目录

[0 引言 3](#_Toc30277)

[1 相关原理、方法及技术等介绍 3](#_Toc28350)

[1.1. 灰度变换 3](#_Toc20921)

[1.2. 灰度直方图和阈值分割 4](#_Toc8069)

[1.2.1. 迭代法 5](#_Toc32281)

[1.2.2. OSTU方法 5](#_Toc17893)

[1.2.3. 波峰中值法（自创） 6](#_Toc20891)

[1.2.4. 自适应阈值分割 6](#_Toc7197)

[1.3. 空域滤波 6](#_Toc28375)

[1.4. 几何变换 7](#_Toc11864)

[1.4.1. 平移 7](#_Toc7547)

[2 算法的实现效果(界面及算法的效果等) 9](#_Toc2314)

[2.1. 灰度的阈值变换 9](#_Toc5540)

[2.1.1. 单一阈值变换 9](#_Toc21365)

[2.1.2. 自适应阈值分割 10](#_Toc15061)

[2.2. 灰度的窗口变换 12](#_Toc32557)

[2.3. 灰度拉伸 13](#_Toc27628)

[2.4. 图像的直方图均衡化 13](#_Toc15563)

[2.5. 滤镜（模板操作） 14](#_Toc30600)

[2.5.1. 反色变换（底片效果） 14](#_Toc10037)

[2.5.2. 平滑/模糊（均值模糊，高斯模糊） 14](#_Toc178)

[2.5.3. 锐化（拉普拉斯） 15](#_Toc25121)

[2.5.4. 边缘检测 15](#_Toc15823)

[2.5.5. 浮雕效果 16](#_Toc3522)

[2.5.6. 改变亮度 16](#_Toc3108)

[2.5.7. 自选滤镜 16](#_Toc16570)

[2.6. 图像的几何变换 17](#_Toc25714)

[2.7. 图像的多种风格化 18](#_Toc714)

[2.8. 应用（三种） 18](#_Toc24947)

[2.8.3. 硬币计数 21](#_Toc21976)

[3 结论 23](#_Toc20286)

[4 参考文献 23](#_Toc2109)

[5 附录(代码部分) 24](#_Toc13025)

[5.1. 迭代法求阈值 24](#_Toc8831)

[5.2. 大津法求阈值 25](#_Toc29618)

[5.3. 波峰中值 26](#_Toc5092)

[5.4. 自适应阈值分割 28](#_Toc23580)

[5.5. 窗口变换 31](#_Toc7291)

[5.6. 灰度拉伸 31](#_Toc5627)

[5.7. 直方图均衡化 32](#_Toc10383)

[5.8. 模板函数 33](#_Toc9930)

[5.9. 图像平移 35](#_Toc12453)

[5.10. 图像镜像 36](#_Toc21027)

[5.11. 图像旋转 37](#_Toc2671)

[5.12. 旧照片效果 40](#_Toc15717)

[5.13. 描边效果（使用opencv库） 42](#_Toc14945)

[5.14. 油画效果 43](#_Toc17797)

[5.15. 区域计数 45](#_Toc4801)

[5.16. 计算物体几何特征 46](#_Toc16457)

[5.17. 边界追踪 49](#_Toc5227)

[5.18. Canny边缘检测 50](#_Toc27780)

5.19.USM锐化.........................................................................................................................55

# **引言**

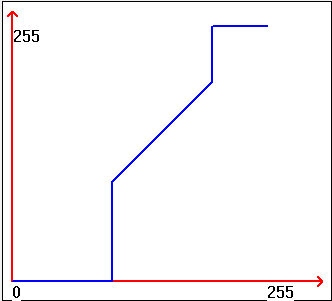
数字图像处理是指用计算机处理图像的技术。《图像处理课程设计》是专业教学中的一个重要的实践性环节。本课程设计的目的是帮助学生巩固和加强已学《数字图像处理》理论知识，并能使学生综合运用所学的理论知识，进行较为复杂的图像处理工作，以此来培养学生理论联系实际、分析和解决技术问题的能力，为以后从事专业技术工作打下必要的基础。综合运用C++语言实现图像处理的程序设计。

# **相关原理、方法及技术等介绍**

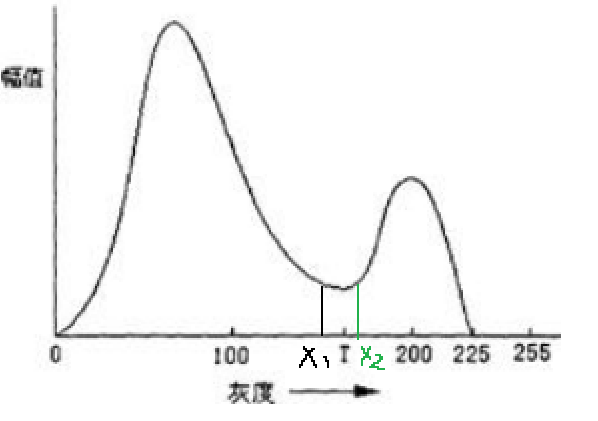
* 1. **灰度变换**

功能“灰度的窗口变换”和“灰度拉伸”都是以灰度变换为基础对图像进行的增强操作。灰度变换是在空间域上对单个像素进行的函数变换，遍历每个像素点，通过函数运算将单位像素上的灰度值映射为新的灰度值从而改变图像达到特定的增强效果。

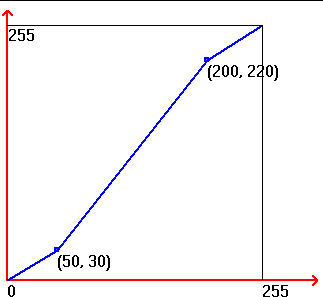
“灰度的窗口变换”函数如下所示。



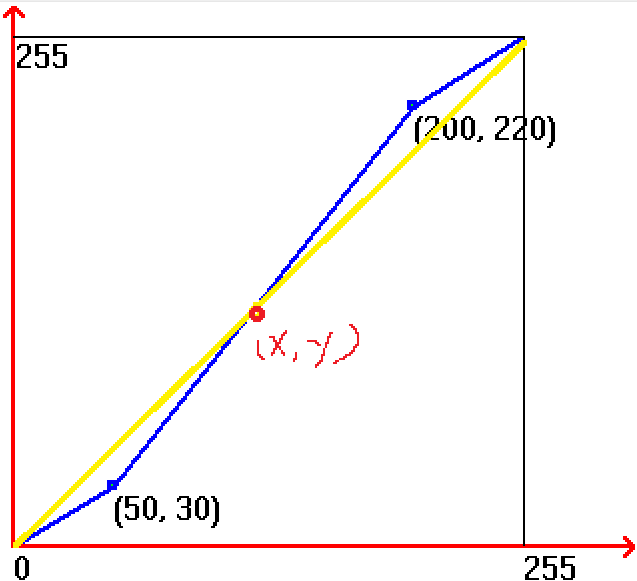
函数分为三段，当灰度值小于X1（左分界点）时，灰度值直接变为0，即小于该亮度的像素全部变为黑色，当灰度值大于X2（右分界点）时，灰度值直接变为255，即大于该亮度的像素全部变为白色，在X1和X2之间的像素点保持不变（变换函数为y=x）。该功能能实现将图像中特定亮度值区分开的效果。如果图像中灰度分布（见1.2灰度直方图）如下，将X1和X2的值设定为T附近的值（波谷区）可以大致实现图像分割。



“灰度拉伸”的函数如下所示。



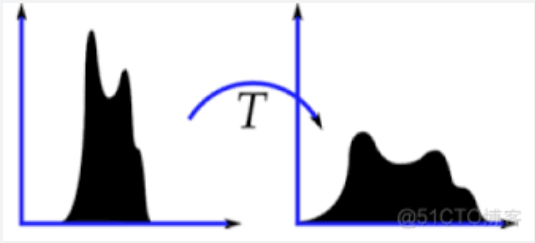
函数分为三段，中间的两个点坐标可以自由变化（左边的x值要小于右边的x值），灰度拉伸的操作较为自由，可实现多种函数变换，应用较为广泛。此外将y=x放进图中比较可看出相较于原灰度分布的变化。如下图中，灰度值<x的像素都相较原图变暗，>x的像素都较原图变亮。



