COMPUTER VISION FINAL PROJECT REPORT

NAVIGATE A GAME USING MOTION DETECTION BY APPLYING THE OPITCAL FLOW ALGORITHM OF LUCAS KANADE

Titash Mandal (tm2761)
Saniya Alekar (ssa428)
Sen Huo (sh4473)
Hanzhao Wang (hw1445)

INTRODUCTION TO OPTICAL FLOW

1.1 Introduction

Optic flow is defined as the change of structured light in the image, e.g. on the retina or the camera's sensor, due to a relative motion between the eyeball or camera and the scene. Motion estimation is one of the corner stones of computer vision. It is widely used in video processing to compress videos and to enhance video qualities, and used in 3D reconstruction, object/event tracking, segmentation, and recognition. Optical flow is the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer and a scene. The goal of optical flow estimation is to compute an approximation to the motion fields from time-varying image intensity.

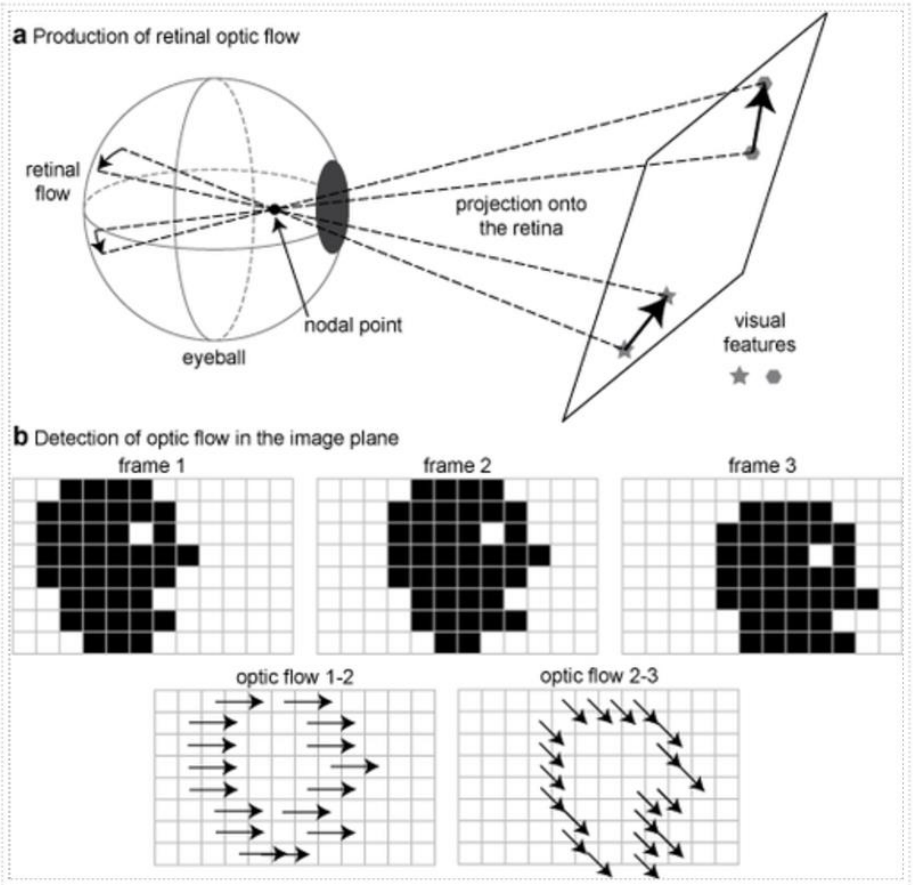


Fig. 1.1: Image showing the production and detection of optic flow.

Sequences of ordered images allow the estimation of motion as either instantaneous image velocities or discrete image displacements. The optical flow methods try to calculate the motion between two image frames which are taken at times t and t + Δt at every voxel position. These methods are called differential since they are based on local Taylor series approximations of the image signal; that is, they use partial derivatives with respect to the spatial and temporal coordinates.

For a 2D+t dimensional case (3D or n-D cases are similar) a voxel at location (x,y,t) with intensity I(x,y,t) will have moved by Δx, Δy and Δ t between the two image frames, and the following brightness constancy constraint can be given:

Assuming the movement to be small, the image constraint at I(x,y,t) with Taylor series can be developed. Ignoring higher-order terms in the Taylor series, and then substituting the linear

Approximation we obtain:

$$\nabla I\,(x, y, t) \cdot u + I_t\,(x, y, t) = 0$$

Tracking points of constant brightness can also be viewed as the estimation of 2D paths x(t) along which intensity is conserved:

$$I(\vec{x}(t), t) \;=\; c\;,$$

the temporal derivative of which yields:

$$\frac{d}{dt}\, I(\vec{x}(t), t) \;=\; 0$$

Expanding the left-hand-side using the chain rule gives us,

$$\frac{d}{dt}\, I(\vec{x}(t), t) \;=\; \frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t}\frac{dt}{dt} \;=\; \nabla I \cdot \vec{u} + I_t\;,$$

1.2 Optical flow methods

Here we simply use I_x, I_y, I_t to derive the normal vector. I_x, I_y are the spatial gradients in the X axis and Y axis respectively and I_t is the temporal gradient.

Notation

$$I_x u + I_y v + I_t = 0$$

$$\nabla I^T \mathbf{u} = -I_t$$

$$\mathbf{u} = \begin{bmatrix} u \\ v \end{bmatrix} \quad \nabla I = \begin{bmatrix} I_x \\ I_y \end{bmatrix}$$

We get at most "**Normal Flow**" – with one point we can only detect movement perpendicular to the brightness gradient. Solution is to take a patch of pixels around the pixel of interest which leads to our second approach.

