ARISTOTLE UNIVERSITY OF THESSALONIKI

Department of Electrical & Computer Engineering
HY1701 - Computer Graphics

Illumination

Prepared by: Nikolaos Andriotis 9472
Instructors: Anastasios Delopoulos, Antonis Karakottas
Date: July 11, 2022

Contents

1 Implementation

1.1 ambient_light()

To implement how an object's color changes with the ambient light, I used equation 8.2 on page 95.

$$I_{amb} = k_a * I_a$$

```
def ambient_light(ka, Ia):
    return ka * Ia
```

1.2 diffuse_light()

To implement diffuse reflection I used equation 8.5 on page 97. For every light source, I first compute the unit vector L and if the dot product is larger than zero, I add its light intensity to point P's color.

$$I_{diff} = k_d * I_{source} * (N \cdot L)$$

I note here, that I sum the intensities of all light sources adhering to equation 8.10 on page 99.

```
def diffuse_light(P, N, color, kd, light_positions, light_intensities):

    I = zeros(shape=(3, 1))

    for (source, intensity) in zip(light_positions, light_intensities):

        L = (source - P) / norm(source - P)

        if N.T @ L > 0:

            I += kd * intensity * (N.T @ L)

    return I * color
```

1.3 specular_light()

To implement this I used equation 8.9 on page 99. I sum the intensities in a similar way as in the diffuse implementation. The main difference is that both dot products should be greater than zero in order for the light source to contribute.

$$I_{specular} = k_s * I_{source} * (V \cdot R)^n$$

where

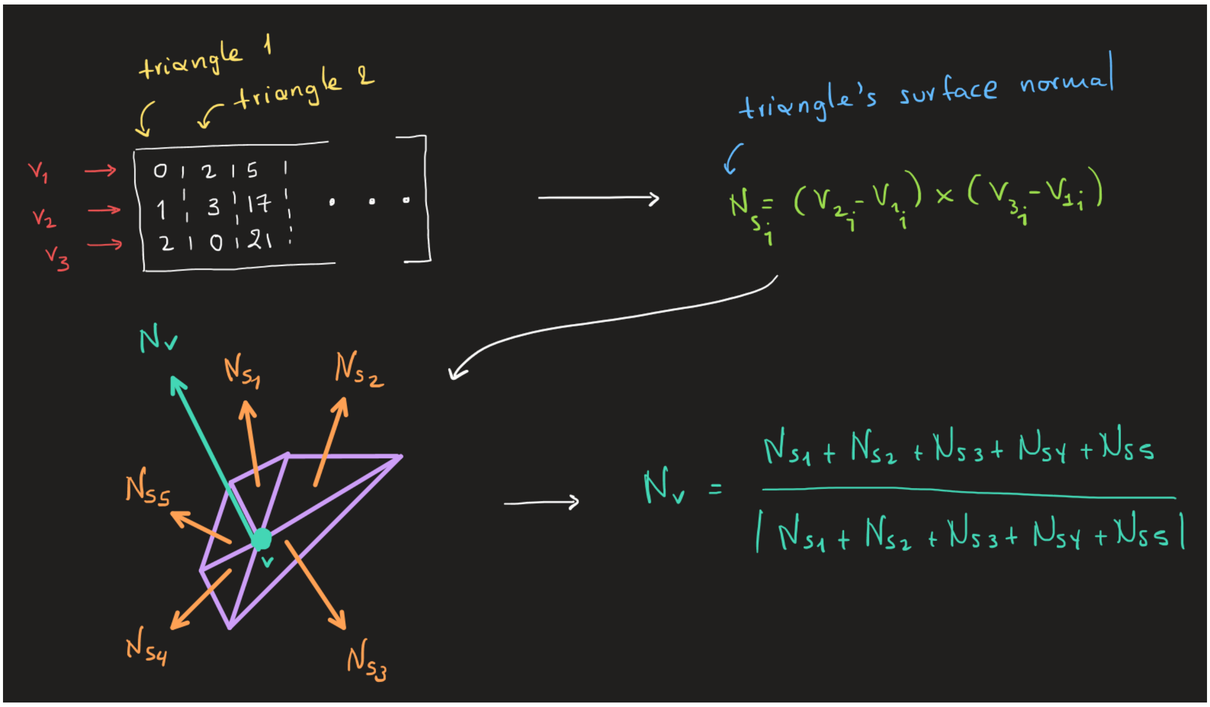
$$R = (2N \cdot N) * N - L$$

```
def specular_light(P, N, color, cam_pos, ks, n, light_positions, light_intensities):

    I = zeros(shape=(3, 1))

    for (source, intensity) in zip(light_positions, light_intensities):

        L = (source - P) / norm(source - P)
        V = (cam_pos - P) / norm(cam_pos - P)
        R = (2 * N.T @ L) * N - L

        if V.T @ R > 0 and N.T @ L > 0:

            I += ks * intensity * ((V.T @ R) ** n)

    return I * color
```

1.4 calculate_normals()

To implement this, I first calculated the unit surface normals for every triangle. Then, to find every vertex normal, I sum and normalize all the triangle's surface normals to which the vertex is adjacent. A visual explanation will be helpful.

```
def calculate_normals(vertices, face_indices):

    normals = empty(shape=vertices.shape)

    surface_normals = cross(vertices[:, face_indices[1]] - \
                            vertices[:, face_indices[0]],
                            vertices[:, face_indices[2]] - \
                            vertices[:, face_indices[0]],
                            axis=0)

    surface_normals /= norm(surface_normals, axis=0)

    for i in range(vertices.shape[1]):

        S_Nk = sum(surface_normals[:, where(face_indices == i)[1]], axis=1)

        S_Nk = S_Nk.reshape(3, 1)

        S_Nk /= norm(S_Nk)

        normals[:, i] = S_Nk.reshape(-1)

    return normals
```



1.5 render_object()

To implement this, I followed the algorithm (three steps) given in the assignment.

