به نام خدا

دانشگاه صنعتی امیرکبیر

دانشکده مهندسی کامپیوتر

پاسخ تمرین سری اول تصویرپردازی رقمی

استاد:

دکتر رحمتی

دانشجو:

حلیمه رحیمی

شماره دانشجویی:

۹۹۱۳۱۰۴۳

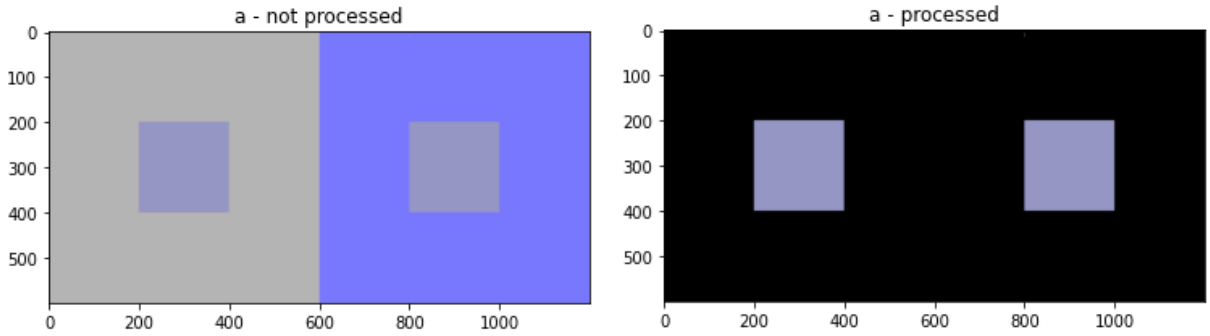
بهار ۱۴۰۰

1. When Our Brain Can't Believe Our Eyes

Each of the given images appear differently from the way they actually are. Your goal is to work on them and produce a new image in which the inaccuracy of our perception is proven.

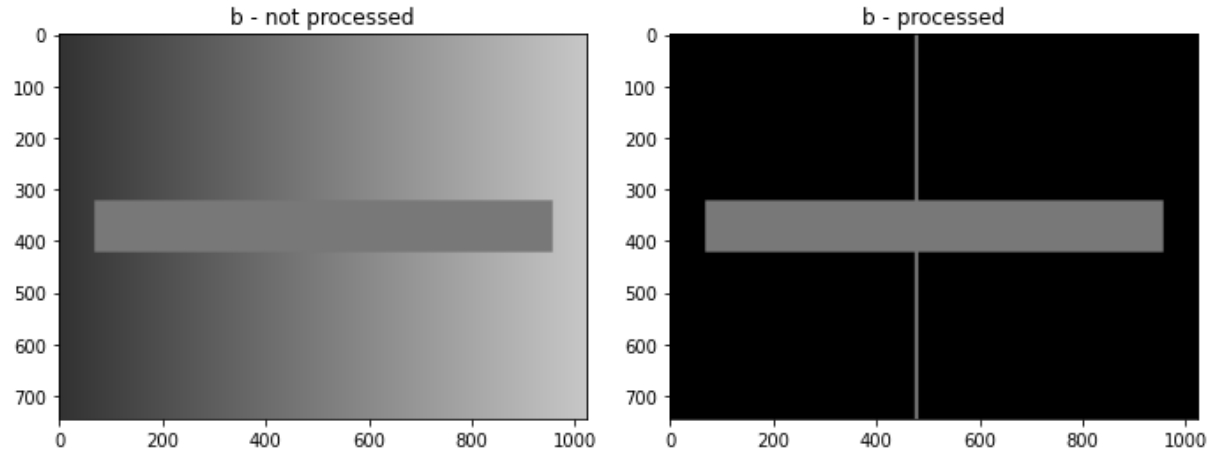
In all parts, at first I load the image and determine the spatial location of a pixel in the specific part of the image that we want to analyze. Then I let all the pixels else than the ones with the same values (RGB color values) as that pixel be equal to zero.

a. Color saturation illusion: The two smaller squares in the larger gray and blue squares in the image appear to be blue or violet (left) and gray (right), while they are, in fact, the same color.



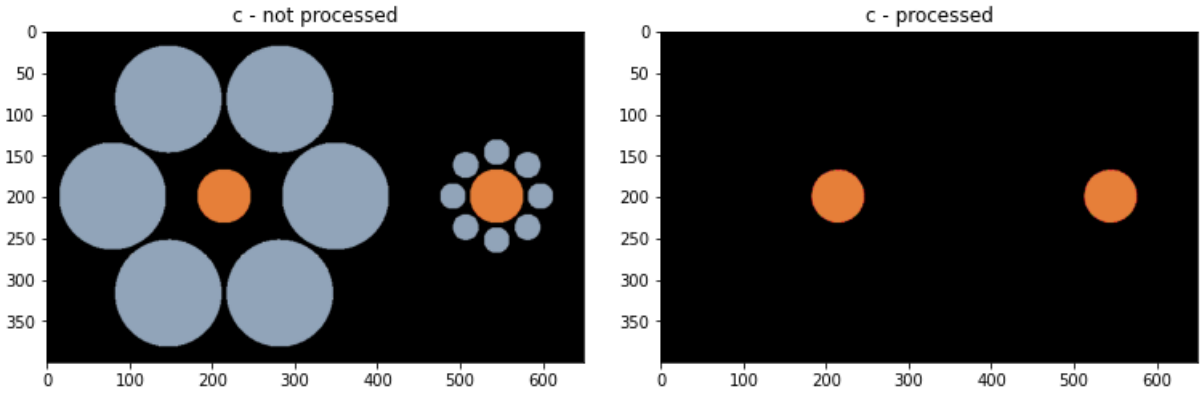
As you can see, both squares are still in the picture. Considering that we only kept the pixels with the same color, the squares have got to be of the same color.
The chosen pixel was the one at (300,300).

b. Gradient optical illusion: The horizontal bar in the middle appears to be brighter in the left and gradually becomes darker in the opposite side, while the truth is that the entire bar is of the same color.



I only kept the pixels with the same color as the pixel at (350,500). You can see all of the triangle in the picture is of one solid color.

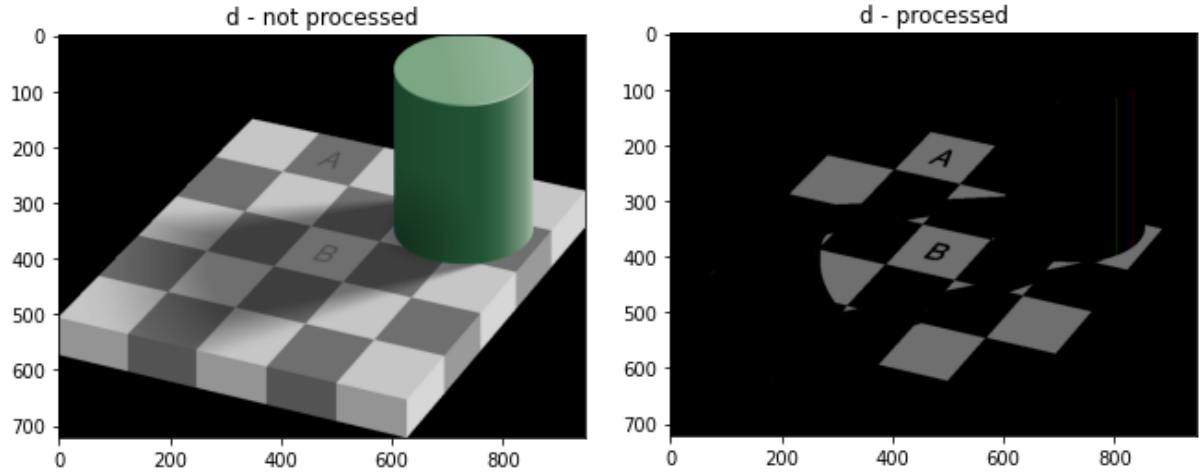
c. Ebbinghaus illusion: Although the orange circle on the right appears larger than its peer on the left, they are exactly the same size.



I only kept the pixels with the same color as the pixel at (200,550). The only colored parts left in the picture are the two circles.
Because if you have the same amount of pixels with same color, you can't create a different circle (anything else would not become a circle), I counted the number of pixels for each circle and they were both equal to 9604.

```
Circle on the left pixel counts:  9604
Circle on the right pixel counts:  9604
```

d. Checker shadow illusion: It may seems impossible, but the squares marked A and B are exactly of the same color.



I only kept the pixels with the same color as the pixel at (250,500) which is in square marked A. You can see both squares marked with A and B are preserved in the processed picture which means they're of the same color.

2. Bringing A Century-Old Dream Into Reality
Choose four composites to your liking and use them as inputs in the following parts.

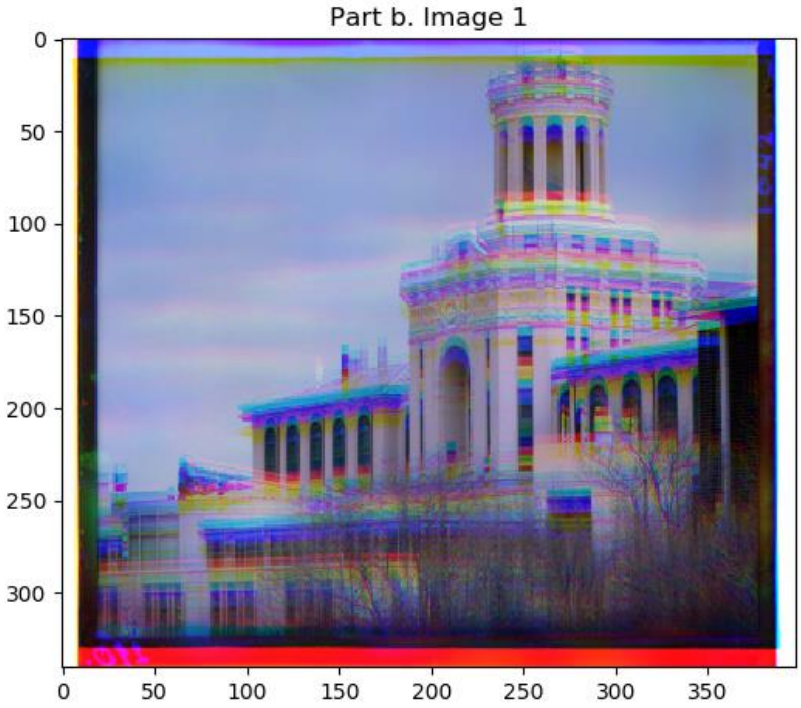
I chose the first four pictures. Simply calculated third of height and length of each picture and separated each channel by assigning those parts of picture to a different numpy array.

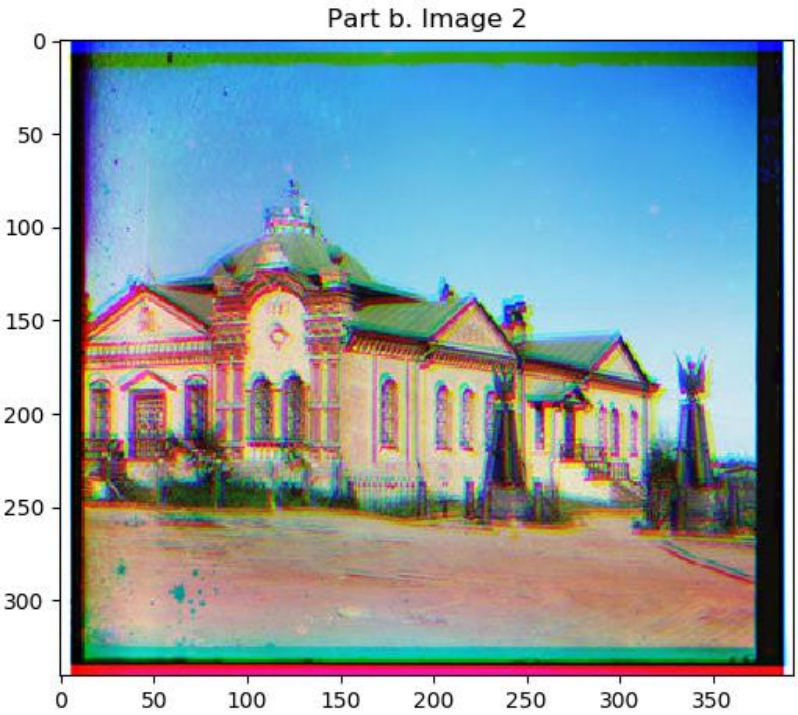
a. Preparation. Write a function extract_channels() which takes a triple-framed image as input, chops it vertically into three separate parts and save these channels with appropriate names.

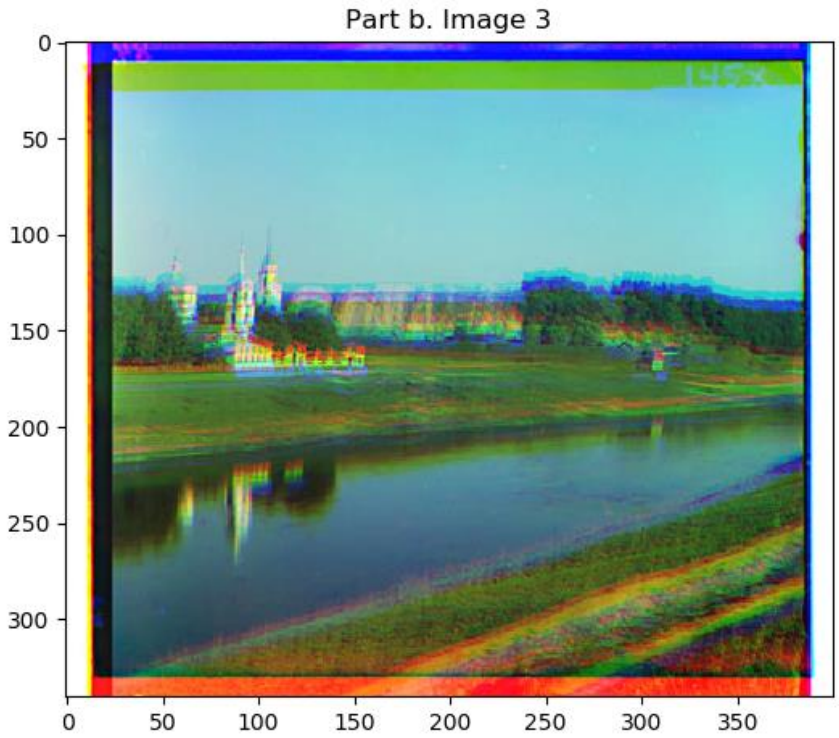
The channels are from top to bottom red channel, green channel, blue channel respectively.

