Sebastian Mueller, Melanie Schepp, Andrey Tyukin

Exercise 3.1 (Polygon picking and snapping) For the vertex-to-line snapping behavior we have to be able to project points to line segments.

Let $a, b, x \in \mathbb{R}^2$ be three points, let $[a, b]$ denote the line segment from a to b. We want to find a point $p \in [a, b]$ such that

$$p = \operatorname{argmin}_{p' \in [a,b]} \operatorname{dist}\left(p', x\right).$$

For this, we first move our coordinate system to a:

$$d := b - a$$
$$r := x - a$$

Now the the value $\xi := \left\langle \frac{d}{\|d\|}, r \right\rangle$ is the signed distance along d from a to the projection of x to the infinite line through a and b. If ξ is negative, then $p = a$ is the point closest to x. If ξ is greater than $\|d\|$, then $p = b$ is the point closest to x. Otherwise p lies between a and b:

$$p = a + \left\langle \frac{d}{\|d\|}, r \right\rangle \frac{d}{\|d\|} = a + \frac{\langle r, d \rangle}{\|d\|^2} d$$

Notice that no expensive square-root operations are required so far. If we now wanted to compute distance from x to $[a, b]$, we would just have to compute $\|x - p\|$, this can be done numerically stable with the `hypot` function.

In the code, this looks as follows:

```
/**
 * For a point ‘(x, y)‘ and a line segment ‘a‘ to ‘b‘, computes a
 * point ‘p‘ on the line-segment ‘[a, b]‘ that is closest to ‘(x, y)‘.
 */
void projectToLineSegment(
    double ax, double ay,
    double bx, double by,
    double x, double y,
    double &px, double &py
) {
  // ‘d‘ is the direction of the line-segment
  double dx = bx - ax;
  double dy = by - ay;

  // ‘r‘ is the position of ‘(x,y)‘ relative to ‘a‘
  double rx = x - ax;
  double ry = y - ay;

  double rdDot = rx * dx + ry * dy;
  double dNormSq = dx * dx + dy * dy;

  // ‘p‘ is the closest point to ‘(x,y)‘ on the line segment
  if (rdDot <= 0 || dNormSq == 0) {
    px = ax;
    py = ay;
```

```
  } else if (rdDot >= dNormSq) {
    px = bx;
    py = by;
  } else {
    double f = rdDot / dNormSq;
    px = ax + f * dx;
    py = ay + f * dy;
  }
}
```

Notice that the code handles the corner-cases where $d = 0$, which is important to make the code work with polygons regardless whether the first vertex is equal to the last vertex or not.

Exercise 3.2 (Texture mapping) The implementation of parts 3.2 (a-e) can be seen in the code, the main functionality is focused in the constructor of `CGView` where the texture coordinates are generated, and in `CGView::paintGL`, where the grid is rendered. Notice that we slightly modified the task: we decided to render an almost-transparent version of the image in the background when the grid-mode is activated. Furthermore we avoided the usage of deprecated `GL_QUADS`, and instead achieved the same functionality by using simple lines and triangles.

For the part (f) we need to find a suitable diffeomorphism $\Psi_{r,\alpha}$ of a disk D_r of the radius r into itself in order to implement local inflation/deflation of the image. We decided to compose $\Psi_{r,\alpha}$ out of three functions. First, consider the following diffeomorphism between the disk D_r and the whole plane $\mathbb{R}^2$ (or $\mathbb{R}^n$, the dimension doesn't matter here):

$$\begin{aligned} \phi_r : \mathbb{R}^2 \to D_r \qquad & \phi_r(x) := \frac{rx}{1 + \|x\|} \\ \phi_r^{-1} : D_r \to \mathbb{R}^2 \qquad & \phi_r^{-1}(x) := \frac{x}{r - \|x\|}. \end{aligned}$$

This diffeomorphism allows us to go from disk to the plane and back. Since the plane carries a vector space structure, the *scaling* operation makes sense there. So we just go from disk to the plane, scale all points by some fixed factor α and then go back to the disk:

$$\Psi_{r,\alpha} := \phi_r \circ (\alpha \cdot -) \circ \phi_r^{-1}.$$

For the last part of the whole exercise, we will need the inverse of $\Psi_{r,\alpha}$, this is obviously given by an analogous operation, but this time we scale with α^{-1}, that is:

$$\Psi_{r,\alpha}^{-1} = \Psi_{r,\alpha^{-1}}.$$

We have included a possibility to change the brush size with “+” and “-”, as well as to reset the resulting mess with `Ctrl+R`. The inflation/deflation effect is demonstrated on the portrait of Escher.

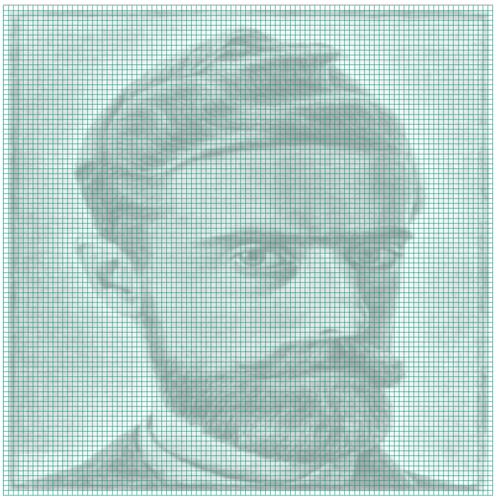

Figure 1: Original portrait of Maurits Cornelis Escher with a grid showing the subdivision into small polygons.

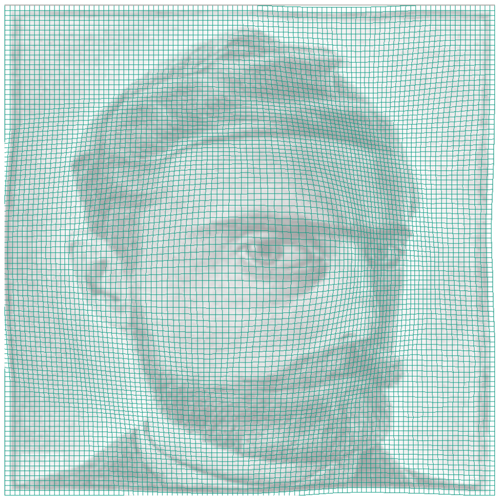

Figure 2: Locally warped portrait of Escher, producing a “fish-eye” effect.

For the deformation in (h-i), we decided to use the distance in texture coordinates, not in world-coordinates. This has the effect that the whole grid behaves more like a sheet of elastic material, and less like a lump of some very viscous sticky liquid. In particular, clicking in y and dragging to x produces the inverse transformation of clicking in x and then dragging to y. In contrast, if one uses the distance in world-coordinates, then two points that once overlap (i.e. have the same world coordinates) can not be separated again.

The distance is computed as

$$r := \frac{2}{N}\sqrt{dr^2 + dc^2}$$

where dr and dc are integer *index* differences of a grid vertex and the dragged vertex, 2 is the width of the originally rendered rectangle, and N is the number of vertex points. Then we weight the offset of the picked point depending on the distance:

$$r \mapsto \exp\left(-\frac{r^2}{\sigma^2}\right)$$

where we used the brush width as σ to make the usage more or less intuitive. Escher seems happy with this choice of the weight function: