CSE 152 Homework 4 Video Understanding

In this homework, you will use the tools learned in class to solve object tracking and object discovery problems.

The due for this homework is scheduled one day after the final exam, which is `Dec. 10th, 11:59 pm`

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage.filters import gaussian_filter, convolve
import scipy
import warnings
from skimage.io import *
warnings.filterwarnings('ignore')
import cv2
from skimage import filters
```

Question 1: Iterative KLT tracker (70 pts)

In this question, you will track a specific object in a given video (test.avi) by implementing an iterative KLT tracker. The KLT tracker works on two frames at a time, and estimates the deformation between two image frames under the assumption that the intensity of the objects has not changed significantly between the two frames. In this homework, we assume the motion is translation only (the matrix P we learned from class is a translation matrix here).

Starting with a rectangle R_t on frame I_t, the KLT tracker aims to move it by an offset (u, v) to obtain another rectangle R_{t+1} on frame I_{t+1}, so that the pixel squared difference in the two rectangles is minimized:

$$min_{u,v} J(u, v) = \Sigma_{x,y \in R_t} (I(x + u, y + v) - I(x, y))^2$$

Question 1.1: Preliminary [10 pts]

Starting with an initial guess of (u, v) (usually $(0, 0)$), we can compute the optimal (u^*, v^*) iteratively. In each iteration, the objective function is locally linearized by first-order Taylor expansion and optimized by solving a linear system that has the form $A\delta_p = b$, where $\delta_p = (\delta_u, \delta_v)^\top$ is the template offset. Please answer the following questions:

1. What is $A^\top A$? Using image gradient to derive it.
2. What conditions must $A^\top A$ meet so that the template offset can be calculated reliably? Explain why.

Your answer here:
1.$A = \nabla I$ where $\nabla I = (I_x, I_y)$ is the image gradient.
Let n be the number of pixels in R_t, then $A^T A = \nabla I^T \nabla I$.

$$A^T A = \begin{bmatrix} I_x(p_1) & I_x(p_2) \dots & I_x(p_n) \\ I_y(p_1) & I_y(p_2) \dots & I_y(p_n) \end{bmatrix} \begin{bmatrix} I_x(p_1) & I_y(p_1) \\ I_x(p_2) & I_y(p_2) \\ \vdots & \vdots \\ I_x(p_n) & I_y(p_n) \end{bmatrix} = \Sigma_{x,y \in R_t} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

2. $A^T A$ should be be nonsingular so that we can reverse it successfully. Also, the eigenvalues of $A^T A$ should be both large and have similar magnitude, which represent corners. Since corners are more invariant to transformation than flat regions and edges, they are more reliable to track.

Question 1.2 Iterative KLT Tracker Implementation (50 pts)

Complete the `Iter_KLT` method below, which computes the optimal local motion from frame I_t to frame I_{t+1} that minimizes the objective in Question 1.

1. Note that moving the template rectangle by u and v will lead to fractional coordinates of the pixels. However, you need to extract information within the rectangle every time step, such as the image intensity, image gradient. To deal with this issue, you can convert the coordinates to integers, or perform some interpolations for floating numbers.
2. You will also need to iterate the estimation until the change in (u, v) is below a threshold, or reach a maximal iteration number. We set the threshold to be 0.005, and max iterations to be 1500.
3. You can adjust every parameter you want to make the tracking more accurate.

The rectangle in the first frame is $[x_1, y_1, x_2, y_2] = [318, 208, 418, 268]$. In other words, the rectangle starts from (318, 208) (row 208 and column 318 in the image) and ends at (418, 268). You need to understand the image coordinations correctly.

If you complete this script correctly, it will play a video with a rectangle tracking the car. You can refer `fig1.png` , `fig2.png` , `fig3.png` for reference.

In [2]:

