3D intersection calculation algorithms on point clouds for mid-air haptic systems

Charlotte Delfosse

June 24, 2020

Contents

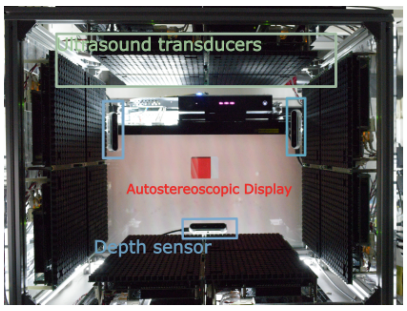
Abstract

Ultrasound emitted from an array of transducers can produce remote tactile sensations on human skin. In the last decade, ultrasound transducers have been used for producing haptic feedback in mid-air haptic systems. Such systems often aim at enabling the users to "feel" virtual objects with different shapes and sizes and possibly interact with them using their hands. However, due to the transducer's limited pressure, the feeling of virtual objects has only been conceived by providing haptic feedback when surface is being touched. In this research, we propose to enhance the user experience of mid-air haptic systems by allowing the inside of tiny virtual objects to be sensed. To achieve the objective, computational geometry techniques have been used on the point cloud of both the virtual objects and the user's hand. After introducing mid-air haptic technology, the principle and results of 3D intersection calculation algorithms for point clouds are presented. Future applications for haptic systems are finally examined.

1 Introduction

The Human-Computer-Interaction (HCI) field has greatly diversified over the past few years. On one end of its wide spectrum, virtual, augmented, and mixed reality are creating increasingly immersive visualization experiences. On the other end of the HCI spectrum are new ways of interacting with machines through the sense of touch, namely haptic technology. Haptic technology refers to devices able to create an experience of touch by interfacing with users through force, vibration, or motion. This trans-disciplinary field includes computer science, robotics, biology, and psychology.

(a) Haptic surgery training tool [1]



(b) Display of mid-air haptic shape with cross sectional feedback (Matsubayashi et al., 2019)

Figure 1. Illustration of a grounded haptic device (a) and a mid-air ultrasonic haptic system (b)

[1] https://www.fundamentalsurgery.com/hapticvr/

Grounded and wearable haptic devices have had a significant impact in the last decade in numerous fields, ranging from the automotive and entertainment to the medical industry. Examples of such devices include screens with haptic feedback, game controllers, gloves, and vests equipped with vibrators. Although those devices may be appropriate to the tasks they are intended to serve, interacting with a haptic system without physical contact can sometimes be preferable.

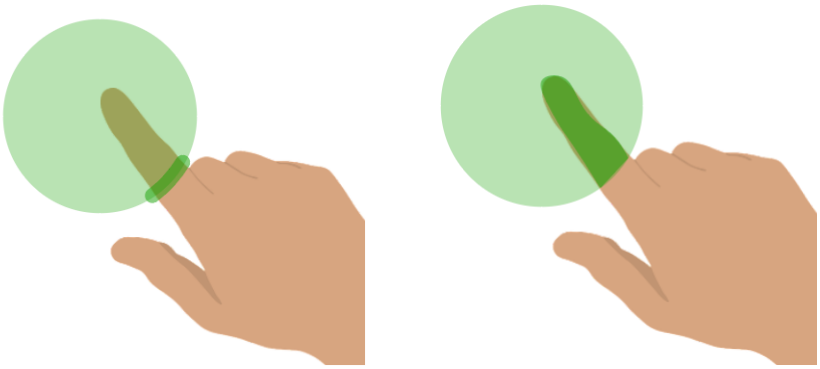
The ultrasonic haptic technology introduced by Hoshi et al. in 2010 enables contactless interaction with haptic systems. The technology relies on arrays of ultrasound transducers to create high acoustic radiation pressure points in mid-air called focus points. By controlling the phase shift and amplitude of the transducers individually, the waves produced by each transducer can interfere at a given point of space with maximal pressure, thereby creating a focus point. Focus points can be felt directly onto users' hands and fingertips, enabling real-time virtual object manipulation. Virtual objects are also made visible by optical processes such as holograms (Kervegant et al., 2017). Examples of such systems are the HaptoMime (Monnai et al., 2014) system that consists of an aerial touch panel with haptic feedback and the HaptoClone (Makino et al., 2016) system that implements a mutual real-time telepresence. Another multi-modal system allowing to feel one's heartbeat in mixed reality with mid-air haptic feedback has also been recently proposed (Romanus et al., 2020). Haptic devices using such non-contact haptic feedback where no wearable is required are called mid-air haptic devices. Fig. 1 shows an example of a grounded haptic device and a mid-air haptic device.

One of the main drawbacks of ultrasonic technology is that the haptic feedback provided is weak. In order to maximize the radiation pressure of the ultrasound waves, feedback has only been provided at the cross section of the users' hand and the virtual objects (Makino et al., 2016; Matsubayashi et al., 2019). This research is an attempt to find a method for feeling the inside of virtual objects in the scope of haptic systems. By touching the inside of virtual objects, we aim at allowing better recognition of shapes. Fig. 2 illustrate the two different haptic feedbacks.

2 Background

2.1 On mid-air haptic systems

Several mid-air haptic systems enabling interaction with virtual objects have been implemented. The array of transducers used in such systems for producing remote tactile sensation on skin is called Airborne Ultrasound Tactile Display (AUTD) (Fig. 3). The current limitations of the technology are a maximal pressure of $50mN/cm^2$ and a spatial resolution comparable to the sound wave length of 40 kHz ultrasound, which is 8.5 mm. Under those constraints, the haptic feedback is relatively weak and not spatially accurate. Radiation pressure patterns can still be controlled precisely by a computer, allowing for a



(a) Haptic feedback at the cross section of a finger touching a sphere

(b) Haptic feedback at the intersection of a finger and a sphere

Figure 2. Illustration of haptic feedback at the cross section of a finger touching a sphere (a) and haptic feedback at the intersection of a finger and a sphere (b) in dark green

range of skin sensation. A key step in providing haptic feedback at the desired position in space, is calculating intersection between the user's hand and the virtual objects; the performance of the algorithm used for this task represents the critical point of a haptic system.
Computing intersection is achieved using one of two hand representations: joint coordinates or point clouds. Joint coordinates of the hand obtained from an image-recognition camera have been used in *Mid-Air Haptic Bio-Holograms in Mixed Reality* (Romanus et al., 2020). Point cloud of a hand obtained by a depth camera have been used in *Display of Haptic Shape Using Ultrasound Pressure Distribution Forming cross sectional Shape* (Matsubayashi et al., 2019). Once the the hand coordinates have been obtained, the intersection with the virtual object can be calculated as a subset of hand points in 3D space. For example, the virtual object manipulation system with autostereoscopic display proposed by Matsubayashi et al. estimates the cross sectional shape by retrieving the points of the hand whose distance to the object points are less than a given threshold radius. A similar method was used in the HaptoClone (Makino et al., 2016) system.