First, I calculate the normal of each vertex.

```python
def render_object():
    ...
    normals = calculate_normals(vertices=verts, face_indices=face_indices)
    ...
```

Secondly, I project and properly fit the projected object onto the image based on its dimensions.

```python
def render_object():
    ...
    verts2d, depth = project_cam_lookat(f=focal,
                                        c_org=eye,
                                        c_lookat=lookat,
                                        c_up=up,
                                        verts3d=verts)

    verts_rast = rasterize(verts2d=verts2d,
                           img_h=N,
                           img_w=M,
                           cam_h=H,
                           cam_w=W)
    ...
```

Thirdly, after some preprocess, for each triangle (whose vertices are all inside the image), I shade using either gouraud or phong combined with the right illumination model. I note here that depending on the shader argument, the right function is called using the "globals" functionality.

```python
def render_object():
    ...
    for triangle in priority_of_triangles:

        if (all(verts_rast[:, face_indices[:, triangle]][0, :]) < W) \
        and (all(verts_rast[:, face_indices[:, triangle]][1, :]) < H):

            X = globals()[f'shade_{shader}'](...)
    ...
```

1.6 shade_gouraud()

To implement this, I first calculate the triangle vertice's colors after the application of the illumination model.

```
def shade_gouraud():
    ...
    for i in range(3):

        I_amb = ambient_light(ka, Ia)

        I_diff = diffuse_light(P=bcoords,
                               N=verts_n[:, i].reshape(3, 1),
                               color=verts_c[:, i].reshape(3, 1),
                               kd=kd,
                               light_positions=light_positions,
                               light_intensities=light_intensities)

        I_spec = specular_light(P=bcoords,
                                N=verts_n[:, i].reshape(3, 1),
                                color=verts_c[:, i].reshape(3, 1),
                                cam_pos=cam_pos.reshape(3, 1),
                                ks=ks,
                                n=n,
                                light_positions=light_positions,
                                light_intensities=light_intensities)

        verts_c[:, i] = (I_amb + I_diff + I_spec).reshape(-1)
    ...
```

After that, with the updated colors, I do the exact same triangle filling algorithm I did for the rest of the assignments.

```
   triangle_edges = fromiter((create_edge(tuples_of_verts[i]) for i in range(3)))

    lowest_scanline = min(triangle_edges['y_min'])
    highest_scanline = max(triangle_edges['y_max'])

    active_edges = triangle_edges[(triangle_edges['y_min'] == lowest_scanline)]

    for y in range(lowest_scanline, highest_scanline):

        lower_edges = active_edges[active_edges['y_max'] == y]
        if lower_edges.size > 0:
            active_edges = delete(active_edges, active_edges == lower_edges)

        active_edges = sort(active_edges, order='intersect')

        leftmost_intersect = ceil(active_edges[0]['intersect'])
        rightmost_intersect = ceil(active_edges[1]['intersect'])

        Cl, Cr = interpolate_color(active_edges['y_max'],
                                   active_edges['y_min'],
                                   y,
                                   active_edges['RGB_max'],
                                   active_edges['RGB_min'])

        for x in range(int(leftmost_intersect), int(rightmost_intersect)):

```

```python
            X[x, y] = interpolate_color(rightmost_intersect,
                                        leftmost_intersect,
                                        x,
                                        Cr,
                                        Cl)

        active_edges['intersect'] += 1 / active_edges['slope']

        upper_edges = triangle_edges[triangle_edges['y_min'] == y + 1]
        if upper_edges.size > 0:
            active_edges = append(active_edges, upper_edges)

    return X
```

1.7 shade_phong()

To implement this, instead of applying the illumination model to the triangle's vertices. I apply it the the point of the current scan line after I interpolate both its normal and its color. I note, since is is a lengthy function, I have simplified it for a better understanding.

```python
def shade_phong():
    ...
    for y in range(lowest_scanline, highest_scanline):
        ...
        Cl, Cr = interpolate_color(...)

        Nl, Nr = interpolate_color(...)

        for x in range(int(leftmost_intersect), int(rightmost_intersect)):

            color = interpolate_color(...)

            normal = interpolate_color(...)

            I_amb = ambient_light()

            I_diff = diffuse_light()

            I_spec = specular_light()

            color = I_amb + I_diff + I_spec

            X[x, y] = color
    ...
```

2 Compilation and Results

2.1 Compilation

You have to run the following command inside the directory containing demo.py, filling.py, transforms_projections.py and illumination.py. The images generated are saved inside it. I used an anaconda environment with python=3.7, numpy and opencv-python.

```
python demo.py
```

2.2 Results

(a) **Gouraud Ambient**

(b) **Gouraud Diffuse**

(c) **Gouraud Specular**

(d) **Gouraud Complete**

(a) **Phong Ambient**

(b) **Phong Diffuse**

(c) **Phong Specular**

(d) **Phong Complete**