1.2.1 Optical flow calculated over a pixel neighborhood

We can calculate the normal flow by solving the equation system as below,

$$\nabla E.v + \frac{\partial E}{\partial t} = 0$$

Using the following method, we can determine the velocity vectors for the surface concerned. The method is shown as below,

$$\begin{aligned} Av + b &= 0 \\ A^T Av &= -A^T b \\ v &= -(A^T A)^{-1} A^T b \\ C &= A^T A \end{aligned}$$

The columns of A are the x and y components of the gradient ∇E and b is a column vector of the t gradient component of E, Et

1.2.2 The Lucas and Kanade Method

The Lucas Kanade method, also known as sparse optical flow, calculates the displacement vectors of individual features rather than tracking all the pixels within a frame and rendering a full motion vector field. This method uses gradient information obtained from consecutive frames to search along the optimal path for a region that best matches the desired feature.

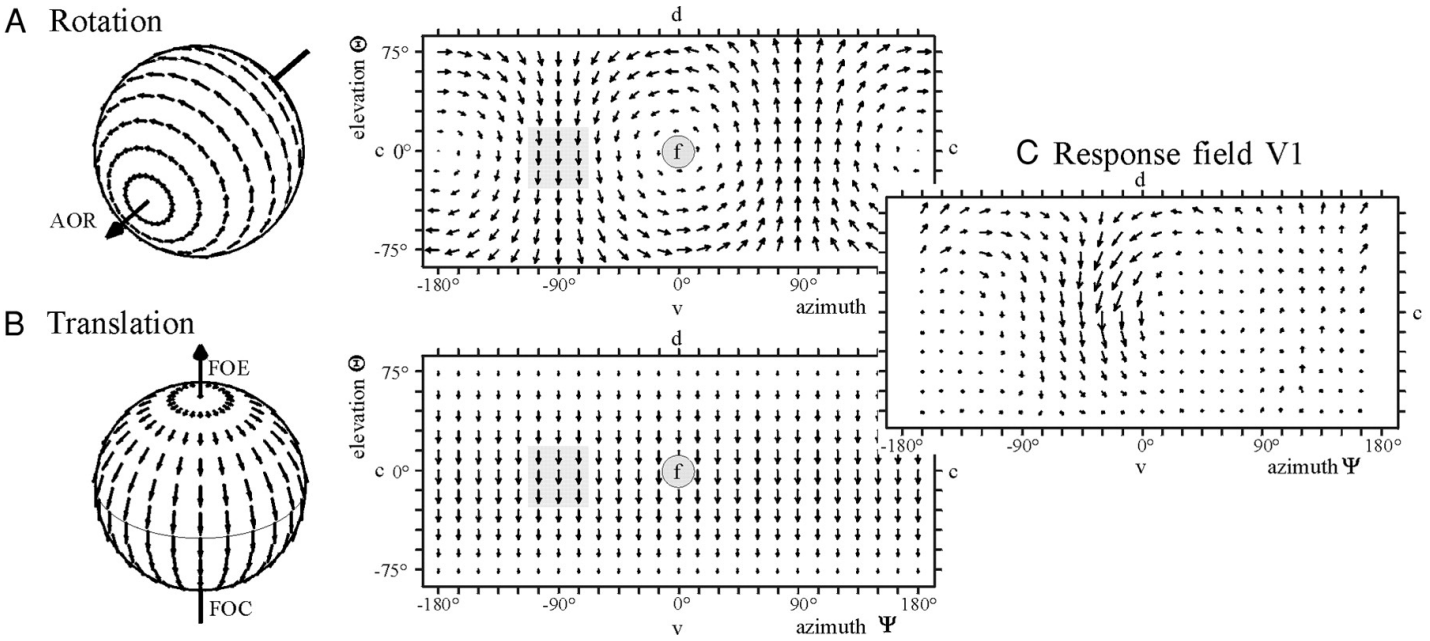


Fig. 1.2: Image showing optical flow induced during a rotation about a horizontal (*A*) or a upward translation along the vertical body axis (*B*).

The displacement vector is updated iteratively until it converges to a match with a sufficiently small error. It will otherwise diverge and fail to locate the target feature in the current frame. By registering a set of features from frame to frame, it is possible to maintain track of the target if it does not demonstrate any significant deformation and

lighting is relatively constant. Features which cannot be registered in a new frame are eliminated and replaced by good features nearby.

The coordinate frame for our tracking system is shown in the figure below. If the image plane is defined by the u and v axes, the coordinates are used to describe the position of a tracked feature in the ith frame of a sequence of images. As an estimate for the location of the target region, we average the coordinates of the tracked features. This is sometimes referred to as the middle mass.

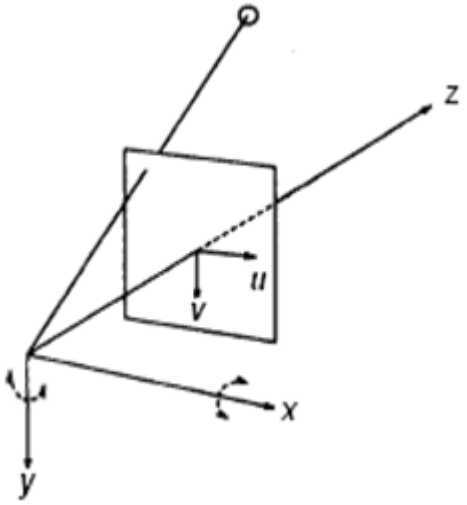


Fig. 1.3: Image showing the three-co-ordinate axis and the velocity vectors in x and y direction.

The Lucas–Kanade method assumes that the displacement of the image contents between two nearby instants (frames) is small and approximately constant within a neighborhood of the point p under consideration. Thus the optical flow equation can be assumed to hold for all pixels within a window centered at p. Namely, the local image flow (velocity) vector (V_{x},V_{y}) must satisfy

$$
\begin{aligned}
I_x(q_1)V_x + I_y(q_1)V_y &= -I_t(q_1) \\
I_x(q_2)V_x + I_y(q_2)V_y &= -I_t(q_2) \\
&\vdots \\
I_x(q_n)V_x + I_y(q_n)V_y &= -I_t(q_n)
\end{aligned}
$$

where q{1},q{2},... ,q{n}} q{1},q{2},...,q{n} are the pixels inside the window, and I{x}(q{i}),I{y}(q{i}),I{t}(q{i}) I{x}(q{i}),I{y}(q{i}),I{t}(q{i}) are the partial derivatives of the image I with respect to position x, y and time t, evaluated at the point {q{i}} q{i} and at the current time.

