GEBZE TECHNICAL UNIVERSITY

ENGINEERING FACULTY

COMPUTER ENGINEER DEPARTMENT

CSE-464 IMAGE PROCESSING HOMEWORK 1

REPORT

AHMET YIGIT

200104004066

Forward Mapping

- Forward Mapping iterates over all the pixels of the image, and the corresponding transformation is applied to each pixel individually.

- Advantages
    - **Direct Transformation:**
        - Each pixel from the original image directly maps to a location in the transformed image, so the mapping is straightforward for simple transformations.

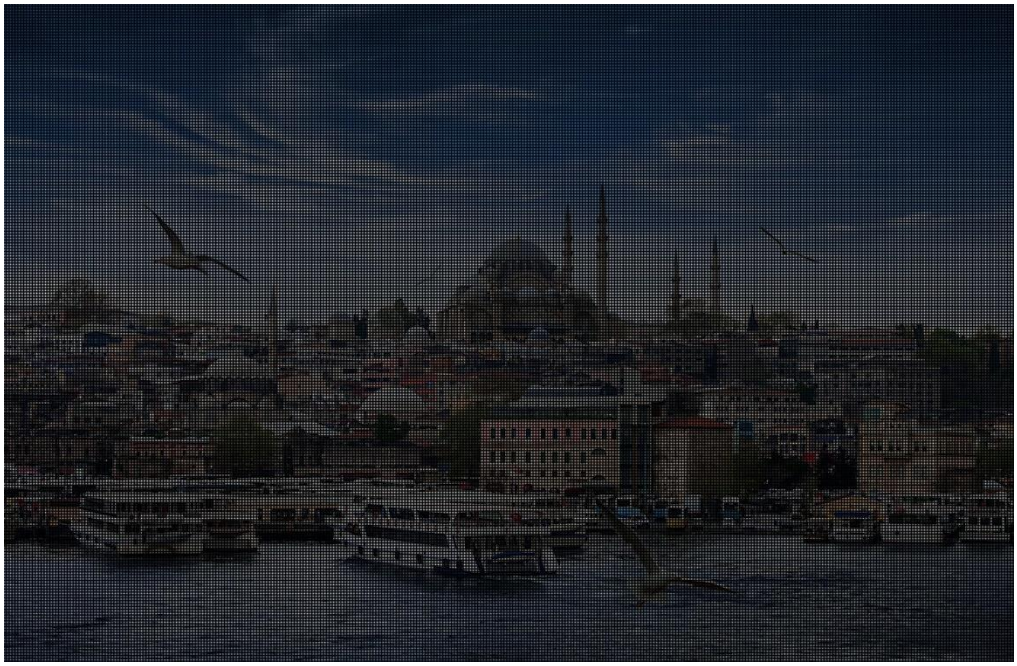
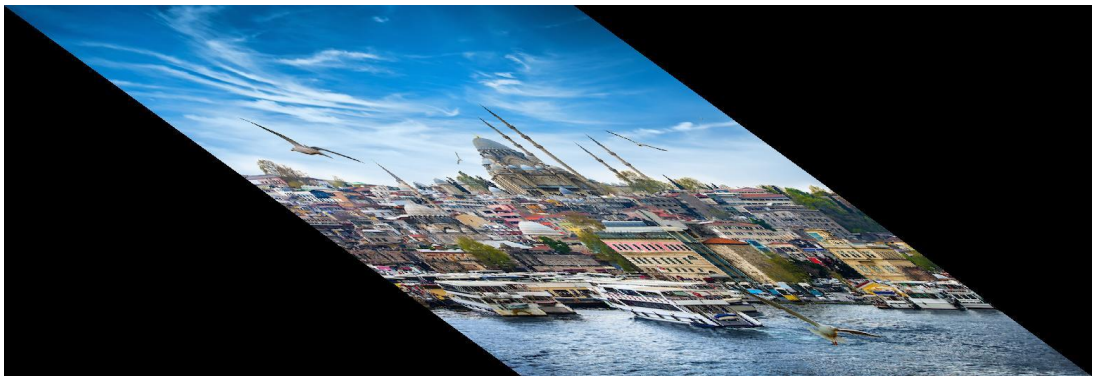
    - **Ideal for Distortion Effects:**
        - Forward mapping can handle complex transformations or distortions, which may be challenging for other methods like inverse mapping.

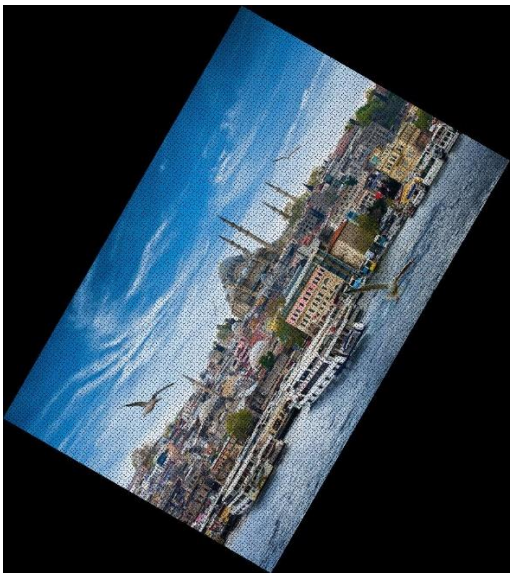
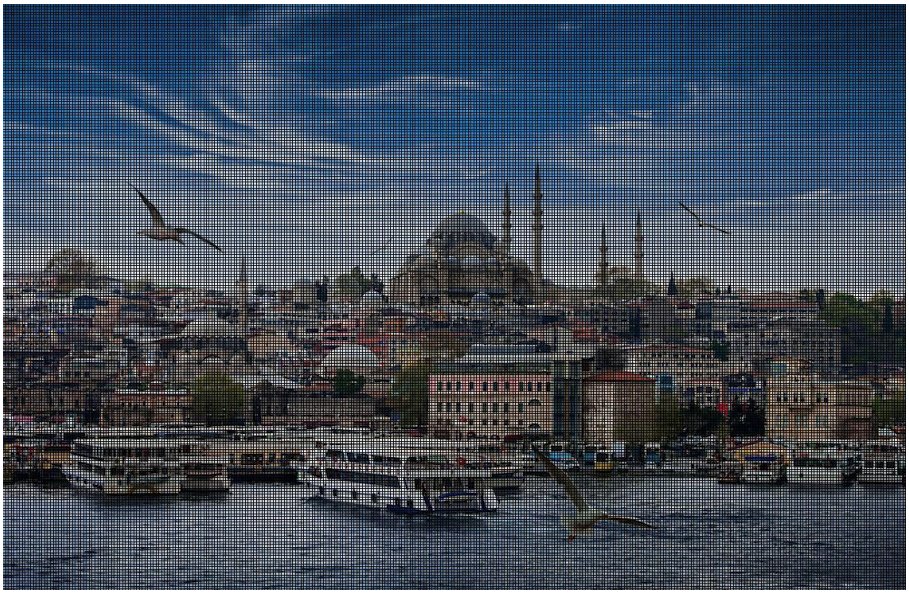
    - **Preserves Original Details:**
        - Since it iterates over every pixel, it tends to maintain detailed information from the original image.

