Project report

Ana Marin , Yimin Deng

February 2022

1 Introduction

Star Wars is a beloved movie franchise created by George Lucas. His films have made a big impact on the geek culture: young or old everyone as heard about star wars at least once. One characteristic thing are the light saber battles. A thrilling futuristic sword fight , with phonic weapons and their unique sounds.

Have you imagined yourself using light saber? Achieving the Jedi Dream? We tried to dot it in our project: it consist of the simulation of a light saber via computer vision. The saber is a green stick that will be transformed into a light saber via computer vision. We capture the footage with a computer webcam. The project consist in 3 parts: the saber detection, the sound generation and the implementation of both parts.

2 Image

2.1 Methods

Our image processing can be divided into 5 parts.

2.1.1 color detection

Objects in green are detected with the percentage of 3 primary colors. The three additive primary colors are red, green, and blue. If a color is green in human eyes, the 3 values of 3 additive primary colors can be extracted and follow this criteria:

$$green > 1.1 * max(red, blue)$$

$green, red, blue =$ values of primary colors in this color

2.1.2 image smoothing

After color detection, the image may have rough edges jagged edges.Therefore, we shall use the Gaussian filter for image smoothing. Gaussian filter avoids overshoot of step functions and minimizes rise and fall time. Then the image is smoothed without too much blurring.

A Gaussian filter transfers input signal by convolution with a Gaussian function. A 2D Gaussian filter works as following function:

$$g(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1}$$

σ= the standard deviation of the Gaussian distribution

2.1.3 line extraction

After color detection (non-green parts are masked) and smoothing, we can easily extract the lines from the image using existing packages.Initially, a lot of lines, defined by a pair of point on it, are given.

A general equation of each line:

$$ax+by+c=0 \tag{2}$$

$$or$$

$$\begin{pmatrix} a \\ b \end{pmatrix}\begin{pmatrix} x & y \end{pmatrix}+c=0 \tag{3}$$

With a pair of point on a certain line we have already known, the normal vector of this line can be obtained with following equations:

$$w=\begin{pmatrix} a \\ b \end{pmatrix}=\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}\begin{pmatrix} \frac{x_1-x_2}{\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}} \\ \frac{y_1-y_2}{\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}} \end{pmatrix} \tag{4}$$

x_1, y_1= coordinates of the first point

x_2, y_2= coordinates of the second point

In addition, it has been noticed that w=$\begin{pmatrix} a \\ b \end{pmatrix}$ and $w=\begin{pmatrix} -a \\ -b \end{pmatrix}$ can be normal vectors of the same line. Uniformly, we let $b \geq 0$.

And we have

$$c=-\begin{pmatrix} a \\ b \end{pmatrix}\begin{pmatrix} x_1 & y_1 \end{pmatrix}$$

$$or$$

$$c=-\begin{pmatrix} a \\ b \end{pmatrix}\begin{pmatrix} x_2 & y_2 \end{pmatrix}$$

2.1.4 line clustering

In our project, the target object is a green stick, which has two main parallel lines. The two lines can be expressed by equations:

$$ax+by+c_1=0$$

$$ax + by + c_2 = 0$$

From part 2.1.3, we got a set of normal vectors with extraction of lines. A normal vectors can be defined by θ with:

$$e^{i\theta} = cos\theta + isin\theta = a + ib$$

where

$$0 \leq \theta < \pi$$

Therefore, normal vectors in the set are points lie on the unit circle of the complex plane. Below is an example.

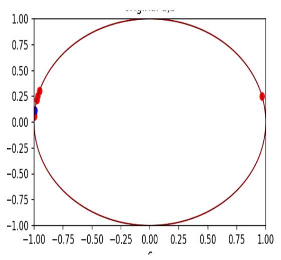

Figure 1: A set of normal vectors, displayed on the complex plane

Generally, only one cluster should appear in the diagram. But We find two in this diagram. It is because $\theta = 0$ and $\theta = \pi$ can define the same line. So some points that are actually very close to each other are going to be on two sides of the diagram. To deal with this problem, we transfer θ into 2θ,

$$\theta \to \theta' = 2\theta$$

and show new vectors in the following figure:

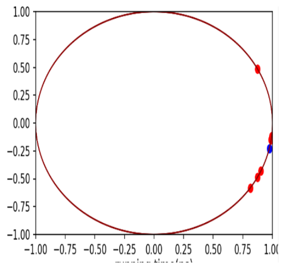

Figure 2: points transfered with $\theta' = 2\theta$

After transfering, the points will converge into a cluster. We get the normal vector of the stick by computing the center point of the cluster. Before computing the center point, it could be noticed that there are some points far away from the cluster, which, can be described as noise caused by the environment.

