

# SHOT: Unique signatures of histograms for surface and texture description

Alejandra C. Callo<sup>1</sup>, Jose H. Jaita<sup>1</sup>

<sup>1</sup>CONCYTEC - Ciencia Activa - Maestría en Ciencias de la Computación  
Universidad Católica San Pablo

{alejandra.callo, josejaita}@ucsp.edu.pe

**Resumen.** Este documento describe los descriptores 3D citados en el artículo SHOT: Unique Signatures of Histograms for Surface and Texture Description; que aplica los métodos de firmas e histogramas en la búsqueda de coincidencia de descriptores, para el reconocimiento de objetos 3D, reconstrucción y recuperación de formas, haciendo una comparación con otros métodos y demostrando porque SHOT logra ser invariante y robusto al ser un método híbrido que combina ambas técnicas.

**Palabras-clave:** Reference Axis, Reference Frame, histograma, firmas.

## 1. Introducción

Un descriptor extrae información geométrica de un objeto 3D, es decir una nube de puntos; que mantiene invarianza respecto a sus transformaciones, es robusto al ruido y los detalles.

Este descriptor guarda la información sobre: posición, Color, ángulo, distancia, normales, etc.

Se ha observado dos diferentes métodos para tratar estos descriptores (Histogramas y Firmas). Se muestra en la figura

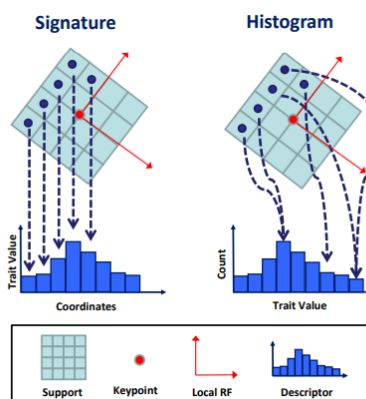


Figura 1. Firmas e Histogramas

## 2. Métodos

Existen diferentes métodos según use histogramas y firmas que se detallaran mas adelante, pero se puede observar un resumen en la siguiente tabla

Table 1: Taxonomy of 3D descriptors.

Method	Category	Unique LRF	Color
StInd [20]	Sign.	No	No
PS [21]	Sign.	No	No
3DPF [22]	Sign.	No	No
3DGSS [8]	Sign.	No <sup>1</sup>	No
KP [5]	Sign.	No	No
LD-SIFT [23]	Sign.	No	No
3D-SURF [24]	Sign.	Yes	No
SI [19]	Hist.	RA	No
LSP [6]	Hist.	RA	No
3DSC [17]	Hist.	No	No
ISS [7]	Hist.	No	No
PFH [27]	Hist.	RA	No
FPFH [16]	Hist.	RA	No
Tensor [18]	Hist.	No	No
HKS [29]	Other	-	No
RoPS [28]	Both	Yes	No
MH [11]	Both	Yes	Yes
SHOT	Both	Yes	Yes

### 2.1. Histogramas

Son una recolección de datos locales (geométricos y topológicos) y guardados en histogramas. Requiere RA o RF Los histogramas describen el soporte codificando contadores de entidades topológicas locales (por ejemplo, vértices, áreas de triángulos de malla) en histogramas de acuerdo con un dominio cuantificado específico (por ejemplo, coordenadas de punto, curvaturas, ángulos normales). Si el dominio del descriptor está basado en coordenadas, entonces también los métodos basados en histogramas requieren la definición de un RF local

#### 2.1.1. Local Surface Patches

Crea un histograma usando las normales entre otros

#### 2.1.2. Spin Image

Crea un histograma en base a los puntos que caen dentro de un radio en base a un eje

### 2.2. Firmas

Son una recopilación de datos alrededor de un punto para luego ser codificado y esto se repite para todos los demás. Requiere un RF

Las firmas son altamente descriptivas gracias al uso de información espacial bien localizada, pero pequeños errores en la definición de la RF local o pequeñas perturbaciones en el rasgo codificado pueden modificar sustancialmente el descriptor final.

