Обнаружение объектов интереса и подсчёт их количества

Для начала импортируем необходимые библиотеки:

In [1]:
```
import numpy as np
```

Библиотека `numpy` добавляет поддержку больших многомерных массивов и матриц, вместе с высокоуровневыми математическими функциями для операций с этими массивами.

In [2]:
```
import matplotlib.pyplot as plt
from matplotlib import patches
```

Библиотека `matplotlib` используется для визуализации данных.

In [3]:
```
import cv2
```

Библиотека `cv2` (OpenCV) — это open source библиотека компьютерного зрения, которая предназначена для анализа, классификации и обработки изображений.

In [4]:
```
from skimage.measure import label, regionprops, regionprops_table
from skimage import feature
```

Ещё одна библиотека, предназначенная для обработки изображений, которую мы будем использовать, это `Scikit-image`.

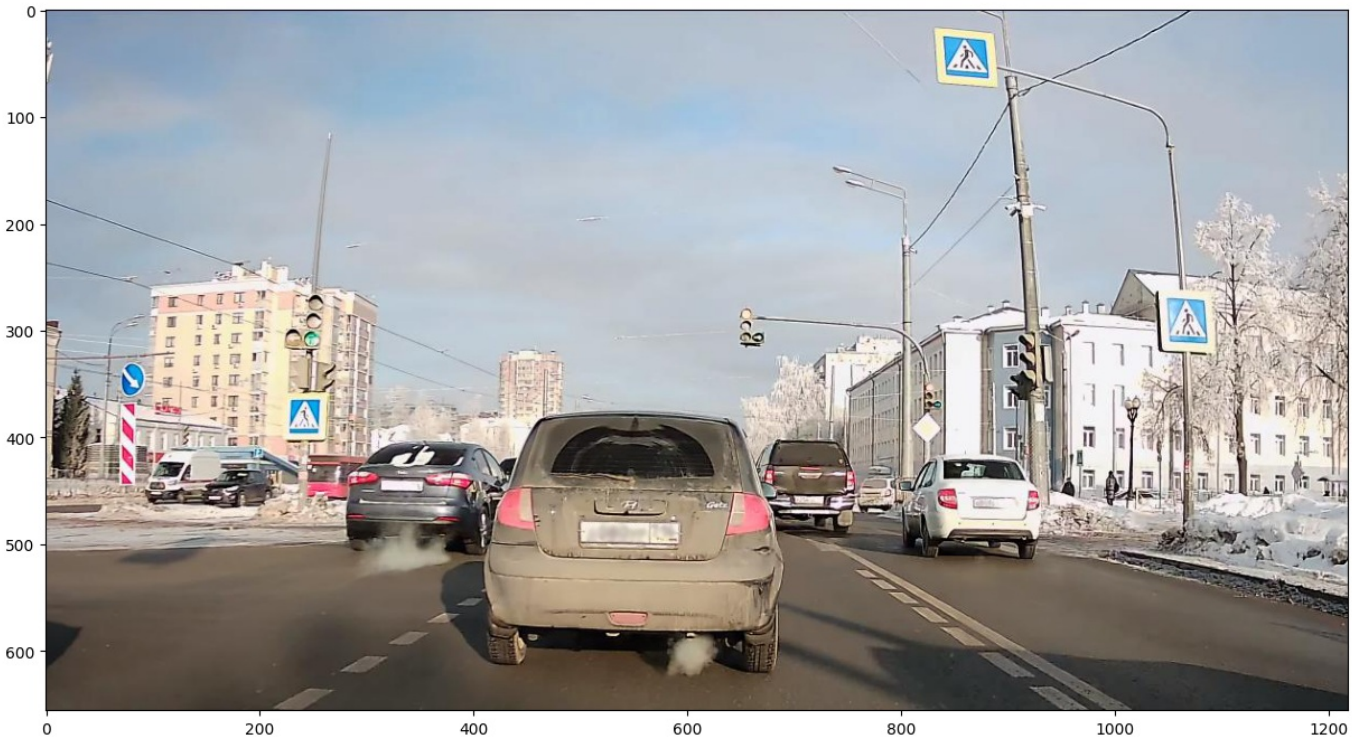
Итак, в качестве примера, возьмём изображение дорожной сцены, полученное с видеорегистратора. На данном изображении присутствуют дорожные знаки «Пешеходный переход». Создадим алгоритм, предназначенный для распознавания, детекции и подсчёта таких дорожных знаков.

Загрузим изображение:

In [5]:
```
image = plt.imread('image.jpg')
```

Отобразим его:

In [6]:
```
plt.figure(figsize=(15,15))
plt.imshow(image);
```



Функция `plt.imread()` предназначена для чтения изображения из файла в массив. На вход функции, *в скобках*, необходимо указать *в кавычках* название файла, если он находится в одной папке с исполняемым файлом вашей программы, или указать путь к необходимому файлу.

Пример:

```
plt.imread('cat.png')
plt.imread('/home/fit/Downloads/cat.png')
```

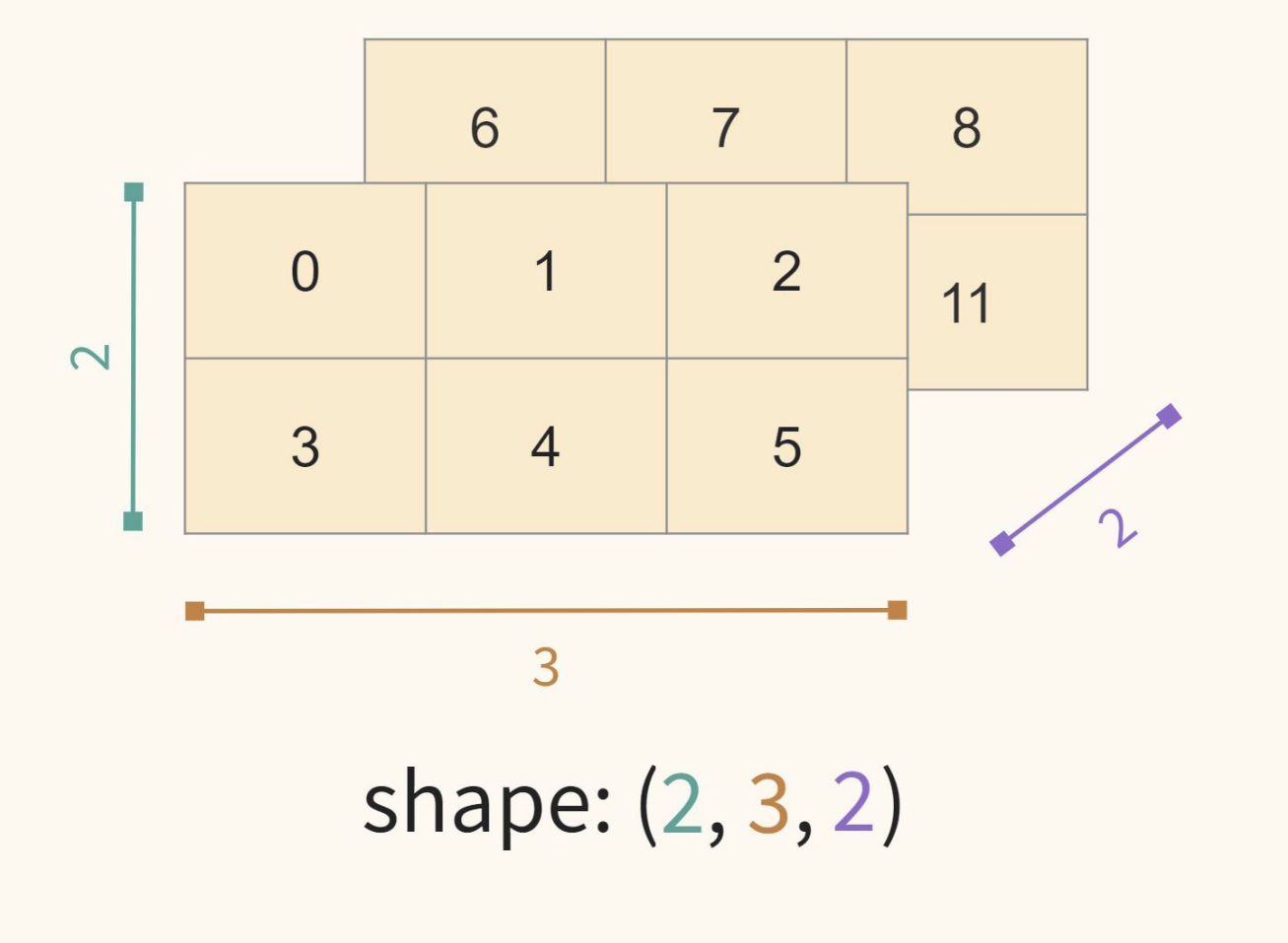
Таким образом, используя данную функцию, мы получаем массив, соответствующий нашему изображению дорожной сцены. Этот массив мы присвоили переменной `image`. Для того чтобы посмотреть размер полученного массива, можно воспользоваться методом `.shape`

In [7]:
```
print(image.shape)
```

```
(656, 1218, 3)
```

Первая цифра **656** значит, что в рассматриваемом массиве **656** строк, вторая цифра представляет собой количество столбцов массива, а именно **1218**. Третья цифра соответствует третьему измерению трехмерного массива, внутри которого будут **3** двумерных массива **656** x **1218**. Визуально это можно представить как стек (наложение) трёх матриц.

Пример:



В таком массиве каждый элемент представляет собой значение пикселя изображения. Размер массива (656, 1218, 3) означает, что рассматриваемое изображение имеет разрешение **656** x **1218** пикселей и **3** канала (R,G,B), соответствующих оттенкам красного, зелёного и синего.

В дальнейшем все операции и преобразования будут осуществляться над `numpy` массивами, соответствующими рассматриваемым изображениям. Для того чтобы отобразить массив в виде изображения используется функция `plt.imshow();`, на вход которой подаётся переменная содержащая массив.

Что касается самого массива, то мы можем обратиться к его элементам по координатам (индексам) строки, столбца и канала, перечисленным в квадратных скобках после названия переменной массива.

Пример:

In [8]:
```
print(image[1,5,0])
```

```
158
```

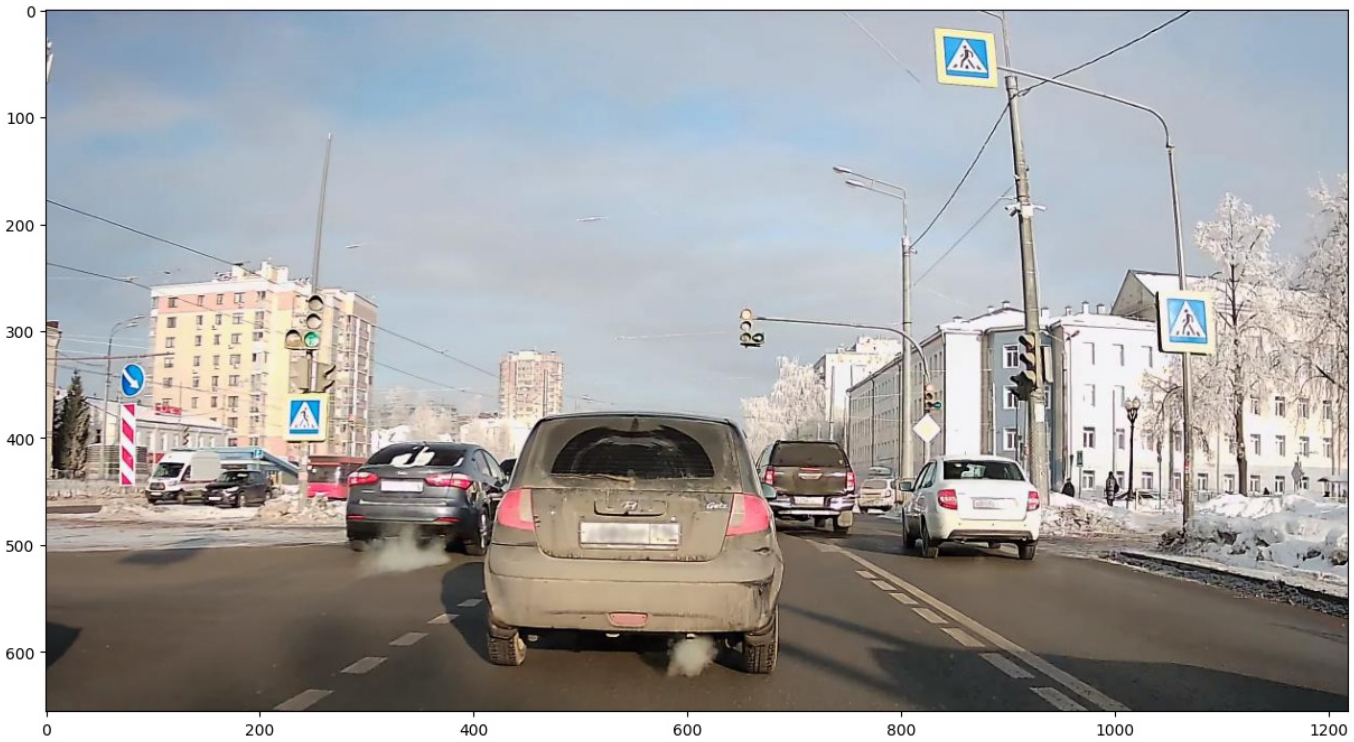
В примере порядок координат, указанных через запятую, соответствует порядку, который выдаёт нам метод `.shape`, а именно:

1. Первая цифра - это координата строки массива;
2. Вторая цифра - это координата столбца массива;
3. Третья цифра - это координата канала массива.

