计算机图形学第二次作业实验报告

- ID：999（旁听）
- 姓名：袁保杰
- 学号：PB21111714

实验描述

本实验要求：

- 必做部分：使用 Eigen 库，实现至少两种图像变形（Warping）算法
  - Inverse distance-weighted interpolation method (IDW)
  - Radial basis functions interpolation method (RBF)
- 选做部分：
  - 使用 Annoy 库求最近邻点，对变形后的图像实现白缝填充
  - 使用 ANN 库，训练神经网络进行拟合，实现图像变形算法

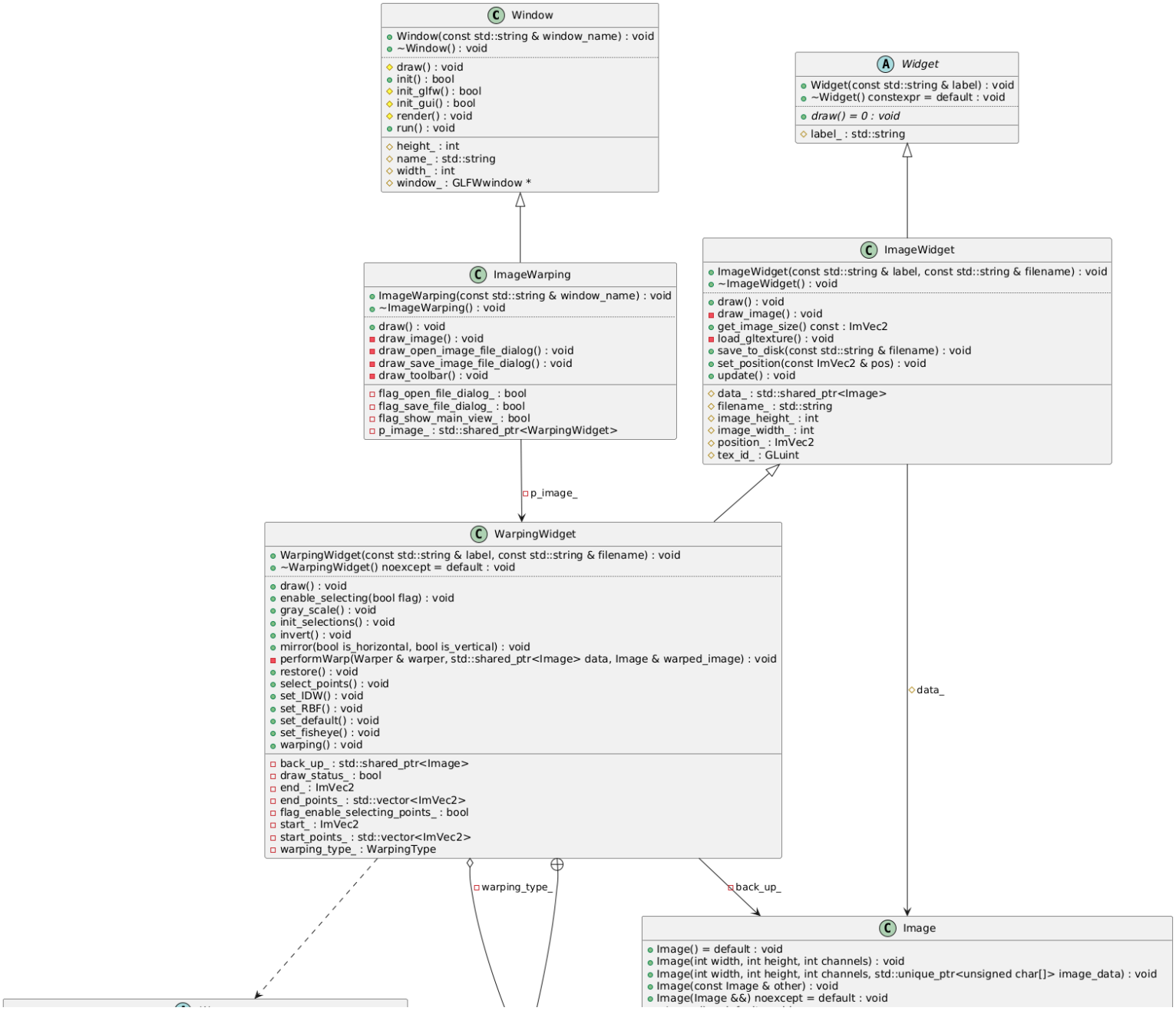
~~由于这周被突发状况淹没~~，我仅完成了必做部分和白缝填充算法，未完成神经网络拟合的部分，比较遗憾。

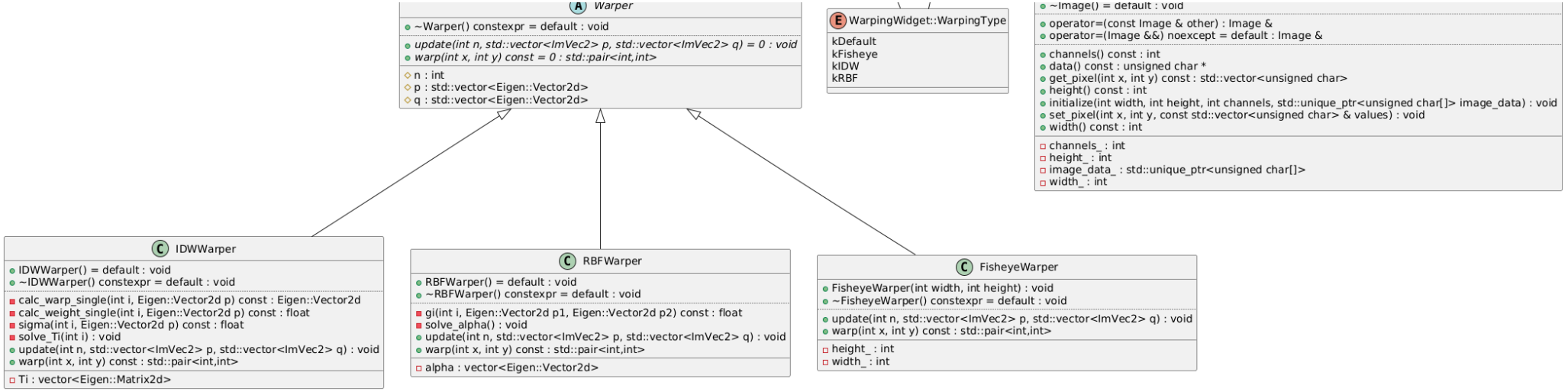
实验过程

整体设计

类图　下所示，`IDWWarper`、`RBFWarper`、`FisheyeWarper` 均为 `Warper` 的派生类，提供相同的接口。`update` 接口用于更新对象的状态，这

