实验四.立体视觉

0.本实验由16030199019赵祺文独立完成

1.实验概述

（1）光度测量立体视觉（详见讲义第 18 讲）：给定在不同的已知光照方向下从相同视角拍摄的一组图像，从中恢复物体表面的反照率(albedo)和法线方向(normals)。（2）平面扫描立体视觉（详见讲义第 16 讲）：给定同一场景从不同的视角拍摄的两幅校准图像，从中恢复出粗略的深度图。（3）基于泊松方程重建深度图（详见讲义第 18 讲）：根据法线图及粗略深度图，恢复出物体每个点的深度，并重建 3D 网格。

2.实施细节

2.1光度测量立体视觉

- 利用朗伯方程描述物体表面对入射光的漫反射

$$I = k_d \mathbf{N} \cdot \mathbf{L}$$

给定若干光线入射方向L及所观察到的图像强度I，就可以求解出每个点的反照率K_d和法线方向N。例如给定物体上一个点的三个方程：

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = k_d \begin{bmatrix} \mathbf{L_1}^T \\ \mathbf{L_2}^T \\ \mathbf{L_3}^T \end{bmatrix} \mathbf{N}$$

$$\underbrace{\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix}}_{\substack{\mathbf{I} \\ 3\times 1}} = \underbrace{\begin{bmatrix} \mathbf{L_1}^T \\ \mathbf{L_2}^T \\ \mathbf{L_3}^T \end{bmatrix}}_{\substack{\mathbf{L} \\ 3\times 3}} \underbrace{k_d \mathbf{N}}_{\substack{\mathbf{G} \\ 3\times 1}}$$

解此方程得： $\mathbf{G} = \mathbf{L}^{-1}\mathbf{I}$

$$k_d = \|\mathbf{G}\|$$

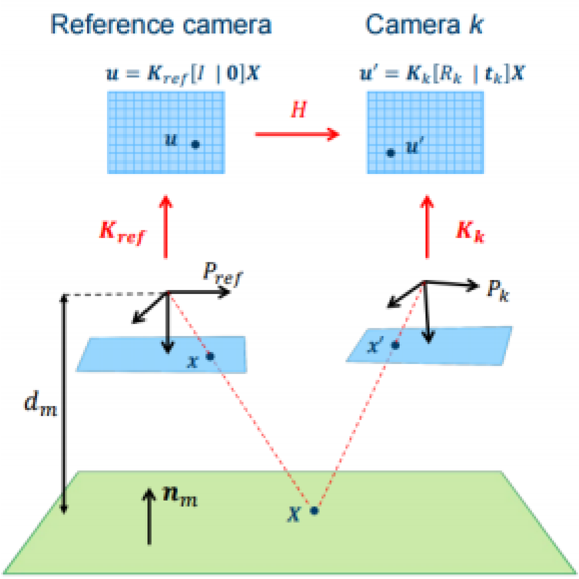
$$\mathbf{N} = \frac{1}{k_d}\mathbf{G}$$

- 代码实现

```python
def compute_photometric_stereo_impl(lights, images):
    """
    Given a set of images taken from the same viewpoint and a corresponding set
    of directions for light sources, this function computes the albedo and
    normal map of a Lambertian scene.

    If the computed albedo for a pixel has an L2 norm less than 1e-7, then set
    the albedo to black and set the normal to the 0 vector.

    Normals should be unit vectors.

    Input:
        lights -- N x 3 array.  Rows are normalized and are to be interpreted
                  as lighting directions.
        images -- list of N images.  Each image is of the same scene from the
                  same viewpoint, but under the lighting condition specified in
                  lights.
    Output:
        albedo -- float32 height x width x 3 image with dimensions matching the
                  input images.
        normals -- float32 height x width x 3 image with dimensions matching
                   the input images.
"""

```

```python
    L = lights
    L_T = L.T
    albedo = np.zeros((images[0].shape[0], images[0].shape[1], images[0].shape[2]), dtype=np.float32)
    normals = np.zeros((images[0].shape[0], images[0].shape[1], 3), dtype=np.float32)
    term1 = np.linalg.inv(L_T.dot(L))
    for channel in range(images[0].shape[2]):
        for row in range(images[0].shape[0]):
            for col in range(images[0].shape[1]):
                I = [(images[i][row][col][channel]).T for i in range(len(images))]
                term2 = L_T.dot(I)  # LT*I
                G = term1.dot(term2)
                k = np.round(np.linalg.norm(G), 5)
                if k < 1e-7:
                    k = 0
                else:
                    normals[row][col] += G / k
                albedo[row][col][channel] = k
    normals /= images[0].shape[2]
    return albedo, normals
```

2.2平面扫描立体视觉



- 平面扫描立体视觉的具体步骤

1. 将每幅图像I_k相对于每个深度平面Π_m投射到参考平面P_{ref}，使用单应映射$H^{-1}_{\Pi_m,P_k}$，得到的卷绕图像记为$\breve{I}_{k,m}$

2. 计算I_{ref}和$\breve{I}_{k,m}$的相似度

   使用 ZNCC (Zero-mean Normalized Cross Correlation)

