

〈국문초록〉

키넥트 카메라를 이용한 평면인식 알고리즘 개발

연구자 : 윤요한(2학년, yohanme@naver.net)

이종현(2학년, jonghyun777@gmail.com)

이진형(2학년, wlsquddl2001@naver.com)

책임지도자 : 김형석(동의대학교, hskim@deu.ac.kr)

공동지도자 : 김용주(KAIST 부설 한국과학영재학교, kyjgifted@kaist.ac.kr)

요약문

IOT의 여러 분야 중, 물체 추적의 기본이 되는 것은 물체를 배경에서 분리하는 것, 즉 배경을 따로 인식하는 것이다. 대부분의 경우, 배경은 벽 혹은 바닥 등의 평면이므로, 3D 카메라를 통한 평면 인식은 물체 인식 및 추적에 있어서 기본이 된다고 할 수 있다. 이 연구에서는 가격이 비싼 장비가 아닌, 비교적 저렴한 키넥트 카메라를 이용해서 평면을 인식하는 알고리즘을 개발하여, 보다 저렴한 장비로 많은 사람들에게 IOT를 사용하게 할 수 있는 계기를 제공해주고자 한다. 이 연구에서는 여러 가지 개발환경 및 방법으로 평면 인식 알고리즘의 구현을 시도해서, 그 중 가장 결과가 좋았던 것을 소개하고자 한다.

I . Introduction

IOT에 대한 관심이 증가하고 있는 와중에, IOT의 일부인 물체 및 사람 추적에 관한 관심도 증가하고 있다. 하지만 이에 필요한 장비들, 특히 3D 카메라는 가격이 굉장히 비싼 경우가 많다. 하지만 비싼 장비는 일반인들이 쉽게 구할 수 없기 때문에 이는 IOT의 대중화에 악영향을 끼칠 수 있다. 따라서 우리는 이러한 문제점을 해결하기 위해서 보다 저렴한 기술로 물체 추적을 가능하게 할 수 있는 방법을 찾기 위해 이 연구를 시작하게 되었다.

그런 와중, RGB 카메라와 Infrared 카메라를 함께 사용해서 대상의 3D 정보 및 RGB 정보를 동시에 출력해주는 마이크로소프트사의 키넥트 카메라에 대해서 알게 되었다. 키넥트 카메라는 본래 게임용으로 개발되었던 3D 카메라지만, 값이 매우 저렴하고 그에 비해 성능이 나쁘지 않았기 때문에 연구용으로도 많이 사용된다. 따라서 우리도 이를 연구에 사용하기로 했다. 우선, 물체 인식을 하기 위해서는 기본적으로 배경으로부터 물체를 분리해 낼 필요가 있다. 즉, 벽, 혹은 바닥 등의 평면을 제대로 인식해야 그 평면을 물체로부터 분리해 낼 수 있다. 따라서 우리는 앞서 설명한 키넥트 카메라를 이용해서 물체 추적에 기본이 되는 평면인식 알고리즘을 구현해보기로 했다.

II . Theoretical Background

1. The Microsoft Kinect

KINECT는 Microsoft에서 만든 가정용 모션인식 카메라이다. 넓은 인식범위를 가지고 있어 최대 4명의 사람까지 인식이 가능하다. 또한 기존 모션인식 카메라보다 훨씬 싼 가격으로 인해 원래 목적이었던 게임용 주변기기가 아닌 연구용으로 더 많이 쓰이고 있다.

KINECT는 적외선 depth 카메라 한 쌍과 RGB 카메라, 그리고 마이크를 가지고 있다. [그림 1] Depth 카메라는 적외선 점을 물체에 투영해 반사되어 돌아오는 빛의 패턴을 검출하여 깊이를 알아낸다. 이 카메라는 0.8m~3.5m의 물체를 감지해 320 x 240의 해상도로 출력한다.

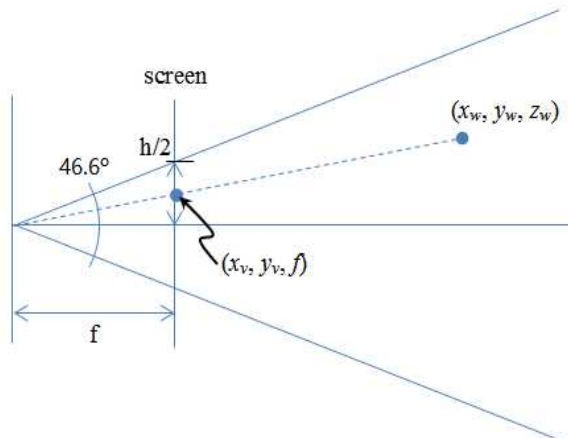


[그림 1]

2. 3-Dimensional Real Coordinates Calibration from the Kinect.

