**Tracking the Speed of a Ball through a Scene**

The goal of this project was to determine the speed of a ball throughout a video sequence. In order to determine the speed of the ball, the position of the ball in real space (x,y,z) needed to be found. A pipeline of techniques were used to determine the (xy,z) coordinates of the ball in each frame of the video. This pipeline ran for each frame in the video. Each frame ran independently from all other frames in the video except the first frame. The techniques used in the pipeline were color space conversion, background subtraction, connected components analysis, region analysis, and stereo vision with one camera. Once the (x,y,z) coordinates of the ball were found at each frame it is trivial to determine the speed of the ball.

First, the pipeline was fed the raw RGB frame (Image 1). This frame was converted to the YIQ color space (Image 2). This was done to make the pipeline more robust to shadows and reflections. The Y color channel is dropped since Y contains intensity information (Davis, Region Extraction Slides). Only, the I and Q channels are used for now on.

Next, image difference is done between the current frame and the first frame in the video. The team used one of the “General Cases of Motion”; “Still camera, single moving object, constant background” (Davis, Lecture slide on Motion). Using this case of motion, the only moving object should be the ball. Therefore, image differencing in simplified. The difference value is calculated from Equation 1.

Equation 1:

This is the Euclidean distance between the first frame’s IQ channels and the current frame’s IQ channels. This operation was performed pixel-wise. This operation produces a difference matrix (Image 3) that is the same size as the original image. The absolute value of the difference matrix is calculated. A threshold with a value of T1 is used to convert the difference matrix into a binary matrix. A zero represents background. A one represents a large change between the first and current frame.

All the connected components are found in the binary matrix. The connected component with the largest area is marked as the ball. The rest of the connected components are set to zero. This is done to remove noise that the image differencing does not catch. No object larger than the ball can be moving since the ball is assumed to be the largest. Some frames in the scene do not have a ball. Therefore, if the largest connected component’s area is not greater than T2 the frame is marked as not containing a ball and is discarded.

The X-centroids and Y-centroids are calculated for the connected component. These points are used represent the center pixel of the ball in the frame. The midpoint of the heigh and the width of the connected component were not used because noise around the edge of the ball could skew the real position of the ball much more than using the centroids. At this point in the pipeline smoothing of the shape using Closing was experimented with. Closing was not used since it had no significant effect on the shape of the binary image and added computation time. Closing was not robust or reliable enough remove noise around the ball consistently.

The height of the connected component was calculated to determine the diameter of the ball in pixels. The height was used because the ball mainly travels horizontally throughout the video. Therefore, the height of the connected component was not affected by motion blur like the width of the connected component was. Image 4 shows the trajectory of the ball through the scene. It is easy to see that the ball moved very little vertically.

Using the X-centroid, Y-centroid, and diameter, the real (x,y,z) values of the ball in 3-D space can be approximated. Only one camera was used, therefore traditional stereo vision could not be used. Therefore a few additional parameters are needed; the diameter of the ball in inches, the camera horizontal field of view in degrees, and the camera’s vertical field of view in degrees.

First, the distance of the ball from the camera was calculated using Equation 2 and 3, where c is circumference of the camera’s horizontal field of view at the position of the ball in inches, w is the width of the frame in pixels, dp is the diameter of the ball in pixels in the frame, din is the diameter of the ball in inches, d is the distance of the ball from the camera in inches, and FOVx is the camera’s horizontal field of view in degrees.

Equation 2:

Equation 3:

This distance can also be calculated using the vertical field of view and the heigh of the frame in pixels.

Next, the horizontal angle of the position of the ball with respect to the camera was calculated. Let this value be . This can be calculated by using equation 4, where x is the center of the ball in pixels from the left side of the image, and w is the width of the frame in pixels.

Equation 4:

Similarly, the vertical angle can be calculated using the vertical center of the ball the vertical field of view, and the height of the frame in pixels. Let this value be Θ.

d, , Θ, are the polar coordinates in the 3D coordinate system. It is trivial to convert these values to (x,y,z) cartesian value. The frame of reference is the camera. The camera’s position is (0,0,0).

