asg03: implementing the transformation!

In general: use 2 images, the source image & the destination image, i.e.,

```
ppm *src, *dst;
```

pointers to two images.

src gets filled in from file:

```
FILE *ifile = NULL;
ifile = fopen("tigerOne.ppm", "rb");
```

read, binary ⟶

```
// read header of open file
ppm_header(ifile, &w, &h, &max);
// allocate mem for image
src = ppm_alloc(h, w, max);
// read image
ppm_read(ifile, src);
   NOTE: pointer          NOTE: pointer
// copy image
dst = ppm_copy(src);
```

- Several important observation

`dst = ppm_copy (src);`

dst image
(PPM *)
is a new
image returned
by ppm_copy

src image
(PPM *)
has all the
info that
ppm_copy
uses to
create the
copy

∴ ppm_copy
must declare & allocate
this internally

```
// now we have dst, write it
ppm_write(dst, "tigerOne_copy.ppm");

// free memory
ppm_free(&src);
ppm_free(&dst);
```

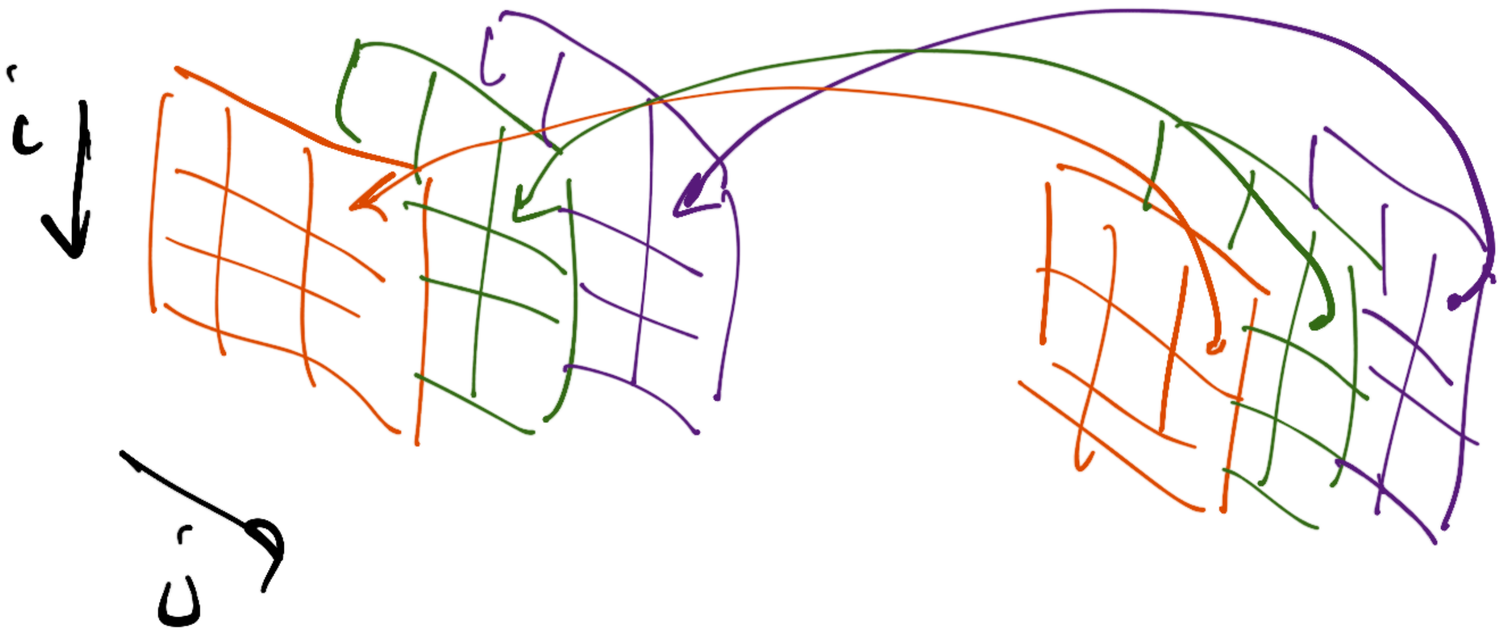
```
PPM *ppm_copy(PPM *src)
{
    int r = src->rows;
    int c = src->cols;

    PPM *dst = ppm_alloc(r, c,
                         src->maxc);
    for(int i=0; i<r; i++){
      for(int j=0; j<c; j++) {
        dst->rpix[i*c+j] = src->rpix[
                                 i*c+j];
        ...
      }
    }
    return(dst); // super important
}
```

'-' '>'

pointer de-ref

what is happening?