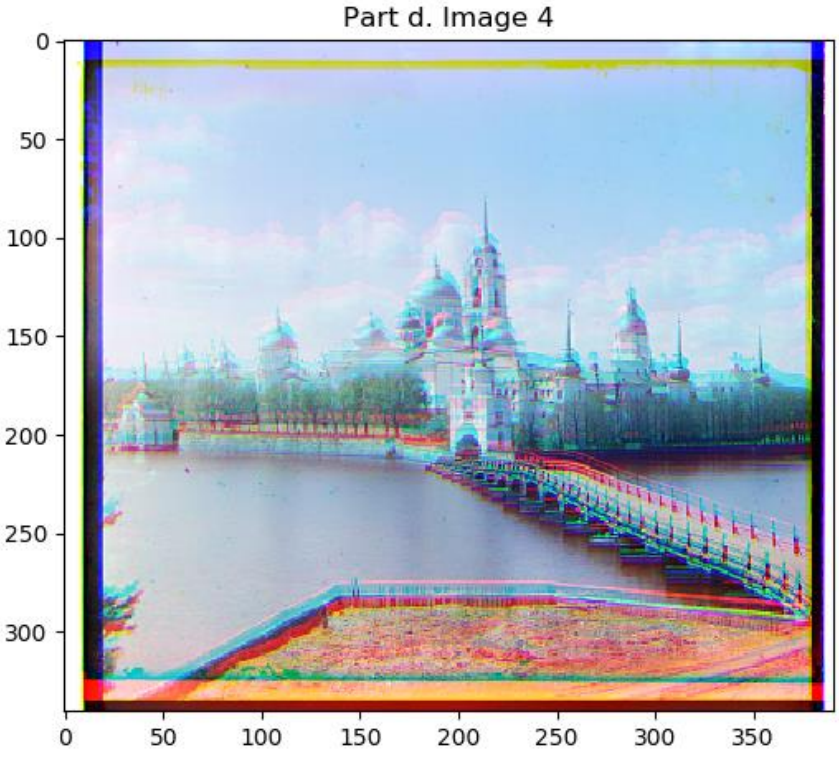
b. Merge. Implement another function, stack_channels(), which takes three channels as input and stack each of the three images into a proper color channel and display a single colored image. It is expected that the stacked photos look wonky and unaligned. Include it in your report.

Each channel is considered a matrix. I used numpy to stack them on their second axis.





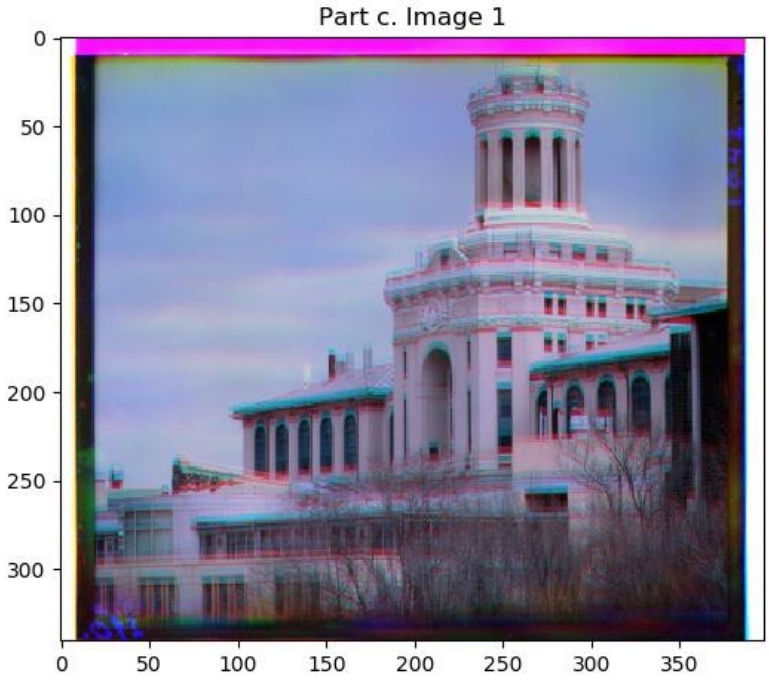

Part b. Image 3
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350

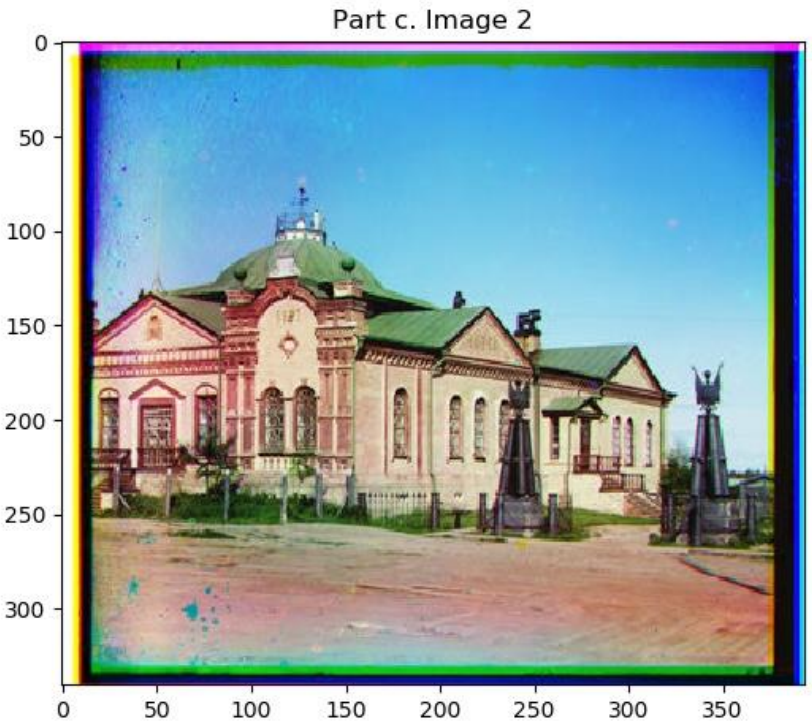

Part d. Image 4
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350

c. Alignment. Now write a third function, align_results(), which takes the image you obtained in the previous part and align it appropriately. This function must search over possible pixel offsets in the range of [-30, 30] to find the best alignment for each channel. One simple way to do so is to keep one channel fixed, and align the other two by searching over the offset range both horizontally and vertically. Pick the alignment that maximises a similarity metric (of your choice) between the channels, e.g. dot products, normalized cross correlation, etc.

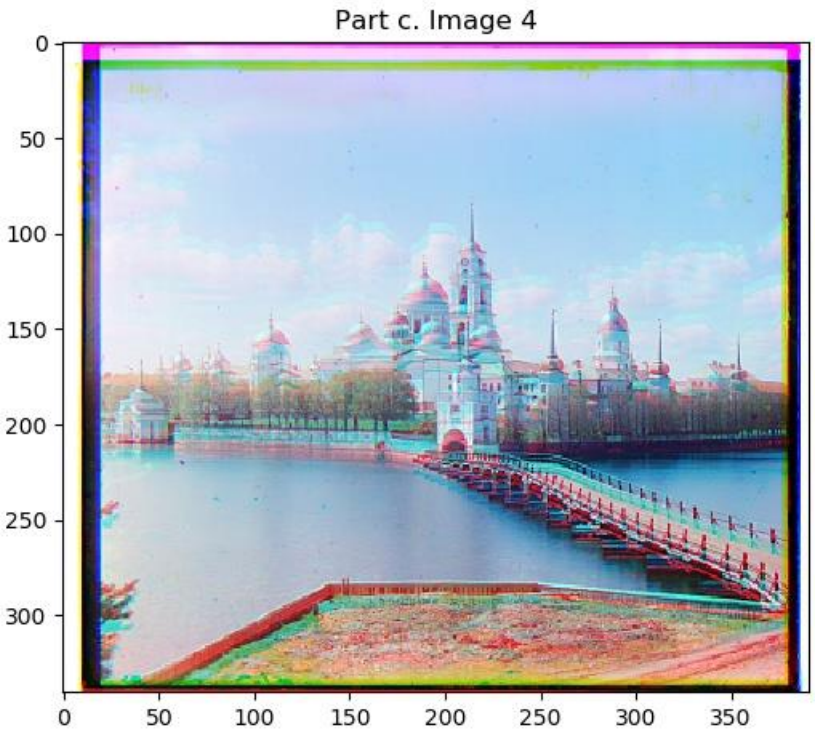
Note: For full credit, your report needs to include properly aligned images. Find a similarity metric that will accomplish this.

I used zero-normalized cross-correlation as the similarity matrix between different channels.
Because values from different channels won't necessarily be close to one another due to the fact that not everything that has same amount of red involved will have close amount of blue or green involved. Also "*For image-processing applications in which the brightness of the image and template can vary due to lighting and exposure conditions, the images can be first normalized.*" (From Wikipedia). So I thought a zero-normalization would do some good (It didn't have that much of an effect though I still kept it).




Part c. Image 2
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350

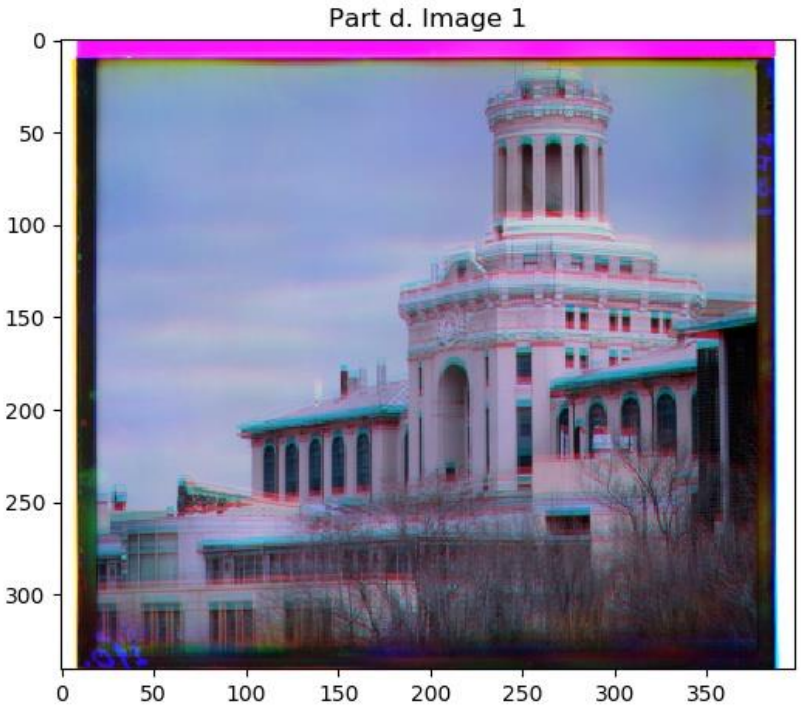

Part c. Image 3
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350

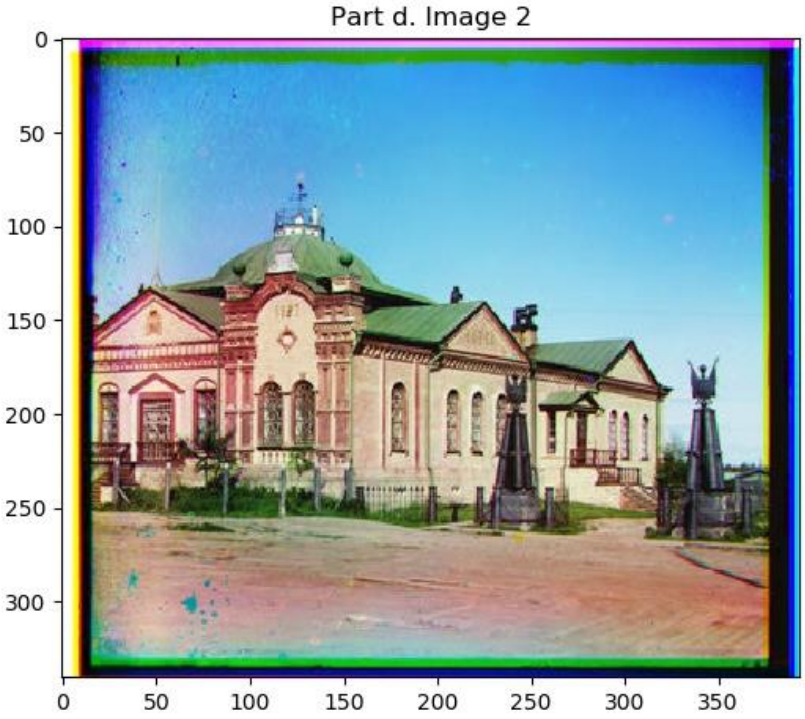


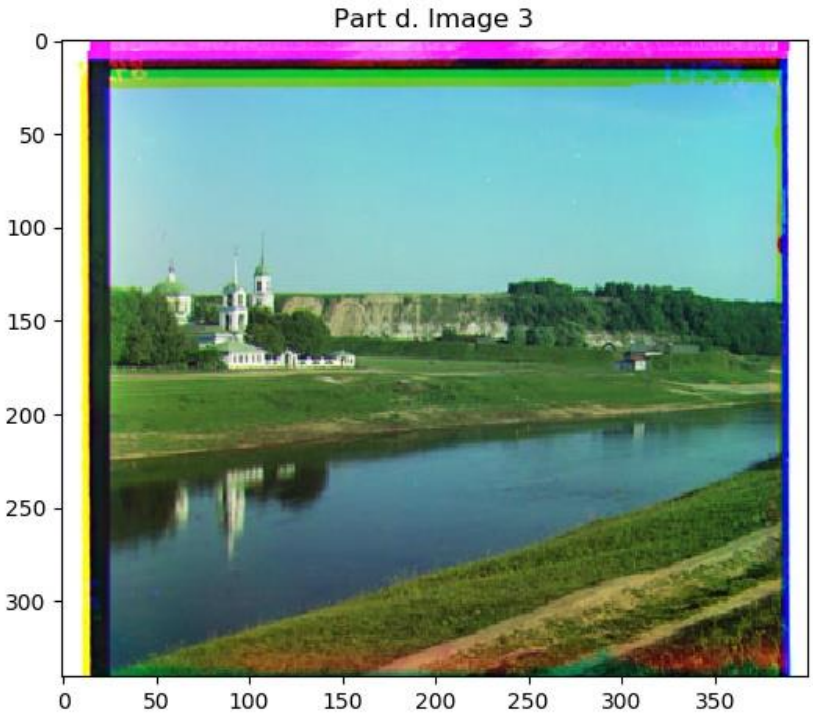
d. Resolution Pyramid. For large offsets and high resolution images, it can be computationally intensive to compare all the alignments for a specific range of displacements (e.g. [-30, 30]). It is often advised to start by estimating an alignment on a low-resolution version of the image before applying it on higher resolutions. In this part, you are asked to implement a two-level image pyramid by scaling the extracted channels down by a factor of 2, and then execute your alignment over the range of offsets [-15, 15]. After choosing the best alignment based on your similarity measure, use it as a starting place to again run the alignment in a small range [-15, 15] in the full resolution images.