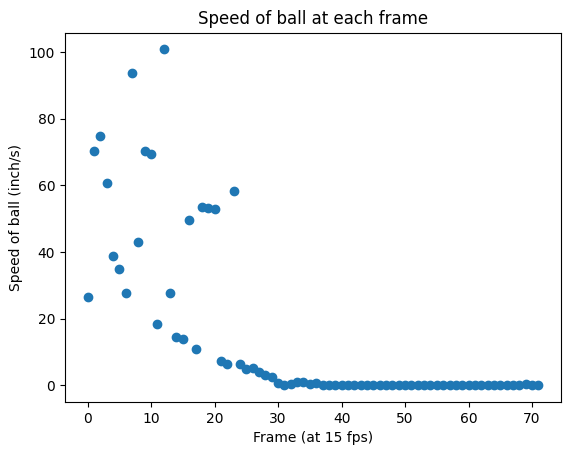
These cartesian coordinates represent the position of the ball in real 3D space. Using these positions, it is trivial to calculate the distance of the ball between frames and the speed of the ball from frame-to-frame. The team used a frame rate of 15 fps.

Figure 1

The speed of the ball between each frame was calculated and shown in Figure 1. This graph only shows the frames when the ball is in view. A smooth decreasing pattern can be seen from frame 0 to frame 30. There is a lot of noise when the ball is moving relatively fast. The ball’s speed should decrease rapidly and in a non-increasing fashion. Several things could account for the noise between frame 0 and 30. The position of the ball could have been incorrectly found due to motion blur and other noise. The height of the connected component could not have been the best approximation for the ball’s radius in each frame. Also, the calculations to convert from (diameter, x-pixel, y-pixel) to (x,y,z) in real space is an approximation and therefore could contribute to the noise.

The sum of the distances traveled between each frame was 74.2 inches. The distance between the first and last frame was 52.7 inches. Ideally these numbers would be the same since the ball moved in a straight line the whole time. The extra 21.5 inches found between the frames can be attributed to the reasons listed in the previous paragraph.

Another problem encountered was finding a lossless file type and software to save the video files into. The team was unable to find software to create and save videos to lossless file type on the old iPhone SE and a Dell XP laptop. Therefore, the team used a Raspberry Pi 4 and the Pi-camera python module to record the video and save it as the lossless H264 file type. This was critical since the background subtraction used needed exact precision between frames; with lossy video files that is not guaranteed.

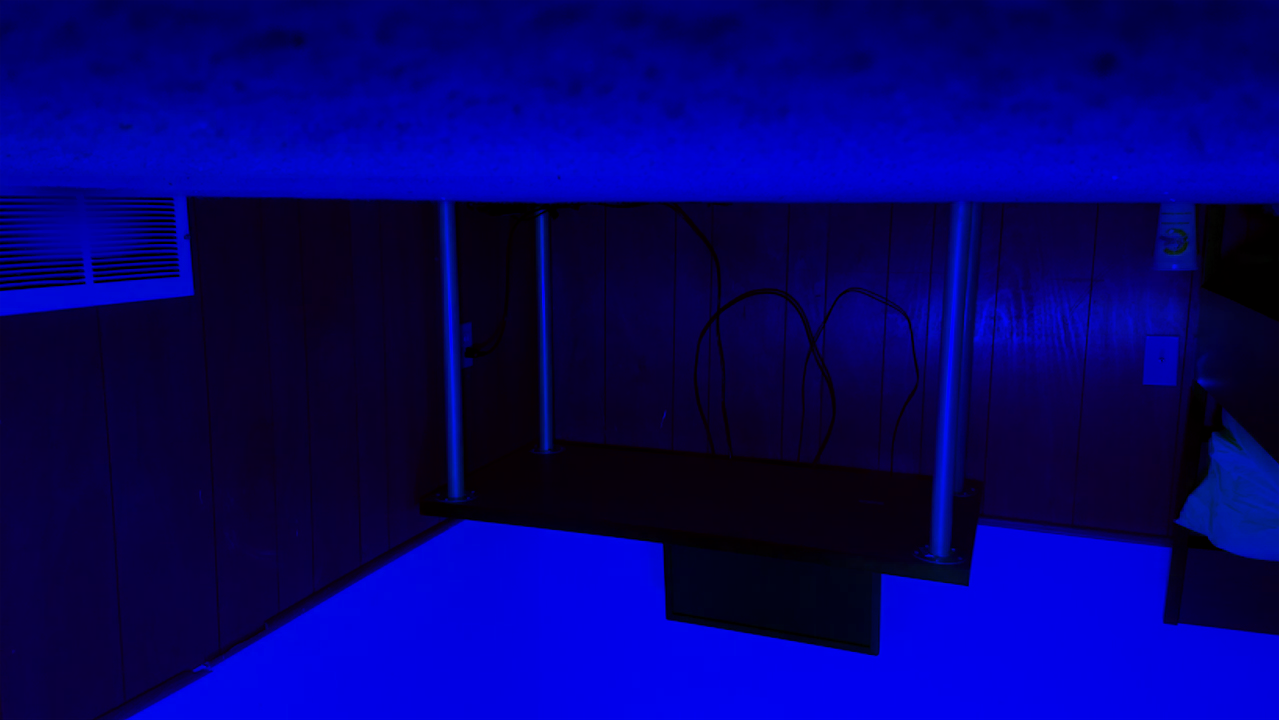
Another lesson the team learned was that creating the right video makes the pipeline produce more reliable results. The video was taken in a carpeted artificially well-lit static room. Nothing in the room changed or was reflective, except for the ball. The carpet was used since there is no reflection off the floor. This was critical to making background subtraction work. Even small shadows or reflections off wooden surfaces can be detected by the background subtraction algorithm and distort the real position of the ball. Although the difference algorithm used was designed to remove reflections and shadows, some reflections were still detected. Images 5 and 6 show how a reflection off a wooden surface ruins the algorithm.