Copying pixel by pixel



dst

src

Now, for the transforms :

1. invert :

```
dst->rpix[i*c+j] =
        1.0 - src->rpix[i*c+j]
```

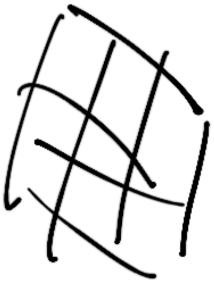
same for gpix, bpix

2. grayscale:

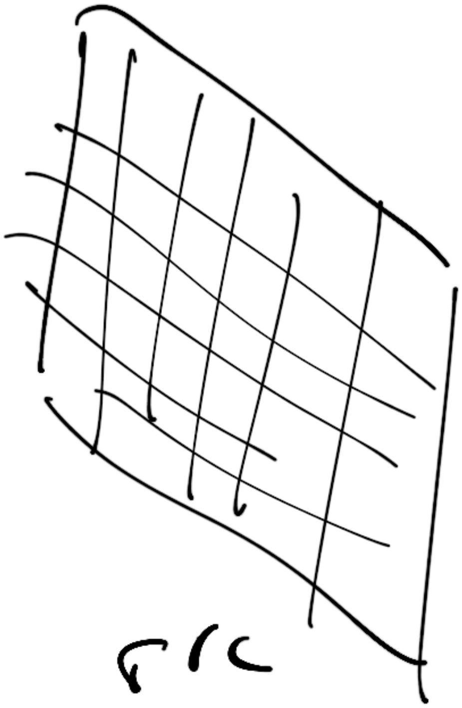
average of 3 colors:

```
dst->rpix[i*c+j]=
    (src->rpix[i*c+j] +
     src->gpix[i*c+j] +
     src->bpix[i*c+j])/3.0;

// save for dst->bpix,
       dst->gpix
```

3. Halfsize: a little trickier



dst



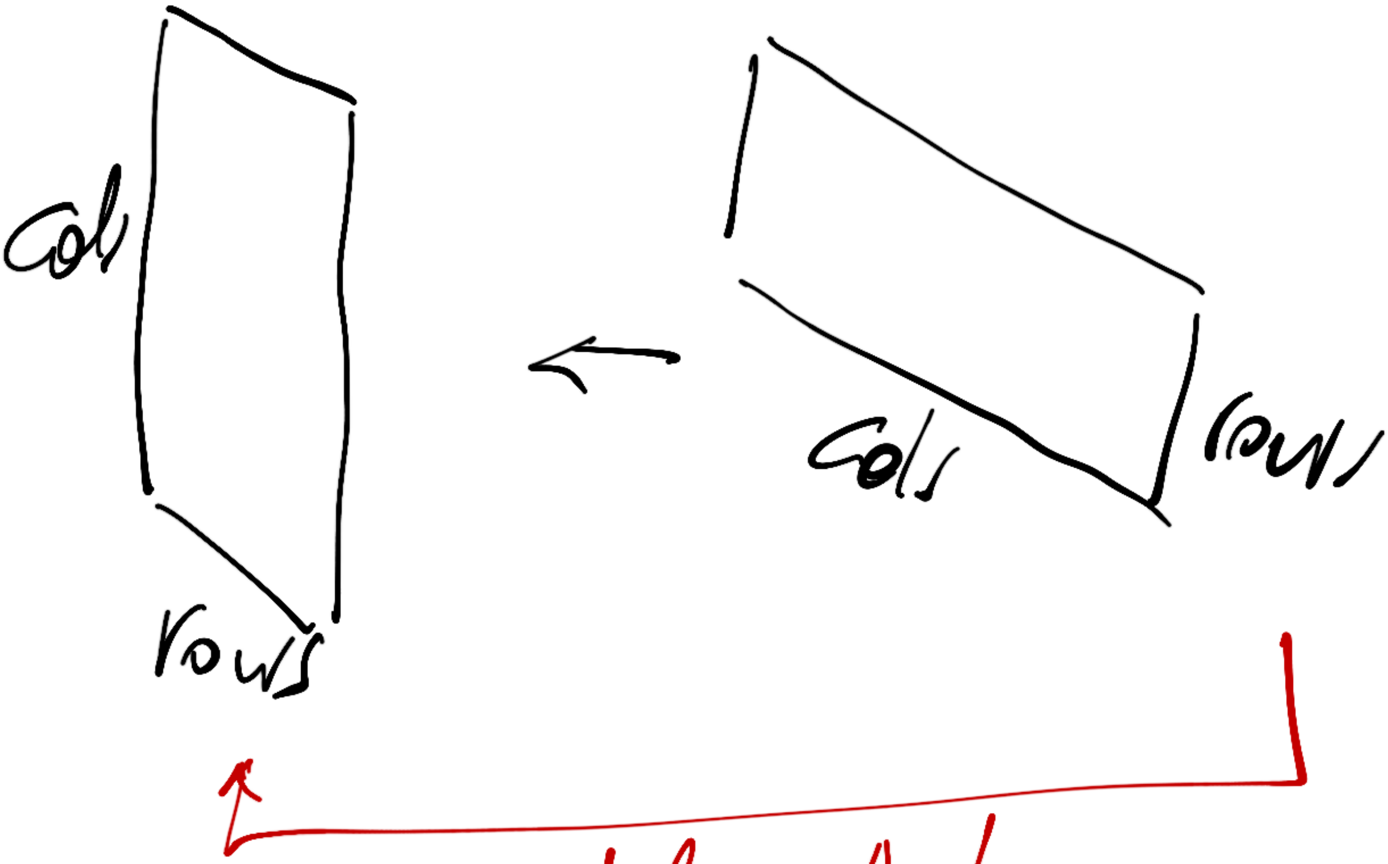
src

which image to loop over?