```
def Iter_KLT(I_t, I_t_1, rect, max_iter=1500, threshold=0.005):
    '''
    Input:
        I_t: image frame at time t
        I_t_1: image frame at time t+1
        rect: tracking rectangle at time t
        max_iter: maximum iteration steps for iterative KLT tracker
        threshold: if delta_p's norm is smaller than the threshold, then the ite

    Return:
        rect_new: tracking rectangle at time t+1
        You need to compute "delta_p" as the translation for the rectangle from
    '''
    u = 0
    v = 0
    img_h, img_w = I_t_1.shape[0], I_t_1.shape[1]
    delta_p_length = 1000
    rect_new = rect.copy()

    # Extract image gradient at time t+1
    Ix = cv2.Sobel(I_t_1,cv2.CV_64F,1,0,ksize=5)
    Iy = cv2.Sobel(I_t_1,cv2.CV_64F,0,1,ksize=5)
    iters = max_iter
    delta_p = [0, 0]
    iters = 0

    # Loop until "delta_p" is sufficiently small, or iteration number reaches ma
    while delta_p_length > threshold and iters < max_iter:
        if rect_new[0] < 0 or rect_new[1] < 0 or rect_new[2] >= img_w or rect_ne
            print('Tracking rectangle out of boundary!')
            break

        ### You should calculate delta_p in the following codes
        ### YOUR CODE HERE
        Ix_t  = Ix[rect[1]+int(v):rect[3]+int(v),rect[0]+int(u):rect[2]+int(u)].f
        Iy_t  = Iy[rect[1]+int(v):rect[3]+int(v),rect[0]+int(u):rect[2]+int(u)].f
        delta_I = (I_t[rect[1]:rect[3],rect[0]:rect[2]]
                       - I_t_1[rect[1]+int(v):rect[3]+int(v),rect[0]+int(u):rect[2]+

        A = np.array([Ix_t, Iy_t])
        H = np.array([Ix_t@Ix_t, Ix_t@Iy_t, Ix_t@Iy_t, Iy_t@Iy_t ]).reshape(2,2
        delta_p = np.linalg.inv(H) @ A @ delta_I

        ### YOUR CODE ENDS

        u = u + delta_p[0]
        v = v + delta_p[1]


        delta_p_length = np.linalg.norm(delta_p)
        rect_new = np.array([rect[0]+u, rect[1]+v, rect[2]+u, rect[3]+v])
        iters += 1

    return np.round(rect_new).astype(int)
```

In [3]:

```
cap = cv2.VideoCapture("test.avi")

# Initialized tracking rectangle of the car
rect = np.array([318, 208, 418, 268])

ret, old_frame = cap.read()
old_frame_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)/255.
# A mask to draw the tracking rectangle for initialization
mask = np.zeros_like(old_frame)
mask = cv2.rectangle(mask, (rect[0], rect[1]), (rect[2], rect[3]), color=(0, 0,
img = cv2.add(old_frame, mask)
cv2.imshow('frame',img)
cv2.waitKey(30) & 0xff

while(1):
    # read video and turn it to grayscale using opencv
    ret,frame = cap.read()

    if ret is True:
        cur_frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)/255.
    else:
        break

    rect_new = Iter_KLT(old_frame_gray, cur_frame_gray, rect)

    # Clear the mask to draw the tracking rectangle every step
    mask = np.zeros_like(frame)
    mask = cv2.rectangle(mask, (rect_new[0], rect_new[1]), (rect_new[2], rect_ne
    img = cv2.add(frame, mask)
    cv2.imshow('frame',img)
    k = cv2.waitKey(30) & 0xff
    if k == 27:
        break

    # update old frame and tracking rectangle
    old_frame_gray = cur_frame_gray.copy()
    rect = rect_new.copy()


cv2.destroyWindow('frame')
cv2.waitKey(1)
cap.release()
```

Question 1.3 Visualization (10 pts)

Plot your tracking result (image + rectangle) at frame 5, frame 20, frame 50 and frame 90 below.

In [4]:

```
cap = cv2.VideoCapture("test.avi")

