



Universidad Católica  
**San Pablo**



**CONCYTEC**

# Unique Signatures of Histograms for Local Surface Description

---

ESTUDIANTE DE MAESTRÍA  
ALEJANDRA CALLO - JOSE JAITA

# ¿Qué es un descriptor?

Un descriptor extrae información geométrica de un objeto 3D, es decir una nube de puntos

# Shot

---

Es un algoritmo híbrido que combina los histogramas con las firmas para la mejor extracción de datos, accediendo al color y su forma tridimensional,

Recolecta información de varios histogramas alrededor imitando el comportamiento de los métodos por firma. Para la firma se usa una malla esférica Almacenando los valores radiales y la elevaciones en los ejes.

# Compiler

Requerimientos:

- VTK
- CMAKE

Modifica

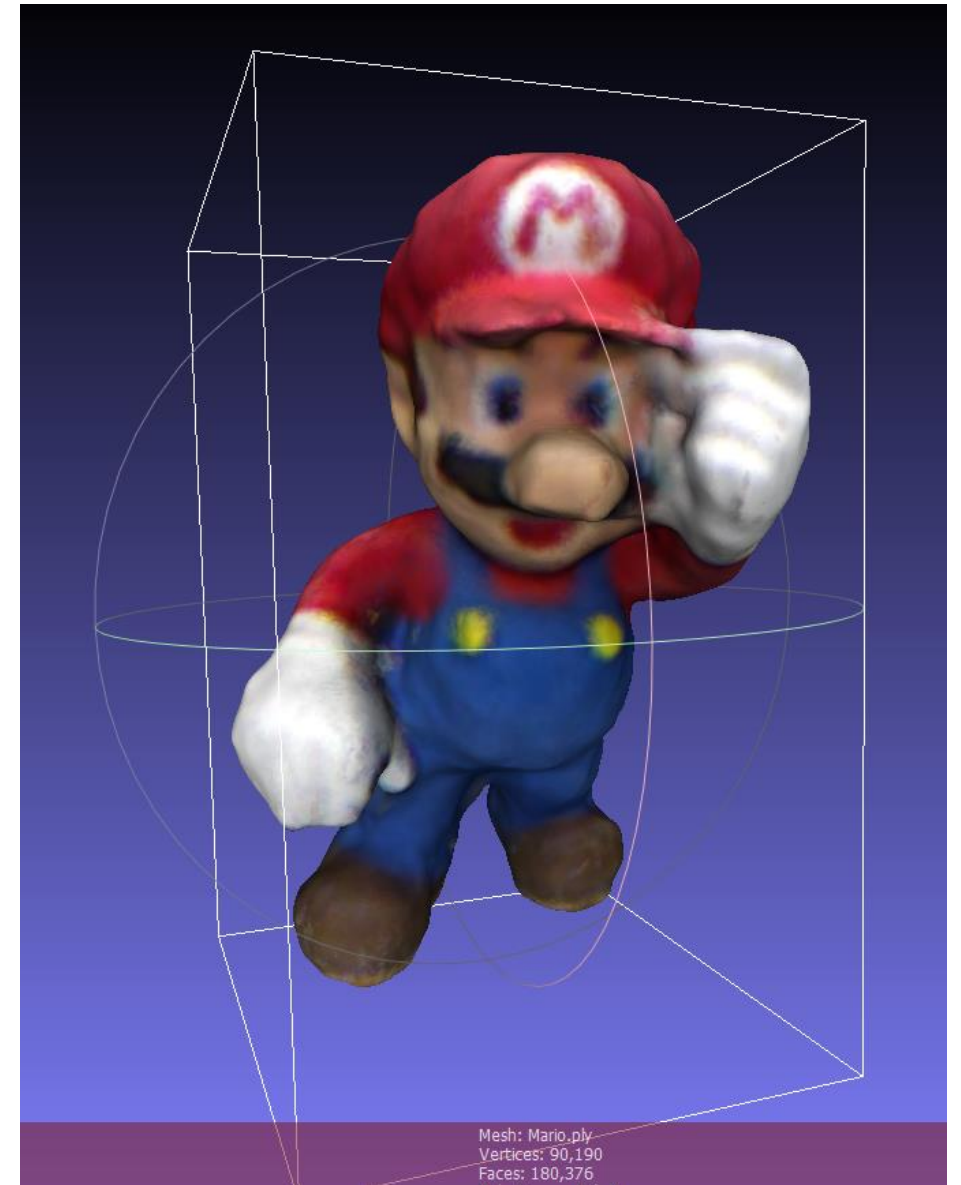
```
vtkPolyData * pd = vtkPolyData :: New ( ) ;  
someAlgorithm - > SetInputConnection ( pd - > GetProducerPort ( ) ) ;
```

Esto puede ser reemplazado por:

```
vtkPolyData * pd = vtkPolyData :: New ( ) ;  
someAlgorithm - > SetInputData ( pd ) ;
```

# Entrada

La entrada es un archivo en formato .PLY que sirve para almacenar objetos gráficos que se describen como una colección de polígonos. El formato de archivo tiene dos subformatos: una representación ASCII y una versión binaria.



