

Basic part:

TODO part:

```
    //! resample pixels of the target view one by one
    for (int r = 0; r < Resolution_Row; r++)
    {
        for (int c = 0; c < Resolution_Col; c++)
        {
            Point3d rayRGB(0, 0, 0);
            //! resample the pixel value of this ray: TODO
            const double HALF_PLANE = Baseline * 4; // view plane
            double Vx_start = -120;
            double Vy_start = 120;
            // neighbors index on the target grid.
            int u1 = floor((HALF_PLANE + Vx) / Baseline); // u
            int u2 = ceil((HALF_PLANE + Vx) / Baseline); // ui+1
            int v1 = floor((HALF_PLANE - Vy) / Baseline); // vi
            int v2 = ceil((HALF_PLANE - Vy) / Baseline); // vi+1

            // scale down to the target grid point
            double scaled_Vx = Vx / Baseline, scaled_Vy= Vy / Baseline;
            double scaled_Vx_start = Vx_start / Baseline, scaled_Vy_start =
Vy_start / Baseline;

            double a = scaled_Vx - (scaled_Vx_start + u1 ); // alpha
            double b = (scaled_Vy_start - v1 ) - scaled_Vy; // beta
            // target grid viewpoint index from image plane = t * 9 + s =>
Vi * 9 + Ui

            Color vp1, vp2, vp3, vp4;
            viewImageList[v1 * 9 + u1].getColor(c, r, vp1.R, vp1.G, vp1.B);
            viewImageList[v1 * 9 + u2].getColor(c, r, vp2.R, vp2.G, vp2.B);
            viewImageList[v2 * 9 + u1].getColor(c, r, vp3.R, vp3.G, vp3.B);
            viewImageList[v2 * 9 + u2].getColor(c, r, vp4.R, vp4.G, vp4.B);




            //target = (1-b)*((1-a)*pixel(u1,v1) + a*pixel(u2,v1)) + b*((1-
a)*pixel(u1,v2)+ a*pixel(u2,v2))
            rayRGB.x = (1 - b) * ((1 - a) * vp1.R + a * vp2.R) + b * ((1 -
a) * vp3.R + a * vp4.R);
            rayRGB.y = (1 - b) * ((1 - a) * vp1.G + a * vp2.G) + b * ((1 -
```




```

a) * vp3.G + a * vp4.G);
    rayRGB.z = (1 - b) * ((1 - a) * vp1.B + a * vp2.B) + b * ((1 -
a) * vp3.B + a * vp4.B);
    //! record the resampled pixel value
    targetView.setColor(c, r, (unsigned char)rayRGB.x, (unsigned
char)rayRGB.y, (unsigned char)rayRGB.z);
    }
}

```

### Sample Output

P(-90 90 0)	
P(-90 120 0)	
P(-110 100 0)	

P(-120 90 0)		
P(-120 100 0)		
P(-120 120 0)		

Enhancement Part:

**Implementation of full version of the algorithm:**

```

Point3d rayRGB(0, 0, 0);
//! resample the pixel value of this ray: TODO
// focal length ratio between the target focal length and the
source focal length (Focal_Length).
double focallengthRatio = targetFocallen / Focal_Length;
// scaled pixel coordinates (dx, dy) in the target image.

```

```

        double dx = (c - ((double)Resolution_Col) / 2) *
focallengthRatio ;
        double dy = (r - ((double)Resolution_Row) / 2) *
focallengthRatio ;

        // real-world coordinates (x, y) based on the scaled pixel
coordinates
        double x = Vx + Vz / Focal_Length/ targetFocallen * ((c -
255.5) / 256 * 17.5);
        double y = Vy + Vz / Focal_Length / targetFocallen * ((r -
255.5) / 256 * 17.5);

        // pixel coordinates (col, row) in the image
        int col = ceil((c - Resolution_Col / 2.0) / targetFocallen *
Focal_Length + Resolution_Col / 2);
        int row = ceil((r - Resolution_Row / 2.0) / targetFocallen *
Focal_Length + Resolution_Row / 2);

        // half plane size
        const double HALF_PLANE = Baseline * 4; // view plane
        double x_start = -120;
        double y_start = 120;

        if ((abs(x) <= HALF_PLANE) && (abs(y) <= HALF_PLANE)
            &&(col >= 0) && (col < Resolution_Col) && (row >= 0) &&
(row < Resolution_Row))
        {
            // neighbors of image plane
            int u1 = floor((HALF_PLANE + x) / Baseline); // u
            int u2 = ceil((HALF_PLANE + x) / Baseline); // ui+1
            int v1 = floor((HALF_PLANE - y) / Baseline); // vi
            int v2 = ceil((HALF_PLANE - y) / Baseline); // vi+1

            Point2d grid_point((HALF_PLANE + x) / Baseline,
(HALF_PLANE - y) / Baseline);

            // scale down to the target grid point
            double scaled_x = x / Baseline, scaled_y = y / Baseline;
            double scaled_x_start = x_start / Baseline, scaled_y_start
= y_start / Baseline;

            double a = scaled_x - (scaled_x_start + u1); // alpha
            double b = (scaled_y_start - v1) - scaled_y; // beta