Kinect의 카메라로 찍은 영상으로 가져온 깊이 정보는 각 점의 index와 그 점과 카메라까지의 수평거리, 즉깊이를 나타낸 1차원 리스트 정보로 입력이 된다. 우리는 이 정보를 3차원 공간에서 다루기 위해서는 object의 실제 좌표를 구해내야 한다.

[그림2]는 Kinect를 옆에서 본 그림이다. x_v, y_v 는 화면상의 점의 좌표를 나타내고, x_w, y_w, z_w 는 실제 물체의 좌표를 나타낸다. Kinect에서 받아온 깊이 데이터를 실제 좌표로 변환하려면 먼저 카메라의 초점 거리가 필요하다.



[그림 2]

$$\tan\left(\frac{46.6}{2}\right) = \frac{h/2}{f}, \quad h = 240\text{px}$$

$$f = 278.64\text{px}$$

실제 좌표는 구한 초점거리와 실제 depth 값의 비례식을 세워 구하면 된다.

$$f : y_v = z_w : y_w, \quad z_w = \text{depth} y_w = \frac{y_v \cdot z_w}{f} = \frac{y_v \cdot \text{depth}}{f} \quad [\text{mm}]$$

같은 방법으로 $x_w = \frac{x_v \cdot \text{depth}}{f} \quad [\text{mm}]$ 을 구할 수 있다.

3. PCA method to finding the normal of the group of points in R^n

PCA(Principal Component Analysis)[3]은 서로 연관 가능성이 존재하는 고차원의 데이터인 표본들을 한 개의 축으로 사상시켰을 때 그 분산 정도에 따라 저차원 공간인 주성분의 표본으로 데이터를 새로운 축으로 선형 변환시키는 알고리즘이다. 주어진 데이터를 데이터의 중심점에서 법선벡터로 사영시켰을 시, 그 분산 정도가 최소가 되는 특징을 이용하여, 이 알고리즘을 R^2 나 R^3 공간에서의 데이터들에 적용 시 데이터들을 대표하는 법선 벡터를 계산하는 것이 가능하다.

이를 단계별로 설명하자면 1)먼저 데이터의 중심점 M 을 정의하여, 이를 새로운 원점으로 삼고 각 점들의 좌표($w_1, w_2 \dots w_n$)는 새로운 원점을 기준($u_1, u_2 \dots u_n$)으로 바꾼다. 2)그 후, $u_1, u_2 \dots u_n$ 을 열 벡터로 갖는 행렬 U 를 만들고, 분산 행렬 S 를 $S=UU^T$ 로 정의한다. 이때, 분산 정도 = Variance(L)=

$$= \sum_{i=1}^n ||w_i' - m||^2, (w_i' \text{는 } w_i \text{가 선 } L \text{에 사영된 점})$$

$$= \sum_{i=1}^n | \langle v, u_i \rangle |^2, (v \text{는 선 } L \text{ 방향의 단위 벡터})$$

$$= \sum_{i=1}^n |v^T u_i|^2$$

$$= ||v^T U||^2 / n$$

$$= (v^T U U^T v) / n$$

$= \langle S v, v \rangle / n$ 으로 나타낼 수 있다. 3)이에 따라 $\langle S v_k, v_k \rangle = \lambda_k$ 이기에, 가장 작은 eigenvalue, 즉 분산 정도를 갖게 하는 Orthogonal한 eigenvector를 구하면 그것이 곧 주어진 데이터들의 법선 벡터가 됨을 확인할 수 있다.

4. Least Square Method

Least Square method(최소자승법)[2]이란 n 개의 모든 점으로부터의 특정 모델과의 거리가 최소가 되는 데이터를 대표할 모델을 찾는 알고리즘이다. 흔히 특정 데이터들의 추세선을 그리기 위해 사용되기도 하는데, 원하는 모델을 표현하는데 사용되는 값들을 변수로 잡아 이로부터 그 모델로부터의 각 데이터의 거리 차들의 제곱의 합이 최소가 되는 변수들을 구하면 최적의 모델을 구할 수 있는 것이다. 예를 들어, 삼차원 $Ax + By + C - z = 0$ 을 만족하면서 주어진 데이터를 가장 잘 나타내는 평면을 구한다고 할 경우, 이미 주어진 데이터인 각 데이터들의 x, y, z 좌표들의 선형결합들이 계수이며, A, B , 그리고 C 가 변수인 연립방정식을 세울 수 있을 것이다. 이를 각각 A, B, C 에 관해 미분하거나, A, B, C 벡터가 각 데이터들로부터 선형생성된 벡터 공간에 사영시킨 벡터가 되도록 하는 등의 수학적 방법을 통해 A, B, C 를 구해낼 수 있다.

