Libro OpenCv

https://drive.google.com/file/d/1UqrQkaPYbJ4DeyneyDDz2Ye4FbPRtWAG/view?usp=share_link

Semana 2: Visión Estéreo

Práctica 1: Calibración Estéreo

Crea un programa llamado *stereo_calibrate* que lea desde fichero las imágenes stereo de calibración en el siguiente enlace y obtenga los parámetros de calibración estéreo:

Para ello use las funciones:

`cv::findChessboardCorners, cv::cornerSubPix y cv::stereoCalibrate`

Como salida el debe crear un fichero .yml .

```
Uso:
./stereo_calibrate dir_with_images out.yml
```

Puede ayudarse de la clase DirReader que lee los archivos de un directorio:

```
DirReader Dir;
auto files=Dir.read(argv[1],".jpg",DirReader::Params(true));
```

`o bien puede usar la clase std::filesystem de C++ 17`

El fichero resultante deberá ser tal y como éste.

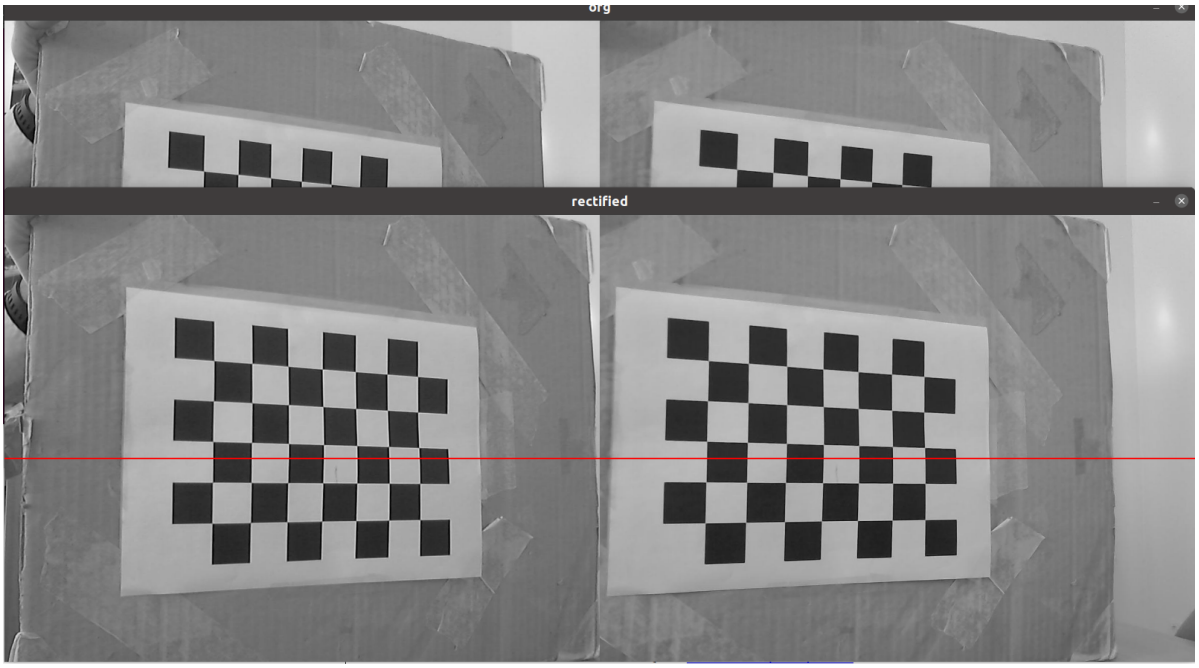
Nota:
cv::Size CheckerBoardSize={7,5};//
double SquareSize=0.02875;//size of each square
Usar el critero de optimización:
cv::TermCriteria(cv::TermCriteria::MAX_ITER + cv::TermCriteria::EPS, 60, 1e-6)

Práctica 2: Comprobación de imágenes rectificadas

Cree el programa *stereo_checkundistorted* que reciba como entrada una imágen estéreo y el correspondiente fichero de calibración estéreo. El programa debe mostrar dos ventanas. En la primera ventana se muestran las imágenes izquierda y derecha originales de la cámara estéreo. Al pasar el ratón por encima, se dibujará de forma dinámica una línea horizontal que cruza ambas imágenes.

Además, deberá mostrar otra ventana similar donde se muestran las imágenes izquierda y derecha después de la rectificación. Igualmente, el programa deberá pintar una línea horizontal al moverse sobre la ventana.