- Disadvantages
    - **Gaps in the Output Image:**
        - Due to rounding errors or transformations that stretch the image, gaps or holes may appear in the transformed image because some transformed coordinates do not align with the pixel grid.

    - **Computationally Intensive:**
        - Iterating over each pixel and applying transformations can be slow, especially for high-resolution images.

    - **Complexity in Interpolation:**
        - Filling gaps usually requires interpolation, which adds computational complexity and can sometimes blur the image.

- Affine Transforms With Forward Mapping
    - **Scale X=2 Y=2**
        - 
        - In this affine transformation, forward mapping is applied by scaling each pixel in the original image by a factor of 2 along both the X and Y axes. This scaling enlarges the image, doubling the size in both directions. Each original pixel is mapped to a 2x2 block in the transformed image, which makes the image appear zoomed in. However, this may create gaps, requiring interpolation to fill in missing pixels for a smooth result without it, pictures holds black holes.
    - **Horizontal Shear X=1.4**
        - 
        - In this transformation, forward mapping is used to apply a horizontal shear, shifting each pixel horizontally by 1.4 times. This creates a slanted effect, making the image appear stretched sideways while keeping vertical alignment.

- **Rotate 60 Degree**
  - 
  - This transformation rotates each pixel in the image by 60 degrees around a central point (given at homework folder), giving the image a rotated appearance. Forward mapping shifts pixels to new locations based on this angle, often resulting in gaps that may need interpolation to maintain image continuity.
- **Zoom 1.4 Times**
  - 
  - This transformation enlarges the image by scaling each pixel by a factor of 1.4 in both X and Y directions, creating a zoomed-in effect. Forward mapping places each pixel in a larger area, often requiring interpolation to fill gaps for a smooth result but we do not do interpolation so gaps are captured by black holes.

- **Backward Mapping**
  - Backward mapping is a transformation method where each pixel in the output image is mapped back to a location in the original image, ensuring all output pixels are filled without gaps.

  - Advantages
    - **No Gaps in the Output:**
      - Backward mapping ensures that all pixels in the output image are filled, avoiding gaps that can occur in forward mapping.

    - **Efficient for Common Transformations:**
      - It's generally efficient for scaling, rotation, and other standard transformations, as it maps directly to known source coordinates.
  - Disadvantages
    - **Requires Interpolation:**
      - Since mapped points often don't align perfectly with source pixels, interpolation is needed, which can introduce minor blurring or artifacts.
    - **More Complex Calculations:**
      - Determining where each output pixel maps back to in the original image can require more complex computations, particularly for irregular transformations.
  - **Interpolation**
    - Interpolation is a process in digital image processing used to estimate and fill in missing or intermediate pixel values. When an image is resized or transformed, some pixels don't align with exact positions, so interpolation calculates new pixel values based on surrounding ones. Common methods include nearest-neighbor, bilinear, and bicubic interpolation, each offering varying levels of accuracy and smoothness.
    - It is like point drawing with ratio between its neighbors.

- **Affine Transforms With Backward Mapping Without Interpolation**
  - **Scale X=2 Y=2**
    - 
      In this transformation, each pixel in the output image is mapped back to the original image with a scaling factor of 2 along both the X and Y axes. Since interpolation is not used, only pixels that map exactly to the scaled coordinates in the original image will be filled in the output, leading to potential gaps and a blocky appearance where no corresponding original pixel exists.
  - **Horizontal Shear X=1.4**
    - 
      In this transformation, each pixel in the output image is mapped back to the original image based on a horizontal shear factor of 1.4. This shifts pixels horizontally according to their Y-coordinates, but without interpolation, only pixels that exactly correspond to the calculated positions will be filled, resulting in gaps or artifacts in the output image where no original pixels exist.

