```python
# import the necessary packages
from scipy.spatial import distance as dist
from collections import OrderedDict
import numpy as np

class CentroidTracker:
	def __init__(self, maxDisappeared=50, maxDistance=50):
		# initialize the next unique object ID along with two ordered
		# dictionaries used to keep track of mapping a given object
		# ID to its centroid and number of consecutive frames it has
		# been marked as "disappeared", respectively
		self.nextObjectID = 0
		self.objects = OrderedDict()
		self.disappeared = OrderedDict()

		# store the number of maximum consecutive frames a given
		# object is allowed to be marked as "disappeared" until we
		# need to deregister the object from tracking
		self.maxDisappeared = maxDisappeared

		# store the maximum distance between centroids to associate
		# an object -- if the distance is larger than this maximum
		# distance we'll start to mark the object as "disappeared"
		self.maxDistance = maxDistance

	def register(self, centroid):
		# when registering an object we use the next available object
		# ID to store the centroid
		self.objects[self.nextObjectID] = centroid
		self.disappeared[self.nextObjectID] = 0
		self.nextObjectID += 1

	def deregister(self, objectID):
		# to deregister an object ID we delete the object ID from
		# both of our respective dictionaries
		del self.objects[objectID]
		del self.disappeared[objectID]

	def update(self, rects):
		# check to see if the list of input bounding box rectangles
		# is empty
		if len(rects) == 0:
			# loop over any existing tracked objects and mark them
			# as disappeared
			for objectID in list(self.disappeared.keys()):
				self.disappeared[objectID] += 1

				# if we have reached a maximum number of consecutive
				# frames where a given object has been marked as
				# missing, deregister it
				if self.disappeared[objectID] > self.maxDisappeared:
					self.deregister(objectID)

			# return early as there are no centroids or tracking info
			# to update
```

```
                        return self.objects

                # initialize an array of input centroids for the current frame
                inputCentroids = np.zeros((len(rects), 2), dtype="int")

                # loop over the bounding box rectangles
                for (i, (startX, startY, endX, endY)) in enumerate(rects):
                        # use the bounding box coordinates to derive the centroid
                        cX = int((startX + endX) / 2.0)
                        cY = int((startY + endY) / 2.0)
                        inputCentroids[i] = (cX, cY)

                # if we are currently not tracking any objects take the input
                # centroids and register each of them
                if len(self.objects) == 0:
                        for i in range(0, len(inputCentroids)):
                                self.register(inputCentroids[i])

                # otherwise, are are currently tracking objects so we need to
                # try to match the input centroids to existing object
                # centroids
                else:
                        # grab the set of object IDs and corresponding centroids
                        objectIDs = list(self.objects.keys())
                        objectCentroids = list(self.objects.values())

                        # compute the distance between each pair of object
                        # centroids and input centroids, respectively -- our
                        # goal will be to match an input centroid to an existing
                        # object centroid
                        D = dist.cdist(np.array(objectCentroids), inputCentroids)

                        # in order to perform this matching we must (1) find the
                        # smallest value in each row and then (2) sort the row
                        # indexes based on their minimum values so that the row
                        # with the smallest value as at the *front* of the index
                        # list
                        rows = D.min(axis=1).argsort()

                        # next, we perform a similar process on the columns by
                        # finding the smallest value in each column and then
                        # sorting using the previously computed row index list
                        cols = D.argmin(axis=1)[rows]

                        # in order to determine if we need to update, register,
                        # or deregister an object we need to keep track of which
                        # of the rows and column indexes we have already examined
                        usedRows = set()
                        usedCols = set()

                        # loop over the combination of the (row, column) index
                        # tuples
                        for (row, col) in zip(rows, cols):
                                # if we have already examined either the row or
                                # column value before, ignore it
```

```
111                             if row in usedRows or col in usedCols:
112                                     continue
113
114                             # if the distance between centroids is greater than
115                             # the maximum distance, do not associate the two
116                             # centroids to the same object
117                             if D[row, col] > self.maxDistance:
118                                     continue
119
120                             # otherwise, grab the object ID for the current row,
121                             # set its new centroid, and reset the disappeared
122                             # counter
123                             objectID = objectIDs[row]
124                             self.objects[objectID] = inputCentroids[col]
125                             self.disappeared[objectID] = 0
126
127                             # indicate that we have examined each of the row and
128                             # column indexes, respectively
129                             usedRows.add(row)
130                             usedCols.add(col)
131
132                     # compute both the row and column index we have NOT yet
133                     # examined
134                     unusedRows = set(range(0, D.shape[0])).difference(usedRows)
135                     unusedCols = set(range(0, D.shape[1])).difference(usedCols)
136
137                     # in the event that the number of object centroids is
138                     # equal or greater than the number of input centroids
139                     # we need to check and see if some of these objects have
140                     # potentially disappeared
141                     if D.shape[0] >= D.shape[1]:
142                             # loop over the unused row indexes
143                             for row in unusedRows:
144                                     # grab the object ID for the corresponding row
145                                     # index and increment the disappeared counter
146                                     objectID = objectIDs[row]
147                                     self.disappeared[objectID] += 1
148
149                                     # check to see if the number of consecutive
150                                     # frames the object has been marked "disappeared"
151                                     # for warrants deregistering the object
152                                     if self.disappeared[objectID] > self.maxDisappeared:
153                                             self.deregister(objectID)
154
155                     # otherwise, if the number of input centroids is greater
156                     # than the number of existing object centroids we need to
157                     # register each new input centroid as a trackable object
158                     else:
159                             for col in unusedCols:
160                                     self.register(inputCentroids[col])
161
162             # return the set of trackable objects
163             return self.objects
```