ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

COURSE PROJECT

Computer Graphics

Tái tạo vật thể 3D từ đám mây điểm 3D

Giảng viên hướng dẫn:

Lý Quốc Ngọc

Người thực hiện:

Dương Thị An – 20120240

Phan Đình Anh Quân – 20120635

Phạm Gia Thông – 20120201

2022 – TP.Hồ Chí Minh

Table of Contents

I. Introduction

1. Motivation

+ Scientific significance:

- Interesting general scientific problem
- Creating, recognizing and analyzing objects from physical world
- Reconstructing objects from the physical world to the digital representation

+ Application:

- Virtual reality
- Medical imaging – Simulation of human body
- Archeological exhibitions
- SLAM
- Urban planning
- …

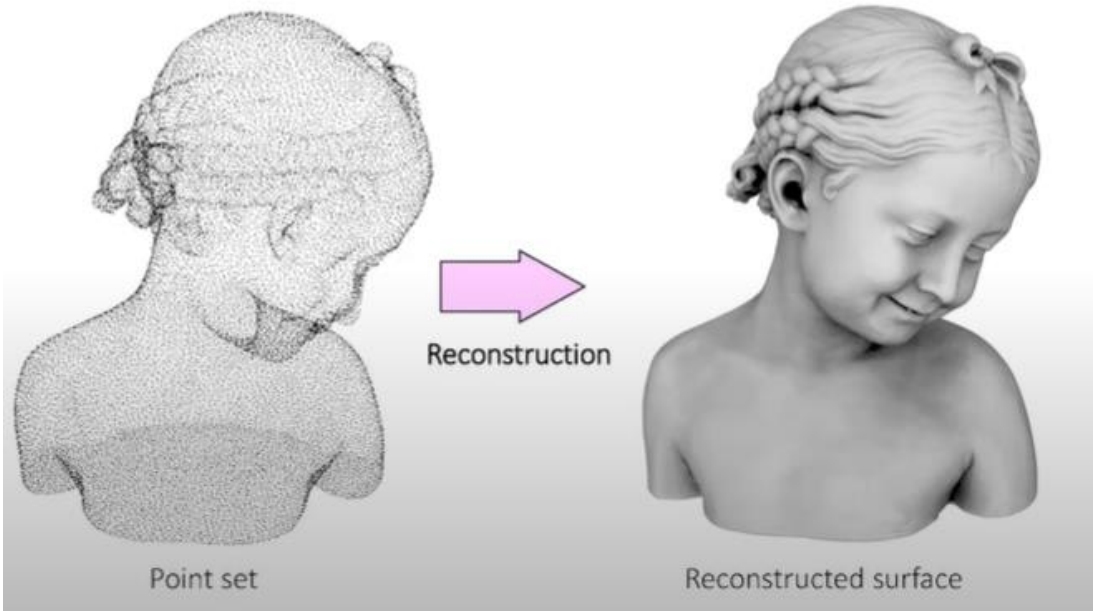
2. Problem statement

A point cloud is a data structure P used to represent a collection of multi-dimensional point $p \in \mathbb{R}^n$. In a 3D point cloud, the elements usually represent the X, Y, and Z geometric coordinates of an underlying sampled surface. When more information about local surface normal n or curvature k, the point $p \in P$ are represented by a longer vector.

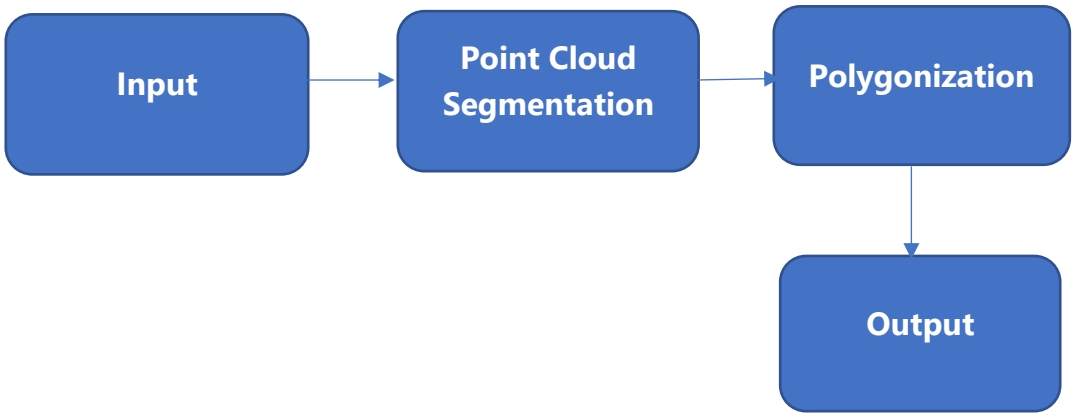
Given a source point cloud P with points $p \in P$ and a target point cloud Q with points $q \in Q$, the problem of registration relies on finding corespondences between P and Q, and estimating a transformation T that, when applied to P, aligns all pairs of corresponding points $p_i \in P, q_j \in Q$.

