Informe de prácticas
3

Alejandro García Montoro
agarciamontoro@correo.ugr.es

13 de enero de 2016

Introducción

Al igual que al principio de las prácticas se definió la clase *Image* para agrupar todos los métodos relativos a las imágenes, en esta última práctica se ha añadido una segunda clase: *Camera*. La definición de la clase, con sus atributos y métodos, es la siguiente:

```
1   class Camera{
2       private:
3       Mat camera;
4       bool isFinite();
5
6       public:
7
8       Camera();
9       Camera( vector< pair<Vec3f, Vec2f> > matches );
10
11      void randomFinite(float min = 0.0, float max = 1.0);
12      Vec2f projectPoint(Vec3f point);
13
14      void printCamera();
15      float error(const Camera& other);
16  };
```

La práctica sigue, por tanto, una estructura dirigida a objetos cuyo núcleo está ahora, además de en *Image*, en *Camera*.

1. Estimación de la matriz de una cámara

1.1. Generación aleatoria de una cámara finita

Para la generación aleatoria de una cámara finita, se ha implementado la siguiente función, que recibe dos valores reales —que definen el intervalo en el que estarán los elementos de la matriz— y devuelve una matriz de dimensión 3×4 que es finita; esto es, cuya submatriz 3×3 derecha tiene determinante distinto de cero:

```
1    void Camera::randomFinite(float min, float max){
2        do{
3            theRNG().state = clock();
4            randu(this->camera, Scalar::all(min), Scalar::all(max));
5        }while( !this->isFinite() );
6    }
```

La primera línea simplemente actualiza el generador aleatorio con el estado del reloj para tener mayor aleatoriedad.

La segunda línea es una llamada a *randu()*, una función de *OpenCV* que rellena todos los valores de una matriz con valores aleatorios en un rango dado.

Para ver que es finita, se llama a la función propia *isFinite()*, cuyo código es el siguiente:

```
1    bool Camera::isFinite(){
2        Mat sub = this->camera(Rect(0,0,3,3));
3        return determinant(sub) != 0.0;
4    }
```

Así, la función *randomFinite()* genera indefinidamente matrices aleatorias hasta que encuentra una que es finita.

1.2. Generación de puntos y proyección

La generación de puntos se hace siguiendo el patrón descrito en el guión de prácticas con el primer bucle doble anidado que aparece en el siguiente código:

```
1    // Generation of 3D points
2    vector<Vec3f> points;
3    for (double x1 = 0.1; x1 <= 1.0; x1 += 0.1) {
4        for (double x2 = 0.1; x2 <= 1.0; x2 += 0.1) {
5            points.push_back(Vec3f(0,x1,x2));
6            points.push_back(Vec3f(x2,x1,0));
7        }
8    }
9
```

```
10      // Projection of points and extraction of minimum and maximum
            coordinates
11      // (for visual purposes only)
12      vector<pair<Vec3f, Vec2f> > projected_points;
13
14      float max_x, max_y, min_x, min_y;
15
16      max_x = max_y = -9999999;
17      min_x = min_y = +9999999;
18
19      for (size_t i = 0; i < points.size(); i++) {
20          Vec3f point = points[i];
21          Vec2f projected_point = simulated.projectPoint(point);
22
23          pair<Vec3f, Vec2f> match(point, projected_point);
24          projected_points.push_back(match);
25
26          updateMinMax(min_x, max_x, min_y, max_y, projected_point);
27      }
```

En el último bucle vemos cómo se proyectan todos los puntos generados con la cámara simulada, guardando en un vector las parejas *punto escena—punto-proyectado* para después poder estimar la cámara. Además, la línea X actualiza unas variables en las que se almacenarán los valores mínimos y máximos de las coordenadas de todas las proyecciones, para después visualizar los puntos de forma óptima en una imagen.

La función importante de este bloque de código es la de la línea Y, cuyo código es el siguiente:

```
1       Vec2f Camera::projectPoint(Vec3f point){
2           float hom_vector[4] = {point[0], point[1], point[2], 1.0};
3           Mat hom_point = Mat(4, 1, CV_32F, hom_vector);
4
5           Mat projection = this->camera * hom_point;
6
7           float p_x = projection.at<float>(0)/projection.at<float>(2);
8           float p_y = projection.at<float>(1)/projection.at<float>(2);
9
10          return Vec2f(p_x, p_y);
11      }
```

El código implementa la proyección usual: el producto de la cámara por el punto escena con coordenadas homogéneas. Para retomar las coordenadas píxel de la proyección, basta dividir todas las coordenadas del resultado por la tercera y quedarse las dos primeras.

1.3. Algoritmo DLT

Tras generar todas las proyecciones de los puntos escena, esta información se usa para estimar la cámara —que debe ser igual a la simulada—. Además, se reproyectan los puntos escena con la cámara estimada para actualizar las variables de coordenads mínimas y máximas:

```
    // Generation of an estimated camera with 3D-2D points pairs
    Camera estimated(projected_points);

    // Update of the minimum and maximum coordinates with the
        estimated camera
    // (in order to update the [min-max]_[x-y] values)
    for (size_t i = 0; i < points.size(); i++) {
        Vec3f point = points[i];
        Vec2f projected_point = estimated.projectPoint(point);

        updateMinMax(min_x, max_x, min_y, max_y, projected_point);
    }
```

El núcleo de este bloque de código es el constructor de la cámara a partir de un vector de parejas *punto escena—-punto-proyectado*. El código de este constructor es el siguiente:

```
    Camera::Camera( vector< pair<Vec3f, Vec2f> > matches ){
        // Build the equations system.
        Mat mat_system, sing_values, l_sing_vectors, r_sing_vectors;

        for (unsigned int i = 0; i < matches.size(); i++) {
            Vec3f pt_3D = matches[i].first;
            Vec2f pt_2D = matches[i].second;

            float coeffs[2][12] = {
                {0,0,0,0, -pt_3D[0], -pt_3D[1], -pt_3D[2], -1,
                    pt_2D[1]*pt_3D[0], pt_2D[1]*pt_3D[1],
                    pt_2D[1]*pt_3D[2], pt_2D[1]},
                {pt_3D[0], pt_3D[1], pt_3D[2], 1, 0,0,0,0,
                    -pt_2D[0]*pt_3D[0], -pt_2D[0]*pt_3D[1],
                    -pt_2D[0]*pt_3D[2], -pt_2D[0]}
            };

            mat_system.push_back( Mat(2, 12, CV_32F, coeffs) );
        }

        // Solve the equations system using SVD decomposition
        SVD::compute( mat_system, sing_values, l_sing_vectors,
            r_sing_vectors );

        Mat last_row = r_sing_vectors.row(r_sing_vectors.rows-1);
```

```
21
22          this->camera = last_row.reshape(1,3);
23      }
```

Esta función simplemente genera, para cada par de puntos, las dos filas siguientes:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & -X & -Y & -Z & -1 & yX & yY & yZ & y \\ X & Y & Z & 1 & 0 & 0 & 0 & 0 & -xX & -xY & -xZ & -x \end{pmatrix}$$

donde $[X, Y, Z, 1]$ es el punto escena y $[x, y, 1]$, el punto proyectado.

