VISIÓN POR COMPUTADOR
Trabajo-3

Estimación de la matriz de la cámara

En este ejercicio se van a dibujar los puntos correspondientes a la cámara aleatoria y simulada de diferentes colores para diferenciarlos, la función general que define este ejercicio es, **drawProjectionPoints()** no obstante debajo se irán explicando las funciones que se han ido usando en ésta:

```
/**
* Primero se crea la cámara aleatoria, saca los puntos 3D, los proyecta y obtiene que puntos se corresponden en la imagen. Repite el paso anterior pero con la cámara simulada. Estima el error entre ambas  y finalmente genera la imagen correspondiente a los puntos de ambas cámaras.
*/
Mat drawProjectionPoints(){
	srand(time(NULL));

	Mat randomCamera = generateRandomCamera();
	vector<Point3f> pattern = pointPattern();
	Mat aux = Mat::zeros(512, 512, CV_32FC3);
	Size size = Size(512,512);
	vector<Point2f> projections = projectPoints(randomCamera, pattern);

	Mat camera = obtainMatrixCamera(projections, pattern);
	vector<Point2f> cameraProjections = projectPoints(camera, pattern);

	vector<Point2f> randomFinalPoints = resizePoints(projections, size);
	vector<Point2f> simulateFinalPoints = resizePoints(cameraProjections, size);

	float err = estimatedError(randomCamera, camera);
	cout << "El error entre la cámara aleatoria y estimada es de " << err << endl;

	Scalar color = Scalar(0,0,255);
	for (unsigned int i = 0; i < randomFinalPoints.size(); i++)
		circle(aux, randomFinalPoints[i], 4, color, -1, CV_AA);

	color = Scalar(0,255,0);
	for (unsigned int i = 0; i < simulateFinalPoints.size(); i++)
		circle(aux, simulateFinalPoints[i], 2, color, -1, CV_AA);

	return aux;
}
```

Lo primero que realizamos es la obtención de la cámara aleatoria; que para que sea correcta el determinante de la matriz de ésta ha de ser distinta de 0, esto se encuentra implementado en la función **generateRandomCamera()**:

```
/**
* Devuelve una matriz aleatoria 3x4 con determinante distinta de 0
*/
Mat generateRandomCamera(){
        cv::theRNG().state = time(NULL);
        Mat matrix = Mat(3, 4, CV_32FC1), aux;
   do{
        randu(matrix, Scalar::all(0), Scalar::all(255));
                aux = matrix(Rect(0, 0, 3, 3));
        }
   while(determinant(aux) == 0);
        return matrix;
}
```

Se crea la matriz cámara de tamaño 3x4 con valores aleatorios con la función **randu()** de OpenCV, se coge de ésta una submatriz 3x3 y se comprueba su determinante con **det()**. Si el determinante es igual a 0 se repite el proceso hasta que sea distinto de 0.

```
 /**
* Dada la matriz @matrix realiza el determinante de sus valores
*/
int det(Mat matrix){
        return round(determinant(matrix));
}
```

A continuación llamamos a **pointPattern()** que se encarga de generar los puntos 3D que más adelante necesitaremos:

```
/**
* Genera el patrón de puntos del mundo 3D correspondientes con 0:0.1:0.1 - 0.1:0.1:0
*/
vector<Point3f> pointPattern(){
        vector<Point3f> pattern;
        for (double k1= 0.1; k1 <= 1; k1 += 0.1) {
          for (double k2 = 0.1; k2 <= 1; k2 += 0.1) {
            pattern.push_back(Point3f( 0, k1, k2));
           pattern.push_back(Point3f(k2, k1, 0));
        }
     }
     return pattern;
}
```

Los puntos generados corresponden al compuesto por el conjunto de puntos con coordenadas {(0,x1,x2) y (x2,x1,0), para x1=0.1:0.1:1 y x2=0.1:0.1:1}.

Teniendo la cámara aleatoria *p* y el conjunto de puntos 3D *pattern* ya podemos proyectarlos en 2D con la función **projectPoints()**:

```
/**
* Dada la matriz cámara @P y el vector de los puntos 3D del mundo @points obtener los puntos 2D resultantes de multiplicar los puntos 3D por la matriz de la cámara
*/
vector<Point2f> projectPoints(Mat P, vector<Point3f> points) {
	vector<Mat> projection;
	vector<Point2f> ppoints;

	vector<Mat> homogeneousPoints;
	for(unsigned i=0; i<points.size(); i++){
		homogeneousPoints.push_back(toHomogeneous(points[i]));
	}

	for(unsigned i=0; i<homogeneousPoints.size(); i++){
		projection.push_back(P*homogeneousPoints[i]);
	}

	for(unsigned i=0; i<projection.size(); i++){
		ppoints.push_back(to2Dpoint(projection[i]));
	}
	return ppoints;
}
```

Pasa los puntos a coordenadas homogéneas, multiplica la cámara por éstos, y el resultado lo pasa a puntos en 2D que son los que finalmente devuelve. Para pasar a coordenadas homogéneas y a 2D se han usado las funciones auxiliares **toHomogeneous()** y **to2Dpoint()**, podremos verlas en la última sección del documento.