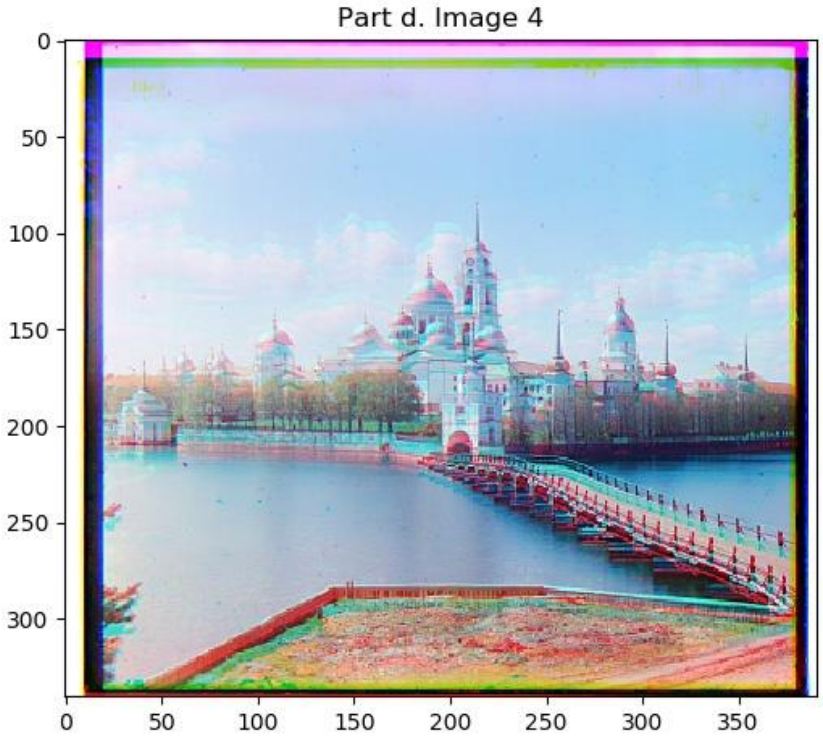
I did all of this in a function named 'efficient_align'.
Both part c. and d. are using a function 'nccAlign' which finds the coordinates that give the best alignment.

```
def ncc(a,b):
    a=a-a.mean(axis=0)
    b=b-b.mean(axis=0)
    return np.sum(((a/np.linalg.norm(a)) * (b/np.linalg.norm(b))))
def nccAlign(a, b, starti, startj, t):
    min_ncc = -1
    ivalue=np.linspace(-t,t,2*t,dtype=int)
    jvalue=np.linspace(-t,t,2*t,dtype=int)
    for i in ivalue:
        for j in jvalue:
            nccDiff = ncc(a,np.roll(b,[starti+i,startj+j],axis=(0,1)))
            if nccDiff > min_ncc:
                min_ncc = nccDiff
                output = [starti+i,startj+j]
    return output
```


Part d. Image 1
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350


Part d. Image 2
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350


Part d. Image 3
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350


Part d. Image 4
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350

3. Blue or Gold? Black or White? Let's End This Debate Once and for All!
This image, known as The Dress, is given to you along with three other images of this type (Figure 6), each with debatable colors. Your goal is to use image processing techniques in order to find the most frequent colors in each image and determine which side of these arguments are correct. Perform the following algorithm, known as K-Means, on each of the given images and clear up any ambiguity about their colors!
- Determine the number of colors needed to be extracted from the image, K.
- Randomly select K pixels as the centroids (not necessarily from the image).
- Assign each pixel to their closest centroid and form K clusters.
- Compute the new centroids by averaging the pixels in each cluster.
- Reassign each pixel to the new closest centroid. Repeat the previous part until no reassignment takes place.

I implemented K-Means (function 'kmeans') by creating three functions 'init_centroids', 'find_closest_centroids' and 'calc_centroids'.

a. Set K = 2, 3, 5, 7, and report the most frequent colors in each images.

In order to show the most frequent colors I used pyplot.pie. You can see the plots in the pictures included in part b.

b. Perform Image Segmentation by assigning each pixel to its closest color among those extracted in the previous part. Include the images in your report (16 images in total).

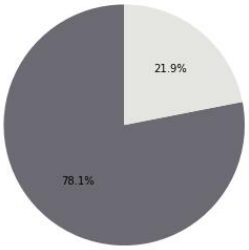
I defined a function 'ClusterColor' to assign the colors to each pixel.

```
def ClusterColor(image, k):
    #Reshaping the Image
    imagepix = image.reshape((image.shape[0] * image.shape[1], 3))
    #KMeans Clustering the Pixels to k Clusters
    cluster_centers_, labels_ = kmeans(k, imagepix)
    #Changing Pixel Colors to the k Centroids Colors
    imagenew = np.zeros_like(imagepix)
    count = np.zeros(k)
    for i in range(k):
        #Counting the amount of pixels that are close to a specific color
        ind = np.where(labels_==cluster_centers_[i])[0]
        count[i] = ind.shape[0]
        #print(count[i])
        imagenew[ind] = cluster_centers_[i]
    imagenew = imagenew.reshape((image.shape[0] , image.shape[1], 3))
    #print(cluster_centers_)

    return imagenew, count, cluster_centers_
```

The dress is indeed blue and black.

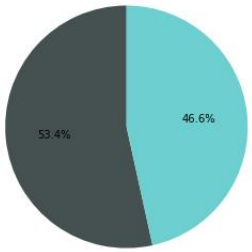
2 Clusters Dress Image

2 Clusters Dress Colors



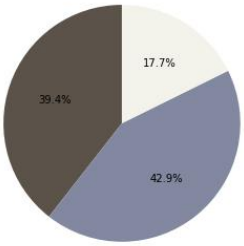
2 Clusters Strawberries

2 Clusters Strawberries Colors



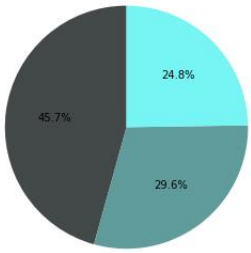
3 Clusters Dress Image

3 Clusters Dress Colors



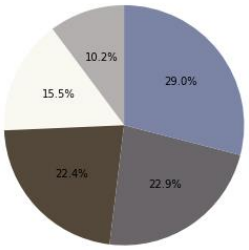
3 Clusters Strawberries

3 Clusters Strawberries Colors



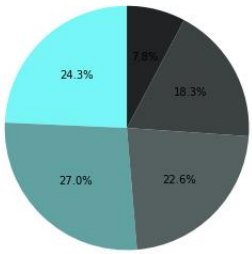
5 Clusters Dress Image

5 Clusters Dress Colors



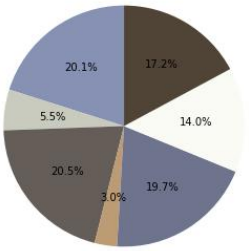
5 Clusters Strawberries

5 Clusters Strawberries Colors



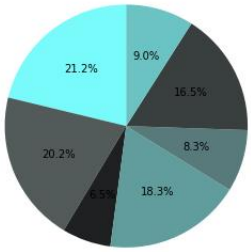
7 Clusters Dress Image

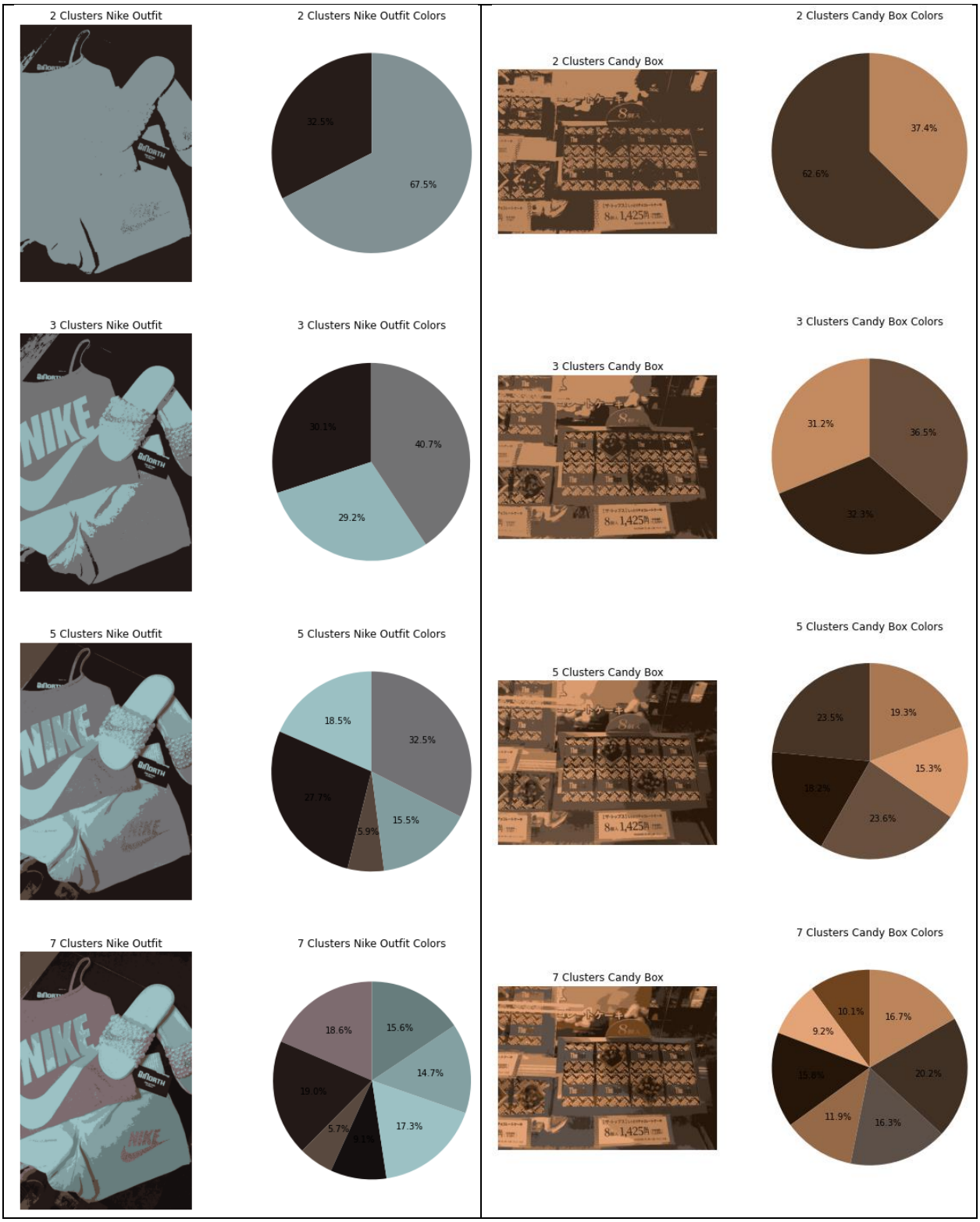
7 Clusters Dress Colors



7 Clusters Strawberries

7 Clusters Strawberries Colors





4. Where's Wally?

The aim of this problem is to get you familiar with the subject of Image Thresholding by solving a couple of "Where's Wally?" puzzles. Bearing in mind the fact that Wally always wears a shirt with the same

pattern, one can easily restrict the search area to those regions who are more similar to his shirt. In other words, the idea is to keep only red and white pixels in the images and perform the search in those areas.
You are provided with four different "Where's Wally?" puzzles. In each puzzle, your goal is to find the whereabouts of Wally and display his location in the original image. You are also given a few sample images of Wally (Figure 8) which you may find useful.

a. Find two approximate RGB values for the red and white parts of Wally's shirt.

I found a red and a white value for each image (given the coordinates of pixels) and then calculated their mean.

```
redmat = np.concatenate((red1,red2,red3,red4,red5), axis=0)
whitemat = np.concatenate((white1,white2,white3,white4,white5), axis=0)
red = redmat.mean(axis=0).astype(int)
white = whitemat.mean(axis=0).astype(int)
```

```
Red:  [245, 79, 86]
White:  [247, 244, 238]
```

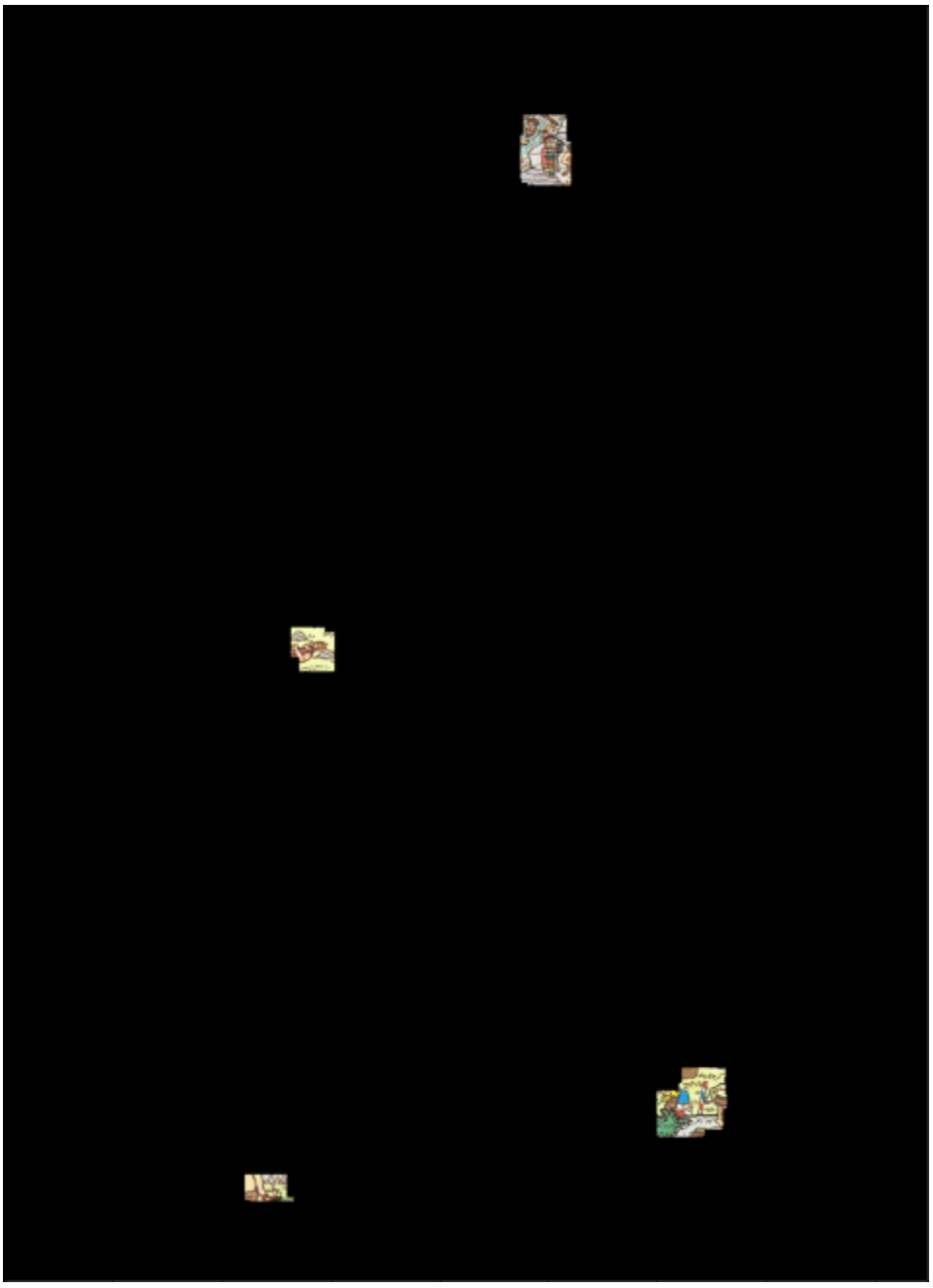
b. Try to preserve those pixels of the puzzles similar to Wally's shirt by setting appropriate thresholds using the values you found in the previous part. You may also need to keep a specific range of pixels around them, e.g. a pixels square. Set the values of other 50 x 50 pixels to zero (black), and display the results you obtained for each puzzle.