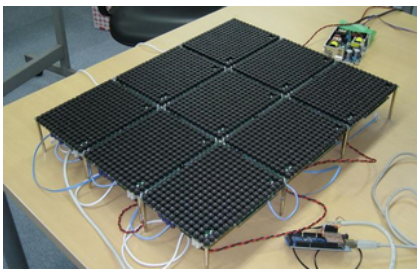


Figure 3. Airborne Ultrasound Tactile Display

2.2 On 3D intersection calculation algorithms

Simply said, the inner sensing of a virtual object can be achieved by identifying which parts of the hand are inside the object. Addressing this problem is essentially realized in two ways: either calculating the intersection of point clouds directly (see section 8 on voxel technique) or reconstructing the underlying surface of either one or both point clouds. Surface reconstruction from 3D point cloud has gained a lot of interest in the past decades with the advent of infrared sensors enabling to extract point clouds from real-time scans of 3D environments. Surface reconstruction can be roughly divided into two categories: continuous and discrete. Each of these categories covers a large set of methods with various implementations. More details on the methods and implementations can be found in *A survey of surface reconstruction from point clouds* (Berger et al., 2017). While continuous reconstruction consists in finding a function that best represents the point cloud, discrete reconstruction focuses on using primitive shapes to generate a mesh. Meshes are a subdivision of a continuous geometric space into discrete geometric and topological cells, usually triangles. This work focuses on discrete surface reconstruction.

3 Problem Statement

3.1 Research goal and first attempt

The goal of this research is to find an efficient 3D intersection algorithm for point clouds and integrate it into a mid-air haptic system.
The research was initially intended to be applied on a larger version of the Haptoclone system, in which interaction between bodies would be possible. In the Haptoclone system, an environment (or scene) A can interact with an environment B in real-time, and vice-versa. The interaction is made possible by using a depth sensor (Kinect v2) in each of the environment and exchanging the data to simulate the existence of environment B within environment A and environment A within environment B (Makino et al., 2016).
Detecting which parts of an object A is contained in an object B essentially requires having a closed surface or representation of object B. However, only partial representation of objects are achievable with one depth sensor, since a sensor can exclusively capture the objects' faces located within its field of view. A minimum of two sensors is therefore required to obtain a usable representation of object B.
A difficult problem is then to correctly align point clouds from different sources, a step known as registration, in a real-time scenario. The problem has not been solved here. Instead, it was decided that the research would be used in a haptic system with pre-loaded models for object B, such as the one developed by Matsubayashi et al.. In this system, only a point cloud of the hand is necessary since the coordinates of object B do not change over time. This system uses tiny object models (size comparable or smaller than the hand),

which allows for more radiation pressure in the case of inner sensing.

3.2 Hypothesis

The research hypothesizes that there is no trivial solution in finding the intersection between the point cloud of a hand and the point cloud of a virtual object, as long as the virtual object presents a non-trivial shape that cannot be expressed by a numerical function. The other hypothesis is that cross section calculation is not sufficient for inferring which points are inside the virtual object using skeletal information (i.e. joint coordinates), as the way a hand sense an object cannot be easily predicted.

3.3 System constraints

Because mid-air haptic systems require real-time performances for the feedback location to update dynamically, the algorithm should also present real-time performances. In the HaptoClone system (Makino et al., 2016), a study on the effect of the refresh rate and delay of the haptic feedback on a dynamic floating image was conducted. The study suggested that users could notice a difference between a 40ms and a 100ms delay, and that the total update time should not exceed the limit of 100ms, above which the user experience would substantially deteriorate. Given that one iteration of such systems includes getting the point cloud, calculating the ultrasound focus point and generating a focus point, it appeared to us that the whole intersection calculation pipeline should be of less than 60ms. Calculation times exceeding this limit are shown in red in the tables.

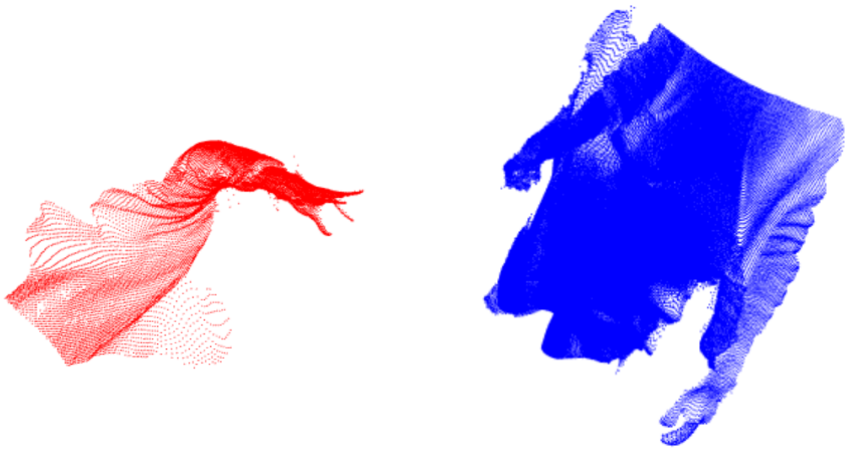
4 Point cloud and dataset

4.1 Definition of a point cloud

A point cloud is a set of data points in a three-dimensional coordinate system. These points are defined here by x, y and z coordinates and represent the outward appearance of an object.