3. 对每个深度平面计算所有 k 幅图片的相似度

$$M(u,v,\Pi_m) = \sum_k \underset{W}{ZNCC}(I_{ref}, \breve{I}_{k,m})$$

4. 对每个像素，选择最佳深度

$$\breve{\Pi}(u,v) = \underset{m}{\operatorname{argmax}}\, M(u,v,\Pi_m)$$

2.2.1 project_impl

该函数将三维点坐标映射到一个标定过的相机坐标（详细原理见第 10 讲：相机），投影矩阵为

$$\Pi = K\,[R \mid t]\quad ,$$

其中 K 为 3*3 的内参数矩阵，$[R \mid t]$为 3*4 的外参数矩阵。

映射公式为

$$u = \Pi\, X\quad ,$$

其中 X 为三维点的齐次坐标(4*1)，u 为映射到相机平面上的二维点的齐次

- 代码实现

```python
def project_impl(K, Rt, points):
    """
    Project 3D points into a calibrated camera.
    Input:
        K -- camera intrinsics calibration matrix
        Rt -- 3 x 4 camera extrinsics calibration matrix
        points -- height x width x 3 array of 3D points
    Output:
        projections -- height x width x 2 array of 2D projections
    """
    projection_matrix = K.dot(Rt)
    height, width = points.shape[:2]
    projections = np.zeros((height, width, 2))
    curr_point = np.zeros(3)

    for row_i, row in enumerate(points):
```

```
        for col_j, column in enumerate(row):
            curr_point = np.array(points[row_i, col_j])
            fourvec = np.array([curr_point[0], curr_point[1], curr_point[2], 1.0])
            homogenous_pt = projection_matrix.dot(fourvec)
            projections[row_i, col_j] = np.array(
                [homogenous_pt[0] / homogenous_pt[2], homogenous_pt[1] / homogenous_pt[2]])

    return projections
```

2.2.2 preprocess_ncc_impl

该函数为 ZNCC (Zero-mean Normalized Cross Correlation)计算做预处理。

图像中每个像素处的 ZNCC 是对以该像素为中心的一小块区域(patch)做以下计算：

$$\mathrm{ZNCC}=\frac{\sum_{x,y}(W_1(x,y)-\overline{W_1})(W_2(x,y)-\overline{W_2})}{\sqrt{\sum_{x,y}(W_1(x,y)-\overline{W_1})^2}\sqrt{\sum_{x,y}(W_2(x,y)-\overline{W_2})^2}}$$

其中$\overline{W}_i=\frac{1}{n}\sum_{x,y}W_i(x,y)$为均值；$W_i(x,y)$是图像 i 中坐标(x,y)处的像素值。

该区域(patch)的大小由 preprocess_ncc_impl 的参数 ncc_size 决定；patch 为 ncc_size*ncc_size 的正方形。

preprocess_ncc_impl 为预处理，即计算一幅图像每个像素点周围 patch 中的值：

$$\frac{W(x,y)-\overline{W}}{\sqrt{\sum_{x,y}(W(x,y)-\overline{W})^2}}$$

当通道数为 channels 时，得到一个长度为 channels * ncc_size* ncc_size 的向量。

求平均时，每个通道单独做。归一化时（即做除法时），对所有通道一起做（即求$\sqrt{\sum_{x,y}(W(x,y)-\overline{W})^2}$是对 channels * ncc_size* ncc_size 个值一起做）。

- 代码实现

```
def preprocess_ncc_impl(image, ncc_size):

    height, width, channels = image.shape
    window_offset = int(ncc_size / 2)
    normalized = np.zeros((height, width, (channels * (ncc_size ** 2))))#
    for row_i in range(window_offset, height - window_offset):
        for col_k in range(window_offset, width - window_offset):
            patch_vector = image[row_i - window_offset:row_i + window_offset + 1,

                                 col_k - window_offset:col_k + window_offset + 1, :]
```

```
            mean_vec = np.mean(np.mean(patch_vector, axis=0), axis=0)#
            patch_vector = patch_vector - mean_vec

            temp_vec = np.zeros((channels * (ncc_size ** 2)))#

            big_index = 0

            for channel in range(channels):
                for row in range(patch_vector.shape[0]):
                    for col in range(patch_vector.shape[1]):
                        temp_vec[big_index] = patch_vector[row, col, channel]
                        big_index += 1

            patch_vector = temp_vec
            if (np.linalg.norm(patch_vector) >= 1e-6):#
                patch_vector /= np.linalg.norm(patch_vector)
            else:#
                patch_vector = np.zeros((channels * ncc_size ** 2))

            normalized[row_i, col_k] = patch_vector

    return normalized
```

2.2.3 compute_ncc_impl

- 对 preprocess_ncc_impl 得到的 patch 向量，将两幅图中每个像素处的两个patch 向量做内积（即对应点相乘并求和）。
- 代码实现

```
def compute_ncc_impl(image1, image2):
    """

    Compute normalized cross correlation between two images that already have
    normalized vectors computed for each pixel with preprocess_ncc.

    Input:
        image1 -- height x width x (channels * ncc_size**2) array
        image2 -- height x width x (channels * ncc_size**2) array
    Output:
        ncc -- height x width normalized cross correlation between image1 and
               image2.
    """
    height, width = image1.shape[:2]
    ncc = np.zeros((height, width))

    for row_i in range(height):
        for col_k in range(width):
            ncc[row_i, col_k] = np.correlate(image1[row_i, col_k], image2[row_i, col_k])

    return ncc
```

