UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
SCHOOL OF COMPUTER SCIENCE & APPLIED MATHEMATICS

COMS4036A & COMS7050A
Computer Vision

Devon Jarvis, Nathan Michlo, Dr Richard Klein

Lab 7: Affine Transformations

Overview

In the previous lab you worked with shape models so that we could extract and match the edges of the puzzle pieces. Now that we have the graph representing the puzzle piece connections, we are finally able to solve the puzzle!

Instructions

The goal of this lab is to extract the puzzle pieces from their own individual images and insert them correctly into the puzzle image or "canvas". To start off with, you have been given a python file "classes.py" containing three classes, namely the Edge, Piece and Puzzle classes. In this lab we will be working with the Piece class.

1 Affine Transformations

It is recommended that before you start you have had a thorough look at classes.py and familiarize yourself with the classes. Take note of the comments which describe the intended purpose of each variable and function. Some supplemental functions which display information about the object have also been implemented. We will be working with the `Piece.insert()` function in Questions 1, 2, 4 and 5. Write all the code from these questions in order in `Piece.insert()`. For these questions, the `Piece` class is also what I will be referring to as `self`.

1. To begin with we have to start the `self.insert()` function by updating the `self.piece_type` attribute of the piece being inserted. This is necessary because at the start of the insertion process `self.piece_type` is `None`. The manner that we want to insert the puzzle piece into the canvas depends on the number of connected edges **already** inserted into the canvas that a piece has. To this end, iterate over the edges in `self.edge_list` and count the number of occurrences of `edge.connected_edge.parent_piece.inserted == True`. If the number of occurrences is in $[0, 1, 2]$ then set `self.piece_type` correspondingly from [`'corner'`, `'edge'`, `'interior'`]. If the number of occurrences is greater than 2 raise an error.

2. We can now proceed with our first insertion case, inserting a corner piece. Thus, begin an if-elif-else block by checking if `self.piece_type == 'corner'`. To do the affine transforms we will be using the OpenCV functions `cv2.getAffineTransform(·)` and `cv2.warpAffine(·)`.

   - `cv2.getAffineTransform(pts_src, pts_dst)`: This function accepts two parameters. `pts_src` and `pts_dst` will both be lists containing 3 sets of coordinates (remember that OpenCV describes coordinates in column-row (x, y) order as opposed to the natural row-column (y, x) order you would usually use to index an array or matrix). `pts_src` contains 3 source coordinates on the original image and `pts_dst` contains their corresponding destination coordinates

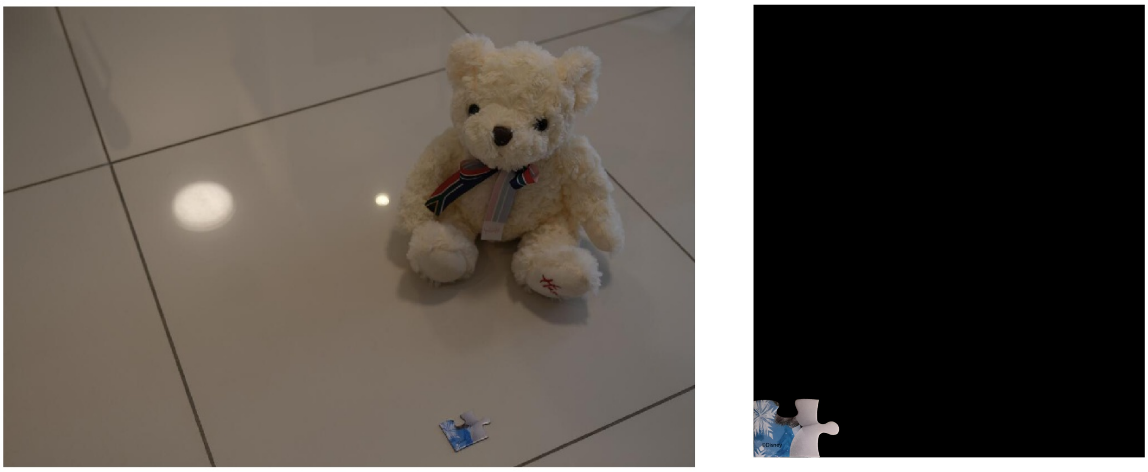

Figure 1: Inserting the corner piece

on the canvas. Thus `pts_src[i]` is going to be the coordinates of a corner of our puzzle piece and maps to `pts_dst[i]` which is where we want the corner to go in the canvas. This function returns an Affine Transform `M`.

