Augmented Reality Assignment01

Author: Joao Vitor Elias Carvalho | Student Number: 15309326

2015/2016 Hilary Term - CS7434 - Trinity College Dublin

1 Introduction

1.1 Goal

The goal of this application is to implement a simple Augmented Reality experiment. The final result must be a visualization of a 3D model in a real ambient captured by a camera. The object must be projected on a planar surface marked using some kind of marker. The object should move and be correctly oriented with the plane. It's necessary to calibrate the camera before augmenting its images with the 3D model.

1.2 Pipeline/Brief Approach

The approach used was to use the following technologies: C++ as main programming language,OpenCV as computer vision library, OpenGL as the graphical library and GLEW/FreeGLUT for the integration of the C++ application with the OpenGL. We are going to use the chessboard as marker for identification of the plane, which the 3D model is going to be projected above.

The Pipeline The pipeline used for the implementation, which is going to be explain in detail later, was the following:

- Get the camera and store N frames when the chessboard is detected.

- Use the N frames to calibrate the camera and get the camera intrinsic matrix.

- Now use every frame to get the extrinsic matrix using the intrinsic matrix and the detected chessboard.

- Convert the rotation vector to a rotation matrix using the Rodrigues operation.

- Generate the frustrum projection matrix and RT (Rotation and Translation) matrix using the rotation matrix,translated vector and intrinsic matrix.

- Generate a polygon and use the frame as texture to it. Render it on the OpenGL screen.

- Render the model using the projection and RT matrix as Model-View-Projection matrix to the OpenGL screen in front of the polygon holding the frame as texture.

1.3 Challenges

The challenges of the project are:

- Detection of the markers

- Calibration of the camera and finding intrinsic matrix.

- Finding the extrinsic matrix using correspondences of points.

- Finding the Model-View-Projection matrix using the extrinsic and intrinsic matrices.

- Sending the MVP matrix to the graphics pipeline and rendering the 3D model correctly.

Others kinds of challenges that I faced were:

- Integration of the computer vision library with the computer graphics library.

2 Approach

2.1 Problem to be solved

When approaching this problem we need to think about the camera model we are dealing with. We need to understand the process involved when the camera takes a picture of the world. We are dealing with a transformation from the 3D point (position of the object in the world) to a 2D point (image pixel) that is made by the camera.

We have a relation between the 2D point, that we going to denote as p , and the 3D point, that we going to denote as P. In this case we know that the following relationship holds with P and p as follow:

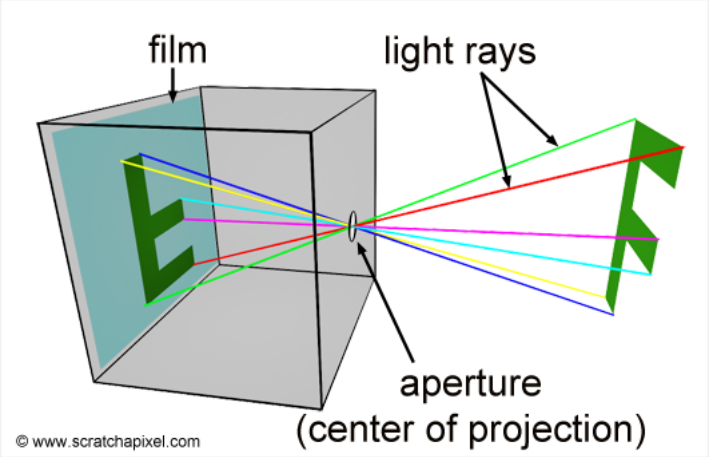
$$p = C * P$$

In which C is the matrix that converts the 3D point in the world frame to the 2D point camera frame. Also we have a non-linear operation to project the 3D to 2D. We are going to define what this transformation is supposed to do and derive its components.