# Función de covarianza

Según los trabajos de Hoppe y Mitra se presenta una matriz de covarianza basada en la descomposición de los autovalores, de los k-vecinos mas cercanos al punto  $P_i$

$$M = \frac{1}{k} \sum_{i=0}^k (P_i - \hat{P})(P_i - \hat{P})^T, \quad \hat{P} = \frac{1}{k} \sum_{i=1}^k P_i .$$

Se modifica para que a los puntos mas lejanos tengan pesos pequeños, a fin de aumentar la repetibilidad para mejorar la solidez al ruido manteniendo el Radio (R) que se utilizan para calcular el descriptor. Reemplazando el centro con el punto de característica P

$$M = \frac{1}{\sum_{i: d_i \leq R} (R - d_i)} \sum_{i: d_i \leq R} (R - d_i) (P_i - P)(P_i - P)^T$$

```

void getSHOTLocalRF(vtkPolyData *cloud, vtkIdList *NNpoints, double radius, int index, float *rfc)
{
    double originDouble[3];
    cloud->GetPoint(index, originDouble);

    int nNeighbours = NNpoints->GetNumberOfIds();

    //double V_Vt[9];
    double* currPoint;
    double* vij = new double[nNeighbours * 3];
    double *covM[3];

    // Initialize covariance matrix
    covM[0] = new double[3];
    covM[1] = new double[3];
    covM[2] = new double[3];
    memset(covM[0], 0.0, sizeof(double)*3);
    memset(covM[1], 0.0, sizeof(double)*3);
    memset(covM[2], 0.0, sizeof(double)*3);

    memset(covM[0], 0, sizeof(double)*3);
    memset(covM[1], 0, sizeof(double)*3);
    memset(covM[2], 0, sizeof(double)*3);

    double distance = 0.0;
    double sum = 0.0;

    int validNNpoints = 0;

```

# Inicializa la matriz

```

for(int ne = 0; ne < nNeighbours; ne++)
{

    if(NNpoints->GetId(ne) != index) { // perchè il KdTree restituisce anche il punto origine

        currPoint = cloud->GetPoint(NNpoints->GetId(ne));

        // Difference between current point and origin
        vij[validNNpoints*3 + 0] = currPoint[0] - originDouble[0];
        vij[validNNpoints*3 + 1] = currPoint[1] - originDouble[1];
        vij[validNNpoints*3 + 2] = currPoint[2] - originDouble[2];

        distance = radius - sqrt(vij[validNNpoints*3 + 0]*vij[validNNpoints*3 + 0]
        + vij[validNNpoints*3 + 1]*vij[validNNpoints*3 + 1] + vij[validNNpoints*3 + 2]*vij[validNNpoints*3 + 2]);

        // Multiply vij * vij'
        covM[0][0] += distance * vij[validNNpoints*3] * vij[validNNpoints*3];
        covM[1][1] += distance * vij[validNNpoints*3+1] * vij[validNNpoints*3+1];
        covM[2][2] += distance * vij[validNNpoints*3+2] * vij[validNNpoints*3+2];

        double temp = distance * vij[validNNpoints*3] * vij[validNNpoints*3+1];
        covM[0][1] += temp;
        covM[1][0] += temp;
        temp = distance * vij[validNNpoints*3] * vij[validNNpoints*3+2];
        covM[0][2] += temp;
        covM[2][0] += temp;
        temp = distance * vij[validNNpoints*3+1] * vij[validNNpoints*3+2];
        covM[1][2] += temp;
        covM[2][1] += temp;

        sum += distance;
        validNNpoints++;

    }
}

```

Diferencia entre el punto actual y origen

Distancia = Radio – Distancia Euclidea

Multiplica Vij \* Vij'



```

covM[0][0] /= sum; covM[0][1] /= sum;
covM[0][2] /= sum; covM[1][0] /= sum;
covM[1][1] /= sum; covM[1][2] /= sum;

covM[2][0] /= sum; covM[2][1] /= sum;
covM[2][2] /= sum;

```

```

double eval[3];
double *evect[3];
evect[0] = new double[3];
evect[1] = new double[3];
evect[2] = new double[3];

```

```

// Diagonalization (eval = eigenvalues, evect = eigenvector)
// - Eigenvalues and eigenvectors are sorted in decreasing order
// - Eigenvectors are already normalized
int resJ = vtkMath::Jacobi(covM, eval, evect);

```

```

int plusNormal = 0, plusTangentDirection1=0;

```

```

for(int ne = 0; ne < validNNpoints; ne++)
{
    double dotProduct = vij[ne*3]*evect[0][0] + vij[ne*3 + 1]*evect[1][0] + vij[ne*3 + 2]*evect[2][0];
    if (dotProduct >= 0)
    {
        plusTangentDirection1++;
    }
    dotProduct = vij[ne*3]*evect[0][2] + vij[ne*3 + 1]*evect[1][2] + vij[ne*3 + 2]*evect[2][2];
    if (dotProduct >= 0)
    {
        plusNormal++;
    }
}

