THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SINGLE IMAGE BASED AUTOMATIC PORTRAIT PHOTOGRAPHY FACE RELIGHTING FRAMEWORK

KUN WANG
SPRING 2018

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Engineering
with honors in Computer Engineering

Reviewed and approved* by the following:

James Z. Wang
Professor of Information Sciences and Technology
Thesis Supervisor

John Sampson
Assistant Professor of Computer Science and Engineering
Honors Adviser

* Signatures are on file in the Schreyer Honors College.

ABSTRACT

This thesis research proposes a novel intelligent face relighting framework to help amateur photographers take high-quality portraits. Face relighting is a challenging problem when there is only a single image of the face available. To solve this problem, we often assume that human faces obey Lambert’s law. Recovering the shape, reflectance and illuminance of a face is essential to relight the face. The proposed framework can decompose a single face image into a normal map (shape), an albedo map (reflectance), and lighting coefficients (illuminance). This framework incorporates the state-of-the-art face landmark detection algorithm, the 3D morphable face model fitting framework, and the spherical-harmonics-base face albedo estimation algorithm to relight a face from a single image. Our proposed framework first detects the face landmarks in a portrait. Then, it fits a 3D morphable face model to the face with detected landmark positions. Next, the framework calculates the vertex normal vectors of the reconstructed 3D model and renders the normal map. In order to obtain the albedo map and illumination coefficients, it solves a linear equation system iteratively. Finally, with normal map and albedo map, the framework allows the user to apply different portrait lighting styles to the input image by changing the illumination coefficients, which are learned from artistic portrait photos. When the input face is under a general smooth lighting, the experiment results are realistic and reliable under the subjective evaluation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Acknowledgements

To Prof. James Z. Wang, my thesis supervisor, for his guidance in my thesis research.

To Prof. John Sampson for his assistance as my honors advisor.

To graduate student Farshid Farhat for his guidance and help.

Chapter 1

Introduction

Taking a high-quality portrait image is of great interest to amateur photographers and ordinary people, who mainly take photos on smartphones. Allowing them to manipulate the illumination of portraits after shooting can help them improve portrait photos taken under varying lighting conditions. In the computer vision and graphics community, relighting face images is an important but challenging problem. Among previous works, learning the texture and geometry of a face from a single image is the most essential and difficult part. One approach to this problem is decomposing a real-world face into its intrinsic components, including albedo and surface normal, by training an end-to-end generative adversarial network [3]. However, their network can only output low-resolution images (128x128), which cannot meet the need for high-quality portrait images. In addition, it is hard to port this GPU-based deep learning algorithm to smartphones or other platforms due to their limited computing power. Another approach is fitting a 3D morphable face model to a face image and estimating the intrinsic components [1]. Our framework is mainly inspired by this approach, but the original method mainly focuses on relighting gray-scale faces to improve the accuracy of face recognition algorithms regardless of the authenticity of relit faces. Therefore, I optimize this approach to obtain high-quality relighted RGB faces.

In this thesis research, I develop our framework based on the state-of-the-art face landmark detection algorithm [11] and the latest lightweight 3D morphable face model fitting library [8]. This framework supports the state-of-the-art 3D morphable face model, Surrey Face

Model (SFM), and does face pose, shape and expression fitting. Furthermore, our framework can be easily ported to any other platforms such as Android and IOS because this framework is written as C++ header files. With the reconstructed 3D face model, the framework calculates and renders the normal map as shown in Figure 1(c). For the albedo estimation, I propose a new algorithm by combining a spherical-harmonics-based algorithm [9] with the one using the median filter [10]. This new algorithm can reduce albedo errors but also keep face texture as shown in Figure 1(b).

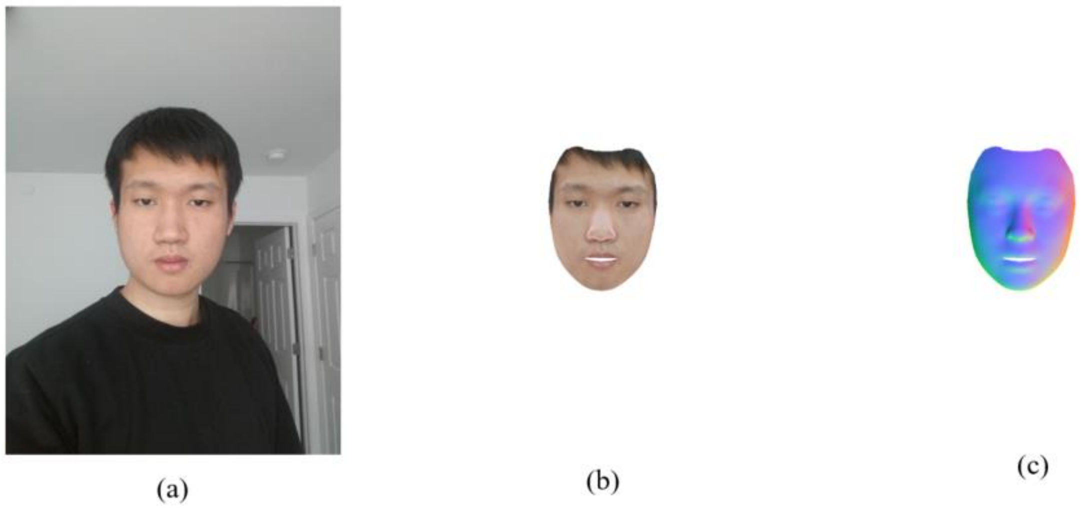


Figure 1. Original Image (a), Albedo (b) and Surface Normal Map (c)

As for the relighting part, Barsi and Jacobs [5] have proved that the reflectance function of a convex Lambertian object can be accurately estimated by a low-dimensional linear subspace with a spherical-harmonics representation. It has been used in recent face relighting methods [1], [2], [3]. However, these methods mainly focus on transferring the illumination condition from a reference image to a target image. As a result, these methods require the user to find a reference image before relighting a target image. Nevertheless, it is difficult for ordinary people to find the reference images of different lighting styles without learning professional photography. To the best of my knowledge, no work has been down on learning spherical illumination coefficients for different portrait lighting styles. Thus, our framework estimates the illumination coefficients of portrait photos with different lighting styles and embed these coefficients in it.

I will start with a brief discussion about background information of spherical harmonics representation and 3D morphable face model Fitting in Chapter 2. Then, I will focus on the implementation details of face albedo estimation in Chapter 3 and the face relighting algorithm in Chapter 4. I conclude my thesis in Chapter 5.

Chapter 2

Background Information

In this chapter, I will briefly introduce the Lambertian reflection model and describe the details of approximating the illumination coefficients by using Spherical Harmonics Representation. Then, I will focus on the background information of the 3D morphable face model and the model fitting library developed by Huber et al. [8].

2.1 Lambertian Reflection Model

Lambertian reflection model is a widely used illumination model for calculating the brightness value of diffuse reflection surfaces. If we assume there is only one light source, and photons are evenly distributed to its surroundings, the surface will have the diffuse reflection when those photons reach and are reflected from a rough surface. It is also called Lambertian Reflection, and the surface is called the Lambertian surface. Spherical Harmonics Representation is based on the Lambertian reflection. Most of the previous works in face relighting also assume a human face is a Lambertian surface.

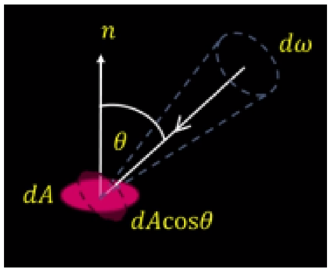
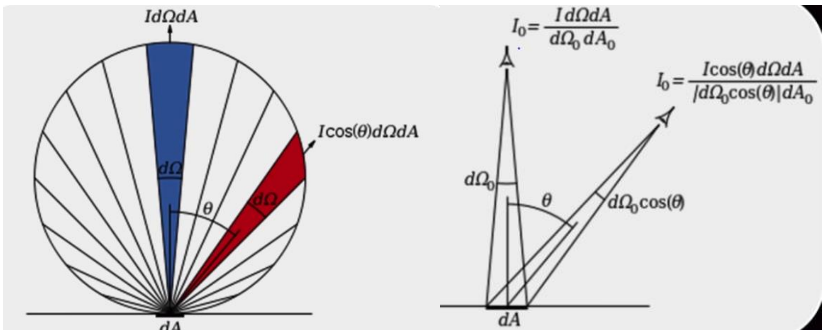


Figure 2. Irradiance Calculation [17]

When we calculate the brightness, or radiation exitance E(θ, φ), of an infinitesimal surface receiving radiance L(θ, φ) as shown in Figure 2, we use the following equation [17]:

$$E(\theta,\varphi) = L(\theta,\varphi)cos\theta d\omega$$

where θ is the angle between the normal vector of the surface and the incident ray, φ is the rotation angle of incident ray and $d\omega$ is the area of the light source patch.