# Initialized tracking rectangle of the car
rect = np.array([318, 208, 418, 268])
ret, old_frame = cap.read()
old_frame_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)/255.

while(1):
    # read video and turn it to grayscale using opencv
    ret,frame = cap.read()

    if ret is True:
        cur_frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)/255.
    else:
        break

    rect_new = Iter_KLT(old_frame_gray, cur_frame_gray, rect)

    # Clear the mask to draw the tracking rectangle every step
    mask = np.zeros_like(frame)
    mask = cv2.rectangle(mask, (rect_new[0], rect_new[1]), (rect_new[2], rect_ne
    img = cv2.add(frame, mask)

    frame_idx = cap.get(cv2.CAP_PROP_POS_FRAMES)

    if frame_idx in [5,20,50,90]:
        print("Frame", int(frame_idx))
        plt.imshow(img)
        plt.show()


    k = cv2.waitKey(30) & 0xff
    if k == 27:
        break

    # update old frame and tracking rectangle
    old_frame_gray = cur_frame_gray.copy()
    rect = rect_new.copy()

cv2.waitKey(1)
cap.release()
```

Frame 5

Frame 20

Frame 50

Frame 90

Question 2 Moving Object Discovery (50 pts, contains 20 pts extra credit)

In this problem, you are provided with a video game, which contains some moving objects as well as fixed objects, which is `video_game.mp4`

Design an algorithm that will find the moving objects in the video (30 pts).

We do not have any specific requirements. Please do you best to achieve this open-ended task. However, following are some hints:

1. Use optical flow method to compute the flow of the moving objects. You can use APIs provided in OpenCV, like [https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_video/py_lucas_kanade/py_lucas_kanade.html) (https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_video/py_lucas_kanade/py_lucas_kanade.html)](https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_video/py_lucas_kanade/py_lucas_kanade.html). To use the sample code in this link, you need to figure out a method to select the points of interest. Simply using the corner extraction will introduce some corners remain fixed for the video.
2. You can represent a moving object in your own way. For example, you can use rectangles like in problem 1 / a binary mask / some keypoints / trajetories. You should select at least 5 representative frames, and plot the moving object in the frame. Binary masks and rectangles are usually more challenging, we will give more credits on that.
3. Give detailed analysis based on your results. (Failure cases? Efficiency?) Propose a potential improvement.

Requirements for extra credits (+20 pts):

1. Propose novel solutions to handle the corner cases.
2. Provide visualization of your improvements and give analysis on that.

We understand that the approach description is vague. But we expect to witness your engineering skills to solve this problem :). Please treat it as a small "final project".

In [5]:

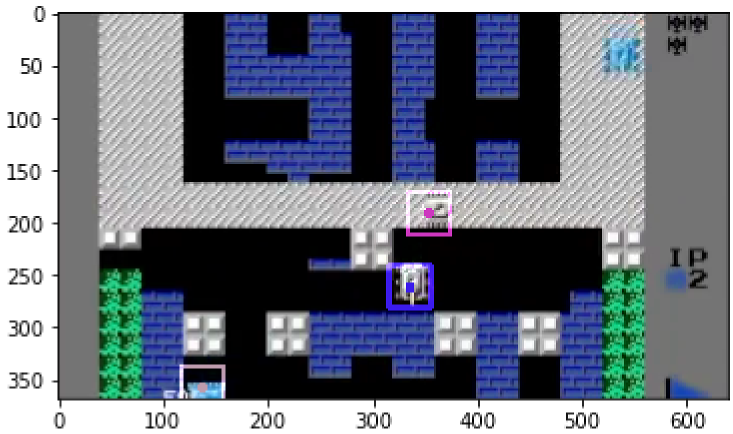
```
import numpy as np
import cv2
from sklearn.neighbors import NearestNeighbors
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
```

In [6]:

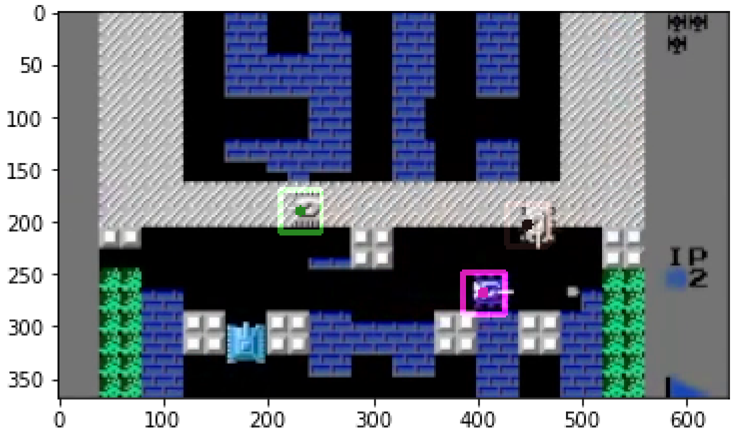
```
import numpy as np
import cv2