Важно помнить: Индексы и координаты начинаются с 0. Таким образом первая строка массива имеет индекс 0, так же как и первый столбец и первый канал.

Также мы можем обратиться к какой-либо части массива, используя в качестве координат срезы. Срезы реализуются с помощью символа `:`, слева от которого указывается с какого индекса должен быть взят срез, а до какого индекса указывается справа от символа `:`. В качестве примера возьмём срез, в котором содержится автомобиль, расположенный на переднем плане изображения.

In [9]:
```
plt.figure(figsize=(15,15))
plt.imshow(image);
```



Сперва определимся с координатами строк. Как видно на осях изображения подписаны номера пикселей по ширине и высоте. Строки соответствуют высоте изображения. Следовательно, автомобиль находится примерно между **350** и **650** строками. Укажем эти координаты в виде среза для строк.

```
image[350:650]
```

Далее через запятую нам нужно указать срез для столбцов. Как видно из изображения, автомобиль находится между **400** столбцом и примерно **700** столбцом. Укажем это в индексах массива.

```
image[350:650, 400:700]
```

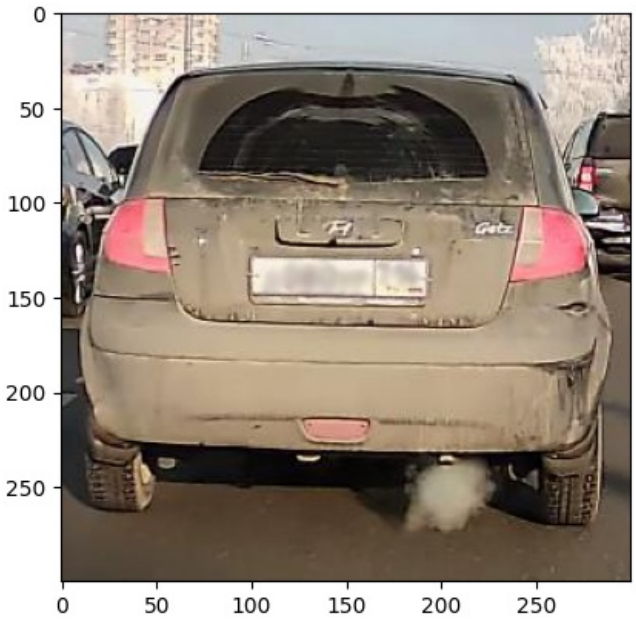
Если мы отобразим в консоли полученный срез, мы увидим массив соответствующей части массива, в которой находится автомобиль:

In [10]:
```
image[350:650, 400:700]
```

Out[10]:
```
array([[[199, 204, 207],
        [199, 204, 207],
        [199, 204, 207],
        ...,
        [227, 219, 208],
        [228, 219, 210],
        [191, 182, 173]],

       [[199, 204, 208],
        [199, 204, 208],
        [199, 204, 208],
        ...,
        [248, 240, 229],
        [216, 207, 198],
        [163, 154, 145]],

       [[199, 202, 207],
        [199, 202, 207],
        [199, 202, 207],
        ...,
        [240, 227, 219],
        [236, 223, 215],
        [167, 154, 146]],

       ...,

       [[ 76,  66,  56],
        [ 76,  66,  56],
        [ 76,  66,  56],
        ...,
        [104,  91,  75],
        [106,  92,  79],
        [105,  91,  78]],

       [[ 75,  67,  56],
        [ 75,  67,  56],
        [ 75,  67,  56],
        ...,
        [103,  90,  74],
        [107,  93,  80],
        [116, 102,  89]],

       [[ 75,  67,  56],
        [ 75,  67,  56],
        [ 75,  67,  56],
        ...,
        [102,  89,  73],
        [106,  92,  79],
        [113,  99,  86]]], dtype=uint8)
```

Если мы хотим посмотреть на изображение полученной части массива, то воспользуемся `plt.imshow`.

In [11]:
```
plt.imshow(image[350:650, 400:700]);
```



Ещё немного о том как обращаться к элементам массива:

Первым делом необходимо указать интересующие вас строки или одну строку.

In [12]:
```
image[0]
```

Out[12]:
```
array([[ 96, 105, 160],
       [ 96, 105, 160],
       [ 96, 105, 160],
       ...,
       [106, 117, 163],
       [106, 117, 163],
       [106, 117, 163]], dtype=uint8)
```

В таком случае вы обращаетесь ко всем столбцам этой строки и ко всем каналам. Далее указываются необходимые столбцы (или столбец).

In [13]:
```
image[0,0]
```

Out[13]:
```
array([ 96, 105, 160], dtype=uint8)
```

В данном случае мы обратились к элементу в первой строке первого столбца и ко всем трём каналам.

Также можно указать интересующий нас канал.

In [14]:
```
image[0,0,0]
```

Out[14]:
```
96
```

96 это значение элемента в первой строке первого столбца в первом канале. Аналогично со срезами, если мы хотим взять несколько строк массива, при этом нам интересны все столбцы и все каналы, то необходимо в квадратных скобках указать с какой строки по какую мы берём срез.

In [15]:
```
image[350:650]
```

Out[15]:
```
array([[[218, 205, 186],
        [197, 184, 165],
        [176, 160, 144],
        ...,
        [198, 182, 166],
        [171, 154, 144],
        [177, 160, 150]],

       [[183, 170, 151],
        [198, 183, 164],
        [227, 211, 195],
        ...,
        [171, 155, 139],
        [200, 183, 175],
        [209, 193, 180]],

       [[162, 149, 141],
        [137, 124, 116],
        [130, 116, 107],
        ...,
        [155, 138, 128],
        [187, 170, 160],
        [187, 170, 160]],

       ...,

       [[ 75,  65,  55],
        [ 75,  65,  55],
        [ 75,  65,  55],
        ...,
        [ 32,  36,  37],
        [ 36,  37,  39],
        [ 36,  37,  39]],

       [[ 75,  65,  55],
        [ 75,  65,  55],
        [ 75,  65,  55],
        ...,
        [ 33,  39,  39],
        [ 36,  36,  38],
        [ 36,  36,  38]],

       [[ 75,  65,  55],
        [ 75,  65,  55],
        [ 75,  65,  55],
        ...,
        [ 33,  39,  39],
        [ 35,  36,  40],
        [ 35,  36,  40]]], dtype=uint8)
```

Отобразим это в виде изображения.

In [16]:
```
plt.figure(figsize=(15,15))
plt.imshow(image[350:650]);
```

Если необходимо взять срез по некоторым столбцам, при этом нас интересуют все строки и все каналы, то в координатах для строк мы указываем только символ `:`.

In [17]:
```
image[:, 400:700]
```

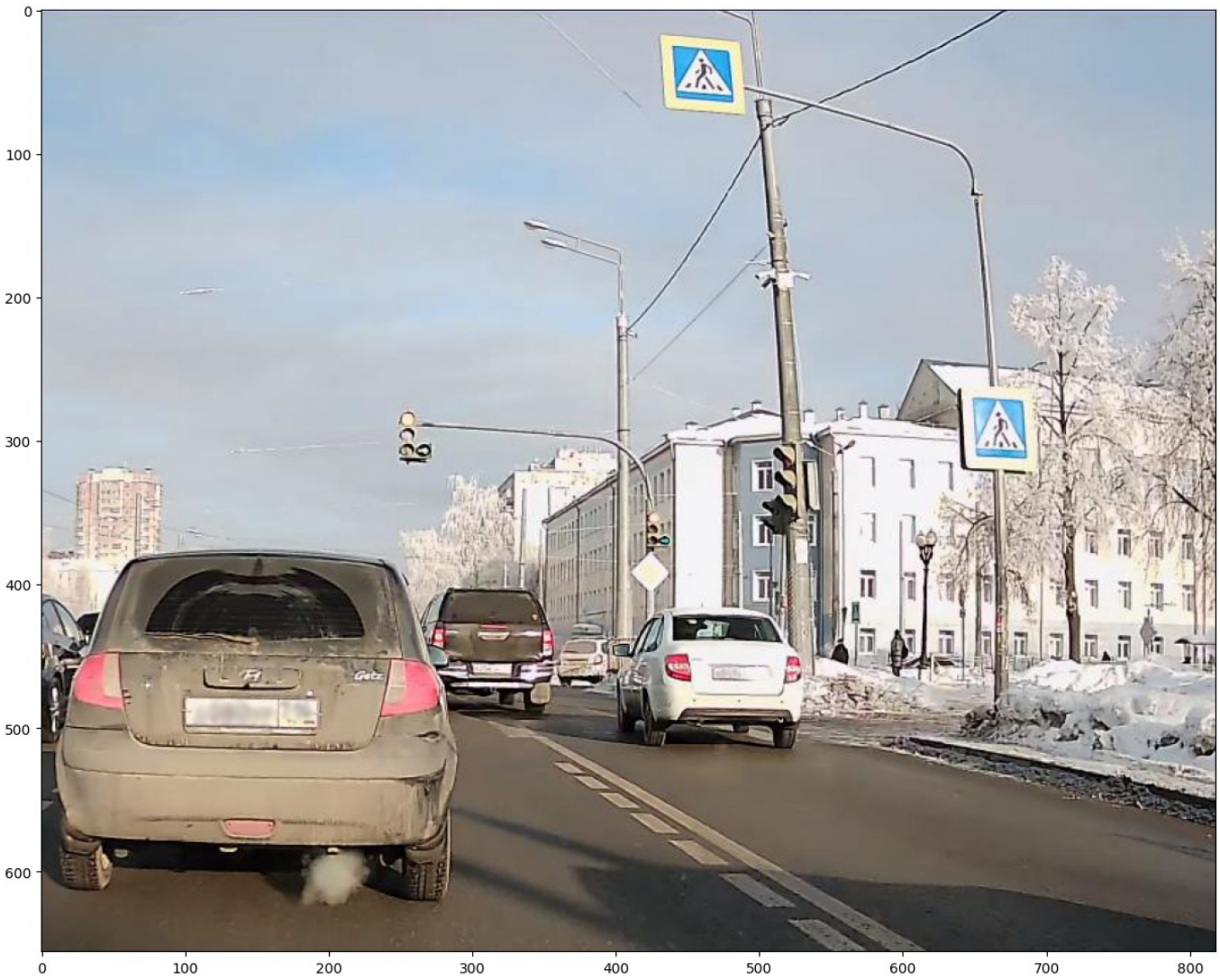
Out[17]:
```
array([[[120, 126, 176],
        [120, 126, 176],
        [120, 126, 176],
        ...,
        [122, 124, 165],
        [122, 124, 165],
        [122, 124, 165]],

       [[202, 209, 255],
        [202, 209, 255],
        [202, 209, 255],
        ...,
        [212, 214, 253],
        [212, 214, 253],
        [212, 214, 253]],

       [[166, 177, 209],
        [166, 177, 209],
        [166, 177, 209],
        ...,
        [184, 188, 217],
        [184, 188, 217],
        [184, 188, 217]],

       ...,

       [[ 71,  65,  53],
        [ 71,  65,  53],
        [ 71,  65,  53],
        ...,
        [102,  89,  73],
        [101,  87,  74],
        [100,  86,  73]],

       [[ 70,  64,  52],
        [ 70,  64,  52],
        [ 70,  64,  52],
        ...,
        [103,  90,  74],
        [102,  88,  75],
        [100,  86,  73]],

       [[ 70,  64,  52],
        [ 70,  64,  52],
        [ 70,  64,  52],
        ...,
        [103,  90,  74],
        [103,  89,  76],
        [100,  86,  73]]], dtype=uint8)
```

In [18]:
```
plt.figure(figsize=(15,15))
plt.imshow(image[:, 400:700]);
```

Важно знать: Если мы указываем только знак `:` в координатах, то мы хотим взять все элементы по этой координате. Если мы укажем индекс слева от знака `:`, но справа ничего не укажем, то мы берём элементы с указанного индекса до конечного. И также, если мы укажем индекс только справа от знака `:`, то мы берём срез от самого первого элемента по координате до указанного элемента.

