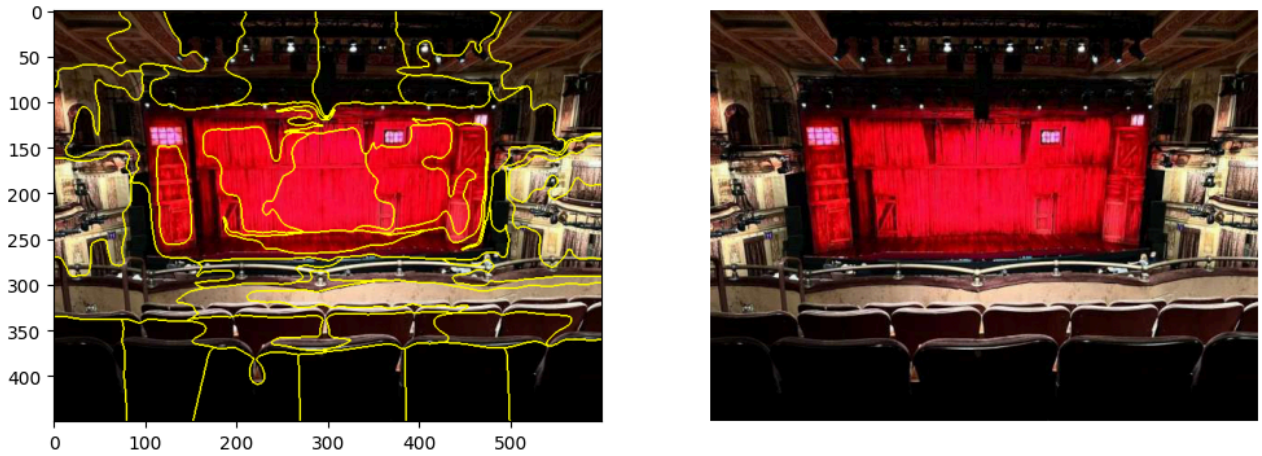
In [1]:
```
import sys
# allow import from Textons folder
sys.path.insert(1, 'Textons-master/')

# import the necessary packages
from skimage.segmentation import slic
from skimage.segmentation import mark_boundaries
from skimage.util import img_as_float
from skimage import io, exposure
from skimage.transform import resize
import skimage.feature as feature
import skimage.color

import matplotlib.pyplot as plt
import argparse
import numpy as np
import cv2
import random
from textons_utils import Textons

image = img_as_float(io.imread("pics/theater62.jpg"))
# loop over the number of segments
numSegments =  50
# apply SLIC and extract (approximately) the supplied number
# of segments
segments = slic(image, n_segments = numSegments, sigma = 5)

# show the output of SLIC
fig = plt.figure("Superpixels -- %d segments" % (numSegments))
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True, sharex=True, figsize=(12, 6))
ax1.imshow(mark_boundaries(image, segments))
ax2.imshow(image)

print(np.max(segments))
print(np.min(segments))
print(segments)
plt.axis("off")
# show the plots
plt.show()
```

```
36
1
[[ 1  1  1 ...  8  8  8]
 [ 1  1  1 ...  8  8  8]
 [ 1  1  1 ...  8  8  8]
 ...
 [31 31 31 ... 33 33 33]
 [31 31 31 ... 33 33 33]
 [31 31 31 ... 33 33 33]]
```

```
<Figure size 640x480 with 0 Axes>
```



In [335]:
```
def saveSegmentsToFile(segmentsByImg):
    fp = 'pics/segments/'
    for imgIdx in range (len(segmentsByImg)):
        fn = 'segments'+str(imgIdx)

        np.savetxt(fp+fn,segmentsByImg[imgIdx])


#saveSegmentsToFile(segmentsByImage)
```

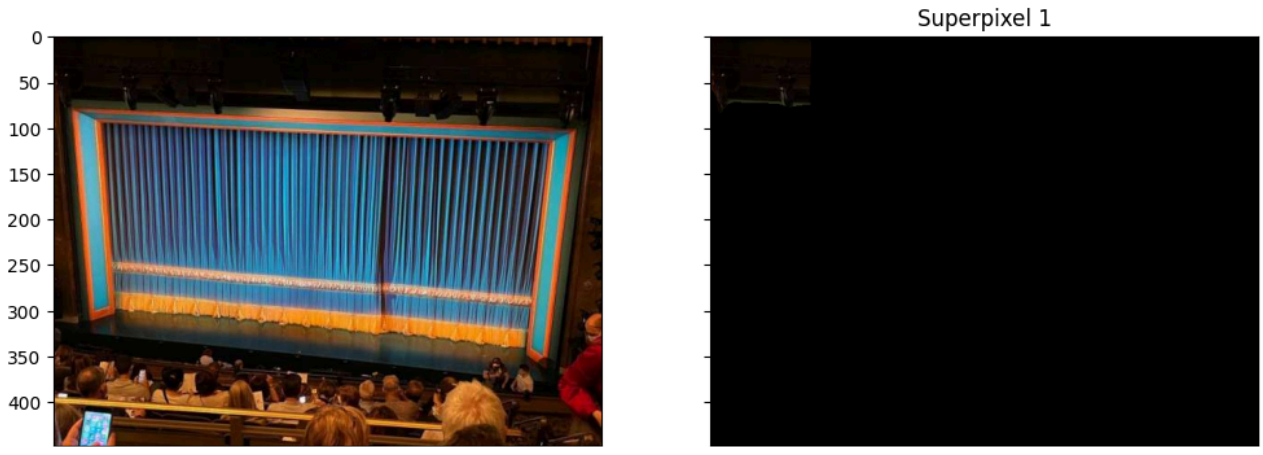
In [2]:
```
def loadSegments():
    segmentsArray = []
    fp = 'pics/segments/'
    for i in range (80):
        fn = 'segments'+str(i)
        segmentsArray.append(np.loadtxt(fp+fn))
    return segmentsArray
```

Function used for labeling images in training set, displays main image and superpixels side by side and takes in user input for the label to assign

Data is written to files inside of pics/groundtruthdata

In [78]:
```
import os, sys
def labelImagesInRange(begin,end):
    for n in range(begin,end+1):
        imfile = 'pics/theater'+str(n)+'.jpg'
        filepath = 'pics/groundtruthdata/gt'+str(n)+'.txt'

        image = img_as_float(io.imread(imfile))
        # loop over the number of segments
        numSegments =  50
        # apply SLIC and extract (approximately) the supplied number
        # of segments
        numSegments = 50
        segments = slic(image, n_segments = numSegments, sigma = 5)
        showAllSuperPixels(segments, image , filepath)


#labelImagesInRange(42,42)
```

```
<Figure size 640x480 with 0 Axes>
```



In [ ]:

1. Generate Superpixels (https://pyimagesearch.com/2014/07/28/a-slic-superpixel-tutorial-using-python/ (https://pyimagesearch.com/2014/07/28/a-slic-superpixel-tutorial-using-python/))
2. For each superpixel, generate feature vector

- C1. RGB values: mean (getRGBMeanandHSVValues)
- C2. HSV values: conversion from mean RGB values: see RGB
- T1. DOOG (derivative of gaussian) Filters: mean abs response

- T2. DOOG Filters: mean of variables in T1
- T3. DOOG Filters: id of max of variables in T1
- T4. DOOG Filters: (max - median) of variables in T1
- T5. Textons: mean abs response
- T6. Textons: max of variables in T5
- T7. Textons: (max - median) of variables in T5
- L1. Location: normalized x and y, mean

In [3]:
```
def showNthSuperPixel(segments, image, n , filename):

  f = open(filename,"a+")
  mask = (segments == n).astype(int)
  H, W, _ = image.shape
  superpixel = np.zeros(image.shape)
  superpixel[:, :, 0] = mask * image[:, :, 0]
  superpixel[:, :, 1] = mask * image[:, :, 1]
  superpixel[:, :, 2] = mask * image[:, :, 2]


  fig = plt.figure("Superpixels -- %d segments" % (numSegments))
  fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True, sharex=True, figsize=(12, 6))

  plt.title("Superpixel %d" % n)
  ax1.imshow(image)
  ax2.imshow(superpixel)
  plt.show(block=False)


  correctValueGiven = False
  while(not correctValueGiven):
        groundTruth = input("Ground Truth Value:" )
        validValues =  {'v','V','s','S','A','a'}
        correctValueGiven = groundTruth in validValues
  f.write(groundTruth.upper()+'\n')
  f.close()
  plt.close()

  #print(generateFeatureVector(n, image, segments))

def showAllSuperPixels(segments, image, filepath):
  maxSegment = np.max(segments)

  for n in range(1, maxSegment + 1):
    showNthSuperPixel(segments, image, n , filepath)


#showAllSuperPixels(segments, image , 'groundtruthim1.txt')
```