$Id\Omega dA$
$I\cos(\theta)d\Omega dA$
$d\Omega$
$d\Omega$
θ
dA
$I_0 = \frac{Id\Omega dA}{d\Omega_0 dA_0}$
$I_0 = \frac{I\cos(\theta)d\Omega dA}{|d\Omega_0\cos(\theta)|dA_0}$
$d\Omega_0$
θ
$d\Omega_0\cos(\theta)$
dA

Figure 3. Lambertian Reflection Property [17]

For a Lambertian surface, the surface appears equally bright from all directions. As shown in Figure 3, although the light coming out of the small patch is not the same in all directions, the irradiance is the same in all directions by the following equation [17]:

$$\frac{Id\Omega dA}{Id\Omega_0 dA_0} = \frac{Icos(\theta)d\Omega dA}{d\Omega_0 cos(\theta)dA_0}$$

where I is the energy of the lighting coming out, θ is the angle between the normal vector of the surface and the incident ray, $d\Omega_0$ is the area of the light source patch and dA_0 is the area of the small reflection surface.

2.2 Spherical Harmonics Representation

Spherical harmonics are a set of orthogonal functions that can be used to represent the functions defined on the surface of a sphere. It is the sphere analog of the Fourier bases in the 2-dimensional space [1].

In the spherical harmonics representation, assuming the object surface is a Lambertian surface, according to Basri and Jacobs [5], the irradiance E of this surface will be:

$$\mathrm{E}\ (\mathrm{n}_i) = \int_{\Omega} L(\theta, \varphi) cos\theta\ d\Omega$$

where n_i is the normal vector (n_x, n_y, n_z) of a point on the sphere surface, L is radiance, and Ω is the integral interval, the episphere.

Basri and Jacobs [5] have also demonstrated that illumination conditions of a convex Lambertian object, which is under a variety of distant isotropic lights, can be accurately estimated by the first nine spherical harmonic bases (the first and second order spherical harmonics). Therefore, the above equation is rewritten by Basri and Jacobs as:

$$\mathrm{E}\ (\mathrm{n}_i) = \sum_{l=0}^{2} \sum_{m=-l}^{l} \alpha_l^m B_l^m(i)$$

where α_l^m represents the illumination coefficient, and $B_l^m(i)$ represents the spherical harmonics bases of pixel i from a face image.

The first nine spherical harmonic bases are described below from [1]:

$$B_0^0 = \frac{1}{\sqrt{4\pi}} \qquad B_1^0 = \frac{2\pi}{3}\sqrt{\frac{3}{4\pi}} * n_z \qquad B_1^{-1} = \frac{2\pi}{3}\sqrt{\frac{3}{4\pi}} * n_y$$

$$B_1^1 = \frac{2\pi}{3}\sqrt{\frac{3}{4\pi}} * n_x \qquad B_2^0 = \frac{\pi}{8}\sqrt{\frac{3}{4\pi}} * (3n_z^2 - 1)$$

$$B_2^{-1} = \frac{3\pi}{4}\sqrt{\frac{5}{12\pi}} * n_y n_z \qquad B_2^1 = \frac{3\pi}{4}\sqrt{\frac{5}{12\pi}} * n_x n_z$$

$$B_2^{-2} = \frac{3\pi}{4}\sqrt{\frac{5}{12\pi}} * n_x n_y \quad B_2^{-2} = \frac{3\pi}{8}\sqrt{\frac{5}{12\pi}} * (n_x^2 - n_y^2)$$

where n_x, n_y, n_z represents the x, y and z components of the surface normal vector.

According to Wang et al. [1], assuming under the general illumination conditions, the spherical harmonics can approximately represent any face images as a linear combination of these nine spherical harmonic bases:

$$I = \rho B^T \cdot \alpha$$

where I is the image intensity value vector of the size p, the number of pixels, ρ is the albedo vector of the size p, B is a (9 x p) spherical harmonic bases matrix and α represents the 9-dimensional illumination coefficients vector. I will focus on a proposed algorithm solving this linear equation system to obtain the albedo and the lighting coefficients in Chapter 3.

2.3 3D Morphable Face Models

In general, reconstructing 3D face model from a single image is a challenging problem due to the lack of geometry information of the face. To solve this problem, Blanz et al. proposed the first 3D morphable face model [12]. To obtain a reliable and representative model, they make a number of 3D face scans from different age and racial groups. These 3D scan data are brought in correspondence by using a registration algorithm, which aligns the corresponding points of scanned faces such as mouth corners and other face landmarks. After establishing the morphable model, a 3D-to-2D model fitting algorithm is needed. However, the most of fitting algorithms are very complex and slow, and the current high-resolution face models are not practical in real-world applications. Thus, Huber et al. proposed their lightweight multi-resolution Surrey Face Model and a header-only model fitting library [8]. Our framework is built upon the low-resolution Surrey Face Model and this library to efficiently relight a face in less than ten seconds.

2.4 PCA Model and New Face Generation

The Surrey Face Model is built on 3D face meshes that are registered to a reference mesh in dense correspondence [8]. There are m faces in the Surrey Face Model. These m faces are a spanning set of a linear subspace. This subspace can be represented as a matrix $S = (s_1, \dots, s_m) \in \mathbb{R}^{3n \times m}$, where s_i contains the Cartesian coordinates (x, y, z components) of vertices on a face mesh and n is the number of vertices on the mesh. Huber et al. [8] do a principal component analysis on the 3D face scan data. The Surrey Face Model contains two PCA models: one for the shape and one for the texture information. The two models are similar. The shape of any novel faces can be generated by [8]:

$$s_{new} = \bar{s} + \sum_{i}^{M} \alpha_i \sigma_i v_i$$

where $\bar{s}$ is the mean of all 3D face samples in the face model, M is the number of principal components, α_i is the instance coordinate of a 3D face sample in the PCA space, and σ_i is the standard deviation of a principal component. Some sample 3D faces generated from the model are shown in Figure 4.

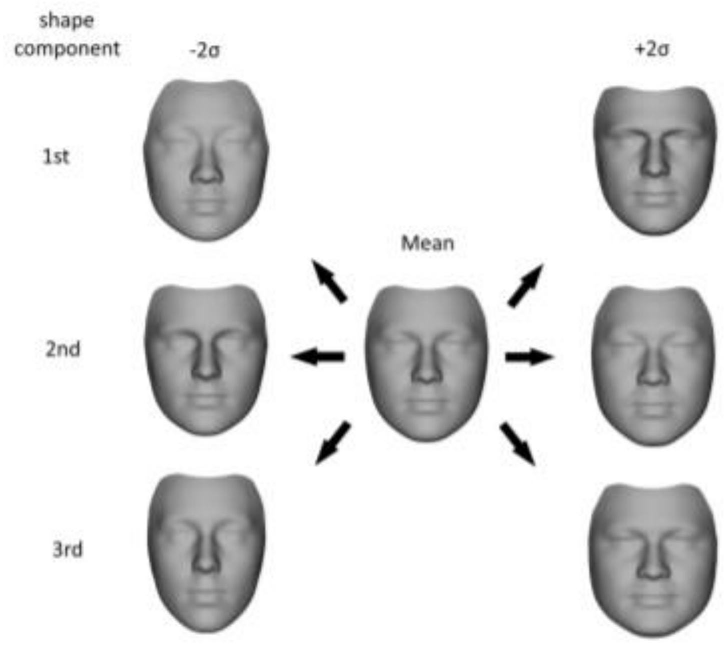


Figure 4. The Mean Face and Shape Variation of The Surrey Face Model [8]

2.5 Lightweight 3D Morphable Face Model Fitting Library

Huber et al. also provide a lightweight 3D face morphable model fitting library written in C++ head files without any other dependency [8]. If we want to build an application, we only need to link the program to OpenCV core library to process input and output images. The library also uses a low-resolution Surrey Face Model, which is more practical and efficient for developing an application.

The library defines a “MorphableModel” class, which contains two “PCAModel” subclass instances for shape and color models respectively. The PCA model class consists of a mean, the PCA basis vectors and eigenvalues. Given the standard 68 face landmark positions, this library can linearly approximate the pose, shape [13], expression and fit the face model to edges [14] accordingly. A sample 3D-to-2D fitting result and a 3D reconstruction result are shown in Figure 5.