Moving forward there are several things the team would like to do to make the pipeline better. First, the pipeline could be made to work in real time. Currently, the video is recorded and saved by one script then is used and analyzed by a separate script later. It would be easy to combine the scripts and analyze the frames in real time as they are recorded. Also, the processing of each frame is independent of all other frames except the first one, therefore each frame can be processed in parallel. This can make the algorithm extremely fast. Secondly, the team would like to make the algorithm more robust to shadows, reflections, and small changes in the background (like in an outside setting). To do this, a different background algorithm will need to be used that uses the mean and standard deviation of each pixel from a series of frames. Thirdly, the team would like to make the algorithm robust to all ball trajectories. Currently, this algorithm works well when a ball moves only horizontally, since the height of the ball is used as the diameter. If a ball moves vertically through the scene, motion blur in the vertical direction will become an issue. The team must find the minimum distance through the connected component at any orientation. Finally, the team wants to experiment with faster balls. The team never experimented with a fast-moving ball and experienced how extreme motion blur will affect the efficacy of the algorithm.

All work was completed by Ryan Strotman.

**Appendix**

****Image 1: RGB Image

****Image 2: One of the channels in the YIQ Image

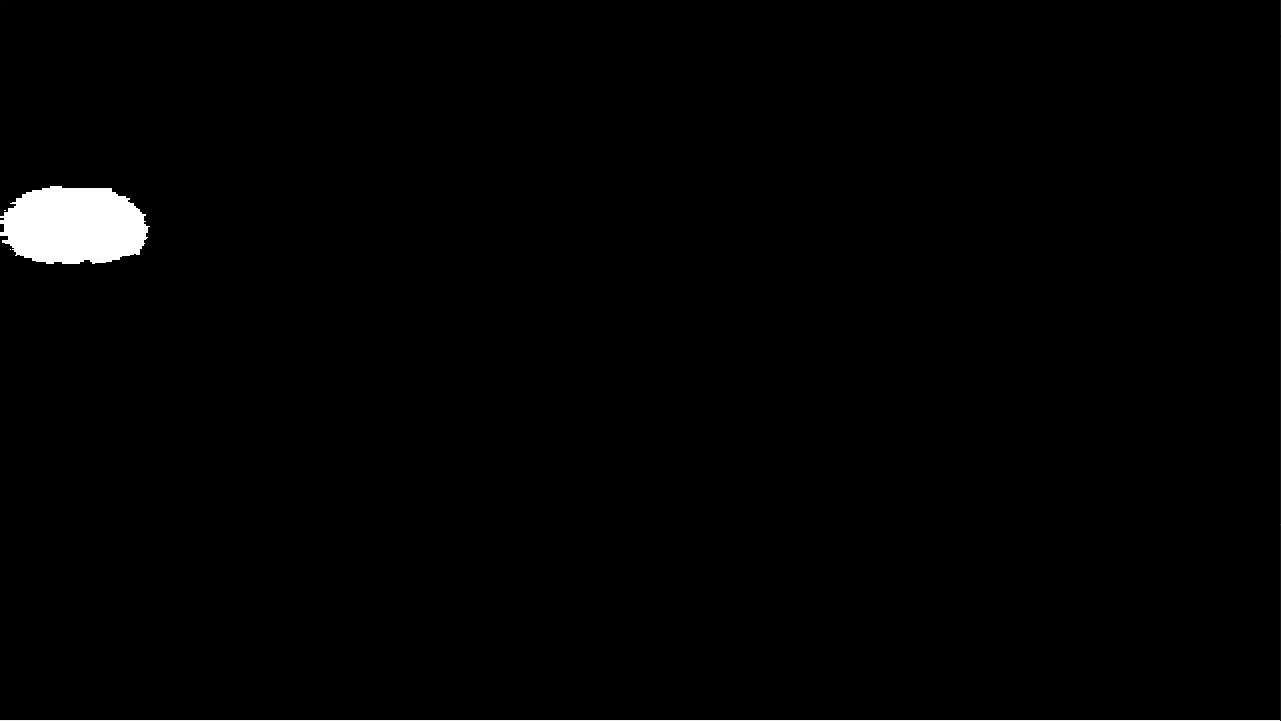
Image 3: Image Difference

Image 4: Sum of all Image Difference Frames



Image 5: RGB Image Image 6: Image difference calculated from Image 4

Constants

|  |  |  |
| --- | --- | --- |
| Variable | Value | Description |
| T1 | .05 | Threshold for image differencing |
| T2 | 30 | Threshold for area of a connected component |

**Code**

**Analysis Script (python)**

from time import sleep

import cv2

import numpy as np

from skimage.color import rgb2yiq

from scipy.ndimage import binary\_closing

import math

import matplotlib.pyplot as plt

def Standardize(image):

xMin = image.min()

xMax = image.max()

return (image - xMin) /(xMax - xMin)

#return true if ball in image

#return false if no ball

def RemoveSmallComponents(image):

nb\_components, output, stats, centroids = cv2.connectedComponentsWithStats(difference.astype(np.uint8), connectivity=8)

if nb\_components < 2:

return False, None

index = np.argsort(stats[:,4])[-2] #take the second largest, the first is the background

