Advanced Topics in Computer Vision and Deep Learning (EEEM071)

Week 2: Digital Image Warping – part 1/2

Dr Xiatian Zhu

Introduction

In this lab you will experiment with digital image warping techniques, putting the material from Week 2 lectures into practice. You will be using MATLAB for this work, and should download the lab code from SurreyLearn.

Please visit the **lab sheets section on SurreyLearn** and download **EEEM071_labcode_weeks2-4.zip** into your home space.

You should ensure you attended the lectures on digital image warping in Week 2 before attempting this worksheet. If you didn't then you should watch on Panopto first.

1. Getting Started

Unzip the lab code into a folder in your home space. Open up a terminal window using Ctrl-Alt-T.

Assuming you downloaded the file into the top folder of your homespace, type

```
unzip EEEM071_labcode_weeks2-4.zip
```

This will create a folder EEEM071 and within that, a subfolder warping. Start MATLAB and ensure that this folder is added to your Matlab search path (Use File -> Set Path.. -> Add with subfolders and pick the EEEM071 folder). If you are unsure how to do this then check with the lab tutors. If MATLAB is complaining that it cannot find functions when you perform these lab exercises it is probably because your path is not set correctly. You can also try typing directly within MATLAB:

```
addpath(genpath('~/EEEM071'))
```

```
cd ~/EEEM071
```

2. Warping using Forward vs. Backward mapping

Run the warping demo ipv_warp_forward which implements basic forward mapping technique for warping – as supplied, the code will rotate the image 45 degrees about its centre.

```
ipv_warp_forward
```

Now run the ipv_warp_backward which implements the same warp, but using the backward mapping technique.

```
ipv_warp_backward
```

Take a look at the code for both functions – you will see that it is nearly identical. E.g. you can type

```
edit ipv_warp_forward.m
```

- **Lines 18-23** load and normalise the image
- **Lines 25-35** set up a 3x3 matrix transformation M to rotate the image 45 degrees anti-clockwise about its centre.
- **Line 38** sets up a blank RGB output image of the same size as the input (pixels are all zero, i.e. black).

The rest of the code is a loop. In the case of forward mapping, we are looping through every pixel in the input image and applying M to figure out where that pixel lands up in the output image.

It is quite possible that not all the pixels will be landed on in the output image, and this is why we see the black holes speckled across the output.

Now look at the backward mapping code – we are again looping, but through every pixel in the output image and applying the INVERSE of M to figure out where the pixel should have 'come from'. In order words, we are working out from where to sample the input image for every output pixel. Since by definition we are visiting every output pixel, the black holes we encountered during forward mapping won't occur.

Notice how the code differs only slightly. In the backward mapping…

- **Line 36** – inverts M so that the loop is applying INVERSE of M i.e. the backward transformation
- **Lines 43-44** – p refers to the coordinates of the pixel in the input image. q refers to the coordinates of the pixel in the output image. Note how we are computing q from p in forward mapping, but p from q in the backward mapping. In line 46-47 the coordinates u and v are derived from p and from q in the forward and backward mapping cases respectively.
- **Line 52** - Out of bounds checking occurs to ensure that we don't try to colour in pixels outside the edge of the image.

Now try to implement some alternative image transformations using ipv_warp_backward. Can you generate a scaling up, a scaling down, or an alternative rotation?

3. Getting the full picture

One problem with the current code is that parts of the image get 'lost' during the warp since they fall outside the image boundaries. For example, the rotation anti-clockwise by 45 degrees causes the four corners of the image to be missed in the output image.

We will now **edit** the **ipv_warp_backward** function to correct this behaviour.

The easiest way to compensate for part of the image loss, is simply to make the output image larger. This will cater for two of the missing corners in this example – where we were simply not iterating over a large enough image to capture that missing content.

We will figure out the correct size of output image by calculating the position of the four corners of the input image, under the matrix transformation M.

After line 36 (inverting the matrix M) add the following:

```
corners=[0 W 0 W; 0 0 H H];
corners(3,:)=1;
```

This creates a variable **corners** containing the four corners of the input image. The coordinates of the corners are in homogeneous form, in each of the four columns of the matrix.