cap = cv2.VideoCapture('video_game.mp4')

# params for ShiTomasi corner detection
feature_params = dict( maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7,
                       blockSize = 7 )

# Parameters for lucas kanade optical flow
lk_params = dict( winSize  = (15,15),
                  maxLevel = 2,
                  criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,

# Create some random colors
color = np.random.randint(0,255,(400,3))

# Take first frame and find corners in it
ret, old_frame = cap.read()
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur(old_gray,(5,5),0)

p0 = cv2.goodFeaturesToTrack(blur, mask = None, **feature_params)

img_h,img_w = old_gray.shape[0], old_gray.shape[1]

offset = 20
pts = []
for p in p0:
    p = p[0]
    xmin = p[0] - offset
    ymin = p[1] - offset
    xmax = p[0] + offset
    ymax = p[1] + offset
    if ( xmin > 0) and (ymin > 0):
        pts.append([[xmin,ymin]])
        pts.append([[p[0],ymin]])
        pts.append([[xmin,p[1]]])
    if ( xmax < img_h) and (ymax < img_w):
        pts.append([[xmax,ymax]])
        pts.append([[xmax,p[1]]])
        pts.append([[p[0],ymax]])
p0 = np.concatenate((p0,np.float32(pts)))


# Create a mask image for drawing purposes
mask = np.zeros_like(old_frame)

while(1):
    ret,frame = cap.read()
    if ret is True:
        frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    else:
        break

    # calculate optical flow
    p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None, **lk
```

```

    # Select good points
    good_new = p1[st==1]
    good_old = p0[st==1]

    gnew = []
    gold = []

    for i,(new,old) in enumerate(zip(good_new,good_old)):
        a,b = new.ravel()
        c,d = old.ravel()
        if (a-c)**2 + (b-d)**2 < 0.1:
            continue

        gnew.append(new)
        gold.append(old)

    if len(gnew) >16:
        n_clusters = 3
    elif len(gnew) > 8:
        n_clusters = 2
    else: n_clusters = 1

    if len(gnew) == 0:
        continue
    X = np.array(gnew)
    kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(X)

    # draw the tracks
    mask = np.zeros_like(old_frame)
    for i,(new,old) in enumerate(zip(gnew,gold)):
        a,b = new.ravel()
        c,d = old.ravel()

        label = kmeans.predict([new])
        t = np.float32(np.round(kmeans.cluster_centers_[label]))
        if (np.linalg.norm(t-new) < 100):
            o1,o2 = t.ravel()

            mask = cv2.rectangle(mask, (int(o1-20),int(o2-20)),(int(o1+20),int(
            frame = cv2.circle(frame,(o1,o2),5,color[i].tolist(),-1)
    img = cv2.add(frame,mask)

    frame_idx = cap.get(cv2.CAP_PROP_POS_FRAMES)

    if frame_idx in [10,50,66,90,135]:
        print("Frame", int(frame_idx))
        plt.imshow(img)
        plt.show()
    cv2.imshow("frame",img)
    k = cv2.waitKey(30) & 0xff
    if k == 27:
        break