```

Se halla Autovalores y Autovectores

Ordena y Normaliza

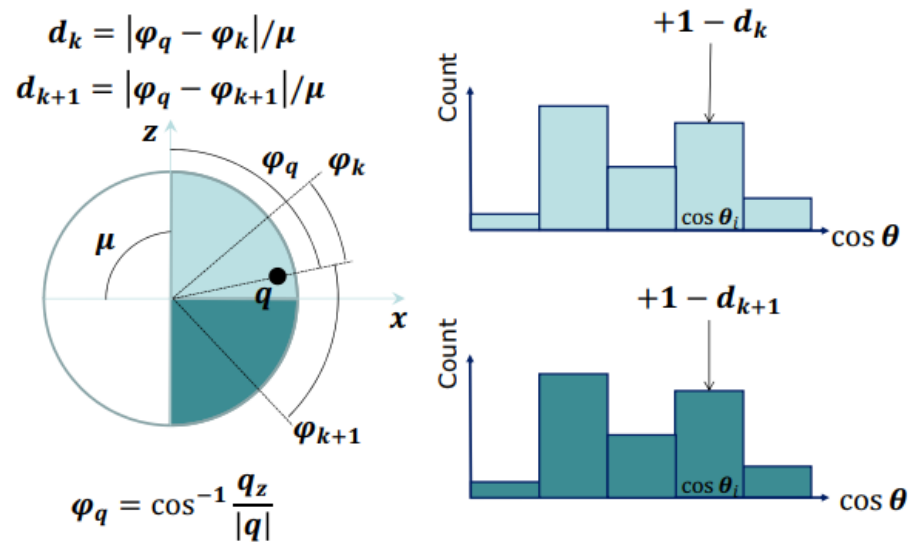
Se halla Tangentes

Producto Punto

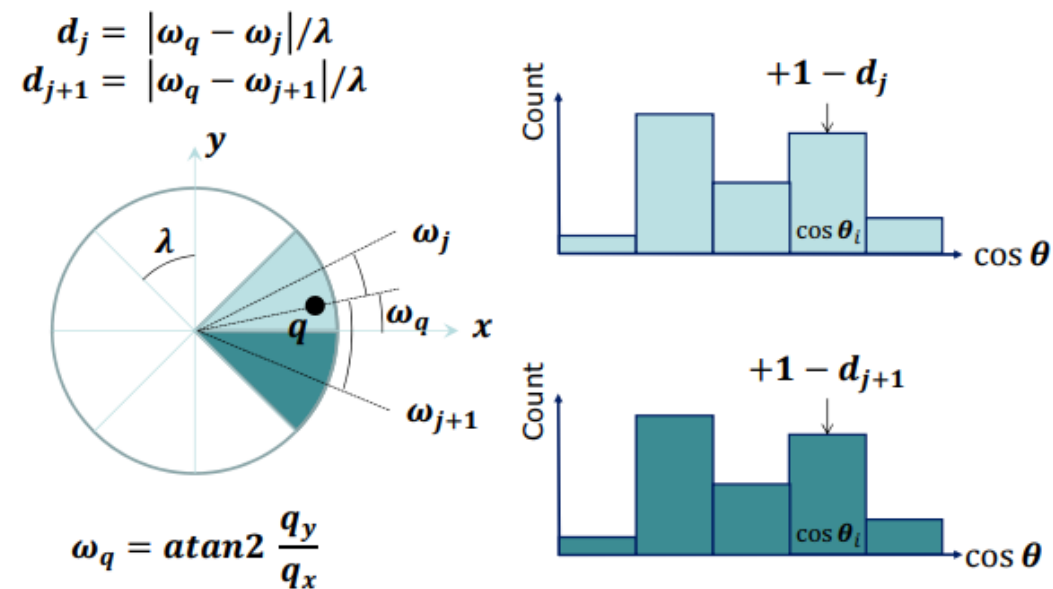
# Si (color)