In [19]:
```
plt.figure(figsize=(15,15))
plt.imshow(image[:400]);
```

In [20]:
```
plt.figure(figsize=(15,15))
plt.imshow(image[:, 400:]);
```

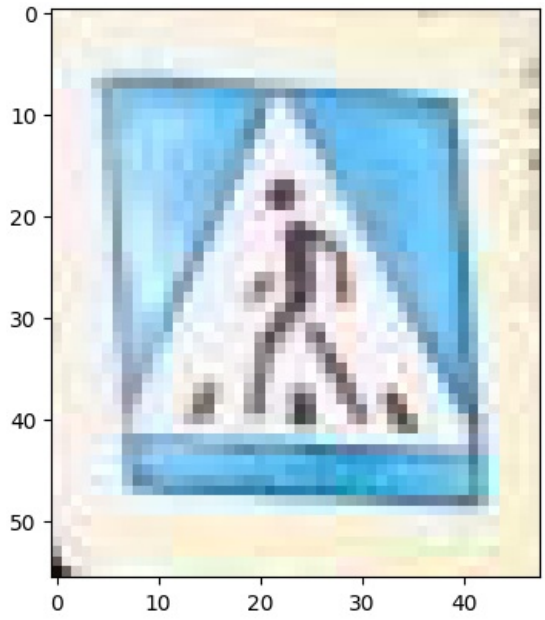


Если нам необходимо взять все строки и все столбцы конкретного канала, то мы указываем символ `:` для строк и для столбцов. Пример:

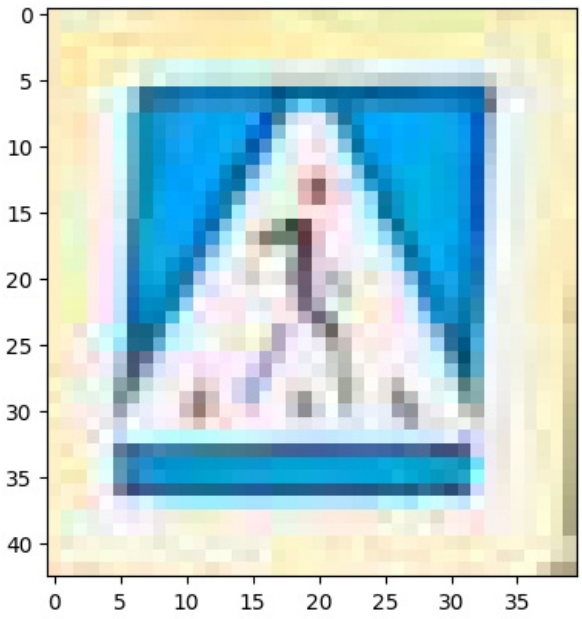
In [21]:
```
plt.figure(figsize=(15,15))
plt.imshow(image[:, :, 1]);
```

Вернёмся к созданию алгоритма распознавания, детекции и подсчёта дорожных знаков «Пешеходный переход». Для нашего алгоритма нам необходимы изображения самих дорожных знаков, которые мы будем использовать в качестве шаблонных изображений. С помощью срезов получим данные изображения из исходного изображения.

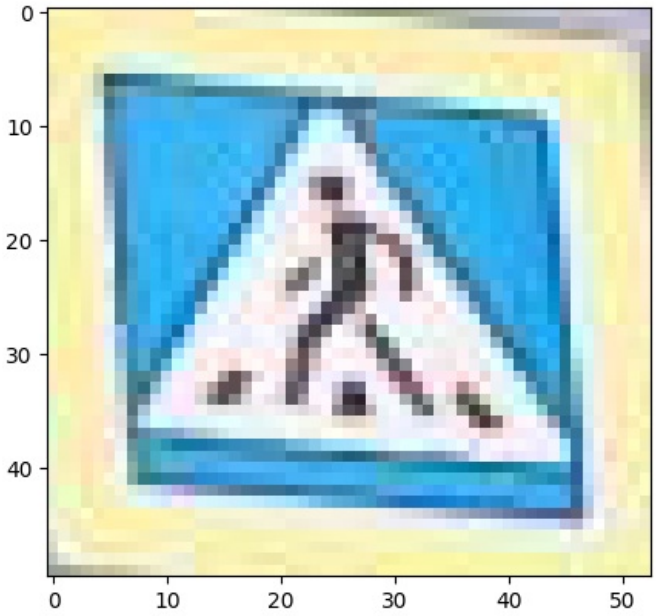
In [22]:
```
plt.imshow(image[264:320, 1044:1092]);
```



In [23]:
```
plt.imshow(image[360:403, 223:263]);
```



In [24]:
```
plt.imshow(image[20:70, 835:888]);
```

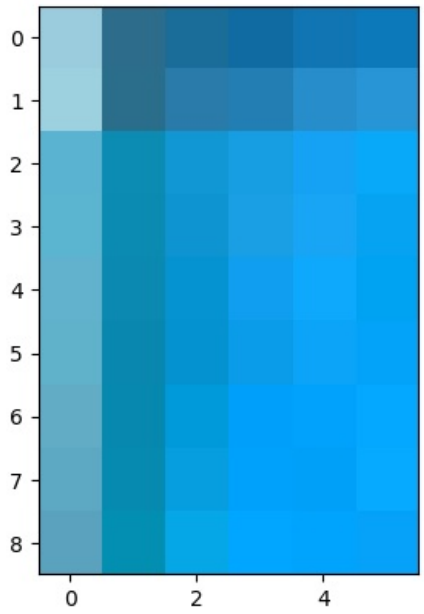


Теперь необходимо создать переменные, соответствующие шаблонным изображениям, но так как для нашего алгоритма размер шаблонных изображений должен быть одинаков, мы преобразуем полученные шаблоны в размер **45** x **45** пикселей (или **45** строк, **45** столбцов). Для этого будем использовать функцию библиотеки OpenCV `cv2.resize()`. Первым аргументом на функцию будем подавать необходимое изображение, вторым аргументом зададим его итоговый размер (dsize=(45,45)) и третьим аргументом укажем метод преобразования размера (interpolation=cv2.INTER_CUBIC). Таким образом создадим переменные: `template`, `template1` и `template2`.

In [25]:
```
template = cv2.resize(image[264:320, 1044:1092], dsize=(45, 45), interpolation=cv2.INTER_CUBIC)
template1 = cv2.resize(image[360:403, 223:263], dsize=(45,45), interpolation=cv2.INTER_CUBIC)
template2 = cv2.resize(image[20:70, 835:888], dsize=(45,45), interpolation=cv2.INTER_CUBIC)
```

Далее, используя особенности массивов `numpy`, мы отфильтруем изображение и выделим только те участки, которые имеют синий цвет как на рассматриваемых дорожных знаках. Цвет пикселя на изображении обусловлен его значением в каждом из трёх каналов. Значения пикселей находятся в диапазоне от 0 до 255. Для того чтобы понять какие значения нас интересуют, мы можем найти на изображении необходимые пиксели и посмотреть их значения для каждого канала. Рассмотрим следующую область изображения:

In [26]:
```
plt.imshow(image[366:375, 229:235]);
```



Затем рассмотрим значения пикселей этой области для первого канала:

In [27]:
```
image[366:375, 229:235,0]
```

Out[27]:
```
array([[155,  45,  26,  16,  17,  12],
       [157,  43,  42,  35,  39,  40],
       [ 90,  12,  17,  23,  21,   8],
       [ 91,  11,  14,  26,  24,   6],
       [ 98,  11,   5,  16,  14,   0],
       [ 96,   9,   4,  10,  12,   3],
       [ 98,   7,   0,   0,   0,   4],
       [ 93,   6,   6,   0,   0,   6],
       [ 91,   3,   5,   0,   0,   6]], dtype=uint8)
```

для второго канала:

In [28]:
```
image[366:375, 229:235,1]
```

Out[28]:
```
array([[204, 108, 109, 107, 117, 121],
       [209, 110, 123, 126, 141, 149],
       [179, 140, 151, 158, 162, 169],
       [181, 139, 148, 159, 165, 163],
       [178, 137, 147, 158, 169, 163],
       [178, 135, 146, 156, 164, 163],
       [172, 136, 153, 159, 162, 168],
       [169, 138, 158, 161, 160, 170],
       [162, 143, 167, 166, 164, 162]], dtype=uint8)
```

для третьего канала:

In [29]:
```
image[366:375, 229:235,2]
```

Out[29]:
```
array([[221, 139, 153, 162, 179, 186],
       [223, 139, 170, 179, 203, 214],
       [209, 179, 212, 226, 244, 249],
       [208, 178, 209, 228, 245, 242],
       [205, 177, 209, 240, 252, 241],
       [202, 175, 208, 233, 248, 249],
       [197, 175, 217, 250, 251, 255],
       [195, 176, 223, 251, 248, 255],
       [190, 178, 231, 255, 253, 249]], dtype=uint8)
```

Как видно в первом канале данные пиксели принимают значения от **0** до **164**, при этом большинство пикселей принимает значение от **0** до **100**. Во втором канале значение пикселей в большинстве своём находятся от **130** до **210**. В третьем канале примерно от **150** до максимального значения в **255**. Теперь зная диапазоны возможных значений рассматриваемых пикселей для каждого канала, мы можем задать условия для выделения таких пикселей. Условия задаются следующим образом:

1. Обращаемся к интересующей нас части массива. Например, ко всем строкам и столбцам первого канала;
2. Указываем необходимую операцию сравнения элементов массива с установленным пороговым значением.

Таким образом каждый элемент массива сравнивается с порогом, и там где условие истинно элемент выходного массива имеет значение `True` (1), а в случае не выполнения условия элемент выходного массива равен `False` (0).

In [30]:
```
image[:,:,0]<120
```

Out[30]:
```
array([[ True,  True,  True, ...,  True,  True,  True],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       ...,
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True]])
```

Отобразим в виде изображения полученный бинарный массив.

In [31]:
```
plt.figure(figsize=(15,15))
plt.imshow(image[:,:,0]<120, cmap='gray');
```

Таким образом белым цветом у нас выделены пиксели, соответствующие условию, что их значение для первого канала должно быть меньше чем **120**.

Для того чтобы сочетать несколько условий, необходимо использовать логические операции **И**, **ИЛИ**, **НЕ** (Соответствующие символы данных операций для массивов `numpy` это **&**, **|**, **~**). Например, мы хотим выделить те пиксели изображения, значения которых в первом канале меньше 120 **и** больше 90.

In [32]:
```
(image[:,:,0]<120) & (image[:,:,0]>90)
```

Out[32]:
```
array([[ True,  True,  True, ...,  True,  True,  True],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       ...,
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]])
```

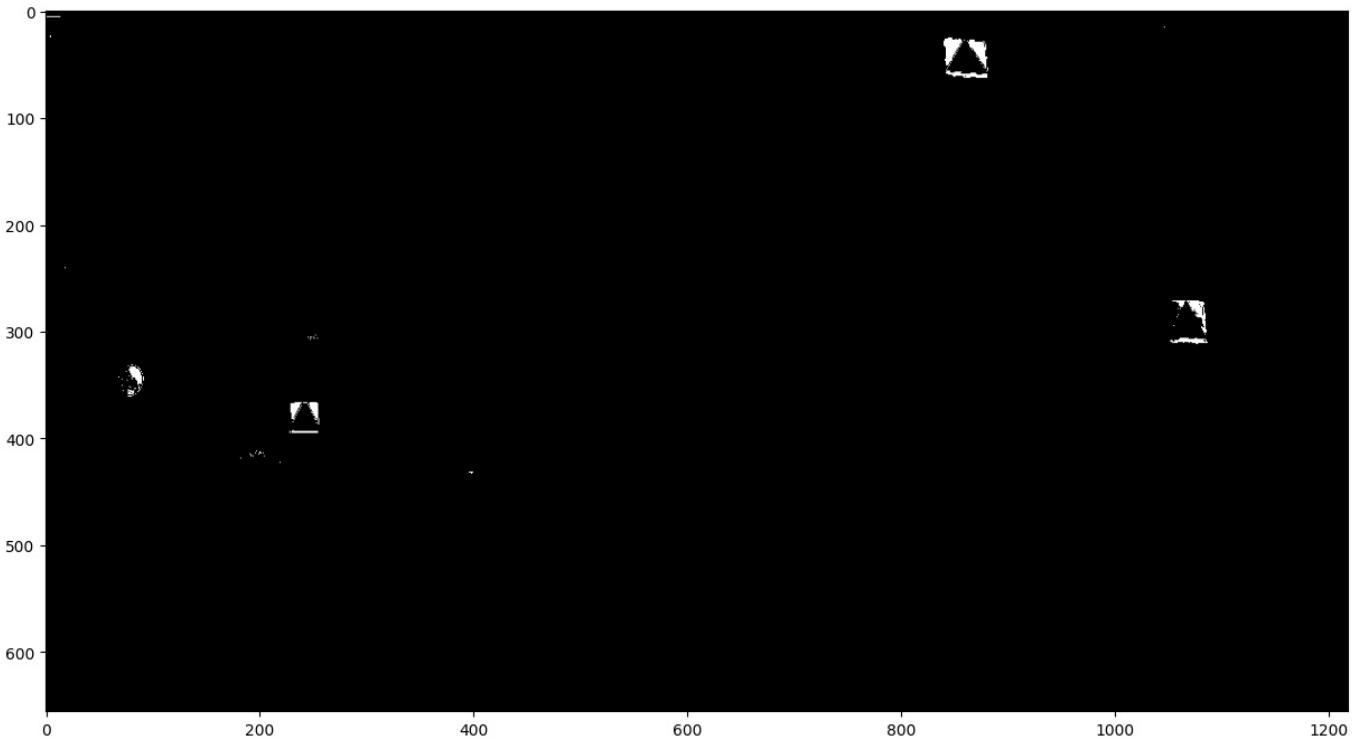
In [33]:
```
plt.figure(figsize=(15,15))
plt.imshow((image[:,:,0]<120) & (image[:,:,0]>90), cmap='gray');
```

Просмотрев все пиксели, интересующего нас синего цвета, и сформировав для всех трёх каналов диапазоны, в которые входит большинство этих пикселей, укажем необходимые условия для выделения и присвоим полученный бинарный массив переменной `test_mask`.

In [34]:
```
test_mask = (image[:,:,0]<120)&(image[:,:,1]>130)&(image[:,:,1]<220)&(image[:,:,2]>150)
```

Отобразим выделенные пиксели.

In [35]:
```
plt.figure(figsize=(15,15))
plt.imshow(test_mask, cmap='gray');
```



Таким образом мы выделили части интересующих нас дорожных знаков. Как видно не только дорожные знаки «Пешеходный переход» могут содержать такой синий цвет. Далее нам необходимо обработать выделенные области, чтобы они покрывали интересующие нас объекты, после чего из всех выделенных объектов распознать именно дорожные знаки «Пешеходный переход».