De esta manera, podremos comprobar visualmente cómo de bien la rectificación ha funcionado, creando dos cámaras paralelas.



Uso:
./stereo_checkundistorted stereo_image.jpg stereocalibrationfile.yml

Para rectificar las imágenes, puede usar el siguiente código:

```
//Structure that contains the Stereo Pair Calbration information.
//This will be calculated using stereo_calibrate
struct StereoParams{
```

```
    cv::Mat mtxL,distL,R_L,T_L;
    cv::Mat mtxR,distR,R_R,T_R;
    cv::Mat Rot, Trns, Emat, Fmat;
};

void rectifyStereoImages(const StereoParams &sti,cv::Mat &left,cv::Mat &rigth){
    cv::Mat rect_l, rect_r, proj_mat_l, proj_mat_r, Q;
    cv::Mat Left_Stereo_Map1, Left_Stereo_Map2;
    cv::Mat Right_Stereo_Map1, Right_Stereo_Map2;
    cv::stereoRectify(sti.mtxL,
sti.distL,sti.mtxR,sti.distR,left.size(),sti.Rot,sti.Trns,
                      rect_l,rect_r,proj_mat_l,proj_mat_r,
                      Q,cv::CALIB_ZERO_DISPARITY, 0);
    cv::initUndistortRectifyMap(sti.mtxL,sti.distL,rect_l,proj_mat_l,
                                left.size(),CV_16SC2,
                                Left_Stereo_Map1,Left_Stereo_Map2);
    cv::initUndistortRectifyMap(sti.mtxR,sti.distR,
                                rect_r,proj_mat_r,
                                left.size(),CV_16SC2,
                                Right_Stereo_Map1,Right_Stereo_Map2);
    cv::Mat AuxImage, Right_nice;
    cv::remap(left, AuxImage, Left_Stereo_Map1, Left_Stereo_Map2,
              cv::INTER_LANCZOS4,cv::BORDER_CONSTANT,0);
    AuxImage.copyTo(left);
    cv::remap(rigth,  AuxImage,  Right_Stereo_Map1,  Right_Stereo_Map2,
              cv::INTER_LANCZOS4,cv::BORDER_CONSTANT,0);
    AuxImage.copyTo(rigth);
}
```

Práctica 3: Mapa Denso de Disparidad Estéreo

Cree el programa *stereo_disparity* que recibe como entrada una imágen estéreo, el fichero de calibración estéreo, y como resultado calcula la disparidad entre las imágenes en cada pixel. Para aquellos puntos con disparidad válida, genera como salida un fichero con formato OBJ que podrá visualizar con el programa MeshLab. Para probar puede utilizar las imágenes en el siguiente enlace.

Uso:

./stereo_disparity stereo_image.jpg calibration.yml out.obj

Para realizar el proceso, deberá

1) Cargar la imágen estéreo
2) Rectificar las imágenes
3) Utilizar la clase cv::StereoBM para realizar el cálculo de la disparidad.

4) Convierta la disparidad obtenida a valores 32bits flotantes

```
// Converting disparity values to CV_32F from CV_16S
    disp.convertTo(disparity,CV_32F, 1.0);
    disparity=disparity/16.f;
```

5) Para aquellos puntos con disparidad > 10, triangule usado las ecuaciones básica del par estéreo: Z= |T|*f /d; X= (x-cx)*Z/f ; Y= (y-cy)*Z/f;
6) Guarde los puntos a formato obj. Puede utilizar la siguiente función para ello:

```
void writeToOBJ(std::string path,std::vector<cv::Point3f> points){

    std::ofstream file(path,std::ios::binary);
    for(auto p:points)
        file<<"v "<<p.x<<" "<<p.y<<" "<<p.z<<endl;
}
```

En el siguiente enlace tenéis el fichero resultado para la primera image reconstruction/m001.jpg

Ahora bien, si en lugar de usar la imagen con el tamaño original, la reducimos a la mitad obtenemos la siguiente reconstrucción.

Para hacerlo, se deberá reducir a la mitad también el *f,cx* y *cy*.