Como queremos mostrar en una imagen los puntos 2D es necesario que los valores de éstos sean los suficientemente grandes como para que puedan verse y distinguirse con claridad, por tanto estos puntos se pasan como parámetro a la función **resizePoints()**:

```
/**
* Los puntos @points son puntos 2D resultantes de la proyección en coordenadas del mundos y @rows y @cols las filas y columnas respectivas a la imagen en la que se va a pintar. Aquí se obtendrán los puntos en una "escala" mayor para que puedan verse correctamente al pintarlos.
*/
vector<Point2f> resizePoints(vector<Point2f> points, Size size){
	vector<Point2f> pixelProjection;
	Point2f pixelPoint;
	for(unsigned i=0; i<points.size(); i++){
		pixelPoint = Point2f((size.width/2)*points[i].x, (size.height/2)*points[i].y);
		pixelProjection.push_back(pixelPoint);
	}
```

```
	return pixelProjection;
}
```

Aumenta el valor del punto en función del tamaño de la imagen para que pueda verse posteriormente en la imagen sobre la que se pintarán.

Ya tenemos los puntos de la cámara aleatoria, faltan los de la cámara simulada. Para ello llamamos a **obtainMatrixCamera()** que devuelve la cámara simulada, y realizamos lo mismo que con la aleatoria. Proyectamos los puntos 3D para pasar a 2D y los aumentamos para que se vean correctamente.

```
/**
* Dados los puntos 2D 3D obtener la matriz de coeficientes y mediante la descomposición en valores singulares, tras generar Vt (V traspuesta) con la función compute obtener la última fila y almacenarlo en la matriz de la cámara.
*/
Mat obtainMatrixCamera(vector<Point2f> points2d, vector<Point3f> points3d){
	Mat coefs = estimateMatrix(points2d, points3d);
	Mat camera = Mat(3, 4, CV_32FC1);
	Mat w, u, vt;

	SVD::compute(coefs, w, u, vt);

	for(int i=0; i<12; i++){
		camera.at<float>(Point(i%4, floor(i/4))) = vt.at<float>(Point(i, vt.rows-1));
	}
	return camera;
}
```

Esta función llama a **estimateMatrix()** para obtener la matriz de coeficientes dependiendo de los puntos 2D y 3D para posteriormente realizar la descomposición en valores singulares y generar la matriz Vt (la traspuesta de V), donde rellenaremos la cámara con los valores de la última fila de Vt.

```
/**
* Una vez que tenemos los puntos en correspondencias 2D 3D obtenemos la matriz de coeficientes.
*/
Mat estimateMatrix(vector<Point2f> points2d, vector<Point3f> points3d){
	Mat aux = Mat::zeros(points2d.size()*2, 12, CV_32FC1);

	for(unsigned i=0; i<points2d.size(); i++){
		aux.at<float>(Point(0,2*i)) = aux.at<float>(Point(4,2*i+1)) = points3d[i].x;
		aux.at<float>(Point(1,2*i)) = aux.at<float>(Point(5,2*i+1)) = points3d[i].y;
		aux.at<float>(Point(2,2*i)) = aux.at<float>(Point(6,2*i+1)) = points3d[i].z;
		aux.at<float>(Point(3,2*i)) = aux.at<float>(Point(7,2*i+1)) = 1;
```

```
            aux.at<float>(Point(8,2*i)) = -points2d[i].x*points3d[i].x;
            aux.at<float>(Point(9,2*i)) = -points2d[i].x*points3d[i].y;
            aux.at<float>(Point(10,2*i)) = -points2d[i].x*points3d[i].z;
            aux.at<float>(Point(11,2*i)) = -points2d[i].x;

            aux.at<float>(Point(8,2*i+1)) = -points2d[i].y*points3d[i].x;
            aux.at<float>(Point(9,2*i+1)) = -points2d[i].y*points3d[i].y;
            aux.at<float>(Point(10,2*i+1)) = -points2d[i].y*points3d[i].z;
            aux.at<float>(Point(11,2*i+1)) = -points2d[i].y;
        }
        return aux;
}
```

