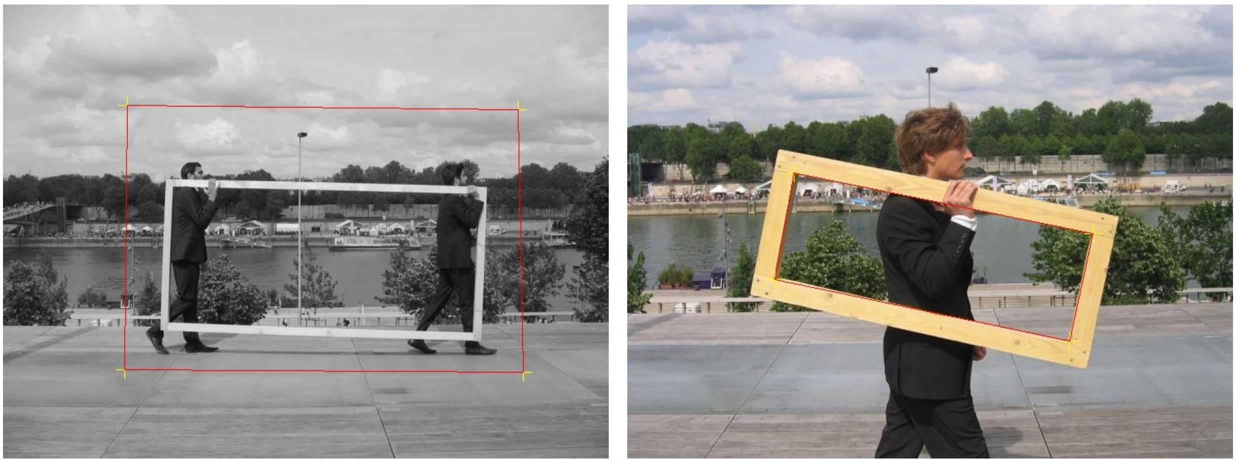
Report for Image Warping

计 76 张翔 2017011568

2020 年 3 月 15 日

1 Affine warping

如图，在程序中选择原图以及目标图对应的 4 对点，可以将相应部分变换到目标上



(a) Source

(b) Target

图 1: 原图以及目标图的选区示意

两张图的变换视为仿射变换，在齐次坐标系下，使用如下方式可以描述这种变换

$$\mathbf{T} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{1}$$

由于有 4 对点，可以得到如下的超定方程

$$\mathbf{Ax} = \mathbf{A}_{(8\times 6)} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \mathbf{b} \tag{2}$$

上述方程可以用最小二乘的方式求解：

$$\mathbf{x} = (\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{b} \tag{3}$$

由此可以确定变换矩阵，对变换取逆得 $\mathbf{T}^{-1}$ 后可以从目标图的像素点位置 (x', y') 推知原图的位置 (x, y)，从而取到相应位置的颜色。由于变换后得到的坐标不一定为整数，这里有三种方式取得需要的颜色

- Round to Nearest：坐标变为 (Round(x), Round(y))

- Bilinear Interpolation: 考虑 $(x\,y)$ 周围的 4 个像素，用下式插值

$$f(x,y) \approx \frac{1}{(x_2-x_1)(y_2-y_1)} \begin{bmatrix} x_2-x & x-x_1 \end{bmatrix} \begin{bmatrix} f(x_1,y_1) & f(x_1,y_2) \\ f(x_2,y_1) & f(x_2,y_2) \end{bmatrix} \begin{bmatrix} y_2-y \\ y-y_1 \end{bmatrix} \tag{4}$$

- Bicubic Interpolation: 同样考虑周围 4 个像素，插值函数是如下的三次函数

$$p(x,y) = \begin{bmatrix} 1 & x & x^2 & x^3 \end{bmatrix} \mathbf{A}_{(4\times4)} \begin{bmatrix} 1 \\ y \\ y^2 \\ y^3 \end{bmatrix} \tag{5}$$

  可以用差分的方法计算出这四个点的 x, y 偏导以及 xy 混合导数，加上四个点的函数值，可以构造线性方程组解出 $\mathbf{A}$ 的系数，从而该小方块内任意点的值均可以用插值函数算出。