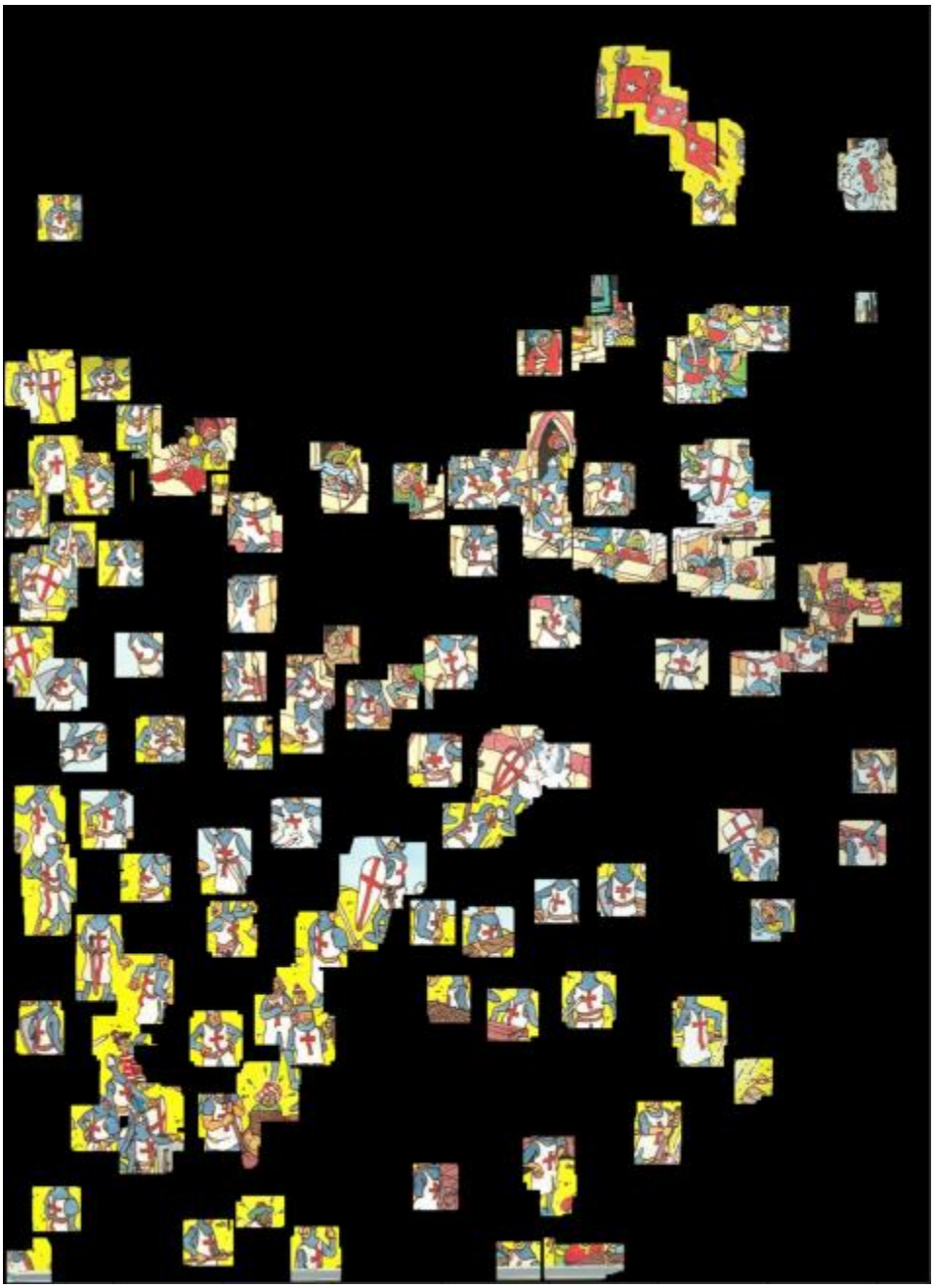
```
def keep_red(img):
    pic = img.reshape(img.shape[0]*img.shape[1],3)
    keep = np.zeros(pic.shape[0])
    keep[np.where(euclidean_distances(pic, np.expand_dims(red, axis=0))<=36)[0]] = 1
    keep = keep.reshape(img.shape[0],img.shape[1])
    numbers = np.where(keep != 0)
    location_red = np.array([numbers[0], numbers[1]]).T

    return location_red
```

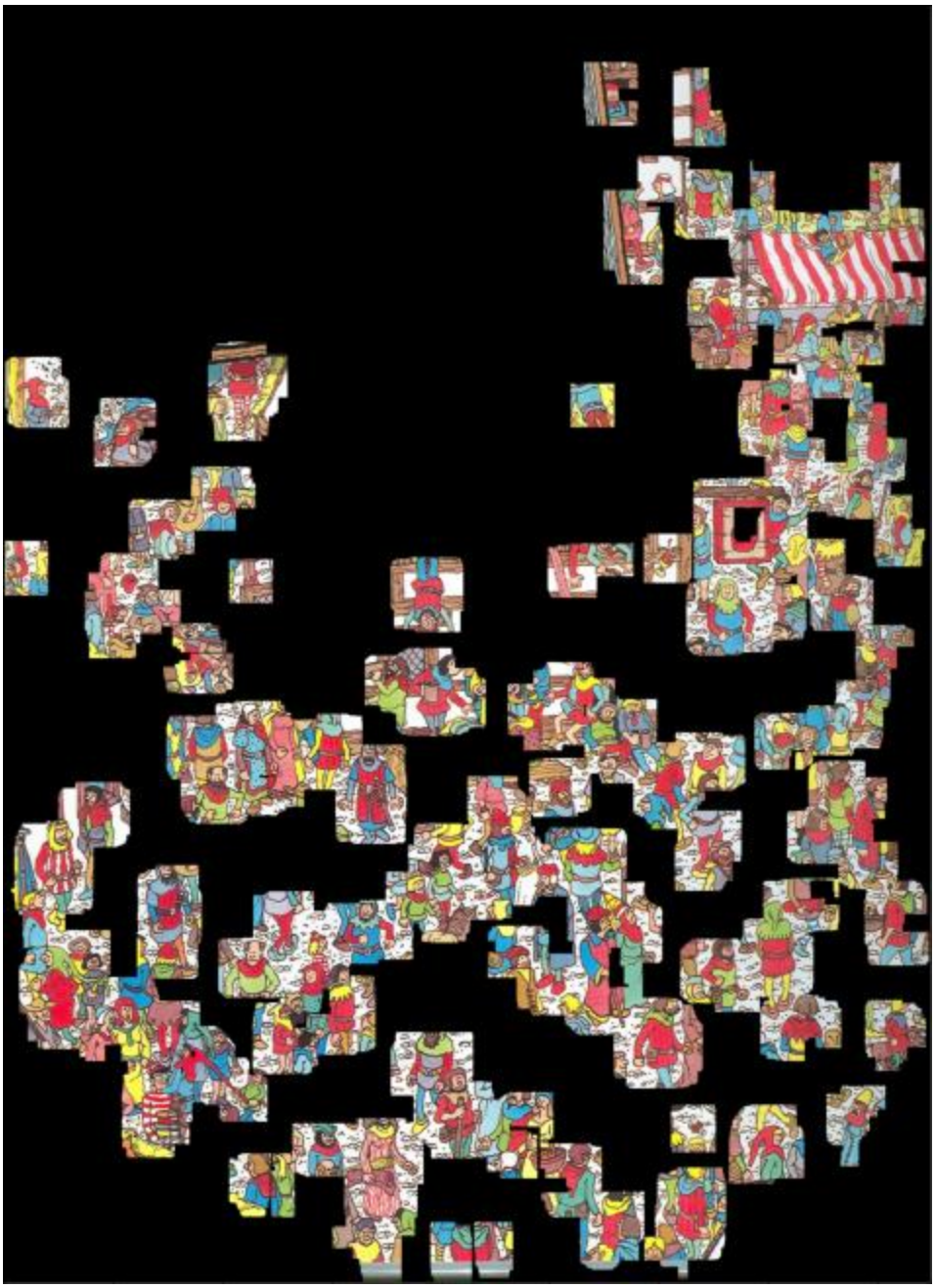
```
def keep_white(img):
    pic = img.reshape(img.shape[0]*img.shape[1],3)
    keep = np.zeros(pic.shape[0])
    keep[np.where(euclidean_distances(pic, np.expand_dims(white, axis=0))<=8)[0]] = 1
    keep = keep.reshape(img.shape[0],img.shape[1])
    numbers = np.where(keep != 0)
    location_white = np.array([numbers[0], numbers[1]]).T

    return location_white
```

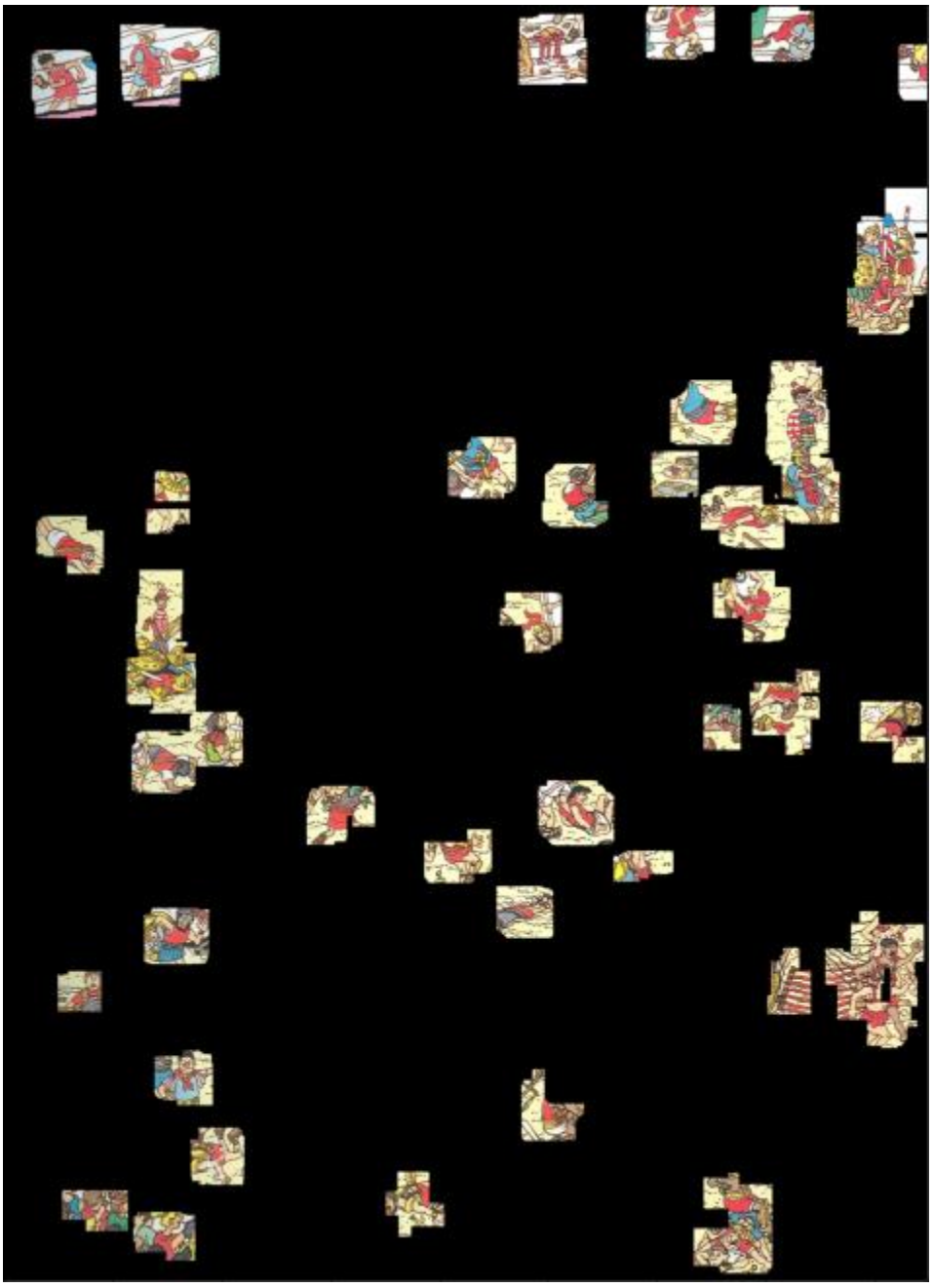
I used Euclidean distance in order to keep the colors closer to the 'red' value. Also later I do the same only for the white pixels in the newly created image 'new'.

```
location_red = keep_red(img)
new = np.zeros(img.shape)
for l in location_red:
    new[l[0]-40:l[0]+40,l[1]-40:l[1]+40] = img[l[0]-40:l[0]+40,l[1]-40:l[1]+40]
fig = plt.figure(figsize=(20,12))
new = new.astype('uint8')
plt.imshow(new)
plt.show()

location_white = keep_white(new)
new2 = np.zeros(img.shape)
for l in location_white:
    new2[l[0]-40:l[0]+40,l[1]-40:l[1]+40] = new[l[0]-40:l[0]+40,l[1]-40:l[1]+40]
new2 = new2.astype('uint8')
fig = plt.figure(figsize=(20,12))
plt.imshow(new2)
plt.show()
```









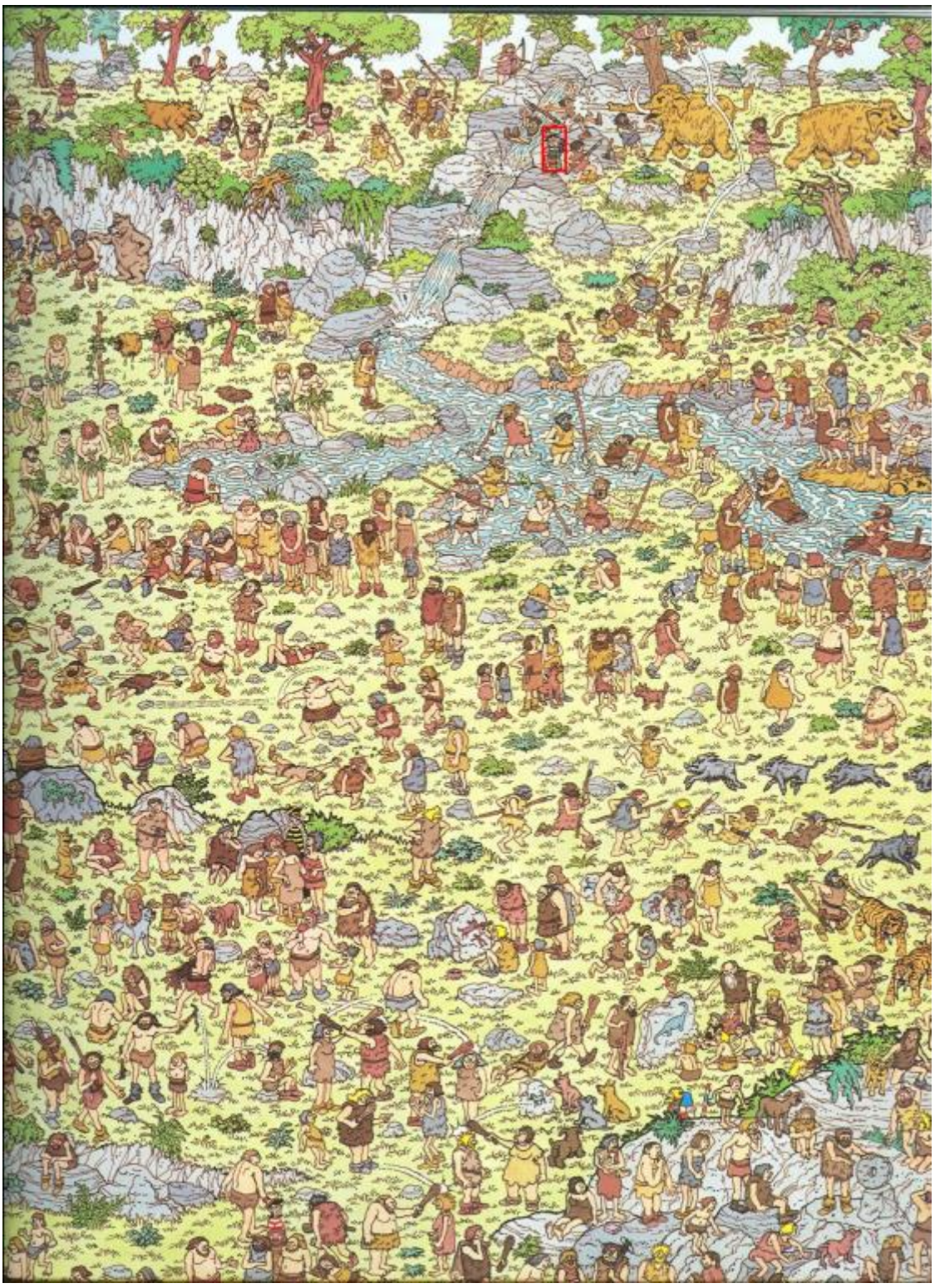
c. Search for Wally in the modified puzzles. You only need to search in non-black pixels, i.e. those who were detected similar to Wally's shirt.