2.3基于泊松方程重建深度图

泊松方程根据法线方向计算深度（详见讲义第 18 讲：光度测量立体视觉）。每个点处的两个方程为：

$$n_x = n_z z_{x+1,y} - n_z z_{x,y}$$
$$-n_y = n_z z_{x,y+1} - n_z z_{x,y}$$

其中(n_x, n_y, n_z)为该点处的法向量(即法线方向)，z 为深度。注：此处法向量沿+x、+y、-z 轴为正。

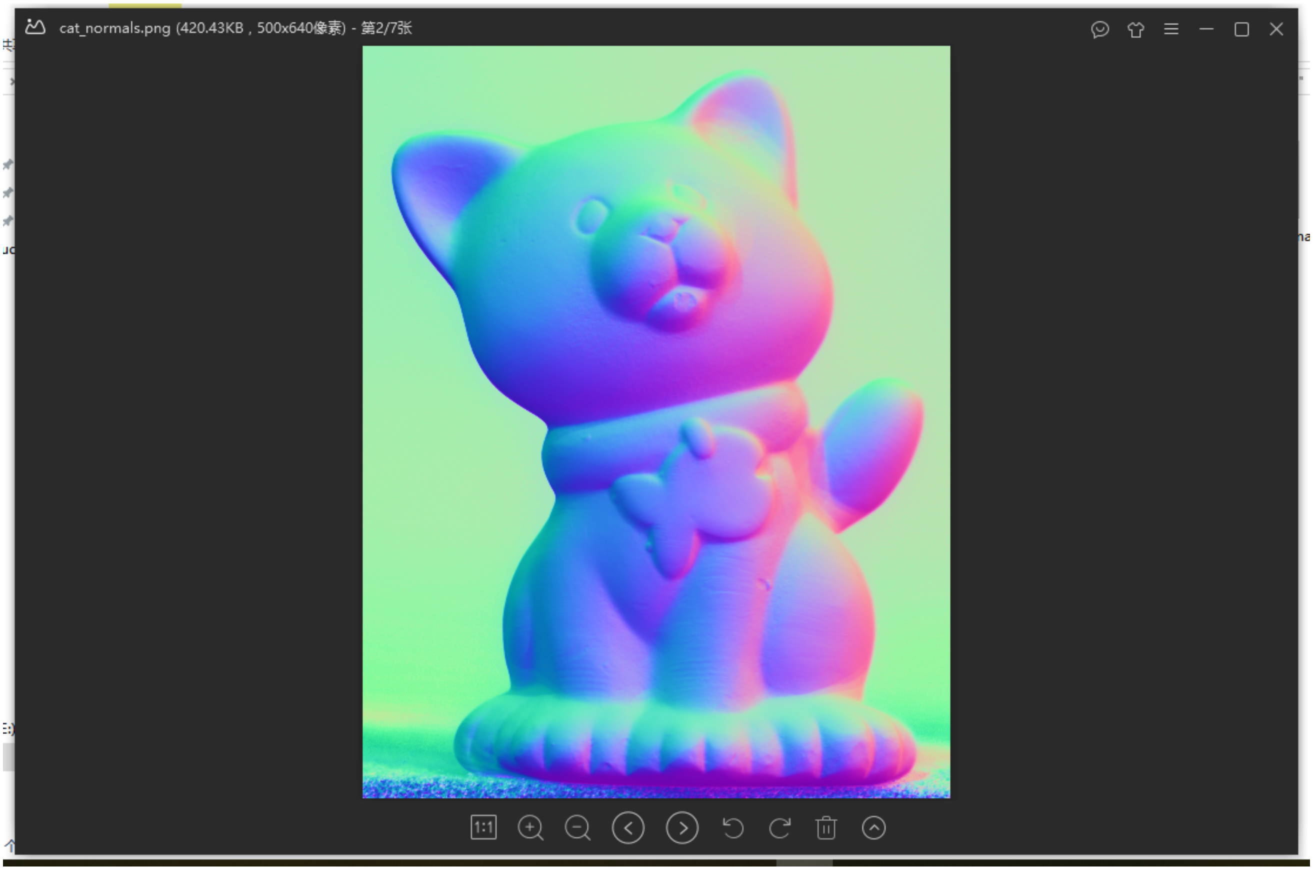
你需要实现的 form_poisson_equation_impl 函数返回线性方程 Ax=b 的参数；其中 x 为所有点的深度，x 为 height*width 大小的向量，是未知数；A 和 b 是要返回的参数。之后会使用最小二乘法求解该方程得到每个点的深度。