些状态用于存储预处理好的数据，从而加速计算。





注意到 `FisheyeWarper` 也被抽象出来，用于减少 `WarpingWidget::warping` 方法的重复代码，简化后的代码　下：

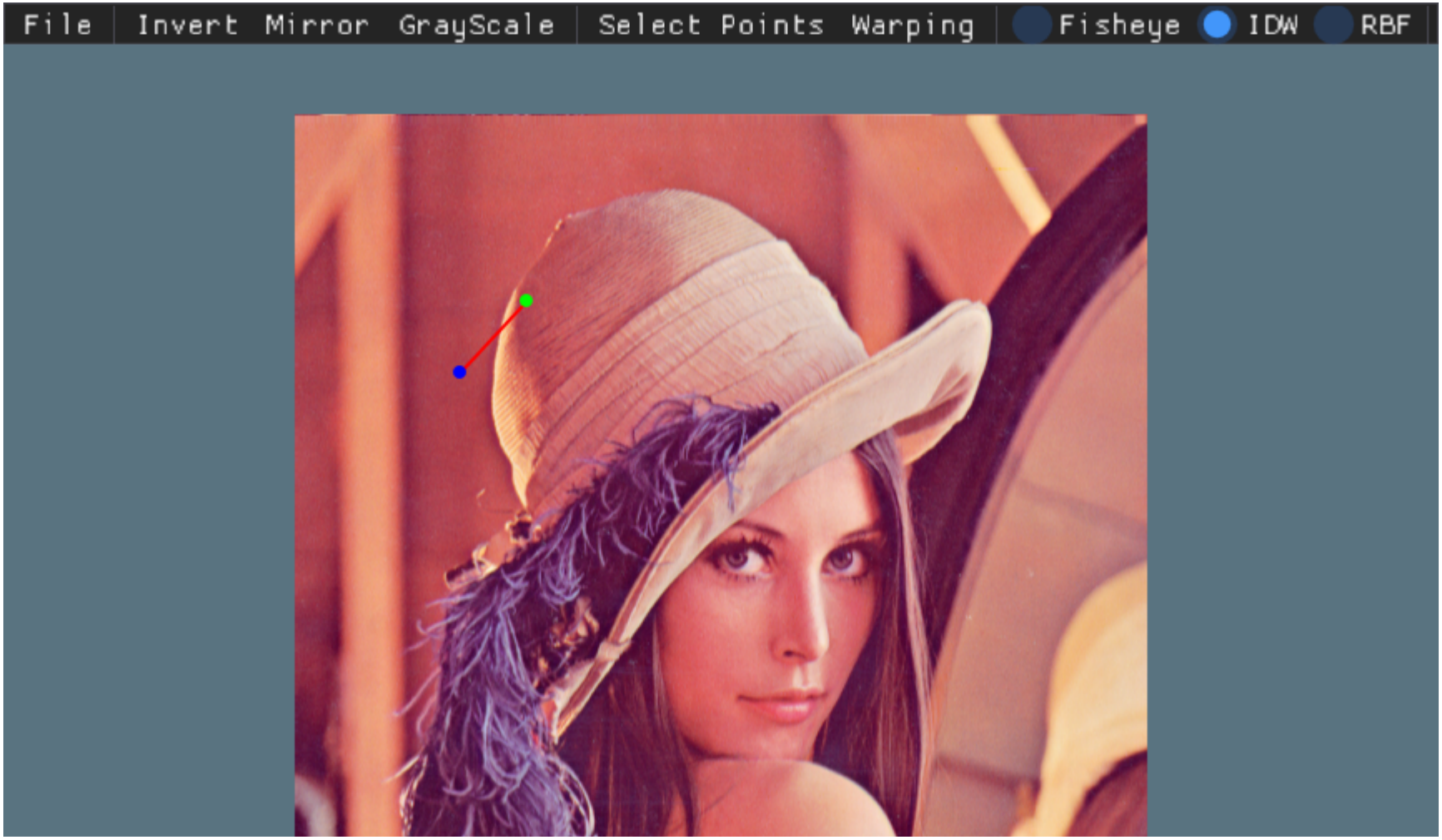
```
std::unique_ptr<Warper> warper;
switch (warping_type_)
{
    case kDefault: break;
    case kFisheye:
        warper = std::make_unique<FisheyeWarper>(data_->width(), data_->height());
        break;
    case kIDW:
    {
        auto idw = std::make_unique<IDWWarper>();
        idw->update(start_points_.size(), start_points_, end_points_);
        warper = std::move(idw);
        break;
    }
    case kRBF:
    {
        auto rbf = std::make_unique<RBFWarper>();
        rbf->update(start_points_.size(), start_points_, end_points_);
        warper = std::move(rbf);
        break;
    }
    default: break;
}

if (warper) {
    performWarp(*warper, data_, warped_image);
    *data_ = std::move(warped_image);
    update();
}
```

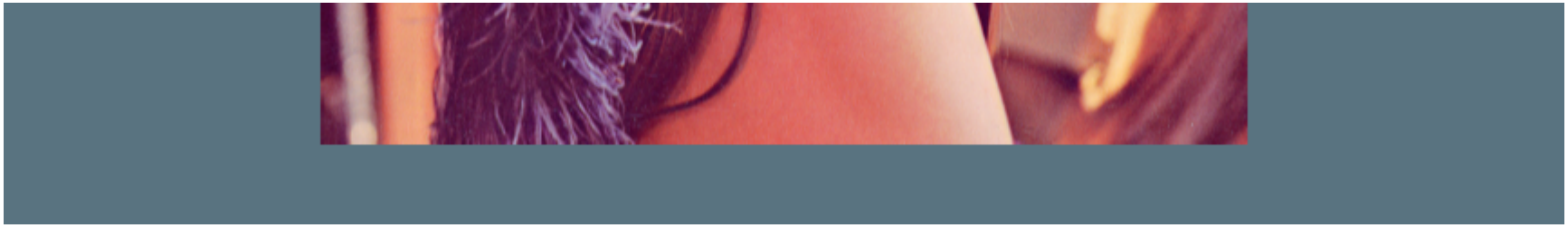
IDW 算法

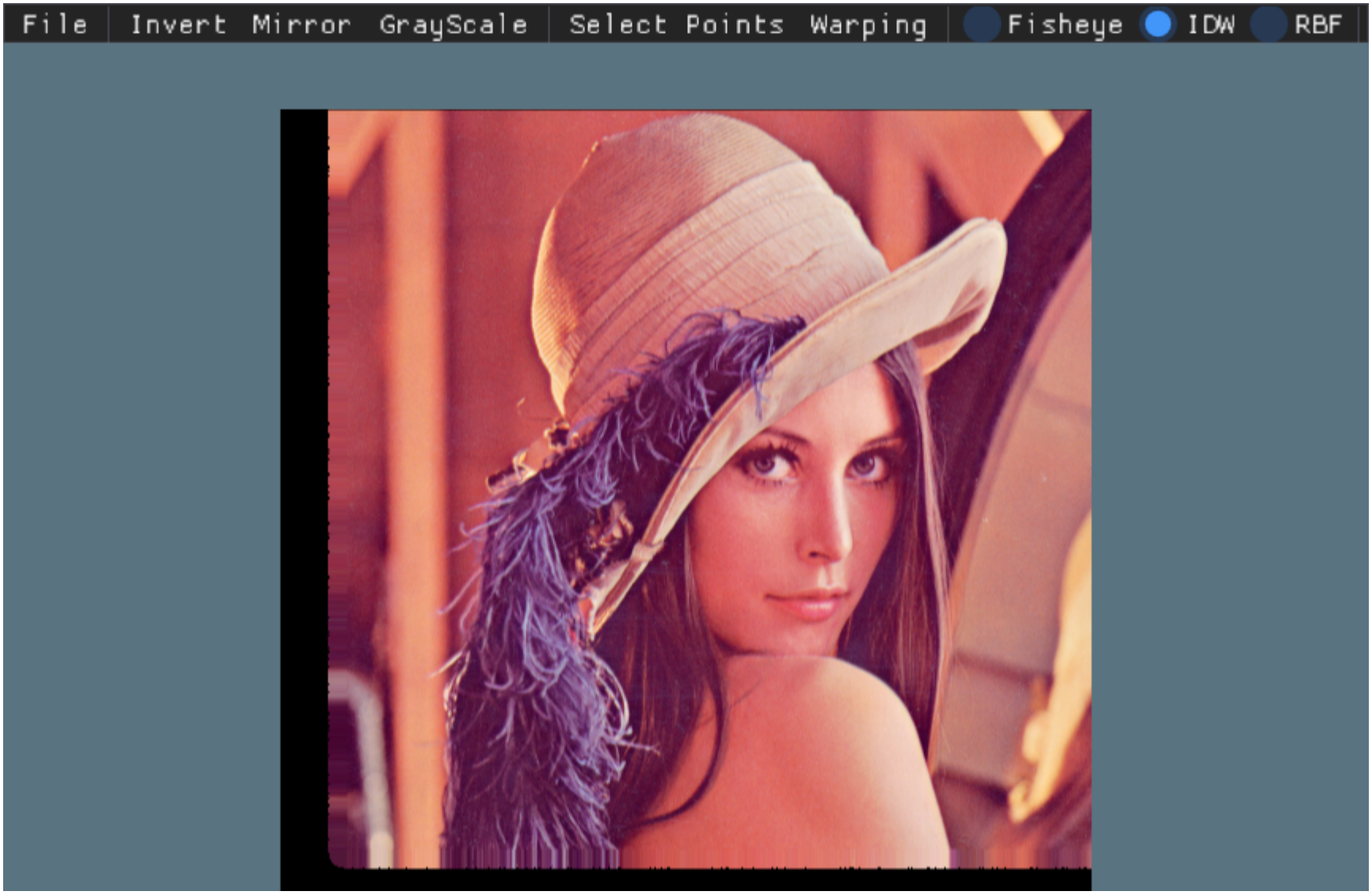
根据给出的公式，需要注意边界条件：

- 当给出的点对小于 3 个时，需要特判处理，我这里直接将 `T_i` 置为单位矩阵了
- 当给出的点对大于等于 3 个时，按公式实现算法即可

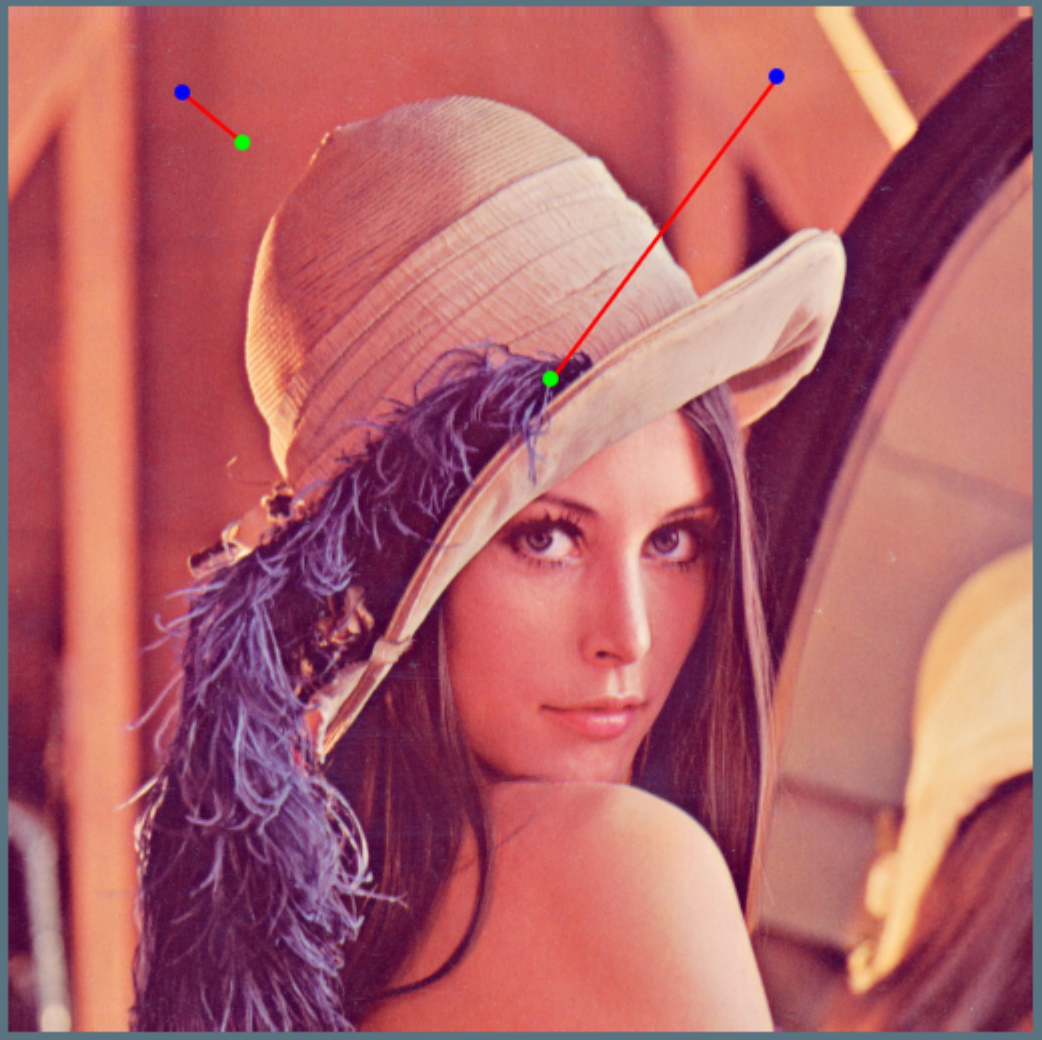
实验结果　下：