### 2.2.1. Point Signature

Se basa en guardar los puntos de intersección alrededor de un punto y su altura

### 2.2.2. Structural Indexing

Es una recolección de ángulos consecutivos de una forma poligonal. (punto y radio)

## 2.3. 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, para ello define un buen reference frame (RF) ya que es invariante a las rotaciones y robusto al ruido. Siendo histogramas que le dan tolerancia al ruido.

Pero para mejorar esto se recolecta información de varios histogramas alrededor imitando el comportamiento de los métodos por firma. Para cada histograma local se acumulan sumas de puntos en contenedores de acuerdo al ángulo que poseen. Para la firma se usa una malla esférica Almacenando los valores radiales y la elevaciones en los ejes. Para seleccionar los histogramas locales ha ser usados.



Figura 2. Estructura

## 2.4. Del código y el Algoritmo

### 2.4.1. De la Compilación

Para poder ejecutar el programa ( en Windows ) se requiere:

- VTK
- CMake
- Visual Studio

1. Primero se descarga los archivos VTK y CMake
2. Para instalar VTK se hace desde CMake ya que requiere ser compilado, se crea una carpeta que denominaremos "bin" y referenciandola desde CMake

3. Se configura utilizando la versión de visual con la que se cuenta
4. Se activa las opciones testingRendering, TestingCore, buildExample y se vuelve a configurar, después de la confirmación se genera el proyecto.
5. Se ejecuta la solución generada y se compila los archivos ALLBUILD e INSTALL
6. Se genera el proyecto SHOT desde CMAKE configurando la dirección del opencv
7. Se accede a la solución desde Visual Studio, compilando primero AllBuild
8. Antes de EJECUTAR el programa se hace 3 modificaciones al código debido a la versión del VTK cambiando:

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

Remplazando por:

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

9. Se cambia el path del archivo de entrada y se Ejecuta

#### 2.4.2. La 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 sub-formatos: una representación ASCII y una versión binaria.



**Figura 3. Entrada: "Mario.ply"**

### 3. De la Matemática

#### 3.0.1. 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  $P_i$

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

Figura 4. Función de Covarianza

En la propuesta 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

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

Figura 5. Función de covarianza modificada

Que en código se encuentra en la función `void getSHOTLocalRF(vtkPolyData *cloud, vtkIdList *NNpoints, double radius, int index, float *rfc)`

```
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;
```

```

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++;
    }
}

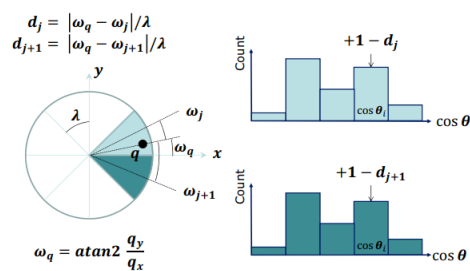
```

**Figura 6. Esta función Calcula la diferencia entre el punto actual y el origen Actualizando el valor de la distancia = Radio - Distancia Euclídea A partir de ahí calcula los autovalores y autovectores para ordenarlos y normalizarlos Utiliza la función de JAcobi y producto cruz para actualizar el kdTree donde se manipula la información**

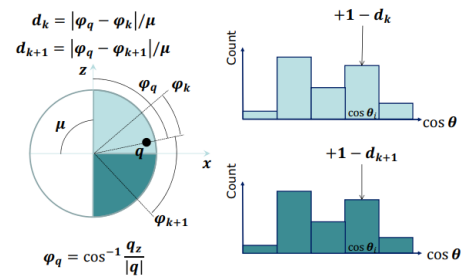
### 3.0.2. Función Interpolación Cuadrilineal