Estas parejas de filas se apilan formando una matriz a la que se la aplica una descomposición SVD. Como en la práctica anterior, a la última fila de V se le da la forma correcta, 3×4, constituyendo así la matriz cámara que queríamos estimar.

1.4. Cálculo del error de la estimación

Para el cálculo del error de la estimación se ha recurrido a la función *norm* de *OpenCV*, que con el *flag NORM_L2SQR*, hace exactamente lo que queremos: la norma de Frobenius al cuadrado de la diferencia de dos matrices. La función que implementa esto es sencilla:

```
1       float Camera::error(const Camera& other){
2           float norm_this = 0, norm_other = 0;
3           bool found = false;
4
5           for (size_t i = 1; i < 4 && !found; i++) {
6               float current_this = this->camera.at<float>(0,i);
7               float current_other = other.camera.at<float>(0,i);
8               if(current_this != 0 && current_other != 0){
9                   norm_this = current_this;
10                  norm_other = current_other;
11                  found = true;
12              }
13          }
14
15          assert(norm_this != 0 && norm_other != 0);
16
17          Mat cam_1 = this->camera / norm_this;
18          Mat cam_2 = other.camera / norm_other;
19
20          return norm(cam_1, cam_2, NORM_L2SQR);
21      }
```

Como vemos, el núcleo de la función se encuentra en la sentencia *return*, que llama a la función *norm* de *OpenCV*. Sin embargo, antes de esto

normalizamos ambas cámaras para medir un error real.

Esto lo hacemos porque el algoritmo *DLT*, por su naturaleza, estima la matriz cámara salvo un factor escala. Así, si ponemos a 1 un elemento común de ambas matrices —en este caso el $a_{1,2}$, $a_{1,3}$ o $a_{1,4}$—, las normalizamos a una escala común, donde podemos medir el error con garantías de que será el real.

Es claro, por otro lado, que alguno de los elementos de la primera fila de la submatriz 3×3 derecha tiene que ser no cero; si no fuera así, el determinante de esta submatriz sería cero, así que estaríamos estimando una matriz que no es finita.

1.5. Generación de una imagen con los puntos

Es aquí donde usaremos las variables min_x, max_x, min_y, max_y, que definen los intervalos exactos en los que se mueven las coordenadas x e y de los píxeles en los que caen las proyecciones por ambas cámaras.

Para generar una imagen de $M \times M$ píxeles, por tanto, definiremos dos homeomorfismos de intervalos:

$$[min\_x, max\_x] \longmapsto [0, M]$$
$$[min\_y, max\_y] \longmapsto [0, M]$$

Así, para cada punto, aplicaremos estos homeomorfismos a ambas proyecciones para generar una imagen de $M \times M$ píxeles en la que mostrar todos los puntos.

El código que implementa toda esta funcionalidad es el siguiente:

```
1    // Drawing of both set of points in a single image, transforming
2    // [min_x, max_x] and [min_y, max_y] intervals into the
         [0,img_size]
3    // interval.
4    const int img_size = 400;
5    const float scale_x = img_size/(max_x-min_x);
6    const float scale_y = img_size/(max_y-min_y);
7
8    const Scalar red(0, 0, 255);
9    const Scalar green(0, 255, 0);
10
11   Mat canvas(img_size, img_size, CV_8UC3);
12
13   for (size_t i = 0; i < points.size(); i++) {
14       Vec3f point = points[i];
15       Vec2f simulated_point = simulated.projectPoint(point);
16       Vec2f estimated_point = estimated.projectPoint(point);
17
18       Point draw_simulated((simulated_point[0]-min_x)*scale_x,
             (simulated_point[1]-min_y)*scale_y);
```

```
19          Point draw_estimated((estimated_point[0]-min_x)*scale_x,
                (estimated_point[1]-min_y)*scale_y);
20
21          circle(canvas, draw_simulated, 3, green);
22          circle(canvas, draw_estimated, 1, red);
23      }
24
25      namedWindow( "Points", WINDOW_AUTOSIZE );
26      imshow( "Points", canvas );
27
28      waitKey(0);
29      destroyAllWindows();
```

Como se puede apreciar, dibujamos un círculo por cada proyección: en verde los proyectados por la cámara simulada y, en rojo, los proyectados por la estimada. Ponemos además radios diferentes para visualizar las dos proyecciones aunque recaigan sobre el mismo píxel.

1.6. Resultados

Los resultados de este apartado son los esperados: al tener 200 parejas de puntos escena y proyectados con la mayor exactitud posible, la estimación tiene que ser casi perfecta.

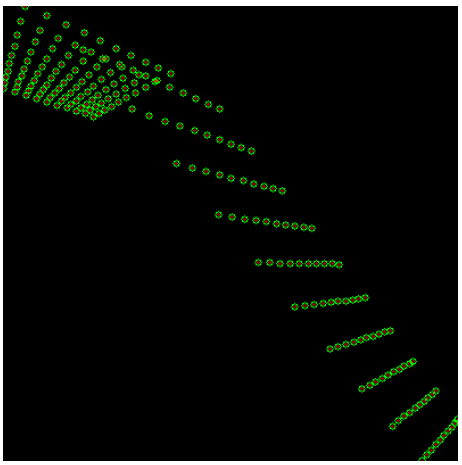

Figura 1: Error: $8{,}89067 \cdot 10^{-13}$

Y así es: los errores, diferentes en cada ejecución por la naturaleza aleatoria de la cámara simulada, son por lo general del orden de 10^{-12}, una cantidad debida únicamente a los errores de redondeo de la máquina, pues la descomposición SVD es matemáticamente correcta y las parejas de puntos son exactas.

Las figuras 1 y 2 corresponden a dos ejecuciones diferentes del código.

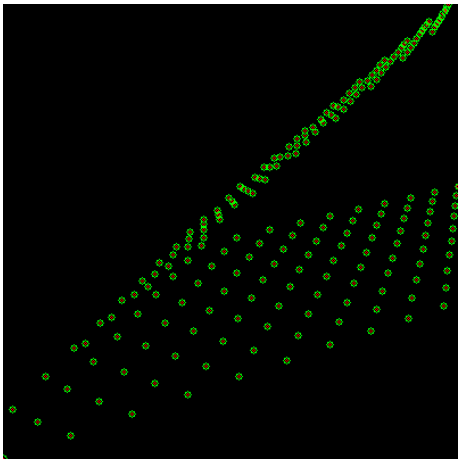

Figura 2: Error: $1{,}74973 \cdot 10^{-12}$

Como se puede apreciar, todos las parejas de puntos están perfectamente alineadas y el error de la estimación, que se lee en el pie de foto, es despreciable.

