Juan Carlos Díaz Rodríguez
Subgrupo U1
juancdia@ucm.es

Geometría Computacional
Práctica 4

20/04/2023

1. Introducción

En esta práctica estudiamos cómo realizar transformaciones sobre un conjunto de puntos en un espacio afín. En concreto estas transformaciones serán isométricas. Para visualizar estas aplicaciones se crearán dos animaciones de transformaciones dependientes de un parámetro t. Este tipo de aplicaciones son importantes en la Geometría Computacional dado que en muchas ocaciones nos interesa transformar imágenes sin deformarlas. En otros casos, querremos realizar una deformación, pero lo que aquí haremos será inmediatamente generalizable a otras aplicaciones lineales dada la matriz asociada correspondiente.

2. Material utilizado

Para la computación de estas tranformaciones se ha usado el lenguaje *Python*, así como las librerías `numpy` y `scipy.spatial` para la parte de cálculos matemáticos y `matplolib` para generar las animaciones. Para poder leer una imagen como un conjunto de valores RGB se ha empleado la librería `skimage`.

La parte más importante del código es cómo realizar la transformación afín. Sea $\mathcal{S}$ un conjunto de puntos afines, d su diámetro y C su centroide, debemos computar para cada punto $P \in \mathcal{S}$ la imagen de la aplicación

$$\begin{aligned} \phi_t \colon \mathbb{R}^3 &\to \mathbb{R}^3 \\ P &\mapsto C + R_{\theta,t}^{(xy)} \overrightarrow{CP} + v_t, \end{aligned}$$

donde

$$R_{\theta,t}^{(xy)} = \begin{pmatrix} \cos(t\theta) & -\sin(t\theta) & 0 \\ \sin(t\theta) & \cos(t\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad v_t = \begin{pmatrix} 0 \\ 0 \\ d \end{pmatrix}, \quad t \in [0,\, 1]$$

Para ello, si consideramos que los puntos vienen en un array donde la última coordenada representa las coordenadas afines, podemos hacer esta tranformación sobre todos los puntos del conjunto como

```
# Matriz de rotación vectorial
def rotacion(t, theta):
    return np.array([[np.cos(t*theta), -np.sin(t*theta), 0],\
                     [np.sin(t*theta),  np.cos(t*theta), 0],\
                     [       0       ,         0       , 1]])

def transformXYZ(S_vectors, C, d, theta, t):
  return  C +\
          np.tensordot(S_vectors, rotacion(t, theta), axes=(len(S_vectors.shape)-1,1)) +\
          t * np.array([0,0,d])
```

Considerando $\mathcal{S}$ como un tensor de n índices, donde el último corresponde con el índice del tensor vector, podemos generalizar la función para que sin importar la pinta del array de vectores de $\mathcal{S}$ hagamos la operación sobre todos los elementos y se devuelva de la misma forma.

Por otro lado, el cálculo del diámetro de un conjunto discreto de puntos puede ser computacionalmente duro si se calcula a fuerza bruta. Hay que tener en cuenta que en el primer apartado se emplean alrededor de 10000 puntos y en el segundo unos 40000. Si en vez de comparar todos los puntos con todos calculamos la envoltura convexa y después el diámetro de ese nuevo conjunto, reducimos el número de puntos a comparar a 234 y 129 respectivamente. Hay que remarcar que el cálculo de la envoltura convexa mediante la librería `scipy.spatial` es mucho más eficiente que hacer lo anterior a fuerza bruta.

3. Resultados

Una vez tenemos la parte matemática resuelta solo nos queda generar las animaciones.

Para el primer apartado, empleamos los datos proporcionados por la plantilla y usamos la función `animateXYZ` en la función de actualización. En la Figura 1 se pueden ver distintos instantes de la animación resultado.