These equations can be written in matrix form {Av=b}, where

$$
A = \begin{bmatrix} I_x(q_1) & I_y(q_1) \\ I_x(q_2) & I_y(q_2) \\ \vdots & \vdots \\ I_x(q_n) & I_y(q_n) \end{bmatrix} \qquad v = \begin{bmatrix} V_x \\ V_y \end{bmatrix} \qquad b = \begin{bmatrix} -I_t(q_1) \\ -I_t(q_2) \\ \vdots \\ -I_t(q_n) \end{bmatrix}
$$

This system has more equations than unknowns and thus it is usually over-determined. The Lucas–Kanade method obtains a compromise solution by the least squares principle. Namely, it solves the 2×2 system

$$A^T A v = A^T b \text{ or}$$
$$\mathbf{v} = (A^T A)^{-1} A^T b$$

where A^{T} is the transpose of matrix A. That is, it computes

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \sum_i I_x(q_i)^2 & \sum_i I_x(q_i) I_y(q_i) \\ \sum_i I_y(q_i) I_x(q_i) & \sum_i I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x(q_i) I_t(q_i) \\ -\sum_i I_y(q_i) I_t(q_i) \end{bmatrix}$$

where the central matrix in the equation is an Inverse matrix. The sums are running from i=1 to n.

The matrix {A^{T}A} is often called the structure tensor of the image at the point p.

Discussion of the Lucas Kanade method - The Lucas-Kanade algorithm makes a "best guess" of the displacement of a neighborhood by looking at changes in pixel intensity which can be explained from the known intensity gradients of the image in that neighborhood. For a simple pixel, we have two unknowns (u and v) and one equation (that is, the system is underdetermined). We need a neighborhood to get more equations. Doing so makes the system overdetermined and we must find a least squares solution. The LSQ solution averages the optical flow guesses over a neighborhood. The Lucas-Kanade algorithm is an efficient method for obtaining optical flow information at interesting points in an image (i.e. those exhibiting enough intensity gradient information). It works for moderate object speeds.

2. Project formulation and implementation:

2.1: Detecting the Webcam of the laptop

The first step in the programming of our project was to detect the web camera of the laptop. Since our goal was to run a game on a laptop and navigate through it with our hands, it was necessary to access the laptop camera to live stream the video feed for further processing. We used the function cv2.videoCapture(0) to connect to the laptop's camera and extract the video feed.

2.2: Implementing the Lucas Kanade Optical flow algorithm

We have included the optical flow algorithm in our code with the help of Python's CV library function cv2.calcOpticalFlowPyrLK(). This method provided by the openCV library in Python is used to calculate the optical flow for sparse features set using the iterative Lucas Kanade method with pyramids. The most important parameters being the prevImg, nextImg, prevPts, nextPts which tell us about the points in the next frame based on data in the current frame. We can also alter the minEigThreshold parameter according to our requirements. For example, if we need to detect very fast movement of an object.

The function consists of the following highly useful parameters:

```
calcOpticalFlowPyrLK ( InputArray        prevImg,
                       InputArray        nextImg,
                       InputArray        prevPts,
                       InputOutputArray  nextPts,
                       OutputArray       status,
                       OutputArray       err,
                       Size              winSize = Size(21, 21),
                       int               maxLevel = 3,
                                         criteria =
                       TermCriteria      TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 0.01),
                       int               flags = 0,
                       double            minEigThreshold = 1e-4
                     )
```

Fig. 2.1: Image showing the arguments supplied to the calcOpticalFlowPyrLK() method
Image Source: docs.opencv.org

The following figure provides a detailed explanation of the parameters of this function. As you can see, the mineigThreshold is a very useful parameter as it helps to remove bad points and gives a performance boost.

The function returns an output vector of 2D points.

Parameters

prevImg	first 8-bit input image or pyramid constructed by buildOpticalFlowPyramid.
nextImg	second input image or pyramid of the same size and the same type as prevImg.
prevPts	vector of 2D points for which the flow needs to be found; point coordinates must be single-precision floating-point numbers.
nextPts	output vector of 2D points (with single-precision floating-point coordinates) containing the calculated new positions of input features in the second image; when OPTFLOW_USE_INITIAL_FLOW flag is passed, the vector must have the same size as in the input.
status	output status vector (of unsigned chars); each element of the vector is set to 1 if the flow for the corresponding features has been found, otherwise, it is set to 0.
err	output vector of errors; each element of the vector is set to an error for the corresponding feature, type of the error measure can be set in flags parameter; if the flow wasn't found then the error is not defined (use the status parameter to find such cases).
winSize	size of the search window at each pyramid level.
maxLevel	0-based maximal pyramid level number; if set to 0, pyramids are not used (single level), if set to 1, two levels are used, and so on; if pyramids are passed to input then algorithm will use as many levels as pyramids have but no more than maxLevel.
criteria	parameter, specifying the termination criteria of the iterative search algorithm (after the specified maximum number of iterations criteria.maxCount or when the search window moves by less than criteria.epsilon.
flags	operation flags: • **OPTFLOW_USE_INITIAL_FLOW** uses initial estimations, stored in nextPts; if the flag is not set, then prevPts is copied to nextPts and is considered the initial estimate. • **OPTFLOW_LK_GET_MIN_EIGENVALS** use minimum eigen values as an error measure (see minEigThreshold description); if the flag is not set, then L1 distance between patches around the original and a moved point, divided by number of pixels in a window, is used as a error measure.
minEigThreshold	the algorithm calculates the minimum eigen value of a 2x2 normal matrix of optical flow equations (this matrix is called a spatial gradient matrix in [14]), divided by number of pixels in a window; if this value is less than minEigThreshold, then a corresponding feature is filtered out and its flow is not processed, so it allows to remove bad points and get a performance boost.