2. Calibración de la cámara usando homografías

Este apartado se ha implementado haciendo uso de las funciones disponibles en *OpenCV* para extraer los parámetros de una cámara basándose en varias imágenes de un mismo patrón: en este caso, un tablero de ajedrez.

2.1. Detección de las esquinas

Para cada una de las 25 imágenes de entrada se ha usado la función *findChessboardCorners()*, que recibe una imagen de entrada y un objeto *Size* que indica el patrón de puntos por fila y columna del tablero y devuelve, además de un booleano que indica si se ha encontrado o no el patrón especificado, un vector de puntos que marcan cada esquina del tablero.

Esta comprobación, junto con el posterior refinamiento de las coordenadas píxel y el dibujado de las marcas, se ha encapsulado en la función *findAndDrawChessBoardCorners()* de la clase *Image*. Su código es el siguiente:

```
1   bool Image::findAndDrawChessBoardCorners(Size pattern_size,
        vector<Point2f> &corners){
2       int flags = CV_CALIB_CB_ADAPTIVE_THRESH +
3                   CV_CALIB_CB_NORMALIZE_IMAGE +
4                   CALIB_CB_FAST_CHECK;
5       bool success = findChessboardCorners(this->image,
            pattern_size,
6                                            corners, flags);
7
8       if(success){
9           cornerSubPix(this->image, corners, Size(5, 5), Size(-1,
                -1),
10                      TermCriteria());
11          drawChessboardCorners(this->image, pattern_size,
                Mat(corners), true);
12      }
13
14      return success;
15  }
```

A los *flags* por defecto se ha añadido *CALIB_CB_FAST_CHECK*, que simplemente ayuda a optimizar el tiempo de procesado en aquellas imágenes en las que no se encuentra el patrón especificado.

En caso de que se hayan encontrado todas las marcas y se hayan podido ordenar tal y como indica el patrón, se refinan las coordenadas de las mismas con *cornerSubPix()*, que ayuda a mejorar la posterior calibración.

Además, se imprime en la imagen el conjunto de marcas ordenadas con ayuda de la función *drawChessboardCorners()*.

Esta función se llama desde el *main* como sigue:

```
    // Chessboard pattern and a vector to store it as many times as
        the number
    // of successfully-detected chessboards, for the later
        calibration.
    vector<vector<Point3f> > patterns;
    vector<Point3f> pattern;
    for (size_t i = 0; i < 12; i++) {
        for (size_t j = 0; j < 13; j++) {
            pattern.push_back(Point3f(i,j,0));
        }
    }

    // Vector to store the corners in each iteration and a vector
        to keep all
    // the iterations for the later calibration.
    vector<vector<Point2f> > boards;
    vector<Point2f> corners;

    string prefix, suffix, filename;
    prefix = "./imagenes/Image";
    suffix = ".tif";

    for (size_t i = 1; i <= 25; i++) {
        //Open i-th image
        filename = prefix + to_string(i) + suffix;
        Image img(filename, false);

        //If the corners are found, treat them, store the detected
            corners
        // and repeat the pattern for the calibration.
        if(img.findAndDrawChessBoardCorners(Size(13,12), corners)){
            boards.push_back(corners);
            patterns.push_back(pattern);

            // Show the image
            img.setName(filename);
            img.draw();
            waitKey(0);
            destroyAllWindows();
        }
    }
```

Como vemos en el bucle principal, para cada imagen se llama a la función que acabamos de describir y, en caso de terminar con éxito, se almacenan en sendos vectores de vectores tanto las esquinas encontrados como un patrón fijo de puntos objeto, igual para todas las imágenes. Estos dos vectores se usarán inmediatamente después para la calibración de la cámara.

El patrón que se va almacenando es el conjunto de puntos $(i, j, 0)$, donde

$i \in \{1, 2, 3, \ldots, 12\}$ y $j \in \{1, 2, 3, \ldots, 13\}$, pues tenemos 12 filas y 13 columnas. Hemos supuesto que las casillas tienen de ancho unidad 1 y que las mediciones son en el plano de la cámara —de ahí que la tercera coordenada sea cero—.

Por otro lado, llamamos a la función *findAndDrawChessBoardCorners()* con el objeto *Size(13,12)*, pues la función *findChessboardCorners()* espera un objeto (puntos por columna, puntos por fila) que determine el número de esquinas a buscar y su disposición.

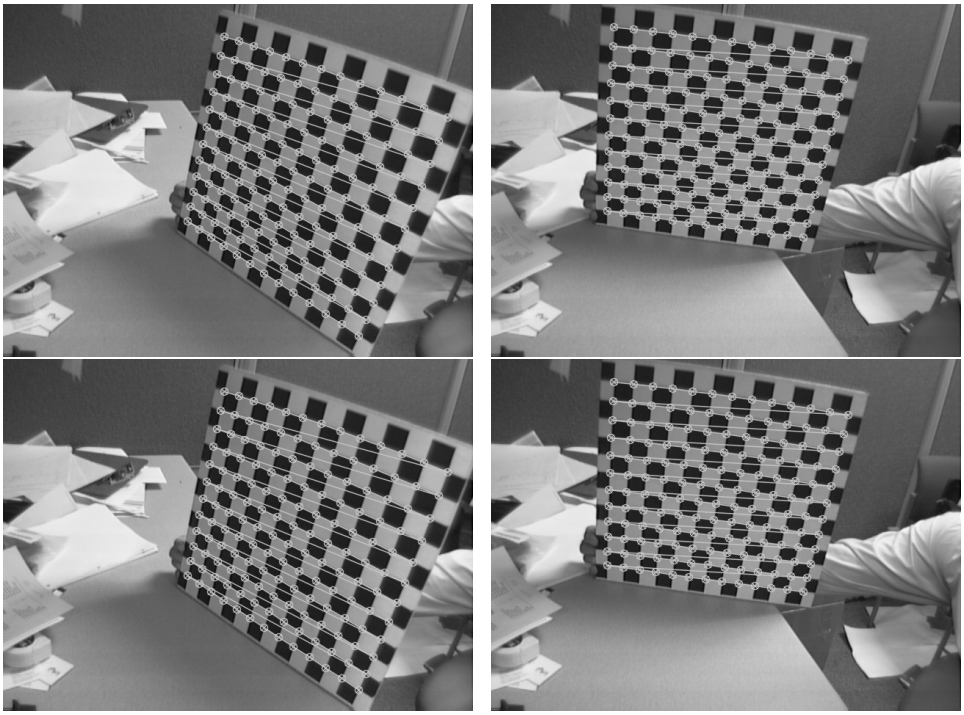

Figura 3: Figuras Tableros

En el caso de que las esquinas sean encontradas, se muestra la imagen con las marcas dibujadas y se espera a que el usuario pulse una tecla. La figura 3 muestra las 4 imágenes que se analizan con éxito.