* 1. **灰度直方图和阈值分割**

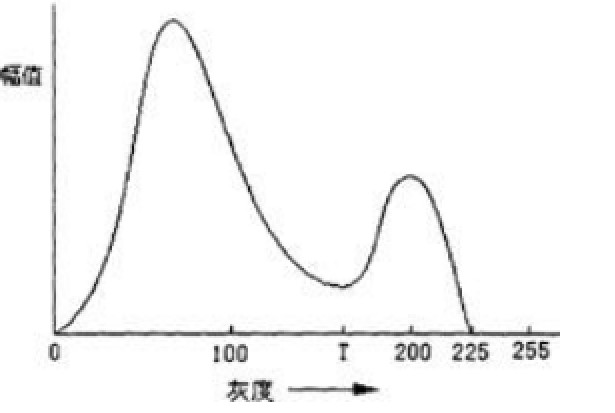
一幅图像由不同灰度值的像素组成，图像中灰度的分布情况是该图像的一个重要特征。图像的灰度直方图就描述了图像中灰度分布情况，能够很直观的展示出图像中各个灰度级所占的多少。图像的灰度直方图是灰度级的函数，描述的是图像中具有该灰度级的像素的个数：其中，横坐标是灰度级，纵坐标是该灰度级出现的频率。

直方图均衡化(Histogram Equalization)是一种增强图像对比度(Image Contrast)的方法，其主要思想是将一副图像的直方图分布变成近似均匀分布，从而增强图像的对比度。



阈值分割法的基本思想是基于图像的灰度特征来计算一个或多个灰度阈值，并将图像中每个像素的灰度值与阈值作比较，最后将像素根据比较结果分到合适的类别中。因此，该方法最为关键的一步就是按照某个准则函数来求解最佳灰度阈值。

利用灰度直方图（灰度分布）的特性，我们能较为方便地进行图像分割。阈值分割适用于目标和背景占据不同灰度级范围的图，例如下图中，前景和背景主要集中在两个波峰，在波谷上确定一个点T作为阈值可以较好地分割出前景和背景。



阈值分割的效果基于阈值的选择，课设中提供了三种方法来计算单一阈值（迭代法、OSTU法，波峰中值法），一种方法来计算多阈值（自适应阈值分割）。

* + 1. 迭代法

当目标和背景像素的灰度分布非常不同时，可对整个图像使用单个（全局）阈值。在大多数应用中，图像之间通常存在足够的变化，即使全局阈值是一种合适的方法，也需要有能对每幅图像估计阈值的算法。下面的迭代算法适用于这一目的：

（1）设置初始阈值 T，通常可以设为图像的平均灰度；

（2）用灰度阈值 T 分割图像：灰度值等于 T 的所有像素集合 G1 和 大于等于 T 的所有像素集合 G2；

（3）分别计算 G1、G2 的平均灰度值 m1、m2；

（4）求出新的灰度阈值 T=(m1+m2)/2；

（5）重复步骤（2）～（4），直到阈值变化小于设定值。

* + 1. OSTU方法

阈值处理本质上是对像素进行分类的统计决策问题，其目的是在把像素分配给两组或者多组（也称分类）的过程中，使引入的平均误差最小。

OTSU 方法又称大津算法，使用最大化类间方差（intra-class variance）作为评价准则，基于对图像直方图的计算，可以给出类间最优分离的最优阈值。

任取一个灰度值 T，可以将图像分割为两类C1和C2，C1和C2的像素数的占比分别为 P1、P2，C1和C2的灰度值均值分别为 m1、m2，整个图像的平均灰度为 m，定义类间方差为：



* + 1. 波峰中值法（自创）

基于有明显两个波峰的灰度直方图，阈值T在波谷是能取得较好的分割效果，波峰的下标获取采用概率分布的算法，默认图中前景和背景分布均匀且集中在两个区域，获取灰度概率累加函数，找出下标index使得在index前的概率密度累计为0.5，将灰度分布函数在index处分开为两部分，在前半部分和后半部分分别寻找最大值得到两个波峰下标，再用波峰中值估计出波谷下标作为阈值T。

* + 1. 自适应阈值分割

在前面的部分我们使用是全局阈值，整幅图像采用同一个数作为阈值。当时这种方法并不适应与所有情况，尤其是当同一幅图像上的不同部分的具有不同亮度时。这种情况下我们需要采用自适应阈值。此时的阈值是根据图像上的每一个小区域计算与其对应的阈值。因此在同一幅图像上的不同区域采用的是不同的阈值，从而使我们能在亮度不同的情况下得到更好的结果。

自适应阈值化计算大概过程是为每一个象素点单独计算的阈值，即每个像素点的阈值都是不同的，就是将该像素点周围B\*B区域内的像素加权平均，然后减去一个常数C，从而得到该点的阈值。

注：由于自适应阈值分割不太好处理边缘的像素点，课设中边缘数据处理采用的是局部阈值获取的方法，即把图像分为n\*m个区域，对每一个区域分别求阈值。

* 1. **空域滤波**

空域滤波是指利用像素及像素邻域组成的空间进行图像增强的方法。模板（也称样板或窗）是实现空域滤波的基本工具。模板运算的基本思路是将赋予某个像素的值作为它本身灰度值和其相邻像素灰度值的函数。模板可看作一幅尺寸为n×n（n一般为奇数，这样有个中心像素）的小图像（远小于常见图像尺寸）。最基本的尺寸为3×3，更大尺寸的模板如5×5，7×7等也常得到使用。模板操作的基本步骤如下：

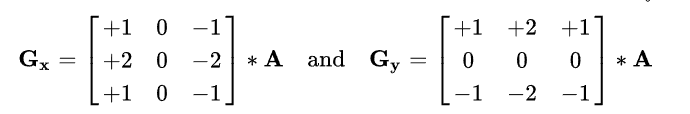
（1）将模板在图中漫游，并将模板中心与图中某个像素位置重合。

（2）将模板上的各个系数与模板下各对应像素的灰度值相乘。

（3）将所有乘积相加（为保持灰度范围，常将结果再除以模板的系数个数）。

（4）将上述运算结果（模板的输出响应）赋给图中对应模板中心位置的像素。

常见模板有均值平滑模板、高斯滤波模板、laplacian算子、sobel算子、prewitt算子、浮雕模板等。例如Sobel算子（如下图）就是对图像中高频部分做出检测并增强，从而达到边缘检测的效果。



此外，模板的函数还能实现灰度变换功能，函数基本思想是对像素和邻域的像素加权求和，再乘以特定系数，加上特定常数得到结果，利用这个特性我们也可以这样设置模板来进行反色操作（如下左图）和增亮操作（如下右图）。



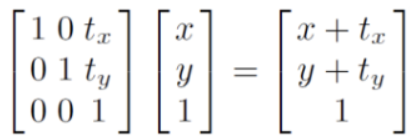
* 1. **几何变换**

几何变换，又称空间变换，是图形处理的一个方面，是各种图形处理算法的基础。它将一幅图像中的坐标位置映射到另一幅图像中的新坐标位置，其实质是改变像素的空间位置，估算新空间位置上的像素值。几何变换不改变图像的像素值， 只是在图像平面上进行像素的重新安排。适当的几何变换可以最大程度地消除由于成像角度、透视关系乃至镜头自身原因所造成的几何失真所产生的负面影响。几何变换常常作为图像处理应用的预处理步骤， 是图像归一化的核心工作之一。此外几何变换也是深度学习中数据增强的一种常用方法。

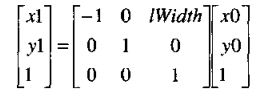
课设中提供了平移、镜像、旋转三种几何变换，前两种都不改变原图像大小，旋转则是因为映射的特殊性质要改变原来的图像，除此外，它们都是通过对应的矩阵变换，复制源图像，在新图像上找到源图像矩阵逆变换的像素进行赋值。

* + 1. 平移

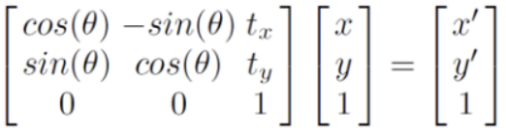
矩阵操作如下。



* + 1. 镜像



* + 1. 旋转



具体映射操作见代码附录。

* 1. **图像风格化**

课设中实现了三种风格变换。

* + 1. 旧照片效果

对输入的图像进行一系列处理，包括调整颜色通道、添加暗角和光晕效果、应用老化纹理和噪点等。

遍历图像的每个像素，提取像素的红色、绿色和蓝色通道值。通过权重调整颜色通道，使其偏向黄色和绿色。计算像素到图像中心的距离，并根据距离添加暗角效果，使离中心越远的像素变暗。根据距离添加光晕效果，使离中心越远的像素增加一定的光晕亮度。根据一定的概率，应用老化纹理和噪点效果，给像素添加一些纹理和随机噪点。

* + 1. 描边效果

这个算法通过边缘提取、滤波、下采样和饱和度降低等一系列处理步骤，实现了一种特定的图像风格化效果。

将输入图像转换为灰度图像，以便使用Canny边缘检测算法对灰度图像进行边缘提取，得到边缘图像。通过阈值化操作，将边缘图像转换为边缘掩膜。边缘掩膜中的边缘区域为白色，其他区域为黑色。

对输入图像进行下采样，即缩小图像尺寸。使用双边滤波算法对下采样后的图像进行滤波处理，以平滑图像并保留边缘信息。再次对滤波后的图像进行下采样，进一步降低图像尺寸。使用金字塔向上采样的方式，将下采样后的图像恢复到原始尺寸。