Fig. 2.2: Image describing the individual arguments to the method. Image source: docs.opencv.org

2.3: The Iterative step:

For the function cv2.calcOpticalFlowPyrLK(), we pass the previous frame, previous points and the next frame. This returns the next points along with some status numbers. It has a value of 1 if next point is found, else 0. Thus, we iteratively pass these points as previous points in the next step of the iteration.

2.4: Setting the boundary for object detection

One of the main challenges while implementing optical flow was to restrict the optical flow detection to only a small part of the frame. To do so, we decided to restrict detection of motion for only a specific object using HSV (Hue, Saturation, Value) values for green. As the color light green stands out among other colors, it simplified matters further for us. This step was extremely crucial in our project and hence contributed a great deal towards the betterment of our project.

In our program, for each video frame, we had to convert it from **BGR to HSV color-space**. We then used a threshold value for the HSV image to include just the green color i.e the green color's HSV values. This resulted in extraction of the green object from the image, thus available for further processing. Thus, in the following figure, we compare the output of our previous code that detected all moving points in the frame, to the output of our new code that detects just one object in the frame.

The **trackpoints** shown in the image below returns the number of points returned by the method while detecting a frame in motion. Initially the trackpoints returned a value between 300-500. Which is a huge value to deal with. Using optimization, the trackpoints reduced to 1. This is because we focused object detection to only a specific color.

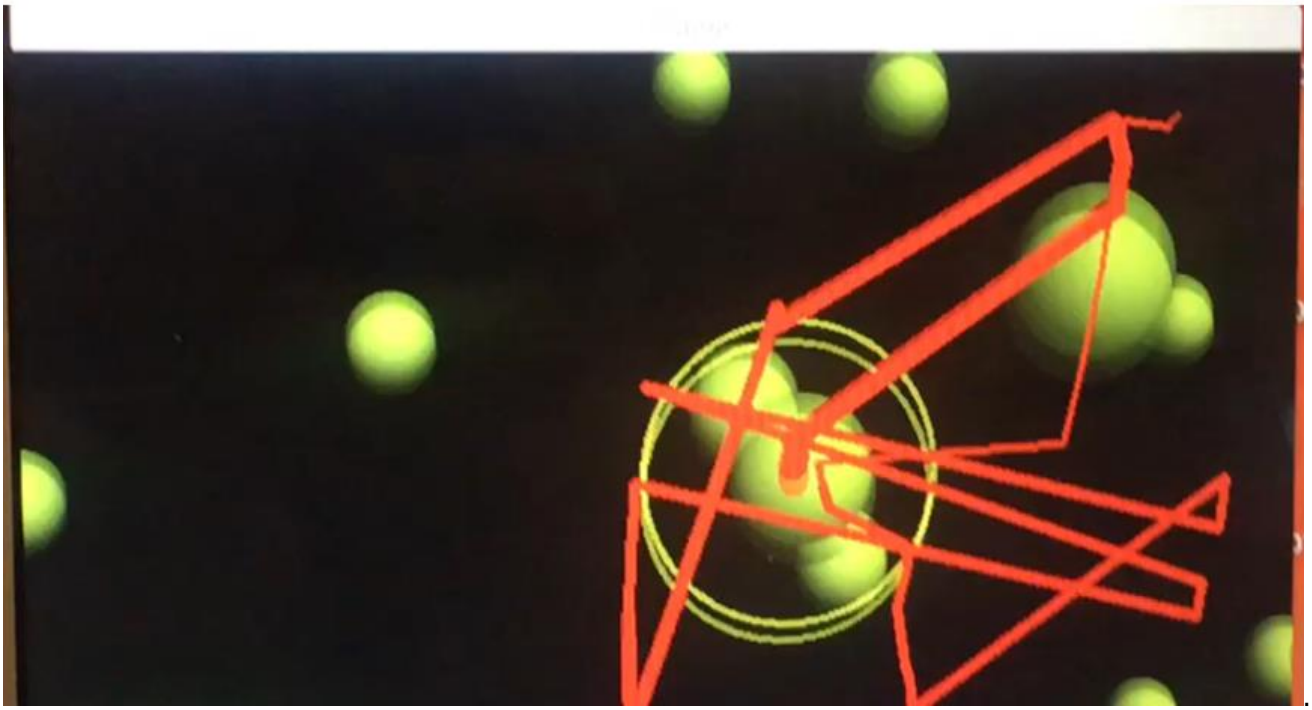


Fig 2.3: Image showing the output of program when trying to detect the motion of green balls in a video frame.

