相机标定(二)——图像坐标与世界坐标转换

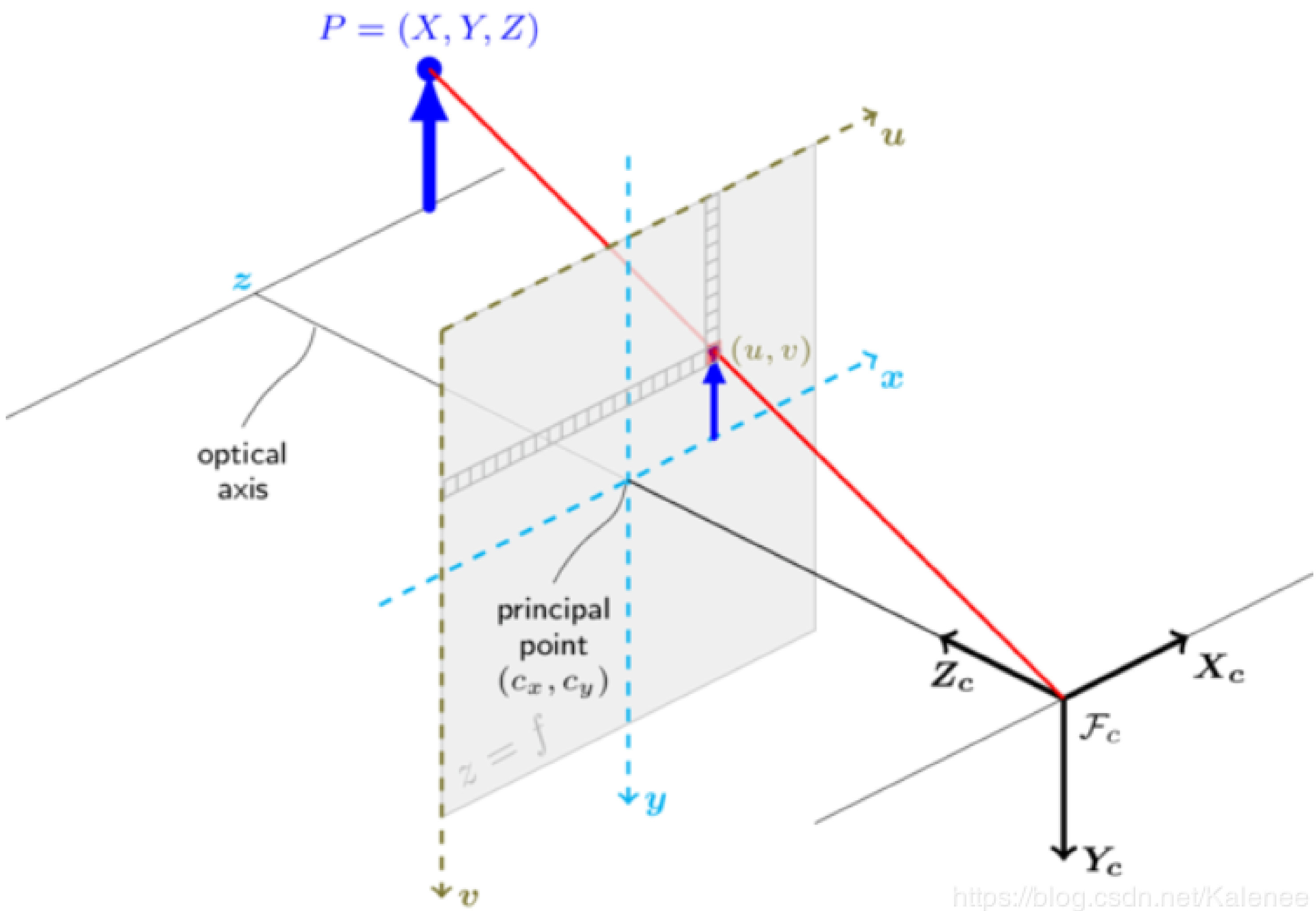
white_Learner

[相机标定(一)——内参标定与程序实现]

相机标定(二)——图像坐标与世界坐标转换

[相机标定(三)——手眼标定]