- **Rotate 60 Degree**

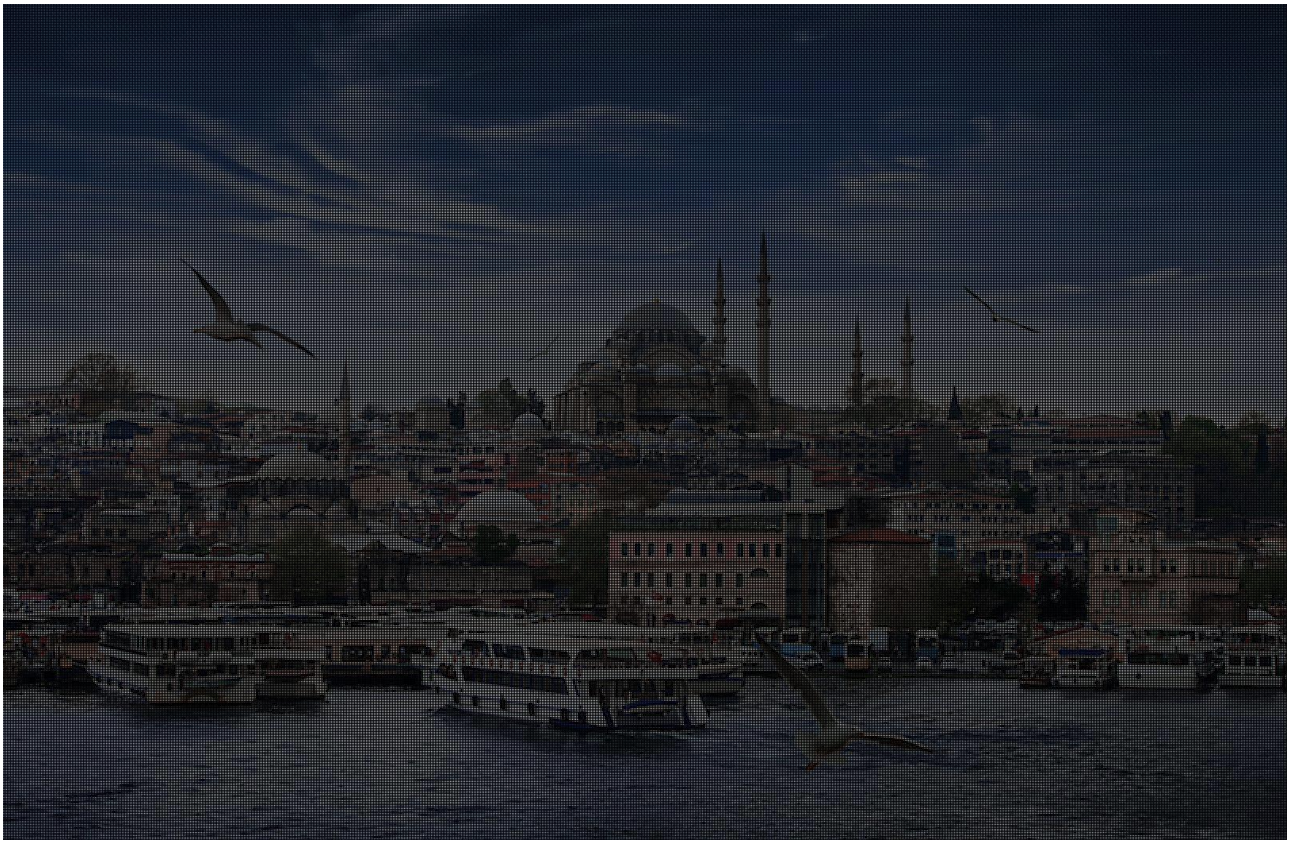
  - In this transformation, each pixel in the output image is mapped back to the original image after a rotation of 60 degrees. Without interpolation, only the pixels that align perfectly with the new coordinates will be filled in the output image, which may result in empty areas where there are no corresponding original pixels.
- **Zoom 1.4 Times**

  - In this transformation, each pixel in the output image is mapped back to the original image with a zoom factor of 1.4. Without interpolation, only the pixels that exactly correspond to the calculated positions will be filled, leading to potential gaps and a pixelated appearance in the output image where no original pixels exist.

- **Affine Transforms With Backward Mapping With Interpolation**
  - **Scale X=2 Y=2**
    - 
    - In this transformation, each pixel in the output image is mapped back to the original image with a scaling factor of 2 along both the X and Y axes. With interpolation, the algorithm estimates pixel values for positions that do not correspond to exact original pixels, filling in gaps smoothly. This results in a scaled-up image that maintains a more continuous appearance and reduces pixelation, providing a better visual quality.
  - **Horizontal Shear X=1.4**
    - 
    - In this transformation, each pixel in the output image is mapped back to the original image based on a horizontal shear factor of 1.4. With interpolation, the algorithm estimates pixel values for new positions created by the shear, filling in gaps smoothly. This approach allows for a continuous and visually appealing output.