To eliminate noise, we first use Density-based spatial clustering of applications with noise(DBSCAN).It is a density-based clustering algorithm, marking points that are in low-density regions as outliers points.

After the processing of DBSCAN, a main cluster is extracted, and points out of this group is neglected. Figure 3 shows an example of this progress, where points in pink are neglected points. Then we get the center point of the main cluster, with k-means clustering. K-means clustering is to divide points into k clusters(In this case, k=1), and every point belongs to the cluster with the nearest cluster center. K-meaning cluster aims to minimize within-cluster variances. Mathematically,

For a set of points $(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}...)$, this algorithm is to find:

$$\min_{s} \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in S_i} \|\boldsymbol{x} - \boldsymbol{\mu_i}\| = \min_{s} \sum_{i=1}^{k} |S_i| Var S_i$$

$\boldsymbol{\mu_i}$= center points of each cluster

The blue point in Figure 3 is the center point we got.

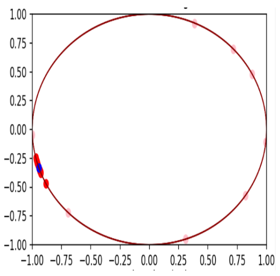

Figure 3: Point cluster filtered by DBSCAN clustering, with the center point in blue

Then, we get back θ:

$$\theta' \rightarrow \theta = \frac{1}{2}\theta'$$

With the normal vector, a new set of parameter 'c' should be computed again (since we uniformly use this vector to describe all lines):

$$c_i = -\begin{pmatrix} a \\ b \end{pmatrix} \begin{pmatrix} x_i & y_i \end{pmatrix}$$

DBSCAN and k-means clustering are applied as well. However, the number of clusters should be two, because the two main lines of the stick have different parameters 'c'. Figure 4 shows the clusters of parameter 'c'.

2.1.5 Sound making while translation and rotation

After the previous steps, we got the two main lines of the stick in every frame. Next, we are going to detect the rotation and translation of the stick. First, we

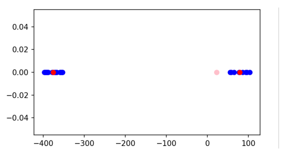

Figure 4: clusters of parameter 'c'

will judge whether it's rotating by following equation:

$$\frac{max(\Delta a, \Delta b)}{\Delta t} \geq v_0$$

v_0 is a value picked by us, which roughly equal to the maximum fluctuation that can occur at rest.

However, we find that between two frames, the time interval is very short, so the point wobble may not be ignored. So we try to smooth the data,

$$a_i = \alpha a_i - 1 + (1 - \alpha) a_i$$

$$b_i = \alpha b_i - 1 + (1 - \alpha) b_i$$

α= a parameter, which determine both curve smoothness and data loss.

And compute the rate of change every 10 frames.If the stick is rotating, a sound will be played. The volume of the sound depends on how fast it is rotating. The sound principle can be seen in Section 3.

If it is not rotating, the same solution is taken with parameter 'c', to determine whether it's in the translation and make sounds with its speed.

2.2 Implements

We implement this project in Python. A lot of library tools are involved, and some functions are written to implement different functions. In this section, we will list the involving files and library tools, the functions, the way to use it, and code implement in details.

2.2.1 File List

1. cluster_with_camera.py: Automatically open your computer camera, detect the moving of stick and make sounds.
2. cluster_with_picture.py: Extract and draw two main lines of the stick in a certain picture.
3. img_functions.py: Some functions used to process images.
4. SaberSound.py: Some functions used to make sounds, introduced with details in section 3.
5. green_sable.png: An example of picture with the stick.
6. saber_sound.wav: The sound file.

2.2.2 Library Tools

1. OpenCV: A real-time optimized Computer Vision library, used to process images (color detection, smoothing, line extraction...).
2. numpy: Mainly to deal with arrays, matrix and do some linear computation.
3. matplotlib: To display some data in figures.
4. sklearn.cluster: To implement DBSCAN and k-means clustering.
5. math: To do some basic math computation.
6. time: To get running time.

2.2.3 How to run it

Simply click 'run' in 'cluster_with_picture.py' , your computer camera will be opened and when you wave the green stick, sounds will be made depending on speeds. If you want to see the line clusters, you can change display_mode to True (line 35). If you want to see the change of parameter 'a', 'b' and 'c', you can change 'False' to 'True' (line 45).

Click 'run' in 'cluster_with_picture.py', a picture contains the green stick will appear, the two long line on the edge of stick will be drawn.