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

En Código se encuentran en la función de interpolación de canales



(b) Interpolation on azimuth



(c) Interpolation on elevation

Se encuentran en la función **void SHOTDescriptor::interpolateDoubleChannel(vtkPolyData\* cloud, vtkIdList\* NNpoints, const std::vector<double> distances, const double centralPoint[3], float rf[9], std::vector<double> binDistanceShape, std::vector<double> binDistanceColor, double\* shot )**

```
//Interpolation on the inclination (adjacent vertical volumes)
double inclinationCos = zInFeatRef / distance;
if (inclinationCos < -1.0) inclinationCos = -1.0;
if (inclinationCos > 1.0) inclinationCos = 1.0;

double inclination = acos( inclinationCos );

assert(inclination >= 0.0 && inclination <= DEG_180_TO_RAD);

if( inclination > DEG_90_TO_RAD || (std::abs(inclination - DEG_90_TO_RAD) < 1e-30 && zInFeatRef <= 0)){

    double inclinationDistance = (inclination - DEG_135_TO_RAD) / DEG_90_TO_RAD;
    if(inclination > DEG_135_TO_RAD)
        intWeight += 1 - inclinationDistance;
    else{
        intWeight += 1 + inclinationDistance;
        assert( (desc_index + 1) * (nr_bins+1) + step_index >= 0 && (desc_index + 1) * (nr_bins+1) + step_index < m_descLength);
        shot[ (desc_index + 1) * (nr_bins+1) + step_index] -= inclinationDistance;
    }
}
else{

    double inclinationDistance = (inclination - DEG_45_TO_RAD) / DEG_90_TO_RAD;
    if(inclination < DEG_45_TO_RAD)
        intWeight += 1 + inclinationDistance;
    else{
        intWeight += 1 - inclinationDistance;
        assert( (desc_index - 1) * (nr_bins+1) + step_index >= 0 && (desc_index - 1) * (nr_bins+1) + step_index < m_descLength);
        shot[ (desc_index - 1) * (nr_bins+1) + step_index] += inclinationDistance;
    }
}

}

//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));

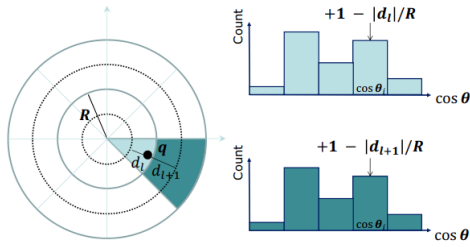
assert((azimuthDistance < 0.5 || areEquals(azimuthDistance, 0.5)) && (azimuthDistance > -0.5 || areEquals(azimuthDistance, -0.5)));

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;
}

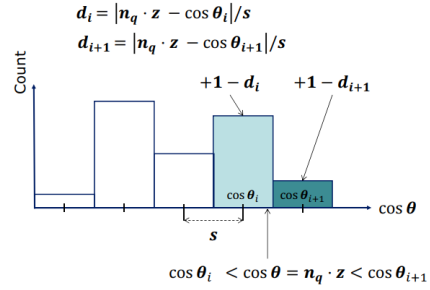
}
```

Amabas funciones Calculan la norma Euclideana, limitando el rango para interpolar.

Recordar que la interpolación cuadrática solo se aplica cuando las formas y colores están activadas dentro del descriptor



(d) Interpolation on distance



(a) Interpolation on normal cosines

Se encuentran en la función **void SHOTDescriptor::interpolateSingleChannel(vtkPolyData\* cloud, vtkIdList\* NNpoints, const std::vector<double> distances, const double centralPoint[3], float rf[9], std::vector<double> binDistance, int nr\_bins, double \* shot)**

```
//Interpolation on the cosine (adjacent bins in the histogram)
binDistanceShape[i_idx] -= step_index_shape;
binDistanceColor[i_idx] -= step_index_color;

double intWeightShape = (1- std::abs(binDistanceShape[i_idx]));
double intWeightColor = (1- std::abs(binDistanceColor[i_idx]));

//19.9.17: (m_params.shapeBins + 1) used instead of m_params.shapeBins (same for colorBins)
if( binDistanceShape[i_idx] > 0)
    shot[ volume_index_shape + ((step_index_shape + 1) % (m_params.shapeBins + 1)) ] += binDistanceShape[i_idx];
else
    shot[volume_index_shape + ((step_index_shape + m_params.shapeBins) % (m_params.shapeBins + 1)) ] -= binDistanceShape[i_idx];

if( binDistanceColor[i_idx] > 0)
    shot[volume_index_color + ((step_index_color + 1) % (m_params.colorBins + 1)) ] += binDistanceColor[i_idx];
else
    shot[volume_index_color + ((step_index_color + m_params.colorBins) % (m_params.colorBins + 1)) ] -= binDistanceColor[i_idx];
```