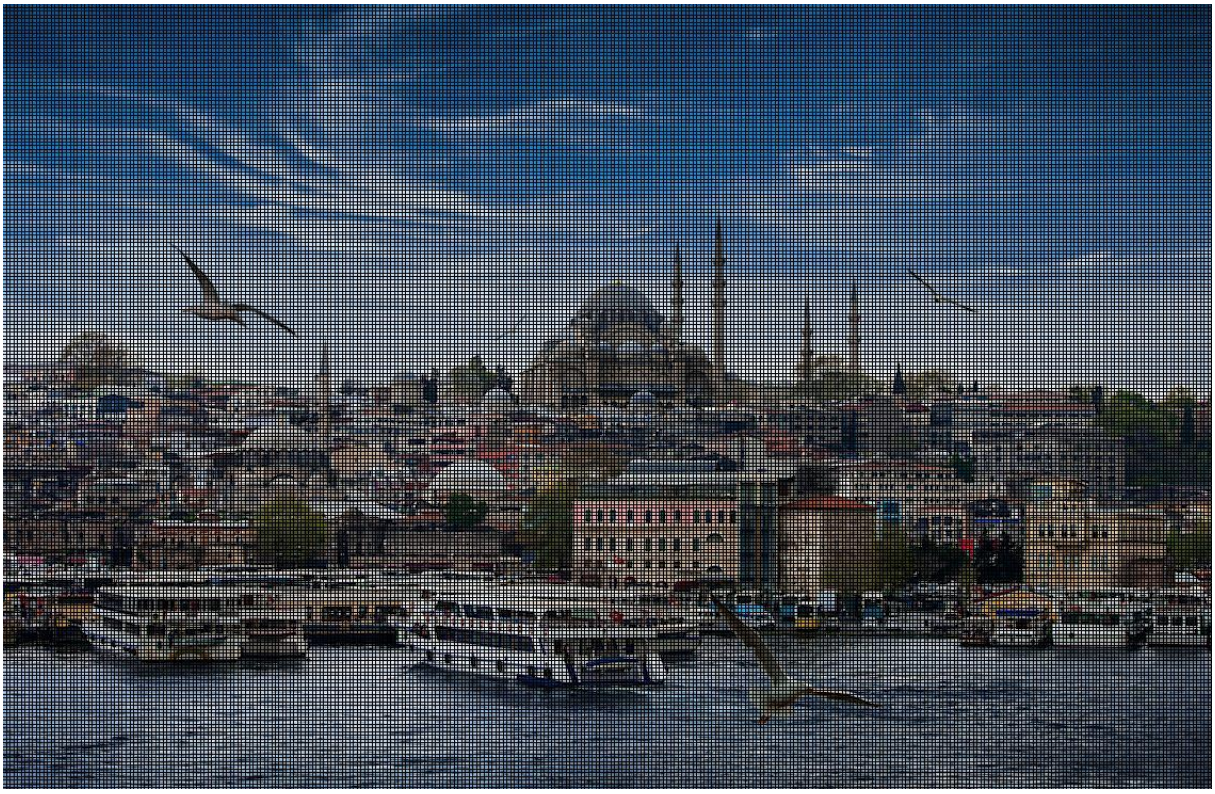
- **Rotate 60 Degree**
  - 
  - In this transformation, each pixel in the output image is mapped back to the original image after a rotation of 60 degrees. With interpolation, the algorithm calculates and estimates pixel values for positions that do not align perfectly with original pixels, effectively filling in gaps. This results in a smoothly rotated image with improved continuity and visual quality.
- **Zoom 1.4 Times**
  - 
  - In this transformation, each pixel in the output image is mapped back to the original image with a zoom factor of 1.4. With interpolation, the algorithm estimates pixel values for positions that do not correspond to exact original pixels, allowing for a smooth filling of gaps. This approach creates a zoomed-in image that retains visual quality and detail, minimizing pixelation and enhancing overall appearance.

- **Comparison**
  - **Forward vs Backward**
    - **Mapping Direction**
      - Forward Mapping: Transforms each pixel from the original image to a new location in the output image.
      - Backward Mapping: Maps each pixel in the output image back to the corresponding location in the original image.
    - **Filling Gaps**
      - Forward Mapping: May leave gaps in the output image if transformed pixels do not cover every pixel location.
      - Backward Mapping: Fills all output pixel locations, reducing the likelihood of gaps, especially when interpolation is used.
    - **Complexity**
      - Forward Mapping: Typically simpler for straightforward transformations but may require complex handling for interpolation.
      - Backward Mapping: Often more computationally intensive due to the need to calculate original pixel coordinates.
    - **Image Quality**
      - Forward Mapping: Can result in a blocky or distorted appearance due to gaps and uneven pixel coverage.
      - Backward Mapping: Generally yields smoother and more visually appealing results when interpolation is applied.
    - **Use Cases**
      - Forward Mapping: More suited for applications where precise pixel control is needed, like creating distortions.
      - Backward Mapping: Preferred for standard transformations (e.g., scaling, rotation) where a complete output image is required.

- Scale Image Comparison
    - Backward 1620*1052 2.1Mb
    - 
    - Forward 1620*1052 – 1.1Mb
    - 

- Rotate Image Comparison
    - Backward 860*964 854.9Kb
    - 
    - Forward 1620*1052 – 1Mb
    - 

- Shear Image Comparison
    - Backward 1546*526 760.4Kb
    - 
    - Forward 1620*1052 – 760.4Kb
    - 