4.2 Dataset

The algorithms were tested with data relevant to the application. A point cloud of an arm and a point cloud of the upper part of a body have been initially used for testing purposes. The point cloud of the arm contains 18545 points and the point cloud of the body contains 95750 points. Those point clouds are unified point clouds that have been acquired from 2 Microsoft Azure Kinect facing each other, 120cm apart (see Fig. 5). The clouds calculated from both Kinects have been aligned coarsely to form the unified point cloud, by a simple change in the second sensor's system coordinates. The result is a rather unclear



(a) Cloud of an arm containing 18545 points

(b) Cloud of a body containing 95750 points

Figure 4. Visualization of the two clouds used as inputs of the algorithms

junction with some missing points, with consequences on surface reconstruction (see concave hull 9). Because the haptic systems in question do not require maximal surface precision for objects, clouds have been downsampled to 1000 points to reduce computing times.

A point cloud of a sphere containing 1605 points with known radius and center has also been used for testing both the voxel and the ray tracing algorithm. The cloud of the Standford bunny (see Fig. 6) was also used.

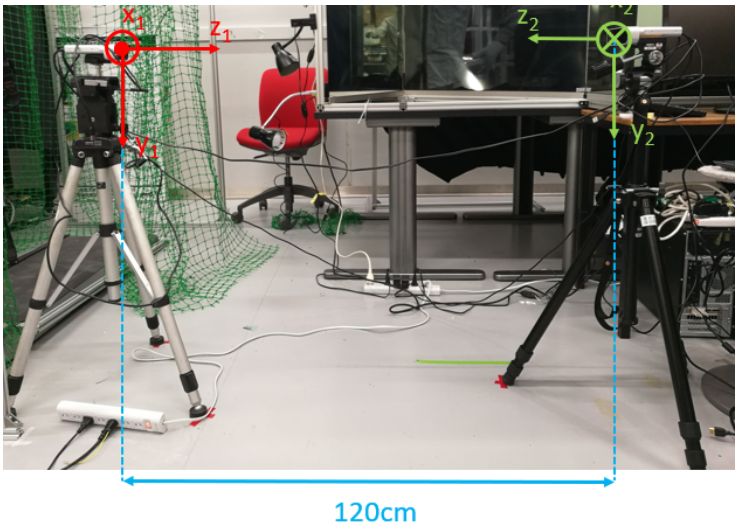


Figure 5. Configuration of the two Azure Kinect used for computing point clouds

5 Point Cloud Library

This research has relied heavily on the Point Cloud Library (PCL) (Rusu and Cousins, 2011) for tests and algorithm implementation. The library pro-

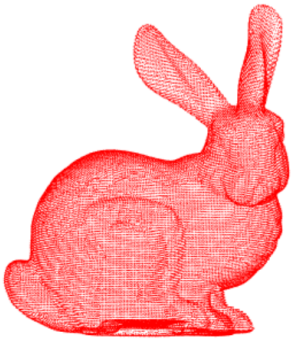


Figure 6. Cloud of the Standford bunny containing 35947 points

vides a wide range of functions allowing multiple point cloud operations, from surface generation to feature estimation and visualization. PCL is divided into 12 modular libraries allowing for different types of operations. Below is a description of the 4 most useful modules for this research.

- *Features Module* : The features module contains data structures and methods for estimating geometric patterns from point cloud data. One of those geometric patterns is normal estimation. This method has been used in section 7.
- *KD-tree Module* : The kd-tree module provides the kd-tree data-structure used for optimizing nearest neighbor search. It has been used in section 7.
- *Surface Module* : The surface module deals with the reconstruction of the original surfaces from 3D scans. Depending on the task, this can be a hull, a mesh representation, or a smoothed/resampled surface with normals. It has been mainly used for computing meshes (see sections 7 and 9).
- *Filters Module* : The filters module contains outliers and noise reduction mechanisms for 3D point cloud data filtering applications. It has been used in this work for downsampling the point clouds and testing the crop hull function that allows filtering points lying outside a concave or convex hull. This method is also detailed in section 9.

6 Downsampling step

PCL provides several ways to downsample a point cloud. One of them is random sampling with uniform probability, which has been mainly used in this work. The algorithm is based on a technique introduced in *Faster Methods for Random Sampling* (Vitter, 1984) and runs in $O(N)$ for a point cloud containing N points. This operation thus takes less than 1ms per cloud on a 2.7 GHz Intel CPU and can be included in the calculation pipeline.

7 Mesh intersection algorithm

7.1 Mesh computing pipeline

The first method used for calculating point cloud intersection was to generate a mesh out of the clouds and calculate their intersection. The first step consists in estimating normal at each of the cloud's point. The next step is computing the mesh itself.

7.1.1 Computing normals

The geometry of a surface can be characterized by its orientation in a coordinate system. Orientation is given by the surface normal at each point of the input cloud, which corresponds to the normal to the plane tangent to the surface at each point. PCL's greedy triangulation algorithm requires to know the orientation, making it necessary to compute normals.

PCL has a normal calculation algorithm in its *features module* that has been used in this work.
For each point p of the input cloud, the algorithm realizes the following operations:

1. getting the k nearest neighbors of p, where k is a parameter of the algorithm)

2. computing the surface normal n of p

3. checking if n is consistently oriented towards the viewpoint and flipping it otherwise

Details about how to find the underlying surface for computing the surface normals and how to compute the normals are not discussed here as the normal calculation steps do not constitute the core of the meshing computation pipeline. An in-depth analysis can be found in *Semantic 3d object maps for everyday manipulation in human living environments* (Rusu, 2010).

7.1.2 Computing meshes

Generating a mesh from point clouds has been realized using PCL's greedy projection triangulation algorithm, which is an implementation of a greedy triangulation algorithm for 3D points based on local 2D projections. As explained in *On fast surface reconstruction methods for large and noisy point clouds* (Marton et al., 2009), the algorithm works by maintaining a list of points from which the mesh can be grown ("fringe" points) and extending it until all possible points are connected. The main steps of the algorithm can be described as follows:

1. A k-nearest neighbor search is performed for each point p of the cloud. The k neighbors are searched in a sphere of radius r using a kd-tree

	normal calculation	**intermediary operations**	**mesh reconstruction**
bunny (35947 points)	224.6	16.4	1809.8
arm (1000 points)	11.4	0.4	856.4

Table 1. PCL's meshing algorithm average running times over 5 runs on a 2.7 GHz Intel CPU (ms)

structure for optimized search. The algorithm allows setting r, the search radius, k, the maximum number of neighbors to search for, along with a μ parameter that specifies the maximum acceptable distance for a point to be considered, relative to the distance of the nearest point. This μ parameter enables to adjust to varying densities.