2.2. Calibración de la cámara

Tras esta primera fase de detección ya sólo falta calibrar la cámara con los datos obtenidos. Veamos primero el código:

```
1 Mat camera_matrix, dist_coeffs;
2 vector<Mat> rvecs, tvecs;
3
4 // Actual calibration of the camera.
5
6 // Without optic distortion
```

```
7  // The 1st flag disables tangential correction; the other three
       disable
8  // radial correction.
9  int flags = CV_CALIB_ZERO_TANGENT_DIST |
10             CV_CALIB_FIX_K1 |
11             CV_CALIB_FIX_K2 |
12             CV_CALIB_FIX_K3;
13 double calib_error_without = calibrateCamera(patterns, boards,
14                                               Size(640,480),
15                                               camera_matrix, dist_coeffs,
16                                               rvecs, tvecs,
17                                               flags);
18
19 // Only radial distortion
20 flags = CV_CALIB_ZERO_TANGENT_DIST;
21 double calib_error_radial = calibrateCamera(patterns, boards,
22                                              Size(640,480),
23                                              camera_matrix, dist_coeffs,
24                                              rvecs, tvecs,
25                                              flags);
26
27 // Only tangential distortion
28 flags = CV_CALIB_FIX_K1 |
29         CV_CALIB_FIX_K2 |
30         CV_CALIB_FIX_K3;
31 double calib_error_tangent = calibrateCamera(patterns, boards,
32                                              Size(640,480),
33                                              camera_matrix, dist_coeffs,
34                                              rvecs, tvecs,
35                                              flags);
36
37 // With all optic distortion
38 //By default, tangential and radial correction are enabled
39 flags = 0;
40 double calib_error = calibrateCamera(patterns, boards,
       Size(640,480),
41                                              camera_matrix, dist_coeffs,
42                                              rvecs, tvecs,
43                                              flags);
44 cout << "SECTION 2:\tCalibration errors:" << endl;
45 cout << "\t\t\tWithout distortion:\t" << calib_error_without <<
       endl;
46 cout << "\t\t\tWith tang. distortion:\t" << calib_error_tangent <<
       endl;
47 cout << "\t\t\tWith radial distortion:\t" << calib_error_radial <<
       endl;
48 cout << "\t\t\tWith distortion:\t" << calib_error << endl;
```

Como vemos, con la estructura que tenemos ya sólo queda llamar a la

función *calibrateCamera* con los parámetros apropiados.

Los dos primeros parámetros corresponden a los vectores de los puntos objeto y de las esquinas, respectivamente; los cuatro siguientes son los parámetros de salida y el último es el de las opciones. Como veremos en la sección de resultados, jugando con estas opciones podremos mejorar el error de calibración.

2.3. Resultados

Como acabamos de observar en el código, tenemos bastante margen de prueba para mejorar el error de calibración. Centrémonos primero en el modelo que se busca al calibrar la cámara.

Las cuatro llamadas a *calibrateCamera* se hacen con opciones diferentes; a saber: indicando que se dejen fijos todos los parámetros de distorsión —suponiendo entonces que no existe distorsión radial ni tangencial—, indicando que se modele toda la distorsión óptica, restringiendo esta a la tangencial dejando fijos los parámetros de la radial y viceversa.

Tenemos así cuatro llamadas diferentes que devuelven errores cada vez mejores, y que reproducimos a continuación:

Without distortion :	1,3254
With tang. distortion :	1,30143
With radial distortion :	0,163034
With distortion :	0,162133

Vemos que si aplicamos un modelo sin distorsión el error es un orden de magnitud mayor que si tenemos en cuenta toda la distorsión. Se ve además, afinando con más detalle en las opciones sólo-distorsión o sólo-tangencial que los parámetros que influyen significativamente en la mejora del error son los de la distorsión radial; los de la tangencial, aunque también mejoran el error, lo hacen en un orden mucho menor, casi despreciable.

Es evidente que el modelo, y con él la calibración, mejoran notablemente teniendo en cuenta todos los parámetros de distorsión óptica.

Por otro lado, podemos estudiar la mejora que introduce el refinamiento de las coordenadas píxel de las esquinas detectadas en los tableros. Si omitimos la llamada a la función *cornerSubPix()*, los errores son los siguientes:

Without distortion :	1,54567
With tang. distortion :	1,52576
With radial distortion :	0,775207
With distortion :	0,773218

Es clara la ventaja de refinar las coordenadas píxel de las esquinas, ya que el mejor error que conseguimos sin refinamiento es casi 5 veces mayor que el mejor error que obtenemos con refinamiento.

Esto se ve incluso en las marcas superpuestas en las imágenes. La figura 4 muestra una de las imágenes que se calculan al ejecutar el código sin la llamada a *cornerSubPix()* en ella se aprecia cómo las marcas no están

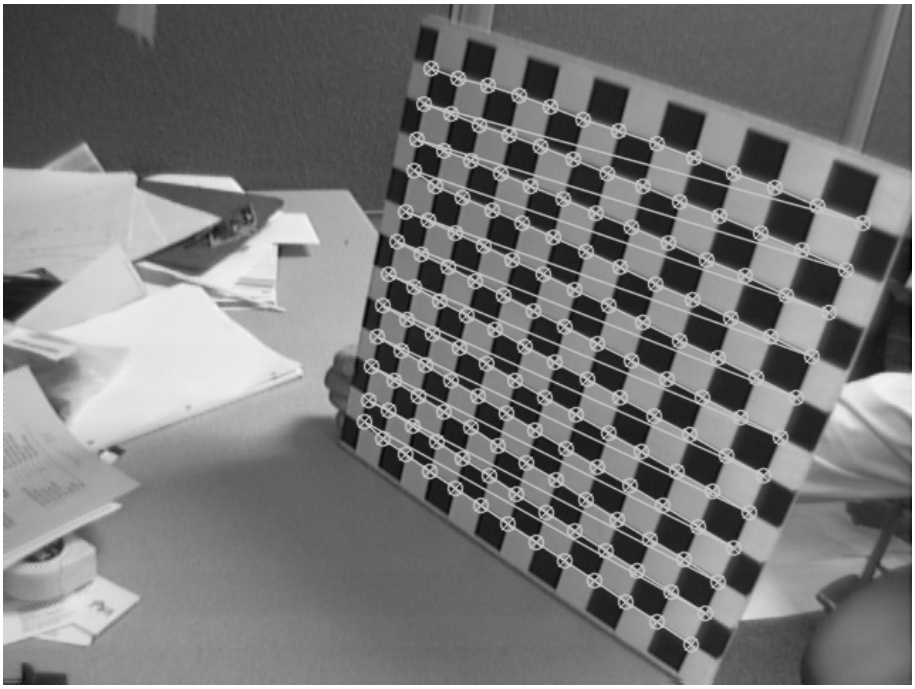

Figura 4: Imprecisiones en las esquinas al no usar *cornerSubPix()*.

exactamente en las esquinas, sino algo desplazadas —ver, por ejemplo, la parte izquierda de la cuarta fila de marcas o la primera marca de la última fila—. Se incluye de nuevo la imagen en la que se ha usado refinamiento, ampliada en la figura 5, para poder comparar.