if stats[:,4][index] < 30:

#area needs to be greater than 30 pixels

return False, None

img2 = np.zeros((output.shape))

for i in range(0, nb\_components):

if i == index:

img2[output == i] = 1

return True, img2

#YIQ image

#range of I and Q coordinates are -.59 to .59

def ImagedDifference(img1, img2, threshold):

difference = img1 - img2

difference = np.square(difference)

difference = difference[:,:,1] + difference[:,:,2]

difference = np.sqrt(difference)

difference[difference > threshold] = 1

difference[difference <= threshold] = 0

return difference

#theta is angle from straing up Z axis

#rho is angle from x axis; 90 at y axis

def PolarToCartesian(r,theta,rho):

thetaRad = math.radians(theta)

rhoRad = math.radians(rho)

x = r \* math.sin(thetaRad) \* math.cos(rhoRad)

y = r \* math.sin(thetaRad) \* math.sin(rhoRad)

z = r \* math.cos(thetaRad)

return x,y,z

#convert (x,y,diameter), to (x,y,z), (0,0,0) is camera

def convertFromDiameterToCartesion(x,y,diameter):

#constants

din = 2.75 #diameter of ball in inches

w = 1280 #width of images in pixels

h = 720 #heigh of images in pixels

xFov = 62.3 #horizontal field of view in degrees

yFov = 48.8 #Vertical field of view in degrees

circumference = w / diameter \* din #circumfrence in inches of horizontal plane

radius = circumference / (2 \* math.pi) \* 360 / xFov #distance from camera

rho = 90 + xFov/2 - x/w \* xFov #angle from x axis ball is at

theta = 90 - yFov/2 + y/h \* yFov #angle from z axis (pointing straing up)

xFinal,yFinal,zFinal = PolarToCartesian(radius,theta,rho)

return xFinal, yFinal, zFinal

def Magnitude(x,y,z):

return math.sqrt(x \* x + y \* y + z \* z)

#points are (x,y,z)

def Distance(firstPoint, secondPoint):

return Magnitude(firstPoint[0] - secondPoint[0], firstPoint[1] - secondPoint[1], firstPoint[2] - secondPoint[2])

def CreateMask(radius, xCenter, yCenter):

w = 1280 #width of images in pixels

h = 720 #heigh of images in pixels

x = np.arange(0, w)

y = np.arange(0, h)

arr = np.zeros((y.size, x.size))

return (x[np.newaxis,:]-xCenter)\*\*2 + (y[:,np.newaxis]-yCenter)\*\*2 < radius\*\*2

def ShowBallCircle(difference, x, y, diameter):

mask = CreateMask(diameter/2, x,y)

differenceCopy = difference.copy()

differenceCopy[mask] = .5

cv2.imshow("Center " + str(numFrames), differenceCopy)

return

'''

#test convertFromDiameterToCartesion function

x = 100

y = 50

diameter = 3

print(convertFromDiameterToCartesion(x,y,diameter))

'''

'''

#test PolarToCartesan function

r = 1

theta = 100

rho = 170

print(PolarToCartesian(r,theta,rho))

