108062213 顏浩昀
HW4

1. Plot cubic Bezier curves

a. Low detail t = {0, 0.5, 1} (Blue line), High detail t = {0, 0.01, 0.02, 0.03 … 0.99, 1} (Red line)

I. Show the image

II. Implement and discussion

```python
def cal_curve(point, ratio):
    Bx = []
    By = []
    for t in np.arange(0, 1.01, ratio):
        x = ((1 - t) ** 3) * point[0, 0] + 3 * ((1 - t) ** 2) * t * point[1, 0] + 3 * (1 - t) * (t ** 2) * point[2, 0]+ (t ** 3) * point[3, 0]
        y = ((1 - t) ** 3) * point[0, 1] + 3 * ((1 - t) ** 2) * t * point[1, 1] + 3 * (1 - t) * (t ** 2) * point[2, 1]+ (t ** 3) * point[3, 1]
        Bx.append(x)
        By.append(y)

    return [Bx, By]
```

```python
def subplot(points, img):

    # low detail
    plt.imshow(img)

    plt.scatter(points[:, 0], points[:, 1],  s=0.5)

    for i in range(0, len(points)-1, 3):
        point = points[i:i + 4]
        [Bx, By] = cal_curve(point, 0.5)
        plt.plot(Bx, By, 'b-' ,linewidth=0.5)

    #high detail
    for i in range(0, len(points)-1, 3):
        point = points[i:i + 4]
        [Bx, By] = cal_curve(point, 0.01)
        plt.plot(Bx, By, 'r-' ,linewidth=0.5)
    plt.savefig('./output/1a.png')
```

I define a function to calculate each point position on curve with every four setting points. The formula is $B = (1-t)^3 * p_0 + t * (1-t)^2 * p_1 + t^2 * (1-t) * p_2 + t^3 * p_3$. The t range is between 0 and 1, and the ratio of low detail is 0.5 and 0.01 for high detail. To plot the image, I scatter the all point at first, then put every four point into cal_curve function to get the curve and plot curve on the image.

I think the significant difference between low detail curve and high detail curve is appeared when the continuous points not in the line such as the line around the man's neck. At these positions, high detail

curve is closer to the man's contour than the low detail curve. Look at the hat we can find that the low detail curve may more likely be a straight line, and high detail line is more smooth.

b. Scale up to 4 times larger

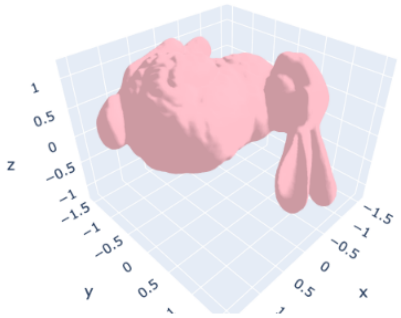

```python
def nnScale(img):
    height, width = img.shape[0], img.shape[1]
    newh, neww = height*4, width*4
    scale_Img = np.zeros((newh, neww, 3), dtype=np.uint8)
    for y in range(newh):
        for x in range(neww):
            oriy = int(np.floor(y/4.0))
            orix = int(np.floor(x/4.0))
            scale_Img[y, x, :] = img[oriy, orix, :]
    return scale_Img
```

```python
newpoints = points*4
```

This scale function is the same function in HW1. I utilize np.floor() to find the nearest original pixel to fill the color for the scaled image pixel. The new points are easily calculated by multiple 4 times with original points.

2. 3D Model

a. Translate

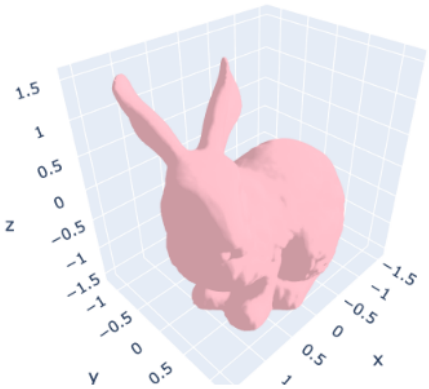


```python
center = (np.max(verts.T, axis=1) + np.min(verts.T, axis=1))/2

tran_x, tran_y, tran_z = verts.T[0] - center[0], verts.T[1] - center[1], verts.T[2] - center[2]
```