将恢复尺寸后的图像转换为HSV色彩空间。降低图像的饱和度，通过减去一个固定值的方式实现。对饱和度降低后的图像进行掩膜叠加，将掩膜中的部分保留下来，其他部分根据原始图像填充。

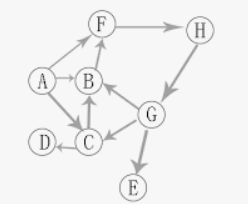
* + 1. 油画效果

主要通过计算局部均值和应用纹理来改变图像的外观。

对图像中的每个像素进行处理，计算以当前像素为中心的局部均值，遍历周围的像素并计算红、绿、蓝通道的均值。将均值量化为离散的颜色值，根据设定的强度级别进行量化，根据纹理缩放比例计算在纹理图像中的坐标，获取纹理像素的颜色值。将局部均值与纹理像素的颜色值进行调制，生成最终的颜色值。

* 1. **区域计数和特征计算**

这里实现的区域计数的前提是图像增强并进行过二值化处理分出了前景和背景，根据图的广度优先算法（如下图）搜索未访问的前景像素的邻域，搜索到邻域为前景时标记为已访问，以此方法遍历所有像素，发现一个新的前景区域计数加一，来确定图中的前景物体数量。



同时，在每个区域，将该区域访问过的像素个数相加得到该区域的面积，并记录这些像素中最小和最大x值y值来获取这个区域的外接矩形。



搜索出所有前景物体后，对最大的那个物体做边界追踪，确定这个物体的周长。边界追踪边界追踪算法通常应用于二值图像（黑白图像），其中边界由具有不同像素值的区域分隔。以下是边界追踪算法的一般步骤：

1. 输入：二值图像（黑白图像）。
2. 选择一个起始点作为边界的起点。
3. 沿着边界的方向追踪像素点，根据邻域像素的连接关系确定下一个像素点的位置。
4. 继续追踪，直到回到起始点，形成一个闭合的边界。
5. 根据需要，可以对边界进行进一步处理，例如计算边界的长度、计算边界的面积等。

边界追踪算法的关键在于确定下一个像素点的位置。常见的边界追踪算法有以下几种：

1. 4-邻域边界追踪：在4个相邻的像素点中选择一个未访问过的像素点作为下一个点进行追踪。
2. 8-邻域边界追踪：在8个相邻的像素点中选择一个未访问过的像素点作为下一个点进行追踪，这样可以更好地处理对角线方向上的边界。

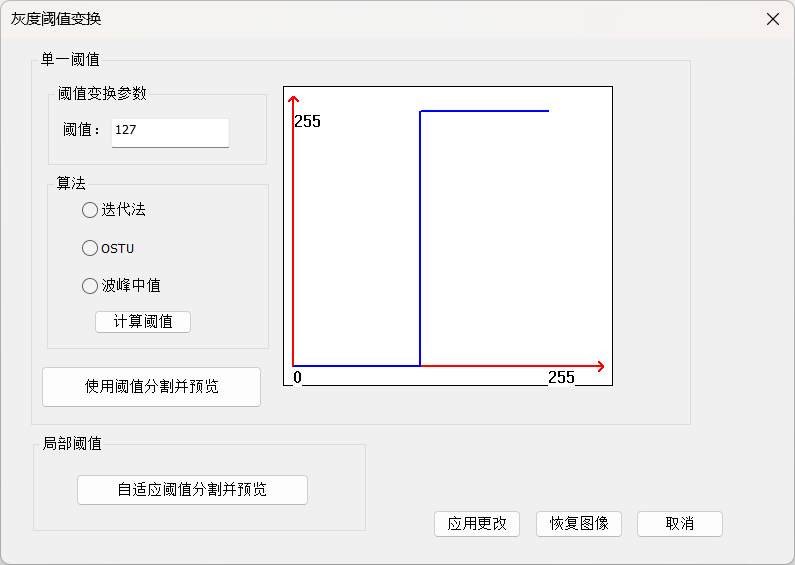
# 算法的实现效果(界面及算法的效果等)

## **灰度的阈值变换**

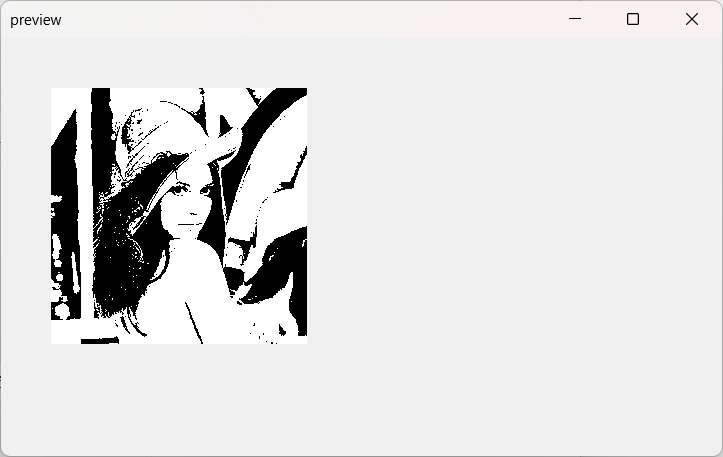
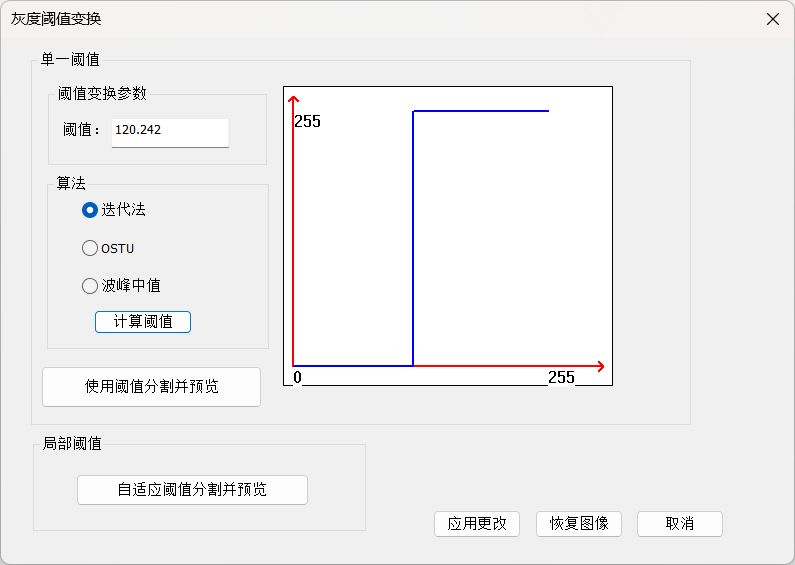
* + 1. 单一阈值变换
       1. 自定义阈值分割

示例图像：

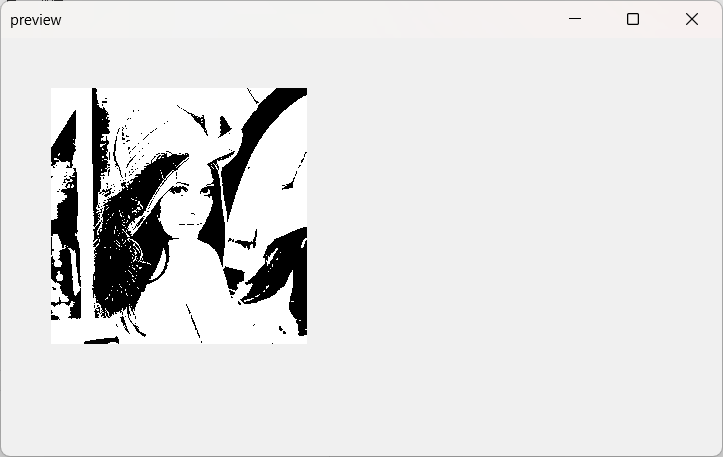
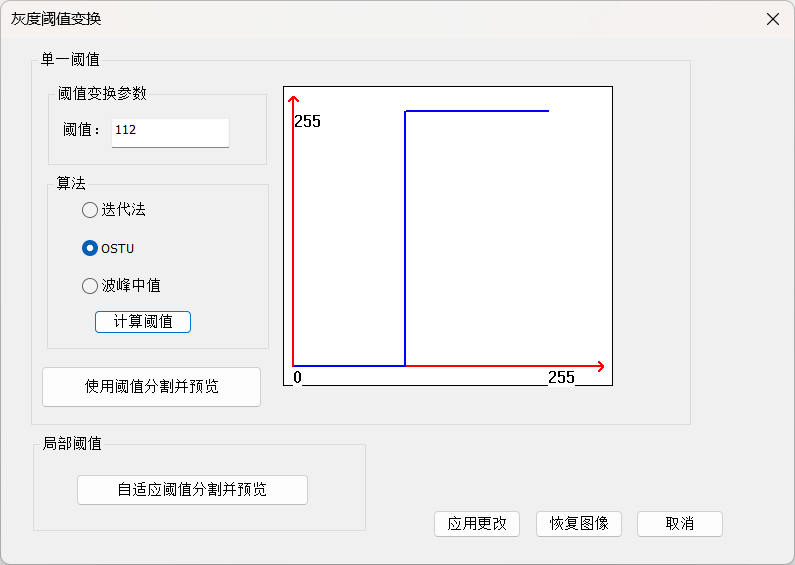