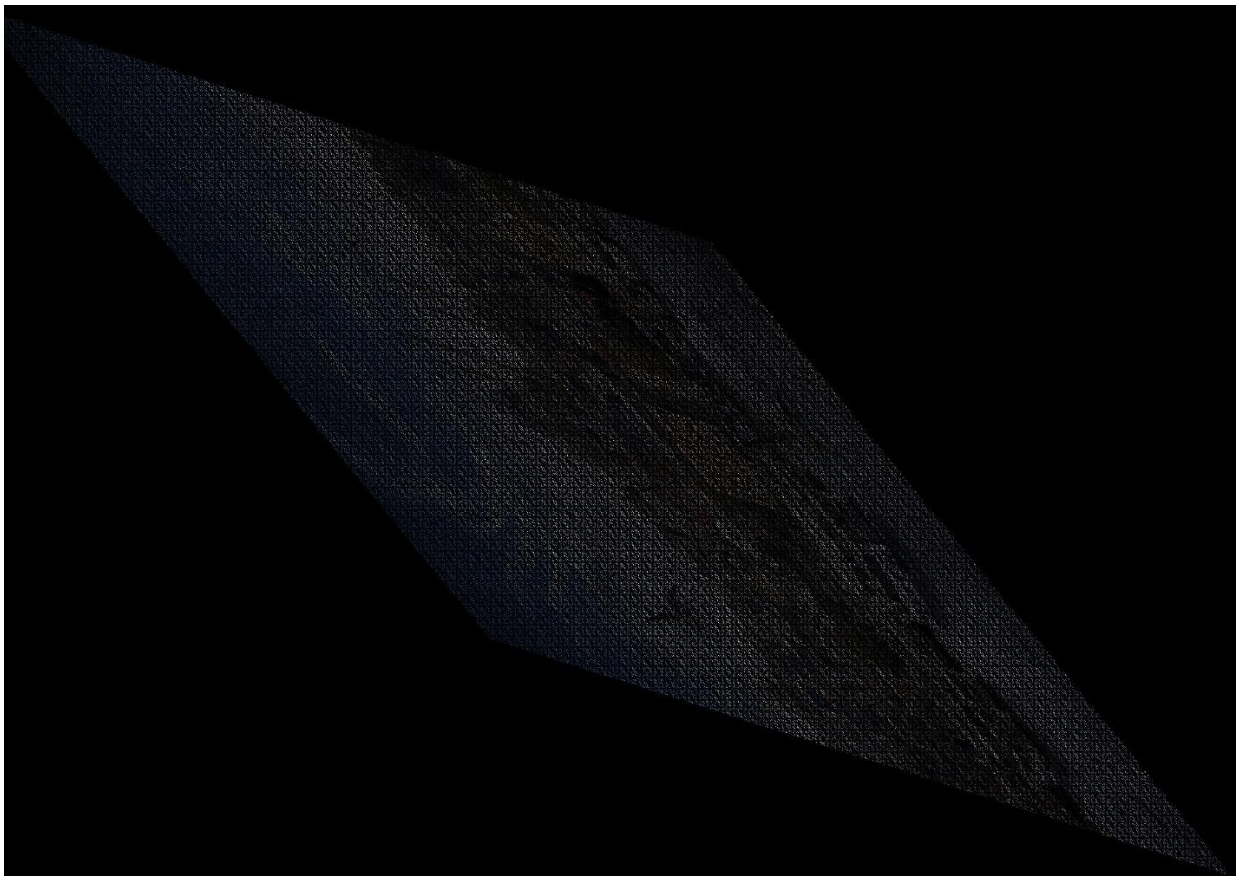
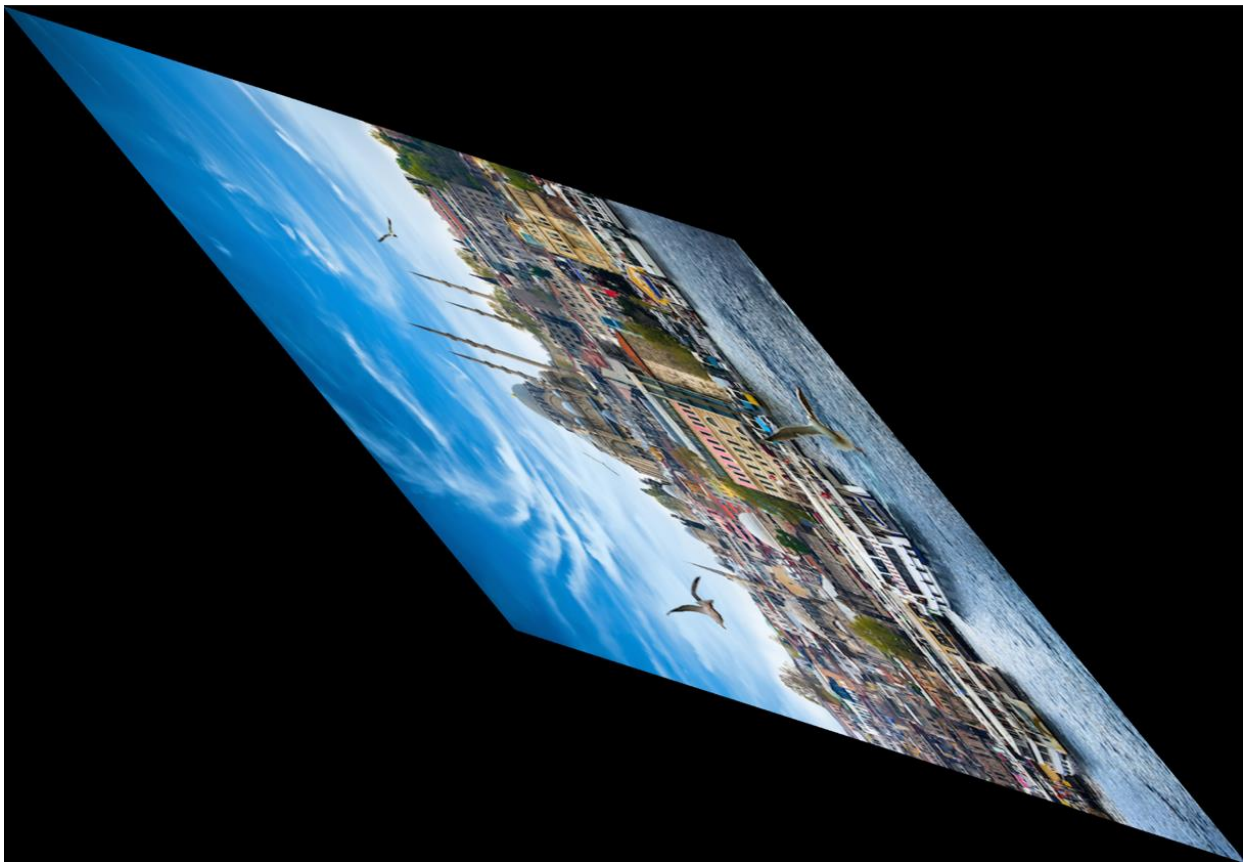
- Zoom Image Comparison
    - Backward 1134*736 1.2Mb
    - 
    - Forward 1134*736 1.2Mb
    - 

- **In the experiments conducted, various affine transformations like scaling, shearing, zooming, and rotating were utilized to compare the outcomes of forward mapping and backward mapping techniques. When both methods were applied to the same original image, significant differences in visual quality and pixel filling became apparent.**

- **In the forward mapping method, some target pixels end up empty because the new position for each pixel is determined individually. This results in the formation of "black holes," which are particularly noticeable during scaling, as not every pixel can find a corresponding position in the target image. Conversely, the backward mapping approach seeks to trace each pixel's location back to the original image, effectively filling in any gaps. When interpolation is applied, it estimates pixel values for these empty spots, leading to a smoother and more uniform output.**