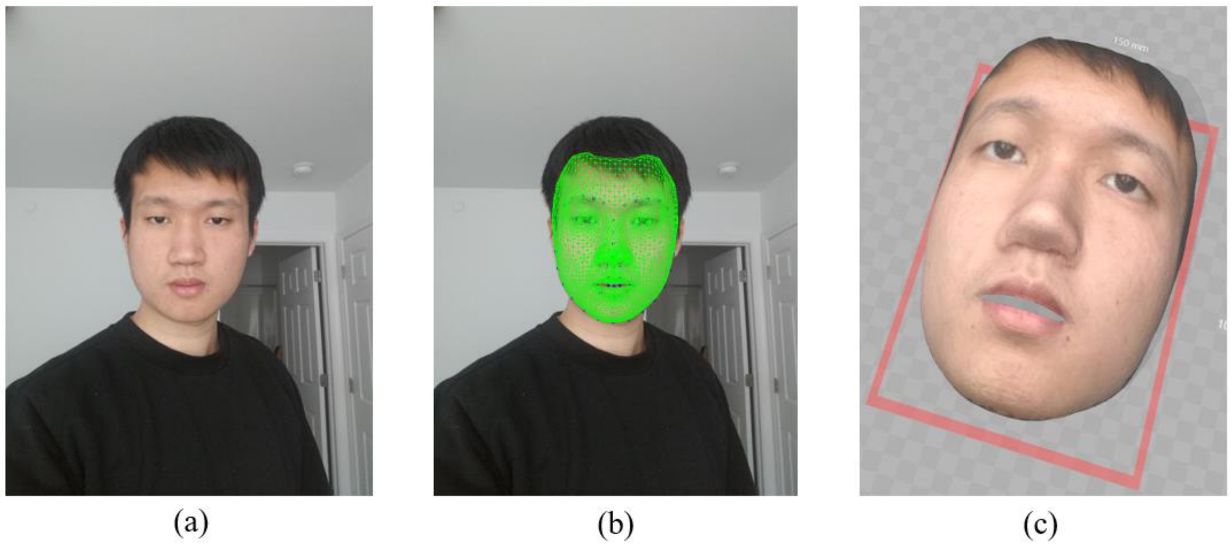


Figure 5. Input Image (a), 3D Mesh Fit (b), and Reconstructed 3D Face (c)

Chapter 3

Face Normal and Albedo Map Estimation

In this chapter, I will focus on my two main contributions, face normal and albedo map estimation algorithms, to the lightweight 3D morphable face model fitting library.

3.1 Face Normal Map Estimation and Render

Beyond only fitting the 3D morphable face model to a 2D image, I improve the aforementioned fitting library by adding algorithms that calculate the normal vector of each vertex on the mesh and render the normal map smoothly. Before calculating the normal vector of each vertex, the normal vectors of mesh surfaces are required. The cross product of two vectors on a triangle mesh surface is the normal vector of a surface. Divided the result vector by the length of it, I will get the unit normal vector of a surface:

$$n = \frac{\overrightarrow{\mathrm{AB}} \times \overrightarrow{\mathrm{AC}}}{\left\|\overrightarrow{\mathrm{AB}} \times \overrightarrow{\mathrm{AC}}\right\|}$$

where n is the normal vector of the surface, A, B and C are the three vertices of the triangle, $\overrightarrow{\mathrm{AB}}$ is the vector pointing from vertex A to vertex B, and $\overrightarrow{\mathrm{AC}}$ is the vector pointing from vertex A to vertex C.

Taking into account that some faces share the same vertex, the algorithm iterates through all triangle surfaces on the mesh, calculates the normal vector of each triangle, adds the surface normal vector to the normal vector of each vertex of the triangle (initialized as zero vector) accumulatively and normalizes the result vector:

$$v = \text{normalize}(\sum_{i=1}^{m} n_i)$$

where v is the normal vector of the vertex, m is the number of triangle surfaces sharing the vertex, and n_i is the surface normal vector.

After obtaining the surface normal vector of each vertex, our framework will render the normal map in affine camera model, which projects the 3D model vertices to image pixels. The RGB values are x component, y component and z component of the vertex normal respectively. An example normal map of the face is shown in Figure 1(c).

3.2 Face Albedo Map Estimation

Previous work [1] estimates the texture information of the face by fitting the trained PCA face texture model to the face. However, unlike the shape of the face, it is difficult to build a realistic PCA face texture model because different people have distinct face textures. Thus, the face textures calculated from PCA model cannot be used in the portrait face relighting framework. Some unrealistic relighting results using PCA face texture model are shown in Figure 6 [1].

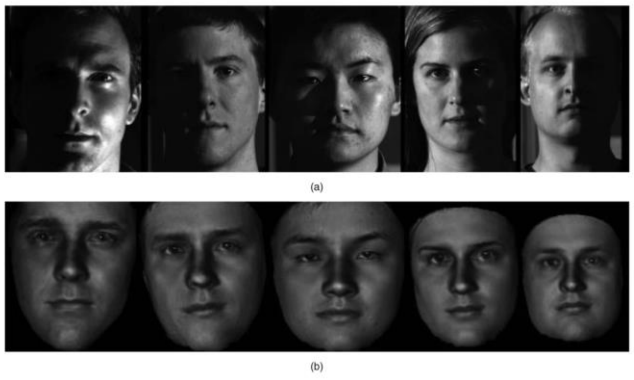


Figure 6. Face Relighting Experiment Results From [1]. Input (a) and Relight Results

Under the Lambertian object assumption, the bidirectional reflectance distribution function (BRDF) of a human face is constant, called albedo. Albedo is the surface reflectance, which can represent the realistic face texture information. Recent researchers have worked on robust face albedo estimation algorithm. Xuan Zou et al. formulates the albedo estimation problem as a linear programming by using the spherical harmonics [9]. Sungho Suh et al. proposed another similar estimation method by applying a median filter to the face image and using dynamic parameters when solving the linear problem [10]. However, the result of spherical harmonics method [9] cannot compensate the estimation errors, and the median-filter algorithm [10] requires the ground truth depth information, which cannot be obtained from a single image. In this thesis research, I propose a new robust albedo estimation algorithm combining the main contributions from the above two algorithms:

Algorithm 1. The outline of the proposed albedo estimation algorithm based on [9] [10]:

1. Set the average intensity of the face area on the image as the initial albedo map ρ^0 [9]

2. Estimate the initial illumination coefficient vector L^0 by solving the following least square estimation [9]:

$$\arg\min_{\mathrm{L}^0} \| I - \rho^0 B \mathrm{L}^0 \|^2$$

   where I is the intensity value matrix of the face, B is the spherical harmonic bases matrix of the size $(p \times 9)$ for one of three channels of the RGB image (see details of how to calculate it in Chapter 2), and p is the number of pixels on the face.

3. Solve the albedo map $\widehat{\rho^m}$ at the mth iteration by calculating $\widehat{\rho^m} = \frac{I}{B\mathrm{L}^{m-1}}$. Solve the current albedo map ρ^m by using a parameterized combination of the previous estimation ρ^{m-1} and the current approximation $\widehat{\rho^m}$ [10]:

$$\rho^m = \alpha\, \widehat{\rho^m} + (1-\alpha)\, \rho^{m-1}$$

where α is a parameter following the constraints below [10] :

Albedos are range from 0 to 1 and have a scale ambiguity, so Suh, Sungho, et al. [10] assume the albedo is near 0.5. Thus, we set a parameter e to 0.75. If the albedo value is larger than 0.75, we compensate albedo estimation errors by setting a smaller α coefficient:

$$\alpha = \begin{cases} 1, & \text{if } \rho_i^m < \mathrm{e} \\ 1 - \dfrac{\rho_i^m - e}{1 - e}, & \text{if } \rho_i^m \geq \mathrm{e} \end{cases}$$

4. Solve the least square estimation above with the albedo map ρ^m to obtain the new illumination coefficient vector L^m:

$$\arg\min_{\mathrm{L}^m} \| I - \rho^m B \mathrm{L}^m \|^2$$

Repeat steps 3-4 for at least two times. The termination criterion may be satisfying. Our proposed algorithm does not require image depth information that is needed by [10]. An example albedo estimation is shown in Figure 1(b).

Chapter 4

Learning Illumination Coefficients and Face Relighting

In this chapter, I will first discuss the process of learning illumination coefficients and present some images of my face relit with different lighting styles.

4.1 Learning Illumination Coefficients from Artistic Portrait Photos and Relighting

I collect a series of artistic portrait photos with different lighting styles and learn their illumination coefficients. The artistic portrait photos are shown in Figure 7:

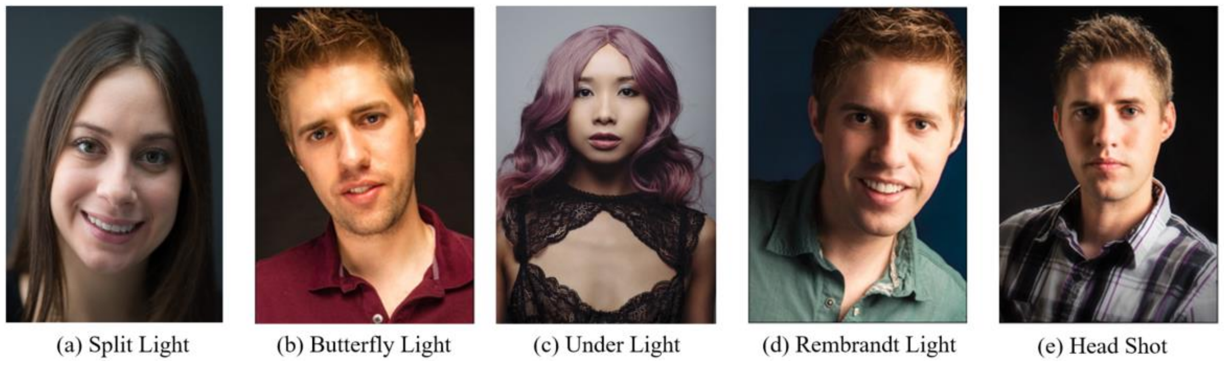


Figure 7. Artistic Portrait Photos for Learning Illumination Coefficients. [15], [18]

Based on our proposed albedo algorithm in section 3.2, I can estimate the illumination coefficients from a single image. The estimated illumination coefficients will be a matrix of size (3 x 9), and each row corresponds to one of the RGB channels. The estimated illumination coefficients of the "Under Light" portrait photo (Figure 7(c)) is shown in Table 1.