5. RANSAC Algorithm

RANSAC[1]이란 Random Sample Consensus의 약자로 데이터를 바탕으로 특정 수식을 도출해내는 알고리즘의 일종이다. 흔히 사용하는 방법인 최소자승법(least square method)과의 차이점은 최소자승법은 모든 점으로부터의 거리가 최소가 되도록 모델을 찾지만, RANSAC은 가장 많은 수의 데이터들로부터 지지를 받는 모델을 선택하는 방법이다. 다른 식으로 설명하자면, 일정 거리 이내의 데이터는 포함하고, 그 외의 데이터는 무시함으로써 그 포함하는 데이터의 개수가 최대가 되는 모델을 선택하는 알고리즘이다. 즉, 만약 데이터에 본래 식으로부터 매우 많이 벗어난 노이즈가 다량 존재할 경우, 최소자승법으로 도출한 모델은 노이즈 쪽으로 모델이 치우쳐져 있을 가능성이 높다. 하지만 RANSAC은 가장 많은 수의 데이터로부터 지지를 받는 모델을 선택하므로 노이즈를 무시하고 본래의 식과 더 가까운 식을 도출할 수 있다.

III. Research Methods and Procedure

1. Programming platform

우리는 Kinect Camera를 이용하기 위해 총 5개의 플랫폼을 사용했다. 가장 먼저 Processing과 Java의 Libfreenect library를 사용하였다. 그러나 이 라이브러리는 개인이 사용하기 위해 개발한 것이라, 우리가 사용할 수 있는 function의 수가 적었다.

그래서 우리는 Kinect에서 받아 올 수 있는 데이터인 Point Cloud를 제어할 수 있는 라이브러리인 Point Cloud Library(이하 PCL)를 사용하기로 했다. PCL은 C++과 Python을 지원한다. 우리는 두 가지 언어 모두를 사용해 보았다. C++은 C계열 언어에서 최고의 성능을 보여주는 Visual Studio에 PCL을 import해 사용하였고, Python은 PCL과 함께 python에서 kinect를 제어할 수 있게 해주는 Kinect for Python Library를 함께 사용하였다. 하지만 PCL의 문제점은 키넥트를 구동시키는 동안에 3D 포인트들을 다루는 두 가지 작업을 동시에 하기에 어려움이 있다는 것이다.

새로운 방법을 찾던 도중 Kinect를 Unity 3D에서도 제어할 수 있다는 것을 알게 되어 사용을 시도하였었다. 그러나 Unity의 특성상 Point Cloud를 받아오는 작업보다 사람의 Skeleton Model을 받아오는 데에 더 특화되어 있었기 때문에 이 연구에는 적합하지 않다고 판단하였다.

최종적으로 이 연구에서 사용한 프로그래밍 플랫폼은 Microsoft에서 제공하는 Kinect SDK이다. Kinect SDK는 Visual Basic, C++, C#을 지원했는데, 우리는 Visual Studio와 함께 C#버전의 SDK를 사용하기로 하였다.

2. Evaluating Surface Normal Vectors for Each Point.

Microsoft Kinect에서 다루는 점들의 개수는 $320 \times 240 = 76800$ 개의 점들이다. 이 점들 중에서 가장 많은 점들을 포함하는 평면을 바로 구하게 한다면 필요 없는 점들을 포함하여 계산하는데 많은 시간이 낭비될 것이다. 그런 경우를 대비하여, 고려해야 하는 점들을 그 점들이 속한 평면이 기울어진 방향, 즉 법선 벡터를 가지고 분류해낼 수 있다. 우리는 책상 혹은 마룻바닥과 같이 지면에 평행한 평면을 인식하는 것이 목적이다. 이렇기에 각 점에서 구한 평면 법선 벡터들이 지평면에 수직한 점들만 골라서 계산할 데이터의 양을 줄일 수 있다.

법선 벡터를 구하는 가장 직관적인 방법은 근처 점들을 가지고 변위 벡터를 형성한 후, 이를 외적한 벡터를 잡는 방법이다. 이는 간단하게 한 점을 기준으로 그 근처 두 점을 잡아서 변위를 구한 두 벡터를 가지고 외적하여 구해내는 것도 가능하다. 이보다는 좀 앞서서 우리는 [그림 3]의 Algorithm 1을 보면 알 수 있듯이, 각 점의 좌우 두 점의 변위벡터와 상하 두 점의 변위 벡터를 외적하여 계산한 벡터를 저장하였다.

두 번째 방법은 각 점에 대해 근처 점들을 수집한 리스트를 대상으로 PCA 알고리즘을 적용하는 방법이다. PCA method은 앞서 설명했듯이, 주어진 n 개의 점들 사이에서 그 점들을 어느 축에 사영시켰을 시 퍼진 정도를 최소화하는 축을 구하는 알고리즘이다. 이 축이 곧 Scatter Matrix의 최소 Eigenvalue에 해당되는 Eigenvector이자 그 점들의 평면 법선 벡터가 된다. 우리는 각 점에 대해서 일정 거리에 있는 점들을 수집한 후, 이 모여진 점들에 대해서 PCA 알고리즘을 적용하면 각 점에서의 평면 법선 벡터를 구할 수 있다. [그림 4]

