重 庆 交 通 大 学

学生实验报告

课 程 名 称：数 字 图 像 处 理 .

开 课实验室：软 件 实 验 中 心 .

学 院：信息学院年级物联网工程专业 2 班

学 生 姓 名：李骏飞学 号 632109160602

指 导 教 师：蓝 章 礼 .

开 课 时 间：2023 至 2024 学年第 二 学期

成 绩	
教师签名	

实验项目名称		实验二：图像的放大和缩小			
姓名	李骏飞	学号	**632109160602**	实验日期	**2024.4.10**
教师评阅： 1:实验目的明确□A□B□C□D 2:内容与原理□A□B□C□D 3:实验报告规范□A□B□C□D 4:实验主要代码与效果展示□A□B□C□D 5:实验分析总结全面□A□B□C□D					
实验记录					

一、**实验目的**

完成图像放大与缩小操作。

基本要求：完成简单倍数的放大和缩小。

扩展内容：完成旋转、任意尺寸的放大缩小。

二、**实验主要内容及原理**

（1）图像旋转：

在图像处理中，图像灰度化是将彩色图像转换为灰度图像的过程。灰度

如果平面上的点绕原点逆时针旋转 θº，则其坐标变换公式为：

$$x' = xcos\theta + ysin\theta$$

$$y' = -xsin\theta + ycos\theta$$

其中，(x, y)为原图坐标，(x', y')为旋转后的坐标。它的逆变换公式为：

$$x = x'cos\theta - y'sin\theta$$

$$y = x'sin\theta + y'cos\theta$$

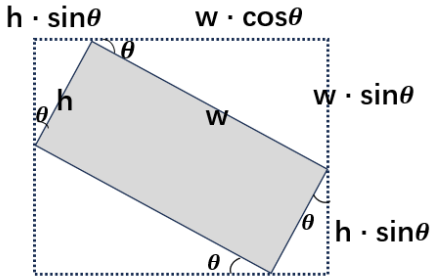
矩阵形式为：

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

和缩放类似，旋转后的图像的像素点也需要经过坐标转换为原始图像上的坐标来确定像素值，同样也可能找不到对应点，因此旋转也用到插值法。在此选用性能较好的双线性插值法。双线性插值是一种常用的图像插值方法，用于在已知离散点的情况下，通过插值计算得到目标点的像素值。在旋转后的图像中，由于像素位置不再对应整数坐标，需要通过双线性插值来估计旋转后像素的值。

图像旋转原理如下图所示：

h是原图像的高度，w是原图像的宽度



旋转之后的图像：

高度：Height = w * sinθ + h * sinθ

宽度：Width = w * cosθ + h * sinθ

（2）图像缩小：

在 C# 中实现图像缩小的原理通常涉及两种方法：最近邻插值和双线性插值。

双线性插值相对于最近邻插值具有更好的图像质量，因为它考虑了目标像素周围的像素值，提供了更平滑和更准确的缩小结果。然而，双线性插值的计算复杂度较高，可能会导致一些性能开销。

（3）图像放大

图像放大的原理是通过对图像像素进行插值计算来实现的。插值是一种基于已知数据点的数学计算方法，用于估计未知位置上的值。

常用的图像放大算法有以下几种：

1、最近邻插值：

最近邻插值是最简单的插值方法之一。它通过在放大后的图像中，对每个像素位置找到最近的原始图像像素，并将其值赋给放大后的像素。这种方法简单快速，但可能会导致图像锯齿状边缘和失真。

2、双线性插值：

双线性插值是一种线性插值方法，通过对目标像素周围的四个邻近像素进行加权平均来计算新像素的值。它考虑了目标像素与周围像素之间的距离和权重，从而实现图像的平滑缩放。双线性插值可以减轻图像锯齿状边缘，并提供一定程度的模糊减少，但仍可能导致细节损失。

3、双三次插值：

双三次插值是一种更高级的插值方法，它通过在目标像素周围的 16 个邻近像素上执行三次插值来计算新像素的值。它考虑了更多的邻近像

素，并使用更复杂的插值函数来计算像素值，从而提供更好的细节保留和图像清晰度。双三次插值通常比双线性插值更耗时，但可以提供更好的图像质量。

三、实验环境

Windows11

Visual Studio2021

C#语言

四、实验主要代码与效果展示

图像旋转任意角度

➢ 算法描述：

旋转后的图像的像素点需要经过坐标转换为原始图像上的坐标来确定像素值，可能找不到对应点，因此旋转用到**插值法**。这里选用性能较好的双线性插值法。

这里的Rotation方法用于处理图像旋转，该方法接受三个参数：srcBmp为**原始图像**，degree为**旋转角度**，dstBmp为**旋转后的目标图像**。方法返回一个布尔值，表示旋转是否成功。

该函数中在开始定义了源图像和目标图像的BitmapData对象，还有相关的旋转变量。并根据源图像的尺寸以及旋转角度确定旋转点，即中心点。

（1）计算旋转后图像**宽高**的方程如下：

```
int widthDst=(int)(srcBmp.Height * Math.Abs(sin) + srcBmp.Width * Math.Abs(cos));
int heightDst=(int)(srcBmp.Width* Math.Abs(sin) + srcBmp.Height * Math.Abs(cos));
```

srcBmp.Height * Math.Abs(sin)表示源图像高度乘以正弦值的绝对值，即旋转后图像在宽度方向上的增量，srcBmp.Width * Math.Abs(cos)表示源图像宽度乘以余弦值的绝对值，即旋转后图像在宽度方向上的保持不变的部分，最终，widthDst表示旋转后图像的宽度，即增量和保持不变部分之和。同理高度也是如此。

（2）计算中心点的方程如下：

```
int dx = (int)(srcBmp.Width / 2 * (1 - cos) + srcBmp.Height / 2 * sin);
```

int dy = (int)(srcBmp.Width / 2 * (0 - sin) + srcBmp.Height / 2 * (1 - cos));

旋转点的位置由源图像的中心点经过一定的位移计算得到，其中 srcBmp.Width / 2 和 srcBmp.Height / 2 分别表示源图像宽度和高度的一半，即源图像的中心点的横坐标和纵坐标，cos 和 sin 是旋转角度的余弦值和正弦值。(1 - cos) 表示旋转角度的余弦值与 1 之差，即余弦值的补数，(1 - cos) * srcBmp.Width / 2 表示余弦值的补数乘以源图像宽度一半，即余弦值的补数对应的横坐标位移，srcBmp.Height / 2 * sin 表示源图像高度一半乘以正弦值，即正弦值对应的纵坐标位移。

（3）双线性插值：

双线性插值计算目标点在源图像中的四个最近邻像素坐标 (Iu, Iv)，四个像素的坐标分别是 (Iu, Iv)、(Iu+1, Iv)、(Iu, Iv+1)、(Iu+1, Iv+1)。然后计算目标点在四个最近邻像素上的权重系数 (a, b)，其中 a 和 b 分别表示 (fu, fv) 相对于 (Iu, Iv) 的水平和垂直距离的比例，对每个颜色通道（在代码中用 k 表示）进行插值计算，若目标点超出源图像范围，将目标点像素值设为灰色。

```
1.  //图像旋转的确定按
2.  private void btnSpinadjust_Click(object sender, EventArgs e)
3.  {
4.      if (!judge_pb_img_exit()) return;
5.      Bitmap bt1 = new Bitmap(original_image.Image);
6.      double angle = 1.0 * trackBar_spin.Value;
7.      //double angle = 1.0 *textBox_showSpin.;
8.      Bitmap bt2 = new Bitmap(original_image.Image);
9.      if (Rotation(bt1, angle, out bt2))
10.     {
11.         pictureBox_show.Image = bt2;
12.     }
13.     else
14.     {
15.         return;
16.     }
17. }
18. public static bool Rotation(Bitmap srcBmp, double degree, out Bitmap dstBmp)
19. {
20.     if (srcBmp == null)
21.     {
22.         dstBmp = null;
23.         return false;
24.     }
25.     dstBmp = null;
26.     BitmapData srcBmpData = null;
27.     BitmapData dstBmpData = null;
28.
29.
```

```
30.     double radian = degree * Math.PI / 180.0;//将角度转换为弧度
31.                                              //计算正弦和余弦
32.     double sin = Math.Sin(radian);
33.     double cos = Math.Cos(radian);
34.     //计算旋转后的图像大小
35.     int widthDst = (int)(srcBmp.Height * Math.Abs(sin) + srcBmp.Width * Math.Abs
    (cos));
36.     int heightDst = (int)(srcBmp.Width * Math.Abs(sin) + srcBmp.Height * Math.Ab
    s(cos));
37.
38.     dstBmp = new Bitmap(widthDst, heightDst);
39.     //确定旋转点
40.     int dx = (int)(srcBmp.Width / 2 * (1 - cos) + srcBmp.Height / 2 * sin);
41.     int dy = (int)(srcBmp.Width / 2 * (0 - sin) + srcBmp.Height / 2 * (1 - cos))
    ;
42.
43.     int insertBeginX = srcBmp.Width / 2 - widthDst / 2;
44.     int insertBeginY = srcBmp.Height / 2 - heightDst / 2;
45.
46.     //插值公式所需参数
47.     double ku = insertBeginX * cos - insertBeginY * sin + dx;
48.     double kv = insertBeginX * sin + insertBeginY * cos + dy;
49.     double cu1 = cos, cu2 = sin;
50.     double cv1 = sin, cv2 = cos;
51.
52.     double fu, fv, a, b, F1, F2;
53.     int Iu, Iv;
54.     srcBmpData = srcBmp.LockBits(new Rectangle(0, 0, srcBmp.Width, srcBmp.Height
    ), ImageLockMode.ReadOnly, PixelFormat.Format24bppRgb);
55.     dstBmpData = dstBmp.LockBits(new Rectangle(0, 0, dstBmp.Width, dstBmp.Height
    ), ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);
56.
57.     unsafe
58.     {
59.         byte* ptrSrc = (byte*)srcBmpData.Scan0;
60.         byte* ptrDst = (byte*)dstBmpData.Scan0;
61.         for (int i = 0; i < heightDst; i++)
62.         {
63.             for (int j = 0; j < widthDst; j++)
64.             {
65.                 fu = j * cu1 - i * cu2 + ku;
66.                 fv = j * cv1 + i * cv2 + kv;
67.                 if ((fv < 1) || (fv > srcBmp.Height - 1) || (fu < 1) || (fu > sr
    cBmp.Width - 1))
68.                 {
69.
70.                     ptrDst[i * dstBmpData.Stride + j * 3] = 150;
71.                     ptrDst[i * dstBmpData.Stride + j * 3 + 1] = 150;
72.                     ptrDst[i * dstBmpData.Stride + j * 3 + 2] = 150;
73.                 }
74.                 else
75.                 {//双线性插值
76.                     Iu = (int)fu;
77.                     Iv = (int)fv;
78.                     a = fu - Iu;
79.                     b = fv - Iv;
```

```
                    for (int k = 0; k < 3; k++)
                    {
                        F1 = (1 - b) * *(ptrSrc + Iv * srcBmpData.Stride + Iu *
    3 + k) + b * *(ptrSrc + (Iv + 1) * srcBmpData.Stride + Iu * 3 + k);
                        F2 = (1 - b) * *(ptrSrc + Iv * srcBmpData.Stride + (Iu +
     1) * 3 + k) + b * *(ptrSrc + (Iv + 1) * srcBmpData.Stride + (Iu + 1) * 3 + k);

                        *(ptrDst + i * dstBmpData.Stride + j * 3 + k) = (byte)((
    1 - a) * F1 + a * F2);
                    }
                }
            }
        }
    }
    srcBmp.UnlockBits(srcBmpData);
    dstBmp.UnlockBits(dstBmpData);
    return true;
}
```