Для удаления шумов и восстановления размера объектов на бинарном изображении используются морфологические операции *сужение* (erosion) и *расширение* (dilation).

Сужение (erosion) — это процесс удаления белых пикселей с границ бинарного изображения. Для этого можно использовать функцию `cv2.erode()`, на вход которой подаются: обрабатываемое изображение, ядро фильтра (будем использовать

`kernel = np.ones((3,3),np.uint8`) и значение количества итераций (сколько раз мы применяем операцию сжатие над изображением)).

```
img = cv2.erode(image,kernel,iterations = 12)
```

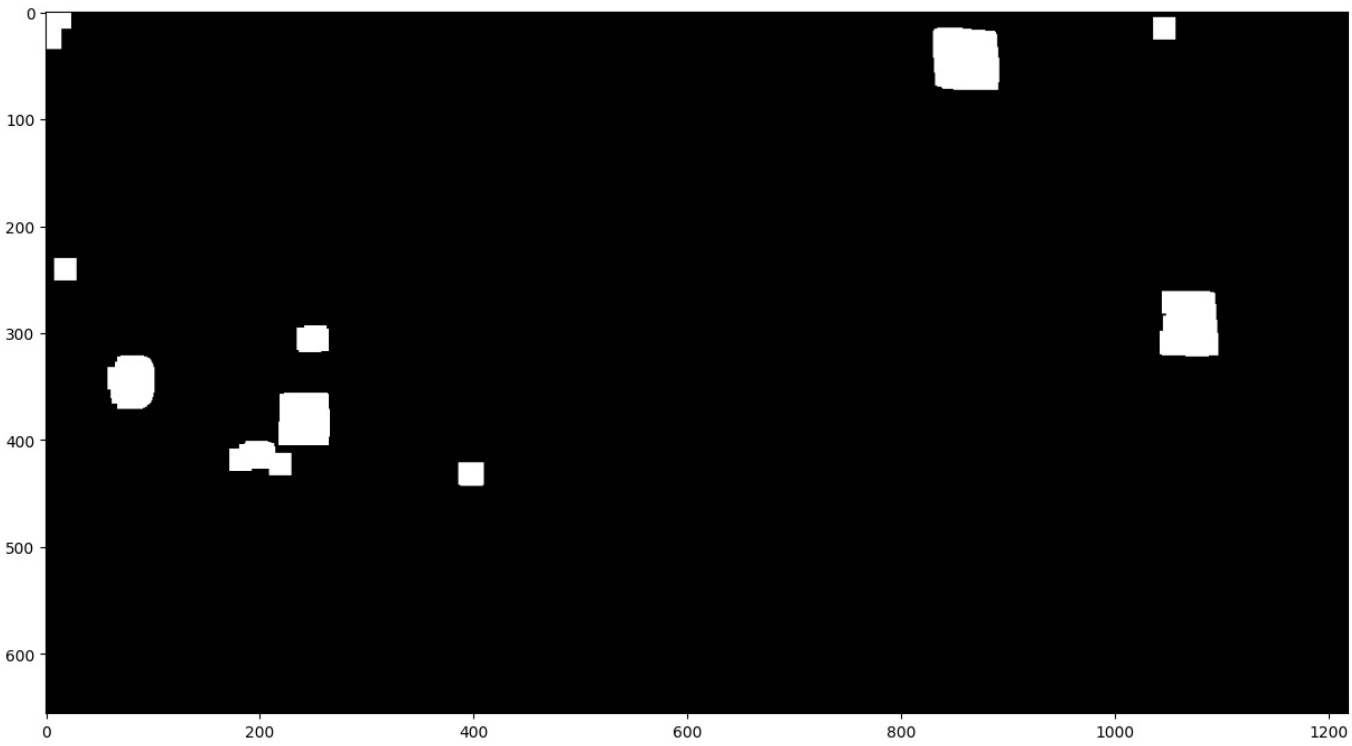
Расширение (dilation) – процесс, противоположный эрозии. Белые пиксели добавляются к границам бинарного изображения. Для этого можно использовать функцию `cv2.dilate()`, на вход которой подаются те же аргументы, что и на функцию сужения.

```
img = cv2.dilate(image,kernel,iterations = 10)
```

Благодаря тому, что преобразование возвращает изменённое изображение, можно создавать цепочки преобразований, то есть осуществлять последовательно операции сужения и расширения.

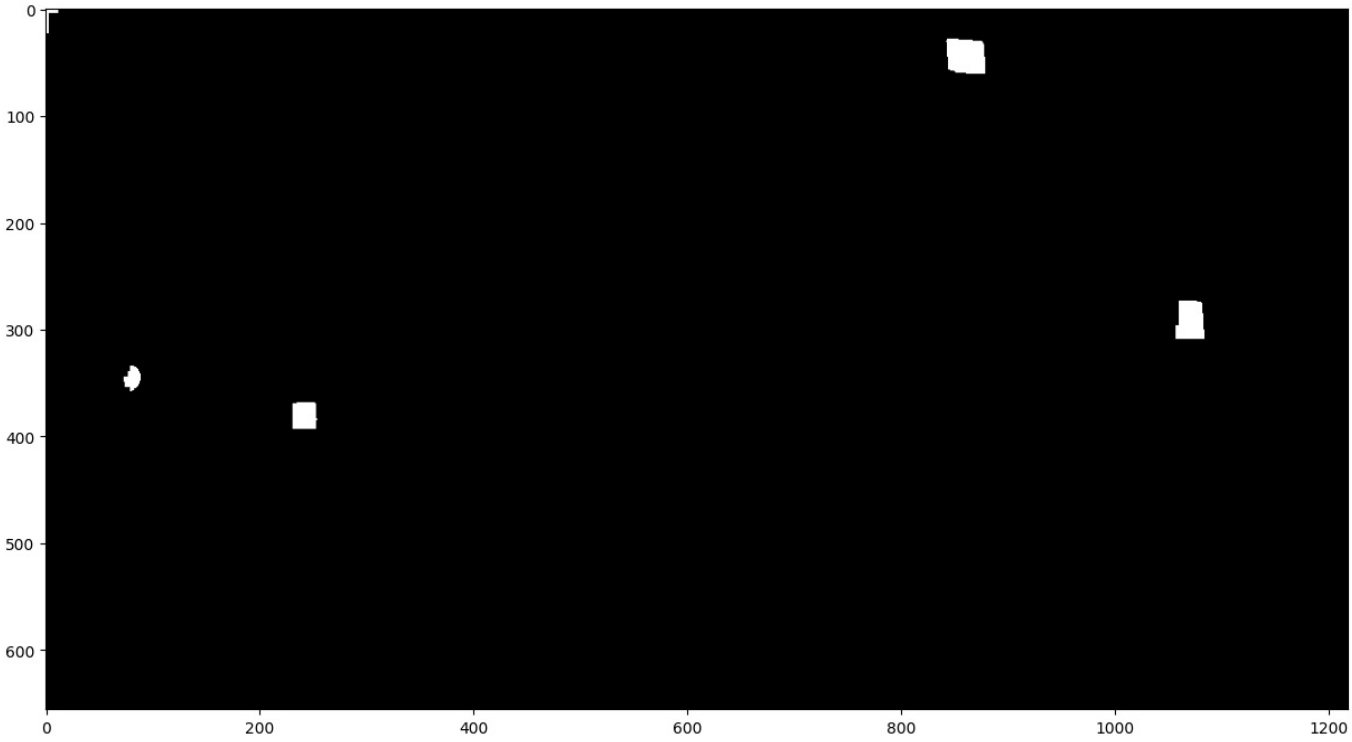
Для объединения выделенных частей дорожных знаков на бинарном изображении `test_mask` применим функцию расширения с 10 итерациями. На вход функции подадим `test_mask.astype(np.uint8)`, где `.astype(np.uint8)` необходим для того, чтобы перевести элементы массива в нужный тип данных.

In [36]:
```
kernel = np.ones((3,3),np.uint8) # формируем ядро фильтра
gray = cv2.dilate(test_mask.astype(np.uint8),kernel,iterations = 10) # применяем операцию расширения
plt.figure(figsize=(15,15)) # устанавливаем размер отображаемого рисунка
plt.imshow(gray, cmap='gray'); # выводим получившийся рисунок
```



После получения областей, которые соответствуют рассматриваемым дорожным знакам, для оптимизации алгоритма путём сокращения числа рассматриваемых объектов интереса, мы удалим с изображения все области недостаточного размера. Такие области являются шумом и не могут соответствовать дорожным знакам. Для этого над полученным изображением применим операцию сужения с 12 итерациями.

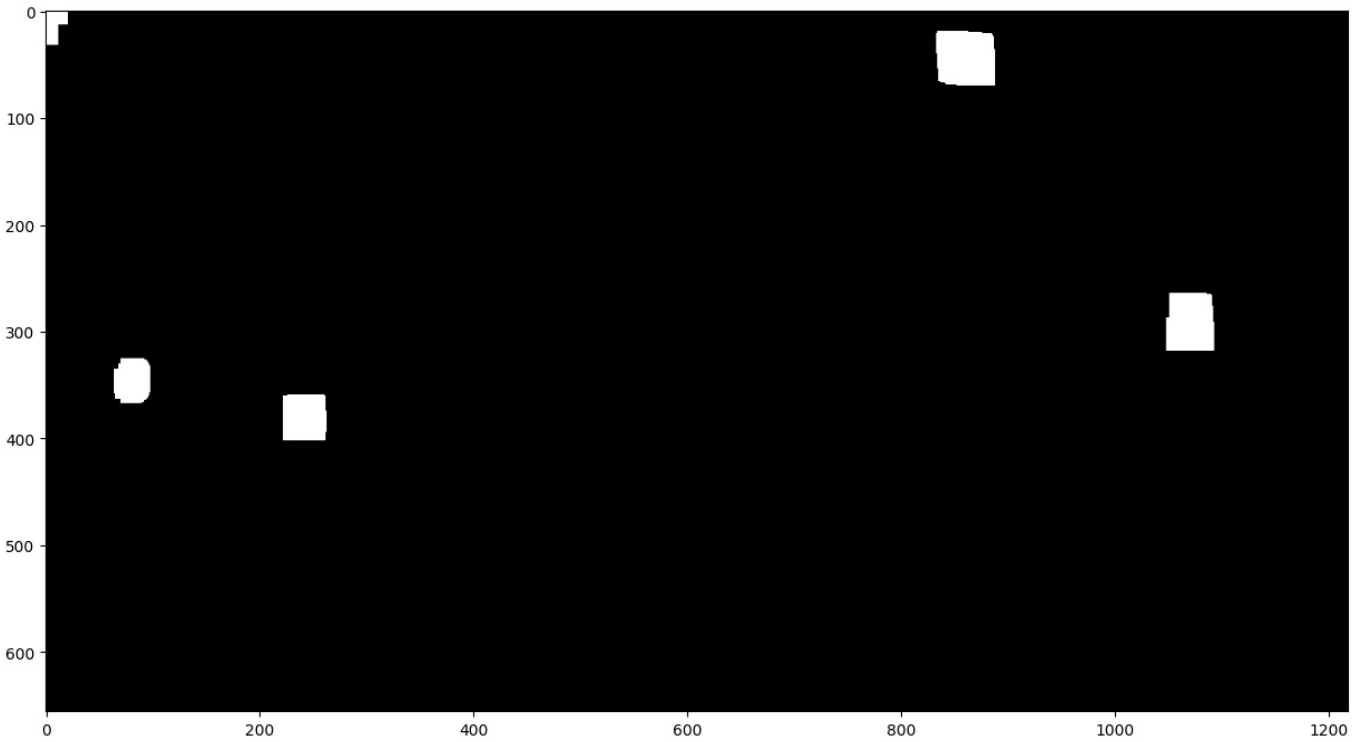
In [37]:
```
gray = cv2.erode(gray,kernel,iterations = 12)
plt.figure(figsize=(15,15))
plt.imshow(gray, cmap='gray');
```



Теперь на изображении осталось меньшее количество объектов, которые могут являться дорожными знаками. Но так как сужение уменьшило размер областей, нам необходимо снова их расширить, чтобы области соответствовали размерам объектов интереса на исходном изображении. Изображение `gray`, полученное после операции сужение, подадим на функцию расширения с 9 итерациями.