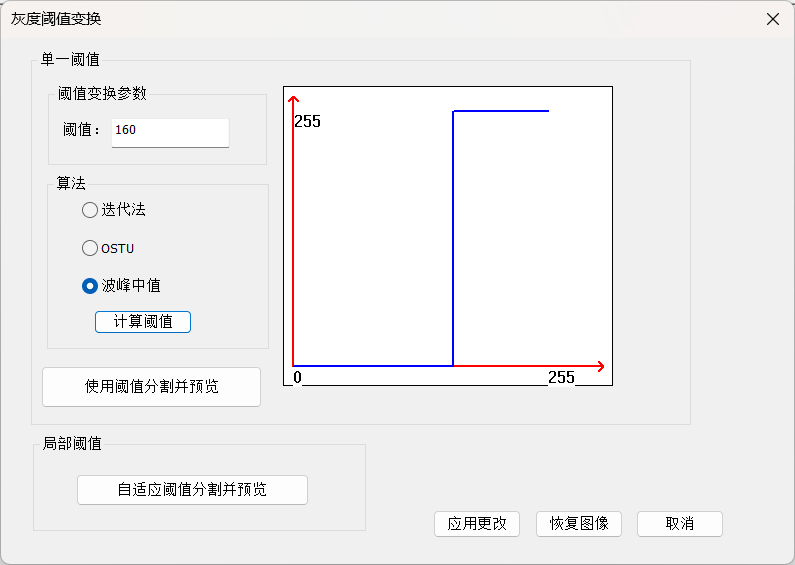
* + - 1. 迭代法



* + - 1. OSTU

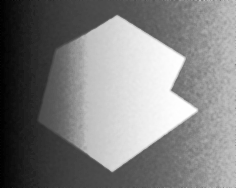


* + - 1. 波峰中值



* + 1. 自适应阈值分割

示例图像：



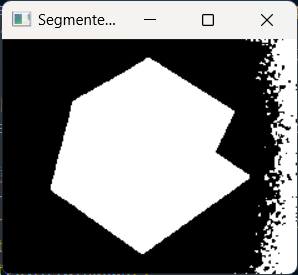
使用单一阈值：



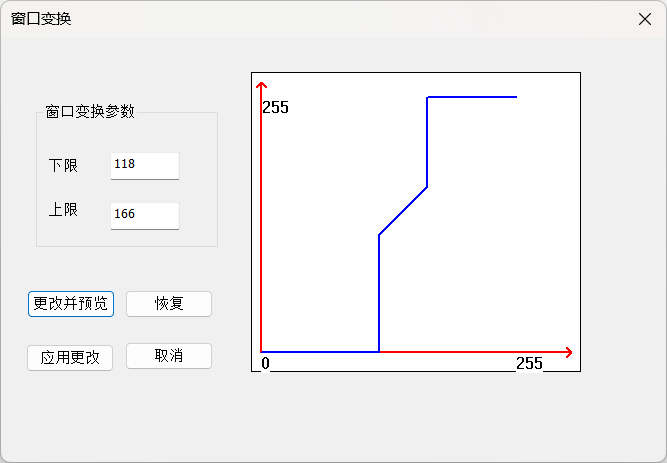
使用自适应阈值：

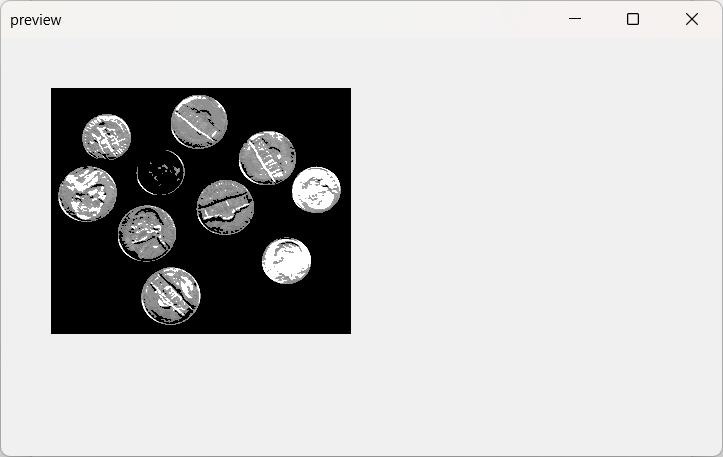


Python中用cv2.adaptiveThreshold函数实现效果（block\_size = 191 # 邻域大小constant = 2 # 阈值偏移量）