2. The neighbors are projected on the plane defined by the point p and its normal.

3. Pruning is performed on the neighbors according to several criteria:
   - distance criterion: the algorithm will prune points that are considered too far from p. A kd-tree is used for finding the candidate points (i.e points that will not be pruned).
   - visibility with regards to previously formed triangles: neighbors will be pruned if they are hidden by triangles to avoid self-intersecting faces.
   - minimum and maximum possible angle: if those parameters are set, edges between p and two of its neighbors will be formed only if the angles of the resulting triangle are in between the minimum and the maximum possible value.

The algorithm input is a point cloud with estimated surface normals. The output is a polygon mesh object containing the data of the initial cloud and an array of point indices triplets defining the triangles.

7.1.3 Results

The algorithm was tested with mainly two clouds, the cloud of the arm presented above and the cloud of the Stanford bunny. The latter is convenient and often used in computer science for performing tests, as it is characterized by a low variation of point density, smoothness and shape of medium complexity.

The r parameter value has been chosen after calculating the minimum distance between two points of the cloud and inflating it according to the point cloud's density so that the maximum distance between two neighboring points can become an edge of the mesh.

(a) View 1 (b) View 2

Figure 7. Views of the meshed bunny with $\mu = 2.5$ and $r = 0.003$ (total of 71355 faces)

The results for the Standford bunny can be seen in fig. 7.
Although the resulting mesh has no self-intersecting faces, it presents some holes in it, as visible on the tail of the bunny on View 2.

Fig. 8 shows the result of the algorithm with the arm as an input. The resulting mesh presents some holes and many intersecting faces, due to the input cloud's lack of smoothness.

Table 1 shows the associated average running time for meshing the bunny and a downsampled arm. Intermediary operations consist of concatenating the clouds with their associated normal fields as well as creating a kd-tree structure used in the algorithm for optimization purposes.

Figure 8. View of the meshed arm with $\mu = 100$ and $r = 0.200$ (total of 35544 faces)

7.2 Intersection Calculation

Different algorithms exist for calculating the intersection between meshes. Although not presenting real-time performances, the *winding number* intersection

calculation implemented by the Libigl library (Jacobson et al., 2016) was studied. The approach, which is explained in detail in *Mesh Arrangement for Solid Geometry* article (Zhou et al., 2016), relies on three stages:

1. checking that the input meshes are valid
2. adding the meshes to an *arrangement*, a special partition of space which is further described below
3. extracting the boundary of the result.

For calculating the faces that are part of the intersection and ensuring that input meshes are valid, the algorithm relies on finding *winding numbers*. A *winding number* represents the total number of turns that an object makes around a point of space.

7.2.1 Notion of *winding numbers*

Any mesh M_i can be characterized by a *winding number* field w_i such that:

$$w_i(p) \in \mathbb{Z} \qquad \forall p \in \mathbb{R}^3 \backslash \mid M_i \mid \tag{1}$$

where $\mid M_i \mid$ is the union of all triangles of M_i viewed as point sets. For a triangle mesh, this is simply the sum of the signed solid angle $\Omega_t(p)$ of each oriented triangle of M_i:

$$w_i(p) = \frac{1}{4\pi} \sum_{t \in M_i} \Omega_t(p) \tag{2}$$

7.2.2 Checking the validity of input meshes

For the algorithm to work properly, the first step consists of checking that the input meshes induce a *piecewise-constant winding number* (PWN). If M_i is a PWN mesh, it must divide $\mathbb{R}^3$ into regions that are either outside ($w_i = 0$) or inside ($w_i \neq 0$) of M_i. This requires the input mesh to be closed, have no self-intersections or degeneracies.

7.2.3 Adding the meshes to an arrangement

After resolving degeneracies between the two input meshes so that intersection occurs exactly at edges and vertices, the algorithm creates a *mesh arrangement*, which consists of a division of space into cells, defined as follows: two points belongs to the same cell if they can be connected with a curve without going through any faces. Each of the cells is assigned a *winding number* vector which contains its *winding number* in regard to the different meshes. In our case, as only 2 meshes are being used, the vector has size 2.

7.2.4 Extracting the boundary of the result

For extracting the resulting intersection, the algorithm retrieves the cells in which *winding number* vector has a value of 1 with regards to both meshes.

7.3 Conclusion and limits of the method

By construction, the mesh intersection calculation algorithm has to be very constraining on the input data: only meshes presenting closed manifold (i.e that can exist in the real world, which excludes intersections and degeneracies) geometry are accepted. The meshing algorithm is not satisfactory on its own to produce valid input data for the intersection algorithm. It is very sensitive to density variations and local roughnesses of point clouds, meaning that degeneracies and holes are extremely difficult to avoid.

Costs of this algorithm are presented in *Fast exact parallel 3D mesh intersection algorithm using only orientation predicates* (Magalhães et al., 2017), along with the cost of other mesh intersection algorithms. Presenting very high costs of several seconds, the technique can not be used for real-time haptic applications.

8 Voxel algorithm

8.1 Principle and complexity

The voxel algorithm is a simple algorithm for calculating the intersection between two point clouds. A "voxel" or "volume pixel", is the name given here to the unit element of a 3D grid.
Given two point clouds A (resp. B) and B (resp. A), $A \neq B$, it calculates the points of A (resp. B) that are part of the intersection between the two clouds. The algorithm relies on the following two steps:

1. Creating a bounding box or 3D grid that encompasses one of the two point clouds, namely A (the one whose points we do not want to appear in the result of the intersection).

2. Finding the points of cloud B that are inside voxel intervals delimited by two voxels containing at least one point of A (those voxels are included in the interval). The points found will be part of the intersection.

The two steps are explained in more detail below.

8.1.1 Creating the 3D grid

For creating the 3D grid, the algorithm simply searches for the minimum and maximum values of x, y, and z in the set of points formed by A (i.e $x_{A_{min}}$, $x_{A_{max}}$, $y_{A_{min}}$, $y_{A_{max}}$, $z_{A_{min}}$, $z_{A_{max}}$).

	M=18545 N=95750
K	
360	61.0
770	123.3
1344	229

	M=18545 N=1000
K	
315	42.0
840	55.0
1340	82.5

Table 2. Voxel algorithm average running times depending on K, M, and N over 5 runs on a 2.7 GHz Intel CPU (ms)

This step is realized in O(N), assuming that A contains N points, as every value of the array has to be tested.