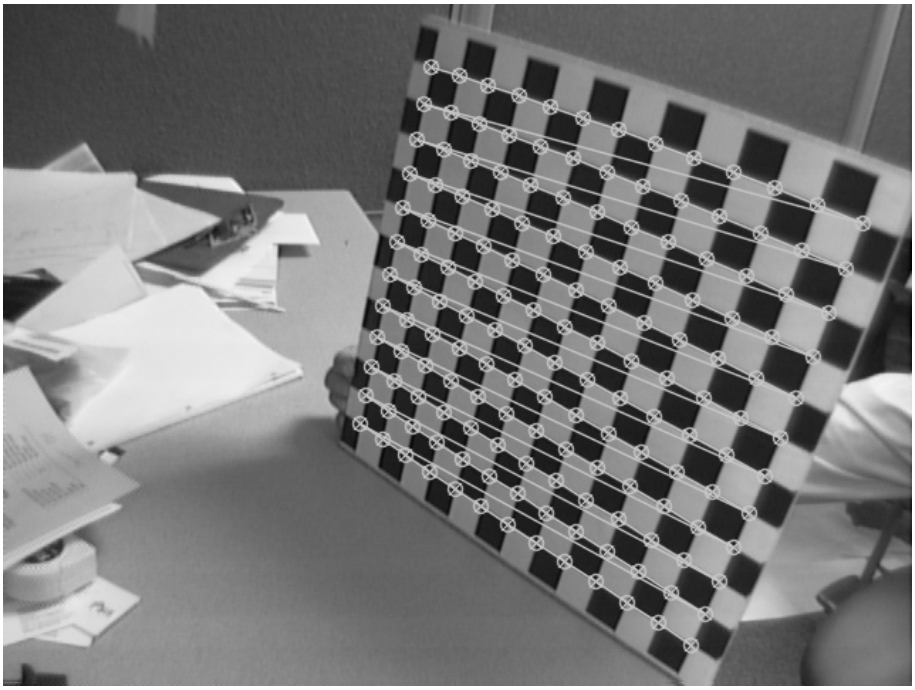

Figura 5: Imagen usando *cornerSubPix()*.

3. Estimación de la matriz fundamental

Todo el código referente a este apartado ha sido encapsulado en la función *computeAndDrawEpiLines()* de la clase *Image*, que actúa sobre dos objetos *Image*: uno sobre el que se llama a la función y el otro pasado por referencia. La función devuelve el error de la estimación de la matriz fundamental como la media de la distancia entre los puntos clave y sus líneas epipolars y, además, dibuja sobra ambas imágenes un subconjunto de las líneas epipolares calculadas.

El código de la función es el siguiente:

```
1   float Image::computeAndDrawEpiLines(Image &other, int
        num_lines){
2       vector<Point2d> good_matches_1;
3       vector<Point2d> good_matches_2;
4
5       Mat fund_mat = this->fundamentalMat(other, good_matches_1,
            good_matches_2);
6
7       vector<Vec3d> lines_1, lines_2;
8
9       computeCorrespondEpilines(good_matches_1, 1, fund_mat,
            lines_1);
10      computeCorrespondEpilines(good_matches_2, 2, fund_mat,
            lines_2);
11
12      RNG rng;
13      theRNG().state = clock();
14
15      // Draws both sets of epipolar lines and computes the
            distances between
16      // the lines and their corresponding points.
17      float distance_1 = 0.0, distance_2 = 0.0;
18      for (size_t i = 0; i < lines_1.size(); i++) {
19          Vec2d point_1 = good_matches_1[i];
20          Vec2d point_2 = good_matches_2[i];
21
22          Vec3d line_1 = lines_1[i];
23          Vec3d line_2 = lines_2[i];
24
25          // Draws only num_lines lines
26          if(i % (lines_1.size()/num_lines) == 0 ){
27              Scalar color(rng.uniform(0, 255),
28                           rng.uniform(0, 255),
29                           rng.uniform(0, 255));
30
31              line(other.image,
32                   Point(0,
```

```
                        -line_1[2]/line_1[1]),
                Point(this->cols(),
                        -(line_1[2] +
                            line_1[0]*this->cols())/line_1[1]),
                color
                );
            circle(this->image,
                   Point2f(point_1[0], point_1[1]),
                   4,
                   color,
                   CV_FILLED);

            line(this->image,
                Point(0,
                        -line_2[2]/line_2[1]),
                Point(other.cols(),
                        -(line_2[2] +
                            line_2[0]*other.cols())/line_2[1]),
                color
                );
            circle(other.image,
                   Point2f(point_2[0], point_2[1]),
                   4,
                   color,
                   CV_FILLED);

        }

        // Error computation with distance point-to-line
        distance_1 += abs(line_1[0]*point_2[0] +
                          line_1[1]*point_2[1] +
                          line_1[2]) /
                      sqrt(line_1[0]*line_1[0] +
                          line_1[1]*line_1[1]);

        distance_2 += abs(line_2[0]*point_1[0] +
                          line_2[1]*point_1[1] +
                          line_2[2]) /
                      sqrt(line_2[0]*line_2[0] +
                          line_2[1]*line_2[1]);
    }

    return (distance_1+distance_2)/(2*lines_1.size());
}
```

3.1. Puntos en correspondencia y matriz fundamental

La primera parte del apartado, correspondiente a la detección de los puntos en correspondencia entre las dos imágenes y a la obtención de la matriz fundamental, se ha encapsulado en la función *fundamentalMat* de la clase *Image*:

```
1   Mat Image::fundamentalMat(Image &other,
2                             vector<Point2d> &good_matches_1,
3                             vector<Point2d> &good_matches_2){
4
5       pair<vector<Point2f>, vector<Point2f> > matches;
6       Mat F;
7
8       matches = this->match(other, descriptor_id::BRUTE_FORCE,
            detector_id::ORB);
9
10      vector<unsigned char> mask;
11      F = findFundamentalMat(matches.first, matches.second,
12                             CV_FM_8POINT | CV_FM_RANSAC,
13                             1., 0.99, mask );
14
15      for (size_t i = 0; i < mask.size(); i++) {
16          if(mask[i] == 1){
17              good_matches_1.push_back(matches.first[i]);
18              good_matches_2.push_back(matches.second[i]);
19          }
20      }
21
22      return F;
23  }
```

Tras obtener las parejas de puntos en correspondencia entre las dos imágenes con la función *Image::match*, que ya describimos en la práctica anterior, se llama a la función *findFundamentalMat* de *OpenCV*, con las opciones *CV_FM_8POINT | CV_FM_RANSAC*, que permiten hacer el cálculo con el algoritmo de los 8 puntos junto con *RANSAC*.