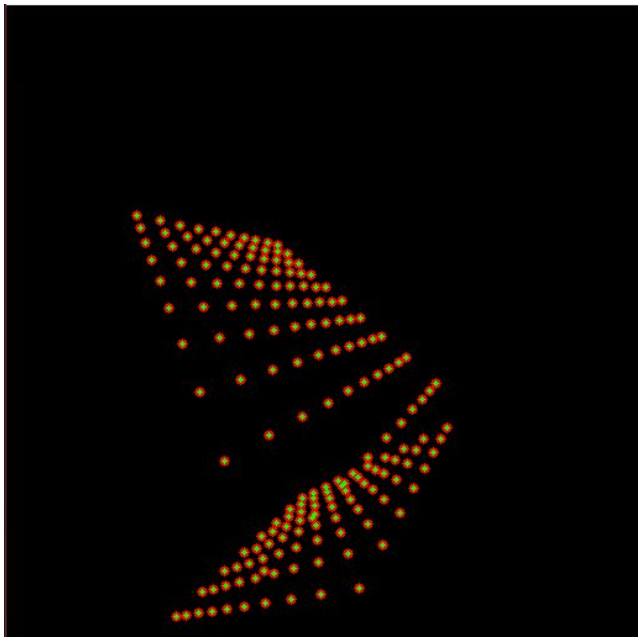
En esta función es donde se calcula la matriz de coeficientes de los puntos en correspondencias 2D y 3D.

Hasta aquí ya tenemos los puntos de la cámara aleatoria y de la cámara simulada por lo que pintamos con la función **circle()** de OpenCV cada uno de los puntos de ambas con colores diferentes.

También vamos a calcular el error estimado (norma de Frobenius) con la función **estimatedError()**:

```
/**
* Estima el error usando la norma de Frobenius dada la cámara aleatoría y la cámara simulada.
*/
float estimatedError(Mat p1, Mat p2){
        Mat normP1, normP2;

        normalizeMatrix(p1, normP1);
        normalizeMatrix(p2, normP2);

        float inc = 0;

        for(int i=0; i<3; i++)
            for(int j=0; j<4; j++)
                inc += pow(normP1.at<float>(Point(j,i)) - normP2.at<float>(Point(j,i)), 2);

        return sqrt(inc);
}
```

Una de las imágenes resultantes es la siguiente:



Puntos de la cámara aleatoria (rojo) y simulada (verde). Error cometido: 1.577715e-05

El error obtenido es muy bajo, éste puede deberse al cálculo o al rendondeo, no obstante es un buen resultado.

Calibración de la cámara usando homografías

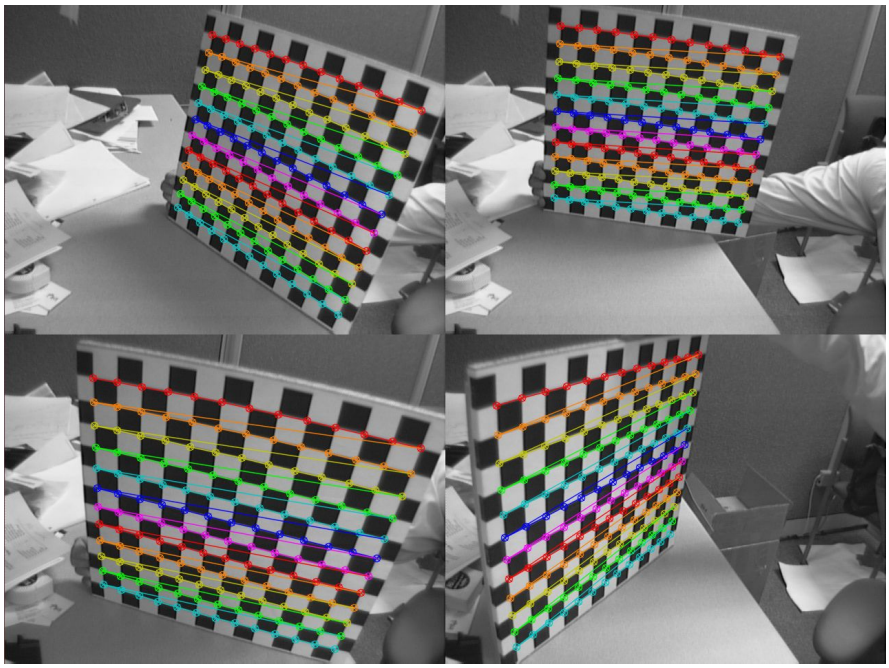
Este ejercicio pide primero obtener y pintar las imágenes válidas para calibrar una cámara con sus puntos estimados y segundo calcular los parámetros intrínsecos y extrínsecos de la cámara para cada imagen.