- `cv2.warpAffine(image, transform, output_dimensions)`: This function has 3 parameters. `image` is the initial image being transformed, `transform` is the Affine Transform matrix (`M` above) and `output_dimensions` is the dimensionality of the output image and in our case is the dimensionality (column-row dimensionality) of the canvas.

Loop through `self.edge_list` of the corner piece and find the two flat edges (lets call them `first_edge` and `second_edge` where `second_edge` is anti-clockwise of `first_edge`). `first_edge.point2` should be the same coordinates as `second_edge.point1`. Map this point to the bottom left coordinate of the canvas (so append `first_edge.point2` to the list `pts_src` and append the bottom left coordinates of the canvas to `pts_dst`. Again be careful because `first_edge.point2` is in row-column coordinates but we need to append them in column-row coordinates). Map `first_edge.point1` to lie along the left edge of the canvas and map `second_edge.point2` to lie along the bottom edge of the canvas. In both cases you must preserve the lengths of the piece edges. In other words do not stretch the corner piece. We just drop it into the bottom left corner with the necessary rotation. Use this mapping with `cv2.getAffineTransform(·)` to get the matrix `M`. Then use `cv2.warpAffine(·)` on `self.image` with transform `M` and store the output in `self.dst`. Also apply the transform to `self.mask` and store the output in `self.mask`. This is to ensure we transform the mask from the original piece coordinates to the canvas coordinates. Then call `self.update_edges(M)` (we will go implement this function in a little bit). Lastly we now have `self.dst` which is our puzzle piece in canvas coordinates and all we need to do now is "tattoo" the correct pixels of `self.dst` onto the canvas. Fortunately we have our mask in canvas coordinates. Use the mask to transfer **only** the puzzle piece pixels onto the canvas by blending the two. **Hint**: `canvas = mask*img + (1-mask)*canvas`. This pseudo-code will give smooth results as the mask is no longer an exact binary mask after the affine warp. The position of the piece in `self.dst` will be the same as in the canvas, so it is only a matter of transferring the correct slice of `self.dst` onto the exact same position on the canvas. That's all there is to inserting the puzzle piece! Find the transform, then warp the image and the mask, update the edges and then tattoo the puzzle piece onto the canvas. Setting up the arrays `pts_src` and `pts_dst` requires some care though. An example of the canvas after inserting the corner piece can be seen in Figure 1.