Se ha elegido el menor umbral posible para RANSAC —el antepenúltimo parámetro, cuyo valor debe estar entre 1 y 3 según la documentación de *OpenCV*—, de manera que se filtren adecuadamente los falsos positivos. Además, se ha indicado que la confianza en la bondad de la matriz estimada sea de un 99 %.

Un aspecto muy importante de esta implementación es el uso del vector *mask*, en el que la posición i contiene un 0 o un 1, indicando si se ha usado o no la pareja i de puntos en correspondencia. Esto se usa después para filtrar el conjunto de puntos en correspondencia, quedándonos sólo con aquellos en los que más confiamos.

3.2. Líneas epipolares

Para calcular las líneas epipolares se ha usado la llamada a la función *computeCorrespondEpilines()*, que devuelve un vector de ternas de valores de la forma $[A, B, C]$. Estos valores se interpretan como los coeficientes de la recta

$$Ax + By + C = 0$$

lo que hace muy sencillo el posterior cálculo de distancias.

Una vez se tienen las líneas epipolares —nótese que las tenemos en el mismo orden que las parejas de puntos en correspondencia—, para dibujarlas basta obtener dos puntos para cada una y superponerlas en las imágenes con la función *line*.

Para dibujar sólo un número M de las N líneas que tenemos, seleccionamos sólo aquellas iteraciones en las que el índice es congruente con 0 módulo N/M.

En el mismo bucle de dibujado, pero esta vez ya sí con todas las iteraciones, calculamos todas las distancias entre puntos y su correspondientes líneas epipolares —esto es directo, pues al tener la recta en la forma $Ax+By+C = 0$, la fórmula de la distancia punto-recta es muy simple—. Vamos entonces sumando todos los valores y al final del bucle dividimos entre todas las iteraciones para tomar la distancia medio.

La función devuelve la media de los dos errores medios de cada imagen.

3.3. Resultados

Los resultados de este apartado son los esperados, con un error entre 0 y 1. Con los parámetros que ahora mismo tiene el código, el error obtenido —entendido como la media de la distancia de los puntos a sus líneas epipolares— es de 0,388278.

En la figura 6 se ven las imágenes de entrada con las líneas epipolares obtenidas superpuestas, además de los puntos en correspondencia.

La cantidad de parámetros a tener en cuenta en este caso es importante, ya que la detección de puntos y su posterior puesta en correspondencia dependen de los detectores usados, cuyo comportamiento depende mucho de los parámetros con los que se llaman, como ya vimos en la práctica anterior.

Por ejemplo, podemos jugar con el tipo de detector usado. Los resultados anteriores corresponden a una ejecución con el detector ORB, que es el que al final se ha dejado por devolver un error menor. La figura 7 muestra, por otro lado, el resultado de una ejecución con el detector BRISK. En este caso, el error es de 0.394629, algo mayor que con OBR.

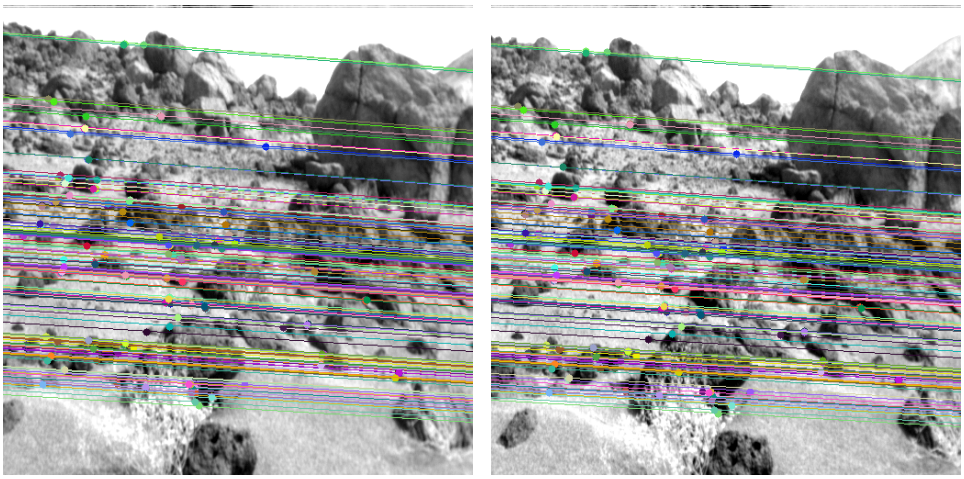

Figura 6: Par de imágenes en correspondencia y sus líneas epipolares (ORB).

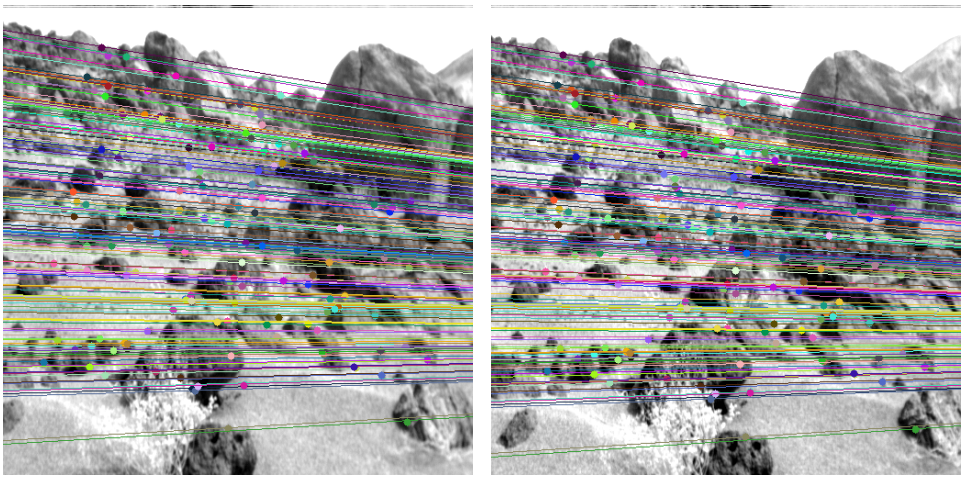

Figura 7: Par de imágenes en correspondencia y sus líneas epipolares (BRISK).