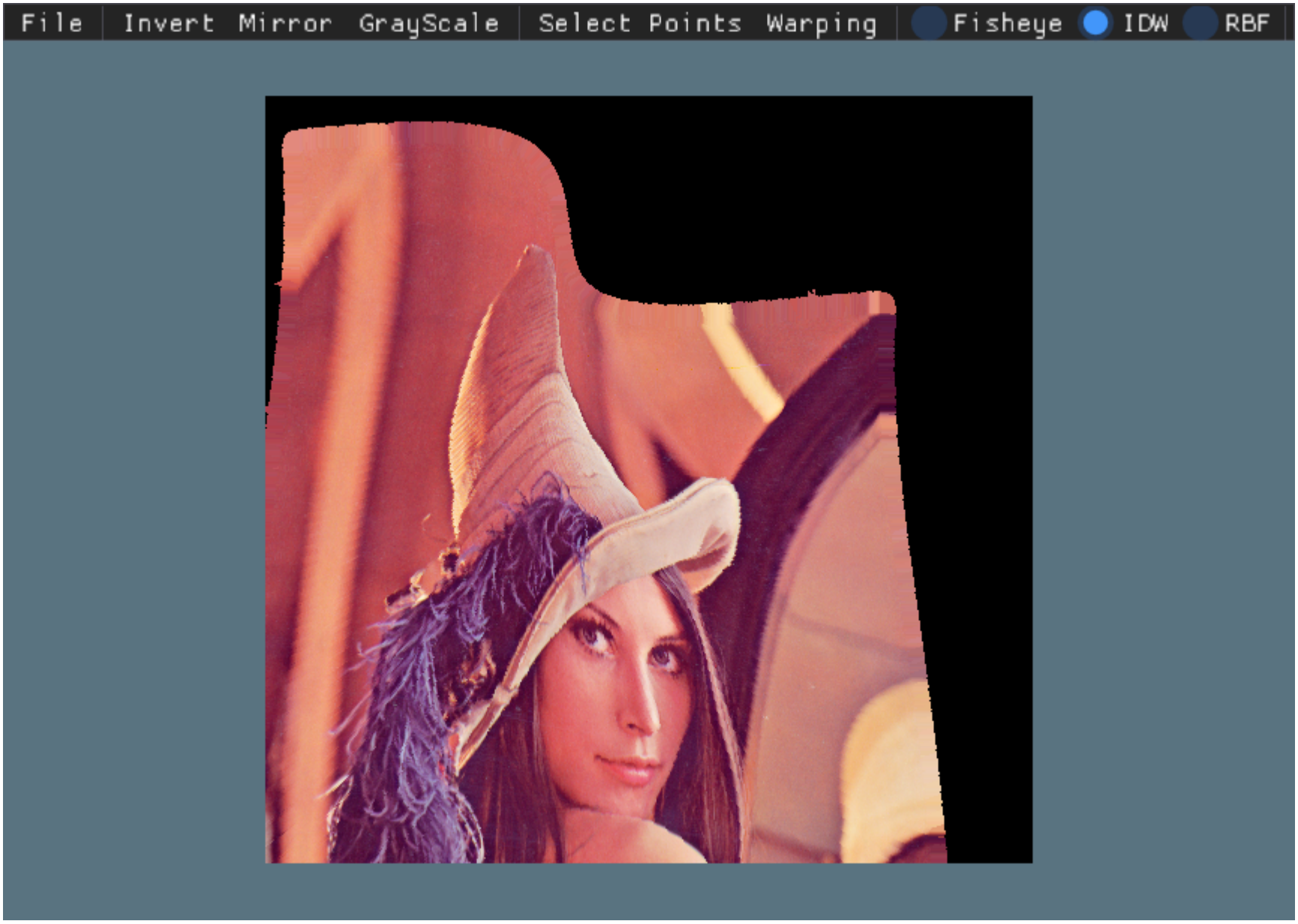





File
Invert
Mirror
GrayScale
Select Points
Warping
Fisheye
IDW
RBF

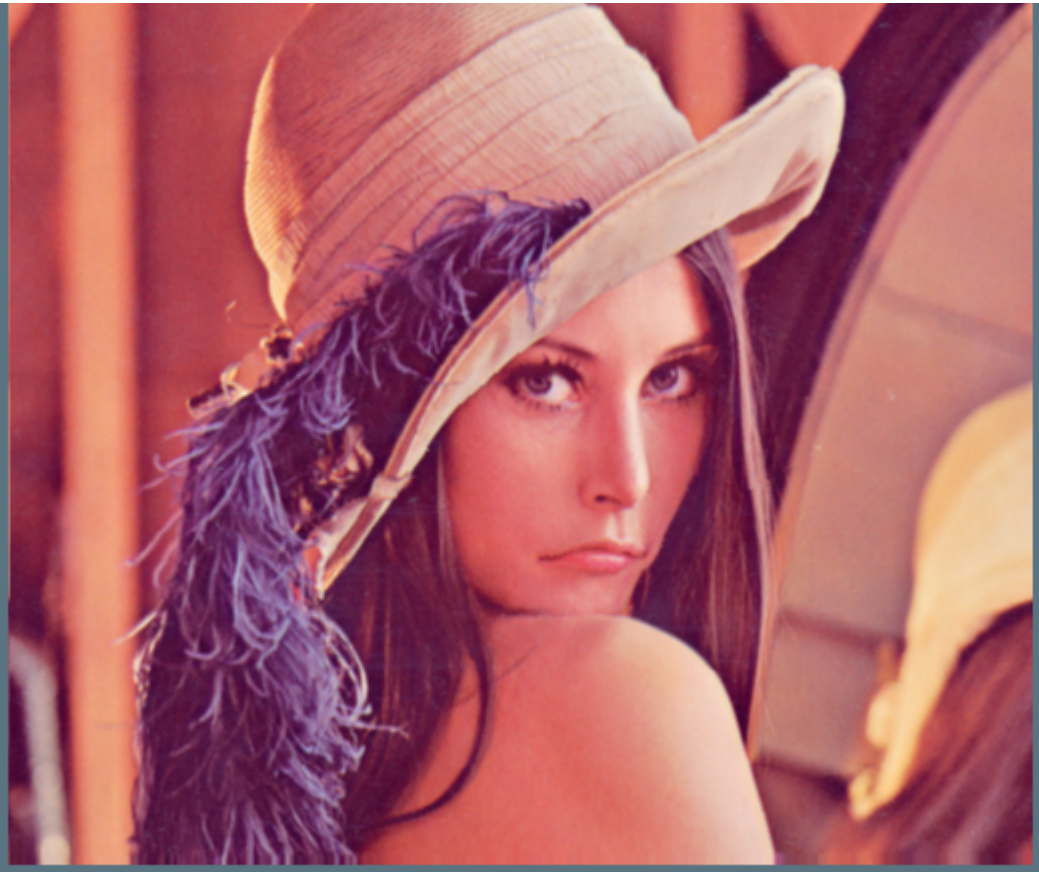
File Invert Mirror GrayScale Select Points Warping Fisheye IDW RBF