    # Now update the previous frame and previous points
    old_gray = frame_gray.copy()
    p0 = good_new.reshape(-1,1,2)

cv2.destroyAllWindows()
cap.release()

```

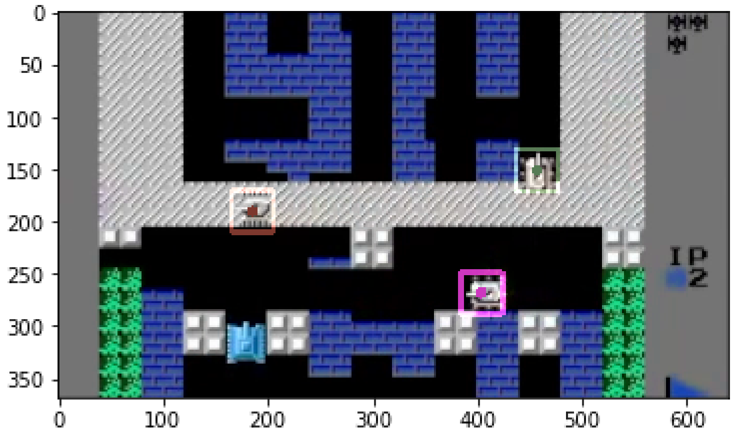
Frame 10



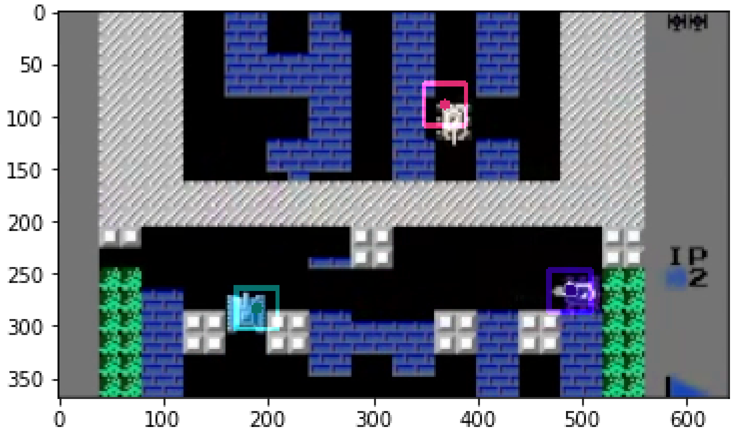
Frame 50



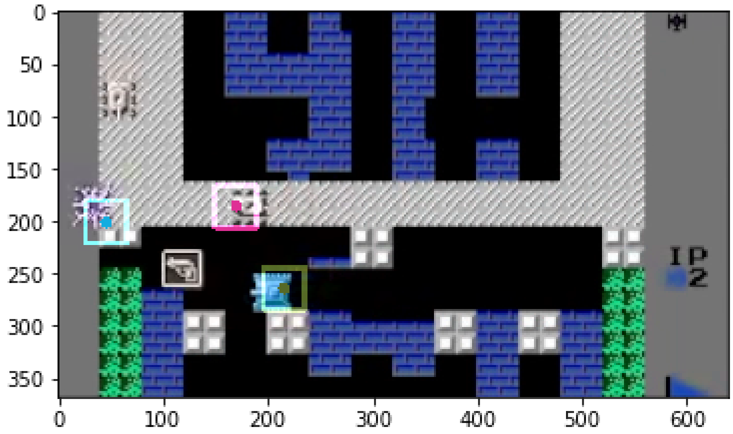
Frame 66



Frame 90



Frame 135



For this part, I first apply gaussian filter to blur the background and denoise. Then I use corner extraction for points of interest, and for each corner point, I add 6 more points which are around the corner point with a small offset in order to get more feature descriptors of a moving object. Then I trying to find the new corresponding points, and ignore those points if they are within 0.1 distance of the old interest points because these corner points remain fixed. Then I use K Mean cluster, to get the centeroid of moving objects and thus represent the moving object with a rectangle.
This method works well at most of time, however, there are some failure cases, for example, those moving objects emitted at the later part of the video are not tracked well. The reason for this would be:1. those objects are grey which are similar to the background so it is hard to detect and track. 2.As they appeared late, there are no old points of interest that describes them, which makes tracking hard. 3. The performance is also not so good when the moving object is going through a narrow aisle, as the surrounding pixels would affect them.
In terms of efficiency, I think it works fine as I already filtered out those fixed corner points. Better feature extraction may help improve the efficiency.
In order to address these problems, I think adding more points of interest which better describes the late appearing objects would help. Also, using background subtraction may help reduce the noise more and make some improvement.

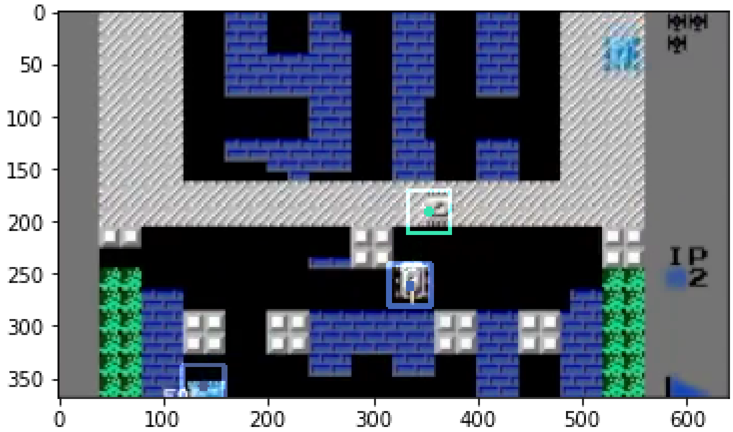
In [7]:

```
import numpy as np
import cv2
import random

cap = cv2.VideoCapture('video_game.mp4')

# params for ShiTomasi corner detection
feature_params = dict( maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7,
                       blockSize = 7 )

# Parameters for lucas kanade optical flow
lk_params = dict( winSize  = (15,15),
                  maxLevel = 2,
                  criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,

# Create some random colors
color = np.random.randint(0,255,(400,3))

# Take first frame and find corners in it
ret, old_frame = cap.read()
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
# fgbg = cv2.createBackgroundSubtractorMOG2()
# blur = fgbg.apply(old_frame)
# blur = cv2.bilateralFilter(img,9,75,75)
blur = cv2.GaussianBlur(old_gray,(5,5),0)

p0 = cv2.goodFeaturesToTrack(blur, mask = None, **feature_params)
img_h,img_w = old_gray.shape[0], old_gray.shape[1]

offset = 20
pts = []
for p in p0:
    p = p[0]
    xmin = p[0] - offset
    ymin = p[1] - offset
    xmax = p[0] + offset
    ymax = p[1] + offset
    if ( xmin > 0) and (ymin > 0):
        pts.append([[xmin,ymin]])
        pts.append([[p[0],ymin]])
        pts.append([[xmin,p[1]]])
    if ( xmax < img_h) and (ymax < img_w):
        pts.append([[xmax,ymax]])
        pts.append([[xmax,p[1]]])
        pts.append([[p[0],ymax]])
p0 = np.concatenate((p0,np.float32(pts)))

rand = []
for i in range(200):
    x = random.randint(0,100)
    y = random.randint(50,550)
    rand += [[[x,y]]]

p0 = np.concatenate((p0,np.float32(rand)))