4. Movimiento de la cámara

El código correspondiente a este último apartado se encuentra en la función *reconstruction()* de la clase *Image*. Este método actúa sobre dos imágenes, recibe además la matriz de parámetros intrínsecos de la cámara y devuelve, en caso de éxito, las matrices de rotación y traslación que definen el movimiento de la cámara entre las dos imágenes. El algoritmo usado es el de reconstrucción euclídea descrito en los apuntes.

Las llamadas a esta función entre las tres parejas de imágenes que podemos formar —de la 0 a la 1, de la 0 a la 2 y de la 1 a la 2— son las siguientes:

```
1   double intrinsic_params[3][3] = {
2       {1839.6300000000001091, 0, 1024.2000000000000455},
3       {0, 1848.0699999999999363, 686.5180000000000291},
4       {0, 0, 1}
5   };
6
7   Mat K(3, 3, CV_64F, intrinsic_params);
8
9   Image reconstruction_0("./imagenes/rdimage.000.ppm");
10  Image reconstruction_1("./imagenes/rdimage.001.ppm");
11  Image reconstruction_2("./imagenes/rdimage.004.ppm");
12
13  bool success01, success02, success12;
14  Mat R01, R02, R12;
15  Point3d T01, T02, T12;
16
17  success01 = reconstruction_0.reconstruction(reconstruction_1,
        K, R01, T01);
18  success02 = reconstruction_0.reconstruction(reconstruction_2,
        K, R02, T02);
19  success12 = reconstruction_1.reconstruction(reconstruction_2,
        K, R12, T12);
20
21  cout << "SECTION 4:\tSuccess of the three reconstructions:" <<
        endl;
22  cout << "\t\t" << success01 << ", " << success02 << ", " <<
        success12 << endl;
23
24  cout << R01 << endl << T01 << endl << endl;
25  cout << R02 << endl << T02 << endl << endl;
26  cout << R12 << endl << T12 << endl << endl;
```

La matriz K, cuyos valores están codificados manualmente, se ha tomado de los archivos de calibración de las cámaras. Durante todo este apartado trabajaremos con el tipo de dato *double* y objetos Mat con datos *CV_64F*, pues la precisión será muy importante.

El código de la función *reconstruction()* es el siguiente:

```
bool Image::reconstruction(Image &other, Mat K, Mat &R, Point3d &T){
    // FUNDAMENTAL MATRIX
    vector<Point2d> img_pts1;
    vector<Point2d> img_pts2;

    Mat F = this->fundamentalMat(other, img_pts1, img_pts2);

    // ESSENTIAL MATRIX
    Mat E = K.t() * F * K;

    double trace_val = abs(trace(E.t() * E)[0]);
    double norm = sqrt(trace_val/2);

    E /= norm;

    // ALGORITHM
    // 1.
    T = obtainT(E);

    // 2.
    R = obtainR(E, T);

    // 3., 4.
    double f = K.at<double>(0,0);
    enum check test = checkTandR(T, R, f, img_pts1, img_pts2);
    bool success;

    // E,-T => 3.
    if(test == check::CHANGE_T){
        T = -T;
        test = checkTandR(T, R, f, img_pts1, img_pts2);
        if(test == check::CHANGE_T){                    // Repeat -> ERROR
            cout << "ERROR: Reconstruction has failed" << endl;
            success = false;
        }
        else if(test == check::CHANGE_E){               // -E,-T => 2.
            E = -E;
            R = obtainR(E, T);
            test = checkTandR(T, R, f, img_pts1, img_pts2);
            success = (test == check::SUCCESS);
        }
        else{
            success = true;
        }
    }
    // -E,T => 2.
```

```
47          else if(test == check::CHANGE_E){
48              E = -E;
49              R = obtainR(E, T);
50              test = checkTandR(T, R, f, img_pts1, img_pts2);
51
52              if(test ==check::CHANGE_T){                    // -E,-T
                    => 3.
53                  T = -T;
54                  test = checkTandR(T, R, f, img_pts1, img_pts2);
55                  success = (test == check::SUCCESS);
56              }
57              else if(test == check::CHANGE_E){              // Repeat
                    -> ERROR
58                  cout << "ERROR: Reconstruction has failed" << endl;
59                  success = false;
60              }
61              else{
62                  success = true;
63              }
64          }
65          // E,T => SUCCESS
66          else{
67              success = true;
68          }
69
70          return success;
71      }
```

4.1. Puntos en correspondencia, matrices fundamental y esencial

Como vemos, lo primero que se hace es llamar a la función *findFundamentalMat()*, descrita anteriormente. De ella obtenemos la matriz fundamental que, junto con la matriz K de parámetros intrínsecos, nos sirve para calcular la matriz esencial. Para ello, usamos su definición:

$$E = K^T F K$$

4.2. Normalización de E

Un paso muy importante en el algoritmo es la normalización de E, que nos asegura obtener un vector traslación normalizado.

Obtenemos así la E normalizada, $\hat{E}$, dividiendo todos los elementos de E como sigue:

$$\hat{E} = E / \sqrt{\frac{Traza(E^T E)}{2}}$$

4.3. Algoritmo de reconstrucción euclídea

Tras estos pasos previos estamos en disposición de implementar el algoritmo, que consiste en calcular el vector de traslación T a partir de la matriz normalizada $\hat{E}$, la matriz de rotación R a partir de $\hat{E}$ y T y comprobar que son correctas; esto es, que las proyecciones de todos los puntos por ambas cámaras tienen una coordenada de profundidad positiva.

En caso contrario, tenemos dos posibilidades:

1. Algún punto tiene profundidad negativa en las dos cámaras.
2. Algún punto tiene profundidad negativa en una cámara y positiva en la otra.

En la primera posibilidad, basta tomar $-T$ como vector de traslación y comprobar de nuevo las profundidades. En la segunda, hay que tomar $-\hat{E}$, recalcular R y comprobar de nuevo las profundidades.

Tenemos entonces cuatro posibilidades:

1. E, T
2. $E, -T$
3. $-E, T$
4. $-E, -T$

En el caso de que el algoritmo intente cambiar sucesivamente una de las dos matrices por su negativa, intentando repetir un proceso ya calculado, debemos abortar y devolver un error, como así hace la función. Toda esta lógica está implementada en el conjunto de *if-else* del código.

El algoritmo es claro, y sólo falta saber cómo calcular T y R. Estos cálculos se encuentran encapsulados en las funciones *obtainT()* y *obtainR()*.