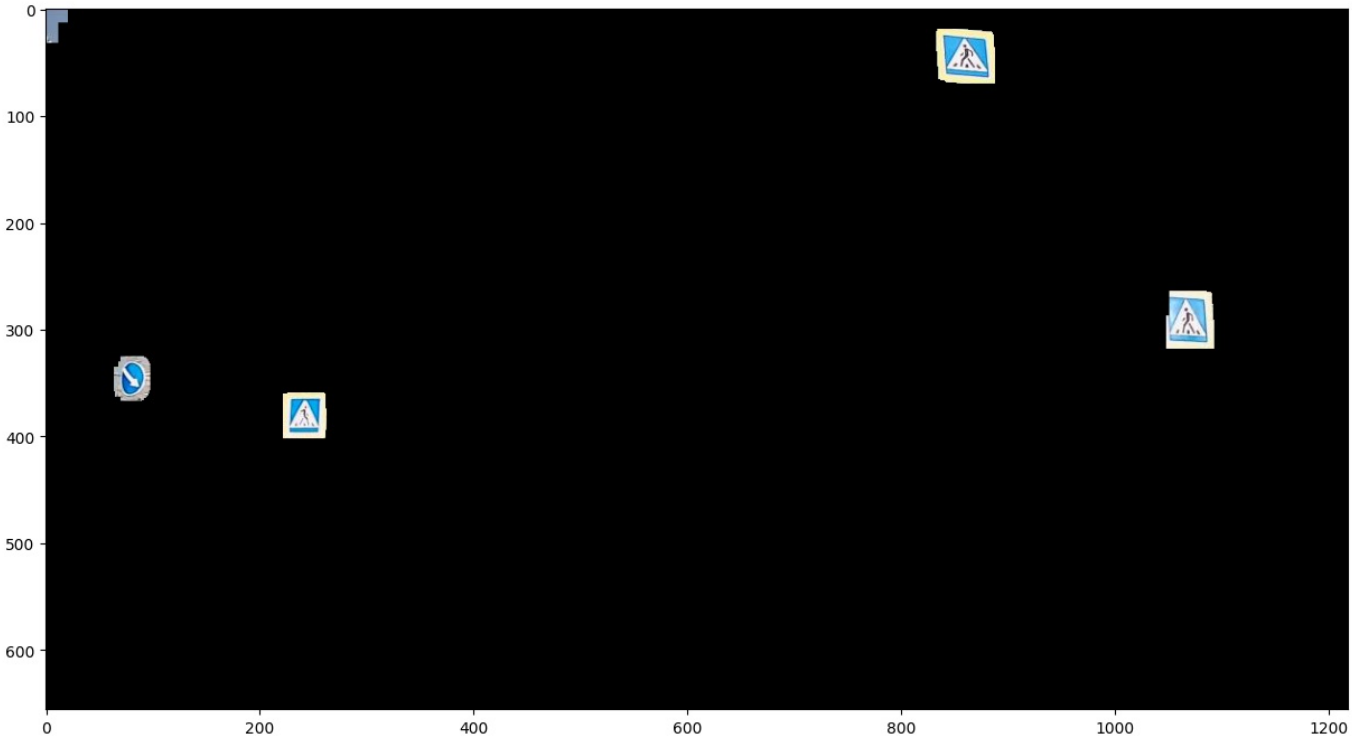
> **Важно:** Количество итераций применения морфологических операций, подбирается разработчиком алгоритма в зависимости от реализуемой им задачи. Для ваших алгоритмов эти параметры вы подбираете самостоятельно.

In [38]:
```
gray = cv2.dilate(gray,kernel,iterations = 9)
plt.figure(figsize=(15,15))
plt.imshow(gray, cmap='gray');
```



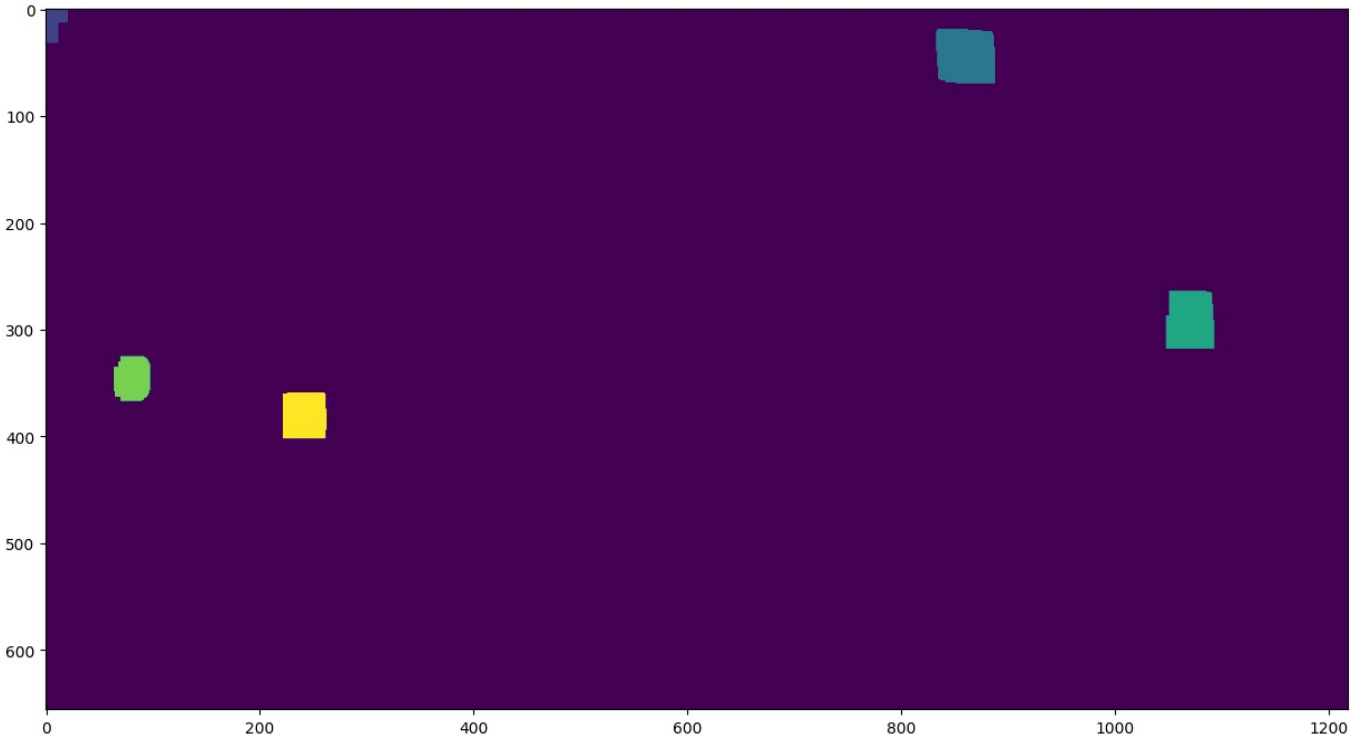
Проверим, соответствуют ли полученные области размерам объектов на исходном изображении. Отобразим (выделим) только те части исходного изображения, которые пересекаются с полученными областями.

In [39]:
```
image_test = image.copy()
image_test[~gray.astype(bool)] = 0
plt.figure(figsize=(15,15))
plt.imshow(image_test);
```



Удостоверившись в корректности результатов, вычислим пространственную информацию о полученных областях с помощью функции `label()`, на которую подадим бинарное изображение `gray`.

In [40]:
```
label_im = label(gray)
plt.figure(figsize=(15,15))
plt.imshow(label_im);
```



В полученном массиве `label_im` элементам каждой связной области присвоены значения соответствующие номеру области, которой они принадлежат. Например, все элементы области, помеченной жёлтым цветом на изображении выше, представляют собой значение 1, элементы зелёной области - значение 2, синей - 3, и так далее. Далее необходимо использовать функцию `regionprops()`, которая формирует описание каждой связной области. На вход функции подаётся результат, полученный с помощью функции `label()`. После чего можно получить доступ к свойствам связных областей через следующие атрибуты (ключи):

`area` - Количество пикселей принадлежащих области.

`bbox` - Ограничительная рамка (min_row, min_col, max_row, max_col) . Пиксели, принадлежащие ограничивающей рамке, находятся в полуоткрытом интервале [min_row; max_row) и [min_col; max_col). (row - строка, col - столбец, представляют собой координаты пикселя в массиве).

`bbox_area` - Количество пикселей принадлежащих ограничивающей рамке.

`centroid` - Кортеж координат центра области (row, col).

`local_centroid` - Кортеж координат центра тяжести (row, col) относительно ограничивающей рамки области.

`equivalent_diameter` - Диаметр круга с той же площадью, что и у связной области.

`extent` - Отношение количества пикселей в связной области к количеству пикселей в соответствующей данной области ограничивающей рамке. Вычисляется как area / (rows * cols).

`label` - Метка в помеченном входном изображении.

`filled_image` - Бинарное изображение такого же размера, что и ограничивающая рамка, которое содержит связную область с заполненными отверстиями.

`image` -Бинарное изображение такого же размера, что и ограничивающая рамка, которое содержит связную область.

In [41]:
```
regions = regionprops(label_im)
```

`regions` представляет собой список с информацией о каждой области бинарного изображения. Ниже представлены примеры получения через атрибуты информации о свойствах одной из областей под индексом **1**.

In [42]:
```
regions[1].area
```

Out[42]:
```
2669.0
```

In [43]:
```
regions[1].bbox
```

Out[43]:
```
(20, 833, 71, 888)
```

In [44]:
```
regions[1].bbox[0]
```

Out[44]:
```
20
```

In [45]:
```
regions[1].bbox[1]
```

Out[45]:
```
833
```

In [46]:
```
regions[1].bbox[2]
```

Out[46]:
```
71
```

In [47]:
```
regions[1].bbox[3]
```

Out[47]:
```
888
```

In [48]:
```
regions[1].bbox_area
```

Out[48]:
```
2805.0
```

In [49]:
```
regions[1].centroid
```

Out[49]:
```
(44.97040089921319, 860.2869988759835)
```

In [50]:
```
regions[1].local_centroid
```

Out[50]:
```
array([24.9704009 , 27.28699888])
```

In [51]:
```
regions[1].equivalent_diameter
```

Out[51]:
```
58.29473685418049
```

In [52]:
```
regions[1].extent
```

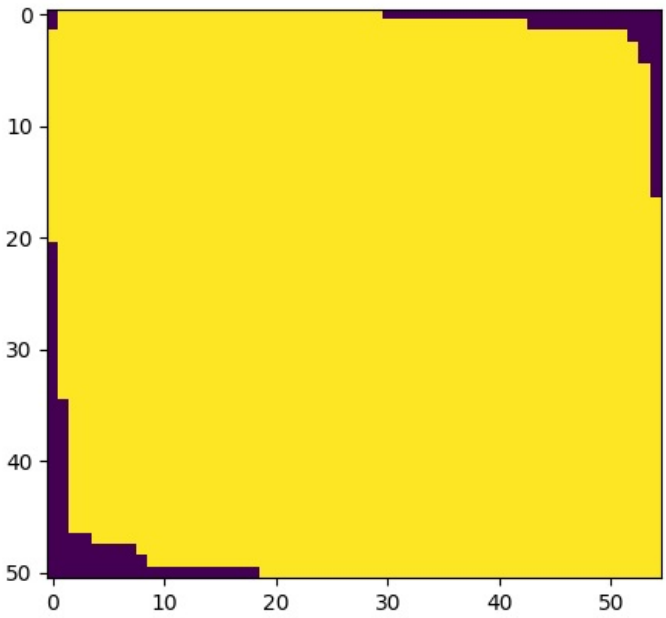
Out[52]:
```
0.9515151515151515
```

In [53]:
```
regions[1].label
```

Out[53]:
```
2
```

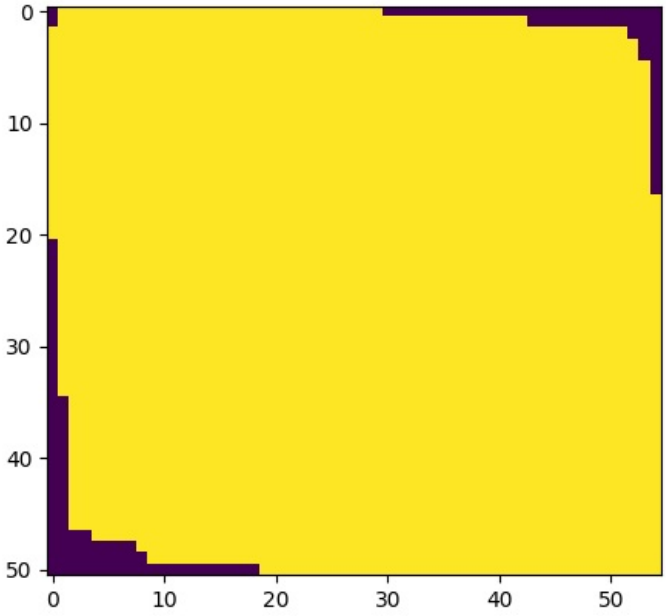
In [54]:
```
plt.imshow(regions[1].filled_image)
```

Out[54]:
```
<matplotlib.image.AxesImage at 0x7f570c689bb0>
```



In [55]:
```
plt.imshow(regions[1].image)
```

Out[55]:
```
<matplotlib.image.AxesImage at 0x7f57106d30b0>
```



После получения информации о всех рассматриваемых областях, желательно вновь провести их фильтрацию по размеру, для того чтобы отбросить области слишком мелких или крупных размеров и не рассматривать их далее. Для этого создадим пустой список `bbox`, затем в цикле "пройдёмся" по всем объектам из списка `regions`, посмотрим на количество пикселей, принадлежащих области, и если это количество находится в нужном нам диапазоне (который мы сами сформировали и установили), то информацию о координатах ограничивающей рамки этого объекта мы заносим в список `bbox`.

In [56]:
```
bbox = [] # создаём пустой список

for x in regions: # реализуем цикл по списку regions
    area = x.area # присваиваем переменной area значение количества пикселей принадлижащих
                  # рассматриваемой на данной итерации цикла области

    if (area>100) and (area<100000): # формируем условие, что area должна быть больше 100 И меньше 100000
        # если условие выполняется то добавляем координатах ограничивающей рамки в список bbox
        bbox.append(x.bbox)
```

Теперь когда мы подготовили области, которые могут содержать дорожные знаки «Пешеходный переход», нам нужно рассмотреть каждый объект исходного изображения, находящийся в выделенных областях. Затем вычислить характерные черты этого объекта и сравнить их с характерными чертами объектов, содержащихся в наших шаблонах `template`, `template1` и `template2`. Для получения характеристик изображения или части изображения можно воспользоваться функциями `feature.hog()` и `feature.local_binary_pattern()`, которые реализуют такие методы извлечения

признаков изображения, как **гистограмма направленных градиентов** и **локальные бинарные шаблоны** соответственно. С помощью данных функций мы сможем получить вектора признаков для объектов интереса.