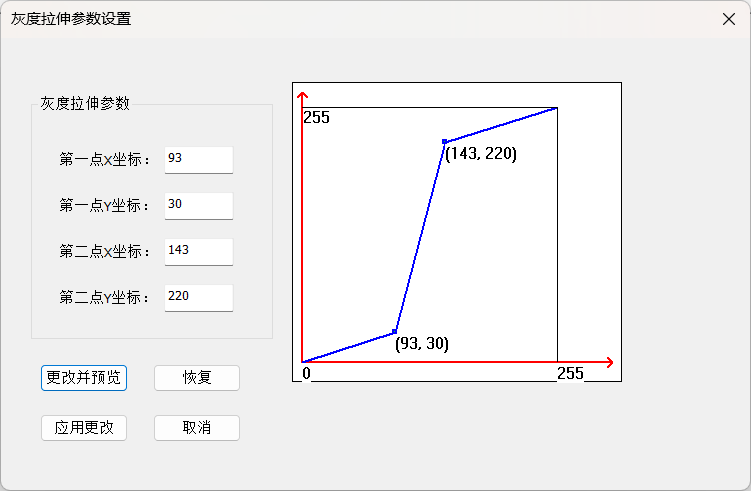
## **灰度的窗口变换**

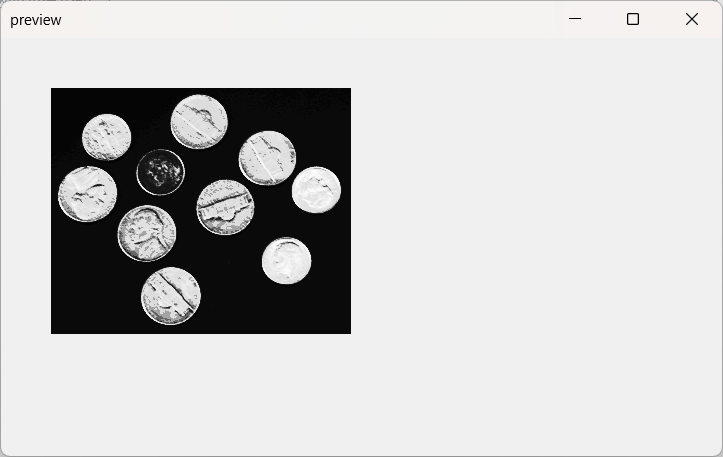


原图 拉伸后

## **灰度拉伸**



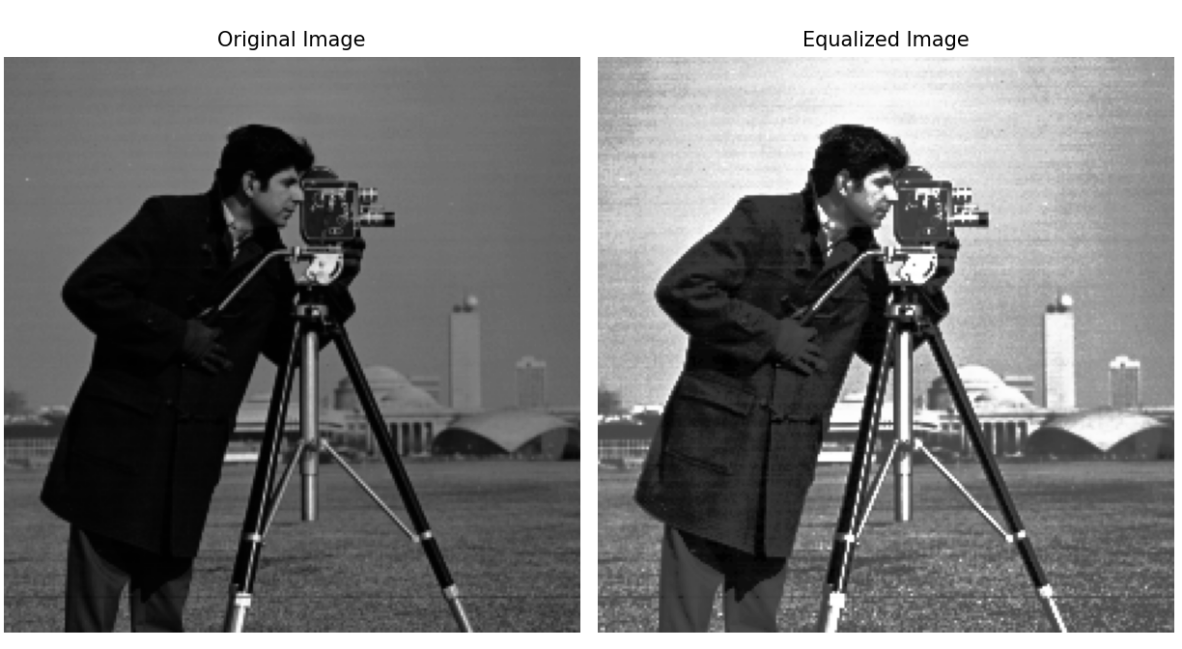
 

原图 拉伸后

## **图像的直方图均衡化**

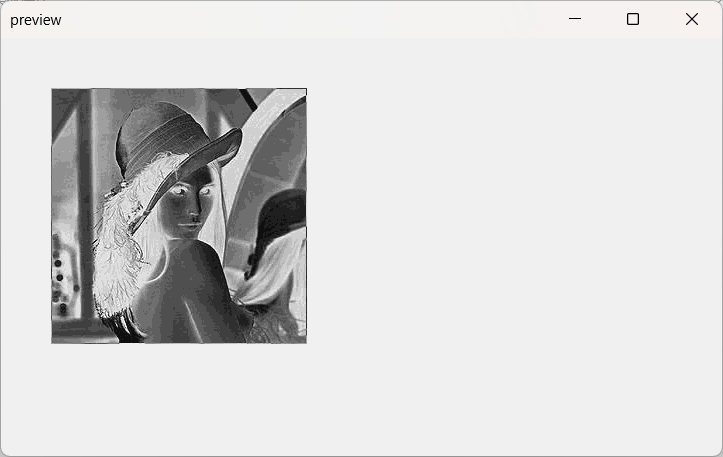
 

原图 均衡化后



## **滤镜（模板操作）**

### 反色变换（底片效果）

原图 反色

### 平滑/模糊（均值模糊，高斯模糊）

噪点图 均值模糊 高斯模糊

### 锐化（拉普拉斯）

原图 锐化后

### 边缘检测

Sobel

Prewitt

### 浮雕效果

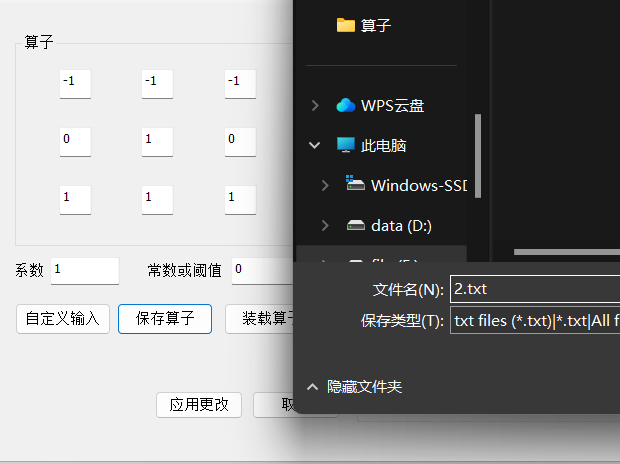
### 改变亮度

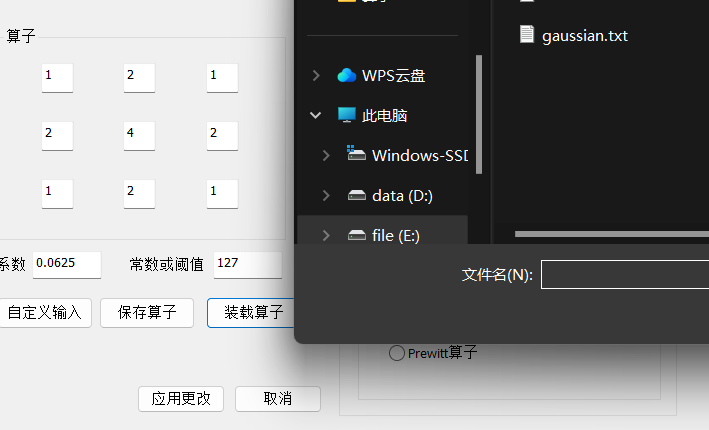
新增按钮可以实现改变亮度：



### 自选滤镜



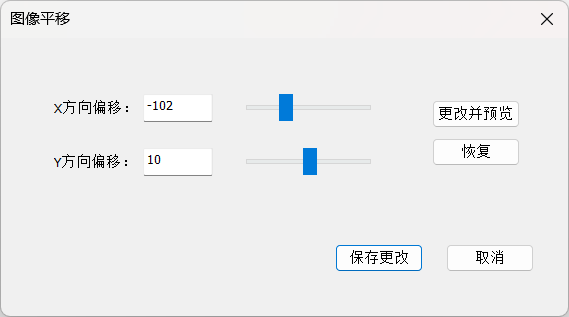
保存为txt文件



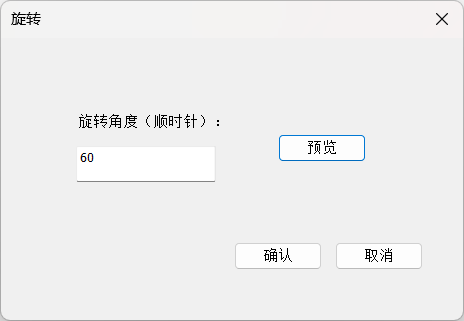
装载算子

## **图像的几何变换**

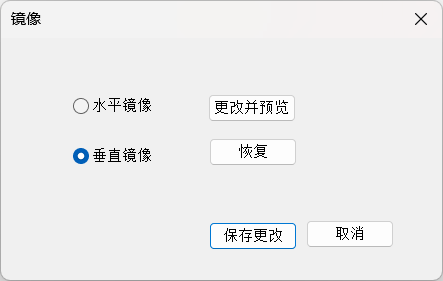
* + 1. 平移

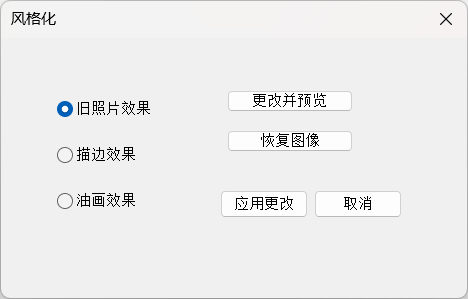
* + 1. 旋转

* + 1. 镜像

## **图像的多种风格化**





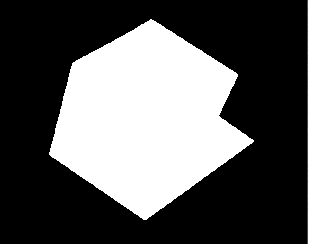
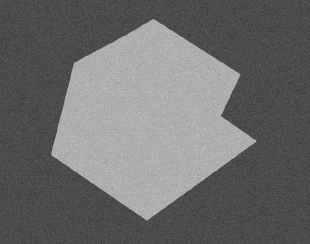
原图 旧照片效果 描边效果

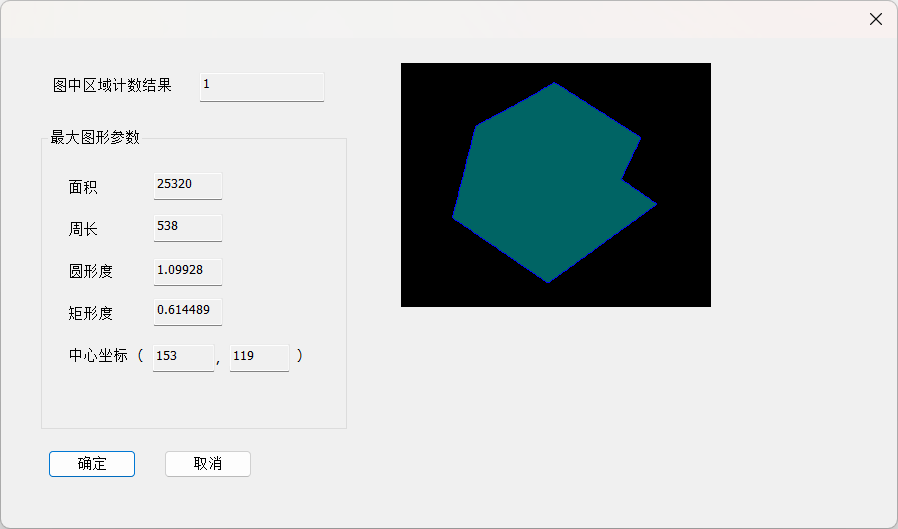


原图 油画效果

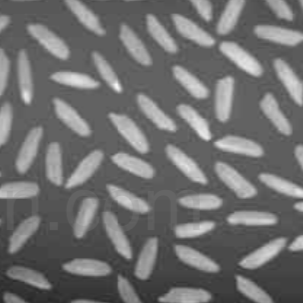
## **应用**

* + 1. 单个多边形计算

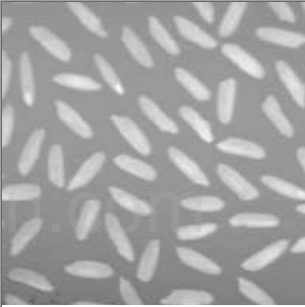
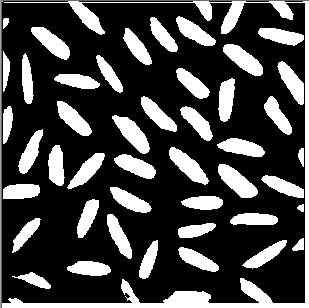