'''

frameRate = 15 #frames per second

timeBetweenFrames = 1 / frameRate #in seconds

video\_path = "./Video/video4.h264"

video\_object = cv2.VideoCapture(video\_path)

yiqFrames = []

rgbFrames = []

frameStats = []

differenceFrames = []

ret,frame = video\_object.read()

numFrames = 0

PrintFrame = 33

while(ret) :

numFrames += 1

rgbFrames.append(frame)

yiqFrames.append(rgb2yiq(frame))

# Press Q on keyboard to exit

#this makes the video play for some reason

if cv2.waitKey(25) & 0xFF == ord('q'):

break

cv2.imshow("Frame", frame)

#140 250 -> for video1

#60 150 -> for video2

#70 150 -> for video 3

#40 70 -> for video 4

#print for report

if PrintFrame == numFrames:

cv2.imshow("RGB " + str(numFrames), rgbFrames[-1])

cv2.imshow("YIQ " + str(numFrames), yiqFrames[-1])

difference = ImagedDifference(yiqFrames[0], yiqFrames[-1], .05)

if PrintFrame == numFrames:

cv2.imshow("Raw Frame Difference: " + str(numFrames),difference)

valid, difference = RemoveSmallComponents(difference)

if valid:

if PrintFrame == numFrames:

cv2.imshow("One Connected Component: " + str(numFrames), difference)

#difference = binary\_closing(difference, iterations=10).astype(float) #this has neglible effect on outcome

if PrintFrame == numFrames:

cv2.imshow("After closing: " + str(numFrames), difference)

nb\_components, output, stats, centroids = cv2.connectedComponentsWithStats(difference.astype(np.uint8), connectivity=8)

diameter = stats[1,3] #the max height of object in pixels

x = centroids[1][0]

y = centroids[1][1]

stats = (numFrames,x,y,diameter)

if PrintFrame == numFrames:

ShowBallCircle(difference,x,y,diameter)

print(stats)

frameStats.append(stats)

differenceFrames.append(difference)

else:

print("Frame " +str(numFrames) + " not valid")

ret,frame = video\_object.read()

differenceCombined = differenceFrames[0]

for difference in differenceFrames:

differenceCombined = differenceCombined + difference

cv2.imshow("Difference Combined", differenceCombined)

results = []

DiameterArray = []

for stats in frameStats:

results.append(convertFromDiameterToCartesion(stats[1], stats[2], stats[3]))

DiameterArray.append(stats[3])

print(DiameterArray)

print("Total distance traveled: " + str(Distance(results[0], results[-1])))

DistanceArray = [] #distance between ith frame and ith + 1 frame

TotalDistanceArray = [] #distance traveled from 0th frame to ith frame

speedArray = [] #speed between ith frame and ith + 1 frame

frameArray = [] #dummy array needed for graphing

totalDistance = 0

for i in range(len(results) - 1):

distance = Distance(results[i], results[i+1])

totalDistance += distance

speed = distance / timeBetweenFrames

DistanceArray.append(distance)

TotalDistanceArray.append(totalDistance)

speedArray.append(speed)

frameArray.append(i)

print("Total Distance traveled between first and last frame: " + str(Distance(results[0], results[-1])))

print("Accumulative distance traveled:" + str(totalDistance))

plt.title("Accumulative Distance Travled")

plt.xlabel("Frame (at 15 fps)")

plt.ylabel("Distance (inches)")

plt.scatter(frameArray,TotalDistanceArray)

plt.show()

plt.title("Distance Traveled between each Frame")

plt.xlabel("Frame (at 15 fps)")

plt.ylabel("Distance (inches)")

plt.scatter(frameArray, DistanceArray)

plt.show()

plt.title("Speed of ball at each frame")

plt.xlabel("Frame (at 15 fps)")

plt.ylabel("Speed of ball (inch/s)")

plt.scatter(frameArray,speedArray)

plt.show()

cv2.waitKey()

#.05 is best for threshold

#d -> 4 3/4 inches

'''

for threshold in range(4,10):

threshold = threshold / 100

difference = ImagedDifference(yiqFrames[0], yiqFrames[-1], threshold)

cv2.imshow("difference" + str(threshold), difference)

'''

**Video Capture Script (python)**

from picamera import PiCamera

from time import sleep

import cv2

import numpy as np

camera = PiCamera()

resolution = (3280,2464)

camera.framerate = 15 #set to 15 fps

videoPath = './Video/video4.h264'

camera.start\_preview()

sleep(3)

print("starting recording")

camera.start\_recording(videoPath)

camera.wait\_recording(7)

camera.stop\_recording()

print("end of recording")