Algorithm 1: Assigning normal vectors for every point

```

1  $xlist \leftarrow$ 
   1 - dimensional list of  $x$  coordinates of each point.
2  $ylist \leftarrow$ 
   1 - dimensional list of  $y$  coordinates of each point.
3  $zlist \leftarrow$ 
   1 - dimensiona list of  $z$  coordinates of each point.
4  $Normalslist$ 
5 for every point except for the
   rightmost and bottom points. do
6    $rightpoint \leftarrow$  the right adjacent point
7    $downpoint \leftarrow$  the downward adjacent point
8    $uppoint \leftarrow$  the upward adjacent point
9    $leftpoint \leftarrow$  the left adjacent point
10   $Vector1$ 
11   $Vector2$ 
12   $Vector1.x \leftarrow (rightpoint.x - leftpoint.x)/2$ 
13   $Vector1.y \leftarrow (rightpoint.y - leftpoint.y)/2$ 
14   $Vector1.z \leftarrow (rightpoint.z - leftpoint.z)/2$ 
15   $Vector2.x \leftarrow (downpoint.x - uppoint.x)/2$ 
16   $Vector2.y \leftarrow (downpoint.y - uppoint.y)/2$ 
17   $Vector2.z \leftarrow (downpoint.z - uppoint.z)/2$ 
18   $Normalvector$ 
19   $Normalvector.x \leftarrow$ 
     $Vector2.y * Vector1.z - Vector1.y * Vector2.z$ 
20   $Normalvector.y \leftarrow$ 
     $Vector2.y * Vector1.z - Vector1.y * Vector2.z$ 
21   $Normalvector.z \leftarrow$ 
     $Vector2.y * Vector1.z - Vector1.y * Vector2.z$ 
22   $Normalslist \leftarrow Normalvector$ 
23 end

```

[그림 3]

Algorithm 2: Assigning normal vectors for every point
- PCA method

```

1 for every point  $p$  do
2   for every point  $q$  where  $q$ 's  $xy$  index
   is within 10 spaces of point  $p$ . do
3     if distance between  $q$  and  $p < threshold$ 
4       then
5          $Nearpoints[p] \leftarrow point\ q$ 
6       end
7   end
8 for every pointvector  $p$  do
9   New Origin  $O'_p \leftarrow$ 
     $(\sum Nearpoints[p]) / \text{number of points}$ 
10   $M_p \leftarrow$ 
     $\sum \text{points from } Nearpoints[p] \text{ relative to } O'_p$ 
11   $Y_p \leftarrow [m1, m2, \dots]$  where  $mi(\text{columnvector}) \in M$ 
12   $S_p(\text{scattermatrix}) \leftarrow Y_p Y_p^T$ 
13   $Normalvector[p] \leftarrow$ 
    Eigenvector corresp to  $\lambda_{min}$  relative to  $S_p$ 
14 end

```

[그림 4]

3. Plane detection.

Microsoft Kinect를 사용하여 가져온 점들은 노이즈가 크다. 따라서 실물에서는 모두 같은 책상에 속한 점일지언정 Kinect으로 인식한 좌표가 그 평면으로부터 이탈되어 있을 가능성이 높다. 따라서 이런 오차에 기여하는 Outlier 점들을 배제하는 알고리즘을 구성할 필요가 있다. Outlier들이 많은 데이터를 다룰 시에는 앞서 설명한 RANSAC알고리즘을 적용하는 것이 유용하다. [그림 5]의 <Algorithm 3>과 [그림 6]의 <Algorithm 4>에서는 각각 점 3개, 점 1개와 평면 벡터를 가지고 만든 평면이 일정 거리 threshold 이내에 포함하는 점들의 개수가 최대인 평면을 구하는 알고리즘이다. 이때, 일반적인 경우와 달리 <Algorithm 4>의 경우, 앞서 이미 각 점에 대한 법선 벡터를 구해놓았기에 계산 절차를 절약하는 데 <Algorithm 3>과 비교했을 시 효율적일 것이다.

우리가 시도한 또 하나의 방법은, Least Square 알고리즘에서 각 항에 가중치(Weight)를

지정하여 곱하는 것이다. [그림 7]의 <Algorithm 5>을 보면 알 수 있듯이 각 점마다 추세평면으로부터의 거리에 반비례하는 가중치(Weight) 함수를 거리 차에 곱한 값들의 최소값을 구하기 위해 미분하여 방정식을 풀면, 주어진 d개의 점들에서 존재하는 Outlier를 배제한 가장 넓은 평면을 구할 수 있다.