	1	2	3	4	5	6	7	8	9
R	-0.85282	1.49631	1.3213	-1.65496	-2.78762	-0.16684	0.894721	1.87503	1.66253
G	-0.63636	1.27988	0.89171	-1.28572	-2.49575	0.192114	0.714314	1.8723	1.4596
B	0.2892	0.537756	0.658405	-0.34559	-1.34802	0.140037	0.580759	1.04124	0.445016

Table 1. Illumination Coefficients of the "Under Light" Portrait Photo (Figure 7(c)).

To relight the input face with the illumination coefficients of artistic portrait photos, L, the framework solves the following the equation:

$$I = \rho_e B \mathrm{L}$$

where I is the intensity values of the relit face, ρ_e is the estimated albedo using our proposed algorithm, and B is the spherical harmonics bases matrix of the input face. Some example results are shown in the next section.

4.2 Experiment Results

I did experiments on an image of my own face with different lighting styles, and the results are shown below:

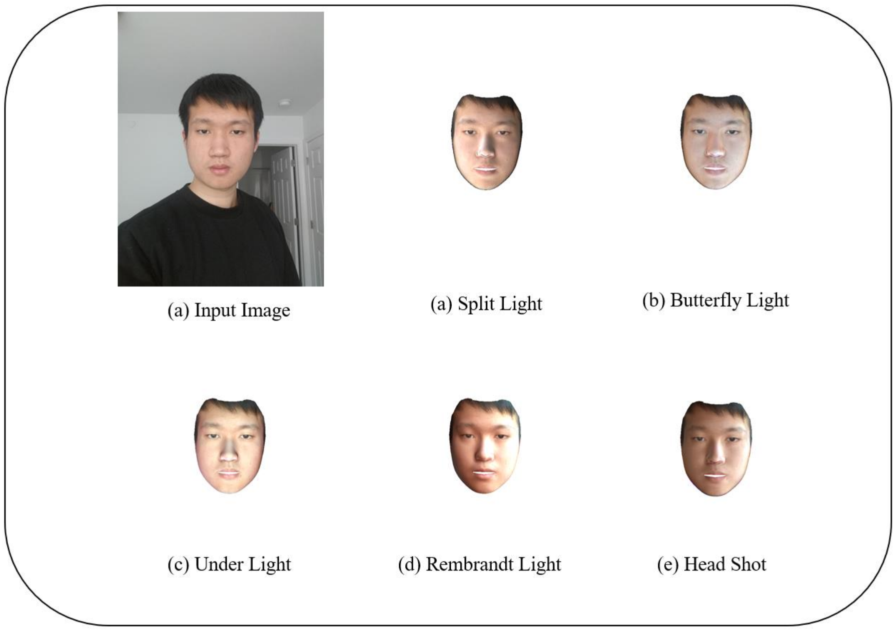


Figure 8. Relit Images of My Face with Different Lighting Styles

The ground truth of a relit face is difficult to be obtained, so we subjectively evaluate our experiment results in this thesis. As shown in Figure 8, relit faces are photo-realistic under a subjective evaluation.

I also tested the framework on images from the Photoface database [19], which contains faces with cast shadow under the harsh light. However, our framework fails to generate realistic relit faces when the input face with cast shadow. As shown in Figure 8, there are some artifacts around the nose and the boundary of the face, but the overall relighting effect is acceptable.:

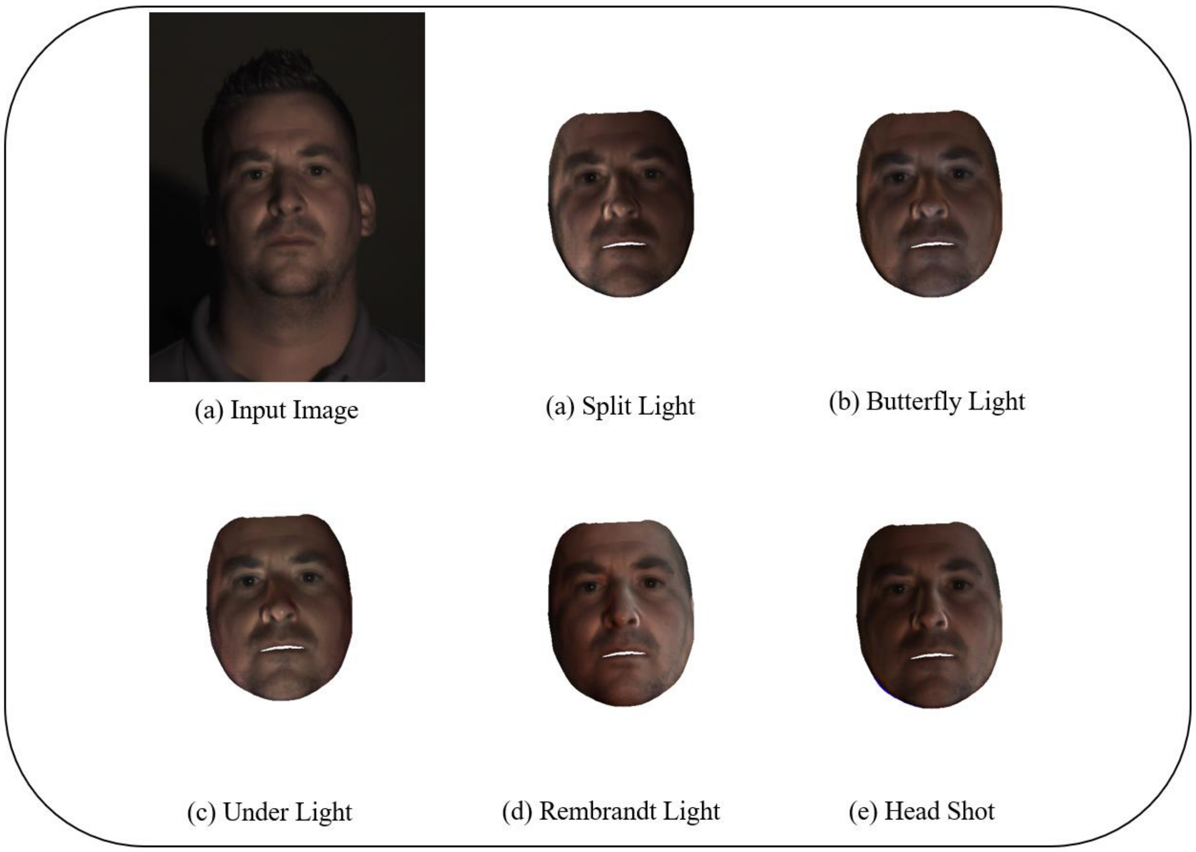


Figure 9. Relit Images of a Face from the Photoface Database [19].

To remove these artifacts, we might further improve our albedo estimation algorithm in the future.

Chapter 5

Conclusions

In this thesis, I propose a face relighting framework for portrait photos by recovering the normal map, albedo map and illumination coefficients of a human face from a single image. The proposed method includes four parts. In the first part, I reconstruct 3D face by using the lightweight 3D morphable face model library [8], which can do a fast pose, shape, and expression 3D-to-2D face fitting. Secondly, in order to obtain spherical harmonics representation of faces, I add a new face normal map calculation and render algorithm to this framework. After representing the face in spherical harmonics, I estimate the albedo map and illumination coefficients of the face by using the proposed albedo estimation algorithm in section 3.2. By using this algorithm, the illumination coefficients from artistic portrait photos are obtained. Finally, by applying the learned coefficients to an input face, I relight the face with the corresponding portrait lighting style. I also demonstrated the performance of my framework by doing experiments on images of my face and the faces from the Photoface database [19] with different lighting styles. In the future, I plan to develop an Android application by using this relighting framework so that the user can relight faces immediately after taking a photo.