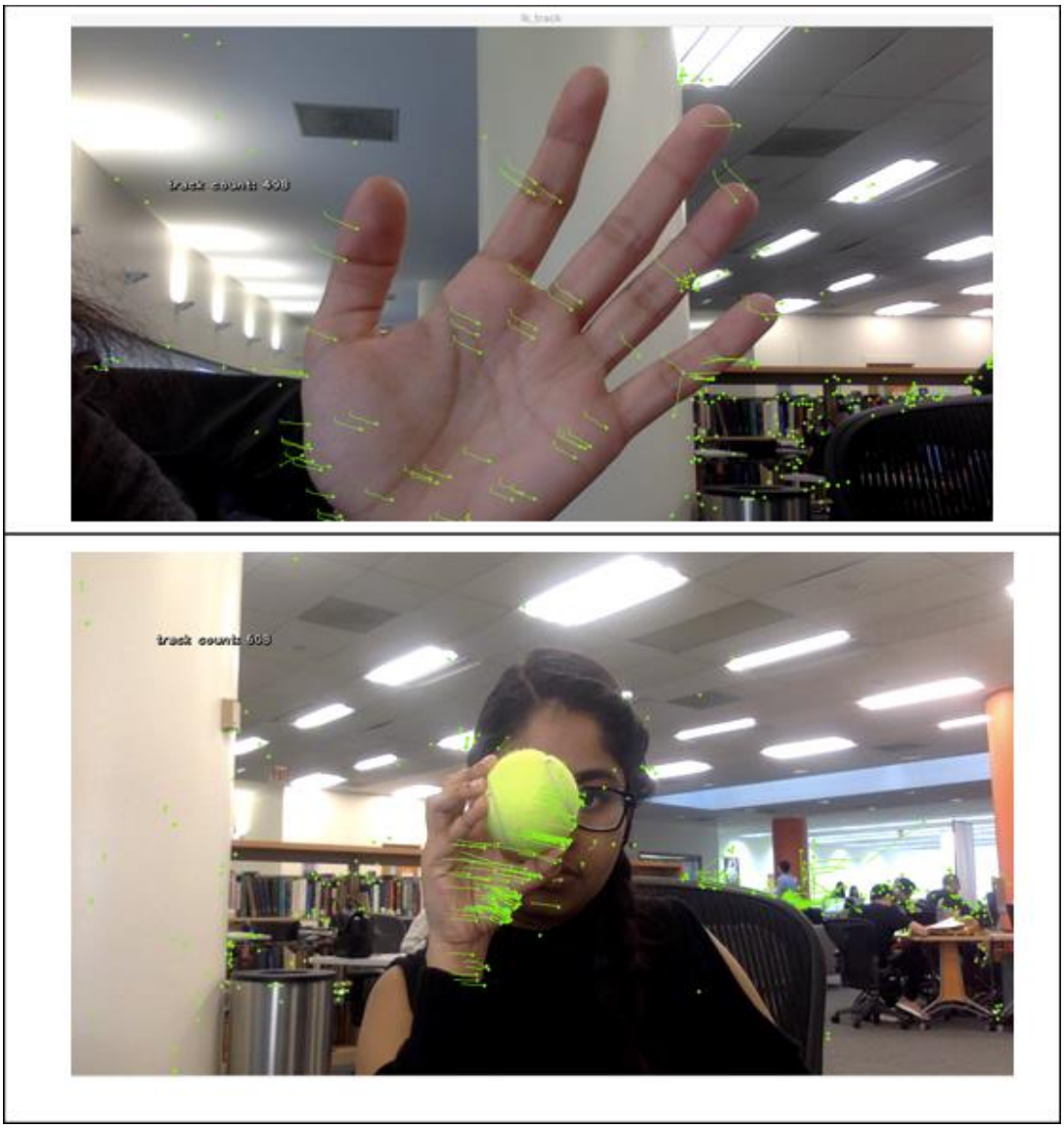
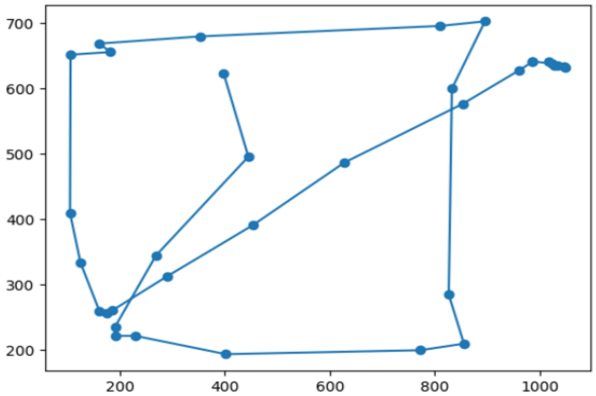


Fig 2.4: Image showing the output of the previous code detecting almost all the points in motion.



Graph generated to detect the direction of movement of hand while playing the game

Fig 2.5: Image showing the graph which depicts the contour of the motion of the ball.

2.5: Determining contours around the detected object

We use a very interesting method cv2.findContours to find he contour of the object and draw its boundary around it, which is our case was a circle. This method takes in a parameter cv2.RETR_TREE to computer the hierarchy between the contours. The method gives us the list of contours it has found.

2.6: Calculating the object centroid

We used the method cv2.moments(), which helps to determine the center of the object, which will be required to track the movement to navigate the game.This step too was one of the crucial steps in our project because without detecting the centroid, we wouldn't have been able to detect in which direction the object is moving. It also helped us pinpoint the location of the object in the frame which was convenient.

The following figure shows us the output till step 6:

Fig. 2.6: Image showing the centroid of the green ball and its boundary detected accurately.

2.7: Determine the direction of movement

To determine whether the object is moving – 'UP', 'DOWN', 'LEFT' or 'RIGHT', we used the co-ordinates of the centroid of the object. The coordinates were very helping in determining the direction of motion of the object. We used a simple technique of subtracting the previous coordinates from the current coordinates in order to determine if the change in motion is greater along the X axis or the Y axis. Thus, we determined the direction of movement of the object.

```
if (xdiff>20 and math.fabs(xdiff)>math.fabs(ydiff)):
    print "LEFT"
    pyautogui.press('left')
elif(xdiff<-20 and math.fabs(xdiff)>math.fabs(ydiff)):
    print "RIGHT"
    pyautogui.press('right')
elif(ydiff>20 and math.fabs(xdiff)<math.fabs(ydiff)):
    print "DOWN"
    pyautogui.press('down')
elif(ydiff<-20 and math.fabs(xdiff)<math.fabs(ydiff)):
    print "UP"
    pyautogui.press('up')
```

Fig. 2.7: Image showing the code snippet which tracks the direction of motion of the object.

2.8: Drawing the graph of the direction of movement of the ball

For drawing the graph of the various locations of the ball in the frame, we've used a python module matplotlib. Using the pyplot method and the functions 'scatter' and 'plot' as shown in the figure below.

```
import matplotlib.pyplot as p
import re
markers=['v']
f=open("textfile.txt",'r')
lines=f.read()
g=re.findall('\w+',lines)
print g
i=0
x=[]
y=[]
for i in range(len(g)):
    if i%2==0:
        x.append(g[i])
    else:
        y.append(g[i])
p.scatter(x,y)
p.plot(x,y)

p.show()
```

Fig. 2.8 Image showing the python code to plot the graph by taking the co-ordinates of the centroid of the moving detected object.

