ECE549 MP1.

#1.

Consider a plane in 3D coordinate as $P: \vec{N}^T \cdot \vec{X} = d$. --- (1)

where $\vec{N} = [N_X \; N_Y \; N_Z]^T$, $\vec{X} = [X \; Y \; Z]^T$.

① Let's calculate the vanishing line (in 2D img. coord.) of this plane first;

By expanding (1), $N_X \cdot X + N_Y \cdot Y + N_Z \cdot Z = d$.

$\xrightarrow{\times \frac{f}{Z}} N_X \cdot \frac{fX}{Z} + N_Y \cdot \frac{fY}{Z} + N_Z \cdot f = \frac{fd}{Z}$

$\rightarrow N_X \cdot x + N_Y \cdot y + N_Z \cdot f = \frac{fd}{Z}$, where $\vec{x} = [x \; y \; 1]^T$ is a 2d homo. coord. in img.

$\xrightarrow{\text{if } Z \to \infty} N_X \cdot x + N_Y \cdot y + N_Z \cdot f = 0$. ---------- (2)

∴ Equation of vanishing line of the 3D plane : $N_X \cdot x + N_Y \cdot y + N_Z \cdot f = 0$.

(x, y : image coordinate (2d)).

② Let's assume we investigate an arbitrary line in this plane,

which is denoted as a 3D-vector form $\ell: \vec{A} + \lambda \vec{D}$. ($\vec{A}$: 3D point on this line ℓ) ($\vec{D}$: 3D directional vector of ℓ)

Since it is on the plane P, $\vec{A}$ and $\vec{D}$ suffices $\vec{N}^T \cdot \vec{A} = d$ --- (3)

$\vec{N}^T \cdot \vec{D} = 0$ --- (4)

(∵ (3) by (1), (4) by the orthogonality b.w. $\vec{N}$ and $\vec{D}$)

A point on ℓ can be then parametrized as $\vec{X}_\ell = [A_X + \lambda \cdot D_X, \; A_Y + \lambda \cdot D_Y, \; A_Z + \lambda \cdot D_Z]^T$.

If this point goes infinity (here, $\lambda \to +\infty$ or $-\infty$), $\lim_{\lambda \to \infty} \vec{X}_\ell = [D_X, D_Y, D_Z]^T$.

Hence, in 2D (homogeneous) img. coord., the vanishing point of ℓ is $[f\frac{D_X}{D_Z}, f\frac{D_Y}{D_Z}, 1]^T$.

$= [x', y', 1]^T$

However, by (4), $N_X \cdot D_X + N_Y \cdot D_Y + N_Z \cdot D_Z = 0$ holds

$\xrightarrow{\times \frac{f}{D_Z}} N_X \cdot \frac{fD_X}{D_Z} + N_Y \frac{fD_Y}{D_Z} + N_Z \cdot f = 0 \rightarrow N_X \cdot x' + N_Y \cdot y' + N_Z \cdot f = 0$. ----(5)

(cont'd)

Hence, by (5) and (2), we can see that

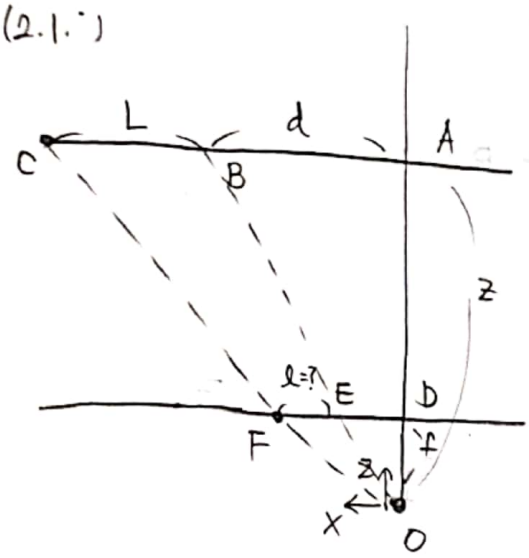
any vanishing points of lines on this plane ($= \forall \ell$ on P), $\left(x' = f\cdot\frac{Dx}{Dz},\ y' = f\cdot\frac{Dy}{Dz}\right)$.

lies on the vanishing line of the plane P, which is represented as $N_x \cdot x + N_y \cdot y + fN_z = 0$

□

#2.

(2.1.)



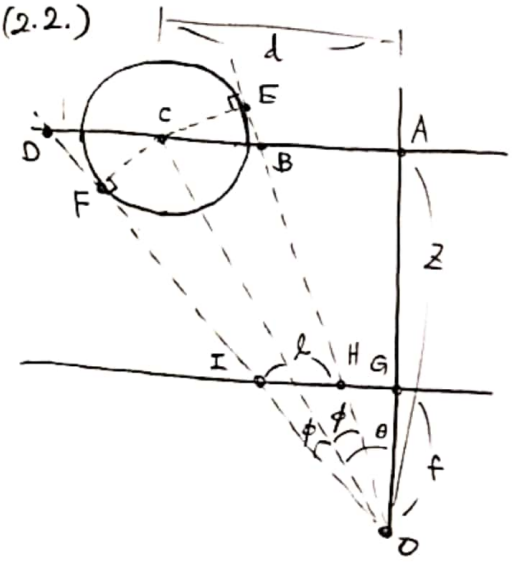
$\overline{DE} = \frac{f}{z} \cdot d$ ($\because \triangle ODE \sim \triangle OAB$)

$\overline{DF} = \frac{f}{z} \cdot (L+d)$ ($\because \triangle ODF \sim \triangle OAC$).