Appendix A

Source Code

A.1 Face Normal Map and Albedo Estimation Function:

```
inline std::pair<core::Image4u, core::Image4u> render_normal_affine(const core::Mesh&
mesh, eos::core::Image3u& Intensity, Eigen::Matrix<float, 3, 4> affine_camera_matrix,
int viewport_width, int viewport_height, Eigen::VectorXf& NewRL, Eigen::VectorXf&
NewGL, Eigen::VectorXf NewBL, bool do_backface_culling = true)
{
    assert(mesh.vertices.size() == mesh.colors.size() ||
           mesh.colors.empty()); // The number of vertices has to be equal for both
shape and colour, or,
    using namespace Eigen;
    using eos::core::Image1d;
    using eos::core::Image;
    using eos::core::Image4u;
    using std::vector;

    MatrixXf SHR(viewport_height*viewport_width, 9);
    MatrixXf SHG(viewport_height*viewport_width, 9);
    MatrixXf SHB(viewport_height*viewport_width, 9);
    MatrixXf SH(viewport_height*viewport_width, 9);
    VectorXf IR(viewport_height*viewport_width);
    VectorXf IG(viewport_height*viewport_width);
    VectorXf IB(viewport_height*viewport_width);
    Image4u relightbuffer(viewport_height, viewport_width);
    SHR.setZero();
    SHG.setZero();
    SHB.setZero();
    IR.setZero();
    IB.setZero();
    IG.setZero();
    SH.setZero();


    Image<std::array<double, 4>, 4> colourbuffer(
        viewport_height,
        viewport_width);
    Image1d depthbuffer(viewport_height, viewport_width);
    std::for_each(std::begin(depthbuffer.data), std::end(depthbuffer.data),
                  [](auto& element) { element = std::numeric_limits<double>::max();
});

    Image1d xbuffer(viewport_height, viewport_width);
    std::for_each(std::begin(xbuffer.data), std::end(xbuffer.data),
                  [](auto& element) { element = std::numeric_limits<double>::max();
});

    Image1d ybuffer(viewport_height, viewport_width);
    std::for_each(std::begin(ybuffer.data), std::end(ybuffer.data),
                  [](auto& element) { element = std::numeric_limits<double>::max();
});

    Image1d fbuffer(viewport_height, viewport_width);
    std::for_each(std::begin(fbuffer.data), std::end(fbuffer.data),
                  [](auto& element) { element = -1; });


    const Eigen::Matrix<float, 4, 4> affine_with_z =
        detail::calculate_affine_z_direction(affine_camera_matrix);


    vector<detail::Vertex<float>> projected_vertices;
    projected_vertices.reserve(mesh.vertices.size());
    for (int i = 0; i < mesh.vertices.size(); ++i)
    {
        const Eigen::Vector4f vertex_screen_coords =
            affine_with_z *
            Eigen::Vector4f(mesh.vertices[i][0], mesh.vertices[i][1],
mesh.vertices[i][2], 1.0f);
        const glm::tvec4<float> vertex_screen_coords_glm(vertex_screen_coords(0),
vertex_screen_coords(1),
                                                         vertex_screen_coords(2),
vertex_screen_coords(3));
        glm::tvec3<float> vertex_colour;
        vertex_colour = glm::tvec3<float>(0, 0, 0);
        projected_vertices.push_back(
            detail::Vertex<float>{vertex_screen_coords_glm, vertex_colour,
                                  glm::tvec2<float>(mesh.texcoords[i][0],
mesh.texcoords[i][1])});
    }

    // All vertices are screen-coordinates now
    vector<detail::TriangleToRasterize> triangles_to_raster;
    for (const auto& tri_indices : mesh.tvi)
    {
        if (do_backface_culling)
        {
            if (!detail::are_vertices_ccw_in_screen_space(
                    glm::tvec2<float>(projected_vertices[tri_indices[0]].position),
                    glm::tvec2<float>(projected_vertices[tri_indices[1]].position),
                    glm::tvec2<float>(projected_vertices[tri_indices[2]].position)))
                continue; // don't render this triangle
        }

        const glm::tvec3<float> v1 =
glm::tvec3<float>(projected_vertices[tri_indices[0]].position);
        const glm::tvec3<float> v2 =
glm::tvec3<float>(projected_vertices[tri_indices[1]].position);
```

```
        const glm::tvec3<float> v3 =
glm::tvec3<float>(projected_vertices[tri_indices[2]].position);

        const glm::tvec3<float> e1 = glm::normalize(v2 - v1);
        const glm::tvec3<float> e2 = glm::normalize(v3 - v1);
        const glm::tvec3<float> e3 = glm::normalize(v2 - v3);

        const float angle1 = glm::acos(e1[0]*e2[0]+e1[1]*e2[1]+e1[2]*e2[2]);
        const float angle2 = glm::acos(e1[0]*e3[0]+e1[1]*e3[1]+e1[2]*e3[2]);
        const float angle3 = PI - angle1 - angle2;


        const Vector4f v0_as_Vector4f(projected_vertices[tri_indices[0]].position[0],
                                      projected_vertices[tri_indices[0]].position[1],
                                      projected_vertices[tri_indices[0]].position[2],
1.0f);
        const Vector4f v1_as_Vector4f(projected_vertices[tri_indices[1]].position[0],
                                      projected_vertices[tri_indices[1]].position[1],
                                      projected_vertices[tri_indices[1]].position[2],
1.0f);
        const Vector4f v2_as_Vector4f(projected_vertices[tri_indices[2]].position[0],
                                      projected_vertices[tri_indices[2]].position[1],
                                      projected_vertices[tri_indices[2]].position[2],
1.0f);

        const Vector3f face_normal =
                compute_face_normal(v0_as_Vector4f, v1_as_Vector4f,
v2_as_Vector4f);

        projected_vertices[tri_indices[0]].color =
projected_vertices[tri_indices[0]].color + glm::tvec3<float>(face_normal[0]*angle1,
face_normal[1]*angle1, face_normal[2]*angle1);
        projected_vertices[tri_indices[1]].color =
projected_vertices[tri_indices[1]].color + glm::tvec3<float>(face_normal[0]*angle2,
face_normal[1]*angle2, face_normal[2]*angle2);
        projected_vertices[tri_indices[2]].color =
projected_vertices[tri_indices[2]].color + glm::tvec3<float>(face_normal[0]*angle3,
face_normal[1]*angle3, face_normal[2]*angle3);


    }

    for (const auto& tri_indices : mesh.tvi)
    {
        if (do_backface_culling)
        {
            if (!detail::are_vertices_ccw_in_screen_space(
                    glm::tvec2<float>(projected_vertices[tri_indices[0]].position),
                    glm::tvec2<float>(projected_vertices[tri_indices[1]].position),
                    glm::tvec2<float>(projected_vertices[tri_indices[2]].position)))
                continue; // don't render this triangle
        }

        // Get the bounding box of the triangle:
```

```
        // take care: What do we do if all 3 vertices are not visible. Seems to work
on a test case.

        const Rect<int> bounding_box = detail::calculate_clipped_bounding_box(
            glm::tvec2<float>(projected_vertices[tri_indices[0]].position),
            glm::tvec2<float>(projected_vertices[tri_indices[1]].position),
            glm::tvec2<float>(projected_vertices[tri_indices[2]].position),
viewport_width, viewport_height);
        const auto min_x = bounding_box.x;
        const auto max_x = bounding_box.x + bounding_box.width;
        const auto min_y = bounding_box.y;
        const auto max_y = bounding_box.y + bounding_box.height;

        if (max_x <= min_x || max_y <= min_y) // Note: Can the width/height of the
bbox be negative? Maybe we only need to check for equality here?
            continue;

        detail::TriangleToRasterize t;
        t.min_x = min_x;
        t.max_x = max_x;
        t.min_y = min_y;
        t.max_y = max_y;
        t.v0 = projected_vertices[tri_indices[0]];
        t.v1 = projected_vertices[tri_indices[1]];
        t.v2 = projected_vertices[tri_indices[2]];
        t.vm0 = mesh.vertices[tri_indices[0]];
        t.vm1 = mesh.vertices[tri_indices[1]];
        t.vm2 = mesh.vertices[tri_indices[2]];
        t.vi0 = tri_indices[0];
        t.vi1 = tri_indices[1];
        t.vi2 = tri_indices[2];

        triangles_to_raster.push_back(t);

    }

    std::vector<double> max(3);
    //max.reserve(3);
    std::vector<double> min(3);
    //min.reserve(3);
    //std::map<int> tri_set;


    for (int i =0; i<3; i++)
    {
        //std::cout<<"max:"<<max[i]<<std::endl;
        //std::cout<<"min:"<<min[i]<<std::endl;
        min[i] = std::numeric_limits<double>::max();
    }

    // Raster all triangles, i.e. colour the pixel values and write the z-buffer
    //for (auto&& triangle : triangles_to_raster)
    for (int i=0; i < triangles_to_raster.size(); i++)
    {
        //detail::raster_triangle_normal_affine(triangle, colourbuffer, depthbuffer, 
xbuffer, ybuffer, fbuffer, wbuffer1, wbuffer2, wbuffer3);
        detail::raster_triangle_normal_affine(triangles_to_raster[i], colourbuffer, 
depthbuffer, xbuffer, ybuffer, fbuffer, i, max, min);

    }
    for (int i =0; i<3; i++)
    {
        std::cout<<"max:"<<max[i]<<std::endl;
        std::cout<<"min:"<<min[i]<<std::endl;
    }
    for (int i =0; i<viewport_height; i++){
        for(int j=0; j<viewport_width; j++){
            if(xbuffer(i,j) == std::numeric_limits<double>::max())
                xbuffer(i,j) = 0;
            else{
                xbuffer(i,j) = xbuffer(i,j) - min[0];
                xbuffer(i,j) = xbuffer(i,j)/max[0];
                //xbuffer(i,j) = xbuffer(i,j)*255;
            }

            if(ybuffer(i,j) == std::numeric_limits<double>::max())
                ybuffer(i,j) = 0;
            else{
                ybuffer(i,j) = ybuffer(i,j) - min[1];
                ybuffer(i,j) = ybuffer(i,j)/max[1];
                //ybuffer(i,j) = ybuffer(i,j)*255;
            }

            if(depthbuffer(i,j) == std::numeric_limits<double>::max())
                depthbuffer(i,j) = 0;
            else{
                depthbuffer(i,j) = depthbuffer(i,j) - min[2];
                depthbuffer(i,j) = depthbuffer(i,j)/max[2];
                depthbuffer(i,j) = depthbuffer(i,j)*255;
            }

        }
    }
    double R_total = 0;
    double G_total = 0;
    double B_total = 0;

    int count = 0;
    for (int i=0; i < viewport_height; i++){
        for(int j=0; j<viewport_width; j++){
            if(fbuffer(i, j) != -1){
                R_total += Intensity(i, j)[0];
                G_total += Intensity(i, j)[1];
                B_total += Intensity(i, j)[2];
                count ++;
                IR(i*viewport_width+j) = Intensity(i, j)[0]/255.0f;
                IG(i*viewport_width+j) = Intensity(i, j)[1]/255.0f;
                IB(i*viewport_width+j) = Intensity(i, j)[2]/255.0f;
            }
        }
    }
    //estimate albedo RGB by taking mean of Intensity
    double R_albedo = R_total/count/255.0f;
    double G_albedo = G_total/count/255.0f;
    double B_albedo = B_total/count/255.0f;
    for (int i=0; i < viewport_height; i++){
        for(int j=0; j<viewport_width; j++){
            if(fbuffer(i, j) != -1){
                int k = fbuffer(i, j);
                float nx = colourbuffer(i,j)[0], ny = colourbuffer(i,j)[1], nz =
colourbuffer(i,j)[2];
                SHR((i*viewport_width+j), 0) = 0.282094792 * R_albedo;
                SHR((i*viewport_width+j), 1) = 1.02332671 * nz * R_albedo;
                SHR((i*viewport_width+j), 2) = 1.02332671 * ny * R_albedo;
                SHR((i*viewport_width+j), 3) = 1.02332671 * nx * R_albedo;