Input: dense point set P sampled over surface S

Output: surface: approximation of S in terms of topology and geometry



For the framework of the system



Point Cloud Segmentation

The goal of our point cloud segmentation is to divide the oriented points into planar pieces and to remove points which are not located on a planar surface. The point cloud is first divided into clusters with similar surface normals and subsequently according to spatial distances. Finally, all unclustered points are evaluated and potentially added to an existing cluster. The segmentation is performed iteratively on the remaining points until no additional clusters can be found. Usually not all points are assigned to cluster, because the input cloud contains outliers or non-planar areas.

Polygonization

The final reconstruction step generates triangle meshes for the planar point clusters. The points are usually densely distributed, but the segment borders are not exactly defined, and holes may occur due to missing data. We provide two different algorithms for creating polygons: the first one, called point rasterization, only depends on the point cloud itself, while the second one uses image information and consists of three substeps.

II. Introduce to PCL

The Point Cloud Library (PCL) is a standalone, large scale, open project for 2D/3D image and point cloud processing, is an open-source library of algorithms for point cloud processing tasks and 3D geometry processing, such as occur in three-dimensional computer vision. The library contains algorithms for filtering, feature estimation, surface reconstruction, 3D registration,[4] model fitting, object recognition, and segmentation.

Module Surface

The pcl_surface library deals with reconstructing the original surfaces from 3D scans.

Smoothing and resampling can be important if the cloud is noisy, or if it is composed of multiple scans that are not aligned perfectly.

Meshing is a general way to create a surface out of points.

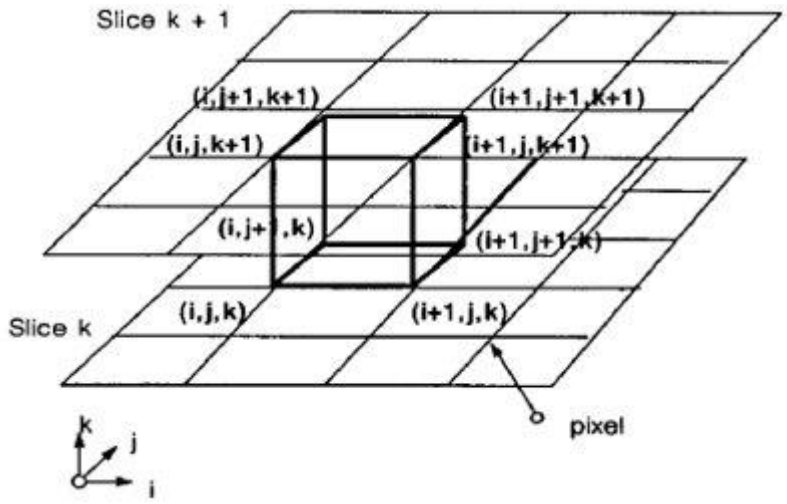
Creating a convex or concave hull is useful.

III. Marching Cube

Marching cubes uses a divide-and-conquer approach in which volume data is processed through voxels. For the processing, multi-image slices are arranged as a mulltidimensional array. And two adjacent slices are taken into consideration at a time.

There are two primary steps in our approach to the surface construction problem. First, we locate the surface corresponding to a user-specified value and create triangles. Then, to ensure a quality image of the surface, we calculate the normals to the surface at each vertex of each triangle.

Marching cubes uses a divide-and-conquer approach to locate the surface in a logical cube created from eight pixels; four each from two adjacent slices.



