Advanced Artificial Vision Assignment #1

Russell Sammut-Bonnici, Jesmar Farruiga

November 2019

1 Paper 1: Intra-object segmentation using depth information

1.1 Replication

In **objSegmentation**, grabcut, thresholding and bitwise and operations were utilized for object segmentation on texture image img_t and depth image img_d with respect to the user selected region of interest ROI. After grabcutting img_d, a call was made to **histMinMax** for finding the deepest layer in the depth mask returned by grab cut. That way, the furthest depth/background was achieved for threshold application. On thresholding, The foreground became further extracted from the background, by making pixels infront of the background white and pixels in the background black. This 'stensil'-like mask refined segmentation between the object and the background. Finally, it was applied to img_t and img_d and their segmentations img_tseg and img_dseg were returned.

In **histMinMax** numpy was used for histogram calculation because calculate histogram method from cv2 had some unclear pre-processing steps that would incorrectly count too little shades for layers, whereas numpy would calculate the expected amount. After histogram numpy calculation, all the non-zero shades were stored in a list called NZshades. The min (furthest) and max (closest) shade were found by finding the first and last element in NZshades respectively. The min was used for the thresholding described in **objSegmentation** above.

In **intraObjSegmentation** intra-object segmentation was performed to obtain a list of layers L_{layers}. This was done by creating a mask out of each shade layer, applying it with a bitwise and to img_t, to result with the texture segmentation of that layer to append to L_{layers}. A layer count of 37 was achieved.

printLayers was created for presentation purposes and simply outputs and saves the intra-object segmented layers in L_{layers} both visually and textually. A waitKey of 250 was used to make the execution initiate some animation-of-sorts, where a focus would travel from the layer. The images written from this function were used to make two gifs. **texture.gif** for L_{layers} and **depth.gif** for L_{layers_d} (L_{layers_d} is further explained in section 1.3).

1.2 Replicated results

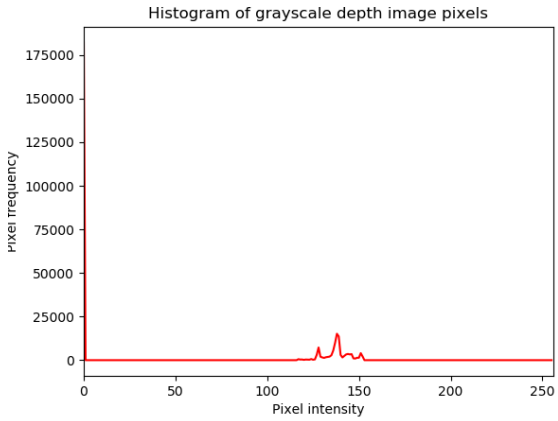
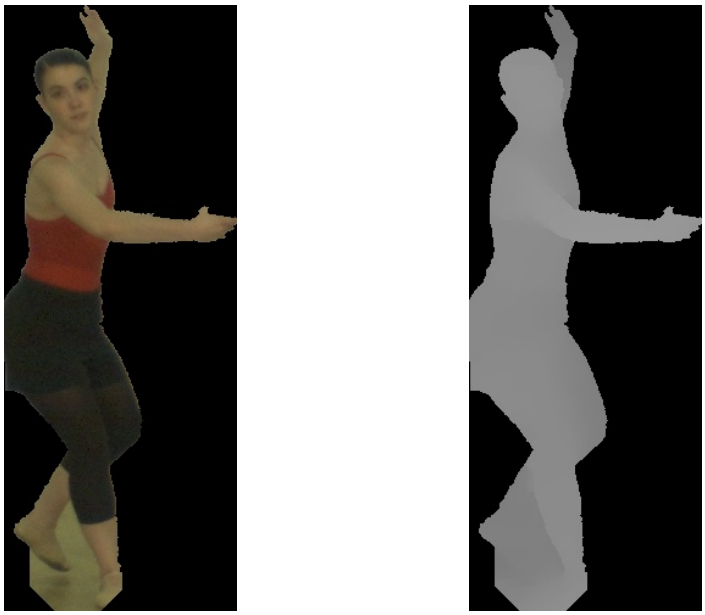
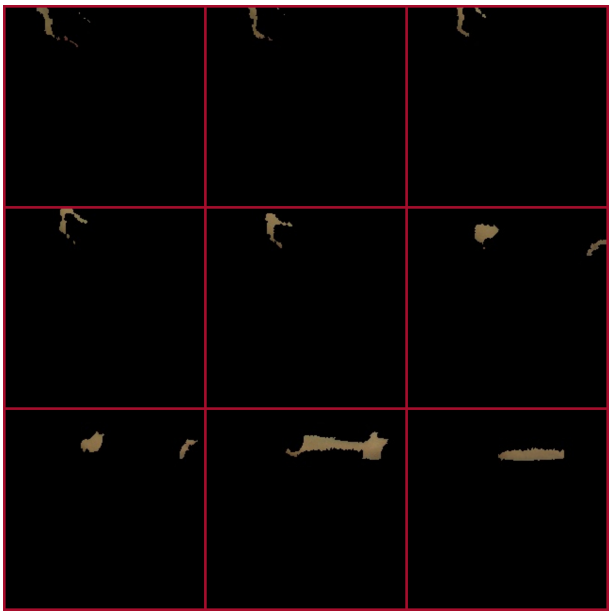