- **Consequently, forward mapping tends to create a darker and more uneven appearance due to the presence of dark areas and gaps in the target image. In contrast, backward mapping, especially when combined with interpolation, provides a more seamless and visually pleasing result by filling in the empty spaces, thereby improving the overall image quality.**

- **They are the images that all transformations performed one by one. As you can see the difference between two operations.**

- 
- 

- **Backward Without Interpolation vs Backward with Interpolation**
  - **Pixel Filling**
    - Without Interpolation: The output image may contain gaps where no original pixel values are assigned, resulting in visible empty spaces.
    - With Interpolation: Pixel values for the output image are estimated, filling in gaps and creating a smoother transition between pixels.
  - **Image Quality**
    - Without Interpolation: The image may appear blocky and less refined due to unfilled areas, leading to a less visually appealing result.
    - With Interpolation: The image quality is generally higher, as interpolation helps create a more continuous and aesthetically pleasing appearance.
  - **Computational Complexity**
    - Without Interpolation: This method is typically less computationally intensive, as it simply maps pixels back without estimating new values.
    - With Interpolation: This approach requires additional calculations to estimate pixel values, resulting in increased computational complexity but improved visual outcomes.
  - **Use Cases**
    - Without Interpolation: Suitable for scenarios where speed is prioritized over image quality, such as real-time processing.
    - With Interpolation: Preferred for applications that require high-quality images, such as photo editing or digital rendering.
  - **Visual Artifacts**
    - Without Interpolation: May exhibit more noticeable artifacts and discontinuities, which can detract from the overall look of the image.
    - With Interpolation: Helps minimize visual artifacts, providing a smoother and more cohesive final image.

- Scale Image Comparison
    - Backward with Interpolation 1620*1052 2.1Mb
    - 
    - Backward without Interpolation 1620*1052 – 970.5Kb
    - 

- We can’t detect differences let’s look closer
  - Backward with Interpolation 460%

  - Backward without Interpolation 460%

- At first seeing of this photos, it’s hard to see much difference, but when we zoom in at 460%, the details really stand out. With interpolation, edges look smoother and more seamless, avoiding the pixelation and roughness that’s pretty obvious without it. This smoother look is great but comes at a cost—almost double the file size. So, while interpolation definitely boosts visual quality, it also demands more storage and processing power. It’s a trade-off, really, where each method has its own perks depending on what’s more important for the project.

- Rotate Image Comparison
    - Backward with Interpolation 860*964 857.5Kb
    - 
    - Backward without Interpolation 860*964 854.9Kb
    - 

- Shear Image Comparison
    - Backward with Interpolation 1546*526 760.4Kb
    - 
    - Backward without Interpolation 1546*526– 760.4Kb
    - 

- We can view that shear-rotate transforms with interpolation or without interpolation in backward mapping do not change the size of the image in kilobytes. Their size is nearly the same. This is because both methods maintain the same number of pixels, and any variations in pixel values do not significantly impact the overall file size. Additionally, the compression algorithms used for image files play a crucial role in determining the final size, often masking the effects of transformations.

- Zoom Image Comparison
    - Backward with Interpolation 1134*736 1.2Mb
- 
    - Backward without Interpolation 1134*736 1.2Mb
- 

- Let’s look closely at images
  - Backward with Interpolation 411%

  - Backward without Interpolation 411%

- You can see those edges more smoothly in interpolation method.

- **In the experiments conducted, various affine transformations like scaling, shearing, zooming, and rotating were utilized to compare the outcomes of forward mapping and backward mapping techniques. When both methods were applied to the same original image, significant differences in visual quality and pixel filling became apparent.**

- **In the forward mapping method, some target pixels end up empty because the new position for each pixel is determined individually. This results in the formation of "black holes," which are particularly noticeable during scaling, as not every pixel can find a corresponding position in the target image. Conversely, the backward mapping approach seeks to trace each pixel's location back to the original image, effectively filling in any gaps. When interpolation is applied, it estimates pixel values for these empty spots, leading to a smoother and more uniform output.**

- **Consequently, forward mapping tends to create a darker and more uneven appearance due to the presence of dark areas and gaps in the target image. In contrast, backward mapping, especially when combined with interpolation, provides a more seamless and visually pleasing result by filling in the empty spaces, thereby improving the overall image quality.**

- **You can view two images. First one no interpolation second one with interpolation. 500% zoomed. You can detect edges and color tranformations at that point difference clearly. Interpolation makes more smother translations in edges.**