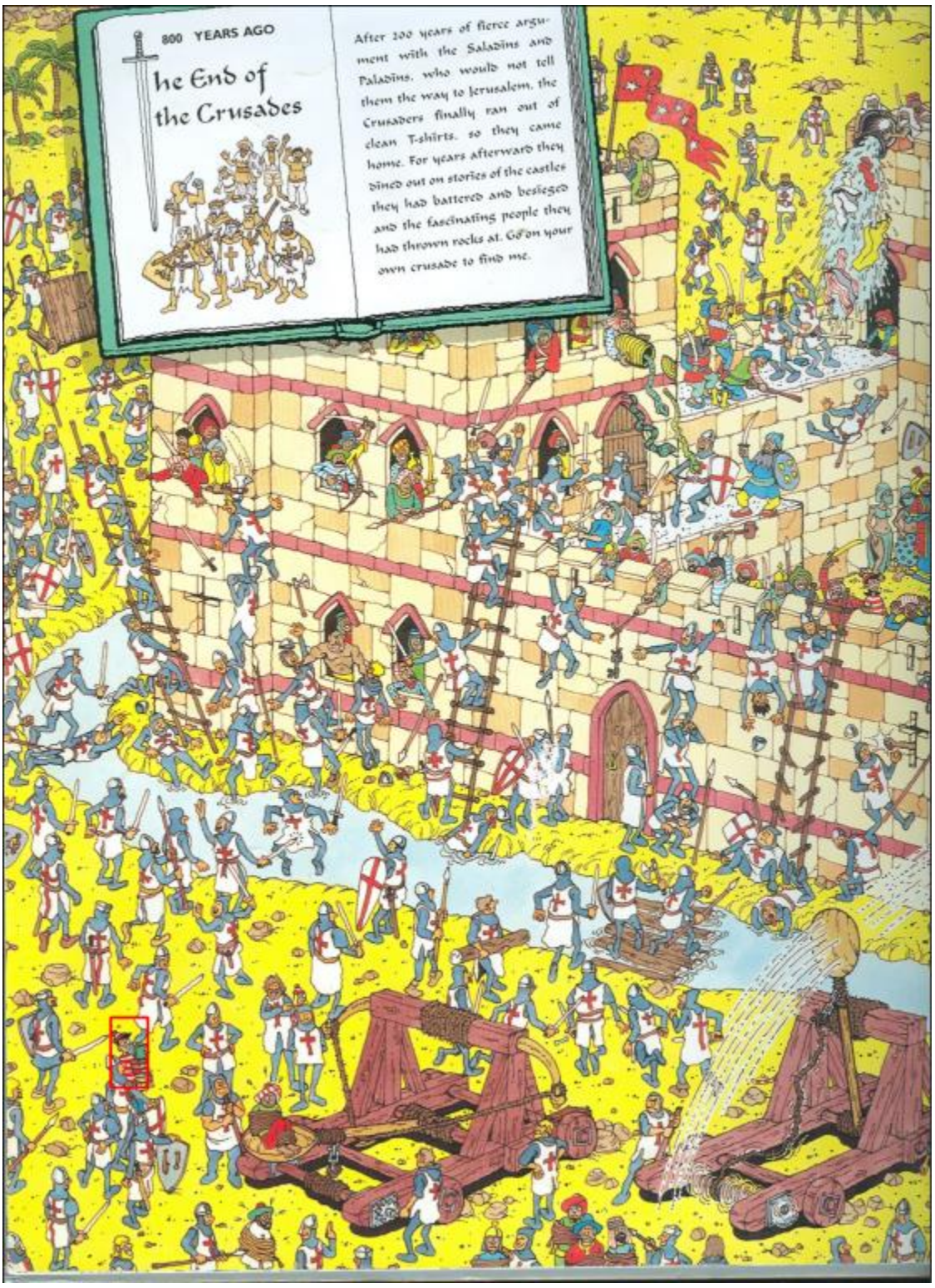
I used normalized cross-correlation and considered a threshold which I found for Wally in each picture through trial and error. Also Wally's picture needed to get resized for some pictures, so I defined a variable percent that was different for each picture.
I only used one Wally_2 as the template.

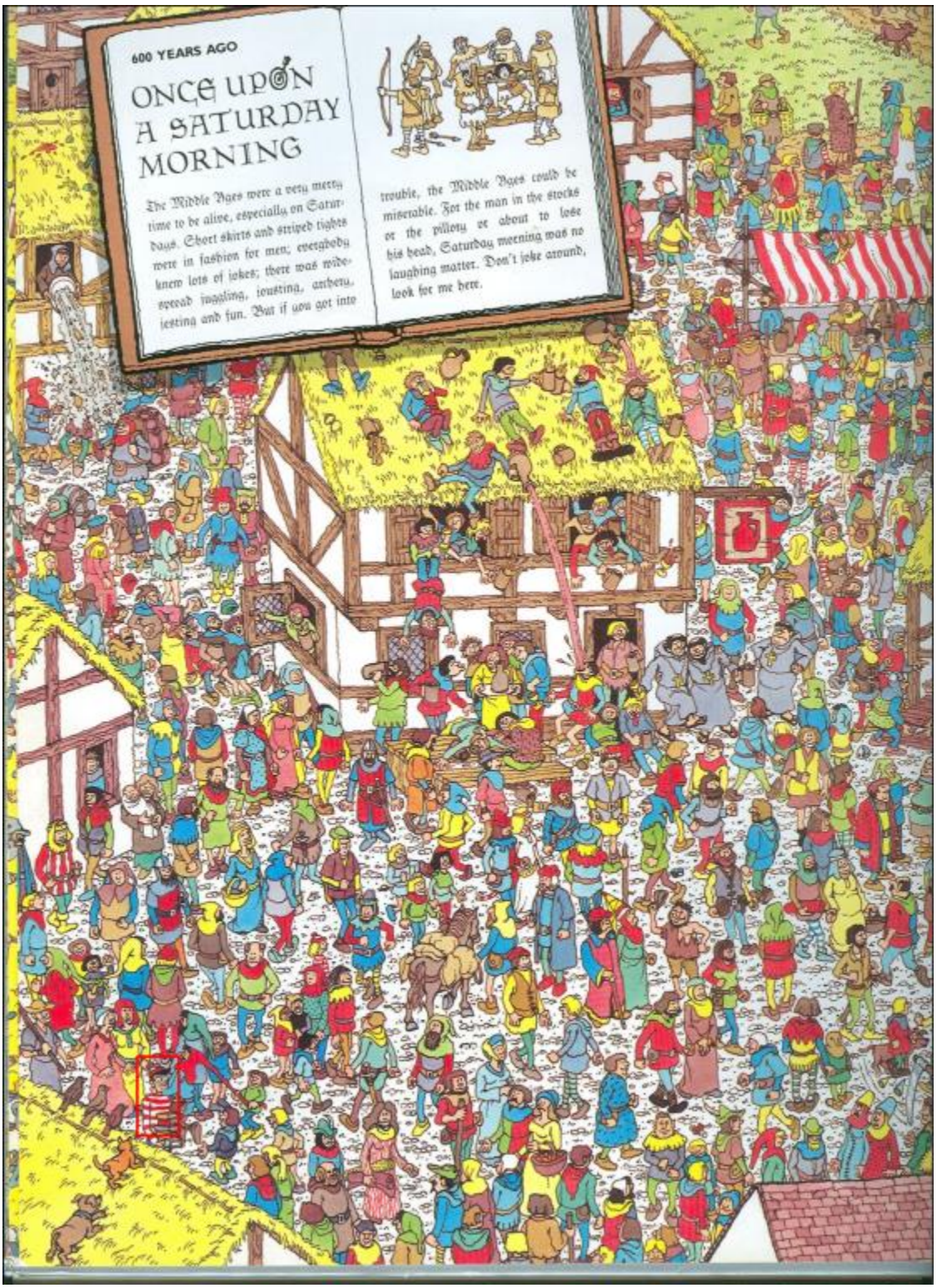
```
def ncc(a,b):
    try:
        return(np.sum(np.corrcoef(a.ravel(),b.ravel())))
    except:
        return 100000

def ncc_similarity(a, b, location, h, w, lower, upper):
    min_ncc = []
    output = []
    for l in location:
        nccDiff = ncc(a, b[l[0]-h:l[0]+h,l[1]-w:l[1]+w])

        if nccDiff > lower and nccDiff < upper:
            min_ncc.append(nccDiff)
            output.append(l)
    return output, min_ncc
```

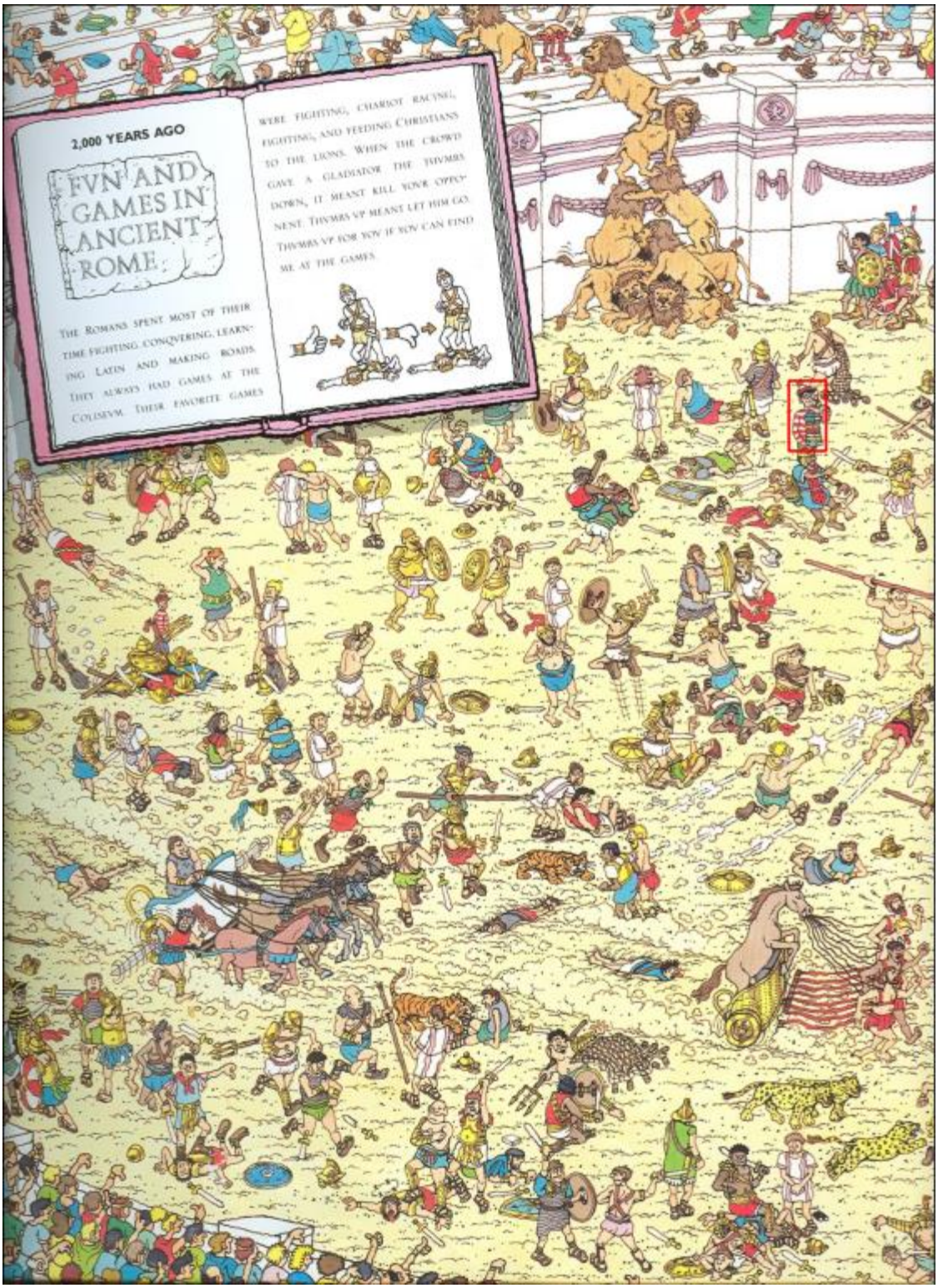
Variable 'location' that is used here is the coordinates of all the non-black pixels.

d. Display each puzzle with Wally's position indicated by a bounding box (rectangular borders around his coordinates).
Note: There are also several Wally lookalikes in each puzzle. Try not to be fooled by them.




800 YEARS AGO
he End of the Crusades
After 200 years of fierce argument with the Saladins and Paladins, who would not tell them the way to Jerusalem, the Crusaders finally ran out of clean T-shirts, so they came home. For years afterward they dined out on stories of the castles they had battered and besieged and the fascinating people they had thrown rocks at. Go on your own crusade to find me.


600 YEARS AGO
ONCE UPON A SATURDAY MORNING
The Middle Ages were a very merry time to be alive, especially on Saturdays. Short skirts and striped tights were in fashion for men; everybody knew lots of jokes; there was widespread juggling, jousting, archery, jesting and fun. But if you got into trouble, the Middle Ages could be miserable. For the man in the stocks or the pillory or about to lose his head, Saturday morning was no laughing matter. Don't joke around, look for me here.



5. Demographists and Statisticians Would Love Image Processing!
First, consider the map in Figure 10 which displays internet censorship and surveillance status across the world in 2018. Considering the legend in the table, answer the following questions:

For this exercise I kept the given color values in a list (the outer region color values were [0, 0, 0]). Then in a loop counted the number of pixels that have the given values. Later for each part I used these counts of pixels to answer the questions.

a. According to this map, what is the percentage of water on earth?
Hint: The RGB values for the outer regions are (255,255,255).

Water is the blue parts over all of the pixels minus the outer region.

```
Water Percentage:  74.68 %
```

b. What is the percentage of countries with pervasive censorship?

For this part I didn't count in the outer region and the waters.

```
Pervasive Percentage:  11.445 %
```

c. What is the percentage of countries which apply selective or little or no censorship?

For this part I didn't count in the outer region and the waters.

```
Selective, Little or no Percentage:  52.07 %
```

Next, consider another map given in Figure 11, which depicts provinces of Iran by population in 2014. The legend of the map is also included in the figure.

Thresholding was necessary for this question. I found the RGB values to the colors in the legend of the map and used Euclidean distance for thresholding. The reason I chose Euclidean distance is because in order to find close colors, it is necessary that all three values of RGB are close to the specified colors and also the only reason for using thresholding here is the pixels close to each province boundary (which have close RGB values to the province color they're in). That is why Euclidean distance would be sufficient here. A new picture was created due to thresholding.
I used these colors to count the pixels for each amount of population. Because of part f I thought it best to consider cells of 100x100 so I could only add up numbers for the provinces in the eastern part of the country (that are in some specific cells). I used these cells for counting.

d. Calculate the percentage of provinces with more than 4 million population.

```
Over 4 M:  27.46 %
```

e. Calculate the percentage of provinces with more than 2 million population.

```
Over 2 M:  64.7 %
```

f. What is percentage of provinces with less than 1 million population in the eastern part of the country?