```

```

// target grid viewpoint index from image plane = t * 9 + s
=> Vi * 9 + Ui

Color vp1, vp2, vp3, vp4;

viewImageList[v1 * 9 + u1].getColor(col, row, vp1.R,
vp1.G, vp1.B);
viewImageList[v1 * 9 + u2].getColor(col, row, vp2.R,
vp2.G, vp2.B);
viewImageList[v2 * 9 + u1].getColor(col, row, vp3.R,
vp3.G, vp3.B);
viewImageList[v2 * 9 + u2].getColor(col, row, vp4.R,
vp4.G, vp4.B);

rayRGB.x = (1 - b) * ((1 - a) * vp1.R + a * vp2.R) + b
* ((1 - a) * vp3.R + a * vp4.R);
rayRGB.y = (1 - b) * ((1 - a) * vp1.G + a * vp2.G) + b
* ((1 - a) * vp3.G + a * vp4.G);
rayRGB.z = (1 - b) * ((1 - a) * vp1.B + a * vp2.B) + b
* ((1 - a) * vp3.B + a * vp4.B);
}

//! record the resampled pixel value
targetView.setColor(c, r, (unsigned char)rayRGB.x, (unsigned
char)rayRGB.y, (unsigned char)rayRGB.z);

```

The whole steps are as follows:

Calculate the focal length ratio between the target focal length and the source focal length (Focal\_Length).

Compute the scaled pixel coordinates (dx, dy) in the target image.

Calculate the real-world coordinates (x, y) based on the scaled pixel coordinates, depth information (Vz), and the focal lengths.

Compute the pixel coordinates (col, row) in the input image.

Define the half plane size (HALF\_PLANE) as four times the baseline (Baseline) and set the starting coordinates (x\_start, y\_start) for the grid.

Check if the real-world coordinates (x, y) are within the half plane size and the pixel coordinates (col, row) are within the input image dimensions.

Compute the grid coordinates (u1, u2, v1, v2) of the four neighboring viewpoints that surround the real-world coordinates (x, y).

Calculate the grid\_point, scale down the real-world coordinates (x, y) and the starting coordinates (x\_start, y\_start) to the target grid.

Compute the alpha (a) and beta (b) values for bilinear interpolation.

Extract the RGB color values of the corresponding pixel (col, row) from the four neighboring viewpoints (vp1, vp2, vp3, vp4).

Perform bilinear interpolation using the alpha (a) and beta (b) values to obtain the resampled RGB pixel.

Sample result:

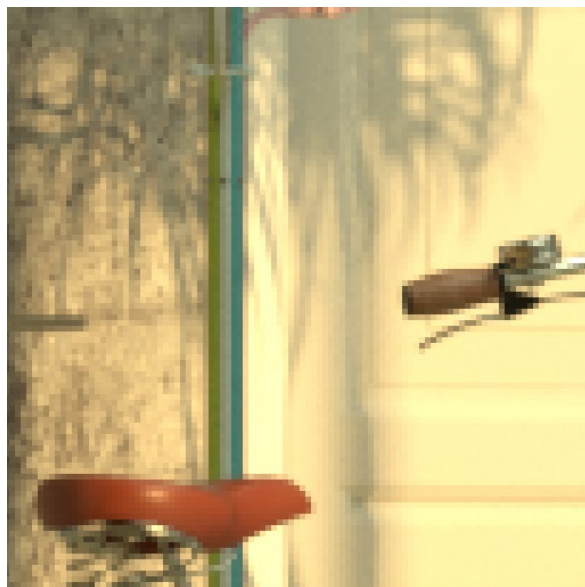
P:0 0 100 50



P:0 0 100 100



P:0 0 100 500





### Histogram Equalization for quality improvement:

```
void equalizeHistogram(Bitmap& image) {
    int width = image.getWidth();
    int height = image.getHeight();
    int numPixels = width * height;

    vector<int> histogram(256, 0);
    vector<int> cdf(256, 0);

    // calculate the histogram
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            unsigned char red, green, blue;
            image.getColor(x, y, red, green, blue);
            histogram[red]++;
        }
    }

    // compute the cumulative distribution function (CDF)
    cdf[0] = histogram[0];
    for (int i = 1; i < 256; i++) {
        cdf[i] = cdf[i - 1] + histogram[i];
    }

    // equalize the histogram
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            unsigned char red, green, blue;
            image.getColor(x, y, red, green, blue);
            int newVal = (int)(round(255.0 * cdf[red] / numPixels));
            image.setColor(x, y, newVal, newVal, newVal);
        }
    }
}
```

For each intensity level  $i$  (0-255), count the number of pixels in the image with intensity  $i$ . This will give the histogram  $h(i)$ .



Then compute the cumulative distribution function (CDF) of the histogram by summing the histogram values up to each intensity level  $i$ . This gives the CDF:  $\text{cdf}(i) = h(0) + h(1) + \dots + h(i)$ .

After that, normalize the CDF by dividing each  $\text{cdf}(i)$  by the total number of pixels in the image (width  $\times$  height). This will give a normalized CDF, ranging from 0 to 1.

Finally scale the normalized CDF by multiply each normalized  $\text{cdf}(i)$  by the maximum intensity level (usually 255 for 8-bit images). This will give the equalized intensity mapping:  $\text{eq}(i) = \text{round}(255 \times \text{cdf}(i))$ .

Calling this function after the main loop to replace the intensity values of the original image with the equalized intensity values  $\text{eq}(i)$

```
if(argc == 7)
    equalizeHistogram(targetView);
```

To use this function, we need to accept more than 6 arguments in the input:

Sample result:

```
./viewSynthesis.exe LF_views 0 0
100 500 eq
```



References:

<https://stackoverflow.com/questions/50578876/histogram-equalization-using-python-and-opencv-without-using-inbuilt-functions>

<https://www.geeksforgeeks.org/histogram-plotting-and-stretching-in-python-without-using-inbuilt-function/>

<https://medium.com/hackernoon/histogram-equalization-in-python-from-scratch->

[ebb9c8aa3f23](#)