## RGB->CIELAB

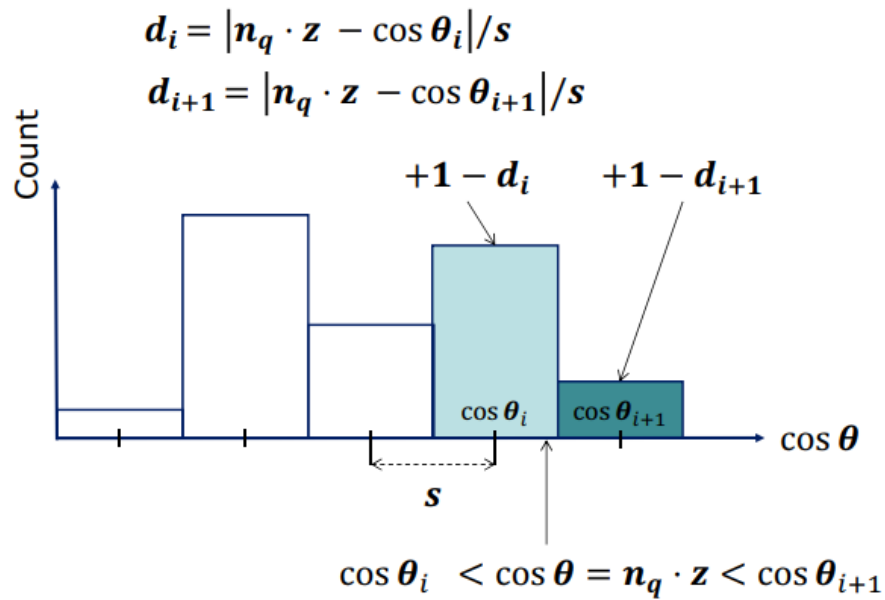
### Interpolación de Canales



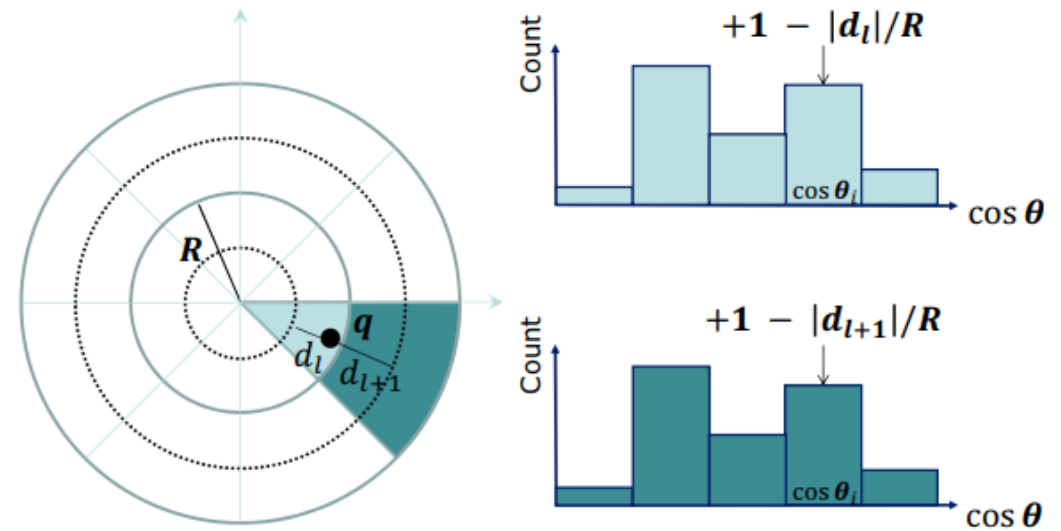
(c) Interpolation on elevation



(b) Interpolation on azimuth



(a) Interpolation on normal cosines



(d) Interpolation on distance

La Función cuadrilineal logra robustez a las variaciones de densidad, normaliza el conjunto descriptor para tener la norma euclidiana igual a 1.

```
void SHOTDescriptor::interpolateDoubleChannel
```

Abre un archivo . ply



Extrae su descriptor



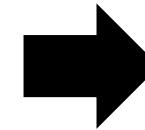
Modifica la malla de  
puntos(rotacion y  
ruido)



Extrae un nuevo  
descriptor



Comparación



Invarianza SHOT

$$L = 116f_y - 16$$

$$a = 500(f_x - f_y)$$

$$b = 200(f_y - f_z)$$

where

$$f_x = \begin{cases} \sqrt[3]{x_r} & \text{if } x_r > \epsilon \\ \frac{\kappa x_r + 16}{116} & \text{otherwise} \end{cases}$$

$$f_y = \begin{cases} \sqrt[3]{y_r} & \text{if } y_r > \epsilon \\ \frac{\kappa y_r + 16}{116} & \text{otherwise} \end{cases}$$

$$f_z = \begin{cases} \sqrt[3]{z_r} & \text{if } z_r > \epsilon \\ \frac{\kappa z_r + 16}{116} & \text{otherwise} \end{cases}$$

$$x_r = \frac{X}{X_r}$$

$$y_r = \frac{Y}{Y_r}$$

$$z_r = \frac{Z}{Z_r}$$

$$\epsilon = \begin{cases} 0.008856 & \text{Actual CIE standard} \\ 216/24389 & \text{Intent of the CIE standard} \end{cases}$$

$$\kappa = \begin{cases} 903.3 & \text{Actual CIE standard} \\ 24389/27 & \text{Intent of the CIE standard} \end{cases}$$

```
void SHOTDescriptor::RGB2CIELAB(unsigned char R, unsigned char G, unsigned char B, float &L, float &A, float &B)
```

```
    if (sRGB_LUT[0] < 0 )
    {
        for (int i = 0; i < 256; i++)
        {
            float f = i / 255.0f;
            if (f > 0.04045)
                sRGB_LUT[i] = (float) pow((f + 0.055) / 1.055, 2.4);
            else
                sRGB_LUT[i] = f / 12.92;
        }
    }
```

```
    for (int i = 0; i < 4000; i++)
    {
        float f = i / 4000.0f;
        if (f > 0.008856)
            sXYZ_LUT[i] = pow(f, (float)0.3333);
        else
            sXYZ_LUT[i] = (7.787 * f) + (16.0 / 116.0);
    }
```

```
    float fr = sRGB_LUT[R];
    float fg = sRGB_LUT[G];
    float fb = sRGB_LUT[B];
```

```
// Use white = D65
```

```
const float x = fr * 0.412453 + fg * 0.35758
const float y = fr * 0.212671 + fg * 0.71516
const float z = fr * 0.019334 + fg * 0.11919
```

```
float vx = x / 0.95047;
float vy = y;
float vz = z / 1.08883;
```

```
//printf("vx:%f vy:%f vz:%f\n", vx, vy, vz);
```

```
vx = sXYZ_LUT[int(vx*4000)];
vy = sXYZ_LUT[int(vy*4000)];
vz = sXYZ_LUT[int(vz*4000)];
```

```
L = 116.0 * vy - 16.0;
if(L>100)
    L=100;
```

```
A = 500.0 * (vx - vy);
if(A>120)
    A=120;
else if (A<-120)
    A=-120;
```

```
B2 = 200.0 * (vy - vz);
if(B2>120)
    B2=120;
else if (B2<-120)
    B2=-120;
```

```

void SHOTDescriptor::interpolateSingleChannel(vtkPolyData* cloud, vtkIdList* NNpoints, const std::vector<double> ,
{

    for (int i_idx = 0; i_idx < NNpoints->GetNumberOfIds(); ++i_idx)
    {
        double point[3];
        cloud->GetPoint(NNpoints->GetId(i_idx), point);

        double distance = sqrt(distances[i_idx]); //sqrt(distance_sqr);

        float delta[3] = { point[0] - centralPoint[0], point[1] - centralPoint[1], point[2] - centralPoint[2]};

        // Compute the Euclidean norm
        if (areEquals(distance, 0.0))
            continue;

        double xInFeatRef = (delta[0] * rf[0] + delta[1] * rf[1] + delta[2] * rf[2]);
        double yInFeatRef = (delta[0] * rf[3] + delta[1] * rf[4] + delta[2] * rf[5]);
        double zInFeatRef = (delta[0] * rf[6] + delta[1] * rf[7] + delta[2] * rf[8]);

        -
        -
        -

        //Interpolation on the cosine (adjacent bins in the histogram)
        binDistance[i_idx] -= step_index;
        double intWeight = (1- std::abs(binDistance[i_idx]));

        //19.9.17: (nr_bins + 1) used instead of nr_bins
        if( binDistance[i_idx] > 0)
            shot[ volume_index + ((step_index+1) % (nr_bins + 1) )] += binDistance[i_idx];
        else
            shot[ volume_index + ((step_index + nr_bins ) % (nr_bins + 1) )] += -binDistance[i_idx];
    }
}