2.2.4 code implements in detail

First, open computer camera with:

```
_, img = cap.read()
```

Then we do color detection with RGB values read from frames by opencv.

```
import cv2
red    = img[..., 2]
green  = img[..., 1]
blue   = img[..., 0]
purple = np.maximum(red, blue)
mask = purple*1.1 < green
img_mask = np.zeros(img.shape[0:-1], dtype=np.uint8)
img_mask[mask] = 255
```

Next, do image smoothing with function 'GaussianBlur' in OpenCV:

```
img_blurred = cv2.GaussianBlur(img_mask, (filter_size, filter_size), 0)
```

Then extract lines with FastLineDetector in OpenCV:

```
fld_detector = cv2.ximgproc.createFastLineDetector()
fld_segments = fld_detector.detect(img_blurred)
```

And get clusters with functions in sklearn.cluster:

```
from sklearn.cluster import KMeans, DBSCAN

model=DBSCAN(eps=eps, min_samples=min_samples)
data_fit=model.fit(data)

w = np.average(matrix_ab, axis=0)
kmeans_c=KMeans(n_clusters=2,random_state=0).fit(distribution_c)
c1,c2=kmeans_c.cluster_centers_
```

Last, make sounds if the stick is moving:

```
angle_v=max((np.absolute(wx_arr[9])-np.absolute(wx_arr[0]))/(t_arr[9]-t_arr[
c_v=(np.absolute(c_arr[9])-np.absolute(c_arr[0]))/(t_arr[9]-t_arr[0])
if (angle_v)>0.05:
    saber_sound.set_value(min(1000*angle_v,100))
    time.sleep(0.1)
    print(angle_v)
else:
    if (c_v)>50:
        saber_sound.set_value(min(c_v*10,100))
        time.sleep(0.1)
```

3 Audio

3.1 Methods

In order to replicate the sound of a saber that changes with movements we will loop through a sound and change it's amplitude based on an external input. The sword movement will be modeled with an increase in the music volume. Usually the transition between one play and another are noticeable: that's why we will smooth the signal in the output. We plan on using a low pass filter following the equation:

$$a_i = \alpha a_{set} + (1 - \alpha) a_{i-1}$$

Where α is a number between 0 and 1, which determines both curve smoothness and data loss.$a_i =$ is the final value for the iteration i, a_{i-1} is the previous value a_{set} is the value set by the mutex.

3.2 Implementation

We will implement a SaberSound class. This script will read an play the audio file saber_sound.wav and change it's amplitude based on a external input. Since the sound program needs to run at the same time with cluster_with_camera.py , we will use threading. This allows us to execute two or more processes at the same time.
Indeed, we will create an audio thread with a buffer. A buffer is a reserved segment of memory that is used to hold the data being processed: they are set up to hold data coming in and going out. When the buffer has finished processing the sound it callbacks to the next audio sequence that needs to be modified. The buffer will be always filled with sound values and the tread can only be started or terminated with the methods start and stop.
Since we're using threading, there could be some collisions when setting the amplitude externally.That's why we will use a mutual exclusion object (mutex). It's a technique used to gain exclusive access to shared resources.
When the class is instantiated, it will read all the frames of our sbaer_sound.wav file and save it in the attribute sound_data . The class also posses an output stream: this is indeed the audio thread. Since we will be processing the data with numpy arrays we will use the output stream class provided in the package soundevice.
In order to make the transitions smoother we will use a low pass filter. It's value is set on 0,1 however it can be changed to any value in these order of magnitude.
.

3.2.1 How to run it

The program that you need to run is Saber_Sound.py, it contains the SaberSound class, and an example of it's implementation. In order to run the script you need to navigate to the FX directory and type on your terminal:

```
python Saber_Sound.py
```

You will hear the test case.

3.2.2 Library tools

These are the libraries used and it's purpose on the processing:

1. soundevice: Used for sound generation (OutputStram).
2. multitreading: Mutex implementation (Lock).
3. scipy.io: Reads the audio file.
4. numpy: Make arrays operations.
5. time: Used in the example for setting values.

3.2.3 Class functions and script structure

When the class SaberSound is instantiated, it reads the audio file and stores it's values and sampling frequency. The class also posses an OutputStream with callback on_sound. Here we enumerate the methods of the class:

1. start: starts the audio tread (OutputStream).
2. stop: stops the audio tread (OutputStream).
3. on_sound: Gives the buffer the sound provided by the file and modifies it's amplitude based on the target value received in the thread.
4. set_end_idx: Set the end index of the audio array in a way that the file can be read in a loop.
5. set_value : sets the value of the amplitude from an external input.

3.3 Results

The results can be heard on Saber_Sound.py and in cluster_with_camera.py.