一、坐标关系



相机中有四个坐标系，分别为world，camera，image，pixel

world为世界坐标系，可以任意指定xw轴和yw轴，为上图P点所在坐标系。
camera为相机坐标系，原点位于小孔，z轴与光轴重合，xw轴和yw轴平行投影面，为上图坐标系XcYcZc。
image为图像坐标系，原点位于光轴和投影面的交点，xw轴和yw轴平行投影面，为上图坐标系XYZ。
pixel为像素坐标系，从小孔向投影面方向看，投影面的左上角为原点，uv轴和投影面两边重合，该坐标系与图像坐标系处在同一平面，但原点不同。

二、坐标变换

下式为像素坐标pixel与世界坐标world的变换公式，右侧第一个矩阵为相机内参数矩阵，第二个矩阵为相机外参数矩阵。

$$s\begin{bmatrix}u\\v\\1\end{bmatrix}=\begin{bmatrix}f_x & 0 & c_x\\0 & f_y & c_y\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}r_{11} & r_{12} & r_{13} & t_1\\r_{21} & r_{22} & r_{23} & t_2\\r_{31} & r_{32} & r_{33} & t_3\end{bmatrix}\begin{bmatrix}x\\y\\z\\1\end{bmatrix}$$

2.1 变换流程

$$P_{uv}=KTP_w$$

该方程右侧隐含了一次齐次坐标到非齐次坐标的转换

- K内参T_{camera}^{pixel}：像素坐标系相对于相机坐标系的变换（与相机和镜头有关）
- T外参T_{world}^{camera}：相机坐标系相对于世界坐标系的变换

顺序变换

从pixel到camera，使用内参变换

$$P_{camera}(3\times1)=T_{camera}^{pixel}{}^{-1}(3\times3)*P_{pixel}(3\times1)*depth$$

从camera到world，使用外参变换

$$P_{world}(4\times 1) = T_{world}^{camera^{-1}}(4\times 4) * P_{camera}(4\times 1)$$

注意：两个变换之间的矩阵大小不同，需要分开计算，从pixel到camera获得的相机坐标为非齐次，需转换为齐次坐标再进行下一步变换。而在进行从camera到world时，需将外参矩阵转换为齐次再进行计算。（齐次坐标的分析）

直接变换

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = R^{-1}(M^{-1} * s * \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} - t)$$

注意：直接变换是直接根据变换公式获得，实际上包含pixel到camera和camera到world，实际上和顺序变换一样，通过顺序变换可以更清晰了解变换过程。

2.2 参数计算

内参：通过张正友标定获得
外参：通过PNP估计获得
深度s：深度s为目标点在相机坐标系Z方向的值

2.3 外参计算

solvePnP函数
Perspective-n-Point是通过n组给定点的世界坐标与像素坐标估计相机位置的方法。OpenCV内部提供的函数为solvePnP()，函数介绍如下：

```
bool solvePnP(InputArray objectPoints,
                     InputArray imagePoints,
                     InputArray cameraMatrix,
                     InputArray distCoeffs,
                 OutputArray rvec,
                 OutputArray tvec,
                 bool useExtrinsicGuess=false,
                 int flags=ITERATIVE )

```

objectPoints，输入世界坐标系中点的坐标；

imagePoints，输入对应图像坐标系中点的坐标；
cameraMatrix, 相机内参数矩阵；
distCoeffs, 畸变系数；
rvec, 旋转向量，需输入一个非空Mat，需要通过cv::Rodrigues转换为旋转矩阵；
tvec, 平移向量，需输入一个非空Mat；
useExtrinsicGuess, 默认为false，如果设置为true则输出输入的旋转矩阵和平移矩阵；
flags，选择采用的算法；
CV_ITERATIVE Iterative method is based on Levenberg-Marquardt optimization. In this case the function finds such a pose that minimizes reprojection error, that is the sum of squared distances between the observed projections imagePoints and the projected (using projectPoints()) objectPoints .
CV_P3P Method is based on the paper of X.S. Gao, X.-R. Hou, J. Tang, H.-F. Chang “Complete Solution Classification for the Perspective-Three-Point Problem”. In this case the function requires exactly four object and image points.
CV_EPNP Method has been introduced by F.Moreno-Noguer, V.Lepetit and P.Fua in the paper “EPnP: Efficient Perspective-n-Point Camera Pose Estimation”.