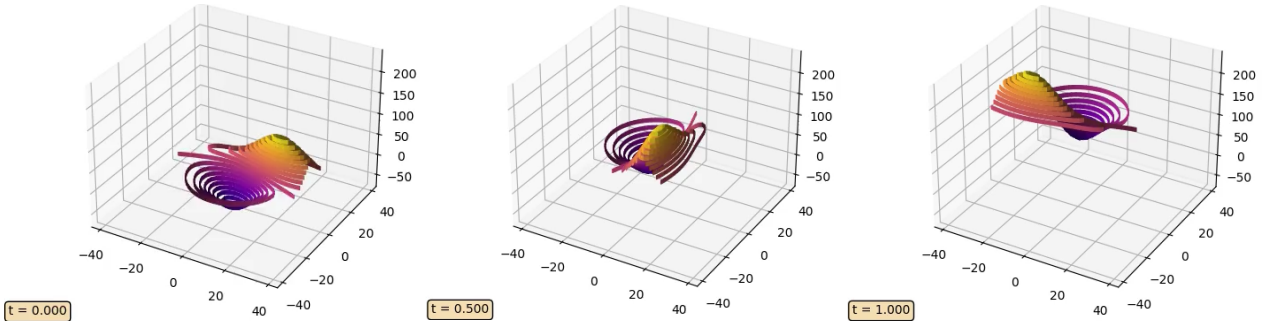

Figura 1: Fotogramas de la animación del *Apartado i)*

En el siguiente apartado, una vez leída la imagen como un array de valores RGB, realizamos un mallado de la imagen en las coordenadas (x, y) y tomamos como coordenada z el valor del color verde del pixel asociado. De esta manera, realizar una tranformación isométrica sobre el plano xy tendrá como resultado al reinterpretar las ternas $(x, y, verde)$ una imagen con el canal verde tranformado. En este caso particular hemos empleado una máscara booleana para seleccionar aquellos puntos de la malla con un valor de verde inferior a 240.

La animación, dado que la coordenada z es ficticia, ha sido hecha en 2D y en la Figura 2 se pueden encontrar de nuevo tres fotogramas distinguidos.

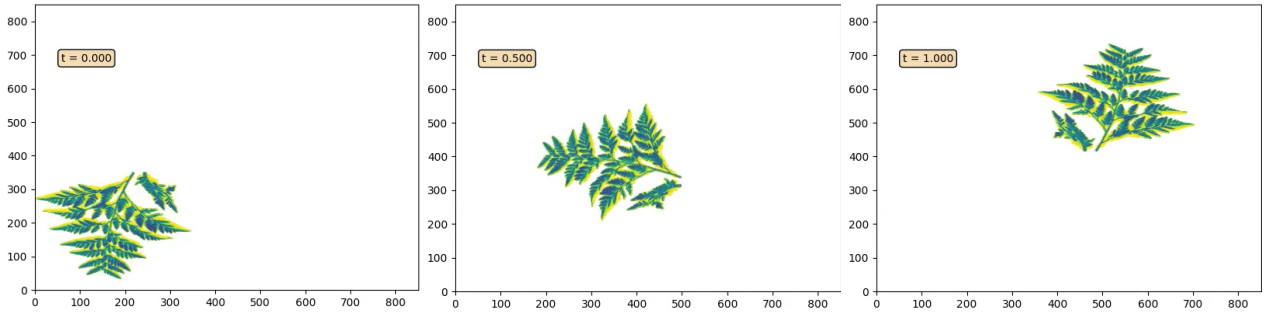

Figura 2: Fotogramas de la animación del *Apartado ii)*

4. Conclusión

De esta práctica podemos aprender lo siguiente:

- Para transformar imágenes, un meodo apropiado es añadir un mallado correspondiente a las coordenadas originales de los píxeles y realizar tranformaciones sobre el plano xy para después poder reinterpretar la imagen. Podemos llevar esta misma idea fuera del plano xy y así modificar los colores también, aunque quizá ya no sea tan deseable restringirse a aplicaciones isométricas. Puede ser interesante cómo se traduce el concepto de una isometría a una escala de colores.
- Dado que estas tranformaciones no se realizan sobre un único punto, sino sobre un conjunto de ellos, resulta de gran utilidad generalizar la aplicación de una matriz sobre un vector a la contracción de dos tensores. De esta forma el código es mucho más limpio y las librerías aprovechan mejor el *kernel* interno implementado en un lenguaje más eficiente que Python.
- Trabajar con imágenes es muy sencillo en Python, no solo por las librerías para representaciones gŕaficas, sino también por las facilidades ofrecidas por las máscaras booleanas de `numpy`. Es bien sabido que en el tratamiento de imágenes, las máscaras de colores juegan un papel fundamental, y estas son fácilmente interpretables como máscaras booleanas.

5. Código

Se muestra a continuación el script de Python empleado para obtener las animaciones:

```
import os
import numpy as np
import matplotlib as mpl
mpl.rcParams['animation.ffmpeg_path'] = '/usr/bin/ffmpeg'
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from scipy.spatial import ConvexHull
import matplotlib.cm as cm
from matplotlib import animation
from skimage import io, color

# Funciones auxiliares para poder guardar las animaciones
# tanto en gif como en mp4 u otros formatos de video similares
def save_gif(fileName, anim, fps):
    writergif = animation.PillowWriter(fps=fps)
    anim.save(fileName, writer=writergif)

def save_video(fileName, anim, fps):
    try:
        writervideo = animation.FFMpegWriter(fps=fps)
        anim.save(fileName, writer=writervideo)
    except FileNotFoundError as e:
        print(e)
        print("Check if you have downloaded FFMPEG and given the path to executable\
            or binary (depending on OS) to matplotlib on line 4")

# Apartado i)

fig = plt.figure()
ax = plt.axes(projection='3d')
props = dict(boxstyle='round', facecolor='wheat')
timelabel = ax.text(0.2,0.2,0.0, "", transform=ax.transAxes, ha="right", bbox=props)

# Tomamos los datos de prueba
X, Y, Z = axes3d.get_test_data(0.05)
# Lo pasamos a un formato de lista de puntos afines
points = np.empty((X.shape[0]*X.shape[1], 3))
shape = X.shape
points[:,0] = X.reshape(-1)
points[:,1] = Y.reshape(-1)
points[:,2] = Z.reshape(-1)
N = points.shape[0]
# Calculamos el centroide
C = 1/ N * np.sum(points, axis=0)
# Calculamos vectores de forma de point[i] = C + vector[i]
vectors = points - C

# Calculamos el diámetro
hull = ConvexHull(points)
d = 0
print(f'There are {len(hull.vertices)} vertices')
for i in range(len(hull.vertices)-1):
    for j in range(i, len(hull.vertices)):
        d = max(d,  (points[hull.vertices[i],0]-points[hull.vertices[j],0])**2+\
                    (points[hull.vertices[i],1]-points[hull.vertices[j],1])**2+\
                    (points[hull.vertices[i],2]-points[hull.vertices[j],2])**2)
d = np.sqrt(d)
```

```
print(f'd={d}')

v = np.array([0,0,d])
# Ponemos los puntos y los vectores como una matriz para poder
# representarlo en el contour
points = np.reshape(points, list(shape)+[3])
vectors = np.reshape(vectors, list(shape)+[3])