Figure 1

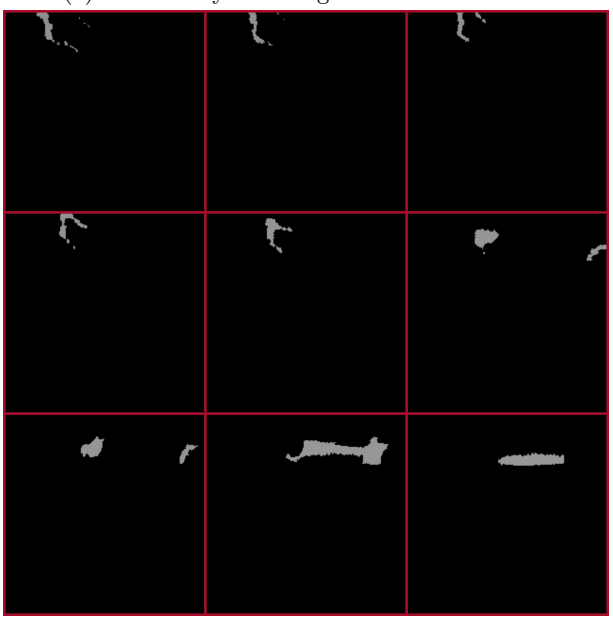


(a) Texture (b) Depth

Figure 2: Object segmentation results



(a) Texture layers for right arm



(b) Depth layers for right arm

Figure 3: Intra-Object segmentation results, Layer-count: 37

1.3 Variations

Method **selectLayers** was the most basic to implement. Taking start and end as parameters it refines L_{layers} by creating a new list from elements in their indicated range. A try catch was used just in case the user would try to use a start or end parameter that was illogically out of bounds from L_{layers}.

getSoftHSVLayers (Algorithm 1) returns two lists. One list called $L_{brightness}$ for the average brightness per layer and another called L_{hsv} for the average hsv per layer. The pseudocode can be seen below in Algorithnm 1. Note that the average brightness and average hsv value per layer are equal. This was useful for comparing and testing that the correct method was used for brightness extraction. The algorithm works by firstly finding the average hsv and brightness of each layer. This was done by iterating through all the pixels in the current layer, and summating the values to sum variables. For each pixel it added to the summation variables, the pixel counter $count_{pxl}$ was incremented. An if condition was used to avoid considering zero-pixels that indicate background pixels. After the sums finished computing, the averages $avg_{brightness}$ and avg_{hsv} were calculated by dividing the sums by $count_pxl$. The layer averages were appended to their respective lists $L_{brightness}$ and L_{hsv}. This was repeated for each layer.