* + 1. 米粒数目计算



原图

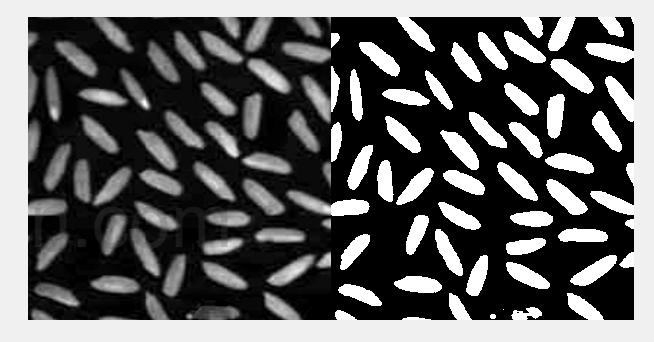
  

灰度拉伸后 自适应阈值分割 形态学处理后的二值图像

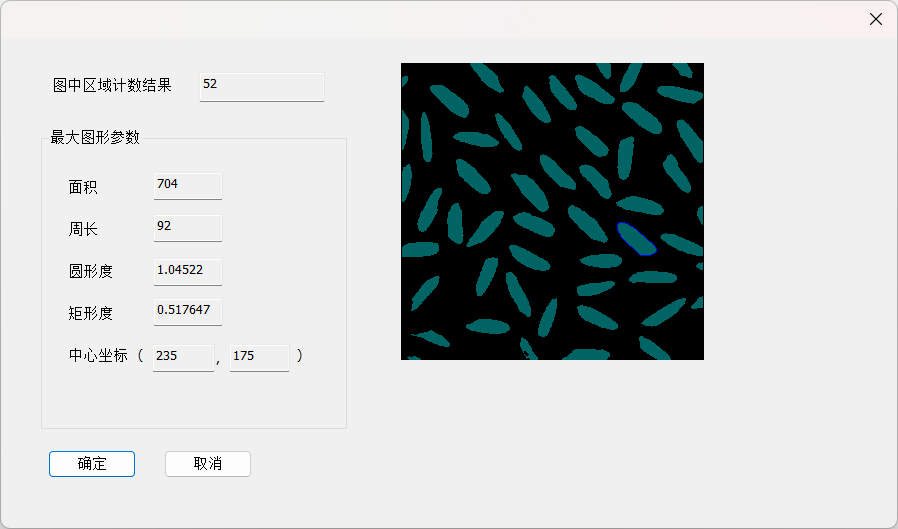
对比：Matlab去除背景的处理方法：

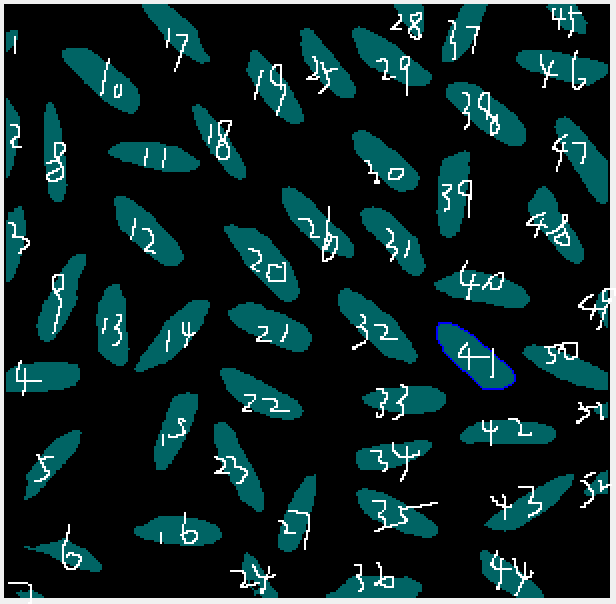


原图 开运算提取的背景 原图-背景



用大津法二值化

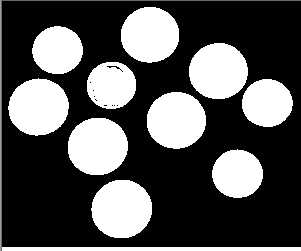
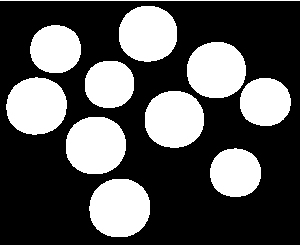




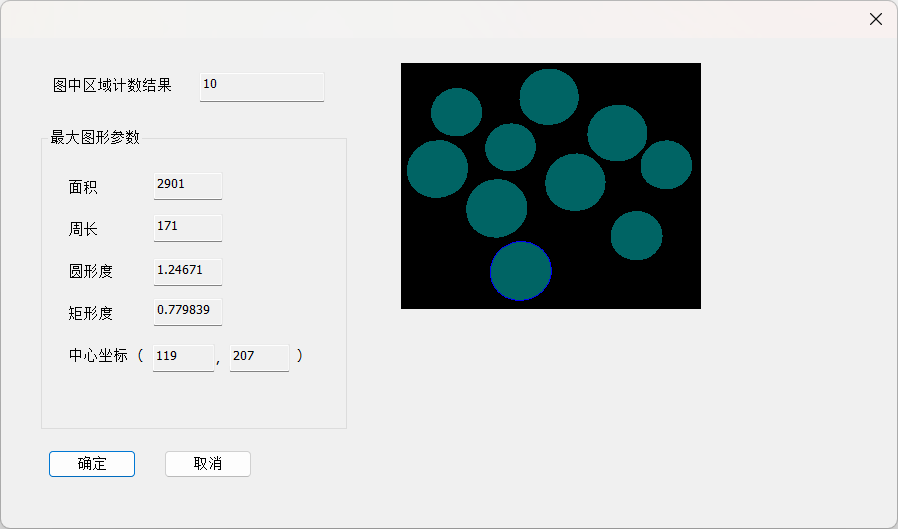
* + 1. 硬币计数

原图 灰度拉伸

自适应阈值分割 形态学处理



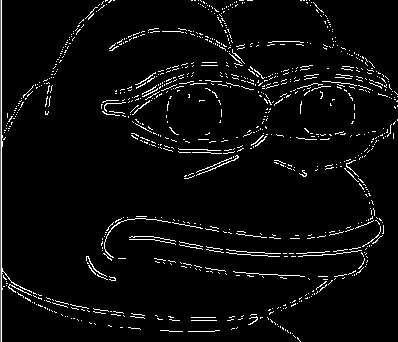
新增功能：区域面积过小的不计入

可以看到原图中处理并不干净，有白色的小污点，但是计数并不影响。



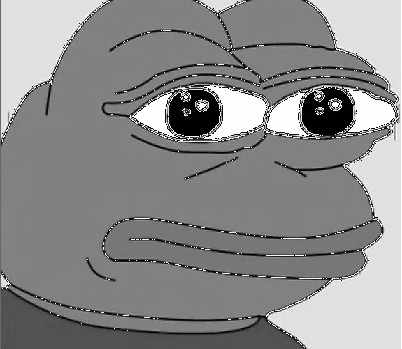
* 1. 选做部分
     1. Canny算子

实现效果如下。

* + 1. Usm锐化

实现效果不太理想。原图+掩膜运算似乎只是将白色边缘做了叠加，没有起到锐化效果。

但是做相反运算（原图-掩膜运算）可以起到模糊作用。

# 结论

课设中实现的功能主要包括图像增强和图像分析计数。这两部分功能相互关联，图像增强的目的是为了提升图像质量，以便更好地进行图像分析计数。

1. 图像增强：

灰度变换：通过调整图像的灰度级别或对比度，改变图像的整体亮度和对比度，以增强图像的视觉效果和细节信息。

模板运算：利用滤波器或卷积核对图像进行模板运算，如均值滤波、高斯滤波等，用于平滑图像、去噪或边缘增强。

灰度直方图运算：通过计算图像的灰度直方图，可以了解图像中各个灰度级别的像素分布情况，进而调整图像的对比度、亮度等参数。

二值化的形态学处理：将图像转换为二值图像，通过形态学处理操作（如膨胀、腐蚀、开闭运算等）对目标进行提取、填充空洞等操作，以便后续的图像分析计数。

1. 图像分析计数：

对增强后的图像进行特定目标的检测、分割和计数。可以利用边缘检测、形态学处理、阈值分割等技术，对目标进行定位和提取。

另外课设中未实现但是进一步地可以基于颜色、纹理、形状等特征提取，还以结合计算机视觉和图像处理技术，进行目标跟踪、轨迹分析等复杂的图像分析任务。

通过图像增强技术，可以改善图像质量和细节，使图像更适合进行后续的图像分析计数任务。图像分析计数的目的是从图像中提取有用的信息，对目标进行定位、计数和分析，以满足特定的应用需求。