# Ponemos los límites del plot
maxXY = max(np.amax(np.abs(X)), np.amax(np.abs(Y)))*np.sqrt(2)
ax.set_xlim(-maxXY, maxXY)
ax.set_ylim(-maxXY, maxXY)
ax.set_zlim(np.amin(Z), np.amax(Z)+d)

# Matriz de rotación vectorial
def rotacion(t, theta=3*np.pi):
    return np.array([[np.cos(t*theta), -np.sin(t*theta), 0],\
                     [np.sin(t*theta),  np.cos(t*theta), 0],\
                     [       0       ,        0        , 1]])

def transformXYZ(t):
    # Es lo mismo que hacer points[i] = M * vectors[i] + [0,0,d], donde * es el
    # producto de una matriz por un vector y donde vectors[i] es un elemento del
    # espacio vectorial asociado al espacio afín
    return np.tensordot(vectors, rotacion(t), axes=(len(vectors.shape)-1,1)) + t*v + C

levels = 24
cmap = plt.cm.get_cmap('plasma')
cont = ax.contour(points[:,:,0], points[:,:,1], points[:,:,2], levels, extend3d=True, cmap = cmap)
def update1(t):
    global cont
    for c in ax.collections:
        c.remove()
    newPoints = transformXYZ(t)
    cont = ax.contour(newPoints[:,:,0], newPoints[:,:,1], newPoints[:,:,2], levels, extend3d=True, cma
    timelabel.set_text("t = {:.3f}".format(t))

precision = 0.002
time = np.arange(0,1,precision)
time = np.array(list(time)+[1.0]*10+list(time)[::-1])
ani = animation.FuncAnimation(fig, update1, frames=time,
                                        interval=int(1000/30), blit=False, repeat=False)

plt.show()
save_video(f'animation1.mp4', ani, 30)
print('Finished 1')

# Apartado ii)
img = io.imread('arbol.png')
xyz = img.shape

x = np.arange(0,xyz[0],1)
y = np.arange(0,xyz[1],1)
xx,yy = np.meshgrid(x, y)
xx = np.asarray(xx).reshape(-1)
yy = np.asarray(yy).reshape(-1)
z  = img[:,:,1]
zz = np.asarray(z).reshape(-1)

"""
Consideraremos sólo los elementos con zz < 240
```

```
"""
#Variables de estado coordenadas
x0 = xx[zz<240]
y0 = yy[zz<240]
z0 = zz[zz<240]/256.
N = x0.shape[0]

points = np.empty((N, 3))
points[:,0] = np.copy(x0)
points[:,1] = np.copy(y0)
points[:,2] = np.copy(z0)
# Calculamos el centroide
C = 1/ N * np.sum(points, axis=0)
# Calculamos vectores de forma de point[i] = C + vector[i]
vectors = points - C

# Calculamos el diámetro
hull = ConvexHull(points)
d = 0
print(f'There are {len(hull.vertices)} vertices')
for i in range(len(hull.vertices)-1):
    for j in range(i, len(hull.vertices)):
        d = max(d,  (points[hull.vertices[i],0]-points[hull.vertices[j],0])**2+\
                    (points[hull.vertices[i],1]-points[hull.vertices[j],1])**2+\
                    (points[hull.vertices[i],2]-points[hull.vertices[j],2])**2)
d = np.sqrt(d)
print(f'd={d}')
v = np.array([d,d,0])
#Variable de estado: color
col = plt.get_cmap("viridis")(np.array(0.1+z0))

fig, ax = plt.subplots()
props = dict(boxstyle='round', facecolor='wheat')
timelabel = ax.text(0.2,0.8, "", transform=ax.transAxes, ha="right", bbox=props)

# Ponemos los límites del plot
maxXY = max(np.amax(np.abs(x0)), np.amax(np.abs(y0)))*np.sqrt(2)+d
ax.set_xlim(0, maxXY)
ax.set_ylim(0, maxXY)

scat = ax.scatter(x0,y0,c=col,s=0.1)

def update2(t):
    scat.set_offsets(transformXYZ(t)[:,:2])
    timelabel.set_text("t = {:.3f}".format(t))
    return [scat, timelabel]

precision = 0.01
time = np.arange(0,1+precision,precision)
time = np.array(list(time)+[1.0]*10+list(time)[::-1])
ani = animation.FuncAnimation(fig, update2, frames=time,
                                        interval=int(1000/30), blit=True, repeat=False)

# plt.show()
save_video(f'animation2_fast.mp4', ani, 30)
```