In [4]:
```
#def processSuperPixels(image,segments)
# Create list of np.max(segments)-1 slots
# for each superpixel, in segments, generate featurevector and store the feature vector in the list at
```

In [4]:
```
def getRGBMeanandHSVValues(superpixel , mask):
  #print(superpixel[mask].shape)

  B = superpixel[:,:,0]
  G = superpixel[:,:,1]
  R = superpixel[:,:,2]
 # B, G , R = cv2.split(superpixel[mask])

  Bmean = np.mean(B[mask==1])
  Gmean = np.mean(G[mask==1])
  Rmean = np.mean(R[mask==1])

  H,S,V = cv2.split(cv2.cvtColor(cv2.merge((Bmean, Gmean, Rmean)).astype('float32'), cv2.COLOR_BGR2HS\
  Hmean = H[0][0]
  Smean = S[0][0]
  Vmean = V[0][0]

  return Bmean , Gmean , Rmean , Hmean, Smean, Vmean
```

In [5]:
```
def getL1Value(superpixel,H,W, mask):
  '''
  Location in the image also provides strong cues for distinguishing
  between ground (tends to be low in the image), vertical structures,
  and sky (tends to be high in the image). We normalize the pixel
  locations by the width and height of the image and compute the
  mean (L1) of the x- and y-location
  of a region in the image.
     '''
  count, sumX, sumY = 0, 0, 0
  for x in range (0,W):
   for y in range (0,H):
     if mask[y,x] == 1 :
      count += 1
      xPos = x / W
      yPos = y / H
      sumX += xPos
      sumY += yPos
  return sumX/count, sumY/count
```

In [6]:
```
def getTextureValues(superpixel_im):
    gray_img = (255 * cv2.cvtColor(superpixel_im.astype('float32'), cv2.COLOR_BGR2GRAY)).astype('uint8

    graycom = feature.graycomatrix(gray_img, [1], [0, np.pi/4, np.pi/2, 3*np.pi/4], levels=256)

    # Find the GLCM properties
    contrast = feature.graycoprops(graycom, 'contrast')
    dissimilarity = feature.graycoprops(graycom, 'dissimilarity')
    homogeneity = feature.graycoprops(graycom, 'homogeneity')
    energy = feature.graycoprops(graycom, 'energy')
    correlation = feature.graycoprops(graycom, 'correlation')
    ASM = feature.graycoprops(graycom, 'ASM')

    return contrast, dissimilarity, homogeneity, energy, correlation, ASM
```

In [7]:
```
# Source for image cropping: https://stackoverflow.com/questions/40824245/how-to-crop-image-based-on-b
def get_segment_crop(img,tol=0, mask=None):
    if mask is None:
        mask = img > tol
    return img[np.ix_(mask.any(1), mask.any(0))]

# Generate feature vector given superpixel number
def generateFeatureVector(superpixel,image,segments):
  mask = (segments == (superpixel + 1)).astype(int)
  superpixelIm = np.zeros(image.shape)
  superpixelIm[:, :, 0] = mask * image[:, :, 0]
  superpixelIm[:, :, 1] = mask * image[:, :, 1]
  superpixelIm[:, :, 2] = mask * image[:, :, 2]
  superpixel_subimg = get_segment_crop(superpixelIm, mask=mask)
  mask_subimg = get_segment_crop(mask, mask=mask)
  Bmean, Gmean, Rmean, Hmean, Smean, Vmean = getRGBMeanandHSVValues(superpixel_subimg, mask_subimg)

  H, W, _= superpixel_subimg.shape

  L1X, L1Y = getL1Value(superpixel_subimg,H,W,mask_subimg)
  featureVector = [Bmean/2, Gmean/2, Rmean/2, Hmean/2, Smean/2, Vmean/2, L1X, L1Y*4]

  #contrast, dissimilarity, homogeneity, energy, correlation, ASM = getTextureValues(superpixel_subimg
  for result in getTextureValues(superpixel_subimg):
    for feature_value in result[0]:
        featureVector.append(feature_value)

  return np.array(featureVector)


#print(np.max(segmentsByImage[61]))
```

Next Steps for Likelihood function:

1. Parse data files in pics/groundtruthdata
2. Determine format for storing superpixel labels for each image

Overarching list (superpixelLabelsByImage) indexed by which image we're looking at image 1 --> index 0 in list ... image N --> index N - 1 in list

Each list element will be a sublist mapping the superpixel number to a corresponding label 1st superpixel in image --> index 0 in sublist

For example, accessing the third superpixel in the 40th image would correspond to: superpixelLabelsByImage[39][2] --> either "V", "A", "S"

3. Sampling to obtain same-label and different-label superpixel pairs Things to note: possibility of sampling a superpixel pair more than once --> we do not want this

each superpixel, we need information about which image it's from (image index), and what superpixel it is (superpixel index)

"V" label, "A" label, "S" label: list indexed by image, each element is a sublist of superpixel indices fitting the label, this could be generated when we actually parse the files because we have the label information immediately

sampling process: generate a random image index for same label: randomly pick one of the three labels, randomly sample 2 superpixels for that image and label for different label: randomly pick two labels, randomly sample a superpixel from each of the labels for that image

if repeat sampling causes an issue --> look into frozen sets or some representation of pairs we have already sampled

same label/different label data: list containing (image index, first superpixel index, second superpixel index)

4. Put data into a format for AdaBoost regression

In [51]:
```
def parseGroundTruthData():
    gt_fp = "pics/groundtruthdata"
    vlabelImageSuperpixels = [None]*80
    alabelImageSuperpixels = [None]*80
    slabelImageSuperpixels = [None]*80
    gtMax = np.zeros((80)).astype(np.uint8)
    imgsuperpixelToLabel = dict()
    for image_idx in range(1, 81):
        alabelSuperpixels = []
        vlabelSuperpixels = []
        slabelSuperpixels = []
        image_gt_fp = gt_fp + "/gt" + str(image_idx) + ".txt"

        gt_data = open(image_gt_fp,"r")
        superpixel_idx = 0
        for line in gt_data.readlines():
            label = line.strip()

            if (label == "A"):
                alabelSuperpixels.append(superpixel_idx)
                imgsuperpixelToLabel[(image_idx - 1, superpixel_idx)] = "A"

            if (label == "V"):
                vlabelSuperpixels.append(superpixel_idx)
                imgsuperpixelToLabel[(image_idx - 1, superpixel_idx)] = "V"

            if (label == "S"):
                slabelSuperpixels.append(superpixel_idx)
                imgsuperpixelToLabel[(image_idx - 1, superpixel_idx)] = "S"

            superpixel_idx += 1

        alabelImageSuperpixels[image_idx-1] = alabelSuperpixels
        vlabelImageSuperpixels[image_idx-1] = vlabelSuperpixels
        slabelImageSuperpixels[image_idx-1] = slabelSuperpixels
        gtMax[image_idx-1]=superpixel_idx
        gt_data.close()

    return alabelImageSuperpixels, vlabelImageSuperpixels, slabelImageSuperpixels,gtMax, imgsuperpixel
```

In [81]:
```
def sampleSuperpixelPairs(alabel, vlabel, slabel, npairs):
    same_label_pairs = []
    diff_label_pairs = []
    label = [alabel, vlabel, slabel]
    for _ in range(npairs):
        # same label pair
        valid_pair_found = False

        while not valid_pair_found:
            rand_img_idx = random.randint(0, 79)
            rand_label_idx = random.randint(0, 2)

            if (len(label[rand_label_idx][rand_img_idx]) >= 2):
                random_pair = random.sample(label[rand_label_idx][rand_img_idx], 2)
                valid_pair_found = True

        pair_info = [rand_img_idx, random_pair[0], random_pair[1]]
        same_label_pairs.append(pair_info)

        valid_pair_found = False
        while not valid_pair_found:
            rand_img_idx = random.randint(0, 79)
            rand_labels = random.sample([0, 1, 2], 2)
            first_label_idx = rand_labels[0]
            second_label_idx = rand_labels[1]

            if (len(label[first_label_idx][rand_img_idx]) == 0 or len(label[second_label_idx][rand_img
                continue

            pair_info = [rand_img_idx,
                         random.sample(label[first_label_idx][rand_img_idx], 1)[0],
                         random.sample(label[second_label_idx][rand_img_idx], 1)[0]]
            diff_label_pairs.append(pair_info)
            valid_pair_found = True


    return same_label_pairs, diff_label_pairs

alabel, vlabel, slabel, gtMax, imgsuperpixelToLabel = parseGroundTruthData()
same_label_pairs, diff_label_pairs = sampleSuperpixelPairs(alabel, vlabel, slabel, 2500)
```