Algorithm 3: Plane detection using 3 points

```

1 truncatedpoints
2 plane
3 Max ← 0
4 Maxplane ← list for storing plane points
5 for every point (A) ∈ truncated points do
6   for every point (B) ∈ truncated points
7     do
8       for every point (C) ∈ truncated
9         points do
10          Vector1 ← pointA - pointB
11          Vector2 ← pointA - pointC
12          Normalvector ← Vector1XVector2
13          Count ← 0
14          for every point (N) from all
15            points do
16              FactorX ← (pointA.x - pointN.x) *
17                Normalvector.x
18              FactorY ← (pointA.y - pointN.y) *
19                Normalvector.y
20              FactorZ ← (pointA.z - pointN.z) *
21                Normalvector.z
22              Denominator ← |Normalvector|
23              distance ← (FactorX + FactorY +
24                FactorZ)/Denominator
25              if distance < threshold then
26                plane ← pointN
27                count ++
28              end
29            end
30          if count > max then
31            count = max
32            maxplane = plane
33          end
34        end
35      end
36    end
37  end
38 end

```

[그림 5]

Algorithm 4: Plane detection using 1 point and surface normals

```

1 truncatedpoints
2 plane
3 Max ← 0
4 Maxplane ← list for storing plane points
5 for every point (A) ∈ truncated points do
6   for every point (N) ∈ all points do
7     Normalvector ←
8       surface normal vector from pointA
9     FactorX ←
10      (pointA.x - pointN.x) * Normalvector.x
11     FactorY ←
12      (pointA.y - pointN.y) * Normalvector.y
13     FactorZ ←
14      (pointA.z - pointN.z) * Normalvector.z
15     Denominator ← |Normalvector|
16     distance ← (FactorX + FactorY +
17       FactorZ)/Denominator
18     if distance < threshold then
19       plane ← pointN
20       count ++
21     end
22   end
23   if count > max then
24     count = max
25     maxplane = plane
26   end
27 end
28 end

```

[그림 6]

Algorithm 5: Plane detection using least squares

```

1 truncatedpoints
2  $W = [w_1, w_2, \dots]$  where  $w_i = 1$ 
3 while  $\sum w_i < \text{threshold}$  do
4   for for every point  $i \in \text{truncated}$ 
     points do
5     3X3 Matrix  $M$ 
6     3X1 Matrix  $N$ 
7      $m_{11} \leftarrow \sum x_i * x_i * w_i$ 
8      $m_{12} \leftarrow \sum x_i * y_i * w_i$ 
9      $m_{13} \leftarrow \sum x_i * z_i * w_i$ 
10     $m_{21} \leftarrow \sum x_i * y_i * w_i$ 
11     $m_{22} \leftarrow \sum y_i * y_i * w_i$ 
12     $m_{23} \leftarrow \sum y_i * z_i * w_i$ 
13     $m_{31} \leftarrow \sum x_i * z_i * w_i$ 
14     $m_{32} \leftarrow \sum y_i * z_i * w_i$ 
15     $m_{33} \leftarrow \sum z_i * z_i * w_i$ 
16     $n_{11} \leftarrow \sum x_i * w_i$ 
17     $n_{21} \leftarrow \sum y_i * w_i$ 
18     $n_{31} \leftarrow \sum z_i * w_i$ 
19   end
20    $[a, b, c] = M^{-1}N$ 
21   for every point  $p$  do
22      $w_i \leftarrow e^{-(a*x_i + b*y_i + c - y_i)^2}$ 
23   end
24 end
25  $\text{plane} \leftarrow ax + bz + c - y = 0$ 

```

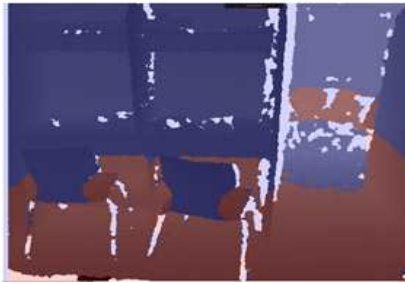
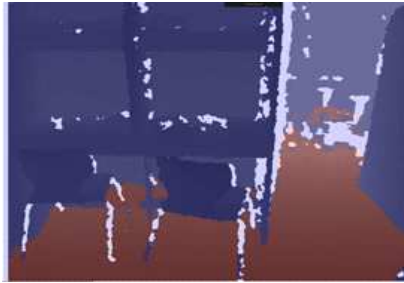
[그림 7]

IV. Results

1. Surface Normal Vector Evaluation



1) Plane detection without truncation

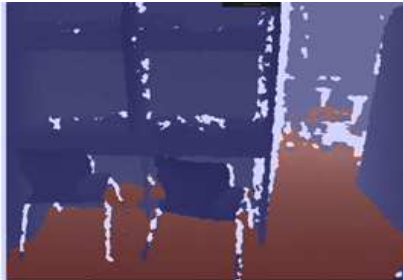
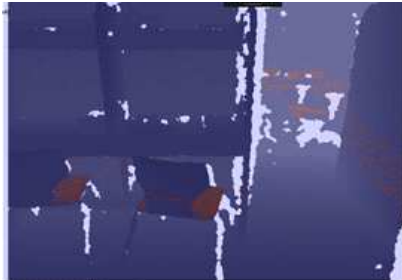
[표 1]

Algorithm	RANSAC Plane Detection w/o Surface Normals	RANSAC Plane Detection using Surface Normals
Red- > Plane Shade is darker as the point is closer to the camera.		
Algorithm Time	$O(n^2)$	$O(d^2+kn)$, where d is # of truncated points.
Elapsed Time _(Avg)	2:10s	0:31s

2) Plane detection after truncation using different surface normal vectors.