                SHR((i*viewport_width+j), 4) = 0.247707956 * (3*nz*nz - 1) *
R_albedo;
                SHR((i*viewport_width+j), 5) = 0.858085531 * (ny * nz) * R_albedo;
                SHR((i*viewport_width+j), 6) = 0.858085531 * (nx * nz) * R_albedo;
                SHR((i*viewport_width+j), 7) = 0.858085531 * (nx * ny) * R_albedo;
                SHR((i*viewport_width+j), 8) = 0.429042765 * (nx*nx - ny*ny) *
R_albedo;

                SHG((i*viewport_width+j), 0) = 0.282094792 * G_albedo;
                SHG((i*viewport_width+j), 1) = 1.02332671 * nz * G_albedo;
                SHG((i*viewport_width+j), 2) = 1.02332671 * ny * G_albedo;
                SHG((i*viewport_width+j), 3) = 1.02332671 * nx * G_albedo;
                SHG((i*viewport_width+j), 4) = 0.247707956 * (3*nz*nz - 1) *
G_albedo;
                SHG((i*viewport_width+j), 5) = 0.858085531 * (ny * nz) * G_albedo;
                SHG((i*viewport_width+j), 6) = 0.858085531 * (nx * nz) * G_albedo;
                SHG((i*viewport_width+j), 7) = 0.858085531 * (nx * ny) * G_albedo;
                SHG((i*viewport_width+j), 8) = 0.429042765 * (nx*nx - ny*ny) *
G_albedo;

                SHB((i*viewport_width+j), 0) = 0.282094792 * B_albedo;
                SHB((i*viewport_width+j), 1) = 1.02332671 * nz * B_albedo;
                SHB((i*viewport_width+j), 2) = 1.02332671 * ny * B_albedo;
                SHB((i*viewport_width+j), 3) = 1.02332671 * nx * B_albedo;
                SHB((i*viewport_width+j), 4) = 0.247707956 * (3*nz*nz - 1) *
B_albedo;
                SHB((i*viewport_width+j), 5) = 0.858085531 * (ny * nz) * B_albedo;
                SHB((i*viewport_width+j), 6) = 0.858085531 * (nx * nz) * B_albedo;
                SHB((i*viewport_width+j), 7) = 0.858085531 * (nx * ny) * B_albedo;
                SHB((i*viewport_width+j), 8) = 0.429042765 * (nx*nx - ny*ny) *
B_albedo;

                SH((i*viewport_width+j), 0) = 0.282094792;
                SH((i*viewport_width+j), 1) = 1.02332671 * nz;
                SH((i*viewport_width+j), 2) = 1.02332671 * ny;
                SH((i*viewport_width+j), 3) = 1.02332671 * nx;
                SH((i*viewport_width+j), 4) = 0.247707956 * (3*nz*nz - 1);
                SH((i*viewport_width+j), 5) = 0.858085531 * (ny * nz);
                SH((i*viewport_width+j), 6) = 0.858085531 * (nx * nz);
                SH((i*viewport_width+j), 7) = 0.858085531 * (nx * ny);
                SH((i*viewport_width+j), 8) = 0.429042765 * (nx*nx - ny*ny);
            }
        }
    }
    Image<std::array<double, 4>, 4> albedo1(viewport_height, viewport_width);
    for (int i=0; i < viewport_height; i++){
        for(int j=0; j<viewport_width; j++){
            if(fbuffer(i, j) != -1){
                albedo1(i,j)[0] = R_albedo;

                albedo1(i,j)[1] = G_albedo;

                albedo1(i,j)[2] = B_albedo;

                albedo1(i,j)[3] = 255;
            }
        }
    }
    Image4u albedo(viewport_height, viewport_width);
    Image1d albedoG(viewport_height, viewport_width);
    Image1d relightOutB(viewport_height, viewport_width);
    VectorXf RL = SHR.colPivHouseholderQr().solve(IR);
    VectorXf GL = SHG.colPivHouseholderQr().solve(IG);
    VectorXf BL = SHB.colPivHouseholderQr().solve(IB);


    VectorXf RHL = SH * RL;
    VectorXf GHL = SH * GL;
    VectorXf BHL = SH * BL;

    double c1 = 0.75;
    double c2 = 0.75;
    for (int k = 0; k < 2; k++){
        for (int i=0; i < viewport_height; i++){
            for(int j=0; j<viewport_width; j++){
                if(fbuffer(i, j) != -1){
                    double a0 = std::max(std::min(IR(i*viewport_width+j) /
RHL(i*viewport_width+j), 1.0f), 0.0f);
                    double alpha = 1 - (a0 - c1)/(1 - c1);
                    if(a0 > c1){
                        albedo1(i,j)[0] = (alpha*a0 + (1-alpha)*albedo1(i,j)[0]);
                    }
                    else{
                        albedo1(i,j)[0] = a0;
                    }
                    double a1 = std::max(std::min(IG(i*viewport_width+j) /
GHL(i*viewport_width+j), 1.0f), 0.0f);
                    alpha = 1 - (a1 - c1)/(1 - c1);
                    if(a1 > c1){
                        albedo1(i,j)[1] = (alpha*a1 + (1-alpha)*albedo1(i,j)[1]);
                    }
                    else{
                        albedo1(i,j)[1] = a1;
                    }
                    double a2 = std::max(std::min(IB(i*viewport_width+j) / 
BHL(i*viewport_width+j), 1.0f), 0.0f);
                    alpha = 1 - (a2 - c1)/(1 - c1);
                    if(a2 > c1){
                        albedo1(i,j)[2] = (alpha*a2 + (1-alpha)*albedo1(i,j)[2]);
                    }
                    else{
                        albedo1(i,j)[2] = a2;
                    }
                    albedoG(i,j) = IB(i*viewport_width+j) / BHL(i*viewport_width+j) * 
255.0f;

                    albedo1(i,j)[3] = 255;
                }
            }
        }

        for (int i=0; i < viewport_height; i++){
            for(int j=0; j<viewport_width; j++){
                if(fbuffer(i, j) != -1){
                    int k = fbuffer(i, j);
                    float nx = colourbuffer(i,j)[0], ny = colourbuffer(i,j)[1], nz = 
colourbuffer(i,j)[2];
                    SHR((i*viewport_width+j), 0) = 0.282094792 * albedo1(i,j)[0];
                    SHR((i*viewport_width+j), 1) = 1.02332671 * nz * albedo1(i,j)[0];
                    SHR((i*viewport_width+j), 2) = 1.02332671 * ny * albedo1(i,j)[0];
                    SHR((i*viewport_width+j), 3) = 1.02332671 * nx * albedo1(i,j)[0];
                    SHR((i*viewport_width+j), 4) = 0.247707956 * (3*nz*nz - 1) * 
albedo1(i,j)[0];
                    SHR((i*viewport_width+j), 5) = 0.858085531 * (ny * nz) * 
albedo1(i,j)[0];
                    SHR((i*viewport_width+j), 6) = 0.858085531 * (nx * nz) * 
albedo1(i,j)[0];
                    SHR((i*viewport_width+j), 7) = 0.858085531 * (nx * ny) * 
albedo1(i,j)[0];
                    SHR((i*viewport_width+j), 8) = 0.429042765 * (nx*nx - ny*ny) * 
albedo1(i,j)[0];

                    SHG((i*viewport_width+j), 0) = 0.282094792 * albedo1(i,j)[1];
                    SHG((i*viewport_width+j), 1) = 1.02332671 * nz * albedo1(i,j)[1];
                    SHG((i*viewport_width+j), 2) = 1.02332671 * ny * albedo1(i,j)[1];
                    SHG((i*viewport_width+j), 3) = 1.02332671 * nx * albedo1(i,j)[1];
                    SHG((i*viewport_width+j), 4) = 0.247707956 * (3*nz*nz - 1) * 
albedo1(i,j)[1];
                    SHG((i*viewport_width+j), 5) = 0.858085531 * (ny * nz) * 
albedo1(i,j)[1];
                    SHG((i*viewport_width+j), 6) = 0.858085531 * (nx * nz) * 
albedo1(i,j)[1];
                    SHG((i*viewport_width+j), 7) = 0.858085531 * (nx * ny) * 
albedo1(i,j)[1];
                    SHG((i*viewport_width+j), 8) = 0.429042765 * (nx*nx - ny*ny) * 
albedo1(i,j)[1];
```

```
                    SHB((i*viewport_width+j), 0) = 0.282094792 * albedo1(i,j)[2];
                    SHB((i*viewport_width+j), 1) = 1.02332671 * nz * albedo1(i,j)[2];
                    SHB((i*viewport_width+j), 2) = 1.02332671 * ny * albedo1(i,j)[2];
                    SHB((i*viewport_width+j), 3) = 1.02332671 * nx * albedo1(i,j)[2];
                    SHB((i*viewport_width+j), 4) = 0.247707956 * (3*nz*nz - 1) *
albedo1(i,j)[2];
                    SHB((i*viewport_width+j), 5) = 0.858085531 * (ny * nz) *
albedo1(i,j)[2];
                    SHB((i*viewport_width+j), 6) = 0.858085531 * (nx * nz) *
albedo1(i,j)[2];
                    SHB((i*viewport_width+j), 7) = 0.858085531 * (nx * ny) *
albedo1(i,j)[2];
                    SHB((i*viewport_width+j), 8) = 0.429042765 * (nx*nx - ny*ny) *
albedo1(i,j)[2];

                }
            }
        }
        RL = SHR.colPivHouseholderQr().solve(IR);
        GL = SHG.colPivHouseholderQr().solve(IG);
        BL = SHB.colPivHouseholderQr().solve(IB);
        RHL = SH * RL;
        GHL = SH * GL;
        BHL = SH * BL;
    }
    for(int i = 0; i<9; i++){
        std::cout<<RL(i)<<" ";
    }
    std::cout<<std::endl;
    for(int i = 0; i<9; i++){
        std::cout<<GL(i)<<" ";
    }
    std::cout<<std::endl;
    for(int i = 0; i<9; i++){
        std::cout<<BL(i)<<" ";
    }
    std::cout<<std::endl;
    for (int i=0; i < viewport_height; i++){
        for(int j=0; j<viewport_width; j++){
            if(fbuffer(i, j) != -1){
                //std::cout<<(IR(i*viewport_width+j) / RHL(i*viewport_width+j) -
rMin)/rMax<<std::endl;
                double a0 = std::max(std::min(IR(i*viewport_width+j) /
RHL(i*viewport_width+j), 1.0f), 0.0f);
                double alpha = 1 - (a0 - c2)/(1 - c2);
                if(a0 > c2){
                    albedo(i,j)[0] = (alpha*a0 + (1-alpha)*albedo1(i,j)[0]) * 255.0f;
                    albedo1(i,j)[0] = (alpha*a0 + (1-alpha)*albedo1(i,j)[0]);
                }
                else{
                    albedo(i,j)[0] = a0 * 255.0f;
                    albedo1(i,j)[0] = a0;
                }
                double a1 = std::max(std::min(IG(i*viewport_width+j) /
GHL(i*viewport_width+j), 1.0f), 0.0f);
                alpha = 1 - (a1 - c2)/(1 - c2);
                if(a1 > c2){
                    albedo(i,j)[1] = (alpha*a1 + (1-alpha)*albedo1(i,j)[1]) * 255.0f;
                    albedo1(i,j)[1] = (alpha*a1 + (1-alpha)*albedo1(i,j)[1]);
                }
                else{
                    albedo(i,j)[1] = a1 * 255.0f;
                    albedo1(i,j)[1] = a1;
                }
                double a2 = std::max(std::min(IB(i*viewport_width+j) /
BHL(i*viewport_width+j), 1.0f), 0.0f);
                alpha = 1 - (a2 - c2)/(1 - c2);
                if(a2 > c2){
                    albedo(i,j)[2] = (alpha*a2 + (1-alpha)*albedo1(i,j)[2]) * 255.0f;
                    albedo1(i,j)[2] = (alpha*a2 + (1-alpha)*albedo1(i,j)[2]);
                }
                else{
                    albedo(i,j)[2] = a2 * 255.0f;
                    albedo1(i,j)[2] = a2;
                }
                albedoG(i,j) = albedo(i,j)[1];
                albedo(i,j)[3] = 255;
            }
            else{
                albedo(i,j)[0] = 255;
                albedo(i,j)[1] = 255;
                albedo(i,j)[2] = 255;
            }
        }
    }


    VectorXf relightR = SH * NewRL;
    VectorXf relightG = SH * NewGL;
    VectorXf relightB = SH * NewBL;
    for (int i=0; i < viewport_height; i++){
        for(int j=0; j<viewport_width; j++){
            if(fbuffer(i, j) != -1){
                relightbuffer(i,j)[0] = std::min((float)(relightR(i*viewport_width+j)
* albedo1(i,j)[0]), 1.0f) * 255.0f;
                relightbuffer(i,j)[1] = std::min((float)(relightG(i*viewport_width+j)
* albedo1(i,j)[1]), 1.0f) * 255.0f;
                relightbuffer(i,j)[2] = std::min((float)(relightB(i*viewport_width+j)
* albedo1(i,j)[2]), 1.0f) * 255.0f;
                relightOutB(i,j) = relightbuffer(i,j)[1];
                relightbuffer(i,j)[3] = 255;
            }
            else{
                relightbuffer(i,j)[0] = 255;
                relightbuffer(i,j)[1] = 255;
                relightbuffer(i,j)[2] = 255;
                relightbuffer(i,j)[3] = 255;
            }
        }
    }
    return std::make_pair(relightbuffer, albedo);

};
```