In [9]:
```
# generate segments for each image
def generateSegmentsAndImages():
    segmentsByImage = []
    images = []
    for img_idx in range(1, 81):
        img_fp = "pics/theater" + str(img_idx) + ".jpg"
        image = img_as_float(io.imread(img_fp))
        # loop over the number of segments
        numSegments =  50
        # apply SLIC and extract (approximately) the supplied number
        # of segments
        segments = slic(image, n_segments = numSegments, sigma = 5)
        segmentsByImage.append(segments)
        images.append(image)
#         if (gtMax[img_idx-1] > np.max(segments)):
#             print(img_idx," ", np.max(segments))
#             print(gtMax[img_idx-1])

    return segmentsByImage, images

_, images = generateSegmentsAndImages()
segmentsByImage = loadSegments()
```

In [82]:
```
# 32 columns
ntraining = 5000

X = np.zeros((ntraining, 32)) # featurevector
y = np.zeros(ntraining)


# same labels
for i in range(int(ntraining/2)):
    curr_pair = same_label_pairs[i]
    img_idx = curr_pair[0]
    first_superpixel_idx = curr_pair[1]
    second_superpixel_idx = curr_pair[2]

    image = images[img_idx]
    segments = segmentsByImage[img_idx]

    if (first_superpixel_idx >= np.max(segments) or second_superpixel_idx >= np.max(segments) ):
        print(first_superpixel_idx," ", second_superpixel_idx," ",img_idx)
    first_fv = generateFeatureVector(first_superpixel_idx,image,segments)
    second_fv = generateFeatureVector(second_superpixel_idx,image,segments)

    X[i, :] = np.abs(first_fv - second_fv)
    y[i] = 1

for i in range(int(ntraining/2), ntraining):
    curr_pair = diff_label_pairs[i-(int(ntraining/2))]
    img_idx = curr_pair[0]
    first_superpixel_idx = curr_pair[1]
    second_superpixel_idx = curr_pair[2]

    image = images[img_idx]
    segments = segmentsByImage[img_idx]

    first_fv = generateFeatureVector(first_superpixel_idx,image,segments)
    second_fv = generateFeatureVector(second_superpixel_idx,image,segments)


    X[i, :] = np.abs(first_fv - second_fv)
    y[i] = 0
```

In [85]:
```
#!pip3 install scikit-learn

from sklearn.ensemble import AdaBoostRegressor
from sklearn import tree
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

# Code obtained: https://www.youtube.com/watch?v=vZ6tuxuYj3U&ab_channel=EducationalResearchTechniques
ada = AdaBoostRegressor()
search_grid = {'n_estimators': [500, 1000, 2000],
               'learning_rate': [.001, 0.01, 0.1], 'random_state': [1]}
cross_validation = KFold(n_splits=10, shuffle=True, random_state=1)
search = GridSearchCV(estimator=ada, param_grid=search_grid, scoring='neg_mean_squared_error', n_jobs=
                      cv = cross_validation)
```

In [86]:
```
search.fit(X, y)
print(search.best_params_)
print(search.best_score_)
```

```
{'learning_rate': 0.001, 'n_estimators': 500, 'random_state': 1}
-0.24262259577872927
```

In [83]:
```
feature_vector = dict()

def probabilitySameLabel(pix1,pix2,imageIdx):
    image = images[imageIdx]
    segments = segmentsByImage[imageIdx]
    if ((imageIdx,pix1) in feature_vector):
        first_fv = feature_vector[(imageIdx,pix1)]
    else:
        first_fv = generateFeatureVector(pix1,image,segments)
        feature_vector[(imageIdx,pix1)] = first_fv
    if ((imageIdx,pix2) in feature_vector):
        second_fv = feature_vector[(imageIdx,pix2)]
    else:
        second_fv = generateFeatureVector(pix2,image,segments)
        feature_vector[(imageIdx,pix2)] = second_fv

    return search.predict([np.abs(first_fv-second_fv)])[0]
```

In [292]:
```
curr_pair = diff_label_pairs[90]
img_idx = curr_pair[0]
first_superpixel_idx = curr_pair[1]
second_superpixel_idx = curr_pair[2]

image = images[img_idx]
segments = segmentsByImage[img_idx]

first_fv = generateFeatureVector(first_superpixel_idx,image,segments)
second_fv = generateFeatureVector(second_superpixel_idx,image,segments)

search.predict([np.abs(first_fv - second_fv)])
```

Out[292]:
```
array([0.57760091])
```

4.2 - Forming Constellations

In [10]:
```
def formConstellations(imageIdx, Nc):
    constellationArray = [None]*Nc
    #constellationArray[nc] = [superpixel1,...,superpixeln]
    segments = segmentsByImage[imageIdx]
    unique = np.unique(segments)
    unique = list(np.subtract(unique,1))
    initialSuperpixels = random.sample(unique,Nc)
    for i in range(0,Nc):
        constellationArray[i]=[initialSuperpixels[i]]
        unique.remove(initialSuperpixels[i])

    probSumArray = np.zeros((Nc)).astype(np.float32)
    for superpixel in unique:

        for cIdx in range (0,Nc):
            nK = len(constellationArray[cIdx])
            probSum = 0.0
            for otherpixel in constellationArray[cIdx]:
                probabilitySame = probabilitySameLabel(superpixel, otherpixel, imageIdx)
                probSum += probabilitySame
            probSum /= nK
#             if (nK !=1 ):
#                 probSum = probSum/ (nK*(1-nK))
#             else:
#                 probSum *= -1


            probSumArray[cIdx] = probSum
        constToAdd = np.argmax(probSumArray)
        constellationArray[constToAdd].append(superpixel)

    return(constellationArray)



# constellations = formConstellations(32,5)
# print(constellations)
# displayConstellation(constellations,32)
```

In [95]:
```
## import copy

colorsArray = [ [ 148,0,211], [75,0,130 ] , [0,0,255], [0,255,0 ] , [255,255,0 ] , [255,127,0 ],[255,0

def displayConstellation(constellationArray,imageIdx):
    image = images[imageIdx]
    # loop over the number of segments
    segments = segmentsByImage[imageIdx]
    constellations = copy.copy(segments)

    colorImage = np.zeros((image.shape))
    H,W, _= image.shape

    superpixelToConstellation = dict()
    for constellationIdx in range (len(constellationArray)):
        constellation = constellationArray[constellationIdx]
        for superpixel in constellation:
            superpixelToConstellation[superpixel] = constellationIdx

    for i in range (0,H):
        for j in range (0,W):
            superpixelIJ = segments[i][j]-1
            constIdx  =  superpixelToConstellation[superpixelIJ]
            colorImage[i][j][0] = colorsArray[constIdx][0]
            colorImage[i][j][1] = colorsArray[constIdx][1]
            colorImage[i][j][2] = colorsArray[constIdx][2]


    bwImage = cv2.cvtColor(image.astype(np.float32),cv2.COLOR_RGB2GRAY)

    bwImage2Color = skimage.color.gray2rgb(bwImage)

    composedImage = bwImage2Color*0.5 + colorImage*0.5/255

    # show the output of SLIC
    fig = plt.figure("Superpixels -- %d segments" % (numSegments))
    fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True, sharex=True, figsize=(12, 6))
    ax1.imshow(mark_boundaries(image, segments))
    ax2.imshow(cv2.cvtColor((composedImage).astype(np.float32),cv2.COLOR_RGB2BGR))




    plt.axis("off")
    # show the plots
    plt.show()
```