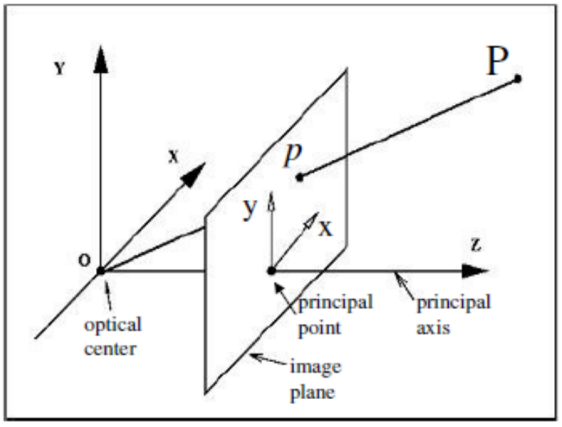
$$P = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

$$p = \begin{bmatrix} x \\ y \end{bmatrix}$$

With the definition of P and p we can think about the process of the camera taking a picture. In theory this process works by projecting the rays through a hole in a lightproof box and the rays are going to be projected in the back of the box as shown below. This is the idea behind the pinhole camera used in the modern cameras.



So we need to calculate the relation between a 3D point and 2D in this camera. We can, hypothetically, bring the image plane to the front of the pinhole and the model still is equivalent. So below we have the representation of this idea:



By similarity of triangles we can get the following relationships between the P and p. In this case f is the distance to the optical center and the principal point (or the image plane).

$$x = f * \frac{X}{Z}$$

$$y = f * \frac{Y}{Z}$$

This operation is a non-linear operation, but using homogeneous coordinate we can define it as a linear operation because when we have a homogeneous representation we can assume that:

$$p = \lambda * p_h$$

In other words we say that the normal representation is equal to the homogeneous representation multiplied with a certain lambda.This lambda is (1/z), we are going to have a step, that we are going to call it as projection step, that we do the non-linear part. In this case we have the following relationship:

$$\lambda * p_h = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} * P$$

$$p = proj(p_h)$$

$$p = \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix}$$

Besides that we need to convert from the image measurement to the pixel based measurement. As the origin of the image pixels are in the top-left corner and the origin of the image may be in any point we need a operation to do this:

$$\begin{bmatrix} x_{pixel} \\ y_{pixel} \\ 1 \end{bmatrix} = \begin{bmatrix} -1/s_x & 0 & o_x \\ 0 & -1/s_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Where Sx and Sy are the pixel size and Ox and Oy is where the center of the view is located within the image. Summarizing the operations we formulate so far we have:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} -1/s_x & 0 & o_x \\ 0 & -1/s_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = K \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

The RT matrix is used to rotate and translate the camera, so it holds information about the orientation and position of the camera in the space. The RT matrix is also known as the extrinsic matrix as it holds information of the world around the camera. The K matrix is called intrinsic matrix and it holds information about how the camera works(field of view, aperture, etc.). The RT and K matrix is what is known as the Camera Matrix.

This is the point we are going to start. Our problem is given a certain camera and a certain pattern detection we need to extract the K matrix and the RT matrix so we can render our 3D model. So we define our problem as follow:

Input: P, p

Output: $C = K * RT$

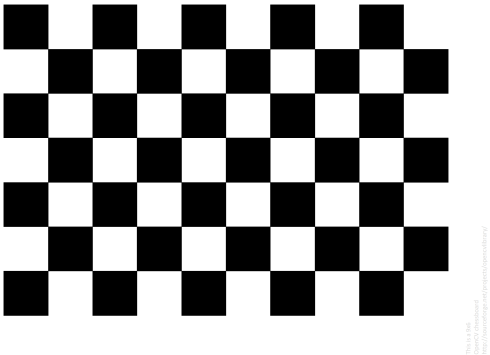
This is known as **camera calibration**.

2.2 Image Capturing and Marker Detection

Initially the capturing of the camera was used with the help of the OpenCV class of VideoCapture. The result is a Mat object with the frame information. I'm using the webcam from my notebook so the device id is always zero.