$\Rightarrow l = \overline{EF} = \overline{DF} - \overline{DE} = \frac{f}{z} \cdot (L+d) - \frac{f}{z} \cdot d = \frac{f}{z} \cdot L$.

$\therefore l = f \cdot \frac{L}{z}$ (independent of d).

(2.2.)



$\angle BCE = \angle BOA = \theta - \phi$ ($\because \triangle BCE \sim \triangle BOA$)

$\rightarrow \overline{BC} = \frac{\overline{CE}}{\cos(\angle BCE)} = \frac{r}{\cos(\theta-\phi)}$ ---- (1)

$\angle DCF = \angle DOA = \theta + \phi$ ($\because \triangle DCF \sim \triangle DOA$)

$\rightarrow \overline{CD} = \frac{\overline{CF}}{\cos(\angle DCF)} = \frac{r}{\cos(\theta+\phi)}$ --- (2)

$\overline{BD} = \overline{BC} + \overline{CD} = r\left[\frac{1}{\cos(\theta-\phi)} + \frac{1}{\cos(\theta+\phi)}\right] = r \cdot \left[\frac{1}{\cos\theta\cdot\cos\phi + \sin\theta\cdot\sin\phi} + \frac{1}{\cos\theta\cdot\cos\phi - \sin\theta\sin\phi}\right]$

$= r \cdot \frac{\cancel{\cos\theta\cos\phi} - \cancel{\sin\theta\sin\phi} + \cancel{\cos\theta\cos\phi} + \cancel{\sin\theta\sin\phi}}{\cos^2\theta\cos^2\phi - \sin^2\theta\sin^2\phi} = r \cdot \frac{2\cos\theta\cdot\cos\phi}{\cos^2\theta\cdot(1-\sin^2\phi) - \sin^2\theta\cdot\sin^2\phi}$

$= r \cdot \frac{2\cos\theta\cdot\cos\phi}{\cos^2\theta - \sin^2\phi}$

$r\sin\phi = \frac{CE}{OC} = \frac{r}{\sqrt{z^2+d^2}}$, $\cos\phi = [1-\sin^2\phi]^{\frac{1}{2}} = \frac{\sqrt{z^2+d^2-r^2}}{\sqrt{z^2+d^2}}$, $\cos\theta = \frac{z}{\sqrt{z^2+d^2}}$, $\sin\theta = \frac{d}{\sqrt{z^2+d^2}}$

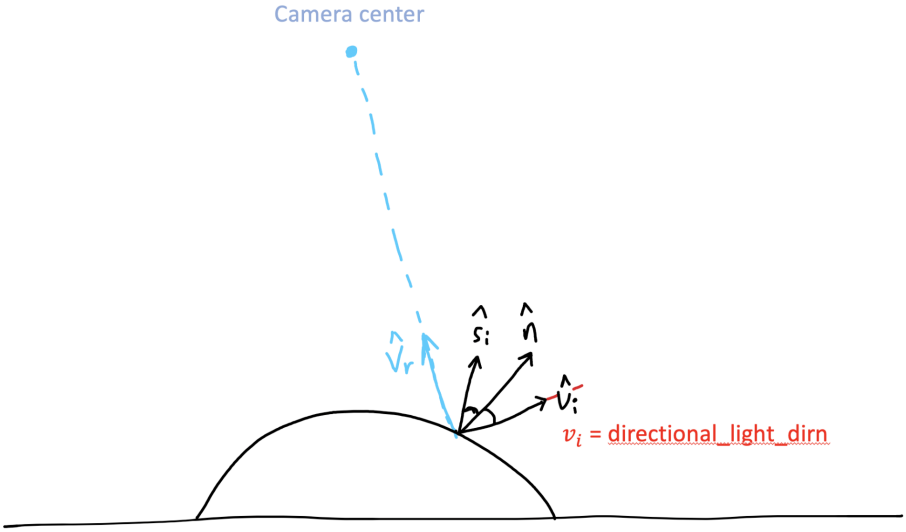
$\therefore l = \overline{HI} = \frac{f}{z} \cdot \overline{BD} = \frac{f}{z} \cdot r \cdot \frac{2 \cdot \frac{z}{\sqrt{z^2+d^2}} \cdot \frac{\sqrt{z^2+d^2-r^2}}{\sqrt{z^2+d^2}}}{\frac{z^2}{z^2+d^2} - \frac{r^2}{z^2+d^2}} = \frac{f}{z} \cdot 2r \cdot \frac{z\sqrt{z^2+d^2-r^2}}{z^2-r^2} = g(d)$.