Por lo que empezando por el primer punto tenemos la función **obtainCalibratePoints()**:

```
/**
* Dadas las imágenes @images y mediante la función findChessboardCorners(), busca aquellas
que son válidas para calibrar una cámara. El tamaño del tablero que se ha seleccionado es de
13x12. Va guardando los puntos 2D que se van obteniendo de la función mencionada
anteriormente y también va generando los puntos 3D mediante el patrón elegido.
*/
vector<Mat> obtainCalibratedPoints(vector<Mat> images, vector<vector<Point2f> > &points,
vector<vector<Point3f> > &validWorldPoints){
        vector<Mat> validImages;
        vector<Point2f> corners;
        Size size = Size(13,12);
        Size winSize = Size(5, 5), zeroZone = Size(-1, -1);
        TermCriteria criteria = TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1);
```

```
	for (unsigned i = 0; i < images.size(); i++) {
		Mat img_gray, aux;
		cvtColor(images[i], img_gray, COLOR_BGR2GRAY);
		if(findChessboardCorners(img_gray, size, corners)){
			img_gray.convertTo(img_gray, CV_32F);
			cornerSubPix(img_gray, corners, winSize, zeroZone, criteria);
			points.push_back(corners);
			images[i].copyTo(aux);
			drawChessboardCorners(aux, size, Mat(corners), 1);
			validImages.push_back(aux);

			vector<Point3f> worldPoints;
			for (unsigned i = 0; i < corners.size(); i++) {
				Point3f point = Point3f(i%size.width*100, i/size.width*100, 0);
				worldPoints.push_back(point);
			}
			validWorldPoints.push_back(worldPoints);
		}
	}
	return validImages;
}
```

Esta función comprueba para cada imagen de las que le pasamos como parámetros si es válida para calibrar una cámara, con **findChessboardCorners()**, de ser así la guarda con los puntos estimados pintados mediante **drawChessboardCorners()**. También va almacenando en una estructura de datos los puntos 2D y 3D. De las imágenes de los tableros el programa ha determinado que solo 4 cumplen para calibrar una cámara, que son:



Imágenes válidas para calibrar una cámara con los puntos señalados

En el segundo apartado del ejercicio es donde queremos obtener tanto el error con los diferentes tipos de distorsión, la cámara y movimientos que tienen las imágenes válidas. Todo esto se encuentra en la función **myCalibrateCamera().**

```
/**
* Con las imágenes válidas para calibrar la cámara se pretende obtener el error con/sin distorsión
y sus diferentes parámetros así como la matriz cámara y el vector de rotaciones y traslaciones.
*/
double myCalibrateCamera(vector<Mat> images, Mat &camera, vector<Mat> &rotVec,
vector<Mat> &trasVec, int type = NO_DIST){
	vector<vector<Point2f> > points;
	vector<vector<Point3f> > worldPoints;
	vector<Mat> validImages = obtainCalibratedPoints(images, points, worldPoints);
	Size size = Size(validImages[0].cols, validImages[0].rows);
	camera = Mat(3, 3, CV_32F);
	Mat coefs = Mat(8, 1, CV_32F);
	double error;

	switch(type){
		case NO_DIST:
			error = calibrateCamera(worldPoints, points, size, camera, coefs, rotVec,
trasVec, CV_CALIB_ZERO_TANGENT_DIST | CV_CALIB_FIX_K1 | CV_CALIB_FIX_K2 |
CV_CALIB_FIX_K3 | CV_CALIB_FIX_K4 | CV_CALIB_FIX_K5 | CV_CALIB_FIX_K6);
		break;
		case RAD_DIST:
			error = calibrateCamera(worldPoints, points, size, camera, coefs, rotVec,
trasVec, CV_CALIB_ZERO_TANGENT_DIST | CV_CALIB_RATIONAL_MODEL);
		break;
```

```
                case TANG_DIST:
                        error = calibrateCamera(worldPoints, points, size, camera, coefs, rotVec,
trasVec, CV_CALIB_FIX_K1 | CV_CALIB_FIX_K2 | CV_CALIB_FIX_K3 | CV_CALIB_FIX_K4 |
CV_CALIB_FIX_K5 | CV_CALIB_FIX_K6);
                break;
                case TANG_AND_RAD_DIST:
                        error = calibrateCamera(worldPoints, points, size, camera, coefs, rotVec,
trasVec, CV_CALIB_RATIONAL_MODEL);
                break;
        }

        return error;
}
```

Llamamos primero a la función del primer apartado **obtainCalibratedPoints()** para obtener los puntos 2D y 3D para pasarlos como parámetros en la función de OpenCV **calibrateCamera()** que nos devuelve el error dependiendo del tipo de distorsión. También devuelve la cámara, la matriz de rotación y de traslación.

Los errores obtenidos con los diferentes tipos de distorsión han sido los siguientes:
El error sin distorsión es de 1.3259
El error con distorsión radial es de 0.162782
El error con distorsión tangencial es de 1.30202
El error con distorsión tangencial y radial es de 0.161846

Como podemos ver, si tuviésemos que quedarnos con sólo un tipo de distorsión sería con el radial al tener el menor error con diferencia, 0.162782, sin embargo la mejor opción es con distorsión tangencial y radial 0.161846 aunque no supera al tangencial por mucha diferencia.