A Chessboard, as showed in the image, was used because of its regular pattern that can be easily detected. The detection of the chessboard works by applying a binary threshold filter, get the connected components (or contours as in

OpenCV), for each countour it checks the aspect size and box size (as given by the minimum rect area). This is methods that the chessboard detection is composed of, but I could not find more information on why it works and how exactly is done.



This is going to let us know if the chessboard is present in the scene and where are the positions of the corners in the image. We are going to use theses images to calibrate the camera and calculate the extrinsic matrix. So when the chessboard is present, we store this image in an array of images that we are going to use to calibrate.

2.3 Camera Calibration

Using the images found in the step 1.1 we can calibrate the camera. This process works by trying to solve for the Camera matrix in the following equation:

$$p = C * P$$

In our case we have the points of the chessboard from the image and we can generate a set of 3D points for the chessboard by knowing how many squares we have and giving a certain size to them. It is important to give them a Z coordinate as well. In this case because it's a plane, we can set the z coordinate to be 0 to all points. We are considering one of the points to be the origin and the others to stay in the plane where z is equal 0.

We can modify our equation so the C is a vector instead of a matrix and P is a matrix as follow:

$$\begin{bmatrix} X & Y & Z & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & X & Y & Z & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & Y & Z & 1 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} C_{11} \\ C_{12} \\ C_{13} \\ C_{14} \\ C_{21} \\ C_{22} \\ C_{23} \\ C_{24} \\ C_{31} \\ C_{32} \\ C_{33} \\ C_{34} \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

This equations is equivalent to the equation that relates P, C and p. If we continue to add rows in the first matrix with the data from the images, as well as the result vector, we will have a linear system that we can solve to find C. We can solve the linear system by using methods like the **Gaussian Elimination.** That way we can find all the components of C.

Next we will need to find the extrinsic and intrinsic matrix. This can be done by doing a QR decomposition we can separate the two matrices that are composing C, K and RT. After doing the decomposition K is the upper triangular matrix and R is the orthogonal matrix.

This is one way to find the Camera matrix. The way actually implemented in the OpenCV library, as stated by Bouquet[1], is to use: a closed-form solution, followed by a nonlinear refinement based on the maximum likelihood criterion as described by Zhang[2]. Bouquet said that: "The main initialization phase has been partially inspired from that paper. The initial estimation of the planar homographies is identical to that presented in that paper. The closed-form estimation of the internal parameters from the homographies is slightly different (we explicitly use orthogonality of vanishing points), and we do not estimate the distortion coefficients at the initialization phase (we found that step unnecessary in practice). The final Maximum Likelihood estimation is identical. Minor difference: we use the intrinsic model presented by Heikkil and Silven including two extra distortion coefficients corresponding to tangential distortion."[3].

In the implementation this all is accomplish by the calibrateCamera function of the OpenCV library. This functions receives as parameters the following:

- objectPoints: vector<vector<Point3f>> formed by the P vectors, the 3D correspondent of the elements.

- imagePoints: vector<vector<Point2f>> formed by the p vectors, the points found by the identification of the chessboard.

- imageSize: Size object representing the size of the image.

- cameraMatrix: result Matrix that is going to be outputted the result.

- distCoeffs: result of the function for finding the distortions.

- rvecs: vector of vectors representing the rotation of the camera.

- tvecs: Output vector of translation vectors estimated for each pattern view.

- flags: Used to modify the behavior of the function.