Práctica 4: Reconstrucción 3D dispersa con par estéreo

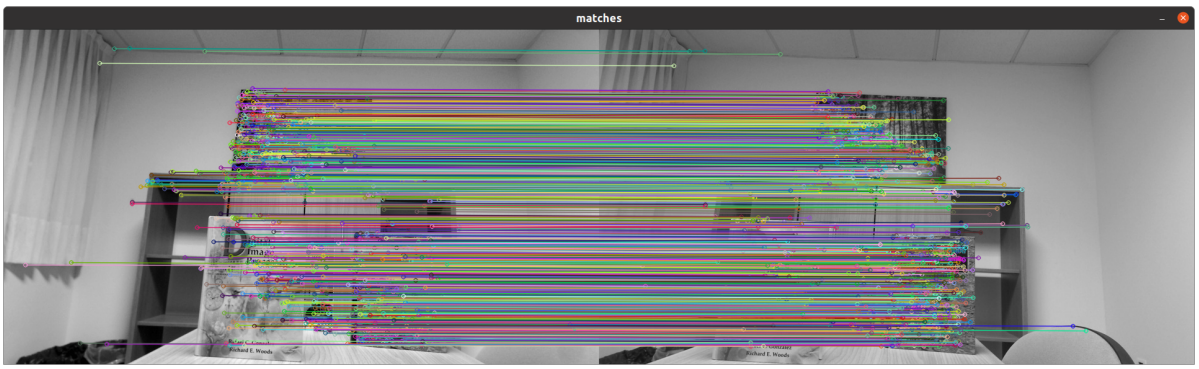
El programa anterior utilizaba block-matching para encontrar emparejamientos entre imágenes. En este ejercicio vamos a buscar los emparejamientos usando keypoints usando el descriptor AKAZE.

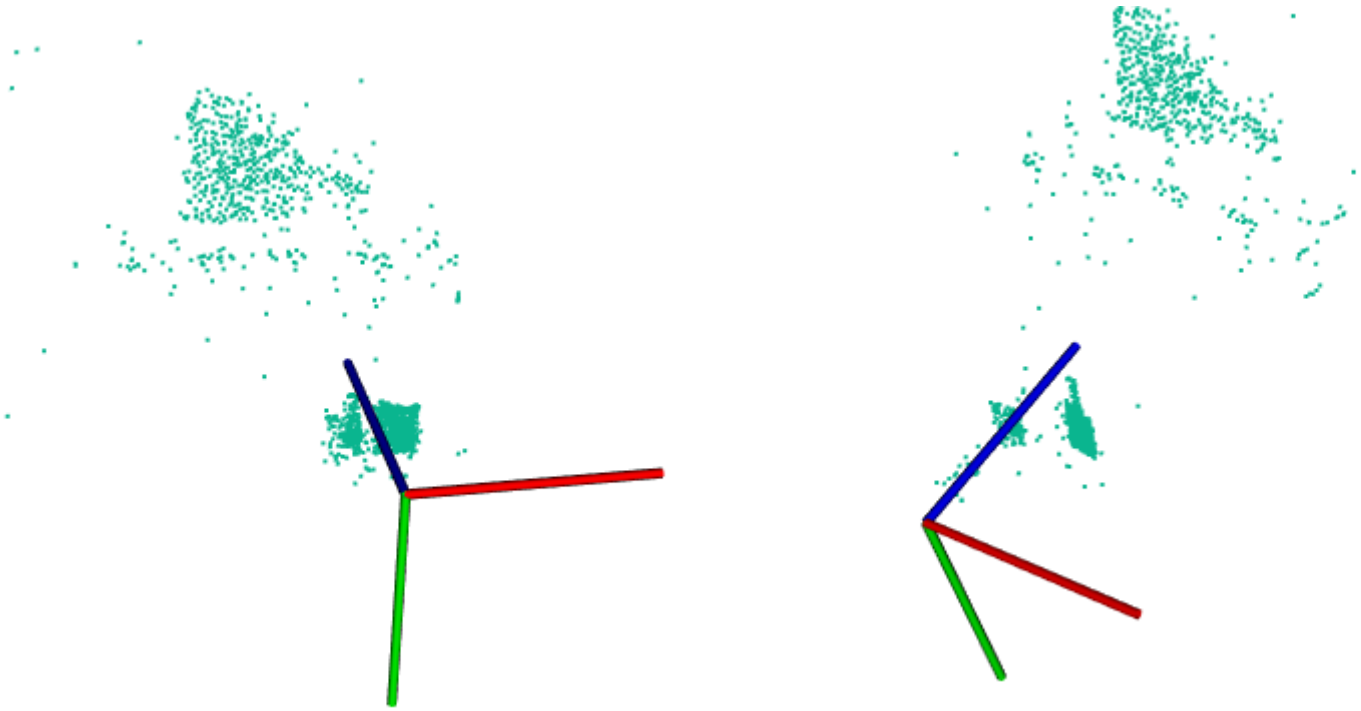
El programa debe tener el nombre *stereo_sparse* y debe usarse como.
Uso:
./stereo_sparse *stereo_image.jpg calibration.yml out.obj*
El programa deberá hacer lo siguiente.