# 参考文献

# 附录(代码部分)

* 1. 迭代法求阈值

float Iteration(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight, float delta\_T)

{

//求最大灰度和最小灰度

int sum\_his = 0;

// 指向源图像的指针

unsigned char\* lpSrc;

// 图像每行的字节数

const size\_t lLineBytes = WIDTHBYTES(lWidth \* 8);

for (size\_t j = 0; j < lHeight; j++)

{

for (size\_t i = 0; i < lWidth; i++)

{

lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* i + j;

sum\_his += \*lpSrc;

}

}

float old\_T = sum\_his \* 1.0 / (lHeight \* lWidth);

float new\_T = 0;

float dT = new\_T - old\_T;

while (abs(dT) > delta\_T)

{

int G1 = 0;

int G2 = 0;

int timer\_G1 = 0;

int timer\_G2 = 0;

for (size\_t j = 0; j < lHeight; j++)

{

for (size\_t i = 0; i < lWidth; i++)

{

lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* i + j;

if(\*lpSrc> old\_T)

{

G1 += \*lpSrc;

timer\_G1++;

}

else {

G2 += \*lpSrc;

timer\_G2++;

}

}

}

new\_T = 0.5 \* ((G1 \* 1.0f / timer\_G1) + (G2 \* 1.0f / timer\_G2));

dT = new\_T - old\_T;

old\_T = new\_T;

}

return new\_T;

}

* 1. 大津法求阈值

int Ostu(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight)

{

// 指向源图像的指针

unsigned char\* lpSrc;

// 图像每行的字节数

const size\_t lLineBytes = WIDTHBYTES(lWidth \* 8);

int T = 0; //大津阈值

int size = lHeight \* lWidth;

float variance; //类间方差

float maxVariance = 0, w1 = 0, w2 = 0, avgValue = 0;

float u0 = 0, u1 = 0, u2 = 0;

//生成灰度直方图

int pixels[256];

float histgram[256];

for (int i = 0; i < 256; i++)

{

pixels[i] = 0;

}

for (size\_t j = 0; j < lHeight; j++)

{

for (size\_t i = 0; i < lWidth; i++)

{

lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* i + j;

pixels[\*lpSrc]++;

}

}

for (int i = 0; i < 256; i++)

{

histgram[i] = pixels[i] \* 1.0f / size;

}

//遍历找出类间方差最大（maxVariance）的阈值（Ostu\_Threshold）

for (int i = 0; i <= 255; i++)

{

w1 = 0;

w2 = 0;

u1 = 0;

u2 = 0;

//计算背景像素占比,平均灰度

for (int j = 0; j <= i; j++)

{

w1 += histgram[j];

u1 += histgram[j] \* j;

}

u1 = u1 / w1;

//计算前景像素占比,平均灰度

w2 = 1 - w1;

if (i == 255)

{

u2 = 0;

}

else

{

for (int j = i + 1; j <= 255; j++)

{

u2 += histgram[j] \* j;

}

}

u2 = u2 / w2;

//计算类间方差

variance = w1 \* w2 \* (u1 - u2) \* (u1 - u2);

if (variance > maxVariance)

{ //找到使灰度差最大的值

maxVariance = variance;

T = i; //那个值就是阈值

}

}

return T;

}

* 1. 波峰中值

float HisValley(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight)

{

// 计算灰度直方图

int histogram[256] = { 0 };

for (size\_t j = 0; j < lHeight; j++)

{

for (size\_t i = 0; i < lWidth; i++)

{

unsigned char\* lpSrc = lpDIBBits + j \* lWidth + i;

histogram[\*lpSrc]++;

}

}

// 计算频率分布图

float freqDistribution[256] = { 0 };

int totalPixels = lHeight \* lWidth;

for (int i = 0; i < 256; i++)

{

freqDistribution[i] = (float)histogram[i] / totalPixels;

}

// 找到使得累计概率和为0.5的分割点

int splitIndex = 0;

float cumulativeSum = 0.0;

while (cumulativeSum < 0.5)

{

cumulativeSum += freqDistribution[splitIndex];

splitIndex++;

}

// 在两段分割后的直方图中分别寻找峰值

int peak1 = 0;

int peak2 = 0;

int peak1\_index = 0;

int peak2\_index = 0;

// 在前半段寻找峰值

for (int i = 0; i < splitIndex; i++)

{

if (histogram[i] > peak1)

{

peak1 = histogram[i];

peak1\_index = i;

}

}

// 在后半段寻找峰值

for (int i = splitIndex; i < 256; i++)

{

if (histogram[i] > peak2)

{

peak2 = histogram[i];

peak2\_index = i;

}

}

// 返回两个峰值的平均值作为阈值

float threshold = (peak1\_index + peak2\_index) / 2.0;

return threshold;

}

* 1. 自适应阈值分割

BOOL WINAPI adaptiveThresholding(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight, int windowSize)

{

// 指向复制图像的指针

unsigned char\* lpNewDIBBits;

// 指向源图像的指针

unsigned char\* lpSrc;

// 指向要复制区域的指针

unsigned char\* lpDst;

int halfSize = windowSize / 2;

int n = 3;

int m = 2;

// 计算结果

FLOAT fResult;

// 图像每行的字节数

LONG lLineBytes;

// 计算图像每行的字节数

lLineBytes = WIDTHBYTES(lWidth \* 8);

lpNewDIBBits = new BYTE[lLineBytes \* lHeight];

// 初始化图像为原始图像

memcpy(lpNewDIBBits, lpDIBBits, lLineBytes \* lHeight);

int regionWidth = lWidth / n;

int regionHeight = lHeight / m;

//区域阈值分割（对边缘做处理）

for (int i = 0; i < m; ++i)

{

for (int j = 0; j < n; ++j)

{

int regionStartX = j \* regionWidth;

int regionStartY = i \* regionHeight;

lpDst = (unsigned char\*)lpNewDIBBits + lLineBytes \* (lHeight - 1 - regionStartY) + regionStartX;

int histogram[256] = { 0 };

// 统计区域的灰度直方图

for (int y = regionStartY; y < regionStartY + regionHeight; ++y)

{

for (int x = regionStartX; x < regionStartX + regionWidth; ++x)

{

lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* (lHeight - 1 - y) + x;

++histogram[\*lpSrc];

}

}

int totalPixels = regionWidth \* regionHeight;

float sum = 0.0;

for (int k = 0; k < 256; ++k)

sum += k \* histogram[k];

float sumB = 0.0;

int wB = 0;

int wF = 0;

float maxVariance = 0.0;

int threshold = 0;

for (int k = 0; k < 256; ++k)

{

wB += histogram[k];

if (wB == 0)

continue;

wF = totalPixels - wB;

if (wF == 0)

break;

sumB += k \* histogram[k];

float meanB = sumB / wB;

float meanF = (sum - sumB) / wF;

float varianceBetween = wB \* wF \* (meanB - meanF) \* (meanB - meanF);

if (varianceBetween > maxVariance)

{

maxVariance = varianceBetween;

threshold = k;

}

}

// 应用阈值

for (int y = regionStartY; y < regionStartY + regionHeight; ++y)

{

for (int x = regionStartX; x < regionStartX + regionWidth; ++x)

{

lpDst = (unsigned char\*)lpNewDIBBits + lLineBytes \* (lHeight - 1 - y) + x;

if (\*lpDst > threshold)

\*lpDst = 255;

else

\*lpDst = 0;

}

}

}

}

//自适应阈值分割

for (int y = halfSize; y < lHeight - halfSize; y++)

{

for (int x = halfSize; x < lWidth - halfSize; x++)

{

int sum = 0;

lpDst = (unsigned char\*)lpNewDIBBits + lLineBytes \* (lHeight - 1 - y) + x;

for (int i = -halfSize; i <= halfSize; i++)