Nota: En el código también pueden verse los parámetros intrínsecos y extrínsecos para cada imagen, al realizarse en 4 imágenes y 4 tipos de errores, se ha comentado dicha parte del código para evitar la saturación de la salida del programa. De querer verlos simplemente habría que descomentar dichas líneas de código (se encuentra al final del ejercicio).

Estimación de la matriz fundamental F

En este ejercicio se pide estimar la matriz fundamental F y dibujar las líneas epipolares. Para ello es necesario usar un detector como los de la práctica anterior. Se llama a la función **myDetector()** ya implementada, la cual se ha ampliado:

```
----------------------------------------
else if(type == TYPE_ORB){
		Ptr<ORB> detector = ORB::create(600, 1.2f, 2, 100);
		detector->detectAndCompute(img1_gray, noArray(), keyP1, descriptors1);
		detector->detectAndCompute(img2_gray, noArray(), keyP2, descriptors2);
		BFMatcher matcher(NORM_HAMMING, true);
		matcher.match(descriptors1, descriptors2, matches);

	}
else if(type == TYPE_BRISK){
		Ptr<BRISK> detector = BRISK::create(10, 0);
		detector->detectAndCompute(img1_gray, noArray(), keyP1, descriptors1);
		detector->detectAndCompute(img2_gray, noArray(), keyP2, descriptors2);
		BFMatcher matcher(NORM_HAMMING, true);
		matcher.match(descriptors1, descriptors2, matches);
	}
----------------------------------------
```

Se han añadido los dos nuevos tipos y generado las correspondencias entre ambas con BruteForce + Crosscheck.

```
----------------------------------------
	if(mode == ALL || mode == ONLY_EPIPOLAR || mode == ONLY_FUNDAMENTAL_MAT){
		vector<Mat> epiImages = epipolarPoints(img1, img2, points1, points2);
		if(mode == ONLY_EPIPOLAR){
			Mat auxImg = epiImages[0];
			hconcat(auxImg, epiImages[1], auxImg);
			result.push_back(auxImg);
		}
		else if(mode == ONLY_FUNDAMENTAL_MAT)
			result.push_back(epiImages[2]);

	}
----------------------------------------
```

Se han añadido nuevos modos para operar en función de lo que se desee, si lo que queremos es obtener las líneas epipolares o la matriz fundamental F. Básicamente la función devolverá las imágenes correspondientes a lo que le digamos en la variable *mode*.

Como hemos visto en la función anterior si queremos obtener las líneas epipolares se llamará a una función **epipolarPoints()**:

```
/**
* Dibuja en las imágenes correspondientes las líneas epipolares de cada imagen. Para ello Se
obtiene la matriz F mediante la función de OpenCV findFundamentalMat() con 8 puntos y RANSAC.
*/
vector<Mat> epipolarPoints(Mat img1, Mat img2, vector<Point2f> points1, vector<Point2f>
points2){
	vector<Mat> epiImages;
	vector<Point2f> fPoints1, fPoints2;
	vector<Point3f> lines1, lines2;
```

```
	vector<uchar> mask;
	RNG rng(12345);
	Mat image1, image2;
	img1.copyTo(image1);
	img2.copyTo(image2);

	Mat F = findFundamentalMat(points1, points2, CV_FM_8POINT | CV_FM_RANSAC, 0.3,
0.95, mask);
	fPoints1 = fundamentalMatPoints(points1, mask);
	fPoints2 = fundamentalMatPoints(points2, mask);

	computeCorrespondEpilines(fPoints1, 1, F, lines2);
	computeCorrespondEpilines(fPoints2, 2, F, lines1);

	for(unsigned i = 0; i < lines1.size(); i++){
		Scalar color = Scalar(rng.uniform(0,255), rng.uniform(0, 255), rng.uniform(0,
255));
		Point2f p1, p2;
		p1 = Point2f(image1.cols, (-lines1[i].z-lines1[i].x*image1.cols)/lines1[i].y);
		p2 = Point2f(0, (-lines1[i].z)/lines1[i].y);
		line(image1, p1, p2, color);
		circle(image1, fPoints1[i], 3, color, -1, CV_AA);
	}
	for(unsigned i = 0; i < lines2.size(); i++){
		Scalar color = Scalar(rng.uniform(0,255), rng.uniform(0, 255), rng.uniform(0,
255));
		Point2f p1, p2;
		p1 = Point2f(image2.cols, (-lines2[i].z-lines2[i].x*image2.cols)/lines2[i].y);
		p2 = Point2f(0, (-lines2[i].z)/lines2[i].y);
		line(image2, p1, p2, color);
		circle(image2, fPoints2[i], 3, color, -1, CV_AA);
	}
	epiImages.push_back(image1);
	epiImages.push_back(image2);
	epiImages.push_back(F);

	float err1 = epipolarError(fPoints1, lines1);
	float err2 = epipolarError(fPoints2, lines2);

	cout << "El error de la primera imagen es de " << err1 << endl;
	cout << "El error de la segunda imagen es de " << err2 << endl;

	return epiImages;
}
```

