoteku '*cloud\_1.pcd*'
- Filtrirati po z-osi na način da se zadrže podaci od 0.15 do 1.0 metara po z-osi, a sve ostale odbacujemo
- Spremamo PC kao '*cloud\_1\_ptf.pcd*' gdje ptf označava pass through filter
- Vizualizirati rješenje

# Primjer 04 – PassThrough filter - vizualizacija



# StatisticalOutlierRemoval filter

- **StatisticalOutlierRemoval filter** = Filter u PCL-u koji uklanja "izolirane" točke iz oblaka točaka
- Koristi statističku analizu udaljenosti između točaka da bi identificirao i uklonio šum
- Kako radi?
  - Za svaku točku pronađe K najbližih susjeda (MeanK)
  - Računa se prosječna udaljenost do tih susjeda
  - Računa se globalni prosjek ( $\mu$ ) i standardna devijacija ( $\sigma$ ) svih tih udaljenosti
  - Svaka točka čija je udaljenost od susjeda veća od izraza se izbacuje

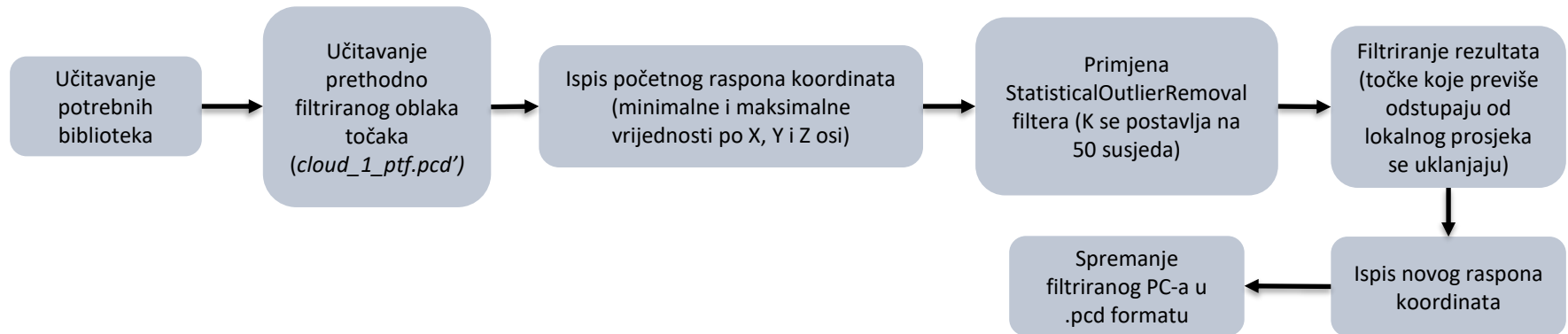
$$outlier\_distance = \mu + \sigma$$



# StatisticalOutlierRemoval filter

- Česte primjene:
  - Uklanjanje šuma nakon akvizicije point clouda (posebno s jeftinijim senzorima)
  - Čišćenje slabo strukturiranih podatak
  - Priprema podataka za segmentaciju, registraciju ili rekonstrukciju