8.1.2 Finding the intersection

To go through the nodes of the grid, we define a step named *inc*. In order to give the voxel a cubic shape, we define the step *inc* as the following:

$$inc = (x_{max} - x_{min})/gridSize \tag{3}$$

where *gridSize* is a parameter corresponding to the number of voxels - 1 on the x-axis of the grid.
The algorithm goes through the grid formed in the direction of increasing x, y and z, with a step of *inc*, such that:

- $x_{A_{min}} \leq x < x_{A_{max}} + inc$
- $y_{A_{min}} \leq y < y_{A_{max}} + inc$
- $z_{A_{min}} \leq z < z_{A_{max}} + inc$

When traversing a line of the grid (i.e x and y fixed, z ascending), as long as a point p_A of cloud A has been found in a voxel v_i and a point p'_A of cloud A has not been found in a voxel v_j such that $i \neq j$ and $p \neq p'$, the algorithm pushes any point p_B of cloud B found in the voxel interval into the intersection data structure.

Assuming that the grid contains K nodes and that cloud A contains N points and cloud B contains M points with $N \geq M$, the complexity for finding the intersection is $O(K \times N)$.

8.2 Results

8.2.1 Test on the arm and the body clouds

Figure 9 shows the graphical result of the voxel algorithm on the arm and the body. The green point cloud represents the result of the intersection. Running times for this algorithm using the arm and the body are presented in table 2.

8.2.2 Test on the arm and the sphere clouds

The algorithm has also been tested using the point of a downsampled arm (1000 points) and the triangles of the sphere. As explained in section 4, the

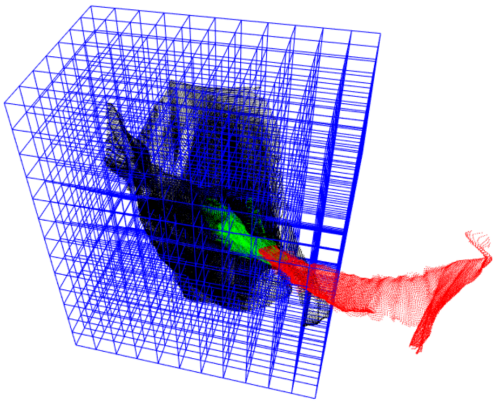


Figure 9. Visualization of the result of the Voxel algorithm with $K = 1344$ using the original arm and body point cloud

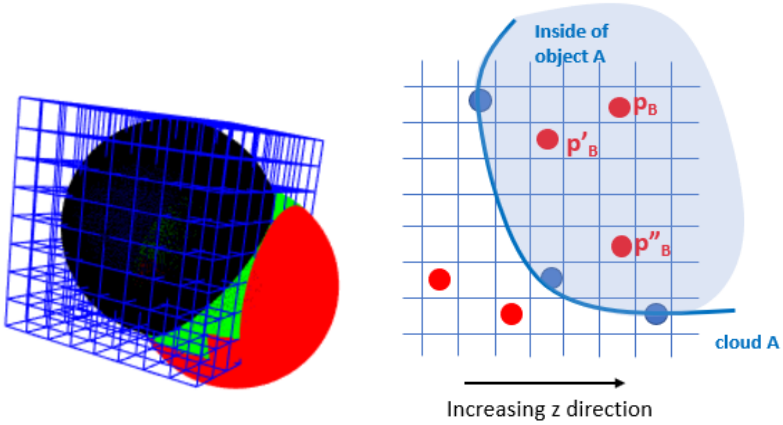
number of voxels	**1150**	**3375**	**8100**	**27050**
number of points in result	139	156	147	144
number of missing points	80	54	61	58
number of extra points	23	14	12	6
accuracy	0.529	0.676	0.649	0.683
running time (ms)	11	30	80	220

Table 3. Result of the intersection of the point cloud of a downsampled arm (1000 points) and the point cloud of the sphere using the voxel algorithm over 3 runs on a 2.7 GHz Intel CPU (ms). 196 points are expected in the result.

radius r and center $c_(x_0, y_0, z_0)$ of the sphere are known so that which points of the arm A should be inside the intersection can be precisely calculated.
Let S be a sphere with radius $r \in \mathbb{R}$ and center $c_(x_0, y_0, z_0) \in \mathbb{R}^3$. Let I be a subset of A such that each point of I is contained in S. A point $p_(x, y, z)$ of A inside I can be defined by the following formula:

$$\forall p_(x, y, z) \in A : p \in I$$
$$\Leftrightarrow$$
$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 <= r^2$$

Running times and accuracy using the arm and the sphere are presented in table 3. The missing points correspond to the points that are contained in the expected result and missing in the algorithm output. The extra points correspond to points that are not expected yet present in the algorithm output. Notice that when $gridSize >= 15$, the accuracy is globally stable.



(a) *inc* too high (b) *inc* too small

Figure 10. Illustration of a small *inc* parameter (a) and visualization of a high *inc* parameter (b)

8.3 Discussion

The *inc* parameter has to be chosen wisely according to the density of both clouds. In fact, a high value for *inc* means that too many points of B will be part of the intersection, resulting in troublesome cubic shapes that could be compared to aliasing in computer graphics, due to the use of voxels (see Fig. 10 (a)). In return, a low value of *inc* means that some points of B we want to be part of the intersection could be missed. Fig. 10 (b) illustrates this problem: although p_B would be considered by the algorithm as part of the intersection, both p'_B and $p"_B$ would be considered outside the intersection.

9 Ray tracing algorithms

Ray tracing is a widely used technique in computer graphics for rendering 3D scenes by reproducing reflection and refraction of lights on object surfaces. To successfully capture how light interacts with an object, the technique uses rays to models light beams and explicit representation of objects in the form of triangular meshes. The intersection between the ray and triangles is then calculated. In this research, ray tracing has been used for finding if points lie inside an object, after creating a convex or concave hull from the point cloud of the object. PCL's *surface module* was used for this first step. The same ray-triangle intersection calculation as in PCL's *crop hull* algorithm (*filters module*) was also used.