2.4 Finding the Extrinsic Matrix

In the last step we have found both the intrinsic matrix and the extrinsic matrix. We have calibrated the camera and now we need to get every frame and find the rotation and translations vectors. With this we can mount the RT matrix to be passed to OpenGL. We are going to use the solvePnP function from OpenCV that follows the same method as the calibration function, only it only calculates the rotation vector and translation vector already knowing the intrinsic matrix. With this we use the Rodrigues method to get the rotation matrix. We use the rotation matrix and the translation vector to create the view matrix as follows:

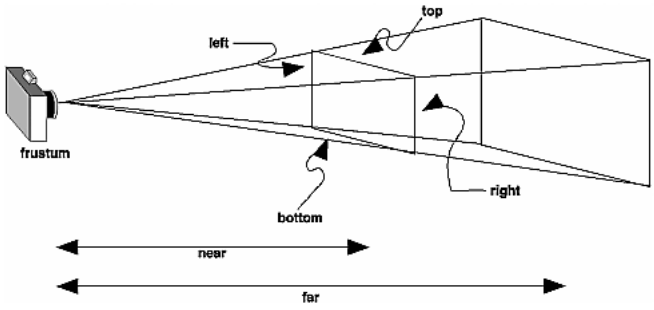
$$view = \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.5 Generating the Frustrum Matrix

Now that we have both the intrinsic matrix and the RT matrix we can generate the right perspective and view for the 3D model. We are going to use the frustrum matrix as our perspective matrix and use the parameters from the intrinsic matrix to use. The RT matrix is going to be used as view matrix.

$$Persp = \begin{bmatrix} \frac{2*near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2*near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2*far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

We are going to define the $right, left, top, bottom$ parameters as well as the far and near. For this we need to understand what each parameter of the intrinsic matrix means. The right and left parameters are interval of the x coordinate in the near clipping plane. The top and bottom parameters are the same for the y coordinate as shown in the image below:



Our intrinsic matrix give us the distance between the image plane and the camera origin both in the x axis and y axis. With this we can derive the following relationship:

$$\frac{right}{near} = \frac{width_{image} - C_x}{fx}$$

$$\frac{left}{near} = \frac{-C_x}{fx}$$

Similarly we can define the top and bottom parameters. This way we can define our perspective projection. The near and far are chosen easily by placing a low near and a great far. This values are arbitrary but you need to consider the size of the board to know the distance from the camera to it and not occlude the 3D model.

2.6 Using the frame as a texture and rendering in the OpenGL screen

In order to display the image and the 3D model we need to render it in some way. I decided to render the frame and the 3D model in the OpenGL pipeline and screen. My idea was to generate a simple plane to use as medium to display the frame and render the 3D model in front of it. I used the frame data as a texture to the plane. Also I temporally disabled the z-buffer test before rendering the plane so the 3D model is always in front of it. I used a simple shader to render the plane. More can be found on the source code of the project.

```
glUseProgram(postProcessingShader->getProgramID());
// Bind the texture to the image
glBindTexture(GL_TEXTURE_2D, mTexture);
glEnable(GL_TEXTURE_2D);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, frame.cols, frame.rows, 0, GL_BGR, GL_UNSIGNED_BYTE, frame.data);

// Draw a Polygon with the texture
glBegin(GL_POLYGON);
glTexCoord2f(0.0f, 0.0f); glVertex3f(1.0f, 1.0f, 1000.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1000.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, 1000.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(1.0f, -1.0f, 1000.0f);
glEnd();

glFlush();

glBindTexture(GL_TEXTURE_2D, 0);
```