注意：solvePnP()的参数rvec和tvec应该都是double类型的

程序实现

```
//输入参数
Mat cameraMatrix = Mat(3, 3, CV_32FC1, Scalar::all(0)); /* 摄像机内参数矩阵 */
Mat distCoeffs = Mat(1, 5, CV_32FC1, Scalar::all(0)); /* 摄像机的5个畸变系数：k1,k2,p1,p2,k3 */
double zConst = 0;//实际坐标系的距离，若工作平面与相机距离固定可设置为0

//计算参数
double s;
Mat rotationMatrix = Mat (3, 3, DataType<double>::type);
Mat tvec = Mat (3, 1, DataType<double>::type);
void calcParameters(vector<cv::Point2f> imagePoints, vector<cv::Point3f> objectPoints)
{
        //计算旋转和平移
        Mat rvec(3, 1, cv::DataType<double>::type);
        cv::solvePnP(objectPoints, imagePoints, cameraMatrix, distCoeffs, rvec, tvec);
        cv::Rodrigues(rvec, rotationMatrix);
}

```

2.4 深度计算

理想情况下，相机与目标平面平行（只有绕Z轴的旋转），但实际上相机与目标平面不会完全平行，存在绕X和Y轴的旋转，此时深度s并不是固定值t3，计算深度值为：

$$s = t_3 + r_{31} * x + r_{32} * y + r_{33} * z$$

若使用固定值进行变换会导致较大误差。解决方案如下：

计算多个点的深度值，拟合一个最优值

通过外参计算不同位置的深度（此处采用该方案）

注意：此处环境为固定单目与固定工作平面，不同情况下获得深度方法不同。

像素坐标pixel与世界坐标world转换公式可简化为

$$s\begin{bmatrix}u\\v\\1\end{bmatrix}=M(R\begin{bmatrix}X\\Y\\Z_{const}\end{bmatrix}+t)$$

M为相机内参数矩阵，R为旋转矩阵，t为平移矩阵，zconst为目标点在世界坐标Z方向的值，此处为0。

变换可得

$$R^{-1}M^{-1}s\begin{bmatrix}u\\v\\1\end{bmatrix}=\begin{bmatrix}X\\Y\\Z_{const}\end{bmatrix}+R^{-1}t$$

当相机内外参已知可计算获得s

三、程序实现

3.1 Matlab

```matlab
clc;
clear;

% 内参
syms fx cx fy cy;
M = [fx,0,cx;
    0,fy,cy;
    0,0,1];

% 外参
%旋转矩阵
syms r11 r12 r13 r21 r22 r23 r31 r32 r33;
R = [r11,r12,r13;
  r21,r22,r23;
  r31,r32,r33];
%平移矩阵
syms t1 t2 t3;
t = [t1;
    t2;
    t3];
%外参矩阵
T = [R,t;
    0,0,0,1];

% 图像坐标
syms u v;
imagePoint = [u;v;1];

% 计算深度
syms zConst;
rightMatrix = inv(R)*inv(M)*imagePoint;
leftMatrix = inv(R)*t;
s = (zConst + leftMatrix(3))/rightMatrix(3);

% 转换世界坐标方式一
worldPoint1 = inv(R) * (s*inv(M) * imagePoint - t);
simplify(worldPoint1)

% 转换世界坐标方式二
cameraPoint = inv(M)* imagePoint * s;% image->camrea
worldPoint2 = inv(T)* [cameraPoint;1];% camrea->world
worldPoint2 = [worldPoint2(1);worldPoint2(2);worldPoint2(3)];
simplify(worldPoint2)

```

3.2 C++

该程序参考《视觉SLAM十四讲》第九讲实践章：设计前端代码部分进行修改获得，去掉了李群库Sopuhus依赖，因该库在windows上调用较为麻烦，若在Linux建议采用Sopuhus。