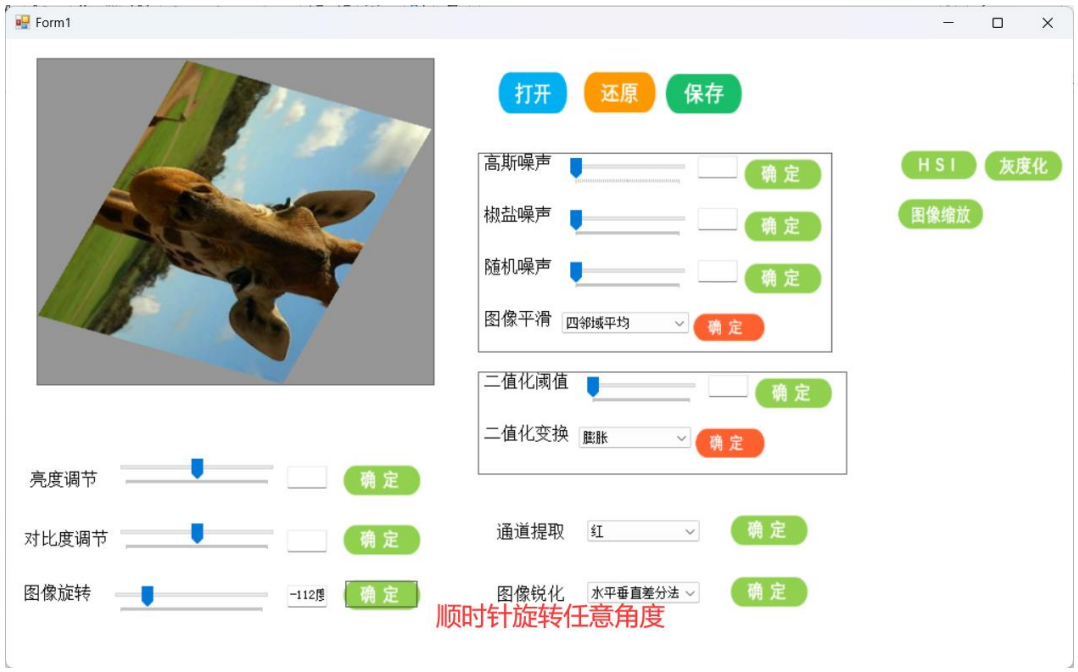
➢ **演示效果：**

原始图片：

旋转之后的图像：


Form1
打开
还原
保存
高斯噪声
椒盐噪声
随机噪声
图像平滑
四邻域平均
确 定
HSI
灰度化
图像缩放
二值化阈值
二值化变换
膨胀
亮度调节
对比度调节
图像旋转
90度
通道提取
红
图像锐化
水平垂直差分法
逆时针旋转90°


Form1
打开
还原
保存
高斯噪声
椒盐噪声
随机噪声
图像平滑
四邻域平均
确 定
HSI
灰度化
图像缩放
二值化阈值
二值化变换
膨胀
亮度调节
对比度调节
图像旋转
-90度
通道提取
红
图像锐化
水平垂直差分法
顺时针旋转90°



图像缩小任意比例

➢ 算法描述：

（1）通过点击按钮实现图片缩小(抛弃周围像素)：

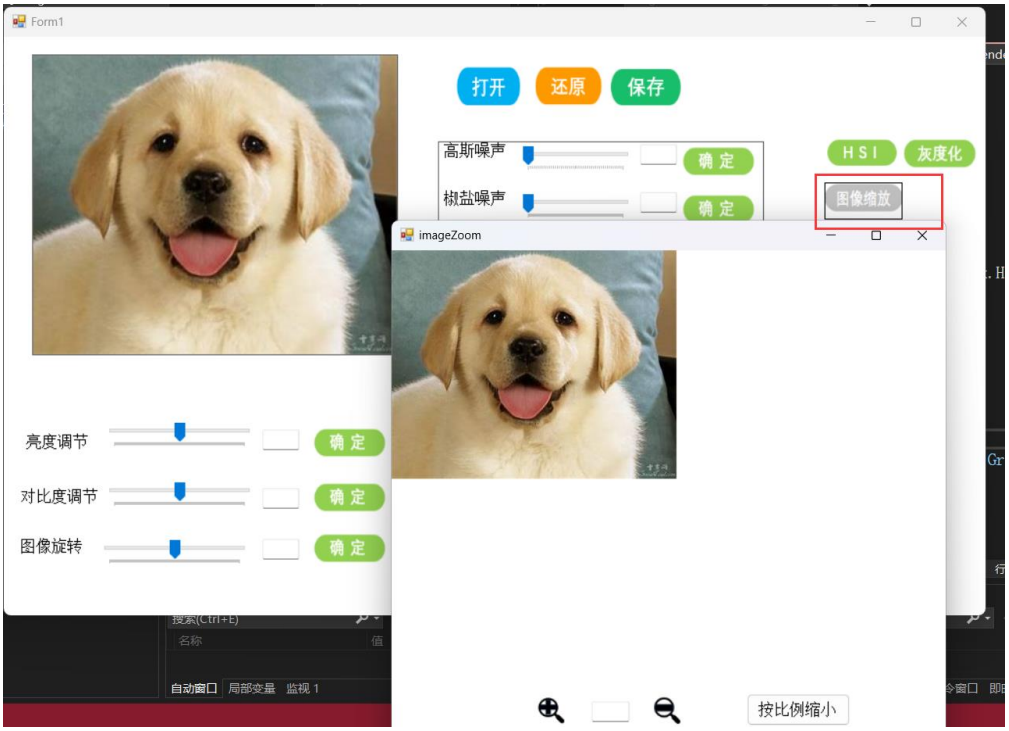
这种方法是通过将原始图像按照一定的比例减小尺寸，并且在缩小过程中抛弃了部分像素点，只有部分像素点被保留下来，而其他像素点被抛

弃。这导致丢失了图像的细节信息，从而使图像看起来更加模糊。并未使用插值算法进行重采样，因此该种方法会导致图像在缩小过程中有**明显的模糊现象：**

```
1.  //按比例缩小按钮，缩小过程中会抛弃像素点
2.  private void button1_Click(object sender, EventArgs e)
3.  {
4.      double k = 1.05; //定义图像缩小的倍数
5.      count++;
6.      Bitmap bt1 = new Bitmap(pictureBox.Image);
7.      Bitmap bt2 = new Bitmap((int)(pictureBox.Width / k), (int)(pictureBox.Height
     / k));
8.      int Red, Green, Blue;
9.      for (int i = 0; i < bt1.Width - 1; i++)
10.     {
11.         for (int j = 0; j < bt1.Height - 1; j++)
12.         {
13.             Red = bt1.GetPixel(i, j).R;
14.             Green = bt1.GetPixel(i, j).G;
15.             Blue = bt1.GetPixel(i, j).B;
16.             bt2.SetPixel((int)(i / k), (int)(j / k), Color.FromArgb(Red, Green,
    Blue));
17.         }
18.         pictureBox.Refresh();//另外建立了一个图片框用于显示处理后的图像
19.         pictureBox.Image = bt2;
20.     }
21.
22.     int y = count * 5;
23.     textBox_showZoomProportion.Text = (100 - y).ToString()+"%";
24. }
```