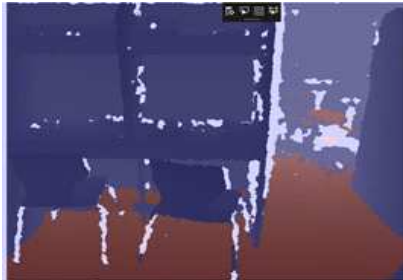
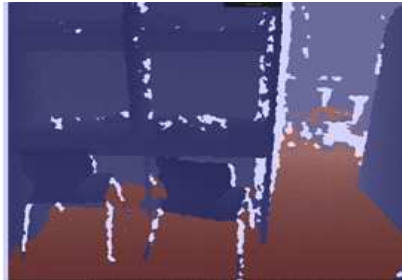
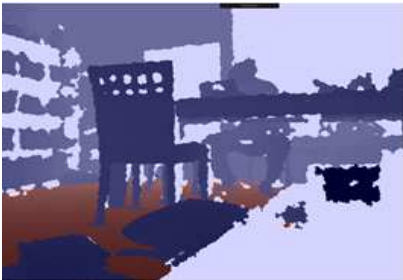
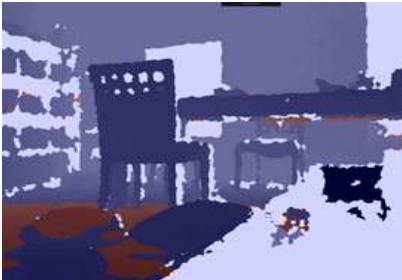
[표 2]

Algorithm	PCA Surface Normals (Algorithm 2)	Cross product Surface Normals (Algorithm 1)
Red- > Plane Shade is darker as the point is closer to the camera.		
# of Truncated points	14779	13344

Algorithm	PCA Surface Normals 〈Algorithm 2〉	Cross product Surface Normals 〈Algorithm 1〉
Algorithm	RANSAC Plane Detection using PCA Surface Normals 〈Algorithm 2〉	RANSAC Plane Detection using Cross product Surface Normals 〈Algorithm 1〉
Red- > Plane Shade is darker as the point is closer to the camera,		
Elapsed Time _(Avg)	0:31s	0:01s
# of Truncated points	22000	1750

2. Plane Detection

[표 3]

Algorithm	RANSAC Plane Detection	Plane Detection using Least Squares
Red- > Plane Shade is darker as the point is closer to the camera,		
Elapsed Time _(Avg)	0:31s	0:13s
Red- > Plane Shade is darker as the point is closer to the camera,		
Algorithm Time	$O(k_1n+d_1^2)$	$O(k_2n+k_3d_2)$
Elapsed Time _(Avg)	0:25s	0: 3 0s

V. Discussion

1. Evaluating Surface Normal Vectors for Each Point.

1) Plane detection without truncation

먼저 주어진 환경에 대해 표면 법선 벡터를 구하지 않은 경우와 구한 경우의 RANSAC 평면 인식 알고리즘의 출력 결과를 비교해보았다. 여러 번 재차 시도하여 평균을 해보았을 때 표면 법선 벡터를 구한 후 필요 없는 점들을 제거한 경우가 그러하지 않은 경우에 비해 월등하게 더 시간을 절약할 수 있음을 확인 할 수 있었다. 그뿐만 아니라, 표면 법선 벡터가 지표면에 대하여 수직인 점들만 고려하지 않았을 경우, 최종적인 평면으로 내놓은 결과에도 실제 구하고자 한 평면에 속하지는 않지만 그 평면과 같은 높이의 벽, 혹은 책상 다리 등에 위치한 점들이 포함되는 것을 관측 할 수 있었다. 이는 이들의 표면 법선 벡터의 각도가 수직한 평면에 속하지는 않음에도 불구하고 계산하는 데이터들에 포함되어, 최종 결정된 평면에 실제 관계가 없음에도 불구하고 포함되어 출력 결과에 나오게 된 것이다. 고로 이 결과를 통해 우리는 표면 법선 벡터를 계산 후 평면을 인식하는 것이 전체 과정을 빠르게 할 뿐더러 더 정밀한 측정에 기여하는 것을 확인할 수 있다.

2) Plane detection after truncation using different surface normal vectors.

이 다음, 가장 효율적인 법선 벡터 계산 알고리즘을 확인하기 위해, 같은 RANSAC 평면 인식 방법에 대해 <Algorithm 1>과 <Algorithm2>를 각각 적용하여 표면 법선 벡터를 구한 출력 결과를 비교해보았다.