```

Se accede a los puntos

Se calcula la distancia entre ellos

Calcula la norma Euclidean

Para cada histograma local  
Se coloca cada punto en un bin

// contenedores no están espaciados  
para analizar la interpolación con  
sus vecinos

```
//Interpolation on the azimuth (adjacent horizontal volumes)
```

```
double azimuth = atan2( yInFeatRef, xInFeatRef );
```

```
int sel = desc_index >> 2;
```

```
double angularSectorSpan = DEG_45_TO_RAD;
```

```
double angularSectorStart = -DEG_168_TO_RAD;
```

```
double azimuthDistance = (azimuth - (angularSectorStart + angularSectorSpan*sel)) / angularSectorSpan;
```

```
azimuthDistance = std::max(-0.5, std::min(azimuthDistance, 0.5));
```

```
assert((azimuthDistance < 0.5 || areEquals(azimuthDistance, 0.5)) && (azimuthDistance > -0.5 || areEquals(azimuthDistance, -0.5)));
```

```
if(azimuthDistance > 0){
```

```
    intWeight += 1 - azimuthDistance;
```

```
    int interp_index = (desc_index + 4) % m_maxAngularSectors;
```

```
    assert( interp_index * (nr_bins+1) + step_index >= 0 && interp_index * (nr_bins+1) + step_index < m_descLength);
```

```
    shot[ interp_index * (nr_bins+1) + step_index] += azimuthDistance;
```

```
}
```

```
else{
```

```
    int interp_index = (desc_index - 4 + m_maxAngularSectors) % m_maxAngularSectors;
```

```
    assert( interp_index * (nr_bins+1) + step_index >= 0 && interp_index * (nr_bins+1) + step_index < m_descLength);
```

```
    intWeight += 1 + azimuthDistance;
```

```
    shot[ interp_index * (nr_bins+1) + step_index] -= azimuthDistance;
```

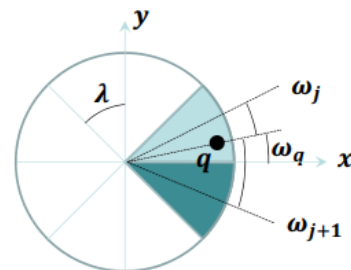
```
}
```

Se calcula la distancia angular hacia el centro

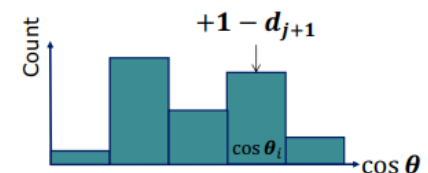
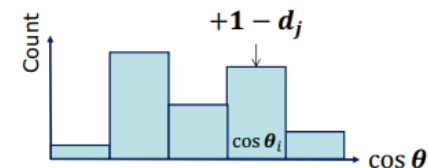
Se mide la distancia entre los bins

$$d_j = |\omega_q - \omega_j|/\lambda$$

$$d_{j+1} = |\omega_q - \omega_{j+1}|/\lambda$$



$$\omega_q = \text{atan2} \frac{q_y}{q_x}$$





# De la manipulación

Función de rotación

Ruido Gaussiano

Estructura KdTree-> información tridimensional

Pruebas

32 volúmenes

8 divisiones azimuth

2 divisiones elevación

2 divisiones radiales

Longitud es 352



# De la Ejecución del Código y la Salida

- Después de abrir el archivo .ply se extrae el descriptor.
- Se modifica esta malla de puntos en su rotación o el ruido
- Se vuelve a extraer el descriptor y se compara con la entrada
- Se prueba que el método SHOT es invariante y robusto a la rotación y el ruido haciendo un match entre el descriptor de entrada y el modificado.
- La salida del código es un archivo "shot.txt" donde se almacena la información del descriptor de salida.
- En consola se muestra la coincidencia que tienen estos puntos.

# Salida

```
Match 993: 993  
Match 994: 994  
Match 995: 995  
Match 996: 996  
Match 997: 997  
Match 998: 998
```

```
Described keypoints: 1000.  
Correct Matches: 703 out of 715.  
Matches under threshold: 285.  
Recall: 0.703000, 1-Precision: 0.016783
```