2.9: Masking the frame so that just the required object is highlighted

We added the masking function in our program which removes all other objects from the current frame being processed except the object detected. To implement this feature, we used the cv2.bitwise_and function. This makes the whole frame black and simultaneously highlights only the object with which we are navigating the game. This improves accuracy and helps in further processing.

The function used: res = cv2.bitwise_and(frame,frame, mask= mask)

Fig. 2.9: Image showing the entire scene has been masked except the presence of the green object which is detected.

2.10 Ultimate step: Navigate the game using the object in our hand

We could finally link program to the game to navigate it using motion detection. We also printed the direction of movement to check the accuracy of the program.

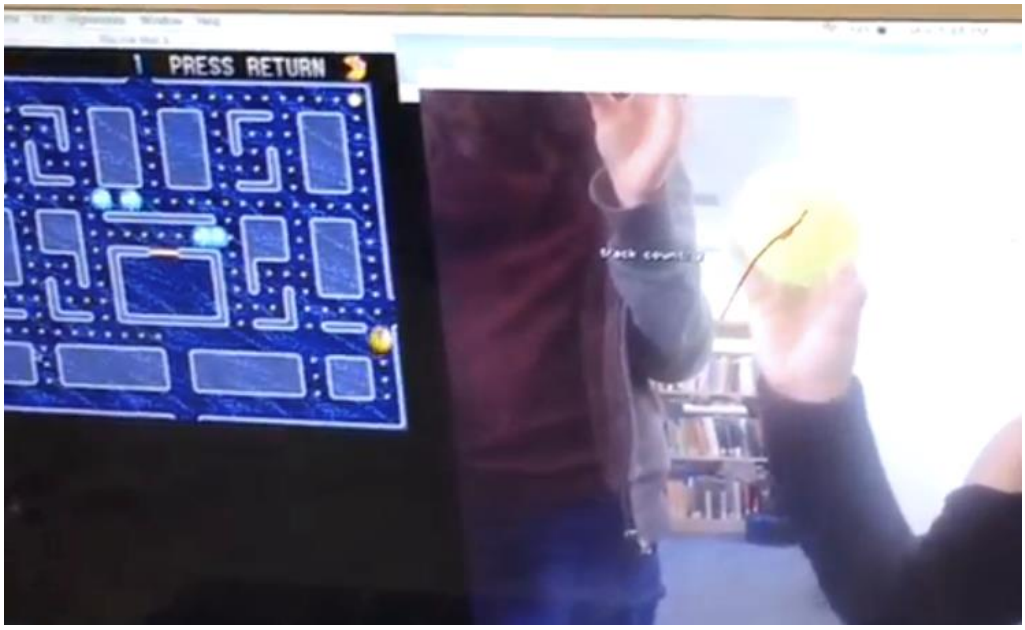


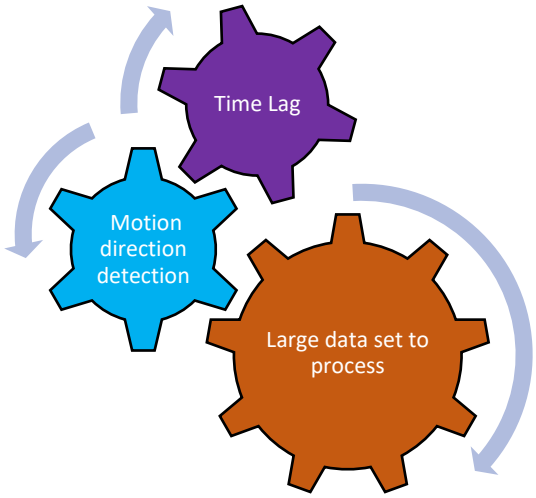
Fig. 2.10: Image showing navigating through the game using the ball detected by the algorithm.

With these steps, we completed the project formulation and implementation.

CHALLENGES FACED WHILE IMPLEMENTING THE PROJECT.

3. The challenges that we stumbled upon while implementing our program is as follows:

1. **Time Lag-** There was a slight delay between the movement of the ball and movement of the character in the game.
2. **Detection of slight movement of the object**- For minute change in movement (even for just 1 pixel), the direction of movement was changing. However, we fixed this later by using threshold values for the number of pixels covered in motion.
3. **Large Data Set to Process-** The initial version of the program was keeping tracks of all the previous centroids earlier, which we later narrowed down to 2, that's the previous and current.