Figura 7. Interpolación sobre la normal de los cosenos



```

//Interpolation on the distance (adjacent husks)
if(distance > m_radius1_2){ //external sphere

    double radiusDistance = (distance - m_radius3_4) / m_radius1_2;

    if(distance > m_radius3_4) //most external sector, votes only for itself
    {
        intWeightShape += 1 - radiusDistance; //weight=1-d
        intWeightColor += 1 - radiusDistance; //weight=1-d
    }
    else{ //3/4 of radius, votes also for the internal sphere
        intWeightShape += 1 + radiusDistance;
        intWeightColor += 1 + radiusDistance;
        shot[ (desc_index - 2) * (m_params.shapeBins+1) + step_index_shape] -= radiusDistance;
        shot[ shapeToColorStride + (desc_index - 2) * (m_params.colorBins+1) + step_index_color] -= radiusDistance;
    }
}
else { //internal sphere

    double radiusDistance = (distance - m_radius1_4) / m_radius1_2;

    if(distance < m_radius1_4) //most internal sector, votes only for itself
    {
        intWeightShape += 1 + radiusDistance;
        intWeightColor += 1 + radiusDistance; //weight=1-d
    }
    else{ //3/4 of radius, votes also for the external sphere
        intWeightShape += 1 - radiusDistance; //weight=1-d
        intWeightColor += 1 - radiusDistance; //weight=1-d
        shot[ (desc_index + 2) * (m_params.shapeBins+1) + step_index_shape] += radiusDistance;
        shot[ shapeToColorStride + (desc_index + 2) * (m_params.colorBins+1) + step_index_color] += radiusDistance;
    }
}

```

**Figura 8. Interpolación sobre la distancia**

Cada punto se acumula en un contenedor de histograma local, se realiza la interpolación cuadrilínea con sus vecinos, es decir, el contenedor vecino en el histograma local y los contenedores que tienen el mismo índice en los histogramas locales correspondientes a la cuadrícula.

En particular, cada contenedor se incrementa en un peso de  $1 - d$  para cada dimensión. En cuanto al histograma local,  $d$  es la distancia de la entrada actual desde el valor central del contenedor. En cuanto a la elevación y el azimuth,  $d$  es la distancia angular de la entrada desde el valor central del volumen. A lo largo de la dimensión radial,  $d$  es la distancia euclidiana de la entrada desde el valor central del volumen. A lo largo de cada dimensión,  $d$  se mide en unidades del histograma o espacio entre cuadrículas, es decir, se normaliza por la distancia entre dos contenedores o volúmenes vecinos.