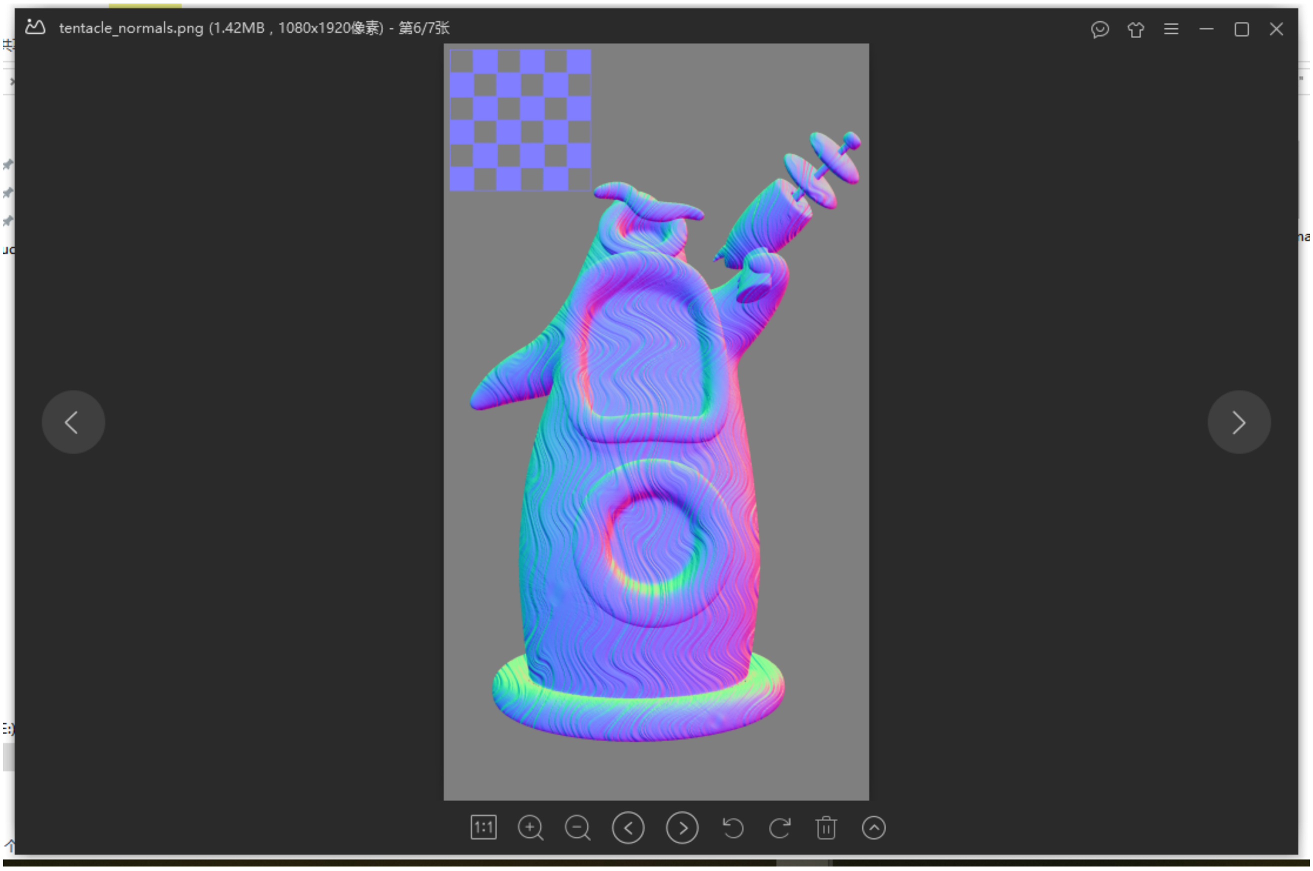
- 代码实现

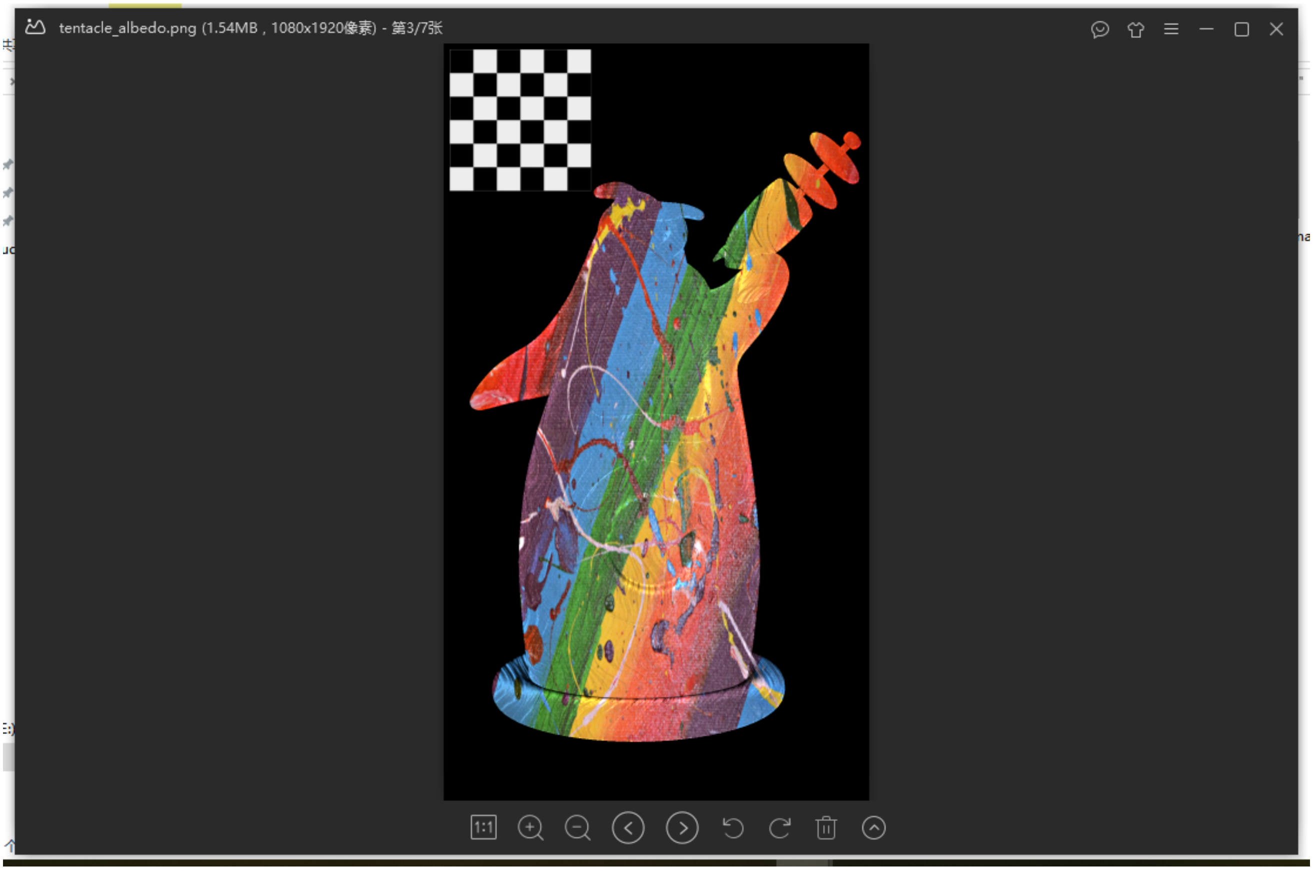
```python
    rn = 0
    for row_i in range(height):
        for col_j in range(width):
            k = row_i * width + col_j
            if alpha[row_i, col_j] != 0:
                if depth is not None:
                    b.append(depth_weight * depth[row_i, col_j])  # depth
                    row_ind.append(rn)  # depth
                    col_ind.append(k)  # depth
                    data_arr.append(depth_weight)  # depth
                    rn += 1

                if normals is not None:
                    if col_j + 1 <= width - 1 and alpha[row_i, col_j + 1] != 0:
                        # normals x-axis
                        b.append(normals[row_i, col_j, 0])
                        row_ind.append(rn)
                        col_ind.append(k)
                        data_arr.append(-normals[row_i, col_j, 2])
                        row_ind.append(rn)
                        col_ind.append(k + 1)
                        data_arr.append(normals[row_i, col_j, 2])
                        rn += 1
                    if row_i + 1 <= height - 1 and alpha[row_i + 1, col_j] != 0:
                        # normals mode y-axis
                        b.append(-normals[row_i, col_j, 1])
                        row_ind.append(rn)
                        col_ind.append(k)
                        data_arr.append(-normals[row_i, col_j, 2])
                        row_ind.append(rn)
                        col_ind.append(k + width)
                        data_arr.append(normals[row_i, col_j, 2])
                        rn += 1
    row = rn
```