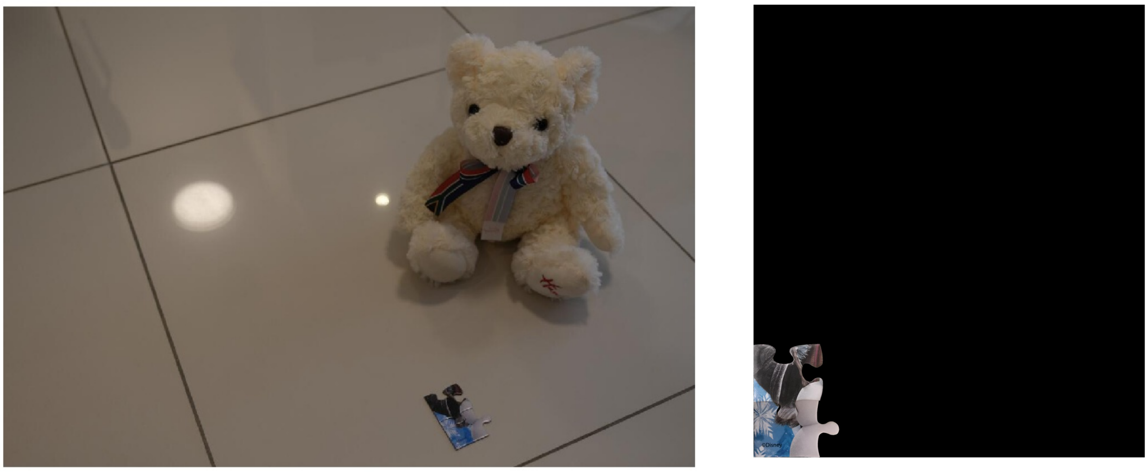

Figure 2: Inserting an edge piece

3. We now need to implement the `self.update_edges(self, transform)` function which we have been using in the `self.insert()` function. This is because inserting edge and interior pieces require the corner piece's edges to be in canvas coordinates. This function accepts one parameter, the Affine Transformation used to insert puzzle pieces into the canvas and maps from piece coordinates to the canvas coordinates. This means we can also use the transform to convert the puzzle piece's corner and edge information to the canvas coordinates. There are three things to note. Firstly we have stored our corners in row-column coordinates but the transform was implemented with column-row coordinates (so you need to flip the coordinates before doing the multiplication and flip them back afterwards). Secondly we need to append 1 to the coordinates (**Hint**: `np.append(·)` or `np.hstack(·)`) and right multiply by the transform (**Hint**: `np.dot(·)`). Lastly for the right multiplication to work we have to transpose the transform. Update all corners of the "`self`" piece being inserted using the transform. Also update `point1` and `point2` for every edge of the puzzle piece.

4. The next insertion case we will deal with is the 'interior' piece type. We are skipping the 'edge' type for now as it is the most tricky, however, when building the puzzle we will usually insert edge pieces before the neighbouring interior pieces. Add this case to your if-elif-else block. For this case we will begin by looping through the edges of `self.edge_list`. When we find an edge that has `edge.connected_edge` is not **`None`** and `edge.connected_edge.parent_piece.inserted == True` we then check if `edge.point1` is already in `pts_src` (we can't insert redundant points). If it isn't then we append `edge.point1` to `pts_src` and append `edge.connected_edge.point2` to `pts_dst`. Likewise check if `edge.point2` is in `pts_src` and if not insert it and `point1` of its connected edge into `pts_src` and `pts_dst` respectively. By the end of the loop over the piece's edges we will have set up `pts_src` and `pts_dst` and the rest of the process is the same as for Question 2 above. Get the Affine Transform `M` using `pts_src` and `pts_dst`, transform the piece's image and mask using `M`, call `self.update_edges(M)` and then tattoo the piece from `self.dst` onto the canvas.

5. We now get to the final case of the 'edge' piece type. Add this case to your if-elif-else block. To obtain an Affine Transform we need 3 points from the source image to map onto 3 points in canvas coordinates. Unfortunately for an edge piece we will only ever have one edge which is connected to another piece that is already inserted in the canvas (lets call it `third_edge`). This only gives us 2 points in canvas coordinates. We can, however, still determine the third canvas coordinate. To do this we will look at how much we have to scale `third_edge` to fit it into the canvas and use the same scaling on all the other