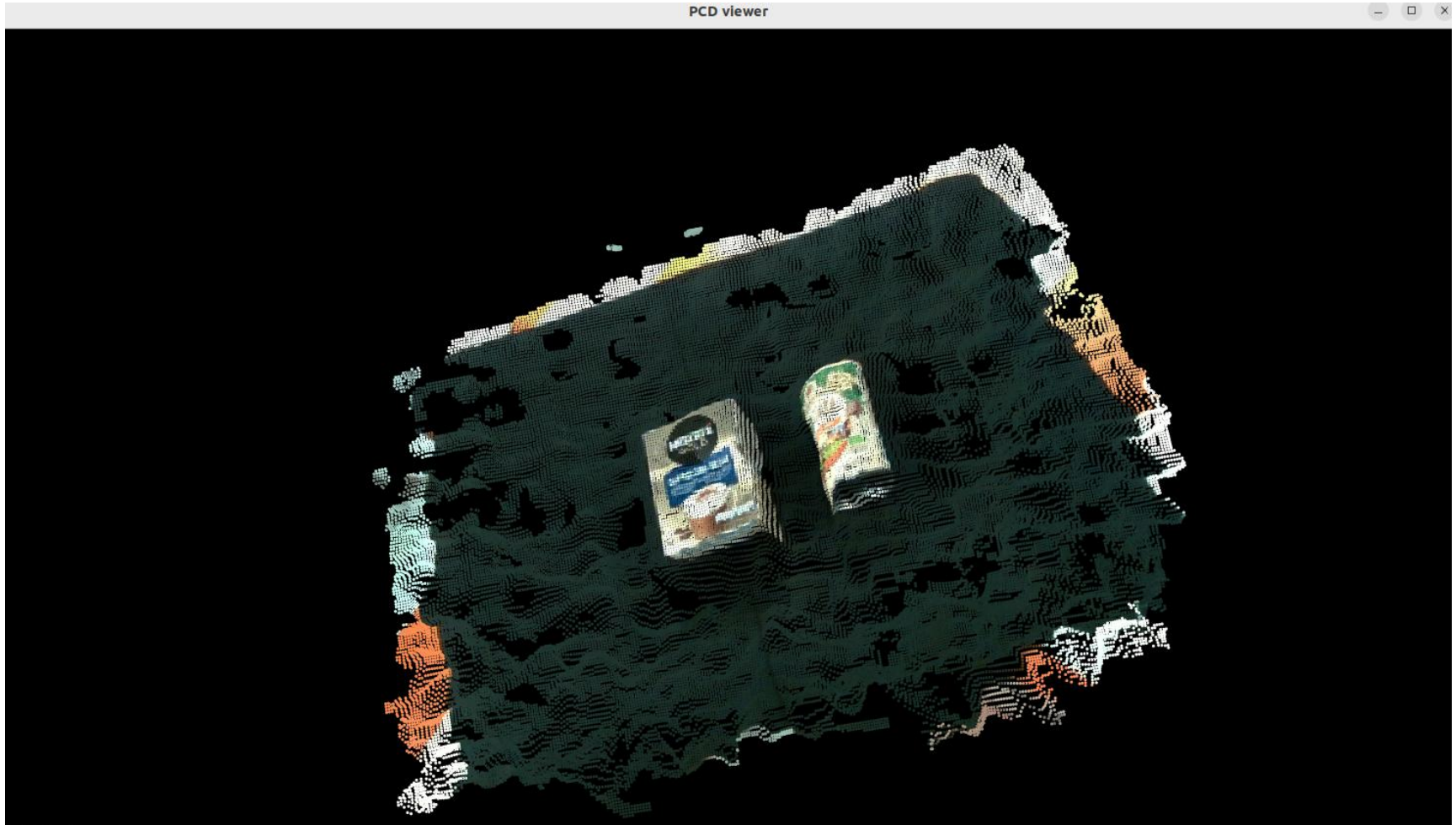
# Primjer 05 – StatisticalOutlierRemoval filter



**Zadatak:** pokrenuti datoteku i vizualizirati rješenje

- Učitati datoteku '*cloud\_1\_ptf.pcd*'
- Filtrirati na način da za svaku točku u oblaku, algoritam pogleda njenih 50 najbližih susjeda i odluči pripada li ona PC-u ili je outlier
- Spremamo PC kao '*cloud\_1\_sof.pcd*' gdje sof označava statistical outlier removal filter
- Vizualizirati rješenje

# Primjer 05 – StatisticalOutlier filter - vizualizacija



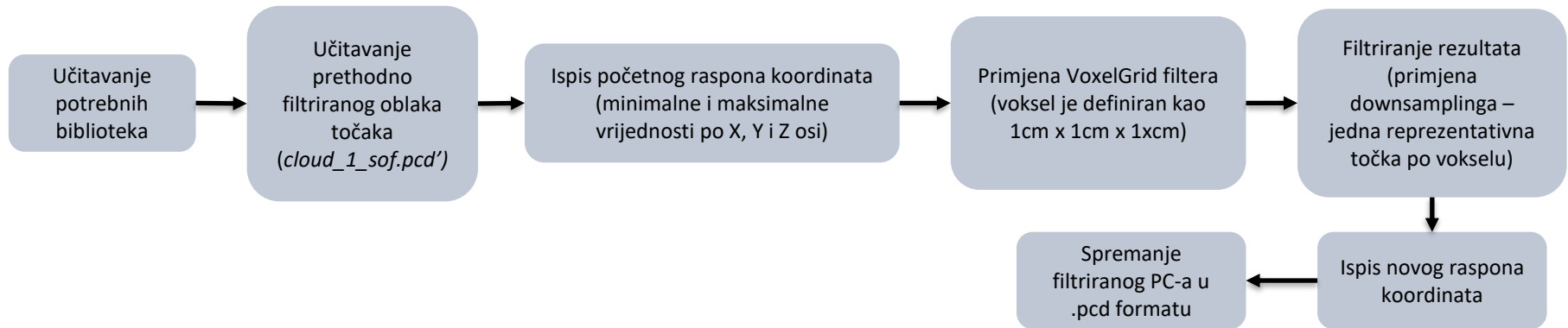
# VoxelGrid filter

- Filter koji smanjuje broj točaka u point cloudu tako da zadrži strukturu, ali smanji podatke
- **Voxel** (od engl. volumetric pixel) je osnovna jedinica volumena u 3D prostoru – mali prostor definirane veličine, u **obliku kocke ili pravokutnog kvadra**
- **Veličina voxela** određuje se parametrima leaf size po X, Y i Z osi, i može biti jednaka (kocka) ili različita (kvadar) po osima
- Sve točke koje se nalaze unutar jednog voxela zamjenjuju se s jednom točkom – obično **centroidom** (prosječnom pozicijom svih točaka unutar tog voxela)