# Pruebas sobre la data del autor y data propia

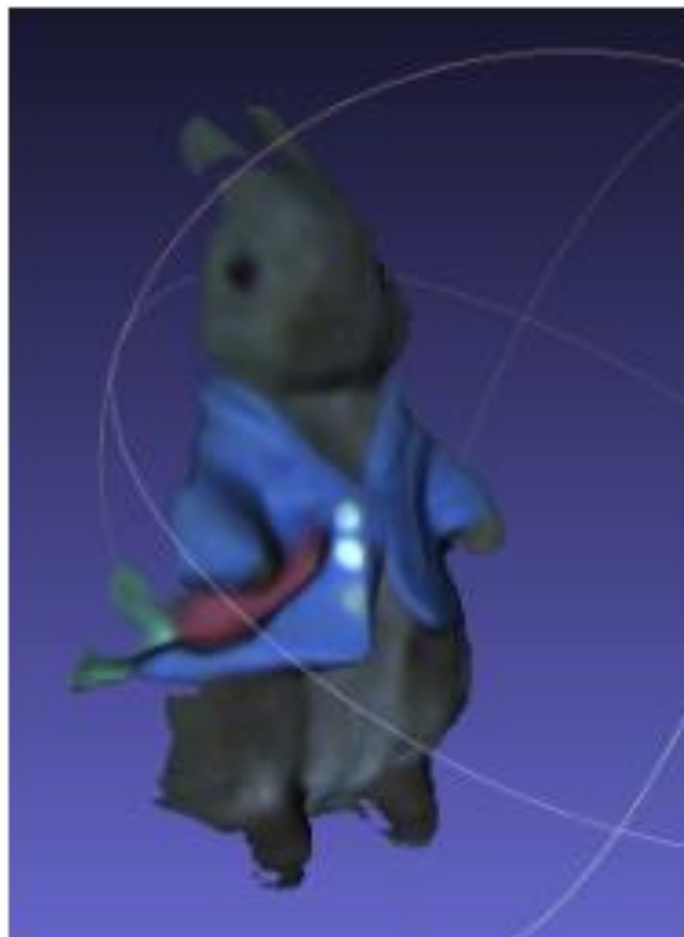


Fig. 10. model 2 - dataset

## Entrada



Fig. 11. escena 1 - dataset

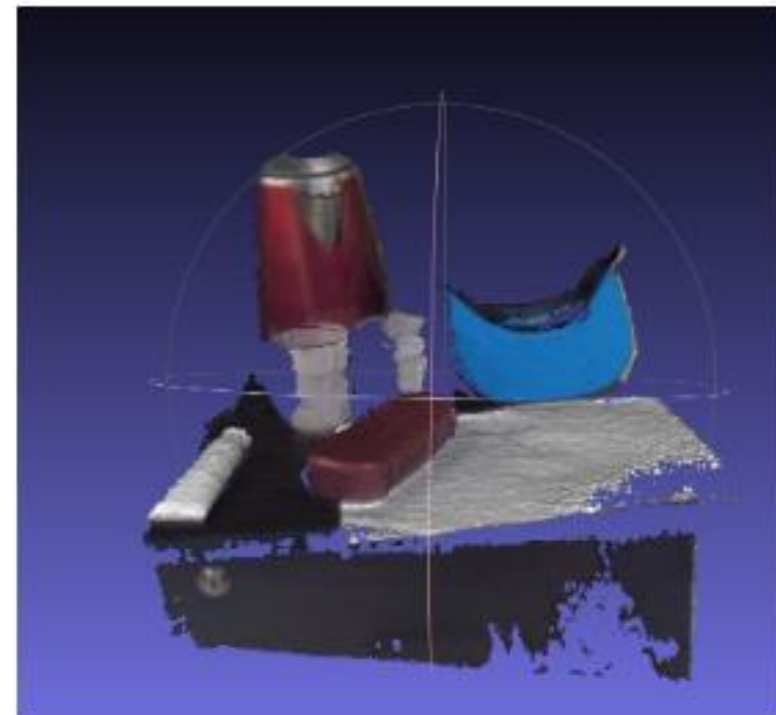


Fig. 14. escena 2 - kinect



Entrada - kinect



Fig. 13. escena 1 - kinect



Fig. 19. modelo 1 - cafetera

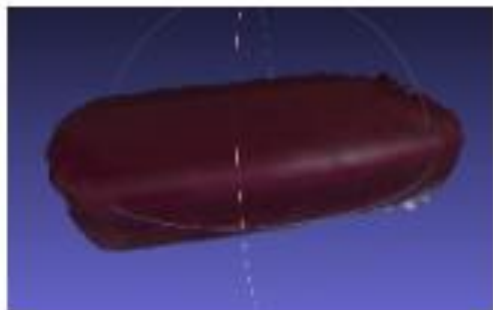
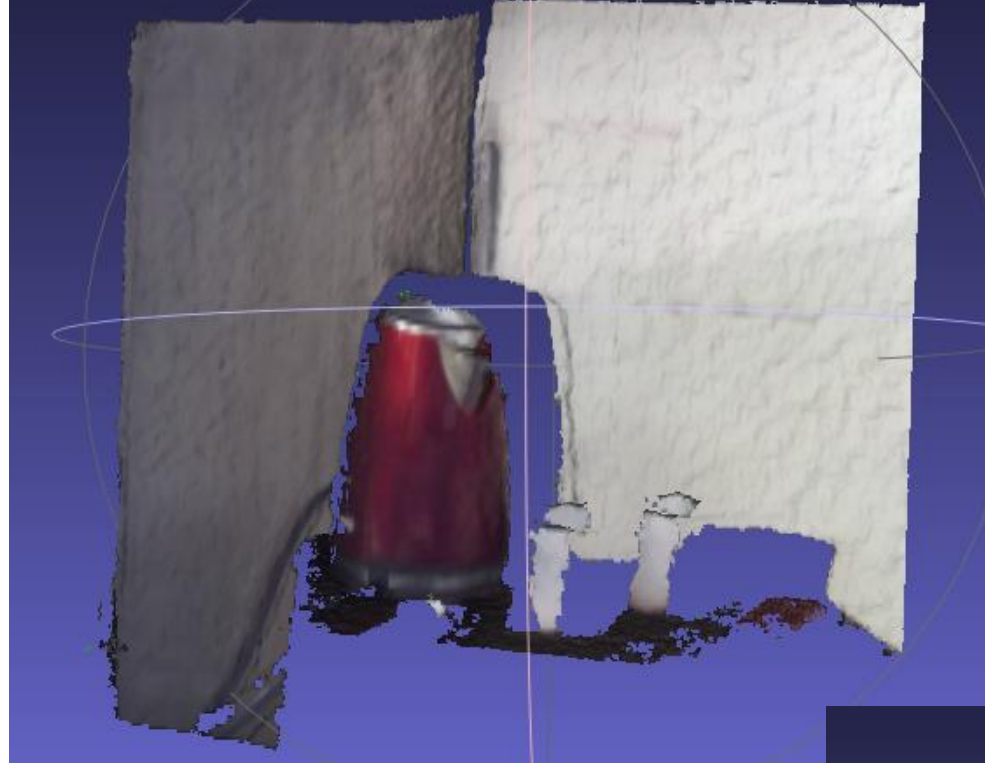
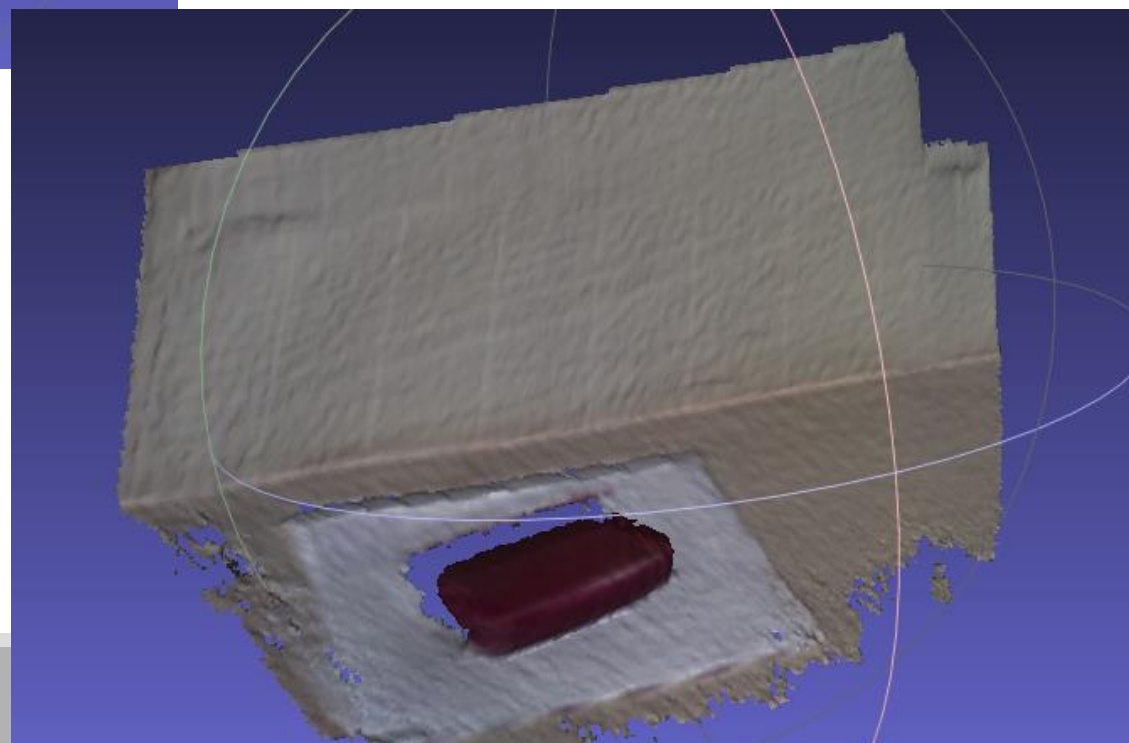


Fig. 20. modelo 2 - caja



Se probaron algunos de los archivos tomados del dataset de los autores y se obtuvieron los siguientes resultados:

Data del Autor			
Nombre del archivo (*.ply)	Matches Correctos	1-Precisión	Recall
model1 (mario)	519 / 524	0.009542	0.519
model2 (conejo)	289 /290	0.003448	0.289
scene1	537 / 544	0.12868	0.537

## Entrada y modificación de ruido y

Se probaron los archivos tomados del kinect y se obtuvieron los siguientes resultados:

Data obtenida del kinect			
Nombre del archivo (*.ply)	Matches Correctos	1-Precisión	Recall
escena 1	15 / 18	0.015	0.16667
escena 2	247 /253	0.023715	0.247
escena 3	6 / 10	0.4	0.006