In [57]:
```
def hog_finder(img):
    #вычисление гистограммы направленных градиентов
    fd = feature.hog(img, orientations=18, pixels_per_cell=(5, 5), cells_per_block=(4, 4), channel_axis=2)
    return fd

def lbp_finder(img):
    # вычисление локальных бинарных шаблонов
    lbp = feature.local_binary_pattern(img[:,:,0], 9, 1, method="uniform")
    # формирование гистограммы, для применения как вектора признаков
    (hist, _) = np.histogram(lbp.ravel(), bins=np.arange(0, 9 + 3), range=(0, 9 + 2))
    # нормализация гистограммы
    hist = hist.astype("float")
    hist /= (hist.sum() + 1e-7)
    return hist
```

Полезные ссылки:

1. Подробнее о Гистограммах направленных градиентов.

https://waksoft.susu.ru/2021/11/01/histogram-of-oriented-gradients/

https://scikit-image.org/docs/stable/api/skimage.feature.html#skimage.feature.hog

2. Подробнее о Локальных бинарных шаблонах.

https://habr.com/ru/articles/193658/

https://scikit-image.org/docs/stable/api/skimage.feature.html#skimage.feature.local_binary_pattern

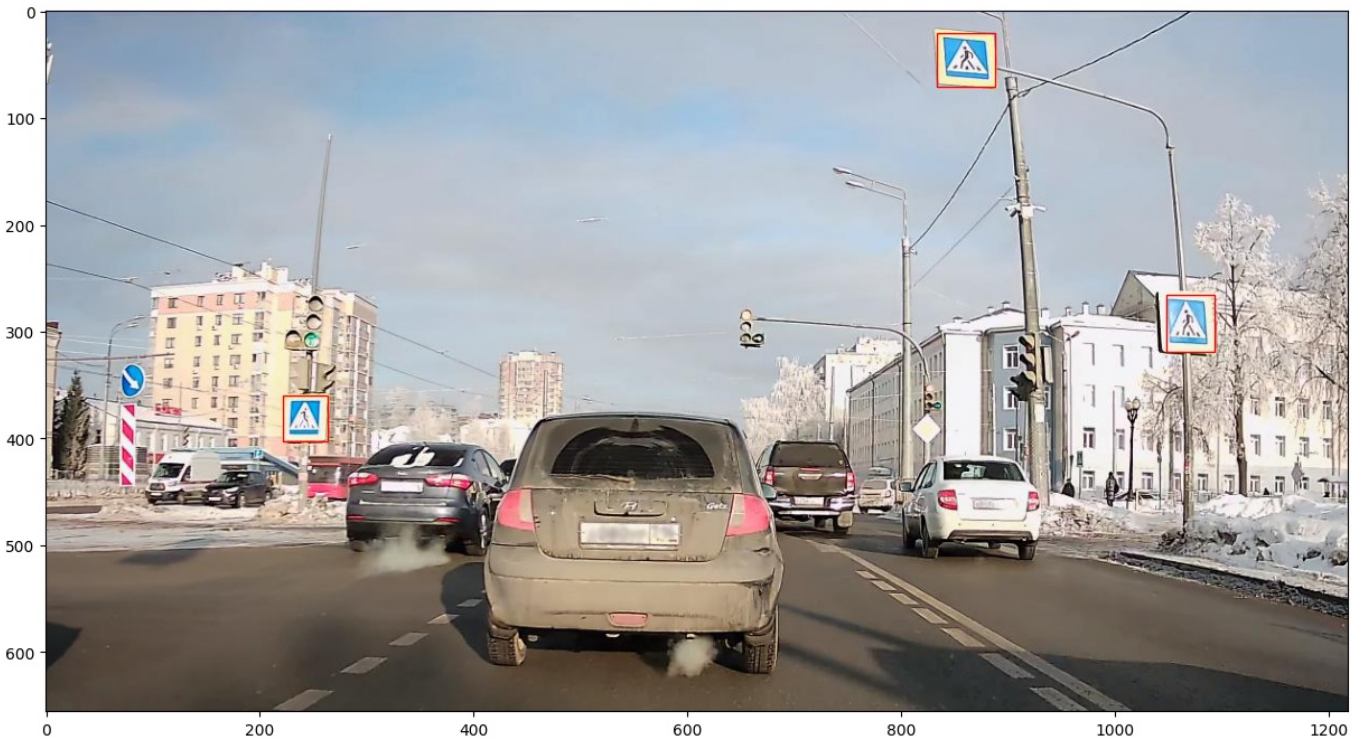
Для каждого шаблонного изображения получим 2 вектора признаков, с помощью наших функций `hog_finder` и `lbp_finder`, после чего соединим их в один вектор признаков с помощью функции `np.hstack()`.

In [58]:
```
# Формирование вектора признаков для первого шаблона
vect = hog_finder(template)
histlbp = lbp_finder(template)
vect1 = np.hstack((histlbp,vect))
# Формирование вектора признаков для второго шаблона
vect = hog_finder(template1)
histlbp = lbp_finder(template1)
vect2 = np.hstack((histlbp,vect))
# Формирование вектора признаков для третьего шаблона
vect = hog_finder(template2)
histlbp = lbp_finder(template2)
vect3 = np.hstack((histlbp,vect))
```

На последнем этапе работы нашего алгоритма мы в цикле проходим по координатам объектов из списка `bbox`. Из исходного изображения вырезаем изображение объекта по этим координатам и присваиваем его переменной `object`. Так как объекты на изображении могут быть различных размеров, для того чтобы сравниваемые вектора признаков имели одинаковую длину, изменим размер изображения `object` на 45 x 45 пикселей. Далее вычисляем два вектора признаков для полученного `object`, объединяем их в один вектор признаков и находим *Евклидово расстояние* между полученным вектором признаков и векторами признаков каждого шаблона. Складываем полученные расстояния и если их сумма меньше установленного нами порога, мы по координатам ограничивающей рамки из списка `bbox` выделяем нужный нам объект, и прибавляем +1 в счётчик найденных объектов.

In [59]:
```
fig, ax = plt.subplots(figsize=(15,15)) # задаём размер изображения
ax.imshow(image) # отображаем исходное изображение
count = 0 # создаём переменную для подсчёта количества распознанных объектов
for box in bbox: # реализуем цикл для рассмотрения координат областей из списка bbox
    object = image[box[0]:box[2],box[1]:box[3]] # вырезаем изображение объекта по этим координатам

    # изменяем размер изображения объекта
    object = cv2.resize(object, dsize=(45,45), interpolation=cv2.INTER_CUBIC)

    hog = hog_finder(object) # вычисляем вектор признаков методом гистограммы направленных градиентов
    histlbp = lbp_finder(object) # вычисляем вектор признаков методом локальных бинарных шаблонов
    vect = np.hstack((histlbp,hog)) # объединяем вектора признаков в один вектор признаков
    # Складываем значения расстояний между шаблонными векторами признаков и вектором признаков vect
    result = np.sqrt(np.square(vect - vect1).sum())
    result = result + np.sqrt(np.square(vect - vect2).sum())
    result = result + np.sqrt(np.square(vect - vect3).sum())


    if result < 18.8: # если сумма растояний меньше заданного порога,
        # используем координаты текущей итерации цикла для отображения красных рамок на исходном изображении
        rect = patches.Rectangle((box[1], box[0]), box[3]-box[1], box[2]-box[0],
```

```
                                linewidth=1, edgecolor='r', facecolor='none')

        # также увеличиваем на 1 значение соответствующее количеству распознаных знаков «Пешеходный переход»
        count = count + 1

        ax.add_patch(rect) # добавляем на исходное изображение красную рамку, сформированную в переменной rect

plt.show() # отображаем итоговое изображение с красными рамками окружающими найденные необходимые объекты
# отображаем количество найденных необходимых объектов
print('Количество дорожных знаков «Пешеходный переход» =',count)
```




```
Количество дорожных знаков «Пешеходный переход» = 3
```

В завершении оформим наш алгоритм в виде функции, на вход которой будет поступать изображение, а на выходе получим количество найденных объектов и изображение с красными рамками, окружающими эти объекты.

Так как наша программа должна использовать шаблонные вектора признаков, сформированные нами, необходимо сохранить наши шаблоны в виде файлов, к которым будет обращаться наша программа. Для сохранения в файл используем функцию `np.save()`, на вход которой подаётся название файла (можно указать путь к файлу вместе с его названием) и переменная, которую нужно сохранить.

In [60]:
```
np.save('vect1.npy', vect1)
np.save('vect2.npy', vect2)
np.save('vect3.npy', vect3)
```

Для загрузки необходимых файлов используется функция `np.load()`, в которой указываем название файла, если он находится в одной папке вместе с исполняемым файлом программы, или указываем полный путь к файлу.

In [61]:
```
vect1 = np.load('vect1.npy')
vect2 = np.load('vect2.npy')
vect3 = np.load('vect3.npy')
```

Создадим функцию `finder`, реализующую наш алгоритм.

In [62]:
```
def finder(image):
    test_mask = (image[:,:,0]<120)&(image[:,:,1]>130)&(image[:,:,1]<220)&(image[:,:,2]>150)
    kernel = np.ones((3,3),np.uint8)
    gray = cv2.dilate(test_mask.astype(np.uint8),kernel,iterations = 10)
    gray = cv2.erode(gray,kernel,iterations = 12)
    gray = cv2.dilate(gray,kernel,iterations = 9)
    label_im = label(gray)
    regions = regionprops(label_im)
    bbox = []

    for x in regions:
        area = x.area
        if (area>100) and (area<100000):
            bbox.append(x.bbox)

    fig, ax = plt.subplots(figsize=(15,15))
    ax.imshow(image)
    count = 0
```

```
    for box in bbox:
        object = image[box[0]:box[2],box[1]:box[3]]
        object = cv2.resize(object, dsize=(45,45), interpolation=cv2.INTER_CUBIC)

        hog = hog_finder(object)
        histlbp = lbp_finder(object)
        vect = np.hstack((histlbp,hog))

        result = np.sqrt(np.square(vect - vect1).sum())
        result = result + np.sqrt(np.square(vect - vect2).sum())
        result = result + np.sqrt(np.square(vect - vect3).sum())


        if result < 18.8:
            rect = patches.Rectangle((box[1], box[0]), box[3]-box[1], box[2]-box[0],
                                     linewidth=1, edgecolor='r', facecolor='none')
            count = count + 1
            ax.add_patch(rect)

    plt.show()
    print('Количество дорожных знаков «Пешеходный переход» =',count)
    #добавляем необходимые строчки строчки, в которых формируем информацию, выдаваемую функцией
    # присваиваем переменной new_image полученное изображение с ограничивающими рамками
    new_image = np.array(fig.canvas.renderer.buffer_rgba())
    return count, new_image # после return перечисляем всё, что функция должна возвращать
```

Проверим нашу функцию на новом изображении:

In [63]:
```
image = plt.imread('image2.jpg')
count, new_image = finder(image)
```

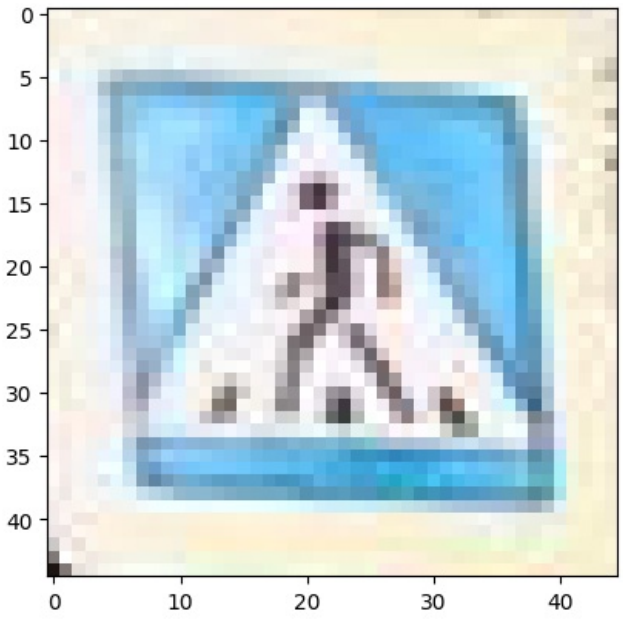

```
Количество дорожных знаков «Пешеходный переход» = 1
```

Теперь созданную функцию можно использовать для нахождения и подсчёта дорожных знаков «Пешеходный переход».

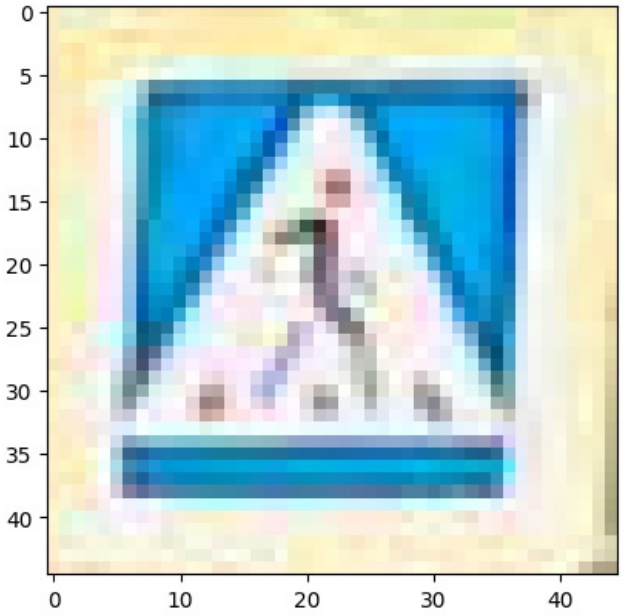
Для обнаружения и подсчёта необходимых объектов на изображении, можно использовать любые доступные алгоритмы, необязательно осуществлять поиск по шаблону, например, можно обучить модель методом опорных векторов и применять её для решения поставленной задачи. Давайте также разберём этот альтернативный вариант алгоритма обнаружения и подсчёта дорожных знаков «Пешеходный переход».