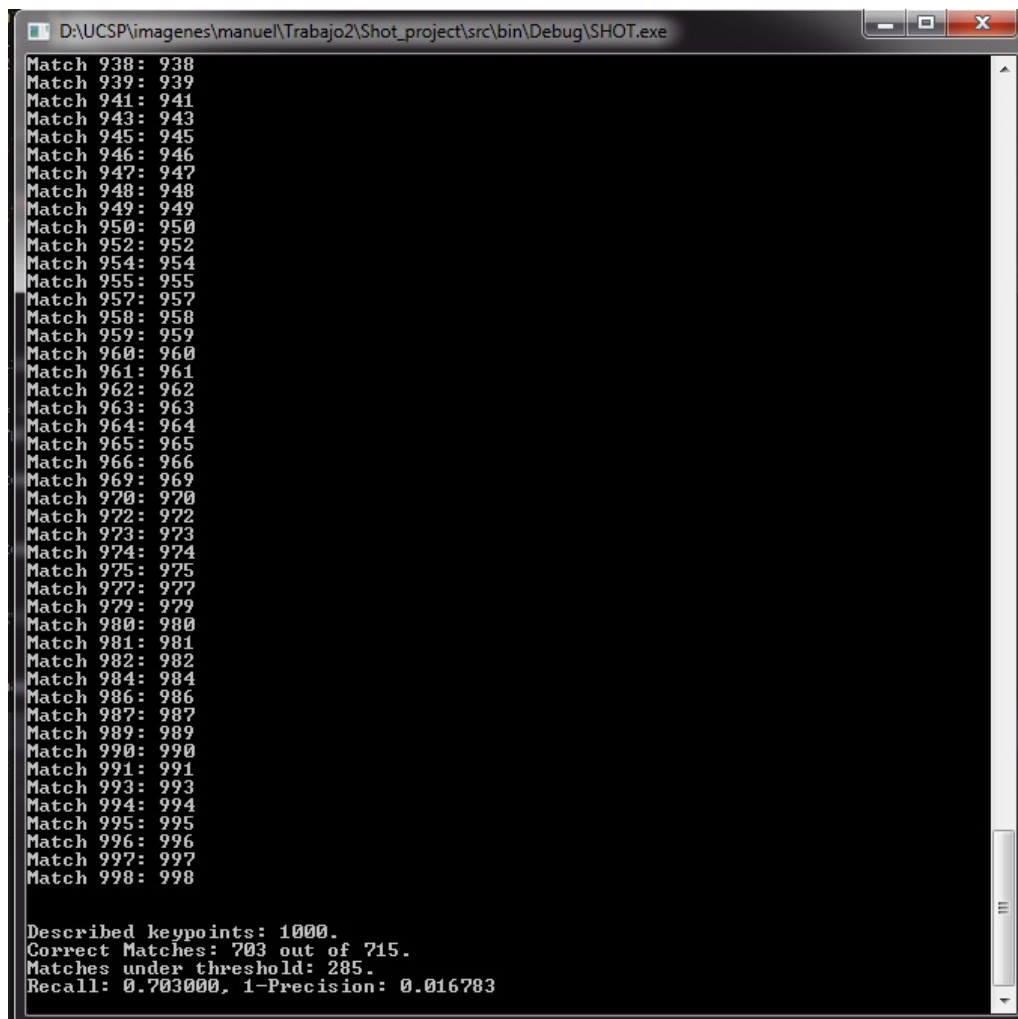
#### 4. De la Ejecución del Código y la Salida

- 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.



```
D:\UCSP\imagenes\manuel\Trabajo2\Shot_project\src\bin\Debug\SHOT.exe
Match 938: 938
Match 939: 939
Match 941: 941
Match 943: 943
Match 945: 945
Match 946: 946
Match 947: 947
Match 948: 948
Match 949: 949
Match 950: 950
Match 952: 952
Match 954: 954
Match 955: 955
Match 957: 957
Match 958: 958
Match 959: 959
Match 960: 960
Match 961: 961
Match 962: 962
Match 963: 963
Match 964: 964
Match 965: 965
Match 966: 966
Match 969: 969
Match 970: 970
Match 972: 972
Match 973: 973
Match 974: 974
Match 975: 975
Match 977: 977
Match 979: 979
Match 980: 980
Match 981: 981
Match 982: 982
Match 984: 984
Match 986: 986
Match 987: 987
Match 989: 989
Match 990: 990
Match 991: 991
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
```

Figura 9. Salida del código

## 5. Conclusiones

- SHOT como método híbrido logra trabajar histogramas y representar el comportamiento de firmas
- 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.
- Presenta también la manipulación del espacio RGB a CIELAB, siendo necesaria su conversión para trabajar texturas