Primero se calcula la matriz fundamental F con 8 puntos y RANSAC con **findFundamentalMat().** Esta función devuelve un vector, *mask*, un vector con valores 0 ó 1

que se encarga de decir qué puntos en correspondencia van a ser los válidos. Esto último se realizará en la función **fundamentalMatPoints()**:

```
/**
* Obtención de los puntos válidos, es decir, aquellos en los que la máscara o estado @mask sea 1 además de que el tamaño de este nuevo vector de puntos ha de ser menor de 200.
*/
vector<Point2f> fundamentalMatPoints(vector<Point2f> points, vector<uchar> mask){
	vector<Point2f> p;
	int cnt=0;
	for (unsigned i = 0; i < points.size(); i++){
		if(mask[i] == 1 && cnt<200){
			p.push_back(points[i]); cnt++;
		}
	}
	return p;
}
```

Esta función solo escoge los puntos donde *mask* tiene valor 1; también limita a coger como mucho 200 puntos.

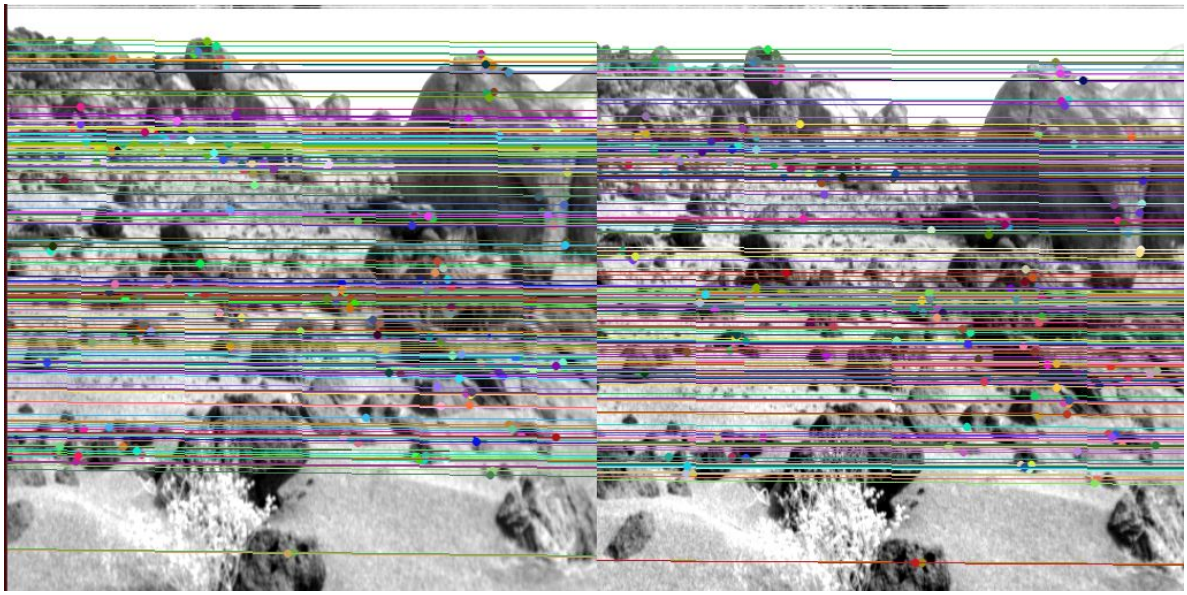
Ya tenemos los puntos que nos van a servir, por lo que llamamos a **computeCorrespondEpilines()** donde le diremos que halle las líneas correspondientes a los puntos contrarios con la matriz F. Esto se realiza dos veces, una vez para cada vector de puntos.

Ahora se recorren los nuevos vectores de las líneas y se obtienen los dos puntos de los extremos por donde pasa para poder pintar dicha línea así como el propio punto.
Para saber el error se ha implementado la función **epipolarError():**

```
/**
* Error obtenido como la media de la didstancia ortogonal entre los puntos soporte y sus líneas epipolares.
*/
float epipolarError(vector<Point2f> points, vector<Point3f> lines){
	float inc = 0;
	for(unsigned i = 0; i < points.size(); i++){
		inc += (fabs(lines[i].x*points[i].x+lines[i].y*points[i].y+lines[i].z))/sqrt(lines[i].x*lines[i].x+lines[i].y*lines[i].y);
	}
	return inc / points.size();

}
```

Esta función equivale a $\frac{Ax+By+C}{\sqrt{A^2+B^2}}$.