만약 Microsoft Kinect가 가져오는 점들의 정보가 실제 점들과 완전히 일치한다면, <Algorithm 1>의 방법이 더 시간이 단축될 것이다. 그뿐만 아니라, 각 점의 표면 법선 벡터를 계산할 시 적은 수들의 근방 점들에 의존하기에 벽면 근처의 점들의 계산이 <Algorithm 2>에 비해 정밀할 것이다. 그러나, Microsoft Kinect가 주는 점들의 정보는 오차가 크기에 <Algorithm 1>에서 바로 옆 점들과의 변위 벡터를 외적 시 바로 근방의 좌표를 다루기에 미세한 오차 하나하나가 법선 벡터의 각도에 큰 변형을 일으킨다. 따라서 각 점에 대해 전혀 표면 법선 벡터가 지표면에 대해 수직이 아닌 벡터들의 점, 예를 들어 벽면의 점들과 같이 이런 점들이 처리하는 데이터에 포함되는 것을 관측할 수 있었다. [표 2] 여기서 처리속도가 아주 빠른 이유는 표면 법선 벡터의 계산이 정확하지 않았기에 실제로 걸려진 점들의 개수가 1600-1900 개 가량으로 굉장히 작은 데이터를 가지고 계산을 하기 때문이다.

반면, <Algorithm 2>에서는 시간이 다소 더 걸리는 편이지만, 같은 현장을 두고 법선 벡터

가 수직인 점들을 뽑아냈을 시, 그 수가 <Algorithm 1>에 비해 월등히 많을 뿐더러, 노이가 존재하는 데이터에서 비교적 정확하게 법선 벡터 각도를 계산할 수 있음을 확인할 수 있다. 단, 한 가지 확인할 수 있는 단점은 한 평면에 대해 다른 평면과 접하는 점선 구역 근처의 점들은 각 점에 대해 근처의 점들을 수집하는 단계에서 자신이 속한 평면뿐만 아니라 인접한 다른 방향의 평면들의 점들 또한 수집함으로써 부정확한 데이터가 계산되는 것을 확인하였다. 이는 근접한 거리의 점들을 수집할 때 포함하는 점으로부터의 거리의 최대 threshold을 줄임으로써 오차를 최소화 할 수 있었다.

2. Plane detection.

1) Outlier neglecting plane detection.

표면 법선 벡터를 통해 처리할 데이터의 양을 줄인 후, 우리는 평면을 구하는 알고리즘을 세가지 제시하였다. 그 중 <Algorithm 4>와 <Algorithm 5>의 출력 결과를 비교 및 분석해보았다. 위 [표 4]를 보면 알 수 있듯이 일반적인 평면을 구하는 경우에 대해서는 <Algorithm 4>, 즉 RANSAC 알고리즘을 통해 평면을 구하는 경우가 <Algorithm 5>, 즉 Weighted Least Squares 알고리즘보다 덜 효율적임을 확인할 수 있다. 출력 결과 및 바닥의 결과는 거의 비슷하나 Weighted Least Squares 알고리즘이 시간이 거의 반감되었음을 관찰 할 수 있었다.

하지만, [표 4]의 첫 번째 줄 그림들과 달리 [표4]의 두 번째 줄 그림들에서 볼 수 있듯이 같이, 데이터가 한 평면에 집중되어 있지 않고, 여러 개의 넓은 평면이 공존할 시, 혹은 평면 위에 여러 개의 납작하고 평평한 물체가 있을 시에는 다른 결과를 관찰 할 수 있었다. 만약 평면 위에 여러 개의 납작한 물체가 놓여 있을 경우 데이터가 한 곳에만 집중되지 않고, 여러 곳에 걸쳐 집중되어 걸쳐 있기에 위와 같이 Weighted Least Squares 알고리즘은 정확한 결과를 출력하지 못한다. 그러나, 최대 데이터 지지 개수를 중심으로 모델을 선택하는 RANSAC 알고리즘은 평평한 물체 위의 점들과 상관없이, 원하는 평면을 잘 찾아냄을 확인 할 수 있었다. 단, 만약 평평한 물체들이 차지하는 영역이 바닥보다 클 경우, RANSAC 알고리즘 또한 정확한 결과를 반환하지 못한다.

2) Threshold calculation & Limitations.

위 결과를 통해 정리할 수 있듯이, 현재 Microsoft Kinect를 사용하여 이러한 다양한 알고리즘을 적용했을 때 우리가 발견한 가장 큰 문제점은 어느 환경이냐에 따라 위에서 다뤘듯이 가장 효과적인 알고리즘이 다를 뿐만 아니라, 알고리즘에서 사용하는 Threshold의 가장 효율적인 값이 매번 다르다는 점이다. Weighted Least Squares 알고리즘을 특수한 납작하고 평평

한 물체가 놓인 바닥에 적용하는 예시를 고려해보자. 알고리즘을 반복하는 횟수 Threshold이 클 때는 여러 번 반복하기에 처리 속도가 느리고, 계산된 모델로부터 조금만 벗어난 점들로 outlier로 취급이 되어, 이들을 포함하기 위해 거리 threshold을 늘리지 않는 이상, 많은 필요한 데이터를 버리게 된다.[그림 7]. 반면 threshold가 작을 때에는 Outlier을 충분히 배제하지 못하여, 계산한 모델이 원하는 데이터로부터 거리가 멀 수도 있다.[그림 6]. RANSAC알고리즘 또한 원하는 평면 외의 점들은 포함하지 않으면서, 원하는 평면 위의 점들의 개수가 다른 평면 위 점의 개수보다 많도록 평면에 포함되는 거리 차의 threshold을 상황에 따라 다르게 맞춰주어야 한다.