$\therefore$ Here, if d increases, l increases. vice versa.

MP1-Q3 by Jongwon Lee (jongwon5)

Preliminary

- The following code has been tested in Python 3.7.9, with packages of `numpy=1.19.1` and `matplotlib=3.3.1`.
- All implementations and results can be found in the directory q3_code/.
- All conventions, especially the direction of vectors in Phong Shading Model, ahere to that in Szeliski. For better clarification, I attach the image from TA, which apparently shows the relationship between the direction of incident light ($\hat{v}*{i}$) and reflected light ($\hat{s}*{i}$), which is parameterized by the normal of the surface ($\hat{n}$). In addition, the viewing direction is depicted as well ($\hat{v}_{r}$). Note that all directional vectors should be normalized before any operation.

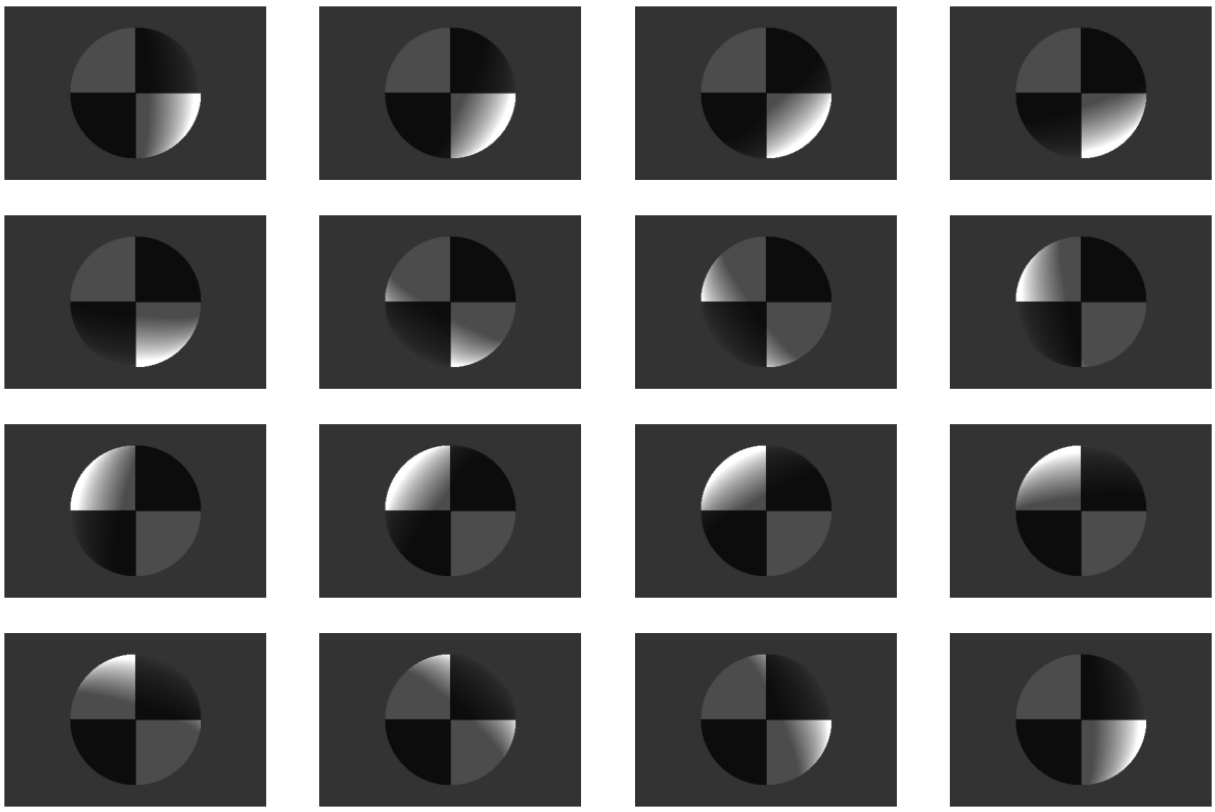


How to run

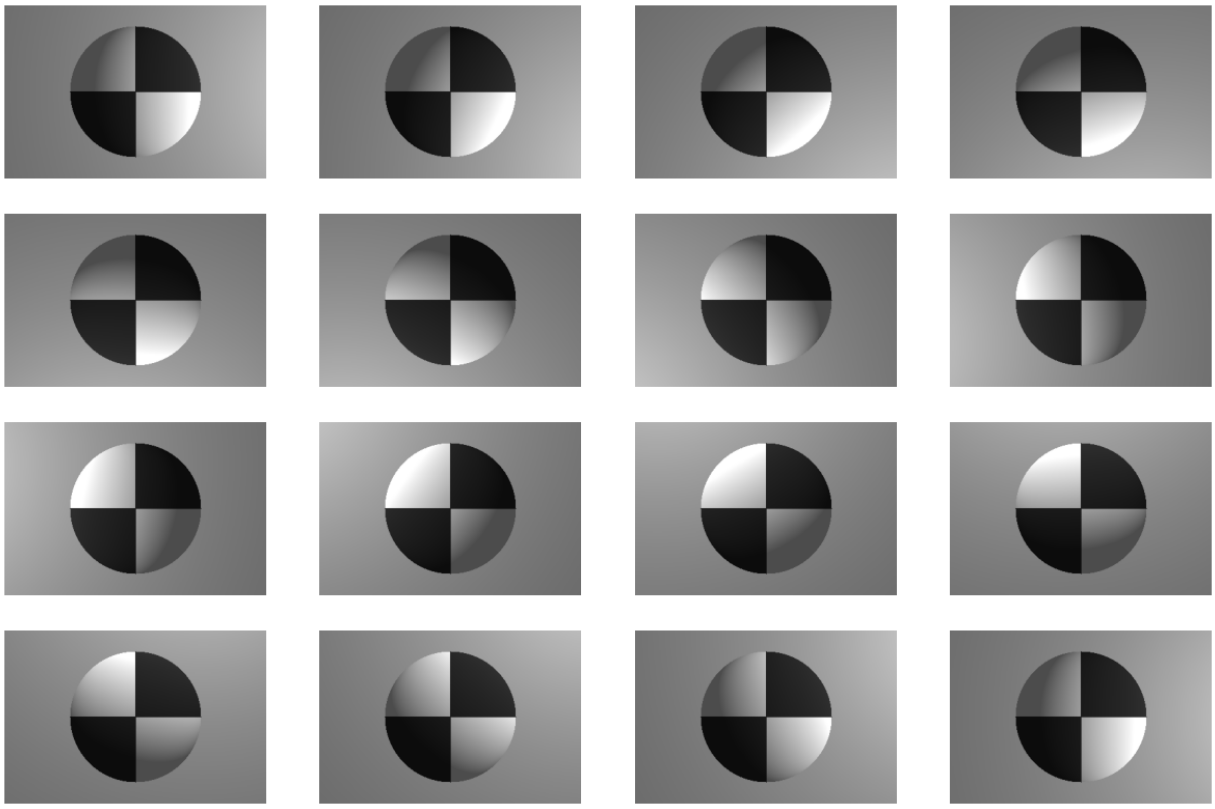
- To recreate the results, you may simply execute the render_image.py. It will automatically generates four distinct images with different light condition.

Results

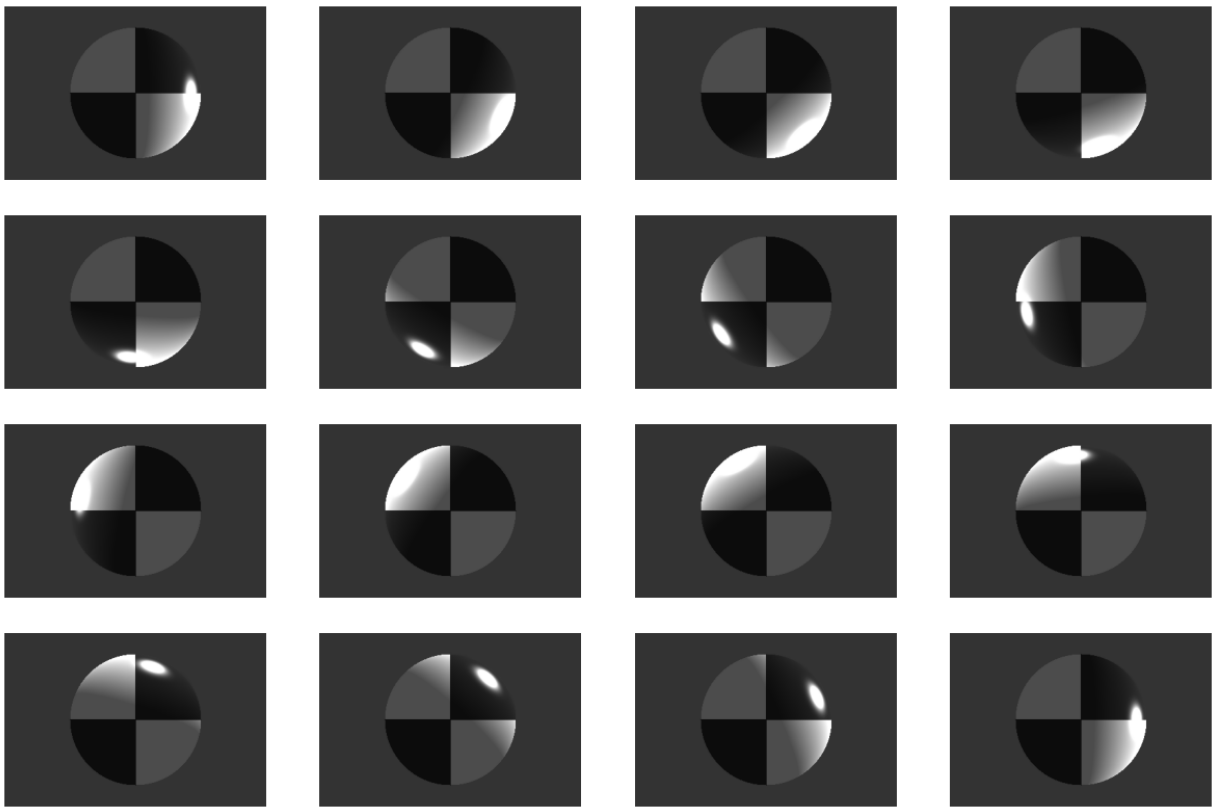
1. No specular reflection and point light source. Just a directional light source that moves around the object (`specular0_move_direction.png`)