Figure 11. Convex hull of the point cloud of the body

9.1 Hull algorithm

9.1.1 Convex Hull algorithm

The convex hull of a set of points is the smallest convex set that contains the points. A convex set C_s can be defined as follows:

$$\forall A, B \neq A \in C_s : [AB] \subset C_s \tag{4}$$

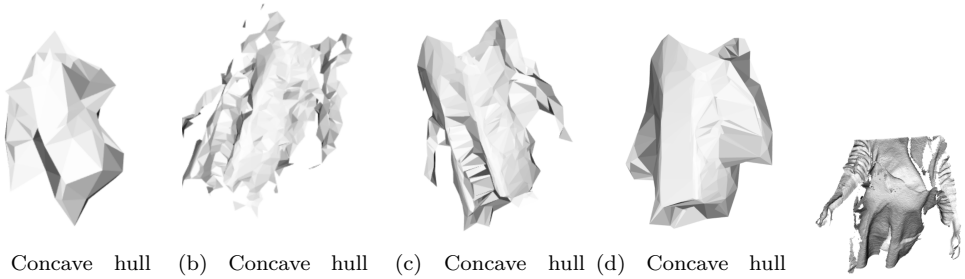
A convex set C_s is a set of points such that, given any two points A, B in that set, the AB-segment joining them lies entirely within that set as stated by equation 4.

PCL's convex hull implementation relies on libqhull library, which implements the QuickHull method (Barber et al., 1996). The method uses a divide and conquer approach to create the hull. Its worst case complexity for 3D space is considered to be $O(nlog(r))$ where n is the number of points in the input cloud and r is the number of processed points. Computing a convex hull for a cloud containing 100000 points is thus a free operation (about 0 ms on a 2.7 GHz Intel CPU). Fig. 11 shows the resulting convex hull of the body point cloud.

9.1.2 Concave Hull algorithm

Like the 3D convex hull, the 3D concave hull corresponds to the volume that minimizes the area that contains all the points. The difference is that the 3D convex hull allows for any angle between faces, including angles greater than 180°. Although a convex hull is unique for a given set of points, many concave hulls are possible for a given set of points. All sets of hulls between the 3D convex hull and the one that completely minimizes the volume are considered concave.

Many algorithms exist for computing the convex hull of a set of points. PCL algorithm is based on Delaunay triangulation and alpha shapes.The α parameter defines the "concavity" of the resulting hull: the algorithm essentially works by removing all faces that have a circumscribed sphere with a radius greater than α.



(a) Concave hull of a downsampled cloud of the body (100 points), α = 99

(b) Concave hull of a downsampled cloud of the body (1000 points), α = 30

(c) Concave hull of a downsampled cloud of the body (1000 points), α = 50

(d) Concave hull of a downsampled cloud of the body (1000 points), α = 99

(e) Concave hull of the cloud of the body, $\alpha = 3$

(f) Concave hull of the cloud of the body, $\alpha = 10$

(g) Concave hull of the cloud of the body, $\alpha = 30$

(h) Concave hull of the cloud of the body, $\alpha = 50$

(i) Concave hull of the cloud of the body, $\alpha = 80$

(j) Concave hull of the cloud of the body, $\alpha = 99$

Figure 12. View of concave hulls made from the cloud of the body

Fig. 12 shows the result of the algorithm for different downsampling of the cloud of the body and different values of α. Time for computing the hull of a cloud containing N points are presented in table 4. This time is not to be taken into account in the haptic system as only pre-loaded object models are used.

N	**100**	**1000**	**95750**
running time (ms)	6.3	32	3388

Table 4. PCL concave hull algorithm average running times depending on N over 5 runs on a 2.7 GHz Intel CPU (ms)

9.2 Crop Hull algorithm

9.2.1 Principle

PCL implements an algorithm called *crop hull* that implements a method known as *crossing number*. It consists of a ray-tracing test on each point of a cloud to decide whether it is inside the hull or not. In PCL algorithm, 3 rays originating from the point to be tested are used. If at least two rays intersect an uneven number of faces, the point is considered to be inside the hull.
Tests on several meshes allowed us to realize that the output of PCL crop hull

algorithm was not correct as the number of intersections found for each ray was much higher than it should have been. Another algorithm using the same method was therefore implemented.

9.2.2 Ray-triangle intersection calculation

Principle Ray-triangle intersection can be calculated following different methods. The algorithm introduced by Möller-Trumbore (Möller and Trumbore, 1997) is considered one of the fastest. While Möller-Trumbore algorithm directly solves a linear system of equations using Cramer's rule and then evaluates determinants, the method use here calculates the parametric coordinates of the intersection point in the plane. The method is the same as the one used in PCL crop hull algorithm, and first calculates the intersection of the ray with the plane containing the triangle. Let R be a ray with origin $P0$ and passing through $P1$ and T a triangle with vertices $V0$, $V1$ and $V2$. Let P be the plane containing T with normal vector $n = (V1 - V0) * (V2 - V0)$. The intersection between R and P is first determined. If no intersection is found, no intersection exists between R and T. If an intersection point P_I is found, the next step consists in determining if P_I lies inside T. The equation of the plane P is given by the following parametric plane equation:

$$V(s,t) = V0 + s(V1 - V0) + t(V2 - V0) = V0 + s\mathbf{u} + t\mathbf{v} \tag{5}$$

Where **u** and **v** represent the directing vectors of P.
Since P_I lies in plane P, $P_I = V(s,t)$. P_I thus lies in the triangle if $s >= 0$, $t >= 0$ and $s + t <= 1$. Let s_I and t_I be the particular values of s and t in the aforementioned conditions.

Using the barycentric coordinate computation, it can be demonstrated that the parameters s_I and t_I can be expressed in the following terms:

$$s_I = \frac{(u \cdot v)(w \cdot v) - (v \cdot v)(w \cdot u)}{(u \cdot v)^2 - (u \cdot u)(v \cdot v)} \tag{6}$$

$$s_I = \frac{(u \cdot v)(w \cdot u) - (u \cdot u)(w \cdot v)}{(u \cdot v)^2 - (u \cdot u)(v \cdot v)} \tag{7}$$

Result After making a convex hull out of the sphere, the crop hullalgorithm for the arm and the sphere was run. Table 5 shows the result.