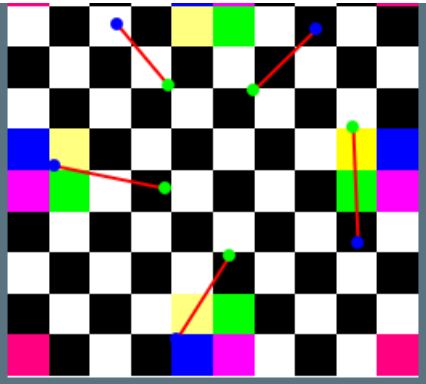

File
Invert
Mirror
GrayScale
Select Points
Warping
Fisheye
IDW
RBF


File
Invert
Mirror
GrayScale
Select Points
Warping
Fisheye
IDW
RBF
Restore


File
Invert
Mirror
GrayScale
Select Points
Warping
Fisheye
IDW
RBF
Restore



File | Invert | Mirror | GrayScale | Select Points | Warping | Fisheye | IDW | RBF | Restore



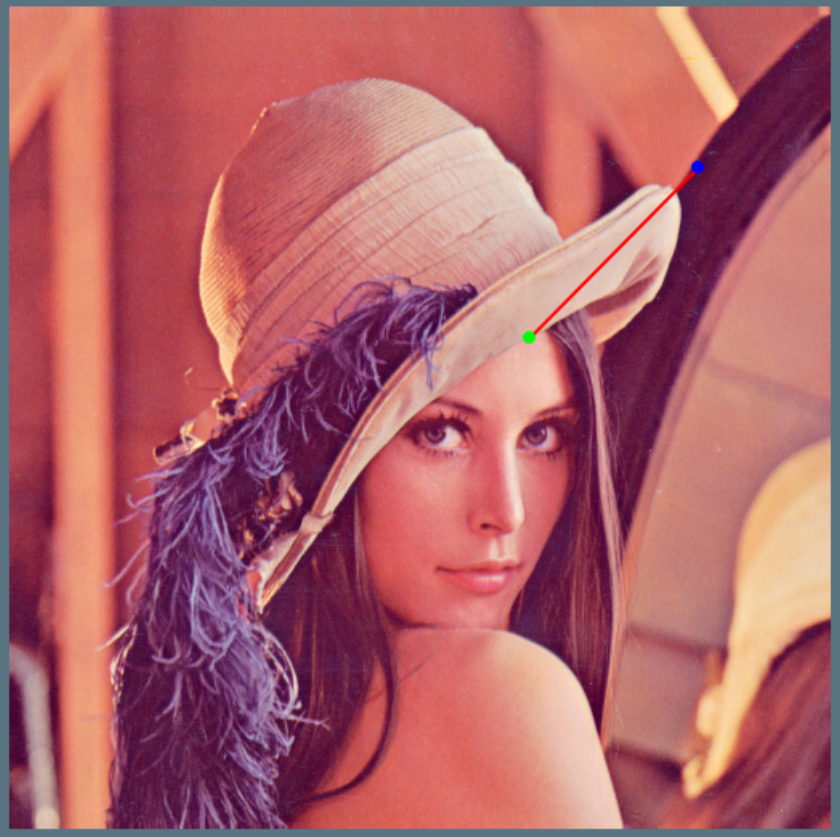
File | Invert | Mirror | GrayScale | Select Points | Warping | Fisheye | IDW | RBF | Restore