I used the cells I had created.
Since it is said “provinces” in the eastern part of the country, I considered the ones that are completely in the eastern part. So I’m going to be keeping only the cells that are in the areas necessary, then count the amount of pixels for less than 1 M in those cells.

```
Less than 1 M East:  10.99 %
```

g. Approximately compare the northern provinces with southern provinces in terms of population.

I assume you meant us to use the percentage of each province in north and south (considering these areas only, not the whole country, for calculating percentage) to have a weighted sum of populations in order to compare them later.
Also I have to include that I considered those provinces as northern or southern that are in that area and are attached to the borders of the country.
I used the cells I had created.
Again I only used the necessary cells.

```
North:  3010548
South:  3213481
```

Now let’s deal with a more complicated case. The evolution of Iran population divided into urban and rural areas is shown in Figure 12.

Needed to crop out the extra parts of the picture.
Again I created cells and counted the colors in each of them.
I found the colors through counting the pixels for different colors in the picture and considering the first three frequent colors as the necessary ones.

h. Find the ratio between the numbers of urban residents to the total population of the country over the given period.

```
Urban in the given period:  0.595856026016999
```

i. What was the percentage of rural residents between the years 1996 to 2006?

```
Rural between 1996 to 2006:  34.14 %
```

j. Calculate the exact number of city dwellers in the decade started from the year 1976.

```
Urban between 1976 to 1986:  21
```

k. Since when the population of urban residents started to be more than the population of rural areas?

```
Urban Percentage in each Decade:  [36.03 43.77 52.1 58.28 65.86 72.11] %
```

Between 1976 and 1986 the population is half urban, so the answer is about 1986.

Finally, you are given Iran population pyramid in the year 2020.

Needed to crop out the extra parts of the picture.
I used the same approach that I had with cells, the difference was that only rows were necessary. I counted the amount of red and blue pixels in each 101 row.

l. Do women outnumber men in Iran? Calculate each gender's population.

For number of women we need the percentage of red pixels to the number of pixels containing the zero to 1 million indicator space and for men the same thing but with blue pixels. Then we can just multiply them by 1 million. Women outnumber men.

```
Number of Women:  338500
Number of Men:  334525
```

m. What is the percentage of those over 50?

```
Over 50 Percentage:  15.29 %
```

n. Compare the population of male and female under 20.

```
Number of Women under 20:  108872
Number of Men under 20:  108068
```

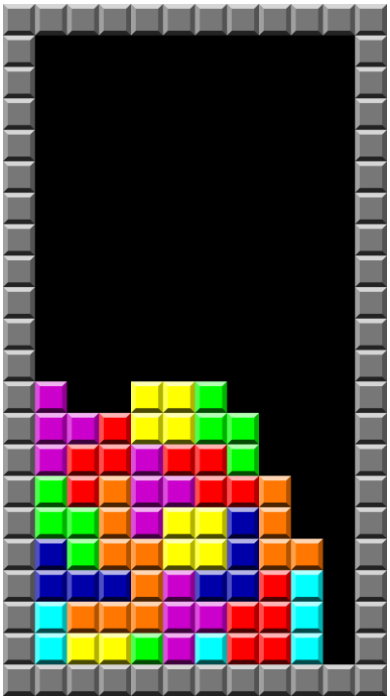
o. Find the exact number of men in their 40's (between 40 and 50 years old).

```
Number of Men in their 40s:  44256
```

Note: The surplus populations indicated with darker colors must be ignored in calculations.

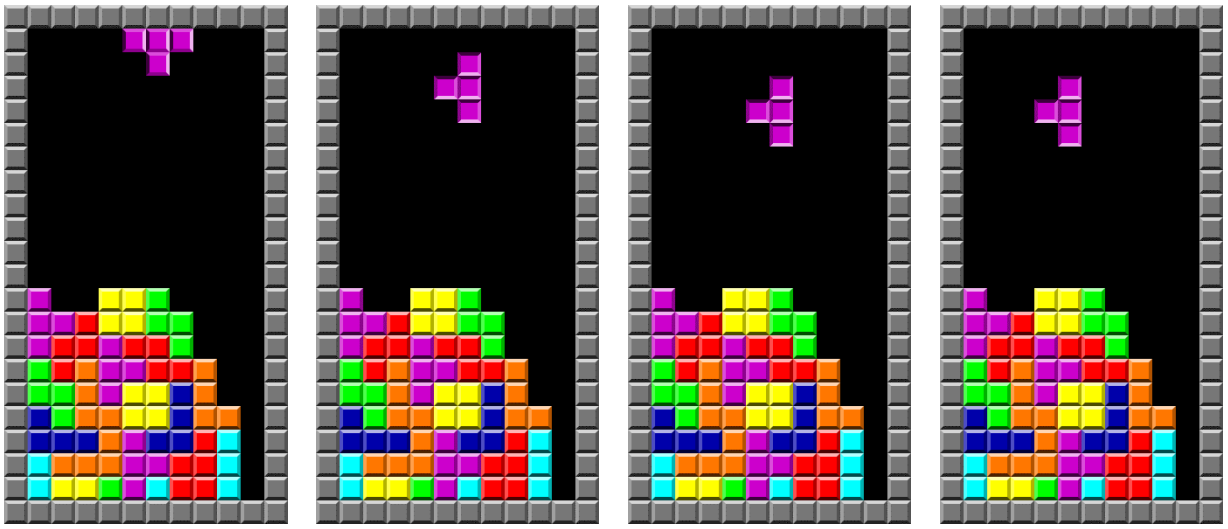
6. Let's Play Tetris!

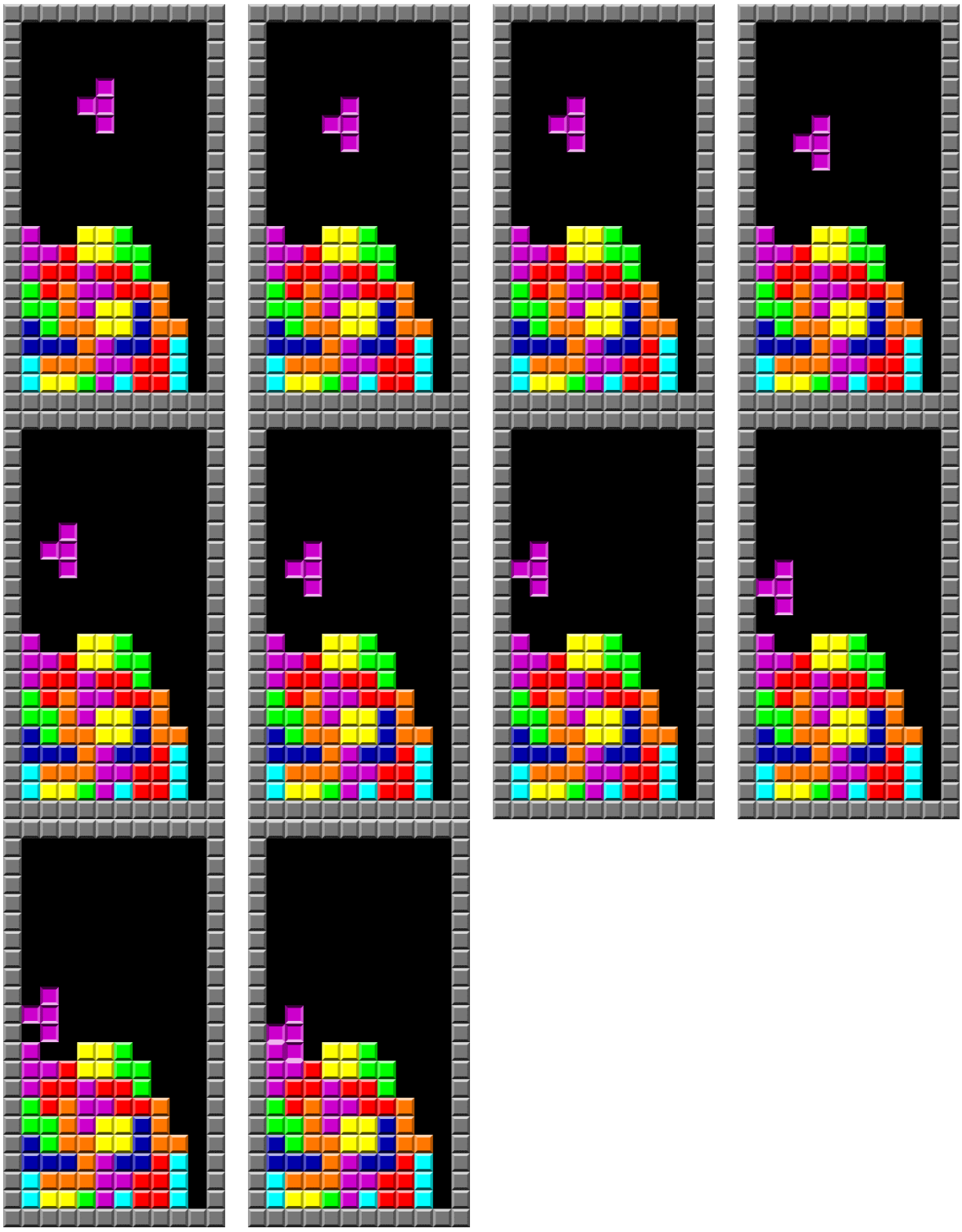
The starting picture:



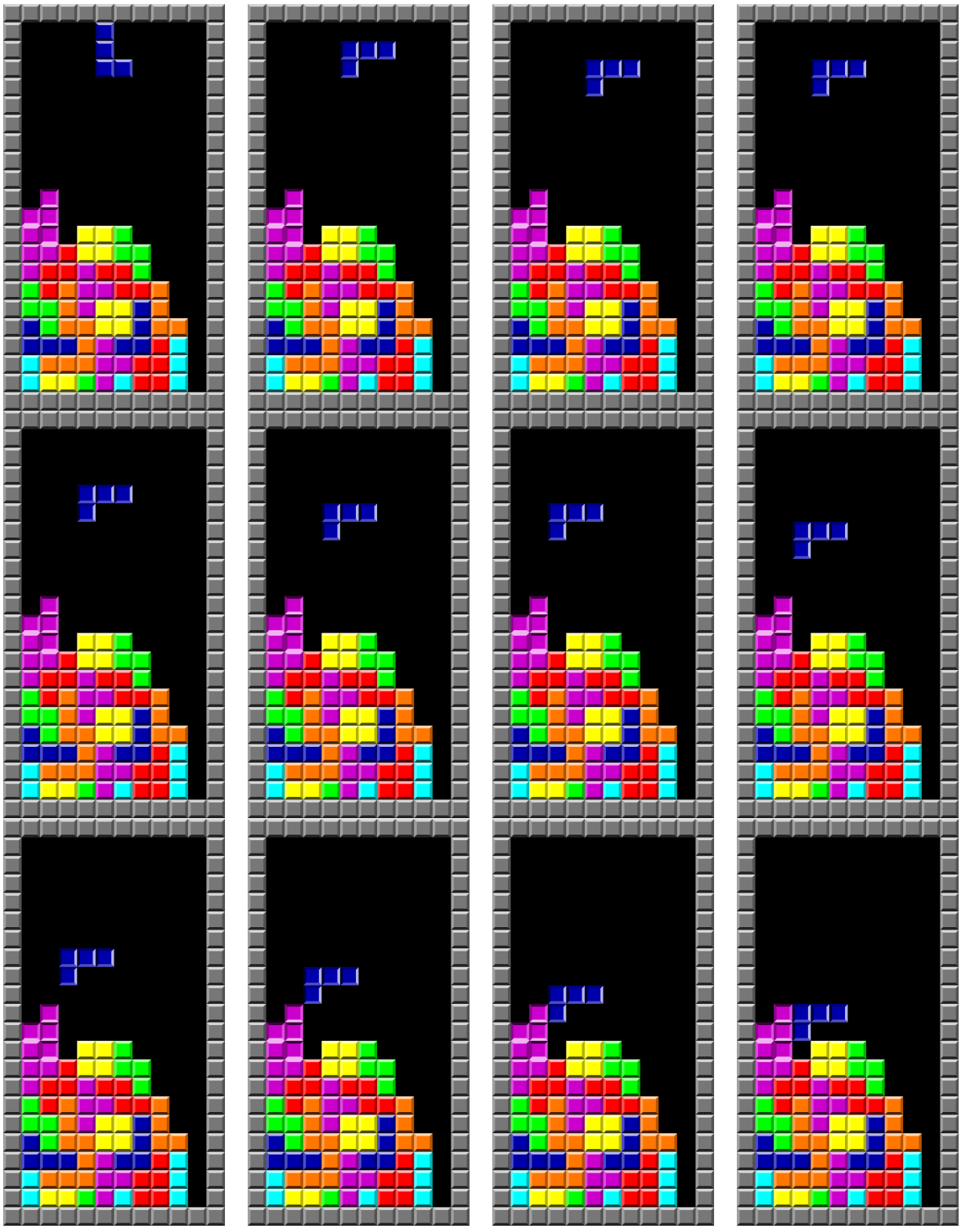
In the end six lines were removed. I have generally explained the code at the end of this question.

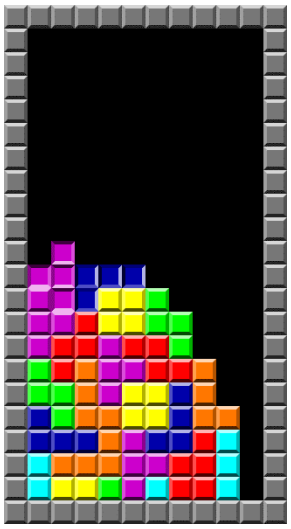
a. Play with 'T' and display all the intermediate states as well as the final one.