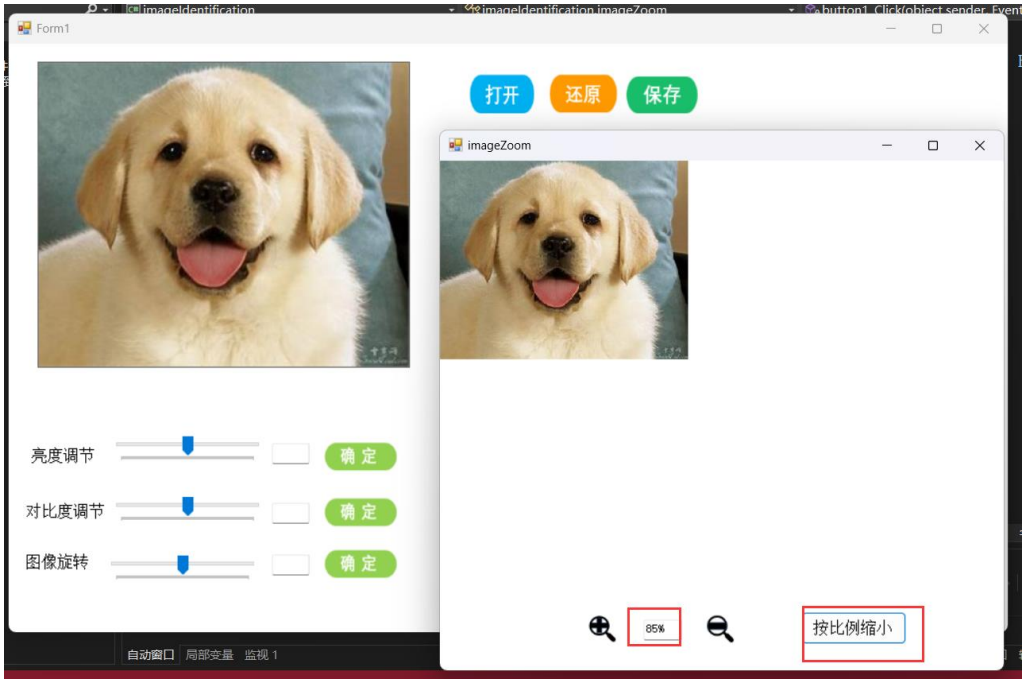
- **演示效果：**

原始图像：

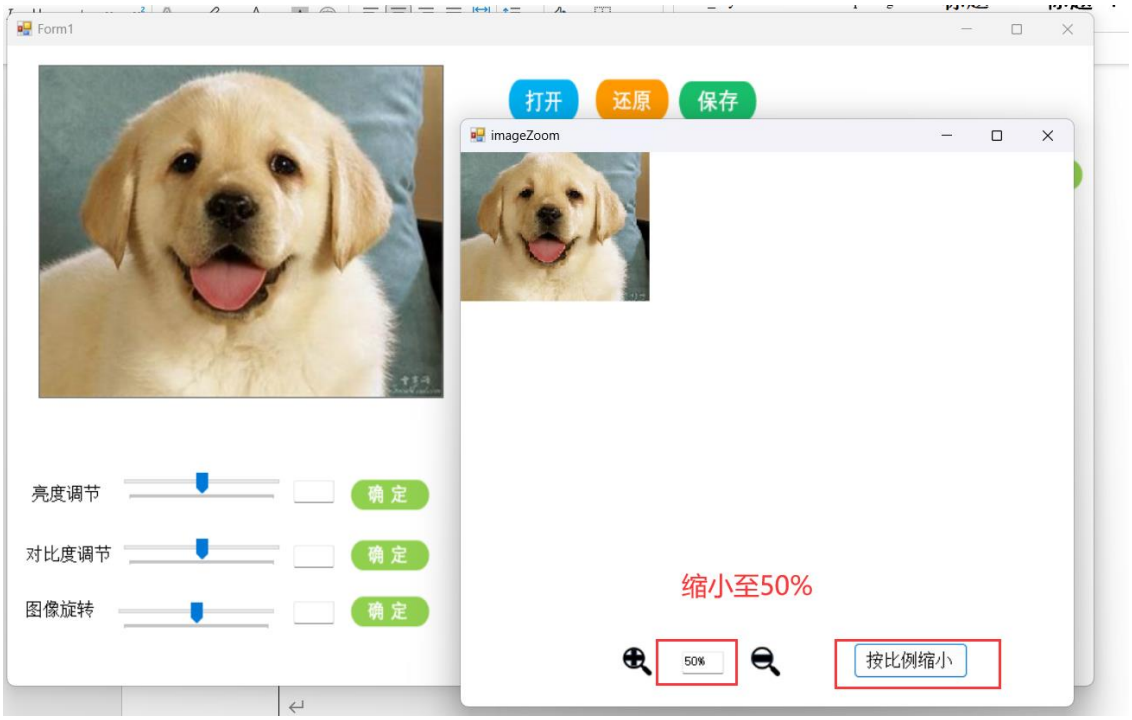


点击**按比例缩小**按钮：

缩小至原图的 85%之后，可以看到**狗的眼睛**有明显的模糊现象：



缩小至原图的 50%及之后，可以看到图像已经发生了明显的形变，周围有锯齿状的模糊现象：



（2）通过点击按钮实现图片缩小(使用插值)：

- **算法描述：**

这种图像缩小使用了 ShrinkImage 方法，创建一个新的位图对象 resizedBitmap，大小为原图像的 95%，作为缩小后的图像容器，使

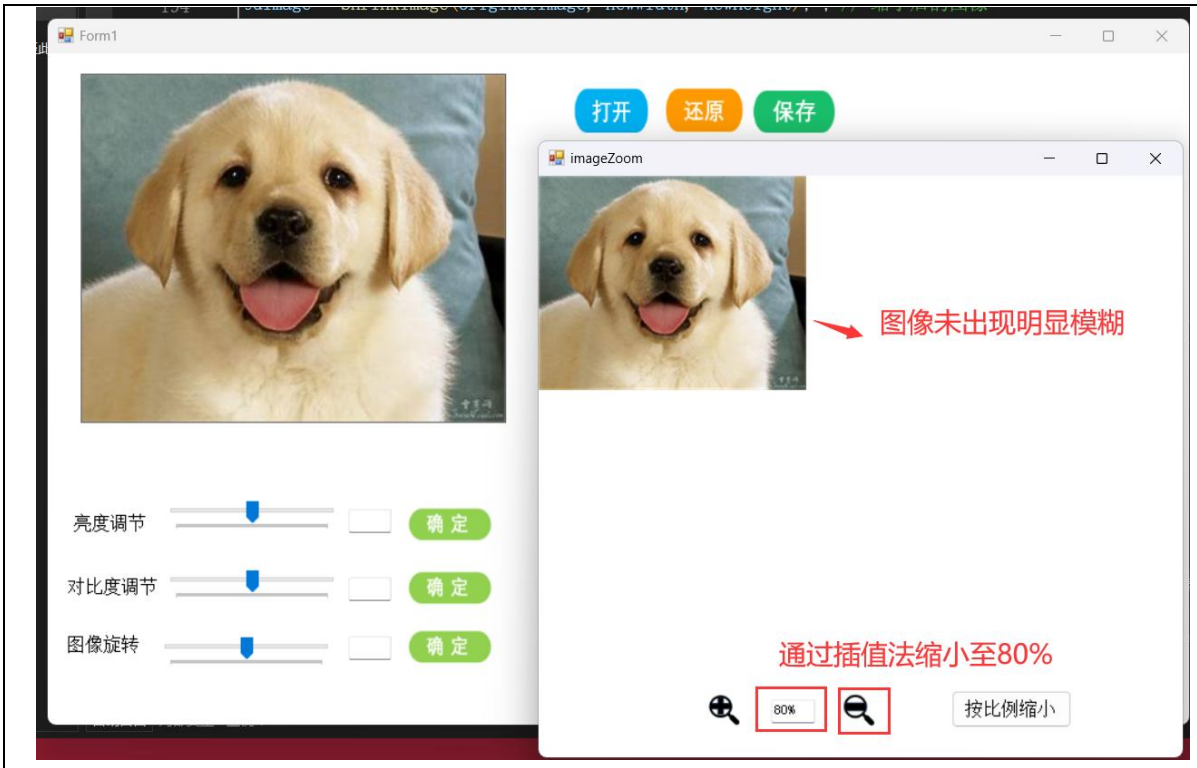
用 Graphics.FromImage 方法创建一个 Graphics 对象（graphics），用于在新的位图上进行绘制操作，并设置插值模式为**高质量、两次立方插值**（HighQualityBicubic）。这是一种高质量的插值方法，可以在缩小图像时保持较好的细节和平滑度，虽然对比抛弃像素的方法效果较好，但仍会导致一些细节丢失和图像模糊。

```
//缩小按钮????????????????????????????????????????????
private void btn_reduce_Click(object sender, EventArgs e)
{
    count++;
    Image originalImage = pictureBox.Image; // 原始图像

    int newWidth = (int)(pictureBox.Width * 0.95);
    int newHeight = (int)(pictureBox.Height * 0.95);

    Image resizedImage = ShrinkImage(originalImage, newWidth, newHeight); ; // 缩小后的图像
    pictureBox.Image = resizedImage;

    int y = count * 5;
    textBox_showZoomProportion.Text = (100 - y).ToString() + "%";


}
private Image ShrinkImage(Image originalImage, int newWidth, int newHeight)
{
    Bitmap resizedBitmap = new Bitmap(newWidth, newHeight);
    using (Graphics graphics = Graphics.FromImage(resizedBitmap))
    {
        graphics.InterpolationMode = System.Drawing.Drawing2D.InterpolationMode.HighQualityBicubic;
        graphics.DrawImage(originalImage, 0, 0, newWidth, newHeight);
    }
    return resizedBitmap;
}
```