In [12]:
```
def displayNthConstellation(constellationArray,imgIdx,n):
    image = images[imgIdx]
    segments = segmentsByImage[imgIdx]
    constellations = copy.copy(segments)

    colorImage = np.zeros((image.shape))
    H,W, _= image.shape

    superpixelToConstellation = dict()
    for constellationIdx in range (len(constellationArray)):
        constellation = constellationArray[constellationIdx]
        for superpixel in constellation:
            superpixelToConstellation[superpixel] = constellationIdx

    bwImage = cv2.cvtColor(image.astype(np.float32),cv2.COLOR_RGB2GRAY)

    for i in range (0,H):
        for j in range (0,W):
            superpixelIJ = segments[i][j]-1
            constIdx  =  superpixelToConstellation[superpixelIJ]
            if (constIdx == n):
                colorImage[i][j][0] = colorsArray[constIdx][0]
                colorImage[i][j][1] = colorsArray[constIdx][1]
                colorImage[i][j][2] = colorsArray[constIdx][2]
            else:
                bwImage[i,j] = 0



    bwImage2Color = skimage.color.gray2rgb(bwImage)

    composedImage = bwImage2Color*0.5 + colorImage*0.5/255

    # show the output of SLIC
    return composedImage
```

Constellation training plan:

1 - Generate 3 constellation images for each original image using Nc = 4, 5, 6. 2 - Store each constellation image array to a file /pics/constellation/imXconst4.txt

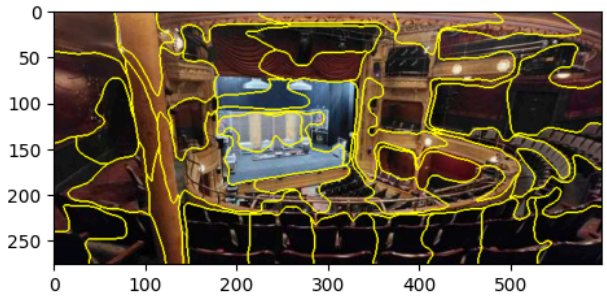
In [106]:
```
import pickle
def generateConstellationTrainingImage(imIdx):
    fp = "pics/constellation/"
    for Nc in range (4,7):
        constellationArrayNc = formConstellations(imIdx,Nc)
        fname = "im"+str(imIdx)+"const"+str(Nc)
        with open(fp+fname, "w+") as f:
            for constellation in constellationArrayNc:
                f.write("$".join(list(map(str,constellation)))+"\n")
```

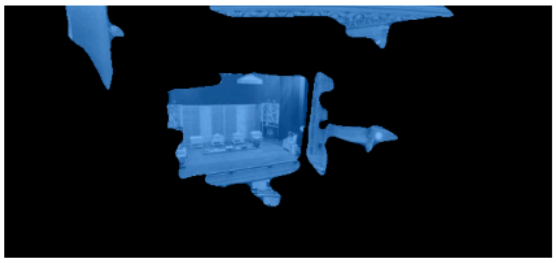
In [107]:
```
def generateandlabelconstellationtrainingforNthIm(imIdx):
    generateConstellationTrainingImage(imIdx)
    readConstellationTrainingImageandLabel(imIdx)
```

In [117]:
```
import copy
def readConstellationTrainingImageandLabel(imIdx):
    fp = "pics/constellation/"
    gtfp = "pics/constellation/groundtruthdata/"
    segments = segmentsByImage[imIdx].astype(np.uint8)
    image = images[imIdx]
    validValues = {'A','a','s','S','v','V','M','m'}
    for Nc in range (4,7):
        print("PRINTING CONSTELLATION Nc "+str(Nc))
        fname = "im"+str(imIdx)+"const"+str(Nc)
        constellationNc = []
        with open(fp+fname,'r') as f:
            for line in f.readlines():
                constellationNc.append(list(map(int,list(map(float,line.split('$'))))))
        for constellationIdx in range (len(constellationNc)):
            composedImage = displayNthConstellation(constellationNc,imIdx,constellationIdx)
            fig = plt.figure("Superpixels -- %d segments" % (numSegments))
            fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True, sharex=True, figsize=(12, 6))
            ax1.imshow(mark_boundaries(image, segments))
            ax2.imshow(cv2.cvtColor((composedImage).astype(np.float32),cv2.COLOR_RGB2BGR))

            plt.axis("off")
            # show the plots
            plt.show(block=False)

            gtfname = 'gt'+str(imIdx)+'nc'+str(Nc)
            f = open (gtfp+gtfname,"a+")
            correctValueGiven = False
            while(not correctValueGiven):
                groundTruth = input("Ground Truth Value:")
                correctValueGiven = groundTruth in validValues
            f.write(groundTruth.upper()+'\n')
            f.close()
            plt.close()



generateandlabelconstellationtrainingforNthIm(4)
```


```
Ground Truth Value:a
```

```
<Figure size 640x480 with 0 Axes>
```





```
Ground Truth Value:s
```

In [16]:
```
def findLineIntersection(lineAB,lineCD):

    parallelThreshold = .01

    ax,ay,bx,by = lineAB[0]
    cx,cy,dx,dy = lineCD[0]

    if (bx==ax):
        slopeAB = 10**9
    else:
        slopeAB = (by-ay)/(bx-ax)
    if (dx == cx):
        slopeCD = 10**9
    else:
        slopeCD = (dy-cy)/(dx-cx)



    a1 = by-ay
    b1 = ax-bx
    c1 = a1*ax + b1*ay


    a2 = dy - cy
    b2 = cx - dx
    c2 = a2*cx + b2*cy


    determinant = a1*b2 - a2*b1

    if ((np.abs(slopeAB-slopeCD)) < parallelThreshold):
        # The lines are parallel or nearly. This is simplified
        # by returning a pair of FLT_MAX
        return (10**9, 10**9)
    else:
        x = (b2*c1 - b1*c2)/determinant
        y = (a1*c2 - a2*c1)/determinant
#         print(x,y)
        return (x, y)
```

In [17]:
```
findLineIntersectionMetrics(lines,sub_img):
H,W,_ = sub_img.shape
pairCount = 0
rightOfCenterCount = 0
aboveCenterCount = 0
farFromCenterCount = 0
veryFarFromCenterCount = 0


for lineAIdx in range(len(lines)):
    for lineBIdx in range(lineAIdx+1,len(lines)):
        lineA = lines[lineAIdx]
        lineB = lines[lineBIdx]
        pairCount += 1
        intersectionX, intersectionY = findLineIntersection(lineA, lineB)
        if (intersectionX > (W/2)):
            rightOfCenterCount += 1
        if (intersectionY > (H/2)):
            aboveCenterCount += 1
        distanceFromCenter = np.sqrt((intersectionX - (W/2))**2 + (intersectionY - (H/2))**2)
        if (distanceFromCenter > 150): # To Do - thresholds for distance from center metrics
            farFromCenterCount +=1
        if (distanceFromCenter > 250):
            veryFarFromCenterCount +=1

if (pairCount < 1):
    return 0,0,0,0

else:
    return rightOfCenterCount/pairCount, aboveCenterCount/pairCount, farFromCenterCount/pairCount, ver
```

In [18]:
```
def findLongLines(lines):
    return len(lines)
```

In [19]:
```
def findParallelLines(lines):
    parallelCount = 0
    pairCount = 0
    for line1Idx in range (len(lines)):
        for line2Idx in range(line1Idx+1, len(lines)):
            line1 = lines[line1Idx]
            line2 = lines[line2Idx]
            pairCount += 1
            intersection = findLineIntersection(line1,line2)
            if (intersection[0] == 10**9 and intersection[1] == 10**9):
                parallelCount+=1
            else:
                intX, intY = intersection
                lineX1,lineY1,_,_ = line1[0]
                distance = np.sqrt((intX-lineX1)**2 + (intY-lineY1)**2)
                if (distance > 10000):
                    parallelCount+=1
    if pairCount == 0:
        return 0
    else:
        return (parallelCount/pairCount)*100



paral = [[
  [0,0,0,1]
],[
 [1,0,55,100000]
]]
print(findParallelLines(paral))
```

```
0.0
```