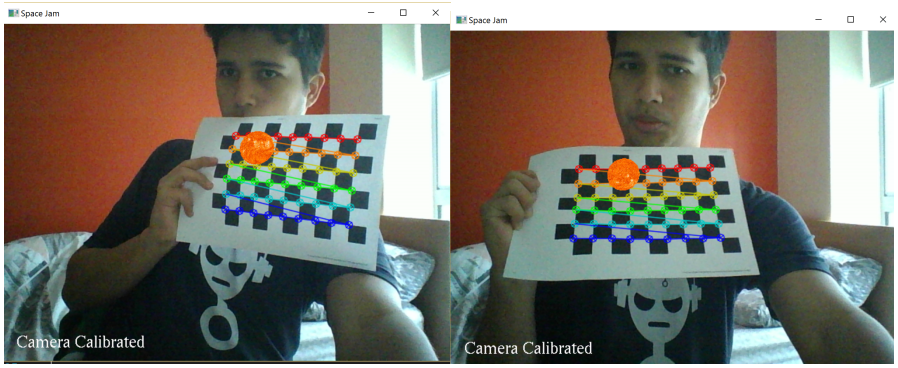
2.7 Rendering the 3D model

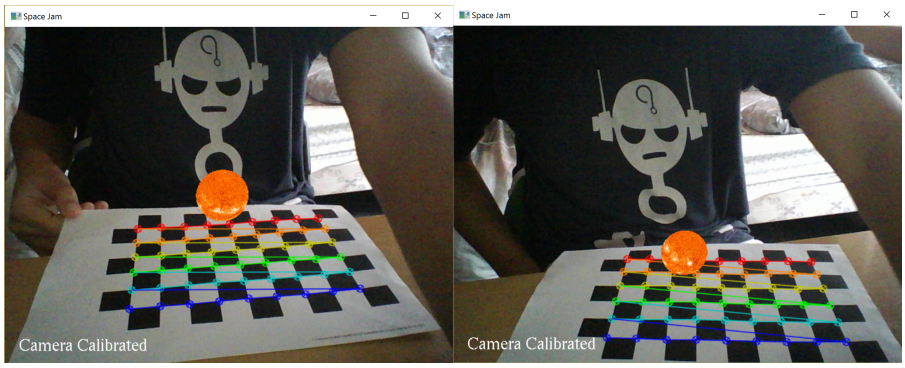
Finally I used the Assimp Library to load the 3D model from a .obj file. With the data from the model I filled the vertex buffer. Also I used a texture in order to improve the graphics and to more easily distinguish the rotations. The 3D model is a sphere with the texture of the sun. I render it using a simple shader using texture and receive a MVP matrix as parameter. Every vector is multiplied by the matrix in order to get the final position. No illumination model was used in order to keep it simple.

3 Evaluation and Results

3.1 Results

The results were good, after a series of complications with the integration of the libraries. I had some major problems with Visual Studio, OpenCV and OpenGL to integrate all together. After that, most of the steps went okay. You can see some of the results below:





3.2 Evaluation

The initial tests were made using an tablet showing the grid of the chessboard. The results using this method were poor due the light emitted from the tablet. This made the detection of the chessboard worse because of lower contrast caused by light emitted. Still you can use the tablet and get sufficient results to test your application, but I would recommend printing the grid for the testing.

Other great tip is to wait a certain number of frames between acquiring the images for the calibration. This allow you to move the camera more and get different samples of the chessboard for the calibration. This improves the calibration significantly. A simple counter that is restarted every time you store an image is enough.

After much experimenting with the number of frames that we use for the camera calibration, I realized that 15 is the good number to use. It doesn't take much time to calibrate and give us good results.

4 Conclusion

In the end I was able to augment the scene with the 3D model. The choose of using the OpenCV library enabled me to easily implement the camera calibration and solve for the extrinsic and intrinsic matrix. The results were okay given that is a simple augmented reality application.

The practical application of the content enabled me to test my knowledge and find out where I was lacking more understanding. I got stuck in transforming the intrinsic matrix into the perspective matrix for a while. Also I had major problems with the configuration of the OpenCV and OpenGL libraries on my computer. After much digging I discovered that I was using the wrong version of the compiled OpenCV library, so you should be careful to that. If you are using Visual Studio be sure to know what plataform toolset you need for OpenCV, Glew/Freeglut and OpenGL.

4.1 Future works and Improvements

Still much can be done to improve performance in this program. Effectively we can use a better method for finding the chessboard grid using some kind of tracking method, because the chessboard is not going to change much between frames. Applying some mip mapping can help with the aliasing in the texture. It needed to figure out a method for antialiasing between the frame and the 3D model.

In the sense of new features we can create a full model of the solar system to be used in a educational environment and maybe use other kinds of markers to generate a interface to the user interact with it.

References

[1] http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/links.html

[2] http://research.microsoft.com/~zhang/Calib/

[3] http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/ref.html