# VoxelGrid filter

- Kako radi?
  - Podijeli prostor point clouda u 3D kocke (voksele) zadane veličine
  - Za svaki voksel uzima jednu točku (centroid) koja zamjenjuje sve susjede oko sebe
- Česte primjene:
  - Priprema podataka prije segmentacije, registracije ili detekcije (manji broj točaka)
  - Ubrzavanje algoritama (manji broj točaka → brža obrada)
  - Smanjenje memorijske potrošnje
  - Očuvanje globalnog oblika objekta uz gubitak fine gustoće

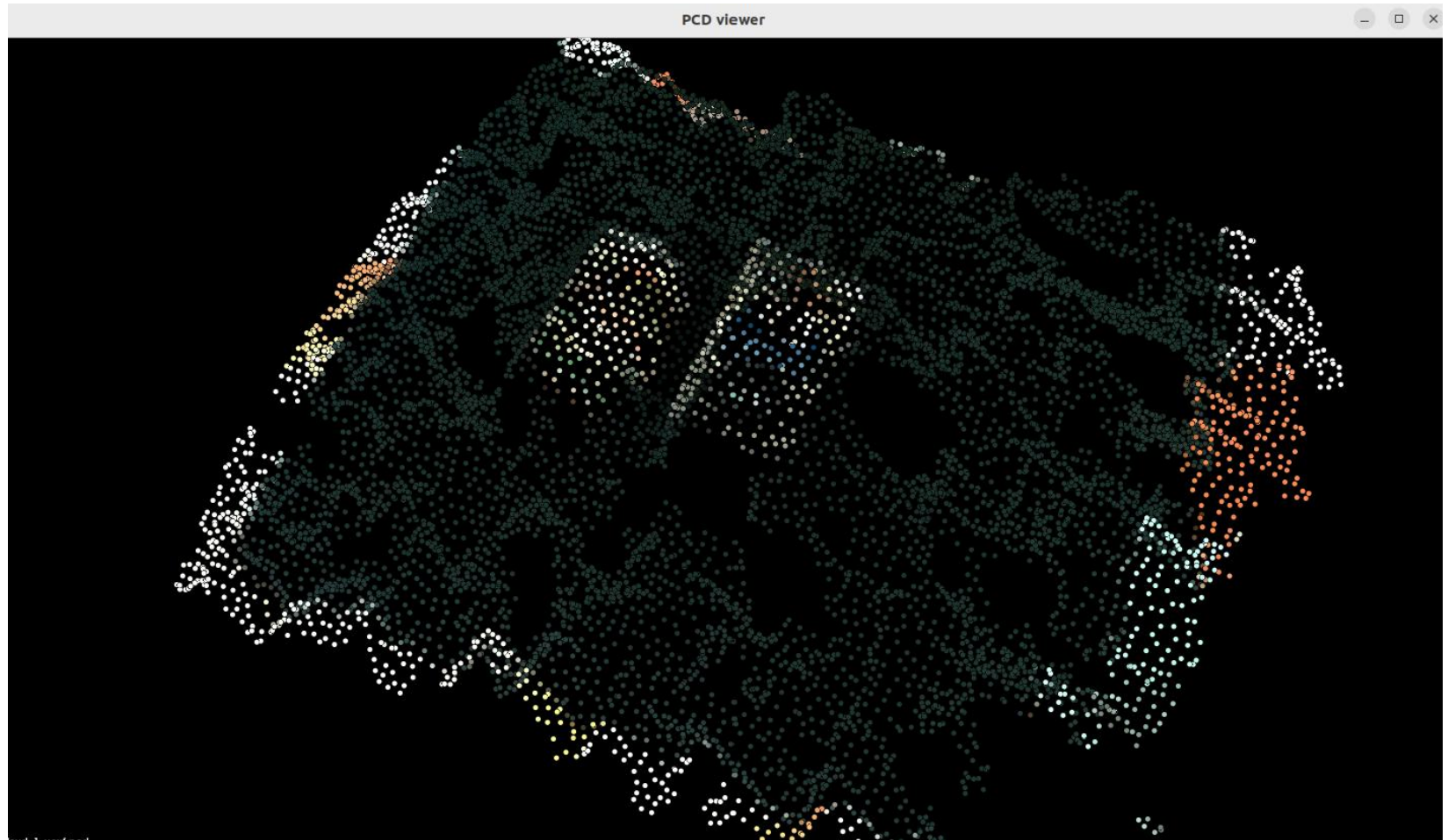
# Primjer 06 – VoxelGrid filter



**Zadatak:** pokrenuti datoteku i vizualizirati rješenje

- Učitati datoteku '*cloud\_1\_sof.pcd*'
- Filtrirati na način da se veličina voksel postavi na 1 cm po svakoj osi (x, y, z)
- Spremamo PC kao '*cloud\_1\_voxf.pcd*' gdje voxf označava voxel grid filter
- Vizualizirati rješenje

# Primjer 06 – VoxelGrid filter - vizualizacija



## Zadatak 02 – obrada PC (filtriranje)



- Zašto ovaj redoslijed?
  1. ***PassThrough*** – brzo uklanja dijelove scene koji nas ne zanimaju
  2. ***StatisticalOutlierRemoval*** – čisti preostali šum i nepouzidane točke
  3. ***VoxelGrid*** – smanjuje broj točaka, ubrzava vizualizaciju i kasniju obradu