# Create a mask image for drawing purposes
```

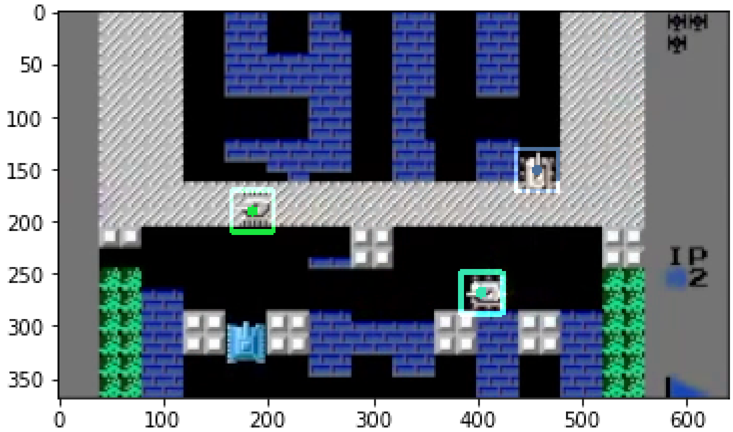
```
mask = np.zeros_like(old_frame)

while(1):
    ret,frame = cap.read()
    if ret is True:
        frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    else:
        break

    # calculate optical flow
    p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None, **lk

    # Select good points
    good_new = p1[st==1]
    good_old = p0[st==1]

    gnew = []
    gold = []

    for i,(new,old) in enumerate(zip(good_new,good_old)):
        a,b = new.ravel()
        c,d = old.ravel()
        if (a-c)**2 + (b-d)**2 < 0.1:
            continue

        gnew.append(new)
        gold.append(old)

    if len(gnew) >16:
        n_clusters = 3
    else: n_clusters = 1

    if len(gnew) == 0:
        continue
    X = np.array(gnew)
    kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(X)

    # draw the tracks
    mask = np.zeros_like(old_frame)
    for i,(new,old) in enumerate(zip(gnew,gold)):
        a,b = new.ravel()
        c,d = old.ravel()

#         frame = cv2.circle(frame,(a,b),5,color[i].tolist(),-1)
        label = kmeans.predict([new])
        t = np.float32(np.round(kmeans.cluster_centers_[label]))
        if (np.linalg.norm(t-new) < 100):
            o1,o2 = t.ravel()
            mask = cv2.rectangle(mask, (int(o1-20),int(o2-20)),(int(o1+20),int(
            frame = cv2.circle(frame,(o1,o2),5,color[i].tolist(),-1)
    img = cv2.add(frame,mask)

    frame_idx = cap.get(cv2.CAP_PROP_POS_FRAMES)

    if frame_idx in[10,66,100,140,160,190,210]:
        print("Frame", int(frame_idx))
        plt.imshow(img)
        plt.show()