➢ **实现效果：**

对比抛弃部分像素的方法：

在图像缩小至原图像的 10%时，才出现了一些模糊现象。


Form1
打开
还原
保存
imageZoom
图像未出现明显模糊
亮度调节
确 定
对比度调节
确 定
图像旋转
确 定
通过插值法缩小至80%
80%
按比例缩小


Form1
打开
还原
保存
imageZoom
缩小至50%仍保持较高质量
图像周围未出现锯齿状模糊
亮度调节
确 定
对比度调节
确 定
图像旋转
确 定
50%
按比例缩小

图像放大任意比例：

（1）最邻近插值

➢ **算法描述：**

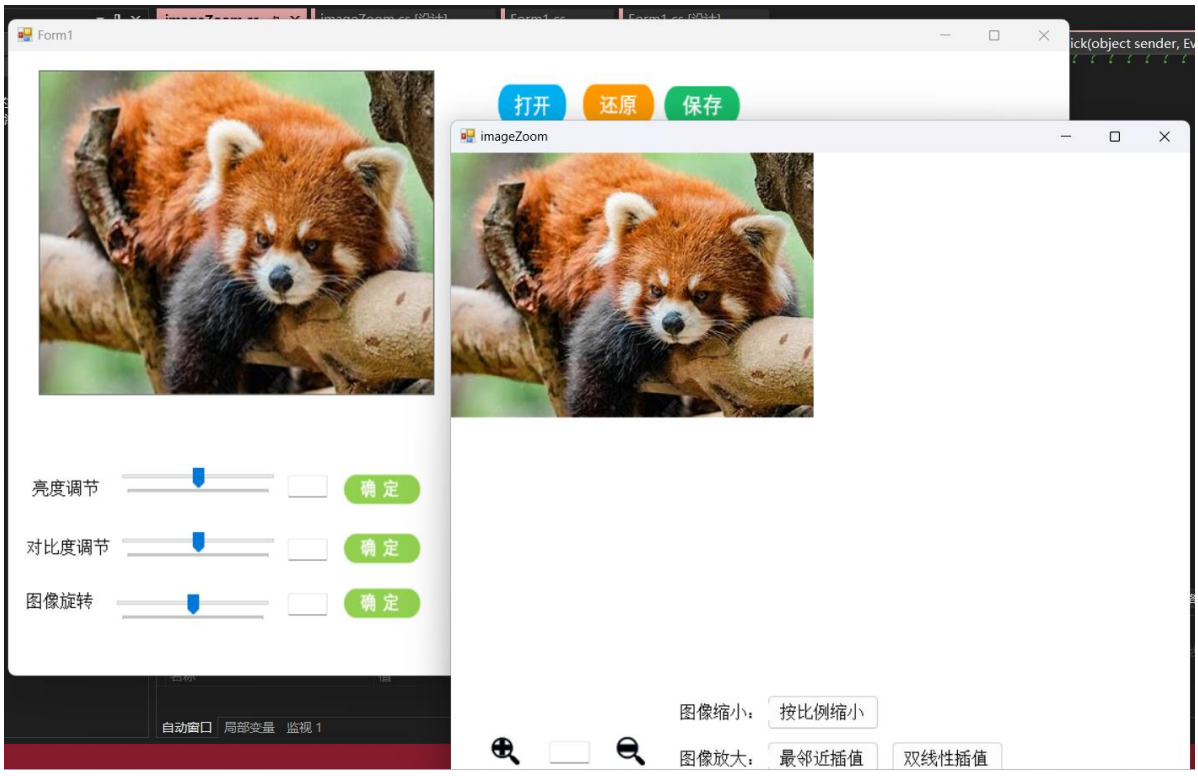
使用**最近邻插值**的思想对新图像进行像素填充。对于放大后的每个像素位置，根据其在原始图像中对应的位置，通过**整数除法**计算出最近的原始图像像素位置。然后获取该位置的像素值，并将其设置为放大后图像的像素值。这是一种简单的插值方法，可能会导致图像的**锯齿状边缘**和**失真**

```
//最邻近插值? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
private void btn_nearestInterpolation_Click(object sender, EventArgs e)
{
    // 计算放大后的图像大小
    int newWidth = (int)(pictureBox.Width * 1.05);
    int newHeight = (int)(pictureBox.Height * 1.05);

    // 创建新的 Bitmap 对象
    Bitmap newImage = new Bitmap(newWidth, newHeight);

    // 最近邻插值
    for (int y = 0; y < newHeight; y++)
    {
        for (int x = 0; x < newWidth; x++)
        {
            // 计算原始图像中对应位置
            int sourceX = (int)(x / 1.05);
```

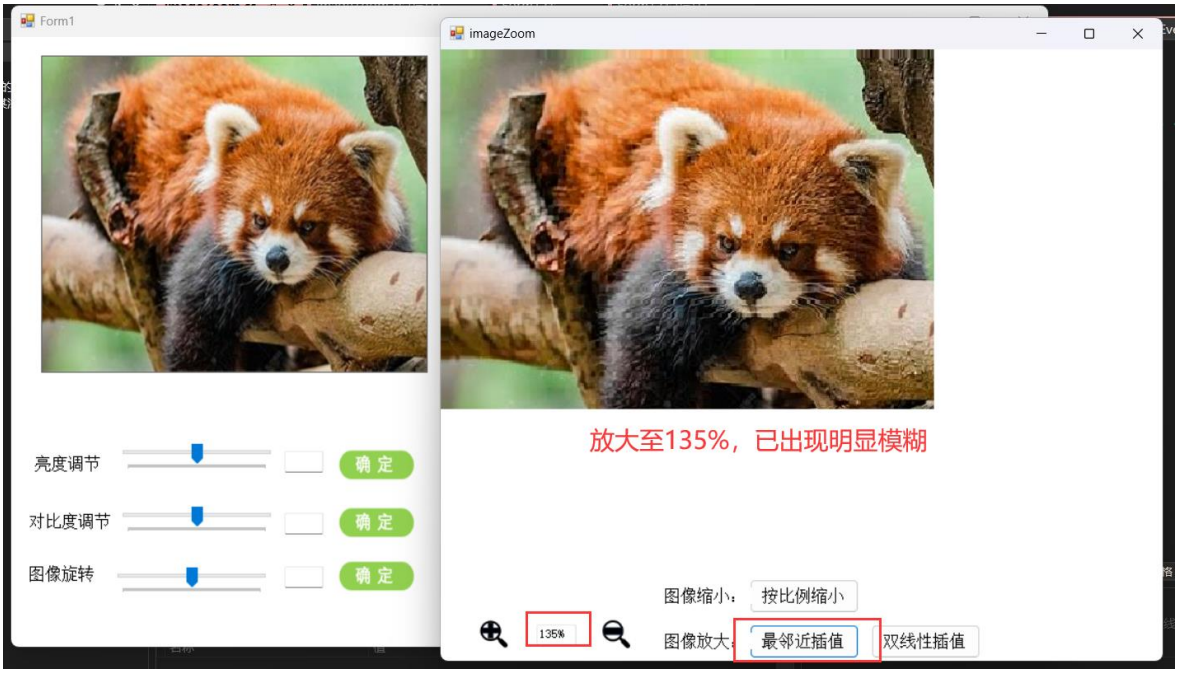
```
            int sourceY = (int)(y / 1.05);

            // 获取最近邻像素值
            Color pixel = ((Bitmap)pictureBox.Image).GetPixel(sourceX, sourceY);

            // 设置放大后图像像素值
            newImage.SetPixel(x, y, pixel);
        }
    }

    count++;
    int a = count * 5;
    textBox_showZoomProportion.Text = (100 + a).ToString() + "%";

    // 更新 pictureBox 的图像和大小
    pictureBox.Image = newImage;
    pictureBox.Width = newWidth;
    pictureBox.Height = newHeight;

}
```