- **No Interpolation %500**
- 
- **With Interpolation %500**
- 

- **Code Implementation**
  - **Forward Mapping**

```python
class ForwardMapping:
    def __init__(self):
        self.image = Image.open("./istanbul.jpg")

    def shearImage(self, shearX=1.4, shearY=0):
        orgWidth, orgHeight = self.image.size

        newWidth  = int(orgHeight * shearX) + orgWidth
        newHeight = int(orgWidth * shearY)+ orgHeight

        returnedImage = Image.new("RGB", (newWidth, newHeight))

        for y in range(orgHeight):
            for x in range(orgWidth):
                xPoint = x + int(y * shearX)
                yPoint = y + int(x * shearY)

                if 0 <= xPoint < newWidth and 0 <= yPoint < newHeight:
                    pixel_value = self.image.getpixel((x, y))
                    returnedImage.putpixel((xPoint, yPoint), pixel_value)

        self.image = returnedImage


    def rotateImage(self, rotateDegrees=60):
        angleRad = rotateDegrees * math.pi / 180

        orgWidth, orgHeight = self.image.size

        newWidth = int(abs(math.cos(angleRad) * orgWidth) + abs(math.sin(angleRad) * orgHeight))
        newHeight = int(abs(math.sin(angleRad) * orgWidth) + abs(math.cos(angleRad) * orgHeight))

        returnedImage = Image.new("RGB", (newWidth, newHeight))

        centerLocX = int(newWidth / 2)
        centerLocY = int(newHeight / 2)

        newImageData = [[[0 for _ in range(3)] for _ in range(newWidth)] for _ in range(newHeight)]

        for y in range(orgHeight):
            for x in range(orgWidth):
                xPointOrg = int(x - orgWidth / 2)
                yPointOrg = int(y - orgHeight / 2)

                xPoint = int(xPointOrg * math.cos(-angleRad) - yPointOrg * math.sin(-angleRad)) +
centerLocX
                yPoint = int(xPointOrg * math.sin(-angleRad) + yPointOrg * math.cos(-angleRad)) +
centerLocY

                if 0 <= xPoint < newWidth and 0 <= yPoint < newHeight:
                    pixel_value = self.image.getpixel((x, y))
                    newImageData[yPoint][xPoint] = list(pixel_value)

        for y in range(newHeight):
            for x in range(newWidth):
                returnedImage.putpixel((x, y), tuple(newImageData[y][x]))

        self.image = returnedImage

    def scaleImage(self, scaleX=2, scaleY=2):
        orgWidth, orgHeight = self.image.size

        newWidth = int(orgWidth * scaleX)
        newHeight = int(orgHeight * scaleY)

        returnedImage = Image.new("RGB", (newWidth, newHeight))

        for y in range(orgHeight):
            for x in range(orgWidth):
                xPoint = int(x * scaleX)
                yPoint = int(y * scaleY)

                if 0 <= xPoint < newWidth and 0 <= yPoint < newHeight:
                    pixel_value = self.image.getpixel((x, y))
                    returnedImage.putpixel((xPoint, yPoint), pixel_value)

        self.image = returnedImage

    def zoomImage(self, zoom = 1.4):
        orgWidth, orgHeight = self.image.size

        newWidth  = int(orgWidth * zoom)
        newHeight = int(orgHeight * zoom)

        returnedImage = Image.new("RGB", (newWidth, newHeight))

        for y in range(orgHeight):
            for x in range(orgWidth):
                xPoint = int(x * zoom)
                yPoint = int(y * zoom)

                if 0 <= xPoint < newWidth and 0 <= yPoint < newHeight:
                    pixel_value = self.image.getpixel((x, y))
                    returnedImage.putpixel((xPoint, yPoint), pixel_value)

        self.image = returnedImage

    def getImage(self):
        return self.image
```

-

- **Backward Mapping Without Interpolation**

```python
class BackwardMapping:
    def __init__(self):
        self.image = Image.open("./istanbul.jpg")

    def rotateImage(self, rotateDegrees=60):
        angleRad = rotateDegrees * math.pi / 180

        orgWidth, orgHeight = self.image.size

        newWidth = int(abs(math.cos(angleRad) * orgWidth) + abs(math.sin(angleRad) * orgHeight))
        newHeight = int(abs(math.sin(angleRad) * orgWidth) + abs(math.cos(angleRad) * orgHeight))

        returnedImage = Image.new("RGB", (newWidth, newHeight))

        centerLocX = int(newWidth / 2)
        centerLocY = int(newHeight / 2)

        newImageData = [[[0 for _ in range(3)] for _ in range(newWidth)] for _ in range(newHeight)]

        for y in range(newHeight):
            for x in range(newWidth):
                xPointOrg = x - centerLocX
                yPointOrg = y - centerLocY

                xOldImagePoint = int(xPointOrg * math.cos(angleRad) - yPointOrg * math.sin(angleRad)) +
int(orgWidth / 2)
                yOldImagePoint = int(xPointOrg * math.sin(angleRad) + yPointOrg * math.cos(angleRad)) +
int(orgHeight / 2)

                if 0 <= xOldImagePoint < orgWidth and 0 <= yOldImagePoint < orgHeight:
                    pixel_value = self.image.getpixel((xOldImagePoint, yOldImagePoint))
                    newImageData[y][x] = list(pixel_value)

        for y in range(newHeight):
            for x in range(newWidth):
                returnedImage.putpixel((x, y), tuple(newImageData[y][x]))

        self.image = returnedImage

    def scaleImage(self, scaleX=2, scaleY=2):
        orgWidth, orgHeight = self.image.size

        newWidth = int(orgWidth * scaleX)
        newHeight = int(orgHeight * scaleY)

        returnedImage = Image.new("RGB", (newWidth, newHeight))

        for y in range(newHeight):
            for x in range(newWidth):
                xPointOrg = int(x / scaleX)
                yPointOrg = int(y / scaleY)

                if 0 <= xPointOrg < orgWidth and 0 <= yPointOrg < orgHeight:
                    pixel_value = self.image.getpixel((xPointOrg, yPointOrg))
                    returnedImage.putpixel((x, y), pixel_value)

        self.image = returnedImage

    def shearImage(self, shearX=1.4, shearY=0):
        orgWidth, orgHeight = self.image.size

        newWidth  = int(orgHeight * shearX) + orgWidth
        newHeight = int(orgWidth * shearY)+ orgHeight

        returnedImage = Image.new("RGB", (newWidth, newHeight))

        for y in range(newHeight):
            for x in range(newWidth):
                xPointOld = x - int(y * shearX)
                yPointOld = y - int(x * shearY)

                if 0 <= xPointOld < orgWidth and 0 <= yPointOld < orgHeight:
                    pixel_value = self.image.getpixel((xPointOld, yPointOld))
                    returnedImage.putpixel((x, y), pixel_value)

        self.image = returnedImage

    def zoomImage(self, zoom=1.4):
        orgWidth, orgHeight = self.image.size

        newWidth  = int(orgWidth * zoom)
        newHeight = int(orgHeight * zoom)

        returnedImage = Image.new("RGB", (newWidth, newHeight))

        for y in range(newHeight):
            for x in range(newWidth):
                xPointOld = int(x / zoom)
                yPointOld = int(y / zoom)

                if 0 <= xPointOld < newWidth and 0 <= yPointOld < newHeight:
                    pixel_value = self.image.getpixel((xPointOld, yPointOld))
                    returnedImage.putpixel((x, y), pixel_value)

        self.image = returnedImage

    def getImage(self):
        return self.image
```