A.2 Render Raster Triangle Normal Affine Function:

```
using Eigen::Vector2f;
using Eigen::Vector3f;
using Eigen::Vector4f;
inline void raster_triangle_normal_affine(TriangleToRasterize triangle,
core::Image<std::array<double, 4>, 4>& colourbuffer,
                                   core::Image1d& depthbuffer, core::Image1d&
xbuffer, core::Image1d& ybuffer,
                                   core::Image1d& fbuffer, int i,
                                   std::vector<double>& max, std::vector<double>&
min)
{
    //std::bool flag = false;

    for (int yi = triangle.min_y; yi <= triangle.max_y; ++yi)
    {
        for (int xi = triangle.min_x; xi <= triangle.max_x; ++xi)
        {
            // we want centers of pixels to be used in computations. Todo: Do we?
            const float x = static_cast<float>(xi) + 0.5f;
            const float y = static_cast<float>(yi) + 0.5f;

            // these will be used for barycentric weights computation
            const double one_over_v0ToLine12 =
                1.0 / implicit_line(triangle.v0.position[0], triangle.v0.position[1],
triangle.v1.position,
                                    triangle.v2.position);
            const double one_over_v1ToLine20 =
                1.0 / implicit_line(triangle.v1.position[0], triangle.v1.position[1],
triangle.v2.position,
                                    triangle.v0.position);
            const double one_over_v2ToLine01 =
                1.0 / implicit_line(triangle.v2.position[0], triangle.v2.position[1],
triangle.v0.position,
                                    triangle.v1.position);
            // affine barycentric weights
            const double alpha =
                implicit_line(x, y, triangle.v1.position, triangle.v2.position) *
one_over_v0ToLine12;
            const double beta =
                implicit_line(x, y, triangle.v2.position, triangle.v0.position) *
one_over_v1ToLine20;
            const double gamma =
                implicit_line(x, y, triangle.v0.position, triangle.v1.position) *
one_over_v2ToLine01;
```

```
            // if pixel (x, y) is inside the triangle or on one of its edges
            if (alpha >= 0 && beta >= 0 && gamma >= 0 && alpha+beta <= 1)
            {
                const int pixel_index_row = yi;
                const int pixel_index_col = xi;
                //std::cout<<alpha<<std::endl;
                //std::cout<<beta<<std::endl;
                //std::cout<<gamma<<std::endl;

                const double z_affine = alpha *
static_cast<double>(triangle.v0.position[2]) +
                                        beta *
static_cast<double>(triangle.v1.position[2]) +
                                        gamma *
static_cast<double>(triangle.v2.position[2]);

                const double x_affine = alpha *
static_cast<double>(triangle.v0.position[0]) +
                                        beta *
static_cast<double>(triangle.v1.position[0]) +
                                        gamma *
static_cast<double>(triangle.v2.position[0]);

                const double y_affine = alpha *
static_cast<double>(triangle.v0.position[1]) +
                                        beta *
static_cast<double>(triangle.v1.position[1]) +
                                        gamma *
static_cast<double>(triangle.v2.position[1]);

                if (z_affine < depthbuffer(pixel_index_row, pixel_index_col))
                {
                    // attributes interpolation
                    // pixel_color is in RGB, v.color are RGB
                    glm::tvec3<float> pixel_color = static_cast<float>(alpha) *
glm::normalize(triangle.v0.color) +
                                                    static_cast<float>(beta) *
glm::normalize(triangle.v1.color) +
                                                    static_cast<float>(gamma) *
glm::normalize(triangle.v2.color);


                    const double red =
                        static_cast<double>((std::max(std::min(pixel_color[0], 1.0f),
-1.0f) + 1)/2); // Todo: Proper casting (rounding?)
                    const double green =
                        static_cast<double>((std::max(std::min(pixel_color[1], 1.0f),
-1.0f) + 1)/2);
                    const double blue =
                        static_cast<double>(std::min(std::abs(pixel_color[2]),
1.0f));

                    // update buffers
                    colourbuffer(pixel_index_row, pixel_index_col)[0] = blue;
                    colourbuffer(pixel_index_row, pixel_index_col)[1] = green;
                    colourbuffer(pixel_index_row, pixel_index_col)[2] = red;
                    colourbuffer(pixel_index_row, pixel_index_col)[3] = 255; // alpha
channel
                    depthbuffer(pixel_index_row, pixel_index_col) = z_affine;
                    xbuffer(pixel_index_row, pixel_index_col) = x_affine;
                    ybuffer(pixel_index_row, pixel_index_col) = y_affine;
                    fbuffer(pixel_index_row, pixel_index_col) = i;

                    if(x_affine > max[0]){
                        max[0] = x_affine;
                    }
                    if(y_affine > max[1]){
                        max[1] = y_affine;
                    }
                    if(z_affine > max[2]){
                        max[2] = z_affine;
                    }
                    if(x_affine < min[0]){
                        min[0] = x_affine;
                    }
                    if(y_affine < min[1]){
                        min[1] = y_affine;
                    }
                    if(z_affine < min[2]){
                        min[2] = z_affine;
                    }
                }
            }
        }
    }
};
```