Now warp the corner positions using the matrix M:

```
warpedcorners = M* corners;
```

To figure out how big to make the output image, let's find the largest x and y coordinate (call this maxx and maxy) in the warped corners

```
maxx=max(warpedcorners(1,:))
maxy=max(warpedcorners(2,:))
```

Let's make the width and height of the output image equal to this

```
newW=ceil(maxx)
newH=ceil(maxy)
```

We should change the command creating the output image that was on line 38 to use these **newH** and **newW** variables.

```
img_out=zeros(newH,newW,3);
```

and change the loop so it reads for **x=1:newW** and **for y=1:newH** in order to iterate over the full output image.

Try running the code – you will obtain a larger output image that correctly contains two of the previously missing corners. But what about the other two? These corners have fallen in the negative part of the output coordinate domain – we should shift (translate) the entire image by a certain amount to get it fully into the positive domain. But how much?

Let's use our corners variable to compute the minimum X and Y coordinate of the transformed corners. Immediately after computing **warpedcorners** you should add:

```
minx=min(warpedcorners(1,:))
miny=min(warpedcorners(2,:))
```

We should add a shift of **–minx** and **–miny** to the transformation M such that the transformed corners end up being positive. Let's create a matrix transformation for that translation, in homogeneous form. Do it immediately after computing **minx** and **miny**

```
T=[1 0 -minx ; ...
   0 1 -miny ; ...
   0 0 1];
```

We want to apply this translation as a final step after the warp – so add after these lines a premultiplication of T on the existing transformation M. Note that you should NOT have inverted M at this point!

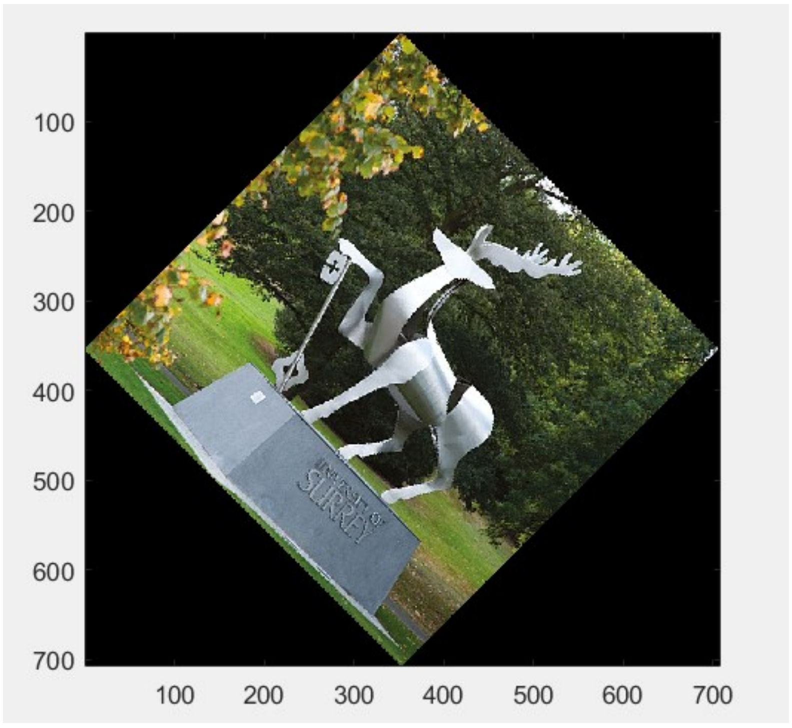
```
M=T*M;
```

Finally we need to calculate the correct new width and height of the image taking this into account

```
newW=ceil(maxx-minx+1);
newH=ceil(maxy-miny+1);
```

Run your code to verify that the image now warps without loss of any corners. Repeat your experiments trying different rotations and other affine transformations to verify that the code correctly compensates by pushing the output image into the positive domain for any transformation.

If this did not work for you, then take a look at the file ipv_warp_backward_with_bounds.m where the above steps have been performed on ipv_warp_backward.m. Consider this the 'solution' to this labsheet but please do attempt to step through this sheet properly.



END