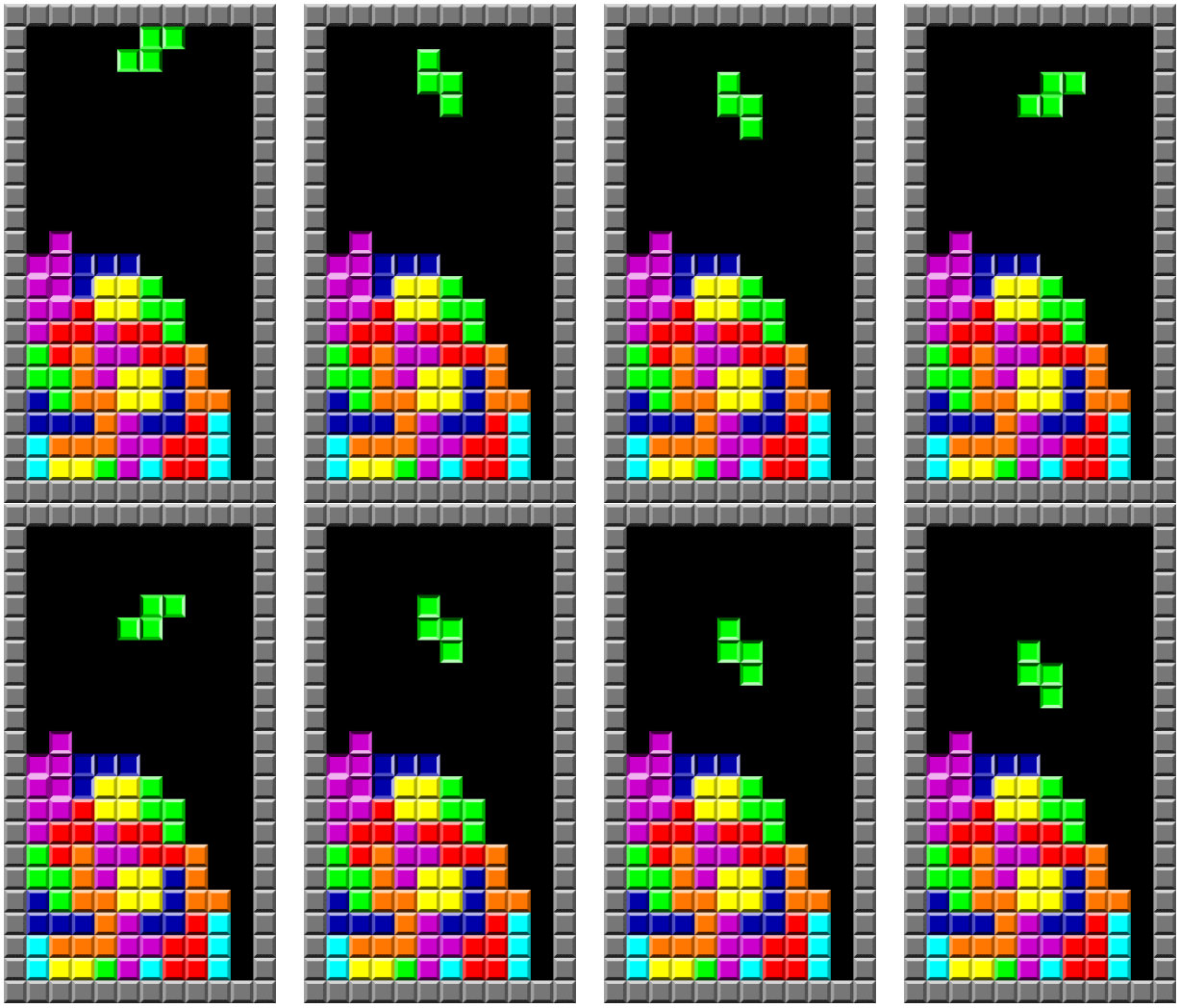


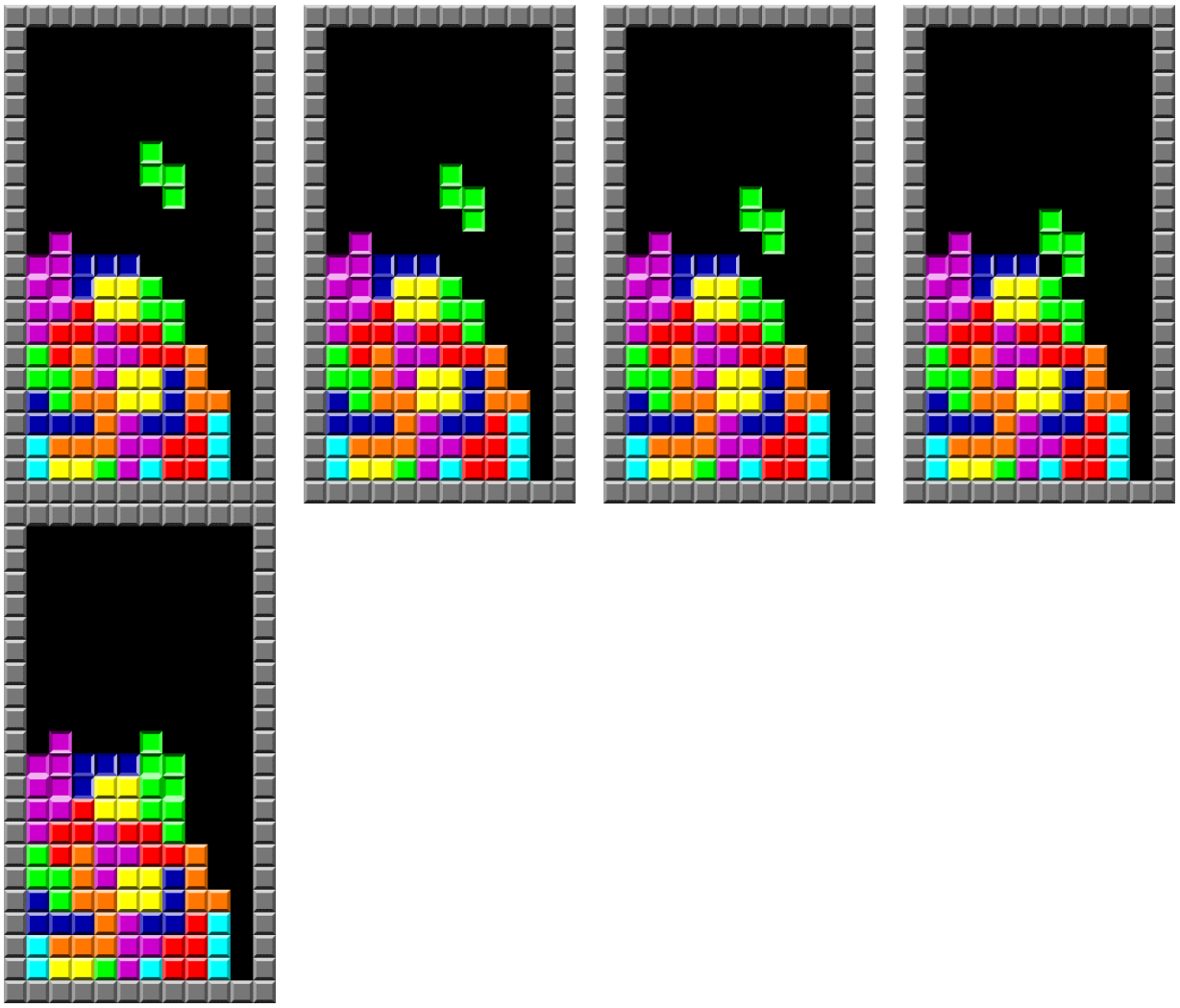
b. Play with ‘L’ and display all the intermediate states as well as the final one.



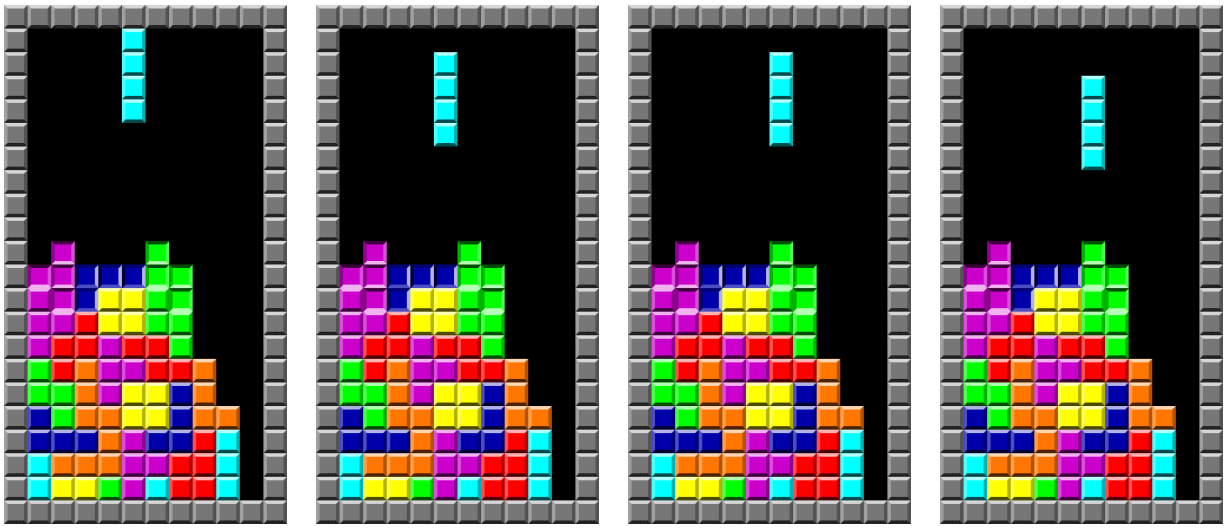


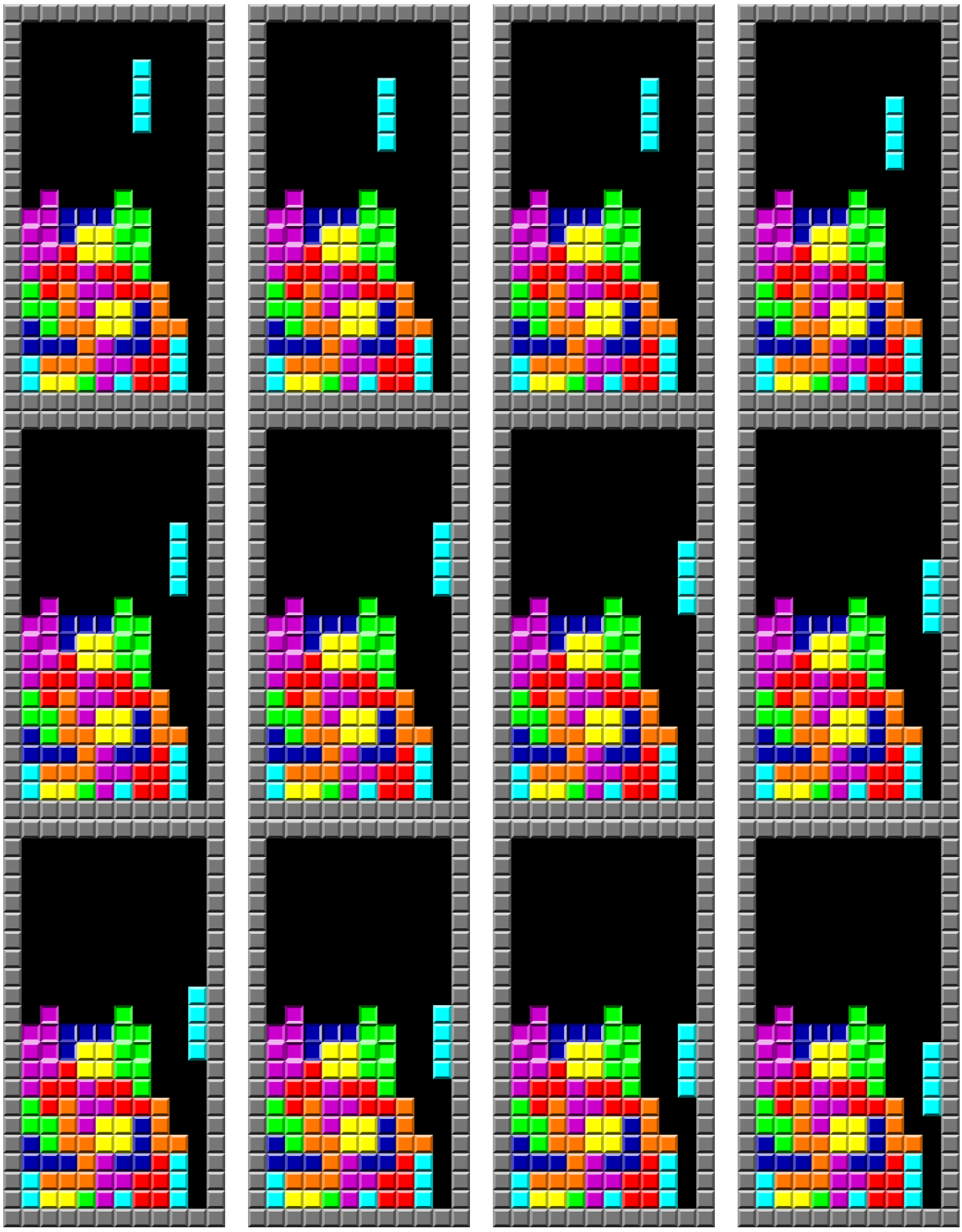
c. Play with ‘S’ and display all the intermediate states as well as the final one.