RBF 算法

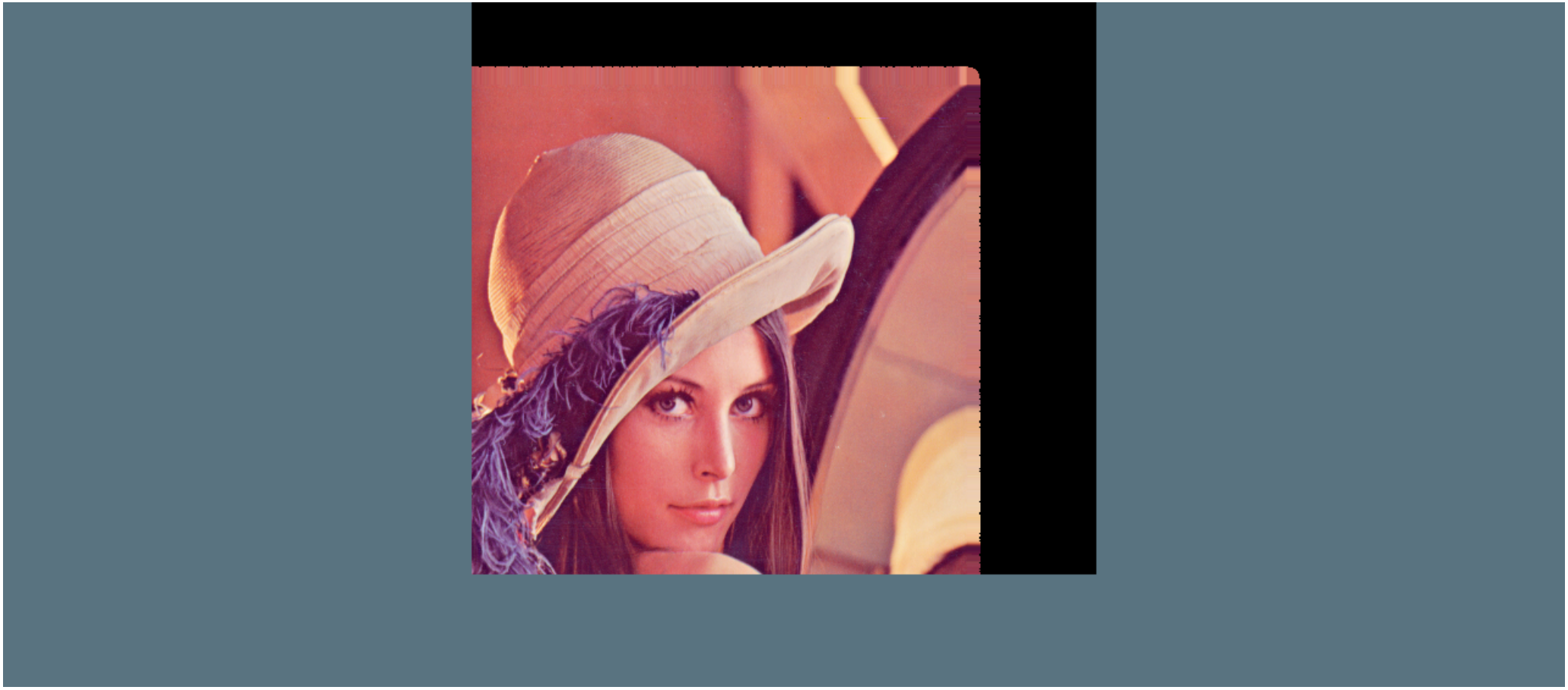
同样需要注意边界条件：

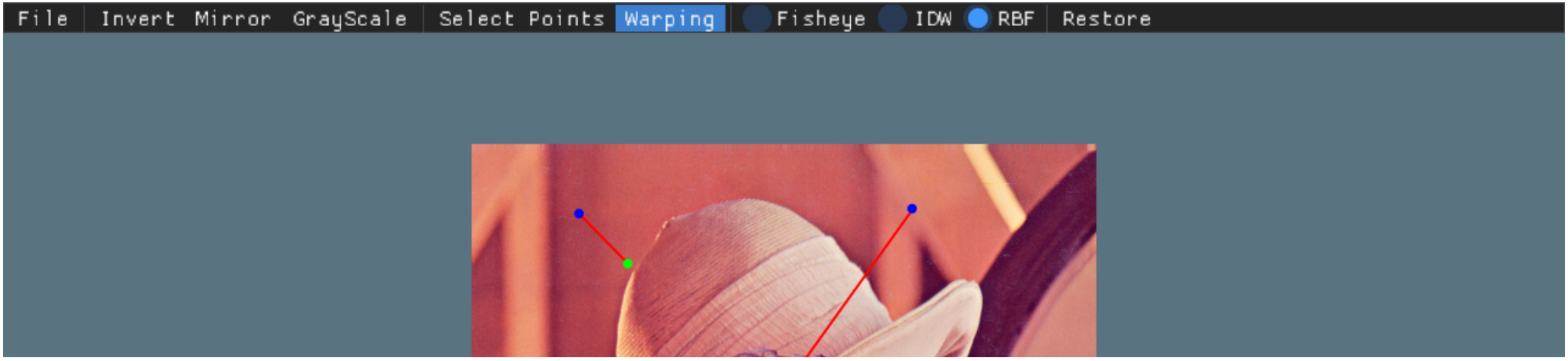
```
    // 当只有一个控制点时，直接采用平移变换
    if (n == 1)
    {
        alpha.resize(1);
        alpha[0] = this->q[0] - this->p[0];
    }
    else if (n > 1)
    {
        solve_alpha();
    }
```

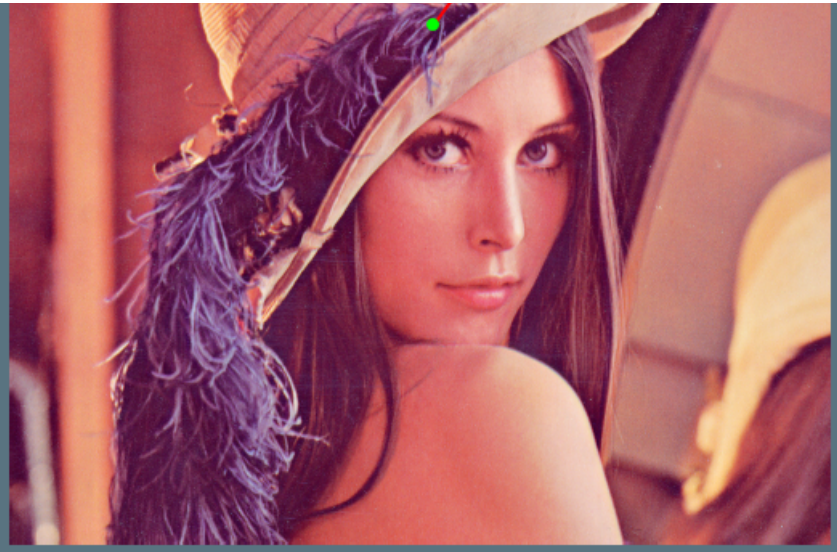
实验结果　下：



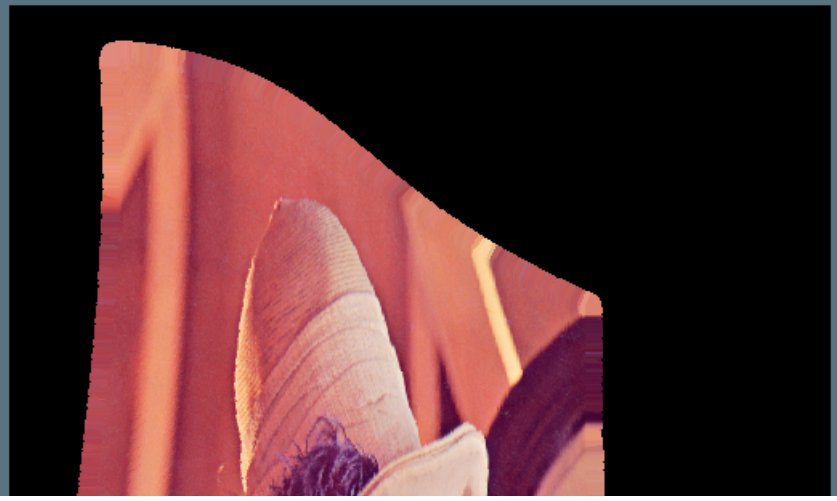
File | Invert | Mirror | GrayScale | Select Points | Warping | Fisheye | IDW | RBF | Restore