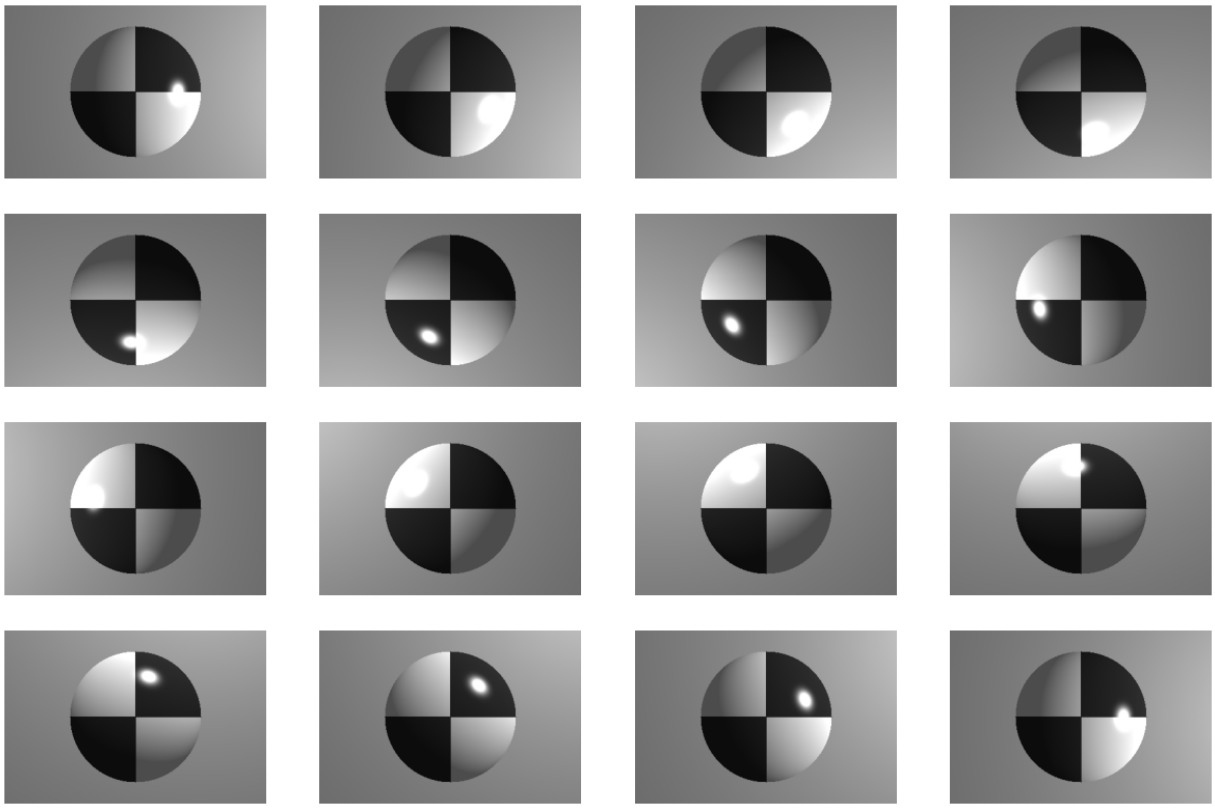
2. No specular reflection and directional light. Just a point light source that moves around the object (`specular0_move_point.png`)



3. No point light source, but a directional light source that moves around the object with specular reflection (`specular1_move_direction.png`)



4. No directional light, but a point light source that moves around the object with specular reflection (`specular1_move_point.png`)



For the case 1 and 3, a directional light starting from the positive part of x-axis and rotating clockwise around z-axis was projected to the source. (Namely, `directional_light_dirn = [np.cos(theta), np.sin(theta), .1]`, where `theta` ranges from 0 to 2π.)

As similar to this, for the case 2 and 4, a point light source starting from the positive part of x-axis and rotating clockwise around z-axis was assumed to be located behind the center of camera coordinate. (Hence, `point_light_loc = [10*np.cos(theta), 10*np.sin(theta), -3]`, where `theta` ranges from 0 to 2π.)

While `theta` changes from 0 to 2π, the subfigures are plotted column-wise followed by row-wise. Hence, each scenarios are drawn in the order of

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
```

while `theta` increases.

Considering that the point light source in the case 2 and 4 is located behind than the directional light source in the case 1 and 3, you may see that the specular reflection in 4 occurs at more frontal part of the sphere than that of 3. Overall, we can conclude that the implementation of Phong's model has been properly done.

How does it work?

The key issue in implementing Phong's model is how to take into account three different directional vectors. In this code, all the directional convention obeys Fig.2.15. in Szeliski.

Here, I attach the code that fills in the array of incident light ($\hat{v}*{i}$), reflected light ($\hat{s}*{i}$), and ($\hat{v}_{r}$), which depend on the object's 3D location that we are looking at in the camera coordinate $(X, Y, depth)$ as well as the normal of the surface ($\hat{n}$), each of which corresponds to the 2d pixel's coordinate (u, v) in the image frame we are looking into.