In [21]:
```
def calcLineFeatures(constellation_subimg,H,W,mask_subimg):

    lineFeatures = []
    gray = cv2.cvtColor(constellation_subimg.astype(np.float32),cv2.COLOR_RGB2GRAY)
    kernel_size = 5
    blur_gray = cv2.GaussianBlur(gray,(kernel_size, kernel_size),0)
    low_threshold = 50
    high_threshold = 150
    edges = cv2.Canny((blur_gray*255).astype(np.uint8), low_threshold, high_threshold)
    rho = 1  # distance resolution in pixels of the Hough grid
    theta = np.pi / 180  # angular resolution in radians of the Hough grid
    threshold = 15  # minimum number of votes (intersections in Hough grid cell)
    min_line_length = 100  # minimum number of pixels making up a line
    max_line_gap = 20  # maximum gap in pixels between connectable line segments
    line_image = np.copy(constellation_subimg) * 0  # creating a blank to draw lines on
    # Run Hough on edge detected image
    # Output "lines" is an array containing endpoints of detected line segments
    lines = cv2.HoughLinesP(edges, rho, theta, threshold, np.array([]),min_line_length, max_line_gap)

    if lines is None:
        return [0] * 6 # TO DO -- populate with zeros for all.

    for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(line_image,(x1,y1),(x2,y2),(255,0,0),5)

    # Draw the lines on the  image
#     lines_edges = cv2.addWeighted(constellation_subimg, 0.8, line_image, 1, 0)

#     plt.imshow(lines_edges)
#     plt.show()

    longLineCount = findLongLines(lines)
    parallelCount = findParallelLines(lines)
    right, above, far, veryfar = findLineIntersectionMetrics(lines,constellation_subimg)
    return [longLineCount, parallelCount, right, above, far, veryfar]

# G1. Long Lines: total number in constellation region (can do)
# G2. Long Lines: % of nearly parallel pairs of lines (can do)
# G3. Line Intersection: hist. over 12 orientations, entropy
# G4. Line Intersection: % right of center (can do)
# G5. Line Intersection: % above center (can do)
# G6. Line Intersection: % far from center at 8 orientations
# G7. Line Intersection: % very far from center at 8 orientations
# G8. Texture gradient: x and y “edginess” (T2) center
```

Constellation Label and Homogeneity Likelihoods End Goal: Estimate the likelihood of a superpixel having a specific label. We need: 1 - A label likelihood for each superpixel: a) Find the the set of constellations that contains the superpixel. b) For each constellation, calculate feature vector. c) For each constellation, calculate likelihood that it has label t (this is done based off of training data w/ feature vector & ground truth data set). d) For each constellation, calculate the homogeneity likelihood (feature vector, was the given constellation homogenous or not - mixed/not mixed).

In [22]:
```
def getConstellation(imIdx, Nc):
    fp = "pics/constellation/"
    gtfp = "pics/constellation/groundtruthdata/"
    segments = segmentsByImage[0]
    image = images[0]
    fname = "im"+str(imIdx)+"const"+str(Nc)
    constellationNc = []
    with open(fp+fname,'r') as f:
        for line in f.readlines():
            constellationNc.append(list(map(int, list(map(float,line.split('$'))))))
    return constellationNc



print(getConstellation(0,4))
```

```
[[6, 0, 1, 2, 3, 4, 5, 8, 9, 12, 13, 15, 17, 38], [34, 14, 19, 20, 21, 23, 27, 28, 29, 30, 35, 36, 3
7, 40], [25, 7, 10, 11, 22, 24], [31, 16, 18, 26, 32, 33, 39]]
```

In [88]:
```
def get_segment_crop(img,tol=0, mask=None):
    if mask is None:
        mask = img > tol
    return img[np.ix_(mask.any(1), mask.any(0))]

def generateTestConstellationFeatures(constellation, image, segments):
    H,W,_ = image.shape
    mask = np.zeros((H,W))
    for superpixel in constellation:
        superPixelMask = ((segments == superpixel + 1)).astype(int)
        mask += superPixelMask


    constellationIm = np.zeros(image.shape)
    constellationIm[:, :, 0] = mask * image[:, :, 0]
    constellationIm[:, :, 1] = mask * image[:, :, 1]
    constellationIm[:, :, 2] = mask * image[:, :, 2]


    constellation_subimg = get_segment_crop(constellationIm, mask=mask)
    mask_subimg = get_segment_crop(mask, mask=mask)
    Bmean, Gmean, Rmean, Hmean, Smean, Vmean = getRGBMeanandHSVValues(constellation_subimg, mask_subim

    H, W, _= constellation_subimg.shape

    L1X, L1Y = getL1Value(constellation_subimg,H,W,mask_subimg)
    featureVector = [Bmean/2, Gmean/2, Rmean/2, Hmean/2, Smean/2, Vmean/2, L1X, L1Y*4]

    #contrast, dissimilarity, homogeneity, energy, correlation, ASM = getTextureValues(superpixel_subi
    for result in getTextureValues(constellation_subimg):
        for feature_value in result[0]:
            featureVector.append(feature_value)

    line_features = calcLineFeatures(constellation_subimg,H,W,mask_subimg)
    for feature in line_features:
        featureVector.append(feature)

    return np.array(featureVector)

def generateConstellationFeatures(imIdx,constellation):
    image=images[imIdx]
    segments = segmentsByImage[imIdx]
    H,W,_ = image.shape
    mask = np.zeros((H,W))
    for superpixel in constellation:
        superPixelMask = ((segments == superpixel + 1)).astype(int)
        mask += superPixelMask


    constellationIm = np.zeros(image.shape)
    constellationIm[:, :, 0] = mask * image[:, :, 0]
    constellationIm[:, :, 1] = mask * image[:, :, 1]
    constellationIm[:, :, 2] = mask * image[:, :, 2]


    constellation_subimg = get_segment_crop(constellationIm, mask=mask)
    mask_subimg = get_segment_crop(mask, mask=mask)
    Bmean, Gmean, Rmean, Hmean, Smean, Vmean = getRGBMeanandHSVValues(constellation_subimg, mask_subim

    H, W, _= constellation_subimg.shape

    L1X, L1Y = getL1Value(constellation_subimg,H,W,mask_subimg)
    featureVector = [Bmean/2, Gmean/2, Rmean/2, Hmean/2, Smean/2, Vmean/2, L1X, L1Y*4]

    #contrast, dissimilarity, homogeneity, energy, correlation, ASM = getTextureValues(superpixel_subi
    for result in getTextureValues(constellation_subimg):
        for feature_value in result[0]:
            featureVector.append(feature_value)

    line_features = calcLineFeatures(constellation_subimg,H,W,mask_subimg)
    for feature in line_features:
        featureVector.append(feature)

    return np.array(featureVector)

features = generateConstellationFeatures(1, getConstellation(1, 4)[0])
```

```
print(len(features))
```

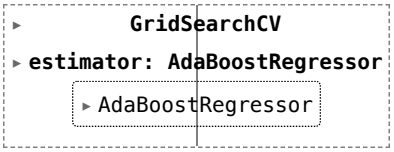
```
38
```

In [118]:
```
def parseConstellationGroundTruthData():
    gt_fp = "pics/constellation/groundtruthdata/"
    ct_fp = "pics/constellation/"
    # X --> feature vectors of all constellations
    # ya --> prob label a
    # yv --> prob label v
    # ys --> prob label s
    # ym --> prob label m
    X = np.zeros((1200, 38))
    ya = np.zeros(1200)
    yv = np.zeros(1200)
    ys = np.zeros(1200)
    ym = np.zeros(1200)

    # 80 images -> 4 + 5 + 6 = 15 total constellations, 80 * 15 = 1200
    constIdx = 0
    for image_idx in range(0,80):
        for Nc in range(4,7):
            constellationSet = getConstellation(image_idx, Nc)
            cur_gt = gt_fp + "gt" + str(image_idx) + "nc" + str(Nc)
            gt_f = open(cur_gt, "r")
            for constellation in constellationSet:
                line = gt_f.readline()
                X[constIdx] = generateConstellationFeatures(image_idx, constellation)

                if (line.strip() == "A"):
                    ya[constIdx] = 1
                elif (line.strip() == "V"):
                    yv[constIdx] = 1
                elif (line.strip() == "S"):
                    ys[constIdx] = 1
                elif (line.strip() == "M"):
                    ym[constIdx] = 1

                constIdx += 1

            gt_f.close()

    return X, ya, yv, ys, ym

X, ya, yv, ys, ym = parseConstellationGroundTruthData()
```

In [119]:
```
from sklearn.ensemble import AdaBoostRegressor
from sklearn import tree
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

# Code obtained: https://www.youtube.com/watch?v=vZ6tuxuYj3U&ab_channel=EducationalResearchTechniques
ada_a = AdaBoostRegressor()
search_grid_a = {'n_estimators': [8],
               'learning_rate': [.001, 0.01, 0.1], 'random_state': [1]}
cross_validation_a = KFold(n_splits=10, shuffle=True, random_state=1)
search_a = GridSearchCV(estimator=ada_a, param_grid=search_grid_a, scoring='neg_mean_squared_error', r
                    cv = cross_validation_a)

ada_v = AdaBoostRegressor()
search_grid_v = {'n_estimators': [8],
               'learning_rate': [.001, 0.01, 0.1], 'random_state': [1]}
cross_validation_v = KFold(n_splits=10, shuffle=True, random_state=1)
search_v = GridSearchCV(estimator=ada_v, param_grid=search_grid_v, scoring='neg_mean_squared_error', r
                    cv = cross_validation_v)

ada_s = AdaBoostRegressor()
search_grid_s = {'n_estimators': [8],
               'learning_rate': [.001, 0.01, 0.1], 'random_state': [1]}
cross_validation_s = KFold(n_splits=10, shuffle=True, random_state=1)
search_s = GridSearchCV(estimator=ada_s, param_grid=search_grid_s, scoring='neg_mean_squared_error', r
                    cv = cross_validation_s)

ada_m = AdaBoostRegressor()
search_grid_m = {'n_estimators': [8],
               'learning_rate': [.001, 0.01, 0.1], 'random_state': [1]}
cross_validation_m = KFold(n_splits=10, shuffle=True, random_state=1)
search_m = GridSearchCV(estimator=ada_m, param_grid=search_grid_m, scoring='neg_mean_squared_error', r
                    cv = cross_validation_m)
```

