So I have calibrated…

What now?

The next step

- Calibration allows on to relate the position of one camera to the position of another camera in 3D space!!!
- We can use this information of warp images as if they were taken in a canonical epipolar configuration.
- Canonical epipolar geometry greatly simplifies dense stereo matching. Why?
- This warping process is called rectification.

Reading

- Fusiello, A., Trucco, E., & Verri, A. (2000). A compact algorithm for rectication of stereo pairs. Machine Vision and Applications, 12(1), 16-22.

Rectification Strategy

- Problem: There are infinitely many canonical epipolar configurations that we can chose for our cameras.
- Solution: Chose one that results in a small amount of warping.
- Strategy:
  - Find a “good” epipolar configuration
  - Derive a warp that will convert our images into this form.

A “good” configuration: Old Projective matrices

- $P_{old} = K[R|T]$
- K is the camera matrix
  - $\begin{bmatrix} \alpha_x & \gamma & c_x \\ 0 & \alpha y & c_y \\ 0 & 0 & 1 \end{bmatrix}$ where image centre is at (c_x, c_y), γ is the skewness factor
  - α_x = focal length divided by pixel size.
- [R|T] is the rotation matrix R combined with the translation vector T to produce a rigid transform.

A “good” configuration: Camera locations

- $P_{old} = [Q|q]$ where Q is a 3x3 matrix and q is a 3x1 column vector.
- Optic centre $c = -Q^{-1}q$ (this formula is in your notes)
- Note that one can calculate the optic centre for each camera, this is the actual locations of the camera in 3D space from the coordinates of the calibration object.

A “good” configuration: A good alignment

- Strategy: leave translations intact, find a new rotation matrix
- Rotation vector for X: $v_1 = (c_1 - c_2)$ where c_1 and c_2 are centres.
- Rotation vector for Y: $v_2 = R_3^T \times v_1$
- Rotation vector for Z: $v_3 = v_1 \times \; v_2$
- Normalize the 3 rotation vectors and construct the rotation vector by stacking them.

Camera Matrix, new projection matrices, Homographies.

- $K_{new} = \frac{k_{old1}+k_{old2}}{2}$
- Ideal Projection Camera 1: $P_{new1} = K_{new}\,[R_{new} \quad -R_{new}C_1]$
- Ideal Projection Camera 2: $P_{new2} = K_{new}\,[R_{new} \quad -R_{new}C_2]$
- Homography 1 : $H_1 = P_{new1}P_{old1}^{-1}$
- Homography 2 : $H_2 = P_{new2}P_{old2}^{-1}$
- Note: A homography (in computer vision) maps images to different planar surfaces in space.

How do homographies work?

- Given coordinate p = (x,y) in the current image, find the coordinate q in the rectified image.
- q = Hp
- Note that in order to avoid holes, we often perform this operation by looping through new coordinates and extracting intensity values from the old position in the image.
- Thus we calculate $p = H^{-1}q$
- Problem: Images are discrete, but homography calculations almost always lead to floating point values. Solution: Bilinear interpolation

Bilinear interpolation

- Problem: Find the intensity values for pixels with floating point coordinates.
- Solution 1: Rounding! Problem, leads to artifacts.
- Solution 2: Bilinear interpolation.
  - Based on 1D blending function: $\alpha A + (1 - \alpha)B$
  - However considers the 4 neighbourhood of a pixel

Bilinear Interpolation continued…

```
/**
 * Interplation functionality
 * @param image The image that we are interpolating
 * @param position The position that we want a color value for
 * @return The color that is returned from the interpolation method
 */
Vec3b Interpolate::GetColor(Mat& image, Point2f& position)
{
  double x1 = floor(position.x), x2 = x1 + 1;
  double y1 = floor(position.y), y2 = y1 + 1;
  Vec3f color1 = ExtractColor(image, x1, y1) * (x2 - position.x) * (y2 - position.y);
  Vec3f color2 = ExtractColor(image, x2, y1) * (position.x - x1) * (y2 - position.y);
  Vec3f color3 = ExtractColor(image, x1, y2) * (x2 - position.x) * (position.y - y1);
  Vec3f color4 = ExtractColor(image, x2, y2) * (position.x - x1) * (position.y - y1);
  Vec3f total = color1 + color2 + color3 + color4;
  return Vec3b(total);
}

//-----------------------------------------------------------------------------------------
// Helper Methods
//-----------------------------------------------------------------------------------------

/**
 * Extract a color from a given image
 * @param image The image that we are extracting a point from
 * @param x The x value of the coordinate that we are extracting
 * @param y The y value of the coordinate that we are extracting
 * @return The color that we are extracting
 */
Vec3f Interpolate::ExtractColor(Mat& image, double x, double y)
{
  if (x < 0 || x >= image.cols) return 0;
  if (y < 0 || y >= image.rows) return 0;
  Vec3b extractedColor = image.ptr<Vec3b>((int)y)[(int)x];
  return Vec3f(extractedColor);
}
```