loop over dst image

```
int r2 = src->rows/2;
int c2 = src->cols/2;
PPM *dst = ppm_alloc(r2,c2,
              src->maxc);
for(i=0; i<r2; i++){
  for(j=0; j<c2; j++){
     dst->rpix[i*c2+j] =
        src->rpix[
   (2*i)*src->cols + (2*j)];
   // same for bpix, gpix.
  }
}
return(dst);
```

4. rotate



watch out!
these switch!

USING rows, cols of SRC img

```
for(int i=0; i<cols; i++) {
  for(int j =0; j<rows; j++) {

      dst->rpix[i*rows+j] =
    src->rpix[j*cols+(cols-i)];
      //same for gpix, bpix
   }
}
return(dst);
```

WHY? linear algebra

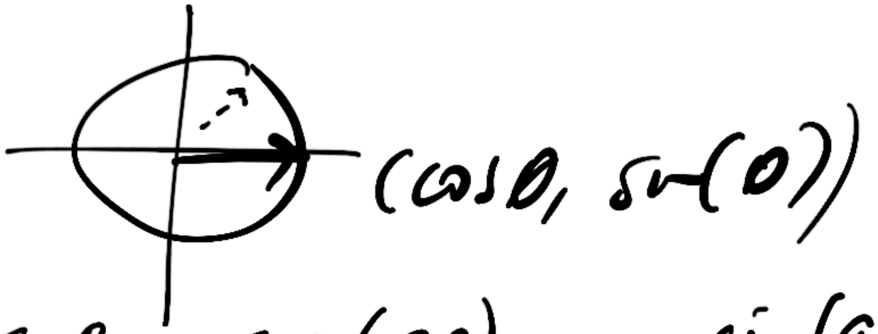
In general, we're using affine transformations — matrix algebra

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$x' = (1)x + (0)y + (0)1 = x$$

$$y' = (0)x + (1)y + (0)1 = y$$

$$1 = (0)x + (0)y + (1)1 = 1$$



when $\theta = 90^\circ$, $\cos(90) = 0$, $\sin(90) = 1$

rotation (about z-axis):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} y & \underbrace{-x}_{} & 1 \end{bmatrix}$$

-ve sign strange: with -x really.

scaling is similar:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2x & 2y & 1 \end{bmatrix}$$

---

there's one more: translate

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} x+dx & y+dy & 1 \end{bmatrix}$$

- these 3 transformations are the bedrock of computer graphics — all CGI is based on these

- what's missing ?

```
PPM * ppm_alloc(int r, int c,
                int m)
{
    PPM *img=NULL; //what to
                     return
  img = (PPM *)malloc(sizeof(PPM));
  img->rows = r;
  img->cols = c;
  img->maxc = m;
  img->magic = (char *)malloc(3 *
                           sizeof(char));
  strcpy(img->magic, "P6");
```

```
img->rpix = (float *)malloc(
                   r*c*sizeof(float));
            "
     //same for bpix, gpix

  return(img);
}
```

```
void ppm_free(PPM** img)
{
                          pointer to
                            pointer

      //free everything malloc
        was used for

  if((*img)->magic) free((*img)->
                                  magic);
      //same for rpix, bpix, gpix
  //finally, free image itself
   free((*img));  (*img)=NULL;
}
```

- the only other majo function that remain are

ppm_write(---)

ppm_read(---) // only pixels

ppm_header // the hardest one

- some help

```
void pnm_header(FILE *in,
                int *cols,
                int *rows,
                int *maxc)
{
        char    c;

  *rows = *cols = 0;
  if ( ((c=fgetc(in))=='P') &&
       ((c=fgetc(in))=='6') ) {
         // eat '\n'
         // check for # comments.
                  ————————→
        // ...
  } else //error
```

// read cols, rows

// read maxc

// eat '\n'

STOP, DO NOT CLOSE FILE

```
do { c = fgetc(in); } while(c != '\n');

if ((c = fgetc(in)) == '#')
  do { c = fgetc(in); } while(c != '\n');
else
  ungetc(c, in);
```

```
fscanf(in, "%u %u",
        cols, rows);
fscanf(in "%u\n", maxc)
```

---

should check to make sure rows, cols are reasonable, (not 0 or -ve) & maxc is 255