In [120]:
```
search_a.fit(X, ya)
search_v.fit(X, yv)
search_s.fit(X, ys)
search_m.fit(X, ym)
```

Out[120]:



In [167]:
```
def formTestConstellations(image, segments, Nc):
    constellationArray = [None]*Nc
    #constellationArray[nc] = [superpixel1,...,superpixeln]
    unique = np.unique(segments)
    unique = list(np.subtract(unique,1))
    initialSuperpixels = random.sample(unique,Nc)
    for i in range(0,Nc):
        constellationArray[i]=[initialSuperpixels[i]]
        unique.remove(initialSuperpixels[i])

    probSumArray = np.zeros((Nc)).astype(np.float32)
    for superpixel in unique:

        for cIdx in range (0,Nc):
            nK = len(constellationArray[cIdx])
            probSum = 0.0
            for otherpixel in constellationArray[cIdx]:
                probabilitySame = probabilitySameLabel(superpixel, otherpixel, imageIdx)
                probSum += probabilitySame
            probSum /= nK
#             if (nK !=1 ):
#                 probSum = probSum/ (nK*(1-nK))
#             else:
#                 probSum *= -1


            probSumArray[cIdx] = probSum
        constToAdd = np.argmax(probSumArray)
        constellationArray[constToAdd].append(superpixel)

    return(constellationArray)

def displayConst(imIdx, constellation):
    image=images[imIdx]
    segments = segmentsByImage[imIdx]
    H,W,_ = image.shape
    mask = np.zeros((H,W))
    for superpixel in constellation:
        superPixelMask = ((segments == superpixel + 1)).astype(int)
        mask += superPixelMask


    constellationIm = np.zeros(image.shape)
    constellationIm[:, :, 0] = mask * image[:, :, 0]
    constellationIm[:, :, 1] = mask * image[:, :, 1]
    constellationIm[:, :, 2] = mask * image[:, :, 2]

    plt.imshow(constellationIm)
    plt.title("Constellation Scores")
    plt.axis("off")
    plt.show()

def labelImage(image):
    numSegments =  50
    # apply SLIC and extract (approximately) the supplied number
    # of segments
    segments = slic(image, n_segments = numSegments, sigma = 5)
    superpixelLabelConfidences_asv = dict()

    unique = np.unique(segments)
    unique = list(np.subtract(unique,1))
    for superpixelidx in unique:
        superpixelLabelConfidences_asv[superpixelidx] = np.array([0.0, 0.0, 0.0])

    for Nc in range(4,15):
        constellations = formTestConstellations(image, segments, Nc)
        for constellation in constellations:
            constellationFeatures = generateTestConstellationFeatures(constellation, image, segments)
            a_score = search_a.predict([constellationFeatures])[0]
            s_score = search_s.predict([constellationFeatures])[0]
            v_score = search_v.predict([constellationFeatures])[0]
            homogeneity = 1.0 - search_m.predict([constellationFeatures])[0]

            for superpixelidx in constellation:
                superpixelLabelConfidences_asv[superpixelidx][0] += a_score * homogeneity
                superpixelLabelConfidences_asv[superpixelidx][1] += s_score * homogeneity
```

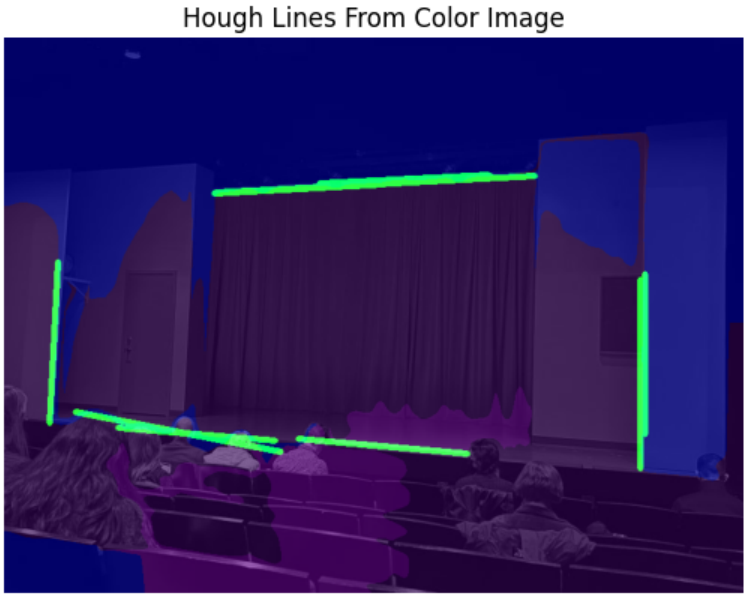
```
                superpixelLabelConfidences_asv[superpixelidx][2] += v_score * homogeneity

    H, W = segments.shape
    finallabels = np.zeros((H, W))
    # 0 --> label a, 1 --> label s, 2 --> label v
    for i in range(H):
        for j in range(W):
            superpixelidx = segments[i][j] - 1
            bestlabel = np.argmax(superpixelLabelConfidences_asv[superpixelidx])
            finallabels[i][j] = bestlabel

    return finallabels

def showLines(img, bwimage):
    H, W, _ = img.shape
    lineFeatures = []
    gray = cv2.cvtColor(img.astype(np.float32),cv2.COLOR_RGB2GRAY)
    kernel_size = 5
    blur_gray = cv2.GaussianBlur(gray,(kernel_size, kernel_size),0)
    low_threshold = 50
    high_threshold = 150
    edges = cv2.Canny((blur_gray*255).astype(np.uint8), low_threshold, high_threshold)
    rho = 1  # distance resolution in pixels of the Hough grid
    theta = np.pi / 180  # angular resolution in radians of the Hough grid
    threshold = 15  # minimum number of votes (intersections in Hough grid cell)
    min_line_length = 170  # minimum number of pixels making up a line
    max_line_gap = 20  # maximum gap in pixels between connectable line segments
    line_image = np.copy(img) * 0  # creating a blank to draw lines on
    # Run Hough on edge detected image
    # Output "lines" is an array containing endpoints of detected line segments
    lines = cv2.HoughLinesP(edges, rho, theta, threshold, np.array([]),min_line_length, max_line_gap)

    if lines is None:
        return [0] * 6 # TO DO -- populate with zeros for all.

    for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(line_image,(x1,y1),(x2,y2),(0,255,0),5)

    #Draw the lines on the  image
    lines_edges = cv2.addWeighted(bwimage*0.5 + img*0.5/255, 0.8, line_image, 1, 0)

    plt.figure()
    plt.title("Image Colored By Constellation")
    plt.imshow(bwimage*0.5 + img*0.5/255)
    plt.axis("off")
    plt.show()

    plt.figure()
    plt.title("Hough Lines From Color Image")
    plt.imshow(lines_edges)
    plt.axis("off")
    plt.show()

    return lines

def displayFinalLabels(image, finallabels):
    colorImage = np.zeros((image.shape))
    H,W, _= image.shape

    bwImage = cv2.cvtColor(image.astype(np.float32),cv2.COLOR_RGB2GRAY)

    for i in range (0,H):
        for j in range (0,W):
            labelIdx  =  int(finallabels[i][j])
            colorImage[i][j][0] = colorsArray[labelIdx][0]
            colorImage[i][j][1] = colorsArray[labelIdx][1]
            colorImage[i][j][2] = colorsArray[labelIdx][2]

    bwImage2Color = skimage.color.gray2rgb(bwImage)

    composedImage = bwImage2Color*0.5 + colorImage*0.5/255
    return showLines(colorImage, bwImage2Color)
#     plt.figure()
#     plt.title("Final Labels")
#     plt.imshow(composedImage)
#     plt.show()
```

```
#     plt.figure()
#     plt.imshow(colorImage)
#     plt.show()

# imageIdx = 50
# Nc = 6
# conIdx = 2

# print("audience_score: ",search_a.predict([generateConstellationFeatures(imageIdx,
#                                                                          getConstellation(imageIdx,Nc
# print("stage_score: ",search_s.predict([generateConstellationFeatures(imageIdx,
#                                                                       getConstellation(imageIdx,Nc)[c
# print("vertical_score: ",search_v.predict([generateConstellationFeatures(imageIdx,
#                                                                          getConstellation(imageIdx, N
# print("mixed_score: ", search_m.predict([generateConstellationFeatures(imageIdx,
#                                                                        getConstellation(imageIdx, Nc)
# displayConst(imageIdx, getConstellation(imageIdx, Nc)[conIdx])
```