Las líneas epipolares para las imágenes Vmort con el detector BRISK y el error de ambas son las siguientes:



Líneas epipolares con BRISK y error de 0.12503 y 0.125318 correspondiente en cada imagen

Que las líneas se vean de esta forma puede deberse a que la diferencia entre la captura de ambas imágenes sea un movimiento de traslación en un eje y otro de rotación en otro eje, de forma que el epipolo se nos iría al infinito.

Calcular movimientos de la cámara (R,t)

Para obtener los movimientos de la cámara primero tenemos que obtener la matriz esencial, esto se hacen en la función **essentialMat()**:

```
/**
* Obtiene la matriz esencial resultante de dos imágenes @img1 e @img2 así como los vectores de las correspondencias.
*/
vector<Mat> essentialMat(Mat img1, Mat img2, vector<Point2f> &points1, vector<Point2f> &points2, Mat K){
	vector<Mat> resultMats;
	Mat essential;

	resultMats.push_back(myDetector(img1, img2, -1, points1, points2, ONLY_FUNDAMENTAL_MAT, TYPE_BRISK)[0]);

	essential = K.t() * resultMats[0] * K;

	resultMats.push_back(essential);

	return resultMats;
}
```

Llamamos al detector de correspondencias y obtenemos los puntos en correspondencias así como la matriz fundamental F. Cómo sabemos los parámetros intrínsecos de la cámara los guardamos en la variable *K* por lo que para obtener la matriz esencial ya solo tenemos que multiplicar *Kt*F*K*.

Ahora queremos obtener los parámetros extrínsecos, para ello llamaremos a la función **obtainMovements()**:

```
/**
* Obtiene los movimientos de Rotación y traslación de la matriz esencial.
*/
void obtainMovements(Mat &essential, vector<Point2f> &points1, vector<Point2f> &points2, Mat &t, Mat &R, float focalDistance1, float focalDistance2){
	Mat positiveTras = obtainTraslationVector(essential);
	Mat negativeTras = -positiveTras;

	Mat fRot = obtainRotationVec(essential, positiveTras);
	Mat sRot = obtainRotationVec(-1*essential, positiveTras);
	Vec4i cntDepth;

	for(unsigned i = 0; i < 4; i++ ){
		Mat auxTras, auxRot;
		if(i<2) auxTras = positiveTras;
		else auxTras = negativeTras;
		if(i%2==0) auxRot = fRot;
		else auxRot = sRot;

		cntDepth[i] = negativeDepth(points1, points2, auxTras, auxRot, focalDistance1, focalDistance2);
	}
	int minNegDepthPos;
	double minNegDepth;

	minMaxIdx(cntDepth, &minNegDepth, NULL, &minNegDepthPos);

	switch(minNegDepthPos)
	{
		case 0:
			t = positiveTras; R = fRot;
		break;
		case 1:
			t = positiveTras; R = sRot;
		break;
		case 2:
			t = negativeTras; R = fRot;
		break;
		case 3:
			t = negativeTras; R = sRot;
```

```
                break;
        }
}
```

Obtiene los dos tipos posibles de traslación y rotación con las funciones auxiliares **obtainRotationVec()** y **obtainTraslationVector()**. Se ejecuta un bucle donde se alternan las 4 posibilidades de los tipos mencionados anteriormente y se almacena en un vector el resultado de la función **negativeDepth()**:

```
/**
* Genera la profundidad negativa dependiendo de los puntos y los movimientos de rotación @R y traslación @t y distancias focales.
*/
int negativeDepth(vector<Point2f> &points1, vector<Point2f> &points2, Mat &t, Mat &R, float focalDistance1, float focalDistance2){
        int cnt = 0;

        for(unsigned i = 0; i < points1.size(); i++ ){
                Mat h = Mat(1, 3, CV_64F);
                h.at<double>(0) = points1[i].x;
                h.at<double>(1) = points1[i].y;
                h.at<double>(2) = 0;

                double ldepth = focalDistance2*(focalDistance1*t.dot(R.row(0)-points2[i].x*R.row(2)))/(focalDistance2*h.dot(R.row(0)-points2[i].x*R.row(2)));

                double rdepth = R.row(2).dot(focalDistance1/ldepth*h-t);

                if(ldepth < 0 || rdepth < 0) cnt++;

        }
        return cnt;
}
```

Obtiene los dos tipos de profundidad y cuenta de todos ellos cuales son negativos.

Terminado el bucle se llama a **minMaxIdx()** de OpenCV que busca el mínimo y máximo global de un vector, para este caso queremos el mínimo que se guarda en la variable *minNegDepthPos*. Una vez que lo tenemos y dependiendo de su valor seleccionaremos los diferentes tipos de rotación y traslación.