三.实验结果

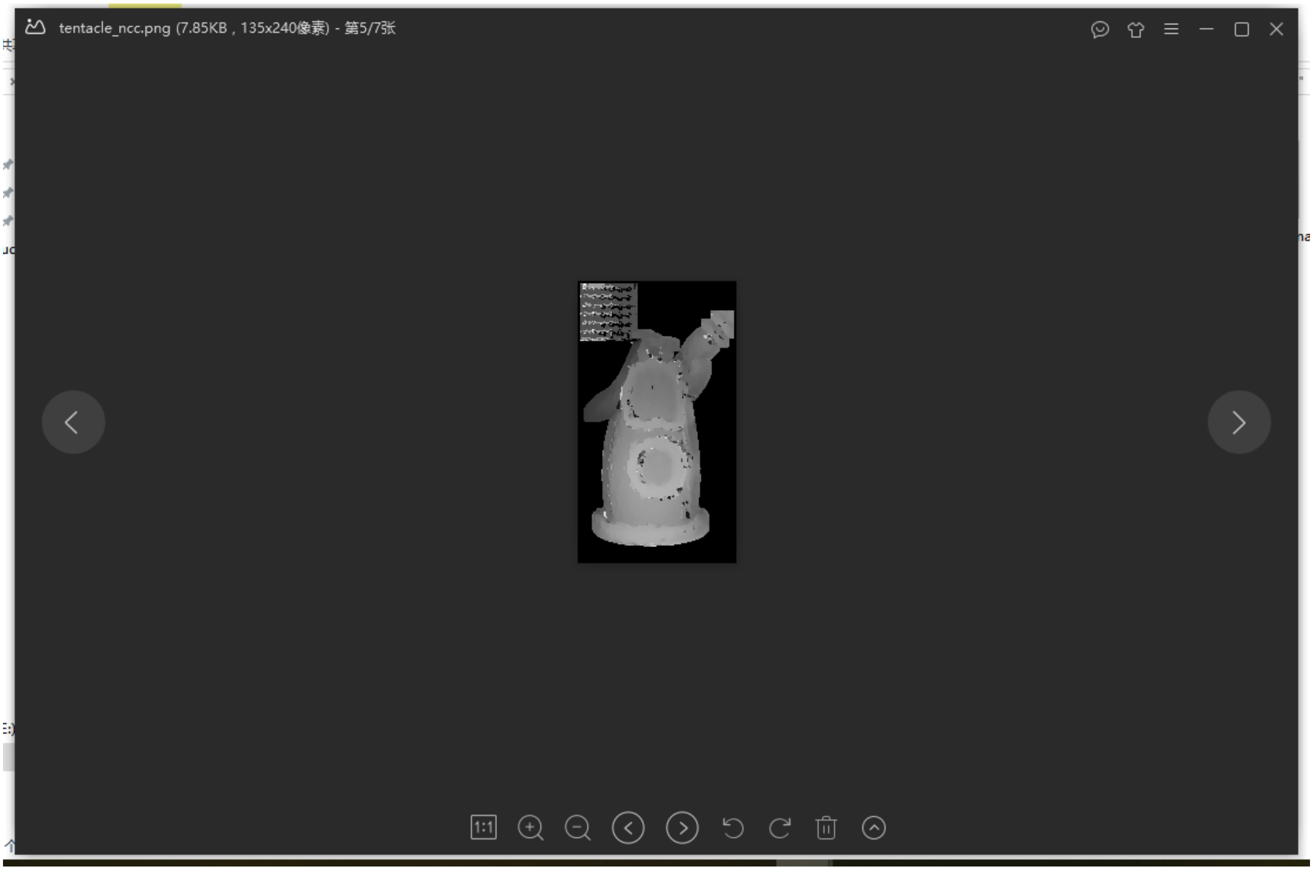
3.1光度测量立体视觉

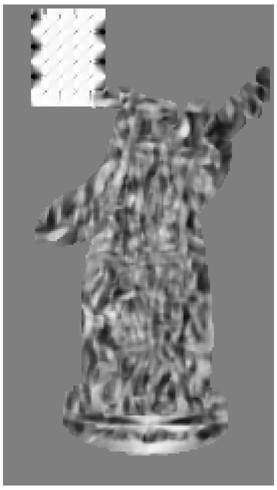


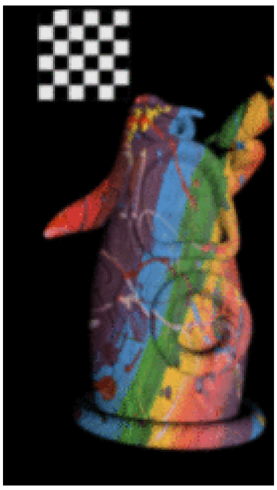


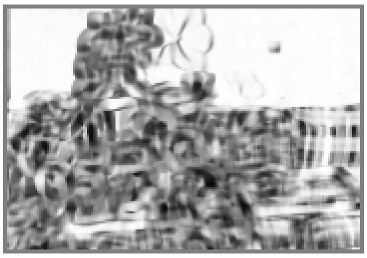


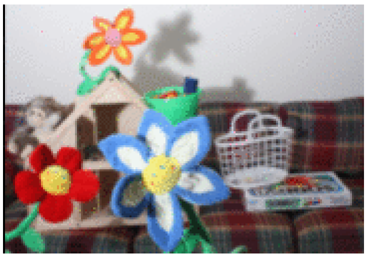
3.2平面扫描立体视觉


tentacle_ncc.png (7.85KB , 135x240像素) - 第5/7张

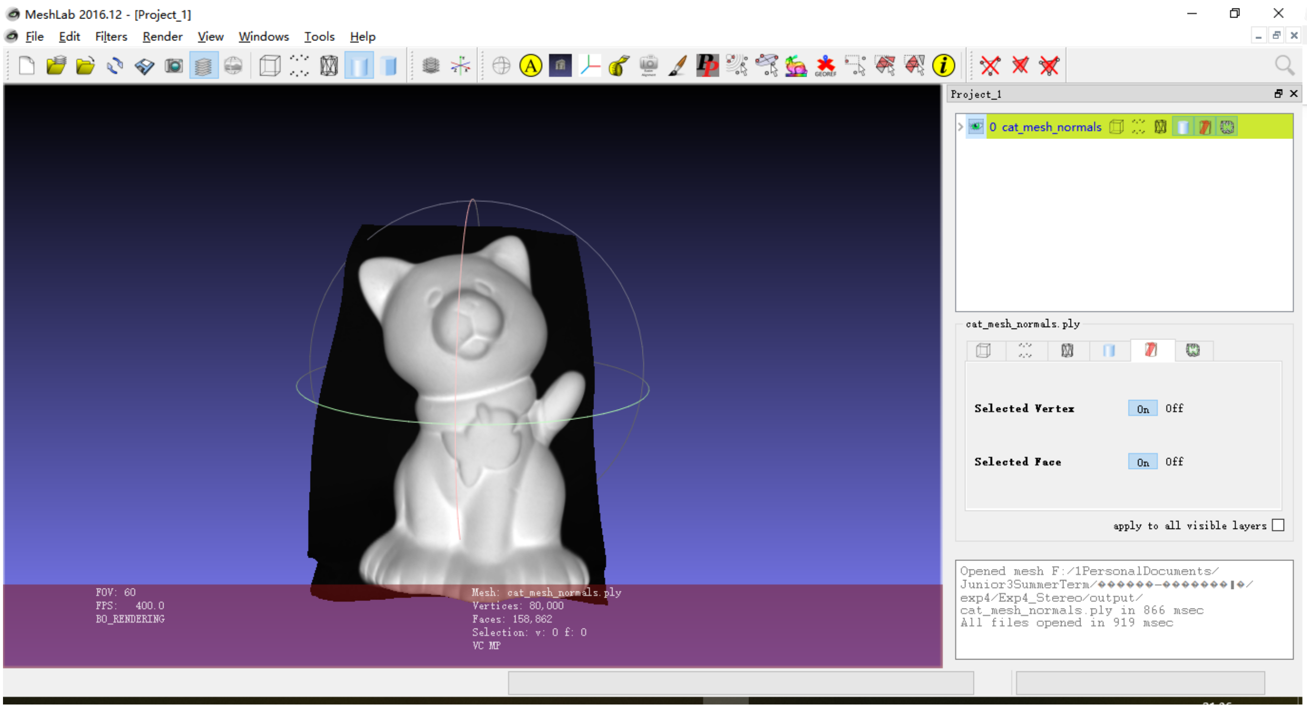


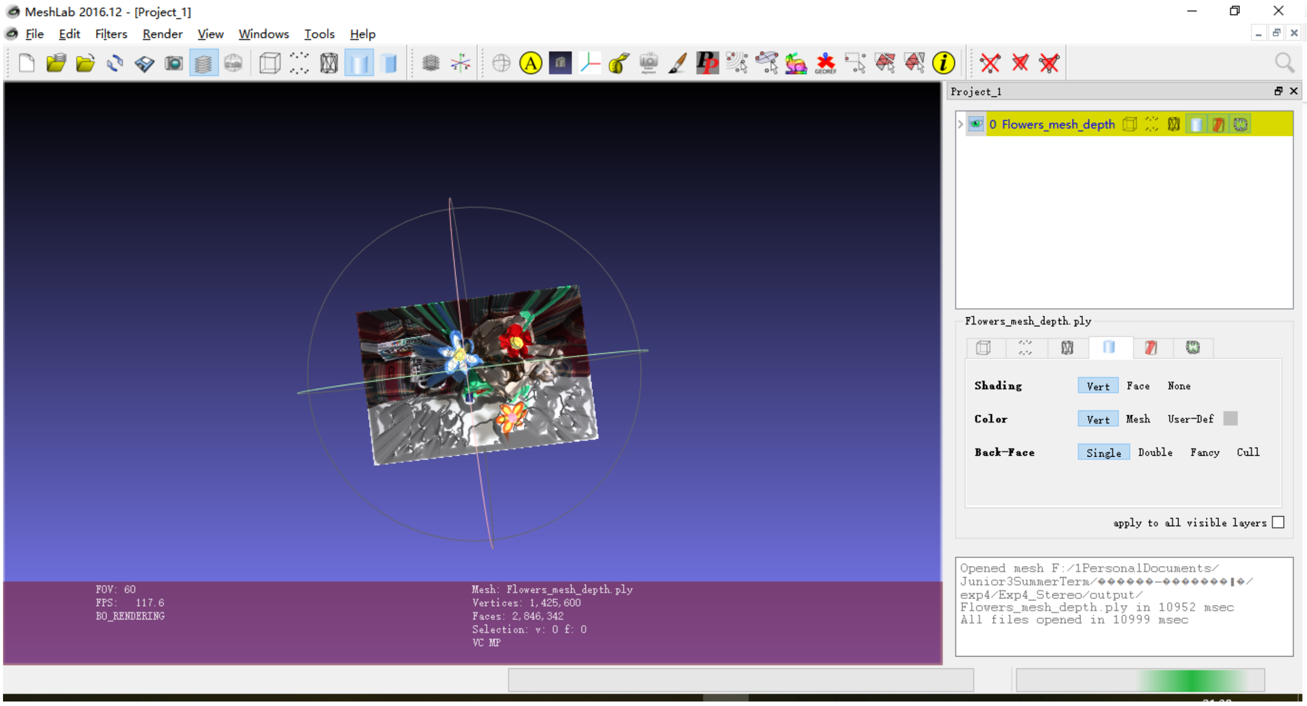


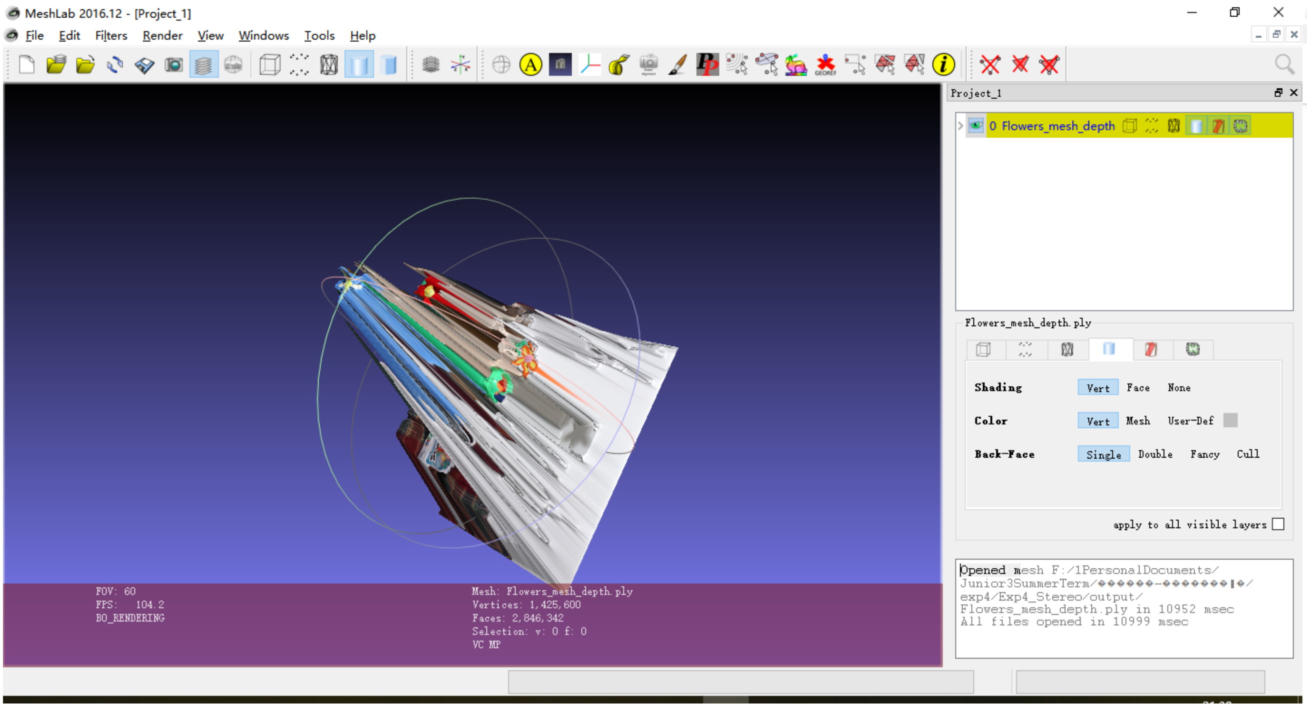