BIBLIOGRAPHY

[1] Wang, Y., Liu, Z., Hua, G., Wen, Z., Zhang, Z., & Samaras, D. (2007). Face Re-Lighting from a Single Image under Harsh Lighting Conditions. *2007 IEEE Conference on Computer Vision and Pattern Recognition*. doi:10.1109/cvpr.2007.383106

[2] Almaddah, A., Vural, S., Mae, Y., Ohara, K., & Arai, T. (2013). Face relighting using discriminative 2D spherical spaces for face recognition. *Machine Vision and Applications,25*(4), 845-857. doi:10.1007/s00138-013-0584-z

[3] Shu, Z., Yumer, E., Hadap, S., Sunkavalli, K., Shechtman, E., & Samaras, D. (2017). Neural Face Editing with Intrinsic Image Disentangling. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr.2017.578

[4] Liao, Z., Karsch, K., & Forsyth, D. (2015). An approximate shading model for object relighting. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr.2015.7299168

[5] Basri, R., & Jacobs, D. (n.d.). Lambertian reflectance and linear subspaces. *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*. doi:10.1109/iccv.2001.937651

[6] Jackson, A. S., Bulat, A., Argyriou, V., & Tzimiropoulos, G. (2017). Large Pose 3D Face Reconstruction from a Single Image via Direct Volumetric CNN Regression. *2017 IEEE International Conference on Computer Vision (ICCV)*. doi:10.1109/iccv.2017.117

[7] Jin, X., Zhao, M., Chen, X., Zhao, Q., & Zhu, S. (2010). Learning Artistic Lighting Template from Portrait Photographs. *Computer Vision – ECCV 2010 Lecture Notes in Computer Science,*101-114. doi:10.1007/978-3-642-15561-1_8

[8] Huber, Patrik, et al. “A Multiresolution 3D Morphable Face Model and Fitting Framework.” *Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, 2016, doi:10.5220/0005669500790086.

[9] Zou, Xuan, et al. “Robust Albedo Estimation from Face Image under Unknown Illumination.” *Biometric Technology for Human Identification V*, 2008, doi:10.1117/12.778599.

[10] Suh, Sungho, et al. “Robust Albedo Estimation from a Facial Image with Cast Shadow.” *2011 18th IEEE International Conference on Image Processing*, 2011, doi:10.1109/icip.2011.6116697.

[11] Sagonas, Christos, et al. “300 Faces In-The-Wild Challenge: Database and Results.” *Image and Vision Computing*, vol. 47, 2016, pp. 3–18., doi:10.1016/j.imavis.2016.01.002.

[12] Blanz, Volker, et al. “Fitting a Morphable Model to 3D Scans of Faces.” *2007 IEEE 11th International Conference on Computer Vision*, 2007, doi:10.1109/iccv.2007.4409029.

[13] Aldrian, Oswald, and William A.p. Smith. “Inverse Rendering of Faces with a 3D Morphable Model.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 5, 2013, pp. 1080–1093., doi:10.1109/tpami.2012.206.

[14] Bas, Anil, et al. “Fitting a 3D Morphable Model to Edges: A Comparison Between Hard and Soft Correspondences.” Computer Vision – ACCV 2016 Workshops Lecture Notes in Computer Science, 2017, pp. 377–391., doi:10.1007/978-3-319-54427-4_28.

[15] “Portrait Lighting Essentials.” *Digital Photo Pro*, www.digitalphotopro.com/technique/lighting-techniques/portrait-lighting-essentials/.

[16] S. Sengupta, A. Kanazawa, C. D. Castillo, and D. Jacobs. SfSNet: Learning shape, reflectance and illuminance of faces in the wild. arXiv preprint arXiv:1712.01261, 2017.

[17] “Learn Computer Vision - Intro to Computer Vision Course.” *Udacity*, Georgia Institute of Technology, www.udacity.com/course/introduction-to-computer-vision--ud810.

[18] Bobby, and Mike. “4 Basic Lighting Setups.” *Improve Photography*, 5 Aug. 2014, improvephotography.com/flash-photography-basics-9/.

[19] Zafeiriou, S., et al. “Face Recognition and Verification Using Photometric Stereo: The Photoface Database and a Comprehensive Evaluation.” *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 1, 2013, pp. 121–135., doi:10.1109/tifs.2012.2224109.

ACADEMIC VITA

Kun Wang

kunwang5309@gmail.com

EDUCATION

The Pennsylvania State University — Expected May 2018
Bachelor of Science in Computer Engineering

WORK EXPERIENCE

Undergraduate Research Programmer, — October 2016-May 2017
Pennsylvania State University James Z. Wang Research Group, State College, PA

- Developed a web-based academic adviser, which can advise students on academic programs, course schedule, tuition and financial aid, powered by the IBM Watson platform
- Implemented an online chat interface, a Node.js-based back-end and a MongoDB database

Computer Science Learning Assistant, — January 2016-July 2016
Penn State Behrend School of Engineering, Erie, PA

- Graded ADTs and Data Structure lab assignments
- Held office hours to answer students' questions about their lab

Mathematics Grader, — January 2015-May 2015
Penn State Behrend School of Engineering, Erie, PA

- Graded Calculus assignments

HONORS & AWARDS

Dean's list every semester
Awarded for Academic Excellence — 2014-2017

- Julie Ann Masteller Memorial Scholarship (2015-2016)
- The President's Sparks Award (2015-2016)
- The Evan Pugh Scholar Senior Award (2016-2017)

TECHNICAL SKILLS

Programming Skills: C/C++ (advanced), Java (advanced), JavaScript (intermediate), Verilog(advanced), VHDL (intermediate), Machine Learning (MATLAB) (intermediate), python(intermediate), Spark(starter)

LANGUAGE SKILLS:

Fluent in English and Mandarin spoken and written