1) Cargar la imágen estéreo
2) Rectificar las imágenes
3) Busca keypoints en ambas imágenes usando AKAZE y el descriptor matcher BruteForce-Hamming

```
auto Detector=cv::AKAZE::create(cv::AKAZE::DESCRIPTOR_MLDB,  0,  3, 1e-4f,8 );
Detector ->detectAndCompute(left, cv::Mat(), keypoints_query, descriptors_query);
Detector ->detectAndCompute(rigth, cv::Mat(), keypoints_train,
descriptors_train);
auto matcher = cv::DescriptorMatcher::create("BruteForce-Hamming");
matcher->match(descriptors_query, descriptors_train, matches, cv::Mat());
```

4) Filtra los matches y deja aquellos que están en líneas horizontales.
5) Dibuja los matches obtenidos antes y después del filtrado
6) Triangula los matches y los guarda en el fichero de salida con el formato OBJ.





Semana 3: 3D Multivista

Esta serie de prácticas usarán los datos proporcionados en la carpeta https://drive.google.com/drive/folders/1_MGGey6ooQRUMrdAV6WTebzDSJPEwEs2?usp=sharing

Práctica 5: Cálculo de la matrix fundamental

Realice una función programa que dadas dos imágenes, calcule la matriz fundamental y muestre, cómo en la práctica 2, la correspondiente línea epipolar de un punto en la imagen opuesta.

El objetivo de esta práctica es crear una función que dadas dos imágenes de la misma cámara (sin distorsión), retorne la matriz fundamental entre ambas cv::Mat fundamental(cv::Mat im1,cv::Mat im2); Esta función será necesaria para la siguiente práctica.

El programa debe pues, en primer lugar, cargar las dos imágenes así como los parámetros de la cámara. Después, eliminará la distorsión de la imagen, obteniendo una versión sin distorsión usando la función cv::undistort() y llamará a la función fundamental().
Una vez calculada la matriz fundamental, el programa mostrará en una ventana la imagen im1, y en otra la imagen im2. Cuando el ratón se mueva sobre la ventana de im1, se dibujará la línea epipolar correspondiente en im2. La línea epipolar se calculará utilizando la matriz fundamental.

```
//ESQUEMA DE LA FUNCION MAIN
int main(int argc,char **argv){
    cv::Mat im1=cv::imread(argv[1]);
    cv::Mat im2=cv::imread(argv[2]);
    auto CP=readCameraParams(argv[3]);
    im1=removeDistortion(im1,CP);
    im2=removeDistortion(im2,CP);
    cv::Mat F=fundamental(im1,im2);
    showEpipolar(im1,im2,camK,F);
    while(cv::waitKey(0)!=27) ;
}
```

Los pasos que la función fundamental() debe seguir son:

1.) Calcular keypoints en ambas imágenes y obtener matches usando un algoritmo robusto. Para ello deberá crear la siguiente función:
std::vector<cv::DMatch> KpMatch(std::vector<cv::KeyPoint> keypoints_query ,cv::Mat descriptors_query, std::vector<cv::KeyPoint> keypoints_train, cv::Mat descriptors_train);
La función recibe los keypoints y sus descriptores y devuelve como salida los matches seleccionados. Para establecer matches, usaremos fuerza bruta y pediremos al matcher que nos devuelva los dos mejores matches:

```
std::vector< std::vector<cv::DMatch> > matches;
auto matcher = cv::DescriptorMatcher::create("BruteForce-Hamming");
matcher->knnMatch(descriptors_query, descriptors_train, matches, 2);
```

Para decidir si un posible match es aceptado, debe cumplir las siguientes condiciones:
    a) La distancia Hamming debe ser menor que 80

b) La diferencia entre octaves debe ser menor que 2
c) La distancia del mejor match debe ser al menos un 20% más baja que la distancia al segundo mejor match.