```
for v in range(h):
  for u in range(w):
    depth = Z[v, u]           # depth to the object from camera center
    X = depth / f * (u - cx)  # X coordinate of the object w.r.t. the
camera center
    Y = depth / f * (v - cy)  # Y coordinate of the object w.r.t. the
camera center

    # incident light direction
    vi_p[v, u, :] = - np.array([X - point_light_loc[0][0], Y -
point_light_loc[0][1], depth - point_light_loc[0][2]])
    vi_d[v, u, :] = np.array(directional_light_dirn[0])
    # specular reflection direction (see Section 2.2., Equation 2.90. in
Szeliski)
    si_p[v, u, :] = np.matmul(2 * N[v, u, :] * N[v, u, :][:, np.newaxis] -
np.eye(3), vi_p[v, u, :])
    si_d[v, u, :] = np.matmul(2 * N[v, u, :] * N[v, u, :][:, np.newaxis] -
np.eye(3), vi_d[v, u, :])
```

```
    # viewing direction
    vr[v, u, :] = - np.array([X, Y, depth])
```

At first, each 2D pixel coordinate (u, v) should be converted into the 3D camera coordinate $(X, Y, depth)$, which is parametrized by the focal length (f) and camera center (c_{x}, c_{y}) as well. After than, incident light vector ($\hat{v}*{i}$) *for every pixel can be estimated as all pixels' location in 3D camera coordinate has been decided in the prior step. Note that the incident light's direction for both the point light source ($\left(\hat{v}*{i}\right)*{p}$) and the directional light ($\left(\hat{v}*{i}\right)*{d}$) needs to be calculated respectively as both of them are in our consideration. Regarding that $\left(\hat{v}*{i}\right)*{p}$ is defined as the direction from the incidental surface to the light source, a minus sign has been appended for the direction from the light source to the incidental surface. Once both $\left(\hat{v}*{i}\right)*{p}$ and $\left(\hat{v}*{i}\right)*{d}$ has been calculated, we can estimate reflective light vector ($\hat{s}*{i}$)*. Since $\hat{s}*{i}$ depends on $\hat{v}*{i}$, $\hat{s}*{i}$ for $\left(\hat{v}*{i}\right)*{p}$ and $\left(\hat{v}*{i}\right)*{d}$ should be calculated respectively as well. Using the Equation 2.90. in Szeliski ($\hat{s}*{i} = \left(2\hat{n}\hat{n}^{T} - I_{3\times3}\right)\hat{v}*{i}$), the reflective direction for both point light source ($\left(\hat{s}*{i}\right)*{p}$) and directional light ($\left(\hat{s}*{i}\right)*{d}$) are obtained. Lastly, the viewing direction ($\hat{v}*{r}$), the direction from the reflected surface to the observer*, are estimated.

Keep in mind that all of these directional vectors should be normalized not to occur any problem while we sum up all light intensity terms in Phong's model. If not, it may cause the overflow and the outcome image will be saturated.

Once all directional vectors are prepared, the only thing to do is just to calculate each intensity term and add them up to generate the resulting image. I attach the code of this part below - Note that any dot product between two identical size of 2D arrays composed with 3D vectors are implemented with `einsum` function in `numpy` package because `dot` function exclusively supports the doc product between two 3D vectors. Also, `clip` function was applied to the outcome to exclude any non-negative parts.

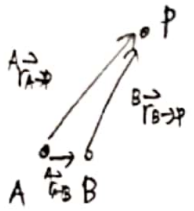
```
# Ambient Term
Ia = A * ambient_light

# Diffuse Term
point_light_strength = point_light_strength[0]
directional_light_strength = directional_light_strength[0]

Id_p = A * point_light_strength * np.clip(np.einsum('ijk,ijk->ij', vi_p,
N), 0, None)
Id_d = A * directional_light_strength * np.clip(np.einsum('ijk,ijk->ij',
vi_d, N), 0, None)

# Specular Term
Is_p = S * point_light_strength * pow(np.clip(np.einsum('ijk,ijk->ij', vr,
si_p), 0, None), k_e)
Is_d = S * directional_light_strength * pow(np.clip(np.einsum('ijk,ijk-
>ij', vr, si_d), 0, None), k_e)