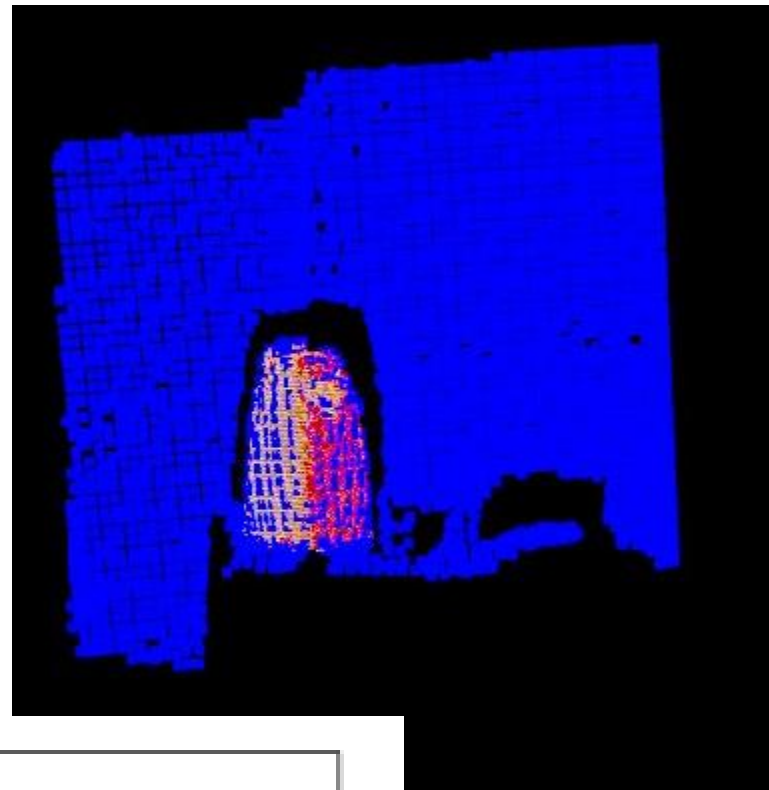
Data del Autor			
Nombre del archivo (*.ply)	Matches Correctos	l-Precisión	Recall
modelo 1 - escena 1 (mario)	508 / 509	0.001965	0.508
modelo 2 - escena 1 (conejo)	259 / 262	0.11450	0.259
modelo 3 - escena 4	167 / 169	0.011834	0.167



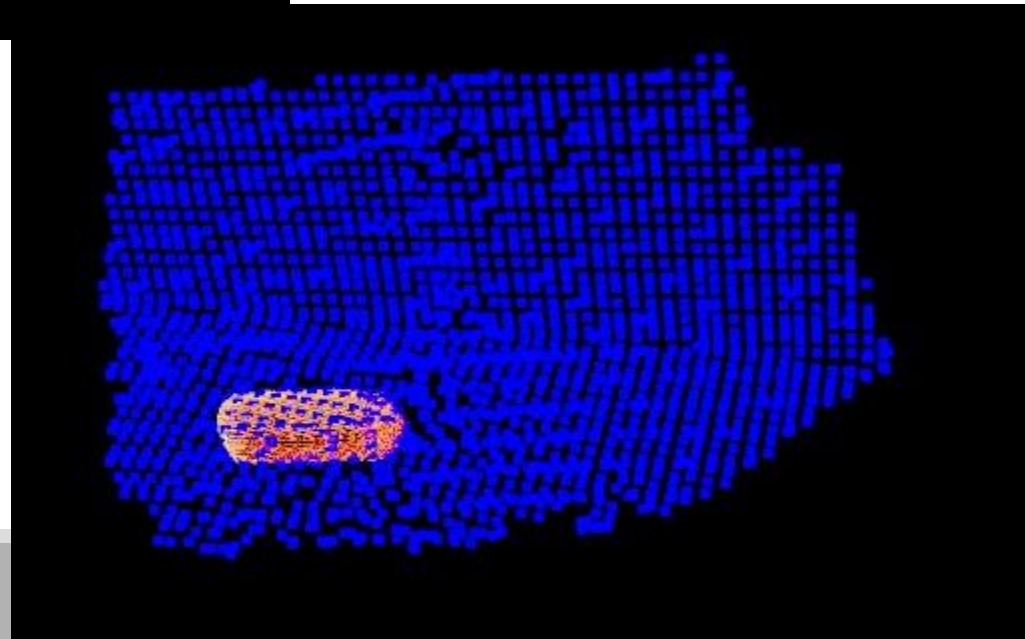
Escenas y objetos







Data obtenida del kinect			
Nombre del archivo (*.ply)	Matches Correctos	1-Precisión	Recall
escena 1 - cafetera	55 / 62	0.112903	0.055
escena 2 - caja	53 / 61	0.131184	0.53



# Original

```
for(int i = 0; i < nActualFeat; i++)
{
    std::vector<float> query;
    for (int j = 0; j < descriptor.getDescriptorLength(); j++)
        query.push_back(noisyDesc[i][j]);

    kdtree.knnSearch(query, knn, dists, 2, cv::flann::SearchParams());
}
```

## Modificación

```
descriptor2.describe(mesh, feat, desc, nActualFeat);
//descriptor3.describe(mesh, feat, desc, nActualFeat);

vtkPolyData* mesh_in2 = LoadPolyData(in2.datapath);
cleanPolyData(mesh_in2);

SHOTParams shotParams2;
SHOTDescriptor descriptor2(shotParams2);
cv::Mat features2(in2.nFeat, descriptor2.getDescriptorLength(), CV_32FC1);
double** desc2;
if (mesh_in2->GetNumberOfPoints())
{
    std::vector<float> query;
    for (int j = 0; j < descriptor.getDescriptorLength(); j++)
        query.push_back(desc2[i][j]);

    kdtree.knnSearch(query, knn, dists, 2, cv::flann::SearchParams());
}
```

# Conclusiones

- SHOT como método híbrido logra trabajar histogramas y representar el comportamiento de firmas
- Diferentes fuentes de datos pueden requerir diferentes parámetros, como radio de búsqueda o numero de muestras. El numero de acierto puede variar ya que los keypoints son aleatorios.
- En comparación con otros métodos SHOT es invariante a la rotación y al ruido ya que presenta mayores coincidencias en la salida de los descriptores y demuestra que es mas eficiente y menos costoso computacionalmente.
- Es posible mediante la información de los descriptores comprobar si un objeto se encuentra en una escena; en este caso para apreciar visualmente se uso la librería PCL.