camera.h

```
#ifndef CAMERA_H
#define CAMERA_H

#include <Eigen/Core>
#include <Eigen/Geometry>
using Eigen::Vector4d;
using Eigen::Vector2d;
using Eigen::Vector3d;
using Eigen::Quaterniond;
using Eigen::Matrix;

class Camera
{
public:
    Camera();

    // coordinate transform: world, camera, pixel
    Vector3d world2camera( const Vector3d& p_w);
    Vector3d camera2world( const Vector3d& p_c);
    Vector2d camera2pixel( const Vector3d& p_c);
    Vector3d pixel2camera( const Vector2d& p_p);
    Vector3d pixel2world ( const Vector2d& p_p);
    Vector2d world2pixel ( const Vector3d& p_w);

        // set params
        void setInternalParams(double fx, double cx, double fy, double cy);
        void setExternalParams(Quaterniond Q, Vector3d t);
        void setExternalParams(Matrix<double, 3, 3>  R, Vector3d t);

        // cal depth
        double calDepth(const Vector2d& p_p);

private:
    // 内参
        double fx_, fy_, cx_, cy_, depth_scale_;
        Matrix<double, 3, 3> inMatrix_;

    // 外参
    Quaterniond Q_;
        Matrix<double, 3, 3>  R_;
    Vector3d t_;
        Matrix<double, 4, 4> exMatrix_;
};

#endif // CAMERA_H

```

camera.cpp

```
#include "camera.h"

Camera::Camera(){}

Vector3d Camera::world2camera ( const Vector3d& p_w)
{
        Vector4d p_w_q{ p_w(0,0),p_w(1,0),p_w(2,0),1};
        Vector4d p_c_q = exMatrix_ * p_w_q;
```

```
        Vector4d p_c_q = exMatrix_ * p_w_q;
        return Vector3d{p_c_q(0,0),p_c_q(1,0),p_c_q(2,0)};
}

Vector3d Camera::camera2world ( const Vector3d& p_c)
{
        Vector4d p_c_q{ p_c(0,0),p_c(1,0),p_c(2,0),1 };
        Vector4d p_w_q = exMatrix_.inverse() * p_c_q;
    return Vector3d{ p_w_q(0,0),p_w_q(1,0),p_w_q(2,0) };
}

Vector2d Camera::camera2pixel ( const Vector3d& p_c )
{
    return Vector2d (
               fx_ * p_c ( 0,0 ) / p_c ( 2,0 ) + cx_,
               fy_ * p_c ( 1,0 ) / p_c ( 2,0 ) + cy_
           );
}

Vector3d Camera::pixel2camera ( const Vector2d& p_p)
{
        double depth = calDepth(p_p);
    return Vector3d (
               ( p_p ( 0,0 )-cx_ ) *depth/fx_,
               ( p_p ( 1,0 )-cy_ ) *depth/fy_,
               depth
           );
}

Vector2d Camera::world2pixel ( const Vector3d& p_w)
{
    return camera2pixel ( world2camera(p_w) );
}

Vector3d Camera::pixel2world ( const Vector2d& p_p)
{
    return camera2world ( pixel2camera ( p_p ));
}

double Camera::calDepth(const Vector2d& p_p)
{
        Vector3d p_p_q{ p_p(0,0),p_p(1,0),1 };
        Vector3d rightMatrix = R_.inverse() * inMatrix_.inverse() * p_p_q;
        Vector3d leftMatrix = R_.inverse() * t_;
        return leftMatrix(2,0)/rightMatrix(2,0);
}

void Camera::setInternalParams(double fx, double cx, double fy, double cy)
{
        fx_ = fx;
        cx_ = cx;
        fy_ = fy;
        cy_ = cy;

        inMatrix_ << fx, 0, cx,
                                0, fy, cy,
                                0, 0, 1;
}

void Camera::setExternalParams(Quaterniond Q, Vector3d t)
{
```

```
        Q_ = Q;
        R_ = Q.normalized().toRotationMatrix();
        setExternalParams(R_, t);
}

void Camera::setExternalParams(Matrix<double, 3, 3> R, Vector3d t)
{
        t_ = t;
        R_ = R;

        exMatrix_ << R_(0, 0), R_(0, 1), R_(0, 2), t(0,0),
                R_(1, 0), R_(1, 1), R_(1, 2), t(1,0),
                R_(2, 0), R_(2, 1), R_(2, 2), t(2,0),
                0, 0, 0, 1;
}

```

参考

image coordinate to world coordinate opencv

Computing x,y coordinate (3D) from image point

单应矩阵

camera_calibration_and_3d

《视觉SLAM十四讲》—相机与图像+实践章：设计前端