上面的两幅图变换后可以得到如下结果

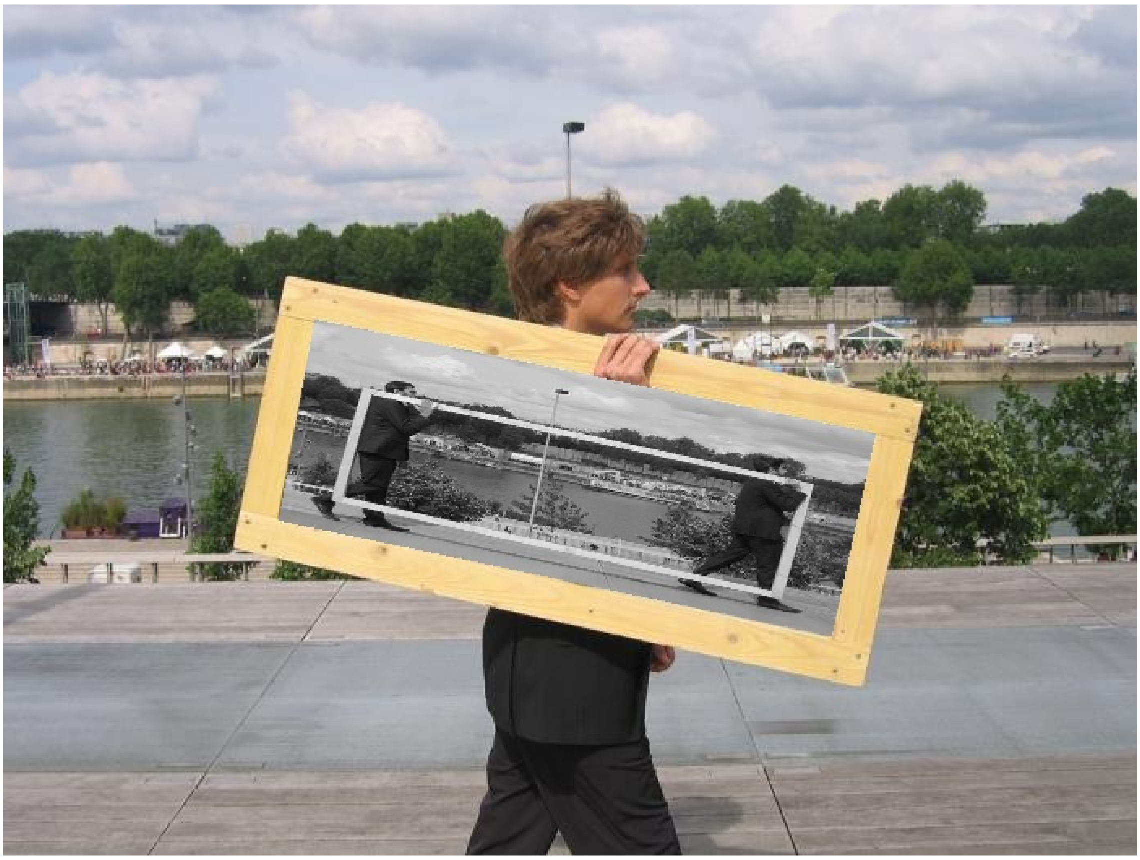


图 2: 变换后的结果

(a) Round to Nearest (b) Bilinear (c) Bicubic

图 3: 三种插值方法结果对比

可以看出后两种插值方法比直接取最近的方法生成的图像更为平滑，而 Bicubic 插值比 Bilinear 保留的细节要更多一些。

确定目标图需要填充的位置时，可以先将边界用直线相连，设置特殊标记（边界标记），然后使用图形学课程上介绍过的 Seed Fill 算法，对于矩形区域填充，优化后的版本大致如下

```
Seed-fill (seed):
    // ... fill the boundaries (just like drawing lines)
    // and label them as filled prior to function call
    if node.label != unfilled: return
    Q = Queue()
    Q.add(seed)
    while !Q.empty():
        elem = Q.pop()
        w=e=elem
        while(w.label != filled): w=west(w)
        while(e.label != filled): e=east(w)
        foreach n between(w,e)
        n.color=get_replace_color(n.x, n.y)
        n.label = filled
        if(north(n).label == unfilled): Q.add(north(n))
        if(south(n).label == unfilled): Q.add(south(n))
```

2 Spherical warping

该变换通过下面的保角映射，将原图映射到目标图像

$$
\begin{aligned}
x_0 &= \frac{2}{\pi} d_0 \phi \sin\theta \\
y_0 &= \frac{2}{\pi} d_0 \phi \cos\theta
\end{aligned} \tag{6}
$$

其中 $\phi = \sin^{-1}\left(\frac{\rho}{\rho_0}\right)$,$\theta = \tan^{-1}\left(\frac{x_1}{y_1}\right)$,$\rho = \sqrt{x_1^2 + y_1^2}$。参数 $d_0 = \frac{1}{2}\min(H_{in}, W_{in})$, $\rho_0 = \frac{1}{2}\min(H_{out}, W_{out})$。这里的坐标都是原点位于图片中心的坐标。相较于课件，我把 d_0 函数取为 min 而不是 max，因为取 max 会使得球顶部和底部部分像素（对应 $\phi = \frac{\pi}{2}$, $\theta = \pm\frac{\pi}{2}$）在原图找不到匹配点（课件给出的样例图片在这一部分为灰色），这样做的缺点是会损失原图中左右两边的一部分信息。如果按照课件给出的方法，

只需要对映射后的值在原图中找一个最近的点，也可以得到一个看起来不错的结果（下面的 Target_2），这张图比 Target 有更大一些的视野

算法比较简单，因为变换的形式是逆变换，用如下方法即可:

```
Transform(dst, src):
    for (i, j) in dst:
        dst(i, j).color = interpolate(src, inverse_transform(i, j))
```

最后结果如下

(a) Source (b) Target (c) Target_2

图 4: Spherical warping

3 Cylindrical warping

考虑将一张图片 $(H \times W)$ 卷绕在高为 H、周长为 $2W$ 的圆柱上，之后将它平行投影到目标图片上 $H \times (\frac{2W}{\pi})$，那么应有如下的坐标变换 (逆变换)

$$\begin{aligned} x_0 &= x_1 \\ y_0 &= \frac{W}{\pi}\left(\pi - \cos^{-1}\left(\frac{y_1 - \frac{W}{\pi}}{\frac{W}{\pi}}\right)\right) \end{aligned} \tag{7}$$

可以使用和 Sphere warping 一样的方法进行处理，如下，Target 是将宽度 W 侧卷绕在圆柱上，而 Target_2 是 Target 图片将高度 H 侧卷绕在圆柱上得到的结果。

(a) Source (b) Target (c) Target_2

图 5: Cylindrical warping