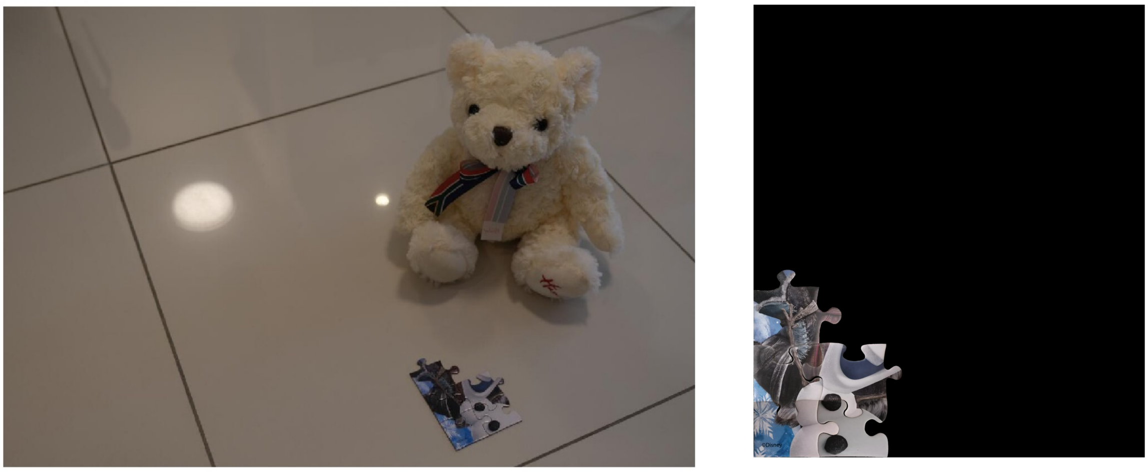

Figure 3: Inserting an interior piece

edges of the piece. This information, along with the knowledge that the piece's one point must lie along the edge of the canvas, can be used to determine the third point in the canvas coordinates. Now that we know where this case is heading, we can begin.
This starts the same as the 'interior' piece case where we loop over `self.edge_list` to find an edge which is not **`None`** and is connected to a piece that has already been inserted into the canvas. When we find such an edge we add its points to `pts_src` and we add the points of its connected edge to `pts_dst`. We now need to calculate the amount this edge has to be scaled to be entered into the canvas. To do this calculate the norm **Hint**: `np.linalg.norm(·)` of this edge (distance between the edge's two points) and the norm of its connected edge already in the canvas. Call these two norms `orig_norm` and `canvas_norm` respectively, we then have our scaling ratio as `ratio = orig_norm/canvas_norm`. Once we reach this point we just need to calculate the third canvas coordinates point. There are two cases:

- **`if`** `(pts_dst[0][0]-pts_dst[1][0]) > (pts_dst[0][1]-pts_dst[1][1]):` then we know we are inserting an edge piece which lies along the bottom of the puzzle. Once again loop through `self.edge_list` and find the next edge anti-clockwise of the known edge (the one that will lie along the bottom of the puzzle, lets call it `fourth_edge`). Append point2 of `fourth_edge` to `pts_src`. Then calculate the norm of `fourth_edge`, call it `edge_norm`. Lastly append `[pts_dst[1][0]+int(ratio*edge_norm),pts_dst[1][1]]` to `pts_dst`. We now have 3 canvas coordinates can can add the puzzle piece to the canvas in the same manner as all cases above. Do this now.
- **`else`** we know that we are inserting an edge piece which lies along the left side of the puzzle. Loop through `self.edge_list` and find the edge which come before the known edge (again looping over the edges in anti-clockwise order, call this edge `fifth_edge`). We know that `point1` of `fifth_edge` needs to lie on the left edge of the canvas. Append `point1` of `fifth_edge` to `pts_src` and calculate its norm. In a similar manner to the first edge-piece case, use this norm to add the third canvas coordinate to `pts_dst` and add this piece to the canvas in the usual way.

That's the last case of the insert function! It is safe to add an **`else`** to your if-elif-else block which catches piece types which aren't valid or accounted for. Figures 2 and 3 show examples of the canvas after adding the first edge and interior pieces respectively.

Figure 4: Finished puzzle

6. We can now do the final part of the implementation. For this part we will move over to the "run.py" file. In the file we first create a Puzzle object using `puzzle = Puzzle(MATCH_IMGS)` which just creates a Puzzle object, loads the images into `Puzzle.pieces` and connects the edges of the puzzle pieces (the ground-truth solution of Lab 6). We will be implementing the puzzle solver using a Breadth-First Search (BFS) over the puzzle graph. We can start with any corner piece, insert it in the bottom left corner of the canvas, and then search out from the bottom left to top right corners of the canvas adding pieces to the puzzle. For simplicity, we start off with `corner_piece = puzzle.pieces[3]` and from "classes.py" we have `canvas = np.zeros((800,700,3))`. We also start off by inserting the corner piece into the canvas in "run.py" as an example of how to use the classes. Now implement the rest of the BFS. The Piece class has a generator function `Piece.return_edge()` which you might find useful for looping over edges in the BFS (you can think of it as a function with memory, so for-loops start where they left off last time the function was called) but make sure you can escape the loop! Lastly, run your BFS and plot the final canvas. An example of the final canvas can be seen in Figure 4 and Tino's reaction in Figure 5.

Figure 5: Tino is free!

2 Submission

Submit your code which builds the full puzzle and displays the canvas at the end. Also submit images of your canvas after each of the first 5 insertions and of each of the last 5 insertions (so 10 images including the full puzzle image).