- **Backward Mapping With Interpolation**
  - 

```python
class BackwardMapping:
    def __init__(self):
        self.image = Image.open("./istanbul.jpg")

    def bilinearInterpolate(self, x, y):
        x0, y0 = int(math.floor(x)), int(math.floor(y))
        x1, y1 = min(x0 + 1, self.image.width - 1), min(y0 + 1, self.image.height - 1)

        if x0 < 0 or y0 < 0 or x0 >= self.image.width or y0 >= self.image.height:
            return (0, 0, 0)

        # q11      q12
        #    TARGET
        # q21      q22

        q11 = self.image.getpixel((x0, y0))
        q21 = self.image.getpixel((x1, y0))
        q12 = self.image.getpixel((x0, y1))
        q22 = self.image.getpixel((x1, y1))

        f_x1 = x - x0 # yüzdelik alma yeri hocanın derste ____0.4X____TARGET____0.6X_____ dediği yer
        f_y1 = y - y0 # yüzdelik alma yeri hocanın derste ____0.4Y____TARGET____0.6Y_____ dediği yer

        r = (q11[0] * (1 - f_x1) * (1 - f_y1) +
            q21[0] * f_x1 * (1 - f_y1) +
            q12[0] * (1 - f_x1) * f_y1 +
            q22[0] * f_x1 * f_y1)

        g = (q11[1] * (1 - f_x1) * (1 - f_y1) +
            q21[1] * f_x1 * (1 - f_y1) +
            q12[1] * (1 - f_x1) * f_y1 +
            q22[1] * f_x1 * f_y1)

        b = (q11[2] * (1 - f_x1) * (1 - f_y1) +
            q21[2] * f_x1 * (1 - f_y1) +
            q12[2] * (1 - f_x1) * f_y1 +
            q22[2] * f_x1 * f_y1)

        r = max(0, min(255, int(round(r))))
        g = max(0, min(255, int(round(g))))
        b = max(0, min(255, int(round(b))))

        return (r, g, b)

    def rotateImage(self, rotateDegrees=60):
        angleRad = rotateDegrees * math.pi / 180

        orgWidth, orgHeight = self.image.size

        newWidth = int(abs(math.cos(angleRad) * orgWidth) + abs(math.sin(angleRad) * orgHeight))
        newHeight = int(abs(math.sin(angleRad) * orgWidth) + abs(math.cos(angleRad) * orgHeight))

        returnedImage = Image.new("RGB", (newWidth, newHeight))

        centerLocX = int(newWidth / 2)
        centerLocY = int(newHeight / 2)

        newImageData = [[[0 for _ in range(3)] for _ in range(newWidth)] for _ in range(newHeight)]

        for y in range(newHeight):
            for x in range(newWidth):
                xPointOrg = x - centerLocX
                yPointOrg = y - centerLocY

                xOldImagePoint = int(xPointOrg * math.cos(angleRad) - yPointOrg * math.sin(angleRad)) +
int(orgWidth / 2)
                yOldImagePoint = int(xPointOrg * math.sin(angleRad) + yPointOrg * math.cos(angleRad)) +
int(orgHeight / 2)

                pixel_value = self.bilinearInterpolate(xOldImagePoint, yOldImagePoint)
                newImageData[y][x] = list(pixel_value)

        for y in range(newHeight):
            for x in range(newWidth):
                returnedImage.putpixel((x, y), tuple(newImageData[y][x]))

        self.image = returnedImage

    def scaleImage(self, scaleX=2, scaleY=2):

        orgWidth, orgHeight = self.image.size

        newWidth = int(orgWidth * scaleX)
        newHeight = int(orgHeight * scaleY)

        returnedImage = Image.new("RGB", (newWidth, newHeight))

        for y in range(newHeight):
            for x in range(newWidth):
                xPointOrg = (x / scaleX)
                yPointOrg = (y / scaleY)

                pixel_value = self.bilinearInterpolate( xPointOrg, yPointOrg)
                returnedImage.putpixel((x, y), pixel_value)

        self.image = returnedImage

    def shearImage(self, shearX=1.4, shearY=0):

        orgWidth, orgHeight = self.image.size

        newWidth  = int(orgHeight * shearX) + orgWidth
        newHeight = int(orgWidth * shearY)+ orgHeight

        returnedImage = Image.new("RGB", (newWidth, newHeight))

        for y in range(newHeight):
            for x in range(newWidth):
                xPointOld = x - int(y * shearX)
                yPointOld = y - int(x * shearY)

                pixel_value = self.bilinearInterpolate(xPointOld, yPointOld)
                returnedImage.putpixel((x, y), pixel_value)

        self.image = returnedImage

    def zoomImage(self, zoom=1.4):
        orgWidth, orgHeight = self.image.size
        newWidth = int(orgWidth * zoom)
        newHeight = int(orgHeight * zoom)

        returnedImage = Image.new("RGB", (newWidth, newHeight))

        for y in range(newHeight):
            for x in range(newWidth):
                xPointOld = x / zoom
                yPointOld = y / zoom

                pixel_value = self.bilinearInterpolate(xPointOld, yPointOld)

                returnedImage.putpixel((x, y), pixel_value)

        self.image = returnedImage

    def getImage(self):
        return self.image
```