Для обучения модели нам нужна обучающая выборка из объектов, принадлежащих интересующему нас классу, и объектов, которые этому классу не принадлежат. Чем больше обучающая выборка тем лучше. Из нашего исходного изображения мы уже получили все объекты, принадлежащие классу дорожный знак «Пешеходный переход». Эти объекты у нас сохранены как шаблоны: `template`, `template1` и `template2`. Также для этих объектов у нас уже вычислены вектора признаков: `vect1`, `vect2` и `vect3`.

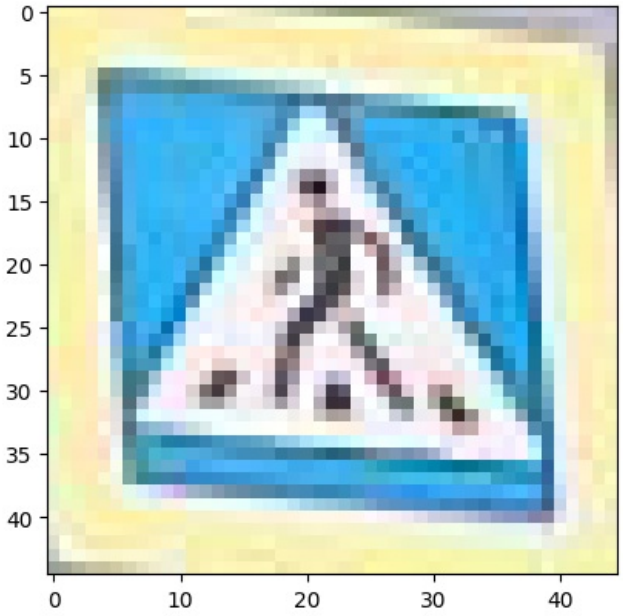
In [64]:
```
plt.imshow(template);
```



In [65]:
```
plt.imshow(template1);
```



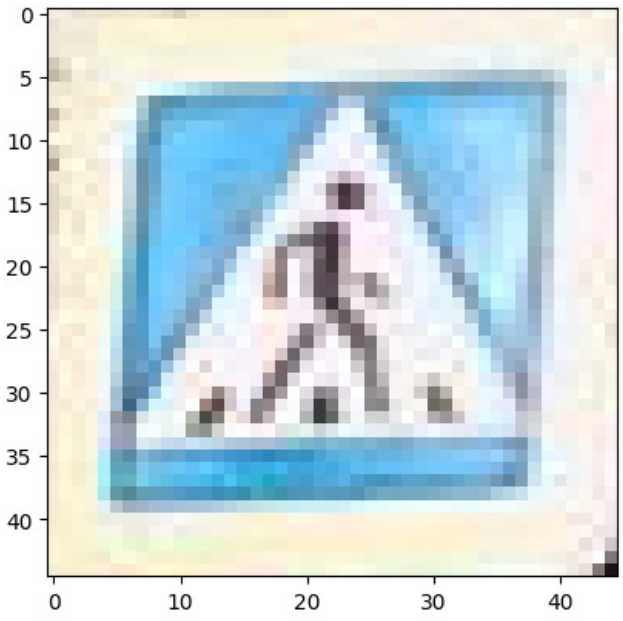
In [66]:
```
plt.imshow(template2);
```



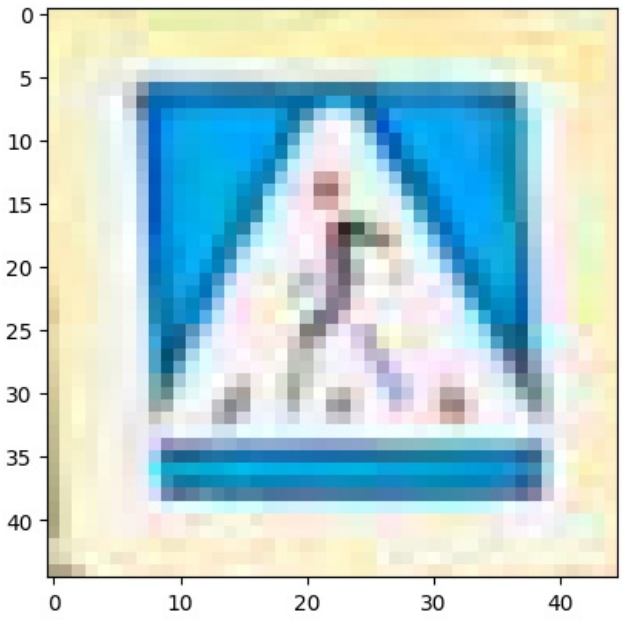
Чтобы увеличить обучающую выборку, мы также можем каждый наш шаблон отразить по горизонтали, так как знаки в дорожной сцене могут иметь различные направления. Для отражения будем использовать функцию `np.flip()` на вход которой подаём массив для его отражения, а также в аргументе `axis` номер оси, по которой это отражение будет осуществляться (0 - по

вертикали, 1 - по горизонтали, 2 - по каналам).

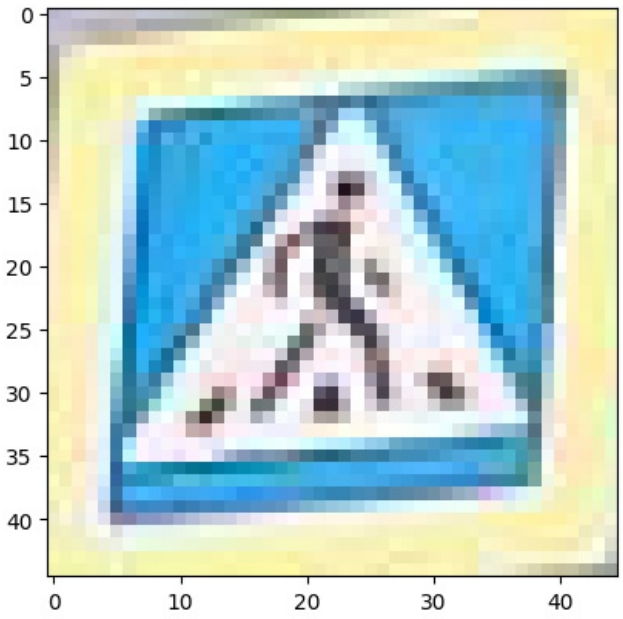
In [67]:
```
template3 = np.flip(template, axis=1)

plt.imshow(template3);
```



In [68]:
```
template4 = np.flip(template1, axis=1)

plt.imshow(template4);
```



In [69]:
```
template5 = np.flip(template2, axis=1)

plt.imshow(template5);
```



Для новых шаблонов вычислим вектора признаков.

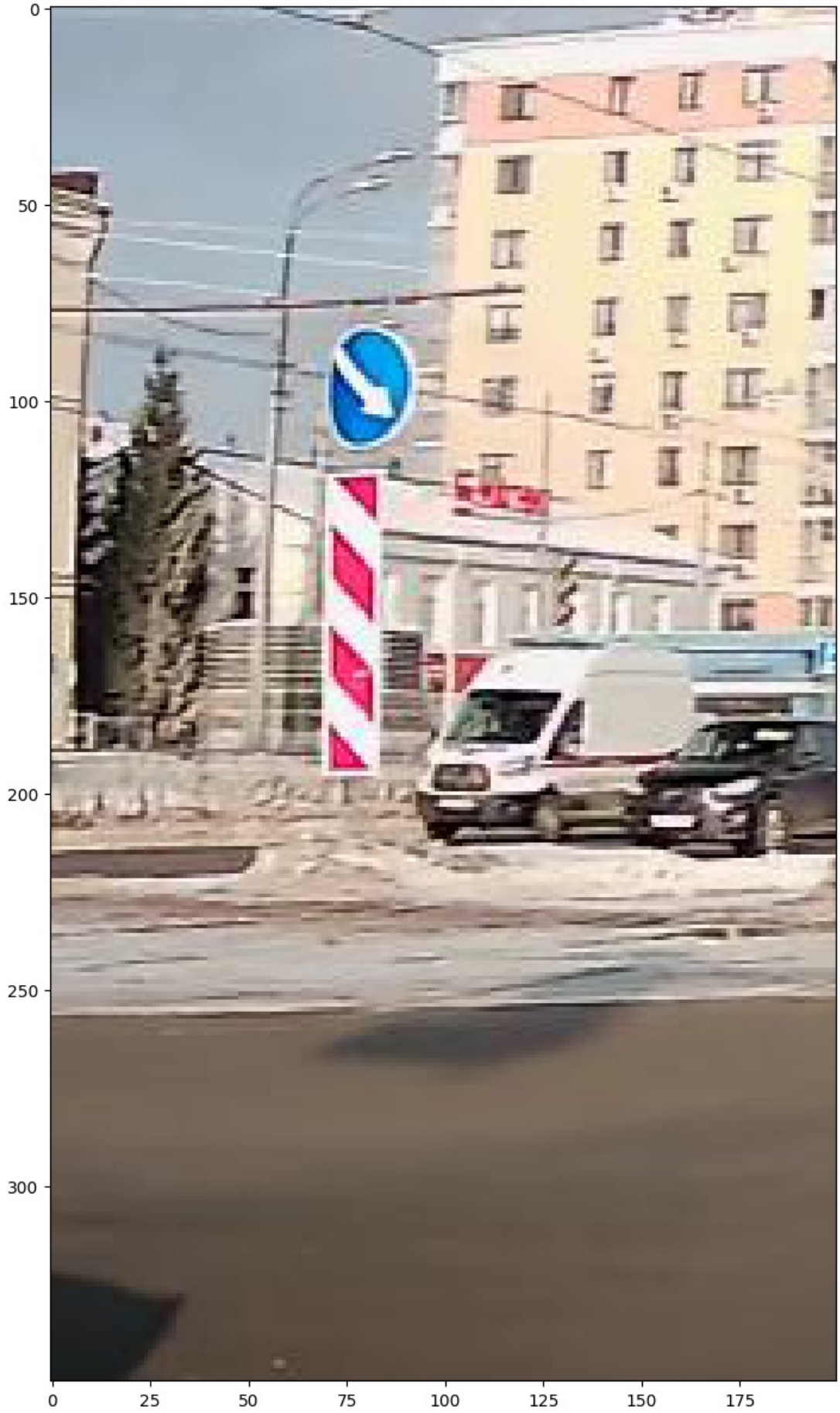
In [70]:
```
# Формирование вектора признаков для четвёртого шаблона
vect = hog_finder(template3)
histlbp = lbp_finder(template3)
vect4 = np.hstack((histlbp,vect))
# Формирование вектора признаков для пятого шаблона
vect = hog_finder(template4)
histlbp = lbp_finder(template4)
vect5 = np.hstack((histlbp,vect))
# Формирование вектора признаков для шестого шаблона
vect = hog_finder(template5)
histlbp = lbp_finder(template5)
vect6 = np.hstack((histlbp,vect))
```

Теперь нам необходимо получить вектора признаков для изображений, не принадлежащих интересующему нас классу. Мы можем из исходного изображения взять область без дорожных знаков и "нарезать" её на маленькие изображения размером **45** x **45** пикселей (так как `изображения/вектора признаков` в обучающей выборке должны быть одинакового размера).

> **Важно знать:** В обучающей выборке в изображениях, не принадлежащих вашему классу, желательно присутствие объектов, которые похожи на ваш класс, для того чтобы модель научилась правильно отличать такие похожие объекты.

In [71]:
```
image = plt.imread('image.jpg')

plt.figure(figsize=(15,15))
plt.imshow(image);
```

In [72]:
```
train_img = image[250:600, :200] # вырезаем область без необходимых нам дорожных знаков

plt.figure(figsize=(15,15))
plt.imshow(train_img);
```



Изображения, не содержащие объект необходимого класса, можно нарезать вручную, но мы сделаем это в цикле. Определим размер изображения `train_img`. Затем "пройдёмся" в цикле по строкам и столбцам этого изображения с шагом **45** (промежуток между рассматриваемыми пикселями составляет 44 пикселя). Для формирования циклов для строк и столбцов будем использовать функцию `range()`, которая задаёт необходимый диапазон.

Функция range() может принимать от одного до трёх целочисленных аргументов:

range(n) — возвращает диапазон целых чисел от 0 до n - 1. Например, range(4) вернёт диапазон целых чисел: 0, 1, 2, 3;

range(k, n) — возвращает диапазон целых чисел от k до n - 1. Например, range(1, 5) вернёт диапазон целых чисел: 1, 2, 3, 4;

range(k, n, s) — возвращает диапазон целых чисел от k до n - 1 с шагом s. Например, range(1, 10, 2) вернёт диапазон целых чисел: 1, 3, 5, 7, 9.

Для того, чтобы в цикле дойти от первой до последней строки и от первого до последнего столбца массива `train_img` необходимо знать количество строк и столбцов этого массива.