# Sum up terms above
I = Ia + Id_p + Id_d + Is_p + Is_d
```

For more detail, it is highly recommended to navigate into render_image.py.

#4.1.

「Lemma: linear velocity of P seen by camera is given as $\dot{\vec{P}} = -\vec{t} - \vec{\omega} \times \vec{P}$.

proof: Let the instant frame where camera is located at as A, (stationary)
the next frame where camera is going to be moved as B. (moving)



$^A\dot{\vec{r}}_{A\to P} = \vec{0}$ (∵ frame A stationary)

$R_A^B \approx I$, $^A\vec{r}_{A\to B} \approx \vec{0}$ (∵ Δt small).

given: $^A\dot{\vec{r}}_{A\to B} := \vec{t}$, $^A\vec{\omega}_{A\to B} := \vec{\omega}$, $^A\vec{r}_{A\to P} = \vec{P}$.

goal: $^B\dot{\vec{r}}_{B\to P} := \dot{\vec{P}} = ?$

By kinematics, $^B\vec{r}_{B\to P} = R_A^B \cdot (^A\vec{r}_{B\to P}) = R_A^B \cdot (^A\vec{r}_{A\to P} - {}^A\vec{r}_{A\to B})$

Taking $\frac{d}{dt}$, $\frac{d\,^B\vec{r}_{B\to P}}{dt} = \frac{dR_A^B}{dt}(^A\vec{r}_{A\to P} - \cancel{^A\vec{r}_{A\to B}}^{\,\vec{0}}) + R_A^B(^A\dot{\vec{r}}_{A\to P} - {}^A\dot{\vec{r}}_{A\to B})$

$\approx \dot{R}_A^B \cdot (^A\vec{r}_{A\to P} - \vec{0}) + I(\vec{0} - {}^A\dot{\vec{r}}_{A\to B})$

$= -[^B\vec{\omega}_{A\to B}]_\times \cdot {}^A\vec{r}_{A\to P} - {}^A\dot{\vec{r}}_{A\to P}$ ($[\cdot]_\times$: skew-symmetric matrix, a cross-product operator)

$\approx -[^A\vec{\omega}_{A\to B}]_\times \cdot {}^A\vec{r}_{A\to P} - {}^A\dot{\vec{r}}_{A\to P}$ ($^B\vec{\omega}_{A\to B} = R_A^B \cdot {}^A\vec{\omega}_{A\to B} \approx {}^A\vec{\omega}_{A\to B}$)

$\therefore \dot{\vec{P}} = -\vec{\omega} \times \vec{P} - \vec{t}$. ∎ *

Therefore, a stationary point P is observed as it has a linear velocity

$$\dot{\vec{P}} = \begin{bmatrix}\dot{X}\\ \dot{Y}\\ \dot{Z}\end{bmatrix} = -\vec{\omega}\times\vec{P} - \vec{t} = -\begin{bmatrix}0 & -\omega_z & \omega_y\\ \omega_z & 0 & -\omega_x\\ -\omega_y & \omega_x & 0\end{bmatrix}\begin{bmatrix}X\\ Y\\ Z\end{bmatrix} - \begin{bmatrix}t_x\\ t_y\\ t_z\end{bmatrix} = \begin{bmatrix}\omega_z Y - \omega_y Z - t_x\\ \omega_x Z - \omega_z X - t_y\\ \omega_y X - \omega_x Y - t_z\end{bmatrix}.$$

Let's define this object's velocity observed in 2D image, by starting with defining $x := f\cdot\frac{X}{Z}$, $y := f\cdot\frac{Y}{Z}$.

$$\dot{x} = f\,\frac{\dot{X}Z - X\dot{Z}}{Z^2} = \frac{f}{Z^2}\cdot\left[\omega_z\cdot YZ - \omega_y\cdot Z^2 - t_x\cdot Z - \omega_y X^2 + \omega_x\cdot XY + t_z\cdot X\right]$$

$$= y\cdot\omega_z - f\cdot\omega_y - \frac{f}{Z}t_x - \frac{1}{f}\cdot x^2\omega_y + \frac{1}{f}\cdot xy\cdot\omega_x + \frac{1}{Z}\cdot x\cdot t_z.$$

$$\dot{y} = f\cdot\frac{\dot{Y}Z - Y\dot{Z}}{Z^2} = \frac{f}{Z^2}\left[\omega_x Z^2 - \omega_z XZ - t_y Z - \omega_y XY + \omega_x Y^2 + t_z Y\right]$$

$$= f\cdot\omega_x - x\cdot\omega_z - \frac{f}{Z}\cdot t_y - \frac{1}{f}\cdot xy\cdot\omega_y + \frac{1}{f}\cdot y^2\cdot\omega_x + \frac{1}{Z}\cdot y\cdot t_z$$

(cont'd)

* Zhao, Shiyu. "Time derivative of rotation matrices: A tutorial"

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} -\frac{f}{Z} & 0 & \frac{x}{Z} \\ 0 & -\frac{f}{Z} & \frac{y}{Z} \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} + \begin{bmatrix} \frac{xy}{f} & \frac{-(x^2+f^2)}{f} & y \\ \frac{(y^2+f^2)}{f} & -\frac{xy}{f} & -x \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

$$= \frac{1}{Z} \begin{bmatrix} -f & 0 & x \\ 0 & -f & y \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} + \frac{1}{f} \cdot \begin{bmatrix} xy & -(x^2+f^2) & y \\ (y^2+f^2) & -xy & -x \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

MP1-Q4 by Jongwon Lee (jongwon5)

Preliminary

- The following code has been tested in Python 3.7.9, with packages of `numpy=1.19.1` and `matplotlib=3.3.1`.
- All implementations and results can be found in the directory q4_code/.
- For the derivation of the relationship between the camera's velocity and any object's location w.r.t. 3D camera coordinate, please refer to my response in Question 4.1. One of the most annoying concept is how we can convert any physical quantity taken in fixed frame to a .rotating frame. For more reference on this, please refer to the following paper:

> Zhao, Shiyu. "Time derivative of rotation matrices: A tutorial." arXiv preprint arXiv:1609.06088 (2016).

How to run

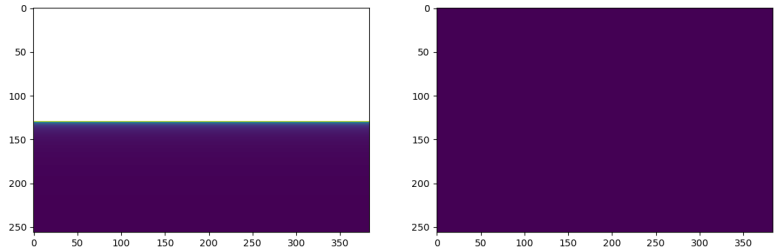
- To recreate the results, you may simply execute dynamic_perspective_starter.py. It will automatically generates five PDF files in different camera motion scenarios.

Implementation

All tasks are listed below:

1. Looking forward on a horizontal plane while driving on a flat road.
2. Sitting in a train and looking out over a flat field from a side window.
3. Flying into a wall head-on.
4. Flying into a wall but also translating horizontally, and vertically.
5. Counter-clockwise rotating in front of a wall about the Y-axis.

The outcome of two functions `get_road_z_image` and `get_wall_z_image` in dynamic_perspective_starter.py are shown below. For simplicity, the outcome of `get_road_z_image` is called as `road` from now on whereas that of `get_wall_z_image` as `wall`.



Hence, we can rewrite the tasks in more intuitive notation as:

1. `road`, $v_{z} = 1$
2. `road`, $v_{x} = 1$
3. `wall`, $v_{z} = 1$
4. `wall`, $\vec{v} = \left(v_{x}, v_{y}, v_{z}\right) = (1, 1, 1)$
5. `wall`, $\omega_{y} > 1$

These five scenarios are executed and plotted in each corresponding PDF file, respectively.

The biggest challenge for the implementation is how to generate corresponding optical flows in an image frame w.r.t. the simulated camera motion. To handle this, the response in Queston 4.1. has been carefully implementation in the function `create_optical_flow` in dynamic_perspective_starter.py. Here, I attach the key algorithm to obtain optical flow:

```
for v in range(szy):
  for u in range(szx):
    depth = Z[v, u]           # depth of the object w.r.t. camera center
    X = depth / f * (u - cx)  # X coordinate of the object w.r.t. the
camera center in 3D
    Y = depth / f * (v - cy)  # Y coordinate of the object w.r.t. the
camera center in 3D

    x = f * X / depth   # x coordinate of pixel w.r.t. the camera center in
2D
    y = f * Y / depth   # y coordinate of pixel w.r.t. the camera center in
2D

    # Construct matrices for translational and rotational velocity
    T = np.array([[-f, 0, x],
                  [0, -f, y]])
    R = np.array([[x*y, -(x**2+f**2),  y],
                  [(y**2+f**2), -x*y, -x]])

    # Compute optical flow
    # eps has been added to avoid zero division
    of = T @ t / (depth + eps) + R @ w / f
    of_x[v, u] = of[0]
    of_y[v, u] = of[1]
```

At first, any 2D pixel coordinate (u, v) should be properly transformed into a corresponding 3D camera coordinate $(X, Y, depth)$. Note that (x, y) is represented in 2D pixel coordinate, whose center is collocated to the camera center (i.e. same as the origin of $(X, Y, depth)$. Once the coordinate conversion has been done, we can construct two different matrices T and R which impose the weight for $\hat{v}$ and $\hat{\omega}$ while calculating the optical flow. After than, we can estimate the optical flow along x and y axes, respectively.

Results

Please refer to the attached appendices. The outcomes align with our intuition.

1. `road`, $v_{z} = 1$

- As the camera is moving into the image, the optical flow is coming toward the camera.

2. `road`, $v_{x} = 1$

- As the camera is moving right, the optical flow heads toward left, which is the opposite direction of the movement.

3. `wall`, $v_{z} = 1$

- As the camera is moving into the image, the optical flow is coming toward the camera.

4. `wall`, $\vec{v} = \left(v_{x}, v_{y}, v_{z}\right) = (1, 1, 1)$

- As the camera is moving into the image's bottom-right side, we perceive the optical flow originated from bottom-right as well as coming toward the camera.

5. `wall`, $\omega_{y} > 1$

- Since the camera is counterclockwise around y-axis, which towards downward, we perceive the optical flow heading toward left.