    cv2.imshow("frame",img)
    k = cv2.waitKey(30) & 0xff
```

```python
    if k == 27:
        break

    # Now update the previous frame and previous points
    old_gray = frame_gray.copy()
    p0 = good_new.reshape(-1,1,2)

cv2.destroyAllWindows()
cap.release()
```

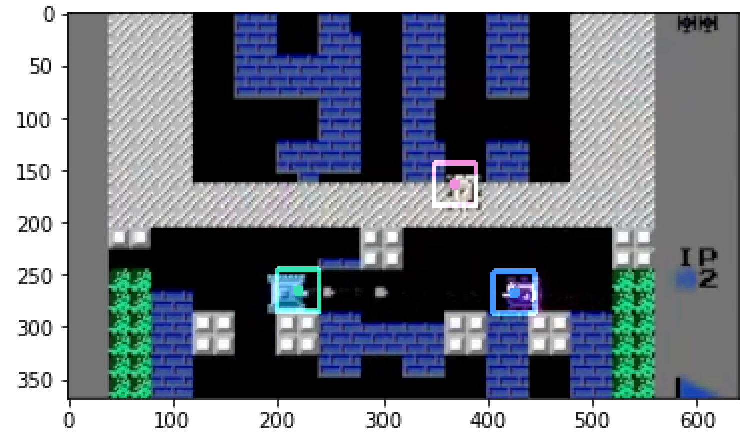
```
Frame 10
```



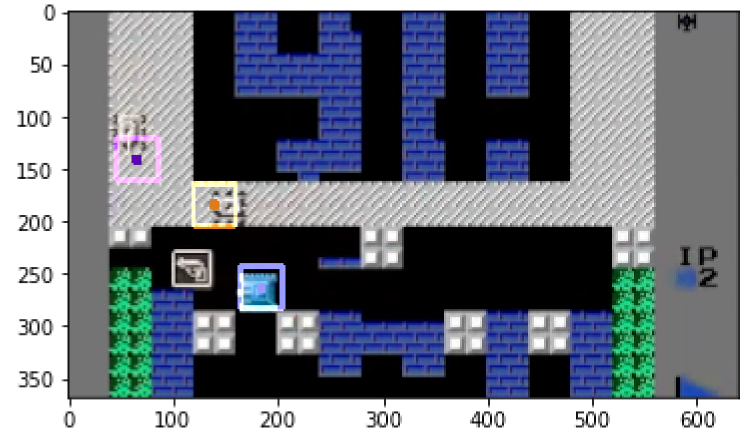

```
Frame 66
```



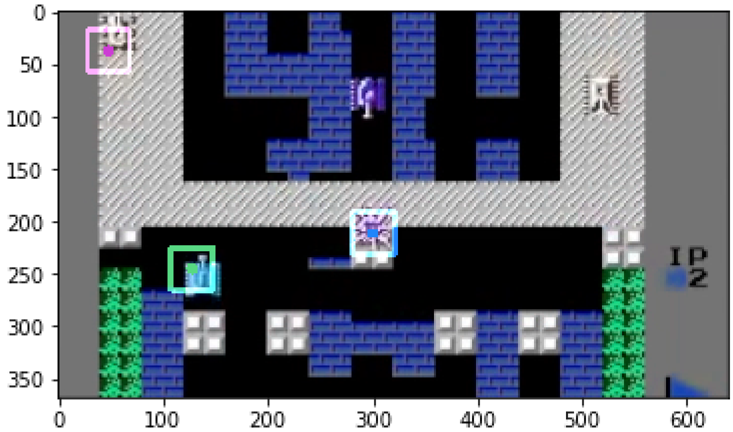

```
Frame 100
```



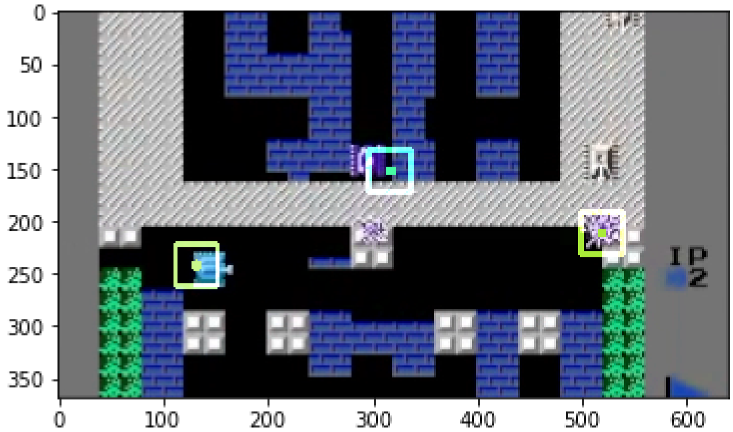
Frame 140



Frame 190



Frame 210



Solution: In order to solve the problem that it is hard to track those later appeared moving objects,I randomly sample many points in the upper part of the whole window, where those objects appear and move. Also, use corner points to describe other already existed objects.
In this way, I successfully track two of the late appearing moving objects. Randomly sampled points doesn;t depend on image feature, so they don't need to be corners which help detect gray objects with a gray background that later appeared.