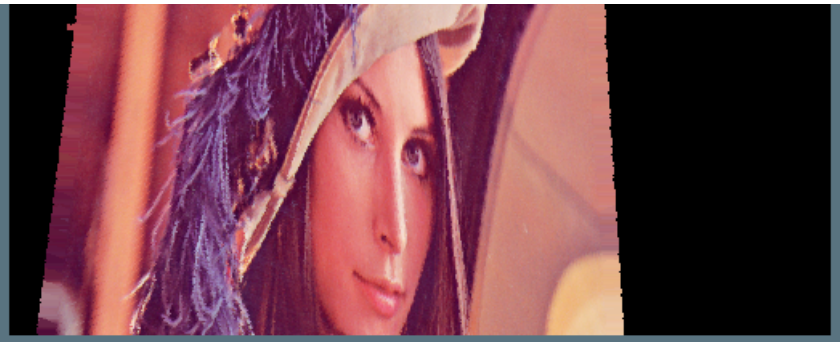



File
Invert
Mirror
GrayScale
Select Points
Warping
Fisheye
IDW
RBF
Restore

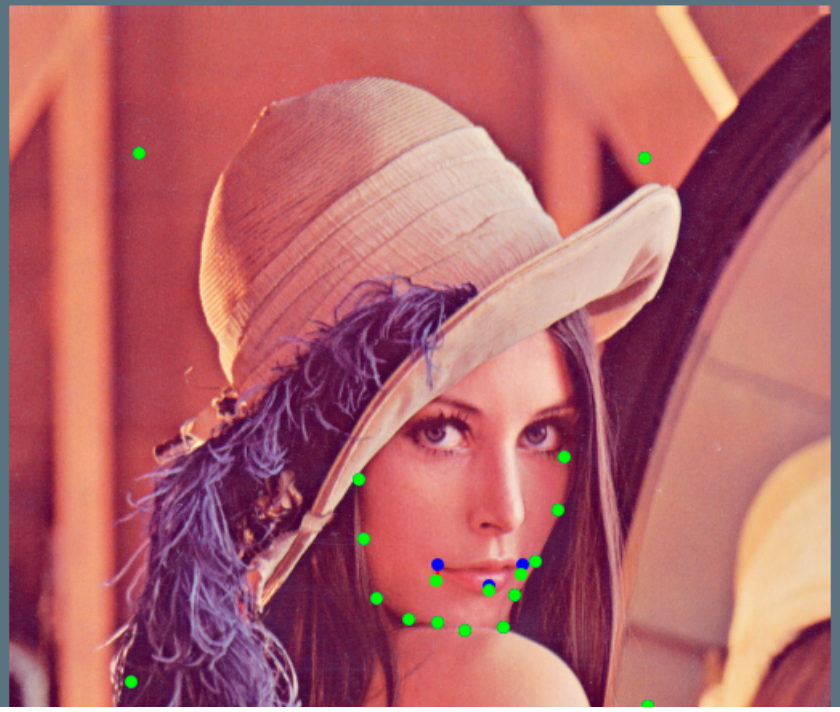


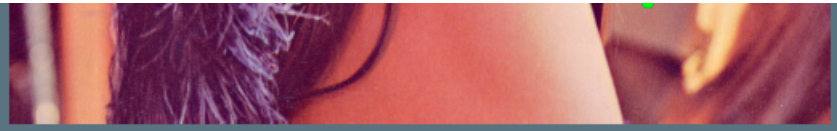
File | Invert | Mirror | GrayScale | Select Points | Warping | Fisheye | IDW | RBF | Restore



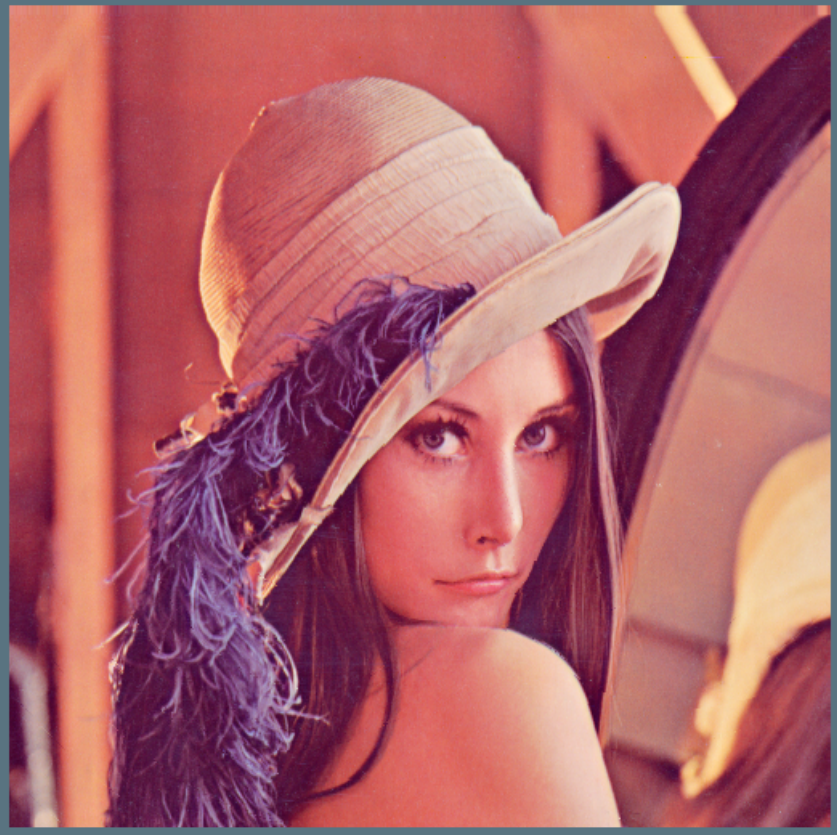


File | Invert | Mirror | GrayScale | Select Points | Warping | Fisheye | IDW | RBF | Restore