Discussion The algorithm output is very closed to the expected result (0.995 accuracy) but the calculation time is too high (220.7 ms). Indeed, for each point a test is performed on each triangle, which lead to a brute force method that can be optimized.
Although the algorithm behaves well on the test, one of the drawbacks of the method is that the limit case of a ray intersecting with an edge or vertice is not considered. If a ray is launched on the edge of a triangle or on a vertice,

number of points in result	195
number of missing points	1
number of extra points	0
accuracy	0.995
number of tests	2328000
running time (ms)	220.7

Table 5. Result of the intersection of the point cloud of a downsampled arm (1000 points) and the point cloud of the sphere using the crop hull algorithm upon 3 runs on a 2.7 GHz Intel CPU (ms). 196 points are expected in the result.

the triangles sharing the same edge or vertice will be considered as intersecting whereas the intersection should only be counted once. This problem has been solved in more recent ray-tracing algorithm (Woop et al., 2013).
It should also be noted that overlapping faces on the hull should be avoided as much as possible for the algorithm to work properly, since an intersection with overlapping triangles would alter the crossing number.

9.2.3 Improvement with Bounding Box

The brute-force algorithm described above cannot be used as such since the real-time constraint is not met. In order to reduce the number of ray-triangle intersection tests, we propose to add a simple test prior to the ray-tracing which consists of determining if the point to test lies in the bounding box of the meshed object. If the point does not lie inside, it also will not lie inside the mesh. The intersection calculation with all triangles of the mesh is thus avoided in this case.
The bounding box is calculated in the same way as in 8.1.1.

Results Results are shown in figure 6. The number of tests has been divided by a factor of 4 and the running time by a factor of 5.
The resulting intersection is the same as the one obtained without using the bounding box since the intersection calculation method has not changed.
Note that the improvement exists only if the number of points outside the intersection area is significant, which is generally the case in our scenario (meshed objects of comparable or smaller size than the hand).

9.2.4 Improvement with Kd-tree

Principle Kd-tree is an often preferred structure for accelerating ray tracing and exists in many variations. Mary variations are covered in detail in (Hapala and Havran, 2011). By dividing the space containing all triangles into sub-space containing a sub-set of triangles, kd-tree effectively reduces the number

number of points in result	195
number of missing points	1
number of extra points	0
accuracy	0.995
number of tests	614592
running time (ms)	45.3

Table 6. Result of the intersection of the point cloud of a downsampled arm (1000 points) and the point cloud of the sphere using the crop hull algorithm with bounding box prior check upon 3 runs on a 2.7 GHz Intel CPU (ms). 196 points are expected in the result.

of intersection tests. The PCL kd-tree exists only for storing points and not triangles so the implementation was carried out by using several pre-existing algorithms. After building the kd-tree, the structure is traversed using depth-first search (DFS) for intersection finding.

Building the kd-tree follows the 3 next steps:

1. A bounding box encompassing the mesh and all of its triangles is assigned to the first node
2. The following two criteria are used to divide the first node in two child nodes:
   (a) The axis (x, y, or z) according to which the box has the longest dimension is first found. Let max_{dim} be that axis.
   (b) The median med of the max_{dim}-coordinate of the first vertice $V0$ is calculated for all triangles. All triangles whose max_{dim} coordinate is inferior to med goes to the left child, the rest to the right child.
3. The build function is called recursively for all nodes. When a node has a child containing no triangles, this node becomes a leaf.

Traversing the kd-tree follows the next steps:

1. For each point of the cloud, a ray with random direction is cast.
2. While an intersection is found with the bounding box assigned to a node, the kd-tree intersection function is called on the child node in a DFS manner.
   (a) If no intersection is found with the bounding box of any given node, the algorithm prunes the current path, i.e none of the child nodes will be visited.

(b) If intersection is found with the bounding box of a leaf, the ray-tracing algorithm is applied on the triangles of the leaf.

The tree contains 1341 nodes and 16 levels. It is not a balanced tree. Note that unlike the implementation that would be preferred in computer graphics, where there is no need to continue running the algorithm once an intersection has been found, here it is crucial to get all the intersection as the *crossing number* is used to determine whether a point is inside a mesh or not.

number of points in result	**number of missing points**	**number of extra points**	**accuracy**	**number of ray-triangle tests**	**time for building kd-tree (ms)**	**time for intersection finding (ms)**
196	1	0	0.995	9965	2.7	1.7

Table 7. Result of the intersection of the point cloud of a downsampled arm (1000 points) and the point cloud of the sphere using a ray-tracing algorithm with kd-tree prior check over 3 runs on a 2.7 GHz Intel CPU (ms). 196 points are expected in the result.

Results The results are shown in table 7. The number of intersection tests have been reduced by a factor of 6 compared to the bounding box improvement, and the algorithm runs in real-time.
Note that we have included the time for building the kd-tree in the table (i.e. 2.7 ms) but that the application would build it only once as the object is static.

10 Conclusion

Three algorithms were presented in this work for 3D point cloud intersection calculation. Two of them, the mesh intersection algorithm and the voxel algorithm, did not perform satisfactorily enough from a qualitative and performance point of view. Although the ray tracing algorithm was too slow on its own, the output on the tested data indicated that the algorithm was satisfactory in terms of quality. Further, it could be improved using a bounding box and a kd-tree structure, for which the computing time was only a few milliseconds.
Future research on the given problem will involve continuous reconstruction of the underlying surface of the point clouds, using Non-Uniform Rational Basis Splines (NURBS)(Bureick et al., 2016). Analytical resolution methods such as the Newton method might then enable to find the points lying in the reconstructed surface in real-time.
After integrating the most satisfactory algorithm in the existing haptic system, the use of different haptic textures and pressure variation would enable to simulate the presence of materials with different properties (i.e more or less soft), inside a given object. The use of an optical system could also make it possible to visualize the inside of the object when it is touched, making this system a 3D visual and haptic scanner.

11 Acknowledgments

First, I would like to thank professor Hiroyuki Shinoda and professor Yasutoshi Makino for letting me join the Haptic Laboratory of the University of Tokyo. The environment I have been immersed in this year gave me many interesting insights about haptics, a field I knew almost nothing about. Not only did the professors help me develop my ideas, they also encouraged me to continue my research despite the many difficulties I encountered. I am also thankful to my lab colleagues, that gave me technical support and helped me improve my japanese.
A lot of thanks as well to my tutor Vasile Marian Scuturici, who was there to support me despite the exceptional circumstances, whether through email or video exchanges.
Last, I am grateful to INSA and the Computer Science department (IF) for making this very enriching academic exchange possible.