뿐만 아니라 표면 법선 벡터를 계산하는 과정에서도 threshold 값이 결과의 효율과 정확성을 좌우한다. 위에서 언급했듯이, 근방의 있는 점들을 수집할 때, 근처 점들을 너무 많이 수집하면, 처리 속도가 저하될 뿐더러, 원하는 데이터만을 얻기가 쉽지가 않다. 하지만 너무 적은 점들을 수집할 경우, 계산한 법선 벡터의 방향의 정확성이 떨어지게 된다.

이처럼 이 연구에서 결과로 제시한 알고리즘은 주어진 환경에 따라 가장 최적의 알고리즘 및 입력 상수값이 달라지고, 이를 사람이 수동적으로 조절해줘야 한다는 한계점을 발견할 수 있었다.

VI. Conclusion

우리는 유니티, 비주얼 스튜디오 wing ide, 등의 다양한 개발환경 및 Oencv, Point Cloud Library, Kinect SDK 등의 다양한 라이브러리 및 python, C++, C#, java 등의 다양한 언어로 평면인식 알고리즘을 구현하는 것을 시도했다. 하지만 많은 경우, 생각보다 키넥트를 컴퓨터에 연결해서 정보를 입출력 하는 과정은 쉽지 않았고, 많은 경우 제대로 코드를 짜 보기도 전에 에러가 나거나 라이브러리를 불러오는 과정에서 오류가 생겼다. 하지만 다양한 시도를 한 결과 Kinect SDK 라이브러리를 불러와서 Windows Visual Studio에서 C#으로 코딩을 하게 되서야 알고리즘 개발을 진행할 수 있었다.

개발환경을 구축한 후에도 RANSAC, PCA, Least Square 등의 다양한 알고리즘을 시도해보았다. 각자 방법의 장단점이 있었지만, 우리에게 필요했던 것은 저렴한 장비를 가지고도 빠른 시간 내에 평면을 인식해내는 알고리즘을 개발하는 것이었으므로 그러한 방향으로 알고리즘 개발을 계속했다. 그 결과, PCA 알고리즘과 Least Square 알고리즘을 사용해서 약 10초 내외에 비교적 정확한 평면을 인식하는 알고리즘 구현에 성공했다.

비록 결과적으로 대단한 무언가를 만들어내지는 못했지만, 외부 프로그램의 도움 없이, 조

원 스스로의 힘으로 다양한 방법들 및 알고리즘을 찾아보고, 참고해가면서 여러 알고리즘 구현에 성공하여, 각각의 한계점을 직접 확인하고 개선할 부분을 재확인했다는 점에 이 연구의 의의가 있다고 생각한다.

VII. References

- [1] Marcel Junemann, Object Detection and Recognition with Microsoft Kinect, Berlin, p. 4, 2012
- [2] Elementary Linear Algebra: Applications Version, 11th Edition. (2014, February 1).
- [3] Greif, C., Ju, T., Mitra, N. J., Shamir, A., Hornung, O. S., & Zhang, H. R. (2015). Preview this Book A Sampler of Useful Computational Tools for Applied Geometry, Computer Graphics, and Image Processing (D. Cohen-Or, Ed.). A K peters.

ABSTRACT

The Development of a plane-detecting algorithm using the Microsoft Kinect

Researcher : Yohan Yun(sophomore, yohanme@naver.net)

Jonghyeon Lee(sophomore, jonghyun777@gmail.com)

Jinhyeong Lee(sophomore, wlsquddl2001@naver.com)

Supervisor : Hyeongseok Kim(Dong Eui University, hckim@deu.ac.kr)

Co-Supervisor : Yongju Kim(Korea Science Academy of KAIST, kyhgifted@kaist.ac.kr)

Abstract

〈Write under 500Words〉

Among many of the various parts of IOT, the basis of object tracking is separating the object from the background, or in other words, detecting the background apart. In most cases, the background happens to be a wall, a floor or any other type of plane. As such, the detection of planes through 3d-cameras are elementary in the sense of object detection and tracking. In this paper, we opt to develop a plane detecting algorithm using a relatively very cheap device known as the Microsoft Kinect to provide many others to utilize IOT without much financial limitations. We are going to create and implement plane detecting algorithms through various platforms and methods onto different real-life environments and see which yields the best result.