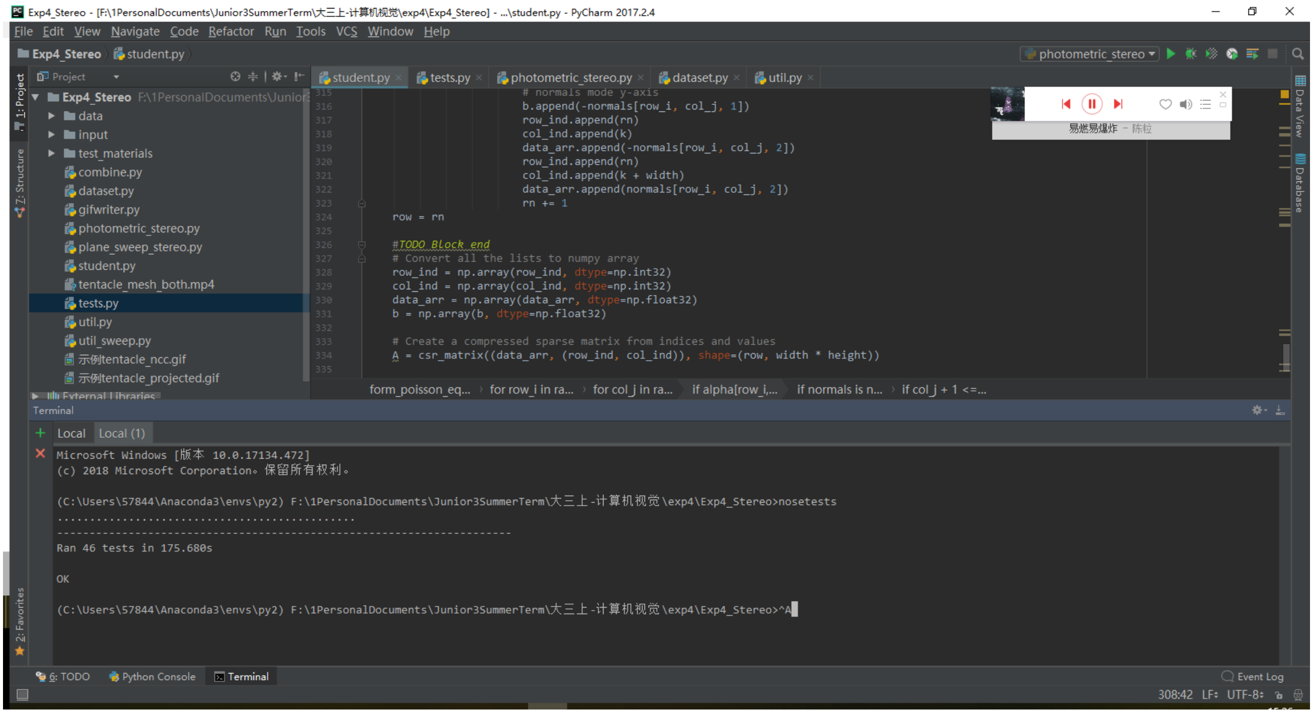
3.3基于泊松方程重建深度图

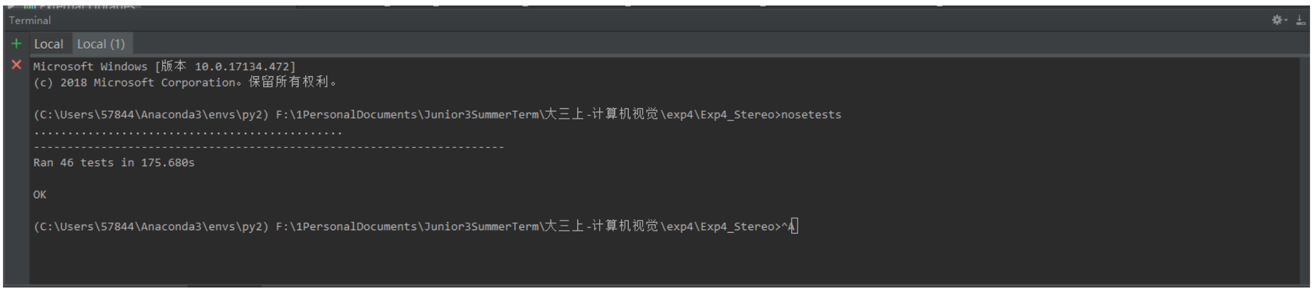



MeshLab 2016.12 - [Project_1]
File Edit Filters Render View Windows Tools Help
Project_1
0 Flowers_mesh_depth
Flowers_mesh_depth.ply
Shading Vert Face None
Color Vert Mesh User-Def
Back-Face Single Double Fancy Cull
apply to all visible layers
Opened mesh F:/1PersonalDocuments/
Junior3SummerTerm/
exp4/Exp4_Stereo/output/