4.Critical Analysis of the Experiment

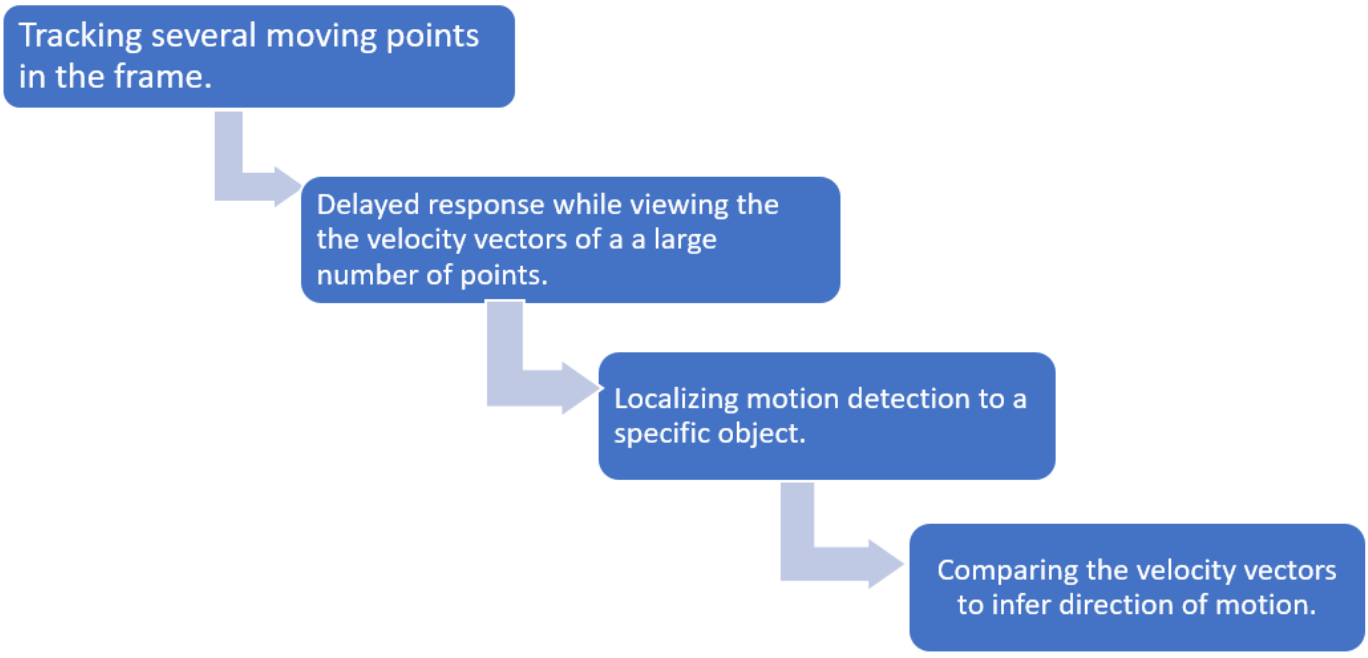


Fig. 4.1: Steps taken to implement the goal of the project

4.1 Tracking the Motion of Several Points in the Frame

There is an inbuilt method in OpenCV which implements the Lucas Kanade Algorithm for Optical Flow which we have implemented in our program. The method with its syntax is given below.

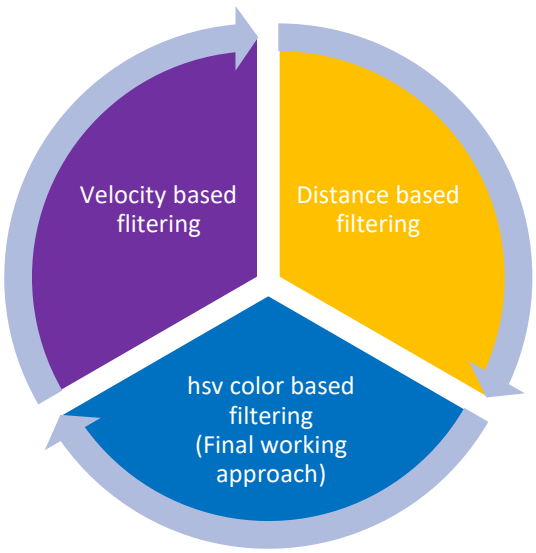
cv2.calcOpticalFlowPyrLK(prevImg, nextImg, prevPts[, nextPts[, status[, err[, winSize[, maxLevel[, criteria[, flags[, minEigThreshold]]]]]]]]) → nextPts, status, err¶

This method returns a set of all the points which are in motion in the current frame. It computes the motion of a sparse set of corner-like points, and tracking behaves better for a dense optical flow result. The method cv2.goodFeaturesToTrack() finds the **N strongest corners** in the image by Shi-Tomasi method (or Harris Corner Detection, if specified). The image supplied to this method should be a grayscale image. When this method was applied to our program, it returned the corners of several objects in the image frame. All the corners below the quality level were rejected. The remaining corners were sorted based on quality in the descending order. The function then took the first strongest corner, threw away all the nearby corners in the range of minimum distance and returned the N strongest corners. This number N was very large for each individual frame taken into consideration. This made tracking the velocity vectors difficult.

4.2 Delayed response while viewing the velocity vectors of many points

To store the co-ordinates of the moving points in frame, we had initially used a dynamic array- "trackingPoints". While running the program, when we print the size of the dynamic array, the real-time values range from 400-500 points. This shows the substantial number of N values returned by the method cv2.goodFeaturesToTrack(). To maintain a data structure to store all the points and compare the previous frame and current frame co-ordinate values of each corresponding point to infer the direction of movement made the **program slow**, **increased the time lag in movement**, and made it **unnecessarily complicated**. This step had to be optimized to properly attain the goal of our program.

4.3 Localizing Motion Detection to only a specific object.



To achieve motion detection for only a specific object, we first needed to understand the properties of different objects in the image frame which could be tracked.

- **VELOCITY - Velocity** is a crucial factor in optical flow which could eliminate a lot of points detected unnecessarily in the frame. However, if a ball is in our hands, the motion of the ball will be equal to the motion of our hands, hence the algorithm still tracks all the moving points on the ball and on our hand. If any moving person comes into the frame where the webcamera is, and his velocity exceeds the threshold velocity, the vectors on his corners shall also be tracked. Hence using velocity as a method to filter the large number of points in frame did not properly solve the problem.

- **DISTANCE FROM THE CAMERA-** We tried to filter points by removing those which are at a farther distance from the webcamera. This would only include points closer to the webcamera in the image frame and prevent any noisy interference due to external agents entering the frame. Thus, even if anyone else enters the frame, or anything flies into the frame, it will not be tracked. But, while playing the game, the user's hand may drift far away from the camera. Moreover, if we are restricting the motion only with distance, it makes it very difficult to navigate the game. Hence, this also did not resolve the problem of reducing the number of velocity vectors tracked per frame.