In [149]:
```
image = img_as_float(io.imread("testimage.jpeg"))
finallabels = labelImage(image)
```

In [168]:
```
lines = displayFinalLabels(image, finallabels)
```

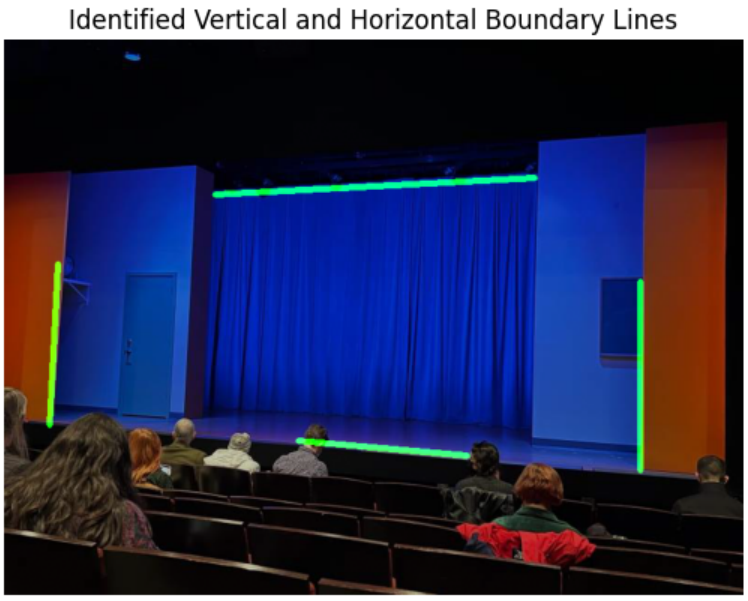

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for i
ntegers).
```



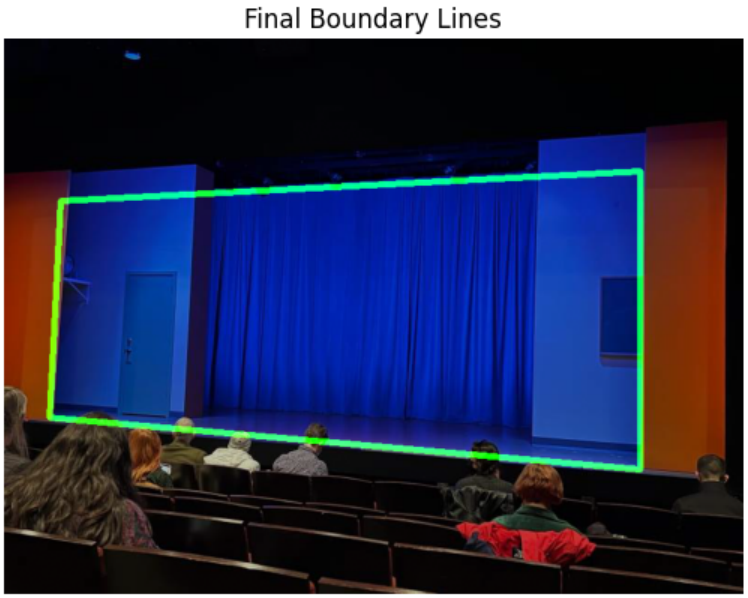
In [215]:
```
def getBoundaryLines(image, lines):
    H, W, _ = image.shape

    vertical_lines = []
    horizontal_lines = []
    line_image = np.copy(image) * 0

    for line in lines:
        x1, y1, x2, y2 = line[0]

        if (x2 == x1):
            slope = 10 ** 9
        else:
            slope = np.abs((y2 - y1)/(x2 - x1))

        if (slope < 0.1):
            horizontal_lines.append(line)
        elif (slope > 10):
            vertical_lines.append(line)

    top_horizontal_lines = []
    bottom_horizontal_lines = []
    left_vertical_lines = []
    right_vertical_lines = []

    for line in horizontal_lines:
        x1, y1, x2, y2 = line[0]
        if (y1 < H/2):
            top_horizontal_lines.append(line)
        else:
            bottom_horizontal_lines.append(line)

    for line in vertical_lines:
        x1, y1, x2, y2 = line[0]
        if (x1 < W/2):
            left_vertical_lines.append(line)
        else:
            right_vertical_lines.append(line)

    result_lines = [left_vertical_lines[0], right_vertical_lines[0], top_horizontal_lines[0], bottom_h

    for line in result_lines:
        for x1,y1,x2,y2 in line:
            cv2.line(line_image,(x1,y1),(x2,y2),(0,255,0),5)

    #Draw the lines on the  image
    lines_edges = cv2.addWeighted(image, 0.8, line_image, 1, 0)
    plt.imshow(lines_edges)
    plt.title("Identified Vertical and Horizontal Boundary Lines")
    plt.axis("off")
    plt.show()

    return result_lines
```

In [277]:
```
def findBoundaryPoints(left, right, top, bottom):
    topleft = findLineIntersection(left, top)
    topright = findLineIntersection(top, right)
    bottomleft = findLineIntersection(left, bottom)
    bottomright = findLineIntersection(bottom, right)

    return topleft, topright, bottomleft, bottomright

def getFinalLines(topleft, topright, bottomleft, bottomright, image, display=False):
    lines = []
    line_image = np.copy(image) * 0
    H, W, _ = image.shape

    # audience separator
    _, y1 = tuple(map(int, bottomleft))
    _, y2 = tuple(map(int, bottomright))
    aud_line = [[0, y1, W, y2]]

    #top line
    x1, y1 = tuple(map(int, topleft))
    x2, y2 = tuple(map(int, topright))
    top_line = [[x1, y1, x2, y2]]

    lines.append(top_line)

    #bottom line
    x1, y1 = tuple(map(int, bottomleft))
    x2, y2 = tuple(map(int, bottomright))
    bottom_line = [[x1, y1, x2, y2]]

    lines.append(bottom_line)

    #left line
    x1, y1 = tuple(map(int, topleft))
    x2, y2 = tuple(map(int, bottomleft))
    left_line = [[x1, y1, x2, y2]]

    lines.append(left_line)

    #right line
    x1, y1 = tuple(map(int, topright))
    x2, y2 = tuple(map(int, bottomright))
    right_line = [[x1, y1, x2, y2]]

    lines.append(right_line)


    if (display):
        for line in lines:
            for x1,y1,x2,y2 in line:
                cv2.line(line_image,(x1,y1),(x2,y2),(0,255,0),5)

        #Draw the lines on the  image
        lines_edges = cv2.addWeighted(image, 0.8, line_image, 1, 0)
        plt.imshow(lines_edges)
        plt.title("Final Boundary Lines")
        plt.axis("off")
        plt.show()

    return aud_line, lines

image = img_as_float(io.imread("testimage.jpeg"))
left, right, top, bottom = getBoundaryLines(image, lines)
topleft, topright, bottomleft, bottomright = findBoundaryPoints(left, right, top, bottom)
getFinalLines(topleft, topright, bottomleft, bottomright, image, display=True)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for i
ntegers).
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for i
ntegers).
```



Out[277]:
```
([[0, 409, 800, 465]],
 [[[61, 175, 688, 143]],
  [[49, 409, 688, 465]],
  [[61, 175, 49, 409]],
  [[688, 143, 688, 465]]])
```

In [383]:
```
def getAudience(image, aud_line):
    x1, y1, x2, y2 = aud_line[0]
    m = (y2 - y1)/(x2 - x1)
    b = y1 - m*x1