Algorithm 1 getSoftHSVLayers

Require: L_{layers}
Ensure: $L_{brightness}$, L_{hsv}
  $L_{brightness}, L_{hsv} \leftarrow [], []$
  for `layer in` L_{layers} **do**
    $brightness_{sum} \leftarrow 0$
    $hsv_{sum} \leftarrow [0, 0, 0]$
    $count_{pxl} \leftarrow 0$
    $hsv_{layer} \leftarrow convertHSV(layer)$
    for `each` pxl `at` $loc[y, x]$ `in layer` **do**
      $red \leftarrow pxl.redChannel$
      $green \leftarrow pxl.greenChannel$
      $blue \leftarrow pxl.blueChannel$
      if $red, green, blue \neq black$ **then**
        $brightness_{sum} \leftarrow max(red, green, blue)$
        $hsv_{sum} \leftarrow hsv_{sum} + hsv_{layer}.pxl[y, x]$
        $count_{pxl} \leftarrow count_{pxl} + 1$
      end if
    end for
    $avg_{brightness} \leftarrow brightness_{sum}/count_{pxl}$
    $avg_{hsv} \leftarrow hsv_{sum}/count_{pxl}$
    $L_{brightness}.append(avg_{brightness})$
    $L_{hsv}.append(avg_{hsv})$
  end for
  return $L_{brightness}, L_{hsv}$

intraObjSegmentationDepth works in the same way as **intraObjSegmentation** from Section 1.1 except instead of returning a list of layers with respect to texture it returns layers with respect to the depth. This was done by simply modifying the code to apply the mask to img_{dseg} rather than img_{tseg}. Used for the texture manipulation methods.

Three texture manipulation methods were implemented:

- **manipByAvgHSV** (Algorithm 2)
- **manipByAvgBright** (Algorithm 3)
- **manipByRandCol** (Algorithm 4)

All of these shared a similar pipeline of manipulating the texture of each layer, by finding non-zero pixels in each layer, painting them, then painting the original segmented image img_t pixel by pixel with the current manipulated layer. Note that for finding the zero pixels depth list L_{layers_d} was used instead of L_{layers} because in texture it found some zero pixels in the ballerina's hair and pants, making the program misclassify foreground as background pixels. Using the layers of the depth image avoided this because here pixels get brighter and brighter the more in the foreground they are, irrespective of the texture colour of the pants and hair.
The differences between the manipulation algorithms was that they would paint with respect to different styles:

- **manipByAvgHSV** uses the average hsv determined in **getSoftHSVLayers** and colours each layer with their average hue, saturation and brightness value.
- **manipByBright** colours each layer by brightness also determined in **getSoftHSVLayers**, resulting in some depth-look-alike image.
- **manipByRandCol** colours with respect to random colour. The three settings available are random ranges in the blue, green or red palette. These 'settings' in the code were implemented by commenting (to switch just comment the current one and uncomment the one you want). As the code is presented it is set to the red palette, resulting in a randomly generated pink shading per layer.

resizePercentage helped enforce the DRY-principle as it was called after each texture manipulation method. It simply resizes an image with respect to a percentage scale. Example, in this case 200% increases the scale by 2x of the original. The outputs were resized for better observation on how the layers deviate in texture from each other.

Algorithm 2 manipByAvgHSV

Require: $img_t, L_{layers}, L_{layers_d}, L_{hsv}$
Ensure: img_t
  for $layer, layer_d$ `in` L_{layers}, L_{layers_d} **do**
    $layer \leftarrow converHSV(layer)$
    for `each` pxl `at` $loc[y, x]$ `in layer` **do**
      $value \leftarrow layer_d[y, x].valChannel$
      $sat \leftarrow layer_d[y, x].satChannel$
      $hue \leftarrow layer_d[y, x].hueChannel$
      if $val, sat, hue \neq black$ **then**
        $pxl \leftarrow L_{hsv}$`[of curr layer]`
        $img_t[y, x] \leftarrow pxl$
      end if
    end for
  end for
  return img_t