Flowers_mesh_depth.ply in 10952 msec
All files opened in 10999 msec
FOV: 60
FPS: 117.6
BO_RENDERING
Mesh: Flowers_mesh_depth.ply
Vertices: 1,425,600
Faces: 2,846,342
Selection: v: 0 f: 0
VC MP


MeshLab 2016.12 - [Project_1]
File Edit Filters Render View Windows Tools Help
Project_1
0 Flowers_mesh_depth
Flowers_mesh_depth.ply
Shading Vert Face None
Color Vert Mesh User-Def
Back-Face Single Double Fancy Cull
apply to all visible layers
Opened mesh F:/1PersonalDocuments/
Junior3SummerTerm/
exp4/Exp4_Stereo/output/
Flowers_mesh_depth.ply in 10952 msec
All files opened in 10999 msec
FOV: 60
FPS: 104.2
BO_RENDERING
Mesh: Flowers_mesh_depth.ply
Vertices: 1,425,600
Faces: 2,846,342
Selection: v: 0 f: 0
VC MP

3.4Tests结果





四.实验心得

通过本次实验，了解了如果通过三种方法对3D的物体，进行：

- 深度图重建
- 利用平面扫描建立立体视觉
- 通过基于泊松方程重建深度图

  通过实践代码，进一步深入了解了，平面扫描立体视觉，光度立体视觉，以及基于泊松方程的立体视觉深度图重建。