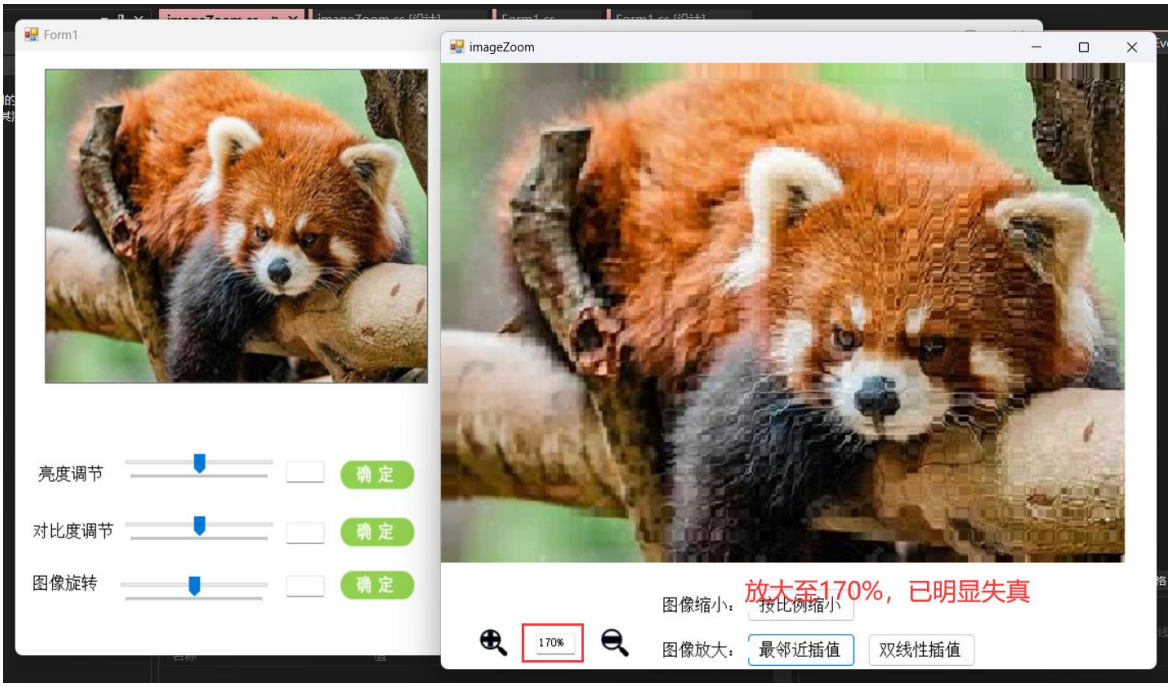
- **实现效果：**

原始图像：



放大图像：





（2）双线性插值

- **算法描述：**

这种图像放大使用了使用**双线性插值**的思想对新图像进行像素填充。对于放大后的每个像素位置，先计算其在原始图像中对应位置的浮点坐标。然后根据该坐标计算四个最近邻像素的坐标，并获取其颜色值。接着，计算出插值权重，即目标像素与其最近邻像素之间的距离和权重。最后，

使用双线性插值公式，根据权重对四个最近邻像素进行插值计算，得到放大后像素的颜色值。

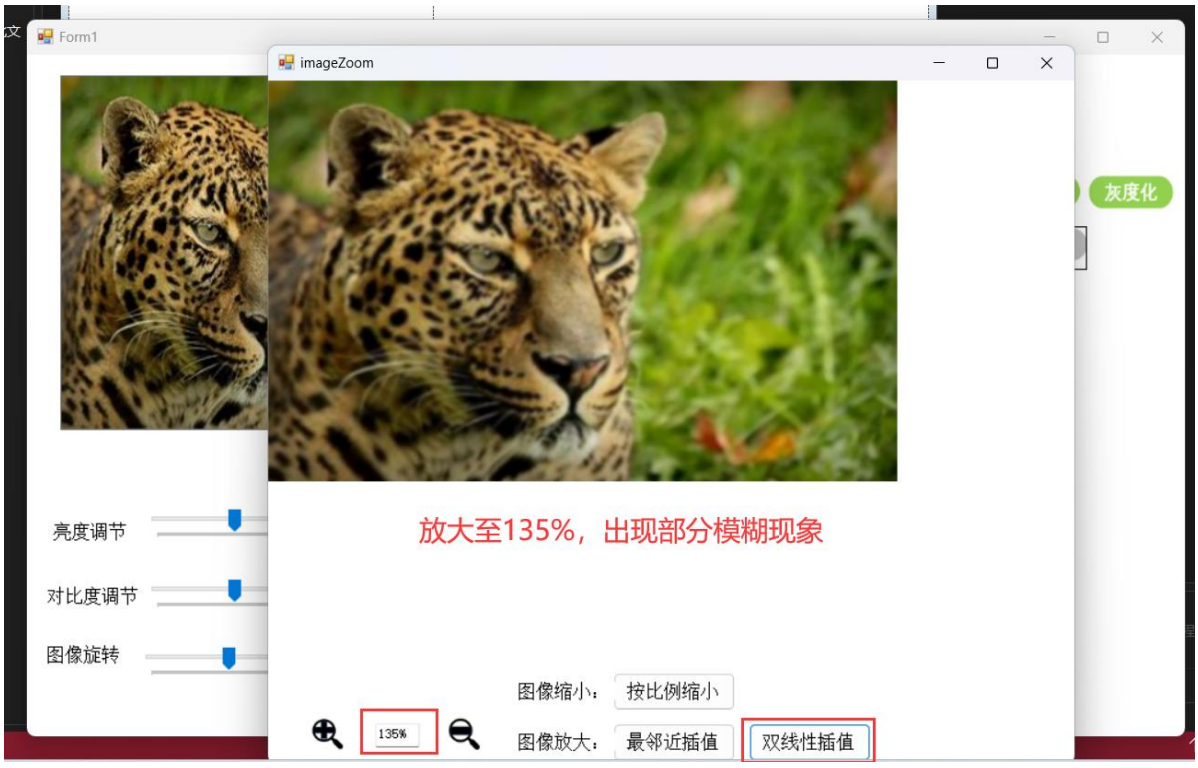
与最邻近插值相比，双线性插值的平滑度较好，虽然再放大过程中也会出现模糊现象，但是锯齿状边缘化程度很小。

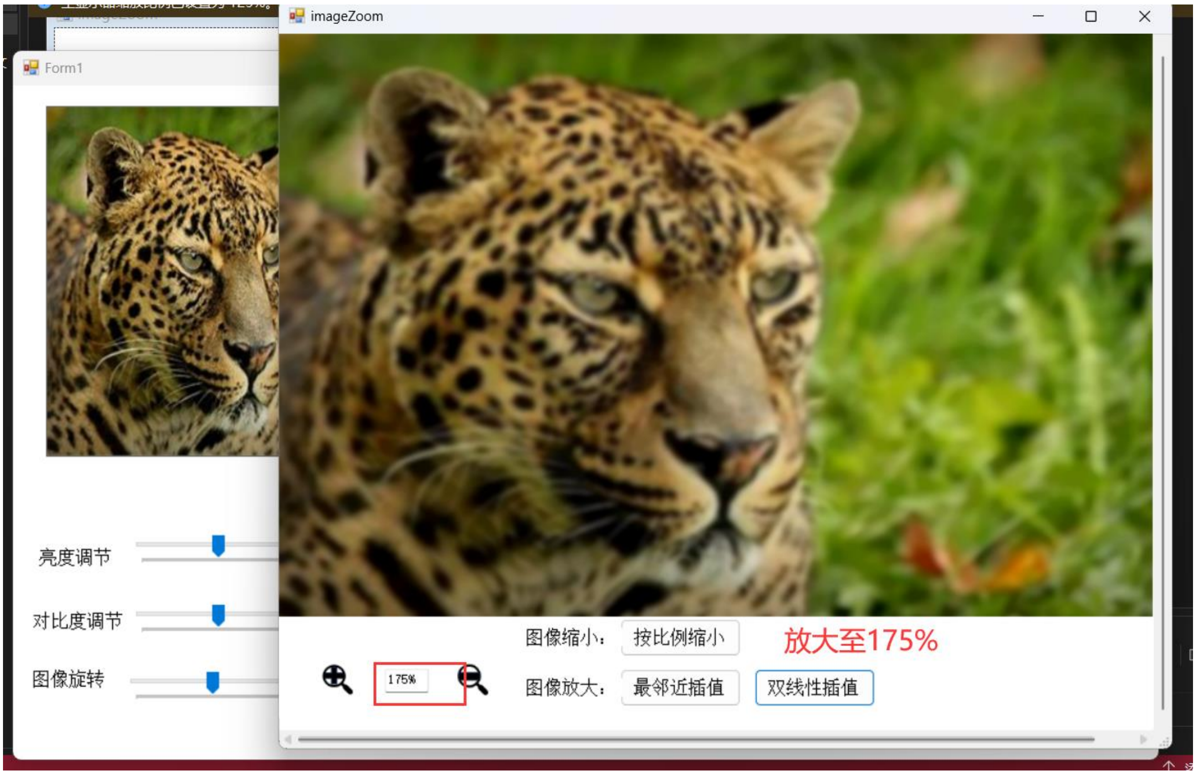
```
//双线性插值? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
private void btn_bilinearityPolation_Click(object sender, EventArgs e)
{
    // 计算放大后的图像大小
    int newWidth = (int)(pictureBox.Width * 1.05);
    int newHeight = (int)(pictureBox.Height * 1.05);

    // 创建新的 Bitmap 对象
    Bitmap newImage = new Bitmap(newWidth, newHeight);

    // 双线性插值
    for (int y = 0; y < newHeight - 1; y++)
    {
        for (int x = 0; x < newWidth - 1; x++)
        {
            // 计算原始图像中对应位置的浮点坐标
            float sourceX = x / 1.05f;
            float sourceY = y / 1.05f;

            // 计算四个最近邻像素的坐标
            int x1 = (int)Math.Floor(sourceX);
            int y1 = (int)Math.Floor(sourceY);
            int x2 = x1 + 1;
            int y2 = y1 + 1;

            // 获取四个最近邻像素的颜色
            Color pixel1 = ((Bitmap)pictureBox.Image).GetPixel(x1, y1);
            Color pixel2 = ((Bitmap)pictureBox.Image).GetPixel(x2, y1);
            Color pixel3 = ((Bitmap)pictureBox.Image).GetPixel(x1, y2);
            Color pixel4 = ((Bitmap)pictureBox.Image).GetPixel(x2, y2);

            // 计算插值权重
            float weightX = sourceX - x1;
            float weightY = sourceY - y1;

            // 根据权重进行双线性插值计算
            Color interpolatedPixel = InterpolateBilinear(pixel1, pixel2, pixel3, pixel4, weightX, weightY);

            // 设置放大后图像的像素值
            newImage.SetPixel(x, y, interpolatedPixel);
        }
    }

    count++;
    int a = count * 5;
    textBox_showZoomProportion.Text = (100 + a).ToString() + "%";