References

C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. 1996. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483.

Matthew Berger, Andrea Tagliasacchi, Lee M Seversky, Pierre Alliez, Gael Guennebaud, Joshua A Levine, Andrei Sharf, and Claudio T Silva. 2017. A survey of surface reconstruction from point clouds. In *Computer Graphics Forum*, volume 36, pages 301–329. Wiley Online Library.

J Bureick, H Neuner, C Harmening, and I Neumann. 2016. Curve and surface approximation of 3d point clouds. *Allgemeine Vermessungs-Nachrichten*, 123(11–12):315–327.

Michal Hapala and Vlastimil Havran. 2011. Kd-tree traversal algorithms for ray tracing. In *Computer Graphics Forum*, volume 30, pages 199–213. Wiley Online Library.

Takayuki Hoshi, Masafumi Takahashi, Takayuki Iwamoto, and Hiroyuki Shinoda. 2010. Noncontact tactile display based on radiation pressure of airborne ultrasound. *IEEE Transactions on Haptics*, 3(3):155–165.

Alec Jacobson, Daniele Panozzo, C Schüller, Olga Diamanti, Qingnan Zhou, N Pietroni, et al. 2016. libigl: A simple c++ geometry processing library.

Cédric Kervegant, Félix Raymond, Delphine Graeff, and Julien Castet. 2017. Touch hologram in mid-air. In *ACM SIGGRAPH 2017 Emerging Technologies*, pages 1–2.

Salles VG Magalhães, W Randolph Franklin, and Marcus VA Andrade. 2017. Fast exact parallel 3d mesh intersection algorithm using only orientation predicates. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–10.

Yasutoshi Makino, Yoshikazu Furuyama, Seki Inoue, and Hiroyuki Shinoda. 2016. Haptoclone (haptic-optical clone) for mutual tele-environment by real-time 3d image transfer with midair force feedback. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 1980–1990.

Zoltan Csaba Marton, Radu Bogdan Rusu, and Michael Beetz. 2009. On fast surface reconstruction methods for large and noisy point clouds. In *2009 IEEE international conference on robotics and automation*, pages 3218–3223. IEEE.

Atsushi Matsubayashi, Hiroki Oikawa, Saya Mizutani, Yasutoshi Makino, and Hiroyuki Shinoda. 2019. Display of haptic shape using ultrasound pressure distribution forming cross-sectional shape. In *2019 IEEE World Haptics Conference (WHC)*, pages 419–424. IEEE.

Tomas Möller and Ben Trumbore. 1997. Fast, minimum storage ray-triangle intersection. *Journal of graphics tools*, 2(1):21–28.

Yasuaki Monnai, Keisuke Hasegawa, Masahiro Fujiwara, Kazuma Yoshino, Seki Inoue, and Hiroyuki Shinoda. 2014. Haptomime: mid-air haptic interaction with a floating virtual screen. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 663–667.

Ted Romanus, Sam Frish, Mykola Maksymenko, William Frier, Loïc Corenthy, and Orestis Georgiou. 2020. Mid-air haptic bio-holograms in mixed reality. *arXiv preprint arXiv:2001.01441*.

Radu Bogdan Rusu. 2010. Semantic 3d object maps for everyday manipulation in human living environments. *KI-Künstliche Intelligenz*, 24(4):345–348.

R.B. Rusu and S. Cousins. 2011. 3d is here: Point cloud library (pcl). In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1 –4.

Jeffrey Scott Vitter. 1984. Faster methods for random sampling. *Communications of the ACM*, 27(7):703–718.

Sven Woop, Carsten Benthin, and Ingo Wald. 2013. Watertight ray/triangle intersection. *Journal of Computer Graphics Techniques (JCGT)*, 2(1):65–82.

Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh arrangements for solid geometry. *ACM Transactions on Graphics (TOG)*, 35(4):1–15.

Bilan Personnel

Cette année d'échange à l'étranger a été très enrichissante pour moi.
Tout d'abord, sur un plan académique, j'ai effectué de la recherche dans un laboratoire pour la première fois. Dans les projets sur lesquels j'ai travaillé à l'INSA, j'étais toujours encadrée par des professeurs qui guidaient les élèves si jamais ils bloquaient. Dans mon contexte, j'ai appris l'indépendance car mon travail était essentiellement individuel. Les professeurs de mon laboratoire n'étaient d'ailleurs pas spécialisés en informatique, ils n'ont pas pu m'aider sur des questions techniques. J'ai appris que la recherche est un travail de longue haleine, ou le résultat est important mais le chemin pour y arriver l'est encore plus.
Il m'est arrivé fréquemment de me démotiver à cause de la dimension individuelle des tâches effectuées. Je n'ai pas eu cette satisfaction à construire un projet à plusieurs et j'en ai ressenti le manque durant cette année. Comme tout un chacun, j'ai également dû faire face à de nombreuses « difficultés » techniques, comme des librairies fastidieuses à installer ou l'utilisation de programmes peu lisibles. Cependant j'ai réussi à surmonter ces difficultés en faisant preuve de résilience et en célébrant chaque succès et chaque nouvelle chose apprise.
A cela s'ajoute le fait que j'étais une des seules étrangères dans mon laboratoire et la seule à ne pas maîtriser le japonais. Toutes les informations relatives au laboratoire ainsi que la quasi-totalité des réunions étaient en japonais. Cela a rendue la communication plus difficile mais a constitué une vraie expérience d'humilité.
Mon seul regret a été de devoir repousser la mise au point d'un système haptique avec les algorithmes sur lesquels j'ai travaillé à cause de la pandémie mondiale, qui a eu pour effet la fermeture de mon laboratoire depuis plus de 2 mois.
Sur un plan culturel, cette année au Japon m'a fait découvrir des modes de penser radicalement différents. Les interactions avec les japonais au laboratoire ou ailleurs, mais aussi avec d'autres étudiants internationaux ont été très instructives. Apprendre une nouvelle langue a été un des points très positifs de cet échange.
Globalement, bien qu'effectuée dans un contexte particulier, cette année m'a donné un aperçu de ce qu'est la recherche. Cela m'a donné envie de continuer, possiblement en R&D dans une entreprise. J'envisage également de m'installer au Japon car je pense avoir encore beaucoup de choses à apprendre ici.