    H, W, _ = image.shape
    audience = np.zeros(image.shape)
    mask = np.zeros((H, W))

    for y in range(H):
        for x in range(W):
            if (y > m * x + b):
                audience[y][x] = image[y][x]
                mask[y][x] = 1

    audience_crop = get_segment_crop(audience,tol=0, mask=mask)
    mask_crop = get_segment_crop(mask, mask=mask)
    return audience_crop, mask_crop

def slopeIntercept(line):
    x1, y1, x2, y2 = line[0]

    if (x2 == x1):
        m = 10 ** 9
    else:
        m = (y2 - y1)/(x2 - x1)
    b = y1 - m * x1
    return m, b

def getStage(image, stage_lines):
    stage = np.zeros(image.shape)
    H, W, _ = image.shape
    mask = np.zeros((H, W))

    top_line, bottom_line, left_line, right_line = stage_lines


    m_top, b_top = slopeIntercept(top_line)
    m_bot, b_bot = slopeIntercept(bottom_line)
    m_left, b_left = slopeIntercept(left_line)
    m_right, b_right = slopeIntercept(right_line)

    if (m_left == 10 ** 9):
        left_x = left_line[0][0]
    else:
        left_x = -1

    if (m_right == 10 ** 9):
        right_x = right_line[0][0]
    else:
        right_x = -1

    for y in range(H):
        for x in range(W):
            if (y > m_top * x + b_top and
                y < m_bot * x + b_bot and
                ((left_x == -1 and x > (y - b_left)/m_left) or (left_x != -1 and x > left_x)) and
                ((right_x == -1 and x < (y - b_right)/m_right) or (right_x != -1 and x < right_x))):
                stage[y][x] = image[y][x]

                mask[y][x] = 1

    stage_crop = get_segment_crop(stage,tol=0, mask=mask)
    mask_crop = get_segment_crop(mask, mask=mask)
    return stage_crop, mask_crop

def getCeiling(image, stage_lines):
    ceiling = np.zeros(image.shape)
    H, W, _ = image.shape
    mask = np.zeros((H, W))

    top_line, _, _, _ = stage_lines
    m_top, b_top = slopeIntercept(top_line)

    for y in range(H):
        for x in range(W):
```

```
            if (y < m_top * x + b_top):
                ceiling[y][x] = image[y][x]
                mask[y][x] = 1

    ceiling_crop = get_segment_crop(ceiling,tol=0, mask=mask)
    mask_crop = get_segment_crop(mask, mask=mask)
    return ceiling_crop, mask_crop

def getLeftVertical(image, stage_lines):
    leftv = np.zeros(image.shape)
    H, W, _ = image.shape
    mask = np.zeros((H, W))

    top_line, bottom_line, left_line, _ = stage_lines


    m_top, b_top = slopeIntercept(top_line)
    m_bot, b_bot = slopeIntercept(bottom_line)
    m_left, b_left = slopeIntercept(left_line)

    if (m_left == 10 ** 9):
        left_x = left_line[0][0]
    else:
        left_x = -1

    for y in range(H):
        for x in range(W):
            if (y > m_top * x + b_top and
                        y < m_bot * x + b_bot and
                        ((left_x == -1 and x < (y - b_left)/m_left) or (left_x != -1 and x < left_x))):
                        leftv[y][x] = image[y][x]
                        mask[y][x] = 1

    leftv_crop = get_segment_crop(leftv,tol=0, mask=mask)
    mask_crop = get_segment_crop(mask, mask=mask)
    return leftv_crop, mask_crop

def getRightVertical(image, stage_lines):
    rightv = np.zeros(image.shape)
    H, W, _ = image.shape
    mask = np.zeros((H, W))

    top_line, bottom_line, _, right_line = stage_lines


    m_top, b_top = slopeIntercept(top_line)
    m_bot, b_bot = slopeIntercept(bottom_line)
    m_right, b_right = slopeIntercept(right_line)

    if (m_right == 10 ** 9):
        right_x = right_line[0][0]
    else:
        right_x = -1

    for y in range(H):
        for x in range(W):
            if (y > m_top * x + b_top and
                        y < m_bot * x + b_bot and
                        ((right_x == -1 and x > (y - b_right)/m_right) or (right_x != -1 and x > right
                        rightv[y][x] = image[y][x]

                        mask[y][x] = 1

    rightv_crop = get_segment_crop(rightv,tol=0, mask=mask)
    mask_crop = get_segment_crop(mask, mask=mask)
    return rightv_crop, mask_crop

aud_line, stage_lines = getFinalLines(topleft, topright, bottomleft, bottomright, image)
audience, audience_mask_crop = getAudience(image, aud_line)
stage, stage_mask_crop = getStage(image, stage_lines)
ceiling, ceiling_mask_crop = getCeiling(image, stage_lines)
leftvertical, lv_mask_crop = getLeftVertical(image, stage_lines)
rightvertical, rv_mask_crop = getRightVertical(image, stage_lines)
image_sections = [stage, ceiling, leftvertical, rightvertical, audience]
image_masks = [stage_mask_crop, ceiling_mask_crop, lv_mask_crop, rv_mask_crop, audience_mask_crop]
```

In [384]:
```
from numpy.linalg import svd

def computeHomography(pts1, pts2,normalization_func=None):
    '''
    Compute homography that maps from pts1 to pts2 using SVD. Normalization is optional.

    Input: pts1 and pts2 are 3xN matrices for N points in homogeneous
    coordinates.

    Output: H is a 3x3 matrix, such that pts2~=H*pts1
    '''
    _, N = pts1.shape
    A = np.zeros((2*N, 9))

    for i in range(N):
        u, v, w = pts1[:, i]
        up, vp, wp = pts2[:, i]
        A[2*i][0] = -u
        A[2*i][1] = -v
        A[2*i][2] = -1
        A[2*i][6] = u * up
        A[2*i][7] = v * up
        A[2*i][8] = up

        A[2*i+1][3] = -u
        A[2*i+1][4] = -v
        A[2*i+1][5] = -1
        A[2*i+1][6] = u * vp
        A[2*i+1][7] = v * vp
        A[2*i+1][8] = vp

    [U, S, Vh] = svd(A)
    return Vh[-1].reshape(3, 3)

def transformImage(image, mask, border, sectionIdx):
    H, W = mask.shape
    border_corners = cv2.goodFeaturesToTrack(border.astype(np.float32), maxCorners=4, qualityLevel=0.1
    corners = []
    for border_corner in border_corners:
        x = border_corner[0][0]
        y = border_corner[0][1]
        corners.append((x-25, y-25))

    pts1 = np.ones((4, 3))
    dists = [9999.0] * 4
    pts2 = np.array([
        [0.0, 0.0, 1.0],
        [W-1, 0.0, 1.0],
        [0.0, H-1, 1.0],
        [H-1, H-1, 1.0]
    ])

    for pt_idx in range(4):
        dest_pt = pts2[pt_idx]
        closest_corner = None

        for corner in corners:
            # calculate distance
            x1, y1 = corner
            x2, y2, _ = dest_pt

            dist = np.sqrt((x1 - x2)**2 + (y1 - y2)**2)

            if (dist < dists[pt_idx]):
                closest_corner = corner
                dists[pt_idx] = dist

        pts1[pt_idx][0] = closest_corner[0]
        pts1[pt_idx][1] = closest_corner[1]

    Homography = computeHomography(pts1.T, pts2.T)
    Tr = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])

    warped_image = cv2.warpPerspective(image, Tr @ Homography, (W,H))
    plt.imshow(warped_image)
```

```
    titles = ["Transformed Stage", "Transformed Ceiling", "Transformed Left Vertical", "Transformed Ri
              "Transformed Audience"]
    plt.axis("off")
    plt.title(titles[sectionIdx])
    plt.show()

    return warped_image

warped_images = []

for sectionIdx in range(len(image_sections)):
    section = image_sections[sectionIdx]
    mask = image_masks[sectionIdx]
    resized_image = cv2.resize(section, (500, 500),
                        interpolation = cv2.INTER_LINEAR)
    border = np.zeros((550, 550))
    resized_mask = cv2.resize(mask, (500, 500),
                        interpolation = cv2.INTER_LINEAR)
    border[25:525,25:525] = resized_mask

    warped_images.append(transformImage(resized_image, resized_mask, border, sectionIdx))
```

In [388]:
```
image_names = ["stage.jpg", "ceiling.jpg", "leftvertical.jpg", "rightvertical.jpg", "audience.jpg"]
for i in range(5):
    cv2.imwrite(image_names[i], 255*cv2.cvtColor(warped_images[i].astype('float32'), cv2.COLOR_BGR2RGB
```

In [ ]:
```

```