```

```
48.     // 更新 pictureBox 的图像和大小
49.     pictureBox.Image = newImage;
50.     pictureBox.Width = newWidth;
51.     pictureBox.Height = newHeight;
52. }
53. // 双线性插值计算
54. private Color InterpolateBilinear(Color c1, Color c2, Color c3, Color c4, float we
    ightX, float weightY)
55. {
56.     int red = (int)(c1.R * (1 - weightX) * (1 - weightY) +
57.                     c2.R * weightX * (1 - weightY) +
58.                     c3.R * (1 - weightX) * weightY +
59.                     c4.R * weightX * weightY);
60.
61.     int green = (int)(c1.G * (1 - weightX) * (1 - weightY) +
62.                       c2.G * weightX * (1 - weightY) +
63.                       c3.G * (1 - weightX) * weightY +
64.                       c4.G * weightX * weightY);
65.
66.     int blue = (int)(c1.B * (1 - weightX) * (1 - weightY) +
67.                      c2.B * weightX * (1 - weightY) +
68.                      c3.B * (1 - weightX) * weightY +
69.                      c4.B * weightX * weightY);
70.
71.     return Color.FromArgb(red, green, blue);
72. }
```

➢ **实现效果：**

放大过程中也会出现模糊现象，但是双线性插值的**平滑度较好：**





(3)双三次插值

➢ **算法描述:**

对于放大后的每个像素位置，首先计算其在原始图像中对应位置的浮点坐标。然后根据该坐标获取最近邻的 16 个像素的颜色值。根据插值计算的思想，首先计算水平和垂直方向上的权重，利用 CalculateWeights 函数计算横向和纵向的权重。最后根据权重和像素颜色进行插值计算，并返回插值后的像素颜色。

通过双三次插值算法实现了图像的放大功能。在放大过程中，通过对原始图像像素的加权平均，生成了新的放大后的像素值，从而实现图像的平滑放大效果。

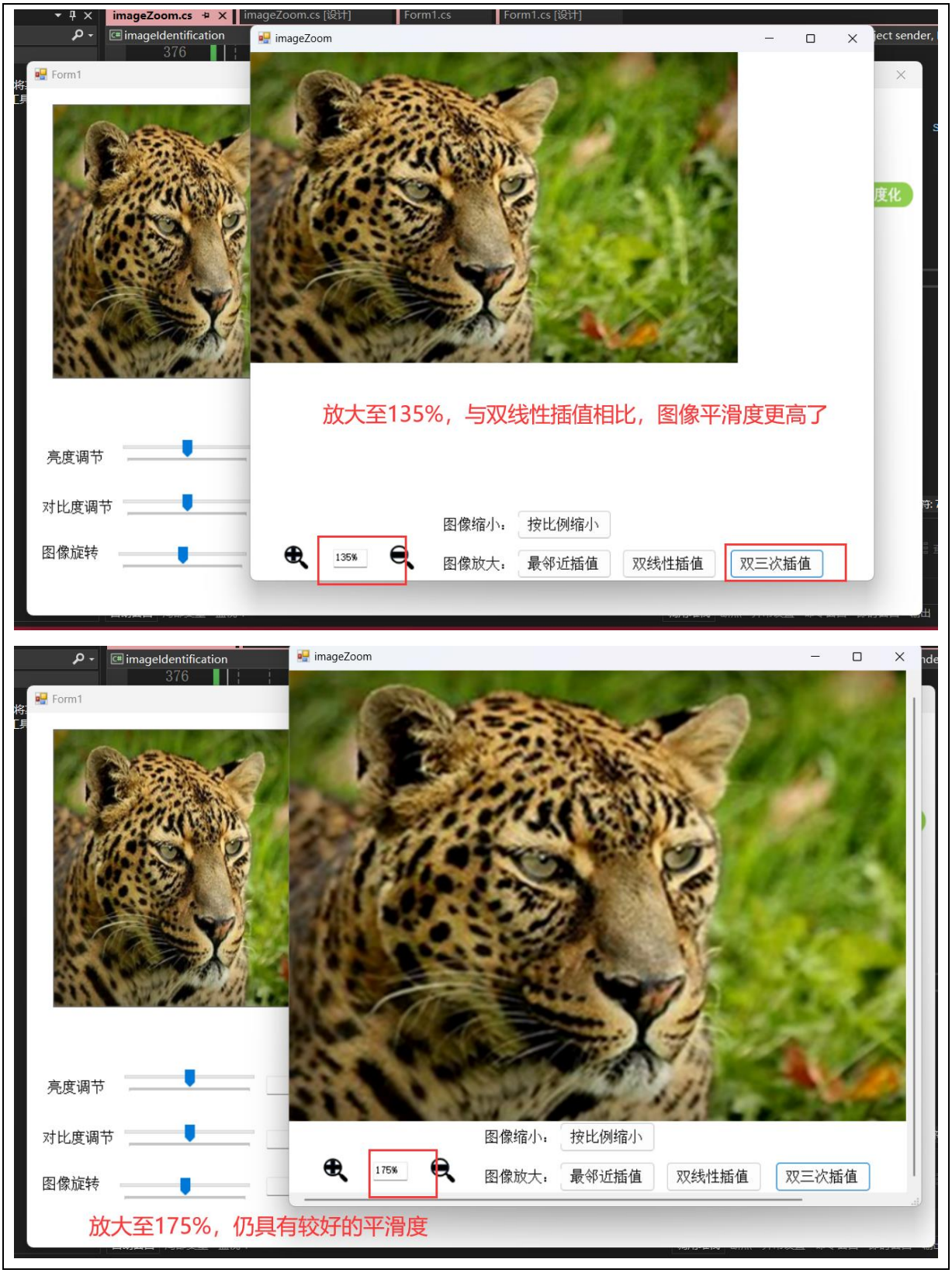
```
//双三次插值? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
private void btn_threePolation_Click(object sender, EventArgs e)
{
    // 计算放大后的图像大小
    int newWidth = (int)(pictureBox.Width * 1.05);
    int newHeight = (int)(pictureBox.Height * 1.05);

    // 创建新的 Bitmap 对象
    Bitmap newImage = new Bitmap(newWidth, newHeight);

    // 双三次插值
```

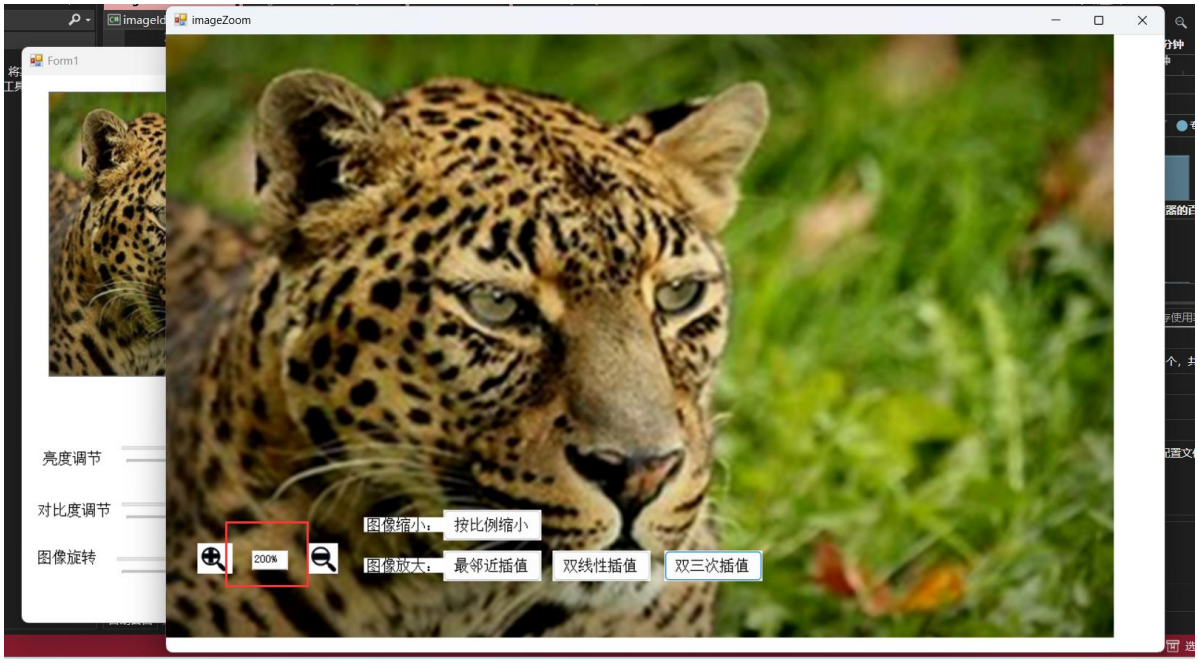
```
    for (int y = 0; y < newHeight; y++)
    {
        for (int x = 0; x < newWidth; x++)
        {
            // 计算原始图像中对应位置的浮点坐标
            float sourceX = x / 1.05f;
            float sourceY = y / 1.05f;

            // 获取双三次插值像素值
            Color interpolatedPixel = InterpolateBicubic((Bitmap)pictureBox.Image,
    sourceX, sourceY);

            // 设置放大后图像的像素值
            newImage.SetPixel(x, y, interpolatedPixel);
        }
    }

    // 更新 pictureBox 的图像和大小
    pictureBox.Image = newImage;
    pictureBox.Width = newWidth;
    pictureBox.Height = newHeight;
}

// 双三次插值计算
private Color InterpolateBicubic(Bitmap image, float x, float y)
{
    int intX = (int)x;
    int intY = (int)y;
    float fractionX = x - intX;
    float fractionY = y - intY;

    Color[,] pixels = new Color[4, 4];

    // 收集 16 个最近邻像素的颜色
    for (int offsetY = -1; offsetY <= 2; offsetY++)
    {
        for (int offsetX = -1; offsetX <= 2; offsetX++)
        {
            int pixelX = Clamp(intX + offsetX, 0, image.Width - 1);
            int pixelY = Clamp(intY + offsetY, 0, image.Height - 1);
            pixels[offsetX + 1, offsetY + 1] = image.GetPixel(pixelX, pixelY);
        }
    }

    // 计算权重
    float[] weightsX = CalculateWeights(fractionX);
    float[] weightsY = CalculateWeights(fractionY);

    // 插值计算
    float red = 0f;
    float green = 0f;
    float blue = 0f;

    for (int offsetY = 0; offsetY < 4; offsetY++)
    {
        for (int offsetX = 0; offsetX < 4; offsetX++)
```

```
        {
            float weight = weightsX[offsetX] * weightsY[offsetY];
            red += pixels[offsetX, offsetY].R * weight;
            green += pixels[offsetX, offsetY].G * weight;
            blue += pixels[offsetX, offsetY].B * weight;
        }
    }

    return Color.FromArgb(Clamp((int)red, 0, 255), Clamp((int)green, 0, 255), Clamp((int)blue, 0, 255));
}

// 计算权重
private float[] CalculateWeights(float t)
{
    float[] weights = new float[4];

    float t2 = t * t;
    float t3 = t2 * t;

    weights[0] = -0.5f * t3 + t2 - 0.5f * t;
    weights[1] = 1.5f * t3 - 2.5f * t2 + 1.0f;
    weights[2] = -1.5f * t3 + 2.0f * t2 + 0.5f * t;
    weights[3] = 0.5f * t3 - 0.5f * t2;

    return weights;
}

// 限制值在指定范围内
private int Clamp(int value, int minValue, int maxValue)
{
    return Math.Max(minValue, Math.Min(value, maxValue));
}
```