Los movimientos de rotación y traslación obtenidos de cada pareja de imágenes son los siguientes:

Para la pareja 1-2 la matriz de rotación es de:
[-0.9578316053244599, 0.2386898384283928, 0.01032744279917457;
0.2383150861996525, 0.9643630306872819, 0.08023890576321635;

0.0188182891357252, 0.07921157029705828, -1.006665376969569]
El vector de traslación:
[0.1198603874505159, 0.9917941536331922, 0.04023831791432026]

Para la pareja 1-3 la matriz de rotación es de:
[0.8934459237993218, 0.09907634834503209, 0.08395509986814928;
-0.08061034015185103, 0.8817513282391494, 0.03448867541579453;
-0.0009595981228922301, 0.009345611399474374, 0.9729791637516458]
El vector de traslación:
[-0.4199042546106034, -0.7888741977943784, 0.3739725395876326]

Para la pareja 2-3 la matriz de rotación es de:
[1.046562052999124, -0.0574503005548573, -0.02728691585716381;
-0.1685033756589876, 0.8989674168361804, -0.1337331441937227;
0.01630544600725836, 0.0746048947758933, 0.9138804684932416]
El vector de traslación:
[-0.1862554535442384, -0.1916564167540604, 0.9481983182883257]

Funciones auxiliares

A continuación se muestran las funciones auxiliares usadas:

```
/**
* Muestra las imágenes que se encuentran en el vector @images en una ventana llamada
* como se establece en @window. Para cambiar de imagen hay que pulsar cualquier tecla.
*/
void showImages(vector<Mat> images, string window = "Ventana"){
	for(unsigned i=0; i<images.size(); i++){
		imshow(window, images[i]);
		waitKey(0);
	}
}


/**
* Devuelve el tipo de imagen con el string correspondiente para que sea entendible.
*/
string type2str(int type) {
  string r;

  uchar depth = type & CV_MAT_DEPTH_MASK;
  uchar chans = 1 + (type >> CV_CN_SHIFT);

  switch ( depth ) {
```

```
    case CV_8U:  r = "8U"; break;
    case CV_8S:  r = "8S"; break;
    case CV_16U: r = "16U"; break;
    case CV_16S: r = "16S"; break;
    case CV_32S: r = "32S"; break;
    case CV_32F: r = "32F"; break;
    case CV_64F: r = "64F"; break;
    default:     r = "User"; break;
  }

  r += "C";
  r += (chans+'0');

  return r;
}

/**
* Dado un punto pasarlo a coordenadas homogéneas
*/
Mat toHomogeneous(Point3f point){
        Mat p = Mat(4,1,CV_32FC1);
        p.at<float>(Point(0,0)) = point.x; p.at<float>(Point(0,1)) = point.y;
        p.at<float>(Point(0,2)) = point.z; p.at<float>(Point(0,3)) = 1;
        return p;
}

/**
* Dado un punto pasarlo a 2D
*/
Point2f to2Dpoint(Mat point){
        Point2f aux;
        float z = point.at<float>(Point(0,2));
        aux.x = point.at<float>(Point(0,0))/z;
        aux.y = point.at<float>(Point(0,1))/z;
        return aux;
}

/**
* Devuelve la matriz resultante del movimiento traslación de la matriz esencial
*/
Mat obtainTraslationVector(Mat essential){
        Mat enorm, tnorm, eye, tras;
        eye = Mat::eye(3, 3, CV_64F);
        tras = Mat(1, 3, CV_64F);
        int max = 0;
```

```
	enorm = essential*essential.t()/(trace(essential*essential.t())[0]/2);
	tnorm = -enorm + eye;

	if(tnorm.at<double>(Point2f(1,1)) > tnorm.at<double>(Point2f(0,0))) max = 1;
	if(tnorm.at<double>(Point2f(2,2)) > tnorm.at<double>(Point2f(max,max))) max = 2;

	tras = (1/sqrt(tnorm.at<double>(Point2f(max,max))))*tnorm.row(max);

	return tras;
}

/**
* Devuelve la matriz resultante del movimiento rotación de la matriz esencial y el vector @v
*/
Mat obtainRotationVec(Mat essential, Mat v){
	Mat rotation, w[3], enorm;
	rotation = Mat(3,3,CV_64F);
	enorm = essential/sqrt((trace(essential*essential.t())[0]/2));

	for(unsigned i = 0; i < 3; i++ ){
		Mat row = enorm.row(i);
		w[i] = row.cross(v);
	}

	for(unsigned i = 0; i < 3; i++ )
		rotation.row(i) = w[i] + w[(i+1)%3].cross(w[(i+2)%3]);

	return rotation;
}
```