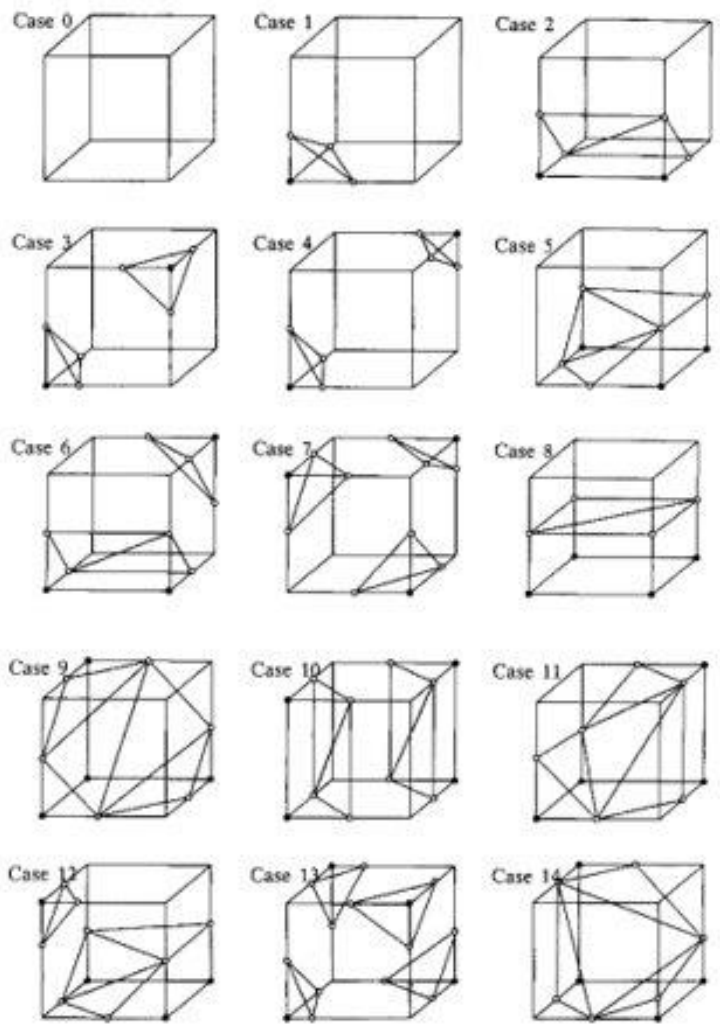
The algorithm determines how the surface intersects this sube, then moves (or marchs) to the next cube. To find the surface intersection in a cube, we assign a one to a cube's vertex if the data value at that vertex exceeds (or equals) the value of the surface we are constructing. These vertices are inside (or on) the surface. Cube vertices with values below the surface receive a zero and are outside the surface. The surface intersects those cube edges where one vertex is outside the surface (one) and the other is inside the surface (zero). With this assumption, we determine the topology of the surface within a cube, finding the location of the intersection later.

Divide the interplanetary inside the limits into a random number of cubes. Then after, the intersection between edges and the isosurface is verified. Test the vertices of every cube for whether they are inside the object. For each cube where some vertices are internal and some vertices are external the object, the surface must pass over that cube, crossing the edges of the cube in between corners of conflicting organization. When the value of the vertex is greater than or equal to isovalue it is internal, and when it is less than isovalue it is external.

Each 8 vertex cells has only two possible states, there is total of $2^8 = 256$ cases of intersection between isosurface and edges. By enumerating these 256 cases, we create a table to look up surface-edge intersections, given the labeling of a cubes vertices. The table contains the edges intersected for each case.

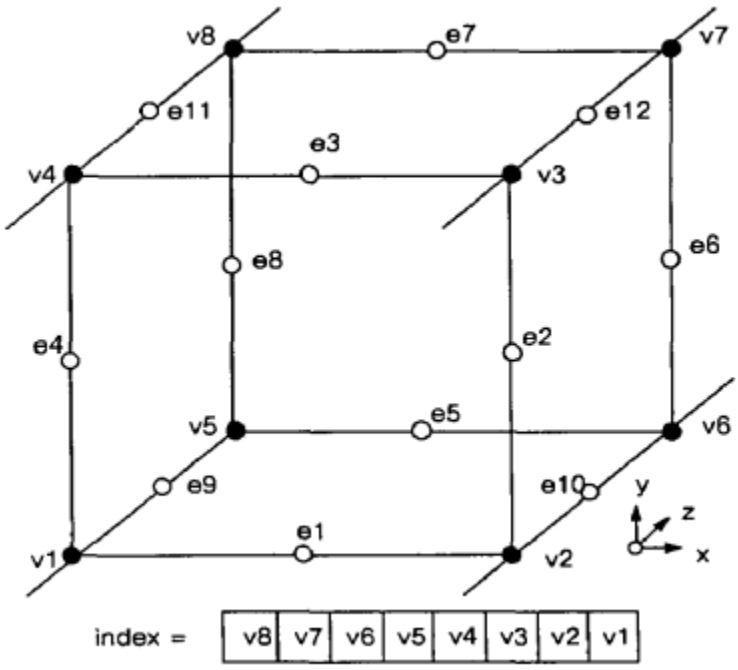
Triangulating the 256 case is possible but tedious and error-prone. Two different symmetries of the cube reduce the problem form 256 case to 14 patterns. First, the topology of the trianguated surface is unchanged if the relationship of the surface values to the cubes is reversed. Complementary cases, where vertices greater than the surface value are interchanged with those less than the value, are equivalent.