I calculate the center point of the object and compute translated coordinates by each point minus center, which means all vertices move and center coordinate move to (0,0,0)

b. Rotate



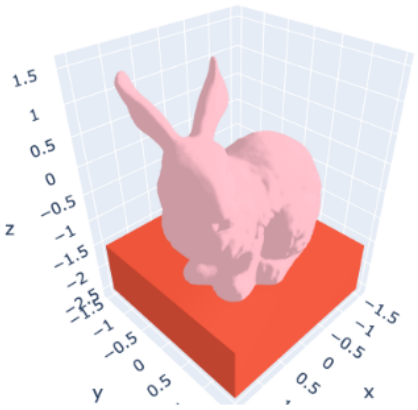
```python
rotateX = np.array([[1, 0, 0],
                    [0, np.cos(-np.pi/2), -np.sin(-np.pi/2)],
                    [0, np.sin(-np.pi/2), np.cos(-np.pi/2)]])
rotateY = np.array([[np.cos(0), 0, np.sin(0)],
                    [0, 1, 0],
                    [-np.sin(0), 0, np.cos(0)]])
rotateZ = np.array([[np.cos(np.pi), -np.sin(np.pi), 0],
                    [np.sin(np.pi), np.cos(np.pi), 0],
                    [0, 0, 1]])

rotate = np.dot(rotateX,rotateY)
rotate = np.dot(rotate,rotateZ)

tran_verts = (verts.T - center.reshape(3, 1)).T
rot_x, rot_y, rot_z = np.dot(rotate, tran_verts.T)
```

Firstly, I define the rotation matrix. RotateX means rotate x-axis, RotateY means rotate y-axis, and RotateZ means rotate z-axis. Then rotate is the final rotation matrix by dotting x, y, z axis matrix. Lastly, dot the rotation matrix with part(a) vertices coordinates, then I can get the rotation coordinates. In this problem, I rotate x-axis with -90 degree and z-axis with 180 degree.

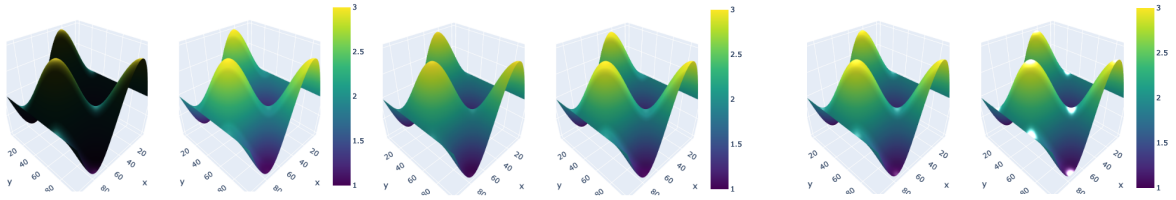
c. Add a cubic under the object




```
cubic = go.Mesh3d(
        # 8 vertices of a cube
        x=[-1.5, -1.5, 1.5, 1.5, -1.5, -1.5, 1.5, 1.5],
        y=[-1.5, 1.5, 1.5, -1.5, -1.5, 1.5, 1.5, -1.5],
        z=[-2.543336, -2.543336, -2.543336, -2.543336, -1.543336, -1.543336, -1.543336, -1.543336],
        colorscale=colorscale,
        # Intensity of each vertex, which will be interpolated and color-coded
        # i, j and k give the vertices of triangles
        i= [7, 0, 0, 0, 4, 4, 6, 1, 4, 0, 3, 6],
        j= [3, 4, 1, 2, 5, 6, 5, 2, 0, 1, 6, 3],
        k= [0, 7, 2, 3, 6, 7, 1, 6, 5, 5, 7, 2],
        showscale=False
    )
```

I add a cubic 3D object. Because the minimum z coordinates in part(b) is -1.543336, I set the top coordinate of the cubic as -1.543336 and the depth(z) is 1 to the negative direction. X and y coordinates are setting as -1.5 to 1.5.

d. Lighting



```python
x = np.linspace(-np.pi, np.pi, 100)
y = np.linspace(-np.pi, np.pi, 100)

Y, X = np.meshgrid(x, y)

Z = 2 + np.cos(X)*np.sin(Y)

fig = make_subplots(rows=1, cols=2, specs=[[{'type': 'surface'}, {'type': 'surface'}],])

trace1 = go.Surface(z=Z, colorscale='Viridis', lighting=dict(ambient=0.1, diffuse=0.8, specular=0.05))
trace2 = go.Surface(z=Z, colorscale='Viridis', lighting=dict(ambient=0.9, diffuse=0.8, specular=0.05))

fig.add_trace(trace1, row=1, col=1)
fig.add_trace(trace2, row=1, col=2)
# fig.show()
fig.write_image("output/2d_1.png")
```

```python
trace1 = go.Surface(z=Z, colorscale='Viridis', lighting=dict(ambient=0.8, diffuse=0.8, specular=0.2))
trace2 = go.Surface(z=Z, colorscale='Viridis', lighting=dict(ambient=0.8, diffuse=0.8, specular=2.0))
```

```python
trace1 = go.Surface(z=Z, colorscale='Viridis', lighting=dict(ambient=0.8, diffuse=0.1, specular=0.05))
trace2 = go.Surface(z=Z, colorscale='Viridis', lighting=dict(ambient=0.8, diffuse=0.9, specular=0.05))
```

According to the template, I adapt the different lighting information show in the spec.