Algorithm 3 manipByAvgBright

Require: $img_t, L_{layers}, L_{layers_d}, L_{brightness}$
Ensure: img_t
  for $layer, layer_d$ `in` L_{layers}, L_{layers_d} **do**
    $layer \leftarrow converHSV(layer)$
    for `each` pxl `at` $loc[y, x]$ `in layer` **do**
      $value \leftarrow layer_d[y, x].valChannel$
      $sat \leftarrow layer_d[y, x].satChannel$
      $hue \leftarrow layer_d[y, x].hueChannel$
      if $val, sat, hue \neq black$ **then**
        $pxl \leftarrow L_{brightness}$`[of curr layer]`
        $img_t[y, x] \leftarrow pxl$
      end if
    end for
  end for
  return img_t

Algorithm 4 manipByRandCol

Require: $img_t, L_{layers}, L_{layers_d}$
Ensure: img_t
  for $layer, layer_d$ `in` L_{layers}, L_{layers_d} **do**
    $r, g, b \leftarrow rand.$*[prominence in red, green or blue]*
    for `each` pxl `at` $loc[y, x]$ `in layer` **do**
      $blue \leftarrow layer_d[y, x].blueChannel$
      $green \leftarrow layer_d[y, x].greenChannel$
      $red \leftarrow layer_d[y, x].redChannel$
      if $blue, green, red \neq black$ **then**
        $pxl \leftarrow (b, g, r)$
        $img_t[y, x] \leftarrow pxl$
      end if
    end for
  end for
  return img_t

1.4 Variation results

(a) HSV

(b) Brightness

(c) Random

Figure 4: Texture manipulation results

1.5 Future improvements

- The program is expensive to run. It takes a while to finish executing. A way to improve this would be by making use of higher level methods. Some ideas:
  - Thresholding could be used with cv2's inbuilt thresholding procedure rather than performing it manually
  - Numpy functions can be utilised within the manipulation methods instead of going through the images manually pixel by pixel.
- Grabcut with rectanglar selection would sometimes extract some parts of the background as the foreground. To avoid this we could implement free draw selection instead of rectangular selection to refine the selection. This would work by than converting then converting the free-hand drawing to a circular mask shape. This could then be fed into grab cut as a mask and instead of $cv2.GC\_INIT\_W\_RECT$ as the third argument it would be $cv2.GC\_INIT\_WITH\_MASK$.
- Make a parameter for **manipByRandColor** that indicates whether the random palette should be in red, green or blue. Use if conditions with respect to that parameter to switch case palettes.
- Error handling for manipulation methods if the layer lists L_{layers} and L_{layers_d} inputted are not of the same size.
- Make **printLayers** delete the previously saved layer images in the layers folder before writing. This is because currently old files are just overwritten with every new section, and if the ROI selected in this program instance is smaller than the ROI of the previous execution, then not all the old files are overwritten. This could lead to generating slightly misleading gifs. Which actually was the case with **texture.gif** and **depth.gif**. The layers of this actual execution was 37 but according to the layers folder and gifs, it was 56. This is because the layer images after the 37 count were old files still kept from a previous execution (where a larger ROI was selected).

2 Paper 2: Efficient object selection using depth and texture information

2.1 Replication

As the paper states the images need to first be blurred for a better result, as such, a Gaussian blur was used. This blur was chosen simply because it yielded better smoothing compared to other blur functions used.

The target function relied on user input, so the first function that needed to be implemented was **cv2.setMouseCallback** which takes the target image and a function as parameters. The function to be used as a parameter (**mouse**) was required to have four parameters: event, x, y, flags, param. It was made so that only the x, y variables were required simply because these variables will contain the coordinates of where the mouse was double clicked in the image. So here the target function is called, stores its result in a temporary variable, then displays the result.

For the target function **dtObj** two other methods were implemented: **patch** and **SSD**. **patch** takes the coordinates from the mouse click, an image matrix and the size of the patch (patch is a square so only one side is required), then stores half the size of the patch and returns a patch from the image matrix using ranges from x-temp to x+temp and y-temp to y+temp. The method **SSD** takes two matrices, then using numpy, sums up their squared difference, and returns the result.

Finally, to show the result of **dtObj**, two methods were created: **plot** and **comm**. **comm** compares two lists, and returns a list with all the common elements. This was done to compare the result of the texture image and depth image, and return the coordinates that match. **plot** takes two lists and an image matrix as parameters. Basically the function uses **comm** to get the list of points mentioned before, then sets all those points in the image matrix (in our case, the texture image) to red, and shows the result.

Algorithm 5 Replication

$img1 \leftarrow read(TextureImage)$
$img2 \leftarrow read(DepthImage)$
$T \leftarrow blur(img1)$
$D \leftarrow blur(img2)$
$treshD \leftarrow 3000$
$treshT \leftarrow 3000$
$s \leftarrow 6$

$show(im1)$

if $mouse == DoubleClicked$ **then**
  $x, y \leftarrow getMouseCoordinate$
  $Ld, Lt \leftarrow dtObj(x, y, treshT, treshD, T, D, s)$
  $commCo \leftarrow getCommPoints(Ld, Lt)$
  for i `in` $range(commCo)$ **do**
    `Where` $im1 = commCo[i]$ `setTo` $[0, 0, 255]$
  end for
end if

$show(im1)$

2.2 Replicated results

Figure 5: Original Image

Figure 6: Single Click Result

In this example, the man's vest was selected and the algorithm selects the entire vest for the most part, and some of his hair. Since his hair and vest have a similar texture and depth they were both selected, but this can be improved by further tweaking the threshold values.

2.3 Variations

The Replication was done with the user making one selection in mind, so in order to allow multiple selection, the previously mentioned functions had to be restructured.

Apart from global variables used before, two more were added, $p1$ and $p2$, which are both lists that will contain all the x and y coordinates where the user clicked. The function **mouse** in this case, just appends the x, y coordinates to $p1$ and $p2$ respectively and nothing else. And so having done that, unlike the Replication, the other functions are called outside, with the difference being that the functions **dtObj** and **plot** are called in a loop that iterates as many times as the size of $p1$ (which should be the same size as $p2$)

To optimise code, at first, we tried to import the same functions as the ones used in the Replications, but this caused the script to work the exact same way as the Replication (allowing the user to make only one selection). That is the reason why this approach was chosen.

Algorithm 6 Variation

$img1 \leftarrow read(TextureImage)$
$img2 \leftarrow read(DepthImage)$
$T \leftarrow blur(img1)$
$D \leftarrow blur(img2)$
$treshD \leftarrow 3000$
$treshT \leftarrow 3000$
$s \leftarrow 6$
$p1 \leftarrow []$
$p2 \leftarrow []$

$show(im1)$

while $button! = pressed$ **do**
  if $mouse == DoubleClicked$ **then**
    $x, y \leftarrow getMouseCoordinate$
    $p1.append(x)$
    $p2.append(y)$
  end if
end while

for i `in` $range(p1)$ **do**
  $Ld, Lt \leftarrow dtObj(p1[i], p2[i], treshT, treshD, T, D, s)$
  $commCo \leftarrow getCommPoints(Ld, Lt)$
  for j `in` $range(commCo)$ **do**
    `Where` $im1 = commCo[j]$ `setTo` $[0, 0, 255]$
  end for
end for

$show(im1)$

2.4 Variation results

Figure 7: Original Image

Figure 8: Multiple Selections

In this example, the same image as before was used but this time the vest, the dancer's arm and shirt were selected. Once again the same thresholds were used, so the hair was selected as before, and a bit of the dancer's skirt.

2.5 Future Improvements

- The patch selection does require a bit of refinement. Experimenting with the thresholds more could give a finer patch selection. Using a different method for blurring on the original images may have improved the patch selection also.
- The script for replication could have been restructured better in order to be used by other scripts, instead of copying and modifying the entire code (as can be seen in the variation script).
- Add some green markers on the pictures to identify where the user clicked.