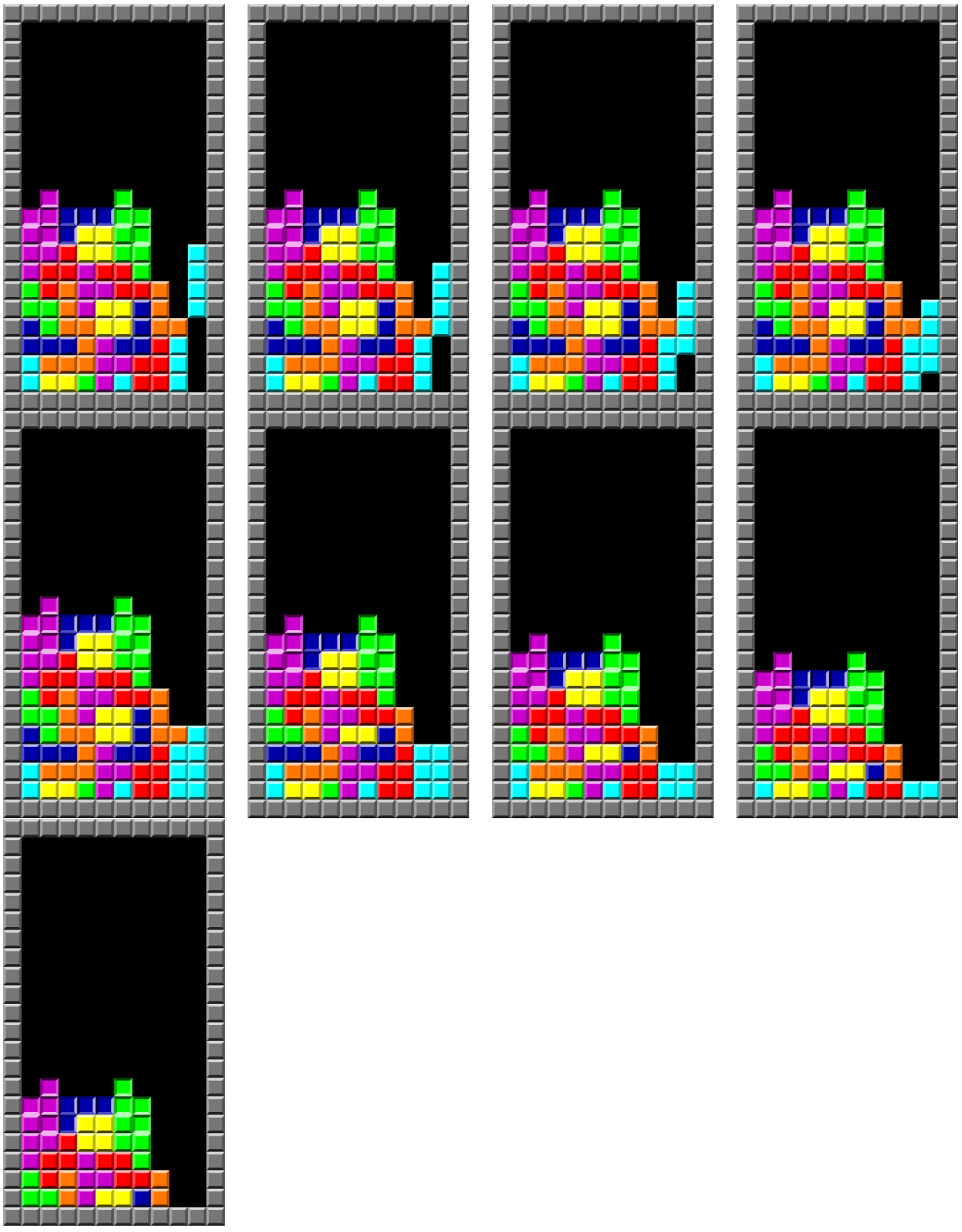




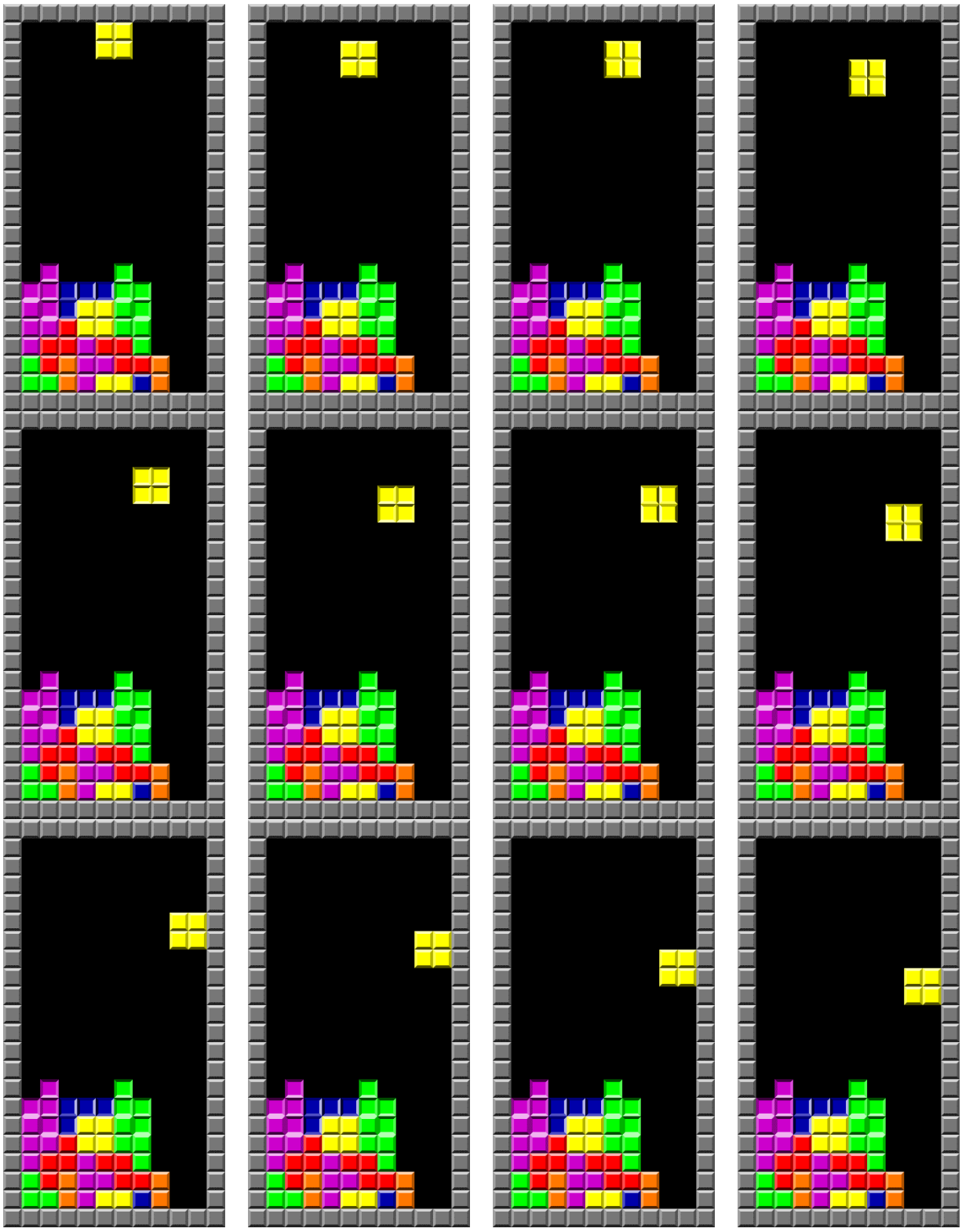
d. Play with ‘I’ and display all the intermediate states as well as the final one.

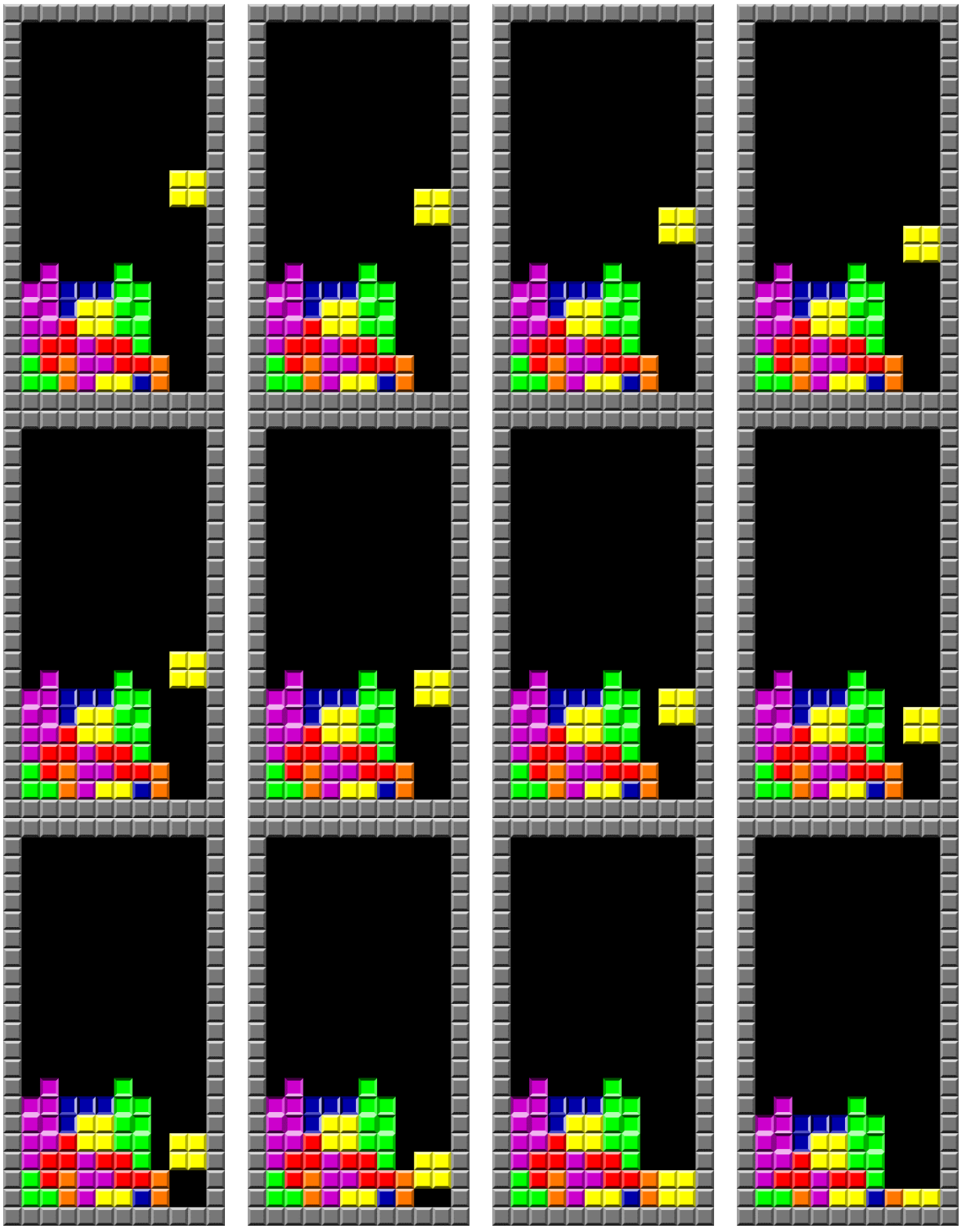


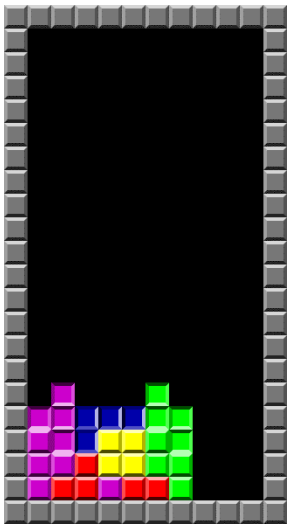




e. Play with ‘O’ and display all the intermediate states as well as the final one.







f. Use the frames obtained above and create a video sequence of all the moves. Consider a 0.5 second pause between each move.

I used opencv library for this part (cv2.videowriter class).
The video is included in the output folder. In order to have a 0.5 second pause, I considered 2 frames per second. The output video is about 46 seconds long and I have 92 pictures.

Code Explanation:
I use the image itself to define the state matrix by checking if the centers of each unit block is colored or black. I used rotation matrix (however because I often get confused with matrices I implemented the multiplications linearly) in order to rotate each block image whenever necessary. Then I did the same thing as the Tetris image to explain the block images in a matrix form (it is necessary when we need to update the state). I didn't keep the rotated blocks though and this rotation is done every time that we need a rotated block. I implemented a function 'next_state' for making the new state and showing the Tetris state and another function named 'run' that uses 'next_state' and does a few other things as well. I used a copy of the image and a copy of the state matrix so no changes will occur in the main image and the main matrix until I set a variable named 'isset' equal to True. Whenever this variable is True, The main image and the main matrix will change. This happens when a block has sat upon another block after the rows of blocks are checked and if necessary removed. As I recalled the rows would get removed one by one in the original game so this is the approach I took. Every time a block has sat we check if a row is completely colored (using a state matrix, a row completely equal to 1 except for the walls that I've considered -1) and if there were any we start removing them.
I defined the moves and ran the function 'run' for as many moves the game took. I did however sometimes rotate the blocks just for the sake of showing that the rotation works well.

7. Some Explanatory Questions

a. Explain how the optical illusion in Figure 1 works.

This happens when the photoreceptors, primarily the cone cells, in our eyes become overstimulated and fatigued causing them to lose sensitivity. As a result, we experience what is known as a negative

afterimage. As we shift our eyes to wall, the overstimulated cells continue to send out only a weak signal, so the affected colors remain muted.
However, the surrounding photoreceptors are still fresh and so they send out strong signals that are the same as if we were looking at the opposite colors. The brain then interprets these signals as the opposite colors, essentially creating a full-color image from a negative photo.
According to the opponent process theory of color vision, our perception of color is controlled by two opposing systems: a magenta-green system and a blue-yellow system.
For example, the color red serves as an antagonist to the color green so that when you stare too long at a magenta image you will then see a green afterimage. The magenta color fatigues the magenta photoreceptors so that they produce a weaker signal. Since magenta's opposing color is green, we then interpret the afterimage as green. [1]

b. Two sets of images are available. Set 1 contains two images, image A is of the size 400x300 whereas image B is of the size 1600x1200. Set 2 also contains two images, image C with 8- bit and image D with 24-bit color depth. Compare the images in each set with each other in terms of quality.

Set 1: The amount of pixels used for capturing a picture of an object means a smoother picture (if both images have the same color depth). Because each pixel is an approximation for the color of a part of the object and more pixels means smaller parts are assigned to each pixel.
Set 2: More bits means we can define more colors. So in this set, image D will have a lot of variety in terms of color.

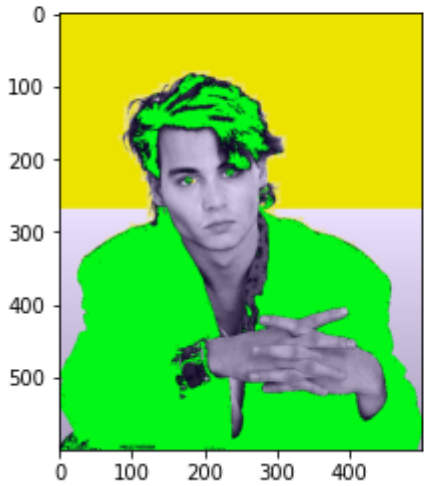
c. Is grayscale to RGB conversion possible? Justify your answer.

Yes, RGB is merely values for red, green and blue channel. So if you give each channel some values you would have RGB image. One could give the same grayscale values to all of the channels, the outcome will be gray (since there is the same amount of each color). But if the values for each channel were different from one another, the output would be colored.
Like This:

[1] https://www.verywellmind.com/the-negative-photo-illusion-4111086

```
plt.imshow(np.stack([image,image-10,image+20], axis=2))
```

```
<matplotlib.image.AxesImage at 0x1a8c524fd68>
```

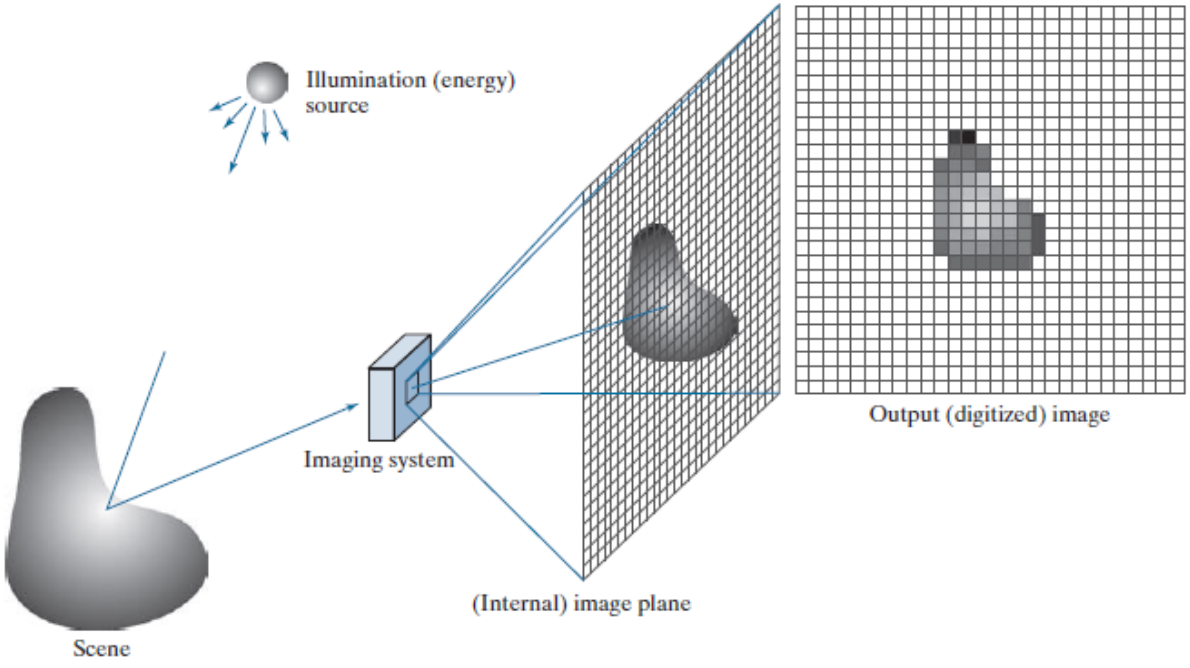


d. Why do we have different color spaces? Do they have specific advantages/disadvantages over each other?

It's about history; having new assumptions about how human eye can perceive colors and trying to create tools to capture photos. As these tools differed new color spaces emerged. Each have their own advantages/disadvantages. For example CMYK is used in the printing process, because it describes what kinds of inks are needed to be applied so the light reflected from the substrate and through the inks produces a given color, or HSV is modeled based upon how colors are organized and conceptualized in human vision in terms of other color-making attributes and it's easier to understand for people like artists what type of combination can make a certain color.

e. Explain the procedure of digitizing a continuous image in detail. Provide example if necessary.

In a continuous image each horizontal line intensity values will be the output of a function getting the x coordinate. For digitization first one must do sampling and sample out values for specific distances of x that gets determined by how many pixels are considered for the image. Then in quantization we consider a value indicating a point in an intensity scale (how many shades of gray we would have depends on the number of bits) for each sample. This should be done for each line.


Illumination (energy)
source
Imaging system
Output (digitized) image
(Internal) image plane
Scene