- **实现效果：**

相比前两种放大方式，三次插值的效果最好，在放大到 200%的时候，模糊效果才开始明显起来，但是整体的平滑度依然很好，锯齿状现象几乎没有。


imageZoom.cs
imageZoom.cs [设计]
Form1.cs
Form1.cs [设计]
imageIdentification
376
Form1
imageZoom
放大至135%，与双线性插值相比，图像平滑度更高了
亮度调节
对比度调节
图像旋转
135%
图像缩小：
按比例缩小
图像放大：
最邻近插值
双线性插值
双三次插值
imageIdentification
376
Form1
imageZoom
亮度调节
对比度调节
图像旋转
175%
图像缩小：
按比例缩小
图像放大：
最邻近插值
双线性插值
双三次插值
放大至175%，仍具有较好的平滑度



鼠标滚轮实现图片缩放

➢ 算法描述：

该事件处理程序首先检查是否按下了 Ctrl 键（Control.ModifierKeys == Keys.Control），以确保只有在同时按下 Ctrl 键时才进行缩放操作。

如果滚轮向上滚动，即进行放大操作，算法原理如下：

1.根据设定的缩放因子 fZoomFactor，计算缩放后的图像大小 iNewWidth 和 iNewHeight。

2.创建一个新的 Bitmap 对象 BitNewImg，大小为缩放后的尺寸。

3.使用 Graphics 对象 graph 绘制新的图像，通过设置 InterpolationMode 为 InterpolationMode.Bilinear 来使用双线性插值算法。

4.将原始图像绘制到新的图像上，通过指定源矩形和目标矩形的方式进行缩放。

5.计算缩放后的中心点距离 pictureBox 左上角的距离，即 iNewCentorX 和 iNewCentorY。

6.将新的图像赋值给 pictureBox 的 Image 属性，并更新 pictureBox 的宽度和高度。

7.调整滚动条的位置，使缩放后的中心点处于可见区域。

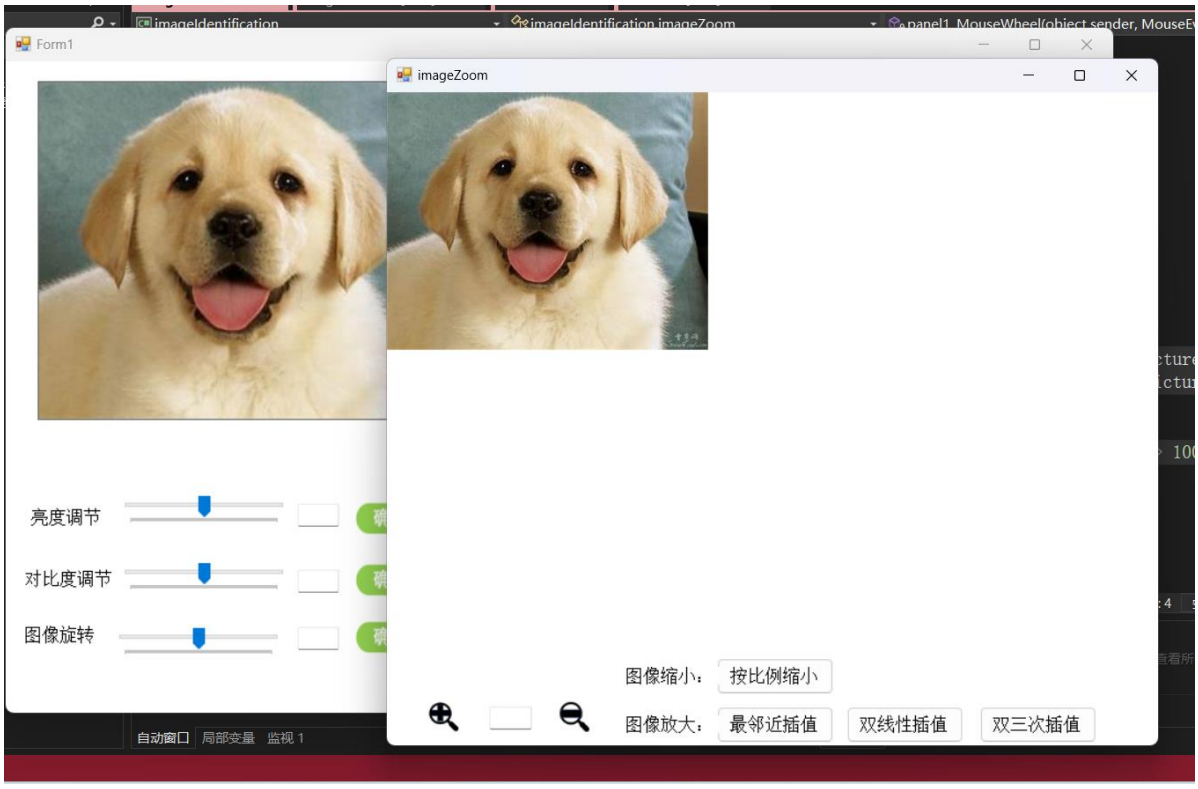
如果滚轮向下滚动，即进行缩小操作，算法原理与放大操作类似。

```
1.  //滚轮滚动实现图像的缩放
2.  private void panel1_MouseWheel(object sender, MouseEventArgs e)
3.  {
4.      if (Control.ModifierKeys == Keys.Control)
5.      {
6.          float fZoomFactor = 1.2f;       // 缩放因子
7.          int iOriginCentorX = (this.panel1.Width / 2 - this.pictureBox.Left);
    //缩放前
8.          int iOriginCentorY = (this.panel1.Height / 2 - this.pictureBox.Top);
9.
10.         //防止无限制的缩放
11.         if ((this.pictureBox.Width < 200 && e.Delta < 0) || (this.pictureBox.Width
     > 10000 && e.Delta > 0))
12.             return;
13.
14.         //鼠标滚轮向上滚动，进行放大操作
15.         if (e.Delta > 0)
16.         {
17.             //缩放后的 picture box 的大小。
18.             int iNewWidth = (int)(fZoomFactor * this.pictureBox.Width);
19.             int iNewHeight = (int)(fZoomFactor * this.pictureBox.Height);
20.
21.             //定义新的 Bitmap;
22.             Bitmap BitNewImg = new Bitmap(iNewWidth, iNewHeight);
23.             //定义对 BitNewImg 的绘制方法;
24.             Graphics graph = Graphics.FromImage(BitNewImg);
25.             graph.InterpolationMode = System.Drawing.Drawing2D.InterpolationMode.B
    ilinear;
26.             //用原始图像绘制新图像;
27.             graph.DrawImage((Bitmap)this.pictureBox.Image, new Rectangle(0, 0, iNe
    wWidth, iNewHeight),
28.                 new Rectangle(0, 0, this.pictureBox.Width, this.pictureBox.Height)
    , GraphicsUnit.Pixel);
29.             //缩放后的中心点距离 picture box 左上角的距离。
30.             int iNewCentorX = (int)(iOriginCentorX * fZoomFactor);
31.             int iNewCentorY = (int)(iOriginCentorY * fZoomFactor);
32.
33.             this.pictureBox.Image = BitNewImg;
34.             this.pictureBox.Width = this.pictureBox.Image.Width;
35.             this.pictureBox.Height = this.pictureBox.Image.Height;
36.             //缩放后的滚轮的位置。
37.             this.panel1.AutoScrollPosition = new Point((int)(iNewCentorX - this.pa
    nel1.Width / 2),
38.                 (int)(iNewCentorY - this.panel1.Height / 2 + e.Delta));
39.
40.             graph.Dispose();
41.         }
42.         else
43.         {
44.             int iNewWidth = (int)(this.pictureBox.Width / fZoomFactor);
45.             int iNewHeight = (int)(this.pictureBox.Height / fZoomFactor);
46.
47.             Bitmap BitNewImg = new Bitmap(iNewWidth, iNewHeight);
48.             //定义对 BitNewImg 的绘制方法;
49.             Graphics graph = Graphics.FromImage(BitNewImg);
```