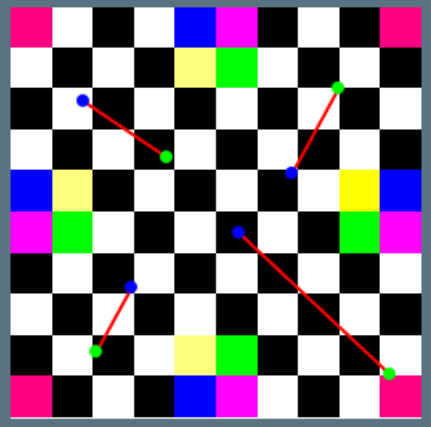




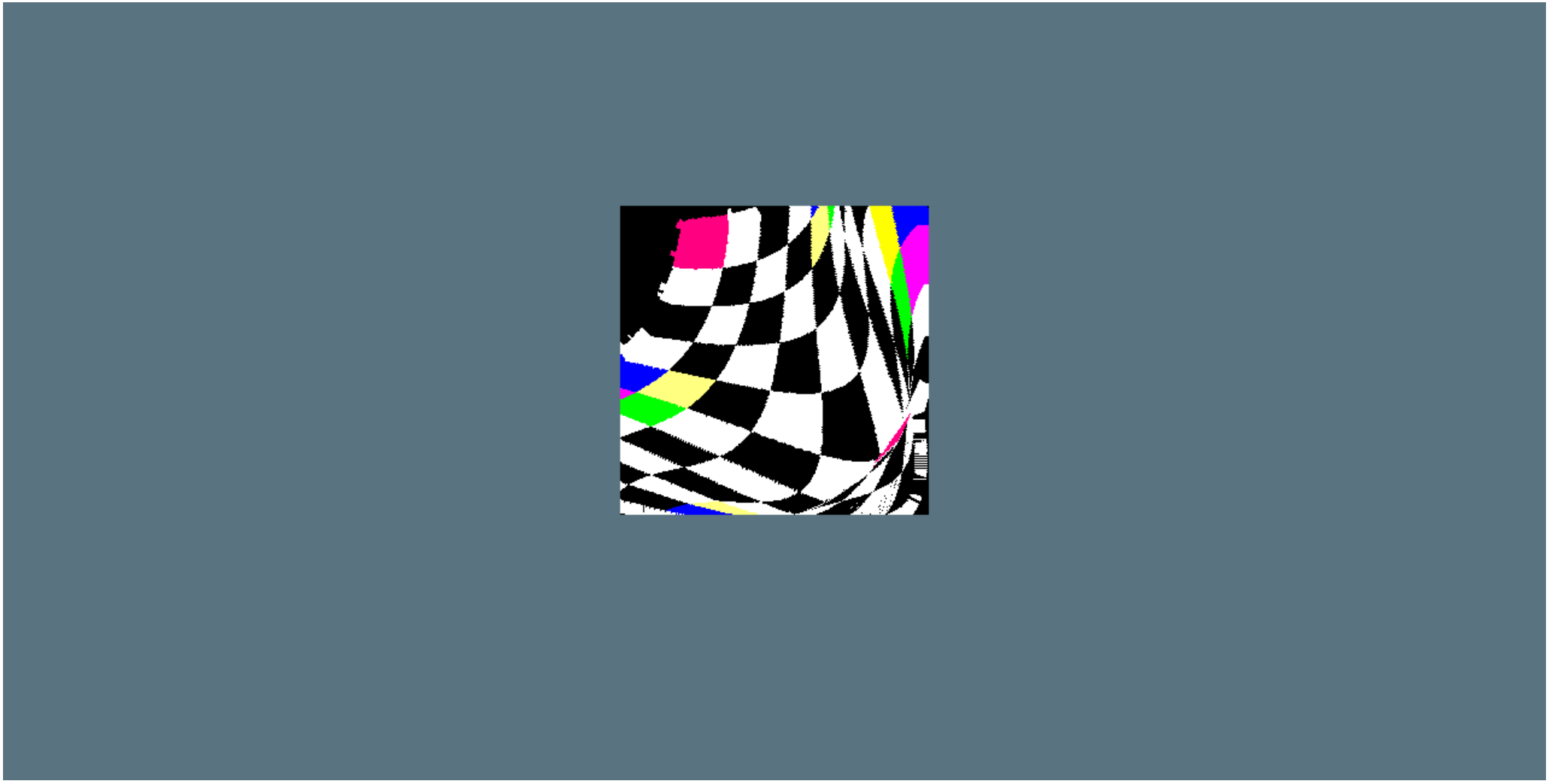
File | Invert | Mirror | GrayScale | Select Points | Warping | Fisheye | IDW | RBF | Restore



File | Invert | Mirror | GrayScale | Select Points | Warping | Fisheye | IDW | RBF | Restore



File | Invert | Mirror | GrayScale | Select Points | Warping | Fisheye | IDW | RBF | Restore



白缝填充（Optional）

作为 `WarpingWidget::performWarp` 的一部分实现，主要函数是 `fillHolesWithNearestNeighbor`。由于时间紧迫，仅实现了最近邻点插值算法，好处是实现简单，坏处是得到的图像锯齿较为明显，效果上不　双线性插值、双三次插值等方法。

这里实现的白缝填充算法不填充变换后图像外的无色区域，而我们只有变换后的像素集合。为了区分一个点是在这些像素之内还是之外，一个可能

的想法是使用计算几何方法，先画出这些像素对应的凸包，再判断这个点是在凸包之外还是之内。然而，计算几何算法手动实现存在难度，相关库也暂时没有时间来研究使用方法。

因此，这里我考虑了一个实现简单的方法，但需要牺牲填充后图像的质量。对于画布上的任意一点 `(x, y)`，寻找其最近邻像素 `(x1, y1)`，两者距离记作 `dis1`；同时寻找 `(x1, y1)` 的最近邻像素 `(x2, y2)`，两者距离记作 `dis2`。 果满足条件 `dis1 <= μ * dis2`，则认为点 `(x, y)` 在变换后的图像之内，使用最近邻像素来填充颜色；否则，不填充颜色。

上面的 `μ >= 1`，是一个可以按照需求取值的正常数。容易看出，`μ` 取得越大，对于变换后图像像素稀疏程度的接受程度越强，但填充后图像的边缘也更有可能也会出现奇怪的变化。正常 `μ` 取 5 左右就可以得到较为满意的效果。

实现代码 下：

```cpp
// 辅助函数：对结果图像中的空洞（背景色为黑）进行最近邻重采样填充
void fillHolesWithNearestNeighbor(const std::vector<pair<int, int>>& new_pixels, Image &warped_image) {
    const int dim = 2;
    int n_points = new_pixels.size();

    Annoy::AnnoyIndex<int, float, Annoy::Euclidean, Annoy::Kiss32Random, Annoy::AnnoyIndexSingleThreadedBuildPolicy> index(d
    for (int i = 0; i < n_points; i++) {
        float vec[dim] = { static_cast<float>(new_pixels[i].first), static_cast<float>(new_pixels[i].second) };
        index.add_item(i, vec);
    }
    index.build(10);

    int width = warped_image.width();
    int height = warped_image.height();
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            // 对于 (x, y)，找其最近邻点 (x1, y1)
            float query[dim] = { static_cast<float>(x), static_cast<float>(y) };
            std::vector<int> result;
            std::vector<float> distances;
            index.get_nns_by_vector(query, 1, -1, &result, &distances);
            if(result.empty()) continue;
            auto [x1, y1] = new_pixels[result[0]];
            float dis1 = distances[0];

            // 对于 (x1, y1)，找其最近邻（要求返回 2 个，第二个为真正的邻居）
            float query2[dim] = { static_cast<float>(x1), static_cast<float>(y1) };
            std::vector<int> result2;
            std::vector<float> distances2;
            index.get_nns_by_vector(query2, 2, -1, &result2, &distances2);
```

```
            if(distances2.size() < 2) continue;
            float dis2 = distances2[1];

            // 如果 dis1 <= dis2 * sigma，则认为 (x, y) 为图片内部产生的空洞，需要填充
            // 实际测试下取 8 还是比较合适的（
            if (dis1 <= dis2 * 8) {
                if (x1 >= 0 && x1 < width && y1 >= 0 && y1 < height) {
                    auto fill_color = warped_image.get_pixel(x1, y1);
                    warped_image.set_pixel(x, y, fill_color);
                }
            }
        }
    }
}
```