- **Cilj zadatka:** Prikazati kako kombinacija različitih filtara omogućuje čišći, manji i fokusiraniji oblak točaka, spreman za npr. registraciju, segmentaciju ili grasping



## Zadatak 02 – obrada PC (filtriranje)

**Zadatak:** iz zadanog PC-a pomoću ***PassThrough*** filtra izoliraj središnji dio scene filtriranjem po X, Y i Z osi, zatim očisti šum pomoću ***StatisticalOutlierRemoval***, i reduciraj broj točaka pomoću ***VoxelGrid***.

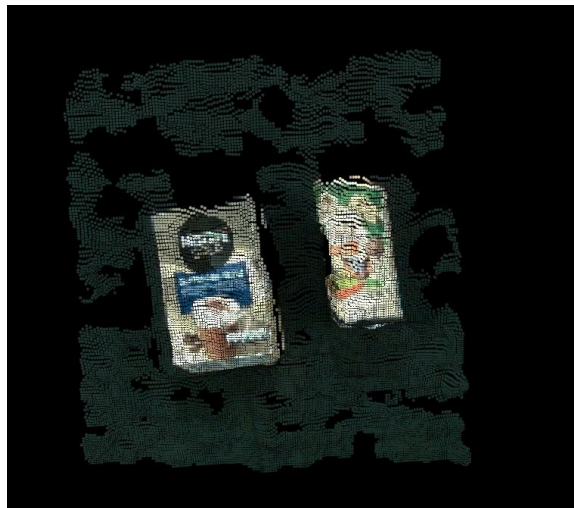
- Kod bazirati na prijašnjim primjerima (možete i njih samo urediti)
- Spremiti rezultate svakog koraka kao zasebnu .pcd datoteku u folder ***data\_z2*** (cloud\_ptfz.pcd, cloud\_ptfx.pcd, cloud\_ptfy.pcd, cloud\_sor.pcd i cloud\_vox.pcd)
- Na kraju, usporedi veličinu datoteka i broj točaka u početnom i zadnjem koraku
- **Napomena: svaki student radi s drugim PC-om**

## Zadatak 02 – obrada PC (filtriranje) – očekivano rješenje

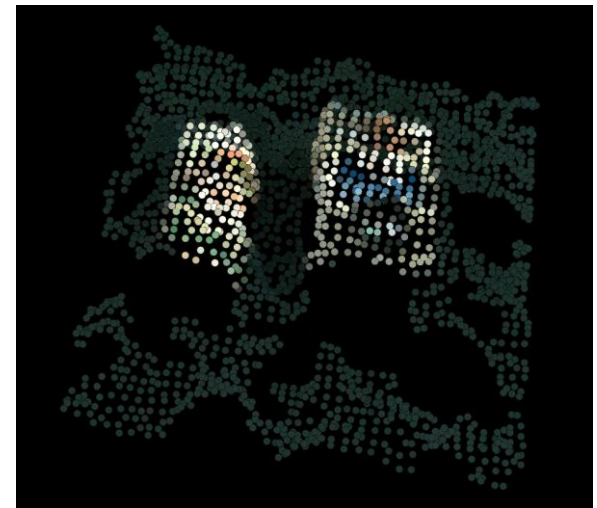
- Očekuje se da kao rješenje prikaže PC u kojem su samo predmeti rada i crna pozadina (ne smiju se vidjeti pod, stupovi niti drugi elementi iz pozadine)



PC nakon PassThrough



PC nakon  
StatisticalOutlierRemoval



PC nakon VoxelGrid