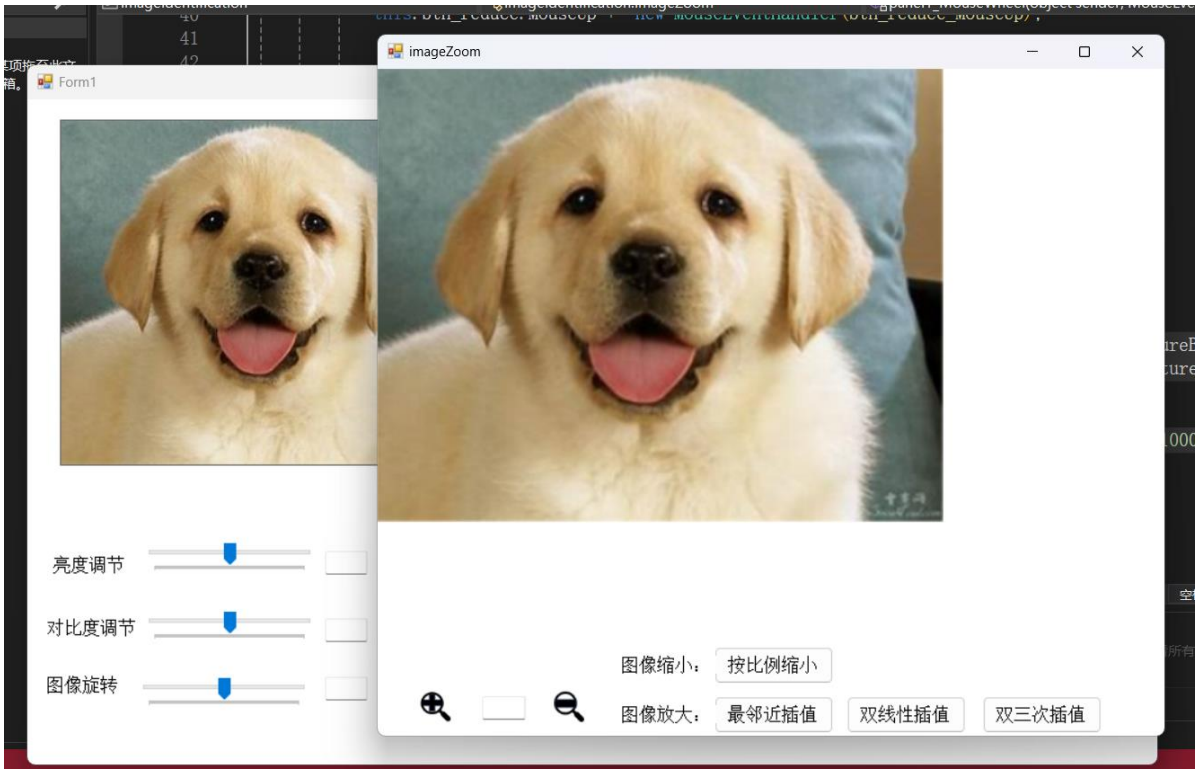
```
            graph.InterpolationMode = System.Drawing.Drawing2D.InterpolationMode.Bilinear;
            //用原始图像绘制新图像；
            graph.DrawImage((Bitmap)this.pictureBox.Image, new Rectangle(0, 0, iNewWidth, iNewHeight),
                new Rectangle(0, 0, this.pictureBox.Width, this.pictureBox.Height), GraphicsUnit.Pixel);

            this.pictureBox.Image = BitNewImg;
            this.pictureBox.Width = this.pictureBox.Image.Width;
            this.pictureBox.Height = this.pictureBox.Image.Height;

            int iNewCentorX = (int)(iOriginCentorX / fZoomFactor);
            int iNewCentorY = (int)(iOriginCentorY / fZoomFactor);

            this.panel1.AutoScrollPosition = new Point((int)(iNewCentorX - this.panel1.Width / 2),
                (int)(iNewCentorY - this.panel1.Height / 2 + e.Delta));
        }
    }
}
```

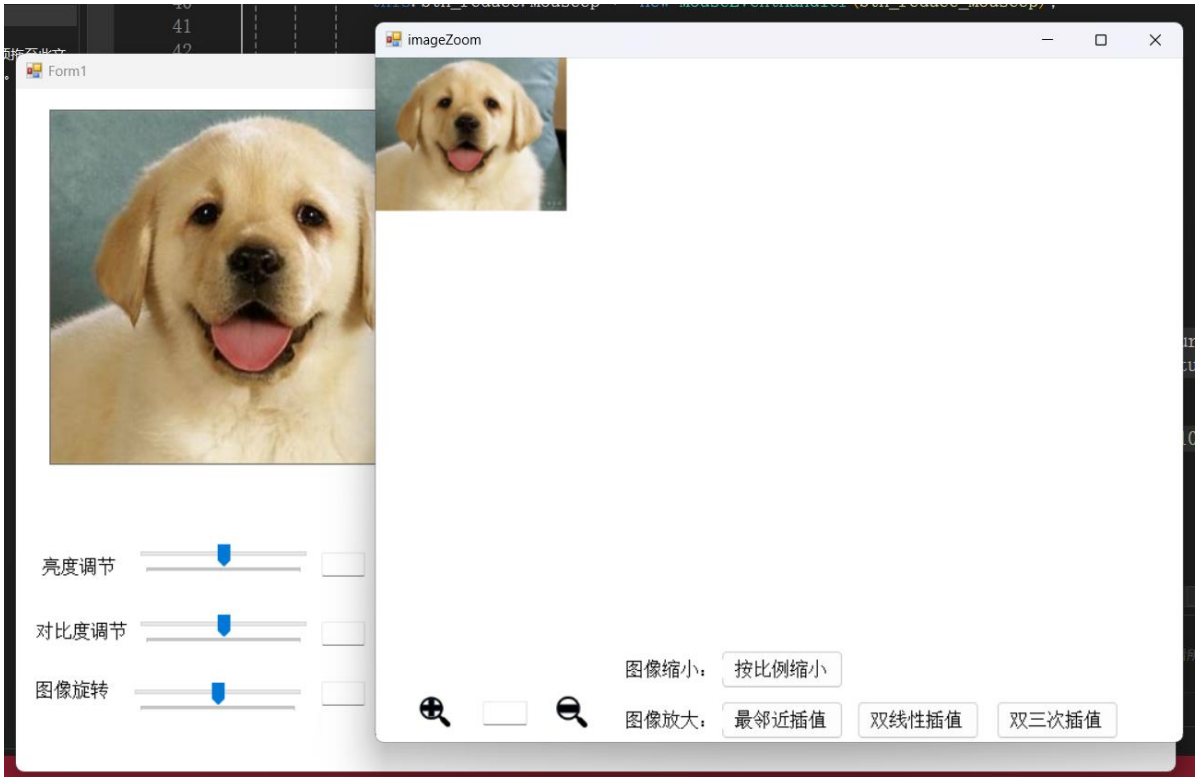
- **实现效果：**

原图像：



按住 ctrl 键，通过鼠标滚轮即可实现缩放效果：





五、实验结果及分析(包括心得体会，本部分为重点，不能抄袭复制)

- **完成情况：**

完成了实验**全部的基本要求**和**全部的扩展要求**，最终的结果基本达到了我的预期

- **实验心得**

图像的缩放和旋转是图像处理中常见的操作，通过对图像进行放大、缩小和旋转，可以实现对图像的变换和调整。在本次实验中，我使用 C# 编程语言实现了图像的缩放和旋转功能，并进行了测试。

在实现图像的缩放功能时，我采用了**鼠标滚轮事件**来触发缩放操作。根据滚轮滚动的方向，我分别进行了图像的放大和缩小操作。通过调整图像的大小和中心点的位置来实现放大效果；

然后也利用点击按钮实现了**放大操作**，使用了**三种算法（最邻近插值，双线性插值和双三次插值）**，对比三者的不同，从而进行平滑的放大处理；**缩小操作**则是通过缩小因子来计算新的图像大小，对比了**抛弃部分像素算法**和**双线性插值算法**的不同。

除了图像的缩放操作，我还进行了拓展，实现了图像的**任意角度旋转功能**。可以在保持图像内容完整的同时改变其方向和角度。但是由于图像旋转算法使用了**双线性插值方法**，对于旋转角度较大的图像，可能会导致一定程度的图像**失真和模糊。**

总结而言，通过完成这个实验，我深入理解了图像的缩放和旋转原理，以及如何在 C#中利用图形库进行图像处理。