2.) Calcular la matriz fundamental usando la función

```
vector<cv::Point2f> points_query,points_train;//rellenar
cv::Mat inliers;
cv::findFundamentalMat(points_query,points_train,cv::FM_RANSAC,0.999,1.0,1000,inliers);
```

Para usar la función, deberemos crear los vectores points_query,points_train que tienen los puntos 2d de los matches establecidos. Estamos utilizando el algoritmo RANSAC que es capaz de eliminar los outliers que se hayan quedado en el match.

3) Eliminamos los outliers del conjunto de matches

```
for(size_t i=0;i<inliers.total();i++)
      if( !inliers.ptr<uchar>(0)[i]){
            //ES OUTLIER
      }
```

4) Mostraremos los nuevos matches por pantalla para ver el cambio producido.

2.) Cálculo y dibujado de de la línea epipolar showEpipolar()
La matriz fundamental calculada nos permite saber la línea epipolar en la imagen im2, de un punto en la imagen im1. Para ello, basta con multiplicar el punto p de im1 por la matriz.

$L=Ep$

obteniendo L=(a,b,c)^t que son los coeficientes de la línea epipolar ax+by+c=0 en la imagen im2.

Para dibujarla, deberemos calcular la intersección de la línea con los límites izquierdo y derecho de la imagen. Esto puede hacerse simplemente sustituyendo x=0 y x=im2.cols en la ecuación de la recta y despejando la y. Si lo desea, puede usar la implementación proporcionada en este link.

Práctica 6: Triangulación de matches usando la matriz epipolar

En este programa vamos a obtener una reconstrucción 3d de los keypoints que se han obtenido en el programa anterior. Vamos a partir del programa anterior y lo vamos a ampliar para calcular los puntos 3D entre las imágenes:

```
//ESQUEMA DE LA FUNCION MAIN
int main(int argc,char **argv){
    cv::Mat im1=cv::imread(argv[1]);
```

```
        cv::Mat im2=cv::imread(argv[2]);
        auto CP=readCameraParams(argv[3]);
        im1=removeDistortion(im1,CP);
        im2=removeDistortion(im2,CP);
        cv::Mat F=fundamental(im1,im2);
        vector<cv::Point2f> vpoints=Triangulate(im1,im2,F,CP);
        writeToPCD(argv[4],vpoints);
        while(cv::waitKey(0)!=27) ;
    }
```

1.) En primer lugar, vamos a modificar la función KpMatch() para que acepte como quinto argumento la matriz fundamental. Ahora, esta nueva versión añadirá como condición para establecer un match, que el punto en la im2, esté a cierta distancia de la línea epipolar que le corresponde (usar el valor 4 pixels). Para ello, puede utilizar la siguiente función que retorna la distancia del punto kp2 a la línea epipolar establecida por kp1 según F.
//kp1 es train
//kp2 es query

```
float epipolarLineSqDist(const cv::Point2f &kp1,const cv::Point2f &kp2,const
cv::Mat &F){
        if(F.empty())return 0;
        // Epipolar line in second image l = x1'F = [a b c]
        auto a = kp1.x*F.at<float>(0,0)+kp1.y*F.at<float>(1,0)+F.at<float>(2,0);
        auto b = kp1.x*F.at<float>(0,1)+kp1.y*F.at<float>(1,1)+F.at<float>(2,1);
        auto den = a*a+b*b;
        if(den==0) return std::numeric_limits<float>::max();
        auto c = kp1.x*F.at<float>(0,2)+kp1.y*F.at<float>(1,2)+F.at<float>(2,2);
        auto num = a*kp2.x+b*kp2.y+c;
        return num*num/den;
    };
```

Nótese que se asume en este caso que F es float. Si no es vuestro caso, realizad el cambio necesario.

Esta nueva versión de la función KpMatch() es más precisa y evita matches que no cumplen la restricción epiplolar.

Pro Tip! :
Para no tener que crear una nueva función KpMatch2(), es posible añadir la matrix como parámetro con valor por defecto vacío:

```
std::vector<cv::DMatch>  KpMatch( std::vector<cv::KeyPoint> keypoints_query ,cv::Mat
descriptors_query, std::vector<cv::KeyPoint> keypoints_train, cv::Mat
descriptors_train,cv::Mat F=cv::Mat());
```

Como se puede observar, si la matriz F está vacía, la función `epipolarLineSqDist()` devolverá 0 y no afectará al cómputo. Así, la función podrá seguir siendo utilizada por la función fundamental() sin necesidad de cambio.