Thus, only cases with zero to four vertices greater than the surface value need be considered, reducing the number of cases to 128. Using the second symmetry property, rotational symmetry, we reduced the problem to 14 patterns by inspection.



The simplest pattern 0, occurs if all vertex values are above (or below) the selected value and produces no triangles. The next pattern 1 occurs if the surface separates on vertex from the other seven, resulting in one triangle defined by the three edge intersections. Other patterns produce multiple triangles. Permutation of these 14 basic patterns using complementary and rotational symmetry produces the 256 cases.

We create an index for each case, based on the state of the vertex. Using the vertex numbering, the eight bit index contains one bit for each vertex.



This index serves as a pointer into an edge table that gives all edge intersections for a given cube configuration.

Using the index to tell which edge the surface intersects, we can interpolate the surface intersection along the edge. We use linear interpolation, but have experimented with higher degree interpolations. Since the algorithm produces at least one and as many as four triangles per cube, the higher degree surfaces show little improvement over linear interpolation.

The final step in marching cubes calculates a unit normal for each triangle vertex. The rendering algorithms use this normal to produce Gouraud-shaded images. A surface of constant density has a zero gradient component along the surface tangential direction; consequently, the direction of the gradient vector, 7, is normal to the surface. We can use this fact to determine surface normal vector, $\vec{n}$, if the magnitude of the gradient, $|\vec{g}|$ is nonzero. The gradient vector, $\vec{g}$ is the derivative of the density function

$$\vec{g}(x, y, z) = \nabla \vec{f}(x, y, z)$$

To estimate the gradient vector at the surface of interest, we first estimate the gradient vectors at the cube vertices and linearly interpolate the gradient at the point of

intersection. The gradient at cube vertex (i, j, k) is estimated using central differences along the three coordinate axes by:

$$G_x(i,j,k) = \frac{D(i+1,j,k) - D(i-1,j,k)}{\Delta x}$$

$$G_y(i,j,k) = \frac{D(i,j+1,k) - D(i,j-1,k)}{\Delta y}$$

$$G_z(i,j,k) = \frac{D(i,j,k+1) - D(i,j,k-1)}{\Delta z}$$

Where $D(i, j, k)$ is the density at pixel (i, j) in slice k and Δx, Δy, Δz are the lengths of the cube edges. Dividing the gradient by its length produces the unit normal at the vertex required for rendering. We linearly interpolate this normal to the point of intersection. Note that to calculate the gradient at all vertices of the cube, we keep four slices in memory at once.

In summary, marching cubes creates a surface from a three-dimensional set of data as follows:

1. Read four slices into memory.

2. Scan two slices and create a cube from four neighbors on one slice and four neighbors on the next slice.

3. Calculate an index for the cube by comparing the eight density values at the cube vertices with the surface constant.

4. Using the index, look up the list of edges from a precalculated table.

5. Using the densities at each edge vertex, find the surfaceedge intersection via linear interpolation.

6. Calculate a unit normal at each cube vertex using central differences. Interpolate the normal to each triangle vertex.

7. Output the triangle vertices and vertex normals.

The Marching Cube has been extended in following ways.

- Computational improvements that limit needless effort, especially during traversal, or that use parallel and dispersed processing.
- Hurtling up the marching cubes algorithm on a graphics dispensation unit
- Removal of Ambiguity

IV. Conclusion

Surface reconstruction is a challenging topic due to the wide variability of input data. The quality of input data highly depends on the scene and surface materials, the acquision technique, and last but not least on the time and carefulness spent by the user to capture a scene on-site. Thus, the data is often noisy, contains outliers, or even large parts of the scene are missing. Generally, there are two solutions for tackling these problems, especially for filling in missing data: Putting the user into the reconstruction loop, or using prior information and extensive knowledge about the type of scene.

We believe, that interactive tools will be important for many applications using reconstruc tion. Providing important information in advance is often more practical than detecting and resolving possible reconstruction errors in a post-processing step. An important question for future reconstruction techniques will be, which tasks can be efficiently and robustly done by automatic algorithms and which ones can be easily performed by a human operator.

However, interactive applications are not suitable for all tasks of 3D reconstruction. For example, they are hardly scalable to large reconstructions such as whole cities. For autonomous systems such as robots it is of course also not possible to include manual interventions from a human user. The main challenge for these applications will be to automatically choose the correct prior information depending on the contents of a scene.