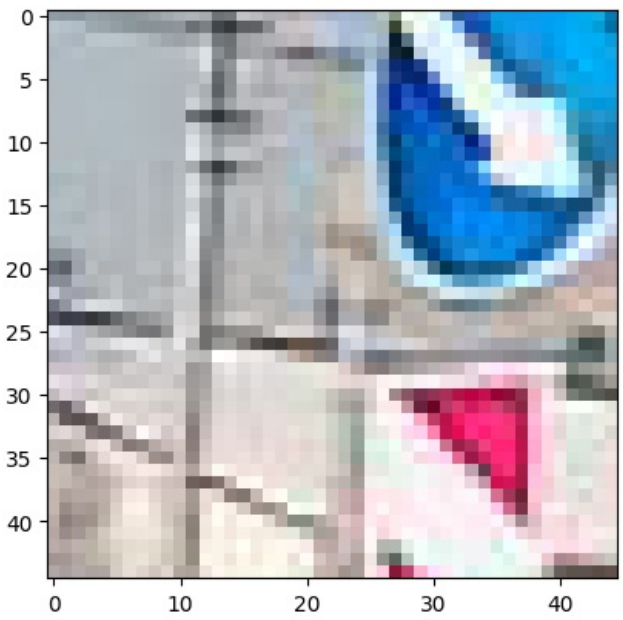
In [73]:
```
r, c, z = train_img.shape # высота (r), ширина (c) и количество каналов (z) изображения,
# другими словами,
# количество строк (r), количество столбцов (c) и количество каналов (z) массива,
```

Создадим два пустых списка: для внесения в них изображений и векторов признаков, соответствующих данным изображениям. Затем "пройдёмся" в цикле от `45` строки до `r` с шагом `45`, где будем рассматривать столбцы начиная с `45-ого` до `c` с шагом `45`. В итоге значения переменных `i` и `j` будут координатами рассматриваемых пикселей. Относительно этих координат мы будем брать срез размером **45** на **45** и добавлять это вырезанное изображение (массив) в соответствующий список. Также рассчитаем вектор признаков для этого среза и добавим его в список векторов признаков.

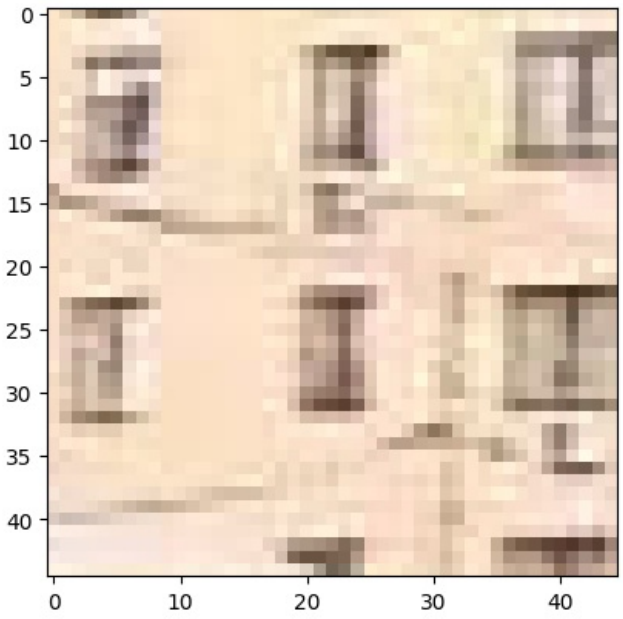
In [74]:
```
images = [] # создаём пустой список для изображений
vectors = [] # создаём пустой список для векторов признаков
for i in range(45, r, 45): # создаём цикл для перемещения по строкам
    for j in range(45, c, 45): # создаём цикл для перемещения по столбцам
        # берём срез необходимого размера относительно координат пикселя в текущей итерации
        cut_image = train_img[i-45:i,j-45:j]
        images.append(cut_image) # добавляем в список images массив cut_image
        vect = hog_finder(cut_image) # вычисляем вектор признаков методом гистограммы направленных градиентов
        histlbp = lbp_finder(cut_image) # вычисляем вектор признаков методом локальных бинарных шаблонов
        vect = np.hstack((histlbp,vect)) # соединяем два вектора признаков в один
        vectors.append(vect) # добавляем сформированный вектор признаков vect в список vectors
```

Давайте посмотрим, что у нас получилось. Возьмём и отобразим из списка `images` изображения под индексами: 9, 11, 18.

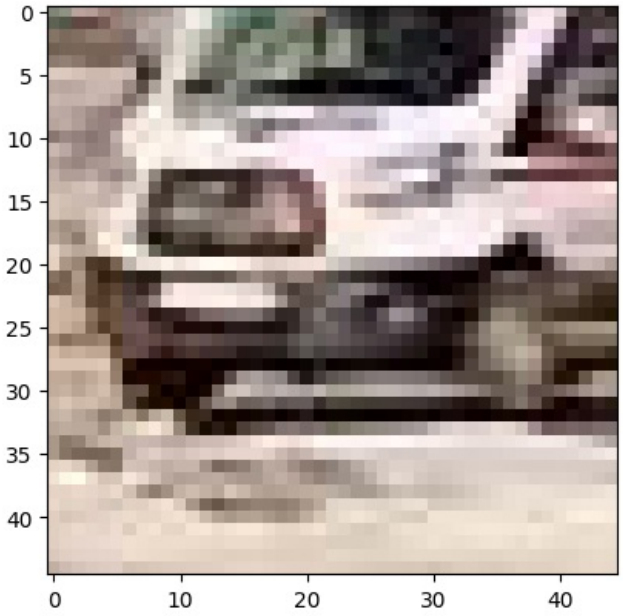
In [75]:
```
plt.imshow(images[9]);
```



In [76]:
```
plt.imshow(images[11]);
```



In [77]:
```
plt.imshow(images[18]);
```



Следовательно, под аналогичными индексами в списке `vectors` у нас хранятся соответствующие этим изображениям вектора признаков. Теперь сформируем обучающую выборку, которая состоит из списка признаков объектов (векторов признаков) и списка соответствующих этим объектам классов. Так как у нас только 2 класса, мы пронумеруем их как класс 1 - **дорожный знак «Пешеходный переход»** и класс 0 - **всё остальное**. Для начала сформируем список признаков объектов:

In [78]:
```
class_1 = [vect1, vect2, vect3, vect4, vect5, vect6] # создаём список из векторов признаков наших шаблонов
# совмещаем список признаков шаблонов с признаками изображений, не принадлежащих необходимому классу
x_train = vectors[9:19] + class_1
```

Когда мы складываем два списка, используя `+`, мы объединяем эти два списка в один (продолжаем первый список элементами второго). Таким образом мы взяли вектора признаков с девятого по 18 (включительно) индексы из списка `vectors` и дополнили их векторами признаков наших шаблонов. В результате в обучающей выборке мы получили признаки для десяти объектов, которые не принадлежат необходимому классу, и для шести объектов относящихся к этому классу.

> **Важно знать:** Для лучшей точности работы модели желательно стараться соблюдать баланс в обучающей выборке между объектами принадлежащими разным классам. Если у нас 6 объектов класса «дорожный знак «Пешеходный переход»», то объектов класса «всё остальное» должно быть примерно такое же количество.

Теперь сформируем соответствующий список объектов.

In [79]:
```
y_train = len(vectors[9:19])*[0]+len(class_1)*[1]

print(y_train)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
```

С помощью функции `len()` мы определяем длину подаваемого на неё списка, далее при умножении `n-ого` числа (в нашем

случае длины списка) на список, данный список повторяется `n-oe` количество раз. Пример:

In [80]:
```
print([1,2,3] * 3)
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Следовательно, для формирования списка классов объектов мы вносили в него столько нулей сколько элементов в списке `vectors[9:19]` (количество элементов = длина списка), и также внесли столько единиц сколько элементов в списке `class_1`.

Получив обучающую выборку, мы можем обучить модель методом опорных векторов. Данный процесс осуществляется с помощью специальных функций (классов) из библиотеки `sklearn`. По этому для начала необходимо осуществить импорт из этой библиотеки.

In [81]:
```
from sklearn.svm import SVC
```

Подробней о классе `SVC` библиотеки `sklearn` вы можете узнать по следующей ссылке:

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

In [82]:
```
svc_model = SVC(C=1.0) # создаём модель
svc_model.fit(x_train,y_train) # обучаем модель на сформированной обучающей выборке
```

Out[82]:
```
SVC()
```

Проверим верно ли предсказывает классы наша обученная модель. На самом деле правильней бы было проверять модель на тестовой выборке (на объектах которые не содержатся в обучающей выборке), но так как у нас малое количество изображений объектов необходимого нам класса (6 изображений) мы проверим нашу модель на обучающей выборке.

Для получения предсказания класса необходимо использовать метод `.predict()`. На вход данного метода подаётся вектор/вектора признаков объектов, класс которых мы хотим определить с помощью модели. Сравним предсказанные классы с действительными и с помощью метода `.score()` вычислим точность работы нашей модели на обучающей выборке.

In [83]:
```
# предсказываем классы объектов обрабатывая их признаки обученной моделью
predictions = svc_model.predict(x_train)

for i in range(len(y_train)): # проходи в цикле по ответам и сравниваем их с предсказаниями модели
    print(f'Объект №{i}, предсказание: {predictions[i]}, истина: {y_train[i]}')

score = svc_model.score(x_train, y_train) # вычисляем точность модели на тестовой выборке
print(f'Точность модели = {score*100}%')
```

```
Объект №0, предсказание: 0, истина: 0
Объект №1, предсказание: 0, истина: 0
Объект №2, предсказание: 0, истина: 0
Объект №3, предсказание: 0, истина: 0
Объект №4, предсказание: 0, истина: 0
Объект №5, предсказание: 0, истина: 0
Объект №6, предсказание: 0, истина: 0
Объект №7, предсказание: 0, истина: 0
Объект №8, предсказание: 0, истина: 0
Объект №9, предсказание: 0, истина: 0
Объект №10, предсказание: 1, истина: 1
Объект №11, предсказание: 1, истина: 1
Объект №12, предсказание: 1, истина: 1
Объект №13, предсказание: 1, истина: 1
Объект №14, предсказание: 1, истина: 1
Объект №15, предсказание: 1, истина: 1
Точность модели = 100.0%
```

После обучения эффективной модели, для того чтобы использовать её в программе, при этом не обучая каждый раз заново, сохраним модель в файл с помощью функции `pickle.dump()`, первый аргумент которой это переменная, представляющая собой обученную модель, а второй аргумент имеет вид `open('svc_model.pkl', 'wb')`, где `svc_model.pkl` это название файла в который мы сохраняем модель (если файл должен сохранятся в другой директории, то здесь необходимо указать полный путь).

In [84]:
```
import pickle
pickle.dump(svc_model, open('svc_model.pkl', 'wb'))
```

Загрузка файла модели осуществляется следующим образом:

In [85]:
```
svc_model = pickle.load(open('svc_model.pkl', 'rb'))
```

Теперь, имея обученную модель, изменим немного наш предыдущий алгоритм обнаружения и подсчёта объектов. Мы будем

распознавать объекты не сравнивая их с шаблоном, а применяя модель для предсказания класса на основе вектора признаков объекта.

In [86]:
```
def finder_v2(image):
    test_mask = (image[:,:,0]<120)&(image[:,:,1]>130)&(image[:,:,1]<220)&(image[:,:,2]>150)
    kernel = np.ones((3,3),np.uint8)
    gray = cv2.dilate(test_mask.astype(np.uint8),kernel,iterations = 10)
    gray = cv2.erode(gray,kernel,iterations = 12)
    gray = cv2.dilate(gray,kernel,iterations = 9)
    label_im = label(gray)
    regions = regionprops(label_im)
    bbox = []

    for x in regions:
        area = x.area
        if (area>100) and (area<100000):
            bbox.append(x.bbox)

    fig, ax = plt.subplots(figsize=(15,15))
    ax.imshow(image)
    count = 0
    for box in bbox:
        object = image[box[0]:box[2],box[1]:box[3]]
        object = cv2.resize(object, dsize=(45,45), interpolation=cv2.INTER_CUBIC)

        hog = hog_finder(object)
        histlbp = lbp_finder(object)
        vect = np.hstack((histlbp,hog))
# Внесём изменения ###############################################################################################
        predictions = svc_model.predict([vect])
        # если модель предсказала класс 1, то мы считаем что необходимый объект найден,
        # если предсказан класс 0, то условие не выполняется.
        if predictions:
##################################################################################################################

            rect = patches.Rectangle((box[1], box[0]), box[3]-box[1], box[2]-box[0],
                                     linewidth=1, edgecolor='r', facecolor='none')
            count = count + 1
            ax.add_patch(rect)

    plt.show()
    print('Количество дорожных знаков «Пешеходный переход» =',count)
    new_image = np.array(fig.canvas.renderer.buffer_rgba())
    return count, new_image
```

Используем изменённую функцию на изображениях:

In [87]:
```
count, new_image = finder_v2(image)
```

Количество дорожных знаков «Пешеходный переход» = 3

In [88]:
```
image = plt.imread('image2.jpg')
count, new_image  = finder_v2(image)
```


```
Количество дорожных знаков «Пешеходный переход» = 1
```

Как видно данный алгоритм также показывает хорошие результаты при распознавании и подсчёте необходимых объектов. Таким образом были продемонстрированы два возможных алгоритма распознавания объектов на изображении, при этом количество способов решения поставленной задачи намного больше. Вы можете придумать собственный алгоритм на основе уже существующих или же попробовать реализовать представленные выше алгоритмы.