2.) Función Triangulate()
La función triangulate va a repetir el proceso de búsqueda de matches entre las imágenes, esta vez usando la matriz fundamental. Después, extraerá de la matriz la

Rotación y traslación entre las cámaras, y finalmente triangulará. Los pasos a seguir son:

a) Calcula matches entre las imágenes usando la nueva función *KpMatch* que usa la matriz F
b) Calcular la pose (R|t) entre la cámara im1 y im2 para poder triangular. Para ello, vamos a derivar la matrix Esencial E, a partir de F.

$$\mathbf{E} = (\mathbf{K}')^{\top}\ \mathbf{F}\ \mathbf{K}.$$

Donde K es la matriz de parámetros intrínsecos de la cámara. Para ello usaremos la función de openCV.

Teniendo E, podemos usar la función __recoverPose() que os proporciono.

NOTA: No debe haber outliers en los puntos proporcionados. Usad los inliers con ransac que se obtienen de llamar a findFundamentalMatrix

que nos calculará la R y t.

c) Triangular los matches. Dados dos puntos en imágenes im1 y im2, así como la posición relativa R,t de las cámaras, podremos obtener la posición 3D del punto correspondiente mediante triangulación. Puede usar el código de la función triangulate proporcionada en el siguiente fichero. Es importante indicar que la función devolverá false si la triangulación obtenida no es fiable. En ese caso, el match obtenido debería ser eliminado.

NOTA: Ejemplos del resultado esperado se pueden obtener en los siguientes links cam0, cam2.

Práctica 7: Añadiendo una tercera imagen (opcional 20%)

El objetivo ahora es añadir una tercera imagen a la ecuación. Del ejercicio anterior, hemos obtenido una serie de keypoints en la imagen im1, sus matches en la imagen im2, así como su posición 3D en la escena (respecto de la imagen im1).
Ahora, vamos a suponer que tenemos disponible otra imagen im3, que está cerca de la imagen im1. Vamos a repetir el proceso de obtener matches de im1 a im3 y triangular. Sin embargo, en esta ocasión no utilizaremos la matriz Fundamental a partir de los matches, si no que obtendremos la R|t entre ambas usando el algoritmo PnP y después la fundamental.

Vamos expandir main para que sea

```
//ESQUEMA DE LA FUNCION MAIN
int main(int argc,char **argv){
```

```
    cv::Mat im1=cv::imread(argv[1]);
    cv::Mat im2=cv::imread(argv[2]);
    auto CP=readCameraParams(argv[3]);
    im1=removeDistortion(im1,CP);
    im2=removeDistortion(im2,CP);
    cv::Mat im3=cv::imread(argv[5]);
    im3=removeDistortion(im3,CP);
    cv::Mat F=fundamental(im1,im2);
    vector<cv::Point2f> vpoints=Triangulate(im1,im2,F,CP,im3);
    writeToPCD(argv[4],vpoints);
    while(cv::waitKey(0)!=27) ;
}
```

La nueva función triangulate debe extenderse para hacer los siguiente un vez triangulados los puntos de im1-im2.

a) Usando solo los keypoints de im1 triangulados, buscaremos sus correspondientes matches (KpMatch()) en la imagen im3. En este caso no tenemos matrix fundamental para realizar el filtrado aún.
b) En lugar de eso, una vez obtenidos los matches iniciales, vamos a usar la función cv::solvePnPRansac. Esta función nos dará la pose (R|t) de la cámara im3 respecto de los puntos, es decir, respecto de im1. Además, la función usa el algoritmo RANSAC para filtrar matches incorrectos. Así, eliminaremos los matches incorrectos para no considerarlos.
c) Conocida la (R|t) entre las cámaras, es posible obtener la matrix Fundamental (paso inverso al que hicimos anteriormente). Para ello, usaremos la función siguiente.
d) Los matches válidos entre im1-im3 son puntos que se ven en ambas cámaras y para las que tenemos información 3D. Ahora, vamos a tratar de buscar nuevos matches entre las dos cámaras y triangularlos. Para ellos, volvemos a considerar todos los keypoints en la im1, y eliminamos aquellos que han sido matcheados con la im3. Lo mismo hacemos para los keypoints de la im3 matcheados con im1. Con los que nos quedan, volvemos a buscar correspondencias usando la función KpMatch(), usando en esta ocasión la matrix fundamental calculada en el paso anterior.
e) Finalmente, triangulamos los matches encontrados y los añadimos al vector de puntos 3D.

NOTA: Ejemplos del resultado esperado se pueden obtener en los siguientes links cam0, cam2.