- **HSV COLOR VALUES**- We finally took the approach of filtering based on the hsv values. The hsv values are the **hue, saturation and value** for the color of objects. Each image frame is in RBG. An RGB color value is specified with: rgb(red, green, blue). Each parameter (red, green, and blue) defines the intensity of the color and can be an integer between 0 and 255 or a percentage value (from 0% to 100%). We could convert each RGB frame to its equivalent HSV frame. The Hue then is a degree on the color wheel (from 0 to 360) - 0 (or 360) is red, 120 is green, 240 is blue. Saturation is a percentage value; 0% means a shade of gray and 100% is the full color. Lightness is also a percentage; 0% is black, 100% is white. Thus, using the values for the color of a specific object, we could restrict the motion detection to only a specific colored object which we want to track.

- The tracking based on color immediately filtered almost 300 points and speeded the program. Using the upper and lower ranges for the colors correctly and applying them to the method **"cv2.inRange(hsv, hsvLower, hsvUpper)"** helped in optimizing the motion detection algorithm. The only disadvantage is, if there is any interference in the frame due to the presence of more than one similar colored object in the frame, all those objects will be tracked. This condition is however very unlikely and not quite often expected to happen.

5. FUTURE GOALS FOR THE PROJECT

- We would like to extend our program in detecting the motion of hands.

- We would also like to resolve the time lag between motion and detection of movement trying to improvise the Horn and Schnuk Algorithm.

- We also want to design our game in python and test our project to see the output.

6. ROLE OF INDIVIDUAL GROUP MEMBERS

- **Titash Mandal** - Worked with team-mate Saniya Alekar to implement the ball tracking program in python. Designed the framework for the output of the program and optimized it to give the best response. Worked on the PowerPoint presentation and writing and editing the final project report.

- **Saniya Alekar**- Worked with team-mate Titash Mandal to implement the ball tracking program in python. Designed the framework for the output of the program and optimized it to give the best response. Worked on writing the final project report.

- **Sen Huo**- Worked on implementing a game in python and the writing the final project report.

- **Hanzhao Wang**- Worked on implementing a game in python and writing the final project report.

CONCLUSION

- Through this project we studied and implemented an interesting computer vision technique to reconstruct the motion between the two images acquired through a live video sequence. The reconstruction of the Optical Flow allows us to estimate the displacement of moving objects of a scene over time. Looking at the direction of the velocity vectors helped us to calculate the motion direction of the moving object. One of the fundamental property used by the previous method was to consider that, the inter frame motion is small.

- There were several factors that were taken into account when considering optical flow concepts.

- **Translation and rotation:** Optical flow results from both rotation and translation. Translational optical flow tells you about other objects in the environment, but rotational optical flow can dominate. If you are undergoing rapid rotations, then rotational optical flow can wash out measurements of translational optical flow. You need to find a way to remove the rotational component, ideally using a gyro to provide rotational methods.

- **Low texture contrast:** A common myth is that optical flow techniques do not work when there is no texture. The reality is that except for perfectly clean and smooth surfaces (which are very rare), there is almost always texture that can be tracked. You just need to have a vision system that is sensitive enough to pick these out. In the past, Centeye has used advanced analog processing circuits on our vision chips to enhance texture contrast, to the point that we were able to measure optical flow caused by the motion of a blank, white sheet of paper, and even control the height of an aircraft when flying over snow on a cloudy day.

- **Mechanical jitter:** When a vision system is mounted on a moving platform, mechanical vibrations can affect the measurement of optical flow. Fortunately we have developed image processing techniques that exploit such mechanical jitter, to allow obstacle detection and even the detection and tracking of tiny targets from a camera in motion.

- **Light levels:** How much light is there? A common myth is that you need substantial light to support optical flow measurement. This myth is prolonged by the fact that optical mouse sensors require either sunlight or an LED mounted millimeters away from the target system, and many other optical flow solutions using standard cameras require tens or hundreds of light rays to work. However, are nocturnal insects that can navigate using optical flow in light levels that are perceived as dark by humans. We have demonstrated optical flow based hover in light levels as low as several lux, and are currently implementing vision systems for use in low light environments using techniques inspired by these insects.

- **Motion transparency:** Optical flow is commonly described as a vector field with a single vector or optical flow measurement for each location. (Abraham Maslow once said, “he who is good with a hammer thinks everything is a nail”, so it is inevitable that when the first drawings of optical flow were shown to computer scientists, they immediately thought of a vector field.) The reality is that one location can have multiple optical flows. Pay attention next time you walk past a chain link fence, or drive past a forest in the winter. There will be multiple optical flows happening in each location, and if we can measure these, we will have more knowledge of the environment.

REFERENCES

[1] Robert Collins CSE486, Penn State Lecture, http://www.cse.psu.edu/~rtc12/CSE486/lecture30.pdf

[2] Lucas-Kanade in a Nutshell Prof. Dr. Raul Rojas, http://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/tutorials/Lucas-Kanade2.pdf

[3] Robustness of the Tuning of Fly Visual Interneurons to Rotatory Optic Flow, http://jn.physiology.org/content/90/3/1626

[4] Lucas-Kanade method, Wikipedia, https://en.wikipedia.org/wiki/Lucas%E2%80%93Kanade_method

[5] OpenCV 3.2.0-dev, Open Source Computer Vision, http://docs.opencv.org/trunk/d7/d8b/tutorial_py_lucas_kanade.html

[6] David Stavens Stanford Artificial Intelligence Lab, http://robots.stanford.edu/cs223b05/notes/CS%20223-B%20T1%20stavens_opencv_optical_flow.pdf

[7] Car Tracking Using Optical Flow OpenCV, Youtube Video - https://www.youtube.com/watch?v=MOaKnCSejXQ

[8] Implementing Lucas Kanade's Optical Flow by Mikel Rodriguez, http://www.cs.ucf.edu/~mikel/Research/Optical_Flow.htm

[9] Coarse-to-fine Optical Flow (Lucas & Kanade), http://eric-yuan.me/coarse-to-fine-optical-flow/

[10] Lucas Kanade python numpy implementation uses enormous amount of memory, http://stackoverflow.com/questions/14321092/lucas-kanade-python-numpy-implementation-uses-enormous-amount-of-memory