4.3.1. Cálculo del vector de traslación

El cálculo del vector $\hat{T}$ es directo de la matriz $\hat{E}^T\hat{E}$. El código es el siguiente:

```
1   Point3d obtainT(Mat E){
2       // Squared essential matrix
3       Mat EtE = E.t() * E;
4
5       // Translation vector
6       double T_x = sqrt(1-EtE.at<double>(0,0));
7       double T_y = -EtE.at<double>(0,1) / T_x;
8       double T_z = -EtE.at<double>(0,2) / T_x;
9
10      return Point3d(T_x, T_y, T_z);
11  }
```

Las líneas anteriores simplemente calculan $\hat{T}$ resolviendo las ecuaciones

$$E'_{1,1} = 1 - \hat{T}_x^2$$
$$E'_{1,2} = -\hat{T}_x \hat{T}_y$$
$$E'_{1,3} = -\hat{T}_x \hat{T}_z$$

donde $E' = \hat{E}^T \hat{E}$.

4.3.2. Cálculo de la matriz de rotación

Calcular R dadas las matrices $\hat{E}$ y $\hat{T}$ es directo, y se ha implementado como sigue:

```
1       Mat obtainR(Mat E, Point3d T){
2           // Rotation matrix
3           Mat row_i, w_i;
4           vector<Mat> w;
5           Mat R;
6
7           for (size_t i = 0; i < 3; i++) {
8               row_i = E.row(i);
9               w_i = row_i.cross(Mat(T).t());
10              w.push_back(w_i);
11          }
12
13          int j,k;
14          for (size_t i = 0; i < 3; i++) {
15              j = (i+1)%3;
16              k = (j+1)%3;
17
18              R.push_back(w[i] + w[j].cross(w[k]));
19          }
20
21          return R;
22      }
```

La matriz R se calcula así:

$$R = \begin{pmatrix} w_1 + w_2 \times w_3 \\ w_2 + w_3 \times w_1 \\ w_3 + w_1 \times w_2 \end{pmatrix}$$

donde $w_i = \hat{E}_i \times \hat{T}$, con $\hat{E}_i$ la i-ésima fila de $\hat{E}$.

4.4. Comprobación de la bondad de R y T

Sólo una de las cuatro posibilidades vistas anteriormente es correcta. Falta explicar cómo, en la lógica de los *if-else* que vimos antes, se decide

cuándo cambiar el signo de $\hat{T}$ y cuándo el de $\hat{E}$. Esto se implementa con la función *checkTandR()*, cuyo código es el siguiente:

```
enum check checkTandR(Point3d T, Mat R, double f,
                      vector<Point2d> img_pts1,
                      vector<Point2d> img_pts2){

    vector<double> ptZ_i, ptZ_d;
    Mat mat_Z_i, mat_Z_d, pt_3D_i;
    double Z_i, Z_d, x_d;
    Mat pt_i, pt_d;

    for (size_t i = 0; i < img_pts1.size(); i++) {
        pt_i = Mat(Vec3d(img_pts1[i].x,img_pts1[i].y,1.));
        pt_d = Mat(Vec3d(img_pts2[i].x,img_pts2[i].y,1.));
        x_d = pt_d.at<double>(0,0);

        mat_Z_i = f * (f*R.row(0) - x_d*R.row(2)) * Mat(T)
                  /
                  ((f*R.row(0) - x_d*R.row(2)) * pt_i);
        Z_i = mat_Z_i.at<double>(0,0);

        pt_3D_i = Z_i * pt_i / f;

        mat_Z_d = R.row(2) * (pt_3D_i - Mat(T));
        Z_d = mat_Z_d.at<double>(0,0);

        if(Z_i < 0 && Z_d < 0){
            return check::CHANGE_T; // Go to step 3.
        }
        else if(Z_i*Z_d < 0){
            return check::CHANGE_E; // Go to step 2.
        }
    }

    return check::SUCCESS; // Finish
}
```

Simplemente se reconstruyen las coordenadas de profundidad Z_i y Z_d —los subíndices se refieren a las imágenes *i*zquierda y *d*erecha— de todos los puntos con las siguientes fórmulas:

$$Z_i = f\frac{(fR_1 - x_d R_3)^T \hat{T}}{(fR_1 - x_d R_3)^T p_i}$$
$$Z_d = R_3^T(Z_i\frac{p_i}{f} - \hat{T})$$

donde

- f es el elemento $(1,1)$ de la matriz K

- R_j es la fila j-ésima de la matriz R
- x_d es la primera coordenada del punto p_d, proyección en la imagen del punto P por la cámara de la derecha.

Si hay algún punto con ambas proyecciones con profundidad negativa, se devuelve el estado *CHANGE_T*; si sólo una de las proyecciones tiene profundidad negativa, *CHANGE_E*; en caso de que todos los puntos tengan ambas proyecciones positivas, se devuelve *SUCCESS*.

Estos estados se han implementado como un *enum* en el archivo *consts.hpp* como sigue:

```
1        enum check{
2            CHANGE_E = -1,
3            CHANGE_T = 0,
4            SUCCESS = 1
5        };
```

4.5. Resultados

En todos los casos probados se encuentran matrices de rotación y traslación coherentes con el modelo, que son las siguientes:

Entre imágenes 0 y 1

$$R_{0,1} \begin{pmatrix} 0{,}9773680768147368 & -0{,}02268527605130709 & -0{,}005673345999255498 \\ 0{,}01640790268699308 & 0{,}9528832785661331 & -0{,}1159468621215086 \\ 0{,}002751299120479188 & 0{,}130216904310353 & 0{,}9693117838582967 \end{pmatrix}$$

$$T_{0,1} \begin{pmatrix} 0{,}0971566 & 0{,}893906 & -0{,}389973 \end{pmatrix}$$

Entre imágenes 0 y 2

$$R_{0,2} \begin{pmatrix} 0{,}7875735463181901 & 0{,}007627829477894865 & 0{,}1829250849645255 \\ -0{,}004885372766241861 & 0{,}766512518229334 & 0{,}1094451690870099 \\ -0{,}06827156302888834 & -0{,}02395283502877557 & 0{,}6597912367539547 \end{pmatrix}$$

$$T_{0,2} \begin{pmatrix} 0{,}314713 & 0{,}265056 & -0{,}697594 \end{pmatrix}$$

Entre imágenes 1 y 2

$$R_{1,2} \begin{pmatrix} 0{,}8639698545556994 & -0{,}1033214448681555 & 0{,}167476964219665 \\ -0{,}008170064326311092 & 0{,}8142781316600718 & -0{,}2748569419796844 \\ 0{,}006007306904464715 & 0{,}1375011844621707 & 0{,}8588456519791023 \end{pmatrix}$$

$$T_{1,2} \begin{pmatrix} 0{,}674835 & -0{,}176223 & -0{,}60376 \end{pmatrix}$$