{

for (int j = -halfSize; j <= halfSize; j++)

{

lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* (lHeight - 1 - y - i) + x + j;

sum += (\*lpSrc);

}

}

int mean = sum / (windowSize \* windowSize);

if (\*lpDst > mean - 5)

\*lpDst = 255;

else

\*lpDst = 0;

}

}

// 复制变换后的图像

memcpy(lpDIBBits, lpNewDIBBits, lLineBytes \* lHeight);

// 返回

return TRUE;

}

* 1. 窗口变换

BOOL WINAPI Winchange(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight,int lowerLimit, int higherLimit) {

// 指向源图像的指针

unsigned char\* lpSrc;

// 图像每行的字节数

const size\_t lLineBytes = WIDTHBYTES(lWidth \* 8);

//逐行扫描

for (size\_t i = 0; i < lHeight; i++)

{

//逐列扫描

for (size\_t j = 0; j < lWidth; j++)

{

// 指向DIB第i行，第j个象素的指针

lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* i + j;

if (\*lpSrc < lowerLimit) \*lpSrc = 0;

else if (\*lpSrc > higherLimit) \*lpSrc = 255;

}

}

return TRUE;

}

* 1. 灰度拉伸

BOOL WINAPI HisStretch(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight, int x1, int y1, int x2, int y2) {

// 指向源图像的指针

unsigned char\* lpSrc;

// 图像每行的字节数

const size\_t lLineBytes = WIDTHBYTES(lWidth \* 8);

std::pair<float, float> line\_equation1 = calculateLine(0, 0, x1, y1);

std::pair<float, float> line\_equation2 = calculateLine(x1, y1, x2, y2);

std::pair<float, float> line\_equation3 = calculateLine(x2, y2, 255, 255);

//逐行扫描

for (size\_t i = 0; i < lHeight; i++)

{

//逐列扫描

for (size\_t j = 0; j < lWidth; j++)

{

// 指向DIB第i行，第j个象素的指针

lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* i + j;

if (\*lpSrc < x1) \*lpSrc = Limit(line\_equation1.first \* (\*lpSrc) + line\_equation1.second);

else if (\*lpSrc > x2) \*lpSrc = Limit(line\_equation3.first \* (\*lpSrc) + line\_equation3.second);

else \*lpSrc = Limit(line\_equation2.first \* (\*lpSrc) + line\_equation2.second);

}

}

return TRUE;

}

* 1. 直方图均衡化

BOOL WINAPI HisEquation(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight) {

// 指向源图像的指针

unsigned char\* lpSrc;

// 图像每行的字节数

const size\_t lLineBytes = WIDTHBYTES(lWidth \* 8);

//像素总数

int sum = lWidth \* lHeight;

//灰度统计表

int hist[256] = { 0 };

//概率

double pHist[256] = { 0 };

//累积概率

double disHist[256] = { 0 };

//新的灰度映射表

unsigned char newHist[256] = { 0 };

for (size\_t i = 0; i < lHeight; i++)

{

//逐列扫描

for (size\_t j = 0; j < lWidth; j++)

{

// 指向DIB第i行，第j个象素的指针

lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* i + j;

hist[\*lpSrc]++;

}

}

for (int i = 0; i < 256; i++) {

pHist[i] = ((double)hist[i] /sum);

}

for (int i = 1; i < 256; i++) {

disHist[i] = disHist[i - 1] + pHist[i];

}

for (int i = 0; i < 256; i++) {

newHist[i] = (unsigned char)(255 \* disHist[i] + 0.5);

}

for (size\_t i = 0; i < lHeight; i++)

{

//逐列扫描

for (size\_t j = 0; j < lWidth; j++)

{

// 指向DIB第i行，第j个象素的指针

lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* i + j;

\*lpSrc = newHist[\*lpSrc];

}

}

return TRUE;

}

* 1. 模板函数

BOOL WINAPI Template(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight, float\* fpArray, float fCoef, int fThre)

{

// 指向复制图像的指针

unsigned char\* lpNewDIBBits;

// 指向源图像的指针

unsigned char\* lpSrc;

// 指向要复制区域的指针

unsigned char\* lpDst;

// 循环变量

LONG i;

LONG j;

LONG k;

LONG l;

// 计算结果

FLOAT fResult;

// 图像每行的字节数

LONG lLineBytes;

// 计算图像每行的字节数

lLineBytes = WIDTHBYTES(lWidth \* 8);

lpNewDIBBits = new BYTE[lLineBytes \* lHeight];

// 初始化图像为原始图像

memcpy(lpNewDIBBits, lpDIBBits, lLineBytes \* lHeight);

// 行(除去边缘几行)

for (i = 1; i < lHeight - 1; i++)

{

// 列(除去边缘几列)

for (j = 1; j < lWidth - 1; j++)

{

// 指向新DIB第i行，第j个象素的指针

lpDst = (unsigned char\*)lpNewDIBBits + lLineBytes \* (lHeight - 1 - i) + j;

fResult = 0;

// 计算

for (k = 0; k < 3; k++)

{

for (l = 0; l < 3; l++)

{

// 指向DIB第i - 1 + k行，第j - 1 + l个象素的指针

lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* (lHeight - i - k)

+ j - 1 + l;

// 保存象素值

fResult += (\*lpSrc) \* fpArray[k \* 3 + l];

}

}

// 乘上系数

fResult \*= fCoef;

fResult += fThre;

// 取绝对值

fResult = (FLOAT)fabs(fResult);

// 判断是否超过255

if (fResult > 255)

{

// 直接赋值为255

\*lpDst = 255;

}

else if (fResult < 0 )

{

// 直接赋值为255

\*lpDst = 0;

}

else

{

// 赋值

\*lpDst = (unsigned char)(fResult + 0.5);

}

}

}

// 复制变换后的图像

memcpy(lpDIBBits, lpNewDIBBits, lLineBytes \* lHeight);

// 返回

return TRUE;

}

* 1. 图像平移

BOOL WINAPI Translation(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight, LONG lXOffset, LONG lYOffset)

{

// 指向复制图像的指针

unsigned char\* lpNewDIBBits;

// 指向源图像的指针

unsigned char\* lpSrc;

// 指向要复制区域的指针

unsigned char\* lpDst;

// 循环变量

LONG i;

LONG j;

LONG i0;

LONG j0;

// 计算结果

FLOAT fResult;

// 图像每行的字节数

LONG lLineBytes;

// 计算图像每行的字节数

lLineBytes = WIDTHBYTES(lWidth \* 8);

lpNewDIBBits = new BYTE[lLineBytes \* lHeight];

// 初始化图像为原始图像

memcpy(lpNewDIBBits, lpDIBBits, lLineBytes \* lHeight);

// 行(除去边缘几行)

for (i = 0; i < lHeight; i++)

{

// 列(除去边缘几列)

for (j = 0; j < lWidth; j++)

{

// 指向新DIB第i行，第j个象素的指针

lpDst = (unsigned char\*)lpNewDIBBits + lLineBytes \* (lHeight - 1 - i) + j;

j0 = j - lXOffset;

i0 = i - lYOffset;

if ((j0 >= 0) && (j0 < lWidth) && (i0 >= 0) && (i0 < lHeight))

{

**//映射公式**

**lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* (lHeight - 1 - i0) + j0;**

\*lpDst = \*lpSrc;

}

else {

\*lpDst = 255;

}

}

}

// 复制变换后的图像

memcpy(lpDIBBits, lpNewDIBBits, lLineBytes \* lHeight);

// 返回

return TRUE;

}

* 1. 图像镜像

BOOL WINAPI Mirror(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight, int Dire)

{

// 指向复制图像的指针

unsigned char\* lpNewDIBBits;

// 指向源图像的指针

unsigned char\* lpSrc;

// 指向要复制区域的指针

unsigned char\* lpDst;

// 循环变量

LONG i;

LONG j;

// 计算结果

FLOAT fResult;

// 图像每行的字节数

LONG lLineBytes;

// 计算图像每行的字节数

lLineBytes = WIDTHBYTES(lWidth \* 8);

lpNewDIBBits = new BYTE[lLineBytes \* lHeight];

// 初始化图像为原始图像

memcpy(lpNewDIBBits, lpDIBBits, lLineBytes \* lHeight);

// 行(除去边缘几行)

for (i = 0; i < lHeight; i++)

{

// 列(除去边缘几列)

for (j = 0; j < lWidth; j++)

{

if (j == 129)

{

int t = 0;

}

// 指向新DIB第i行，第j个象素的指针

lpDst = (unsigned char\*)lpNewDIBBits + lLineBytes \* (lHeight - 1 - i) + j;

**//映射公式**

**//如果是水平镜像**

**if (Dire == 0) {**

**lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* (lHeight - 1 - i) + lWidth - j;**

**}**

**//如果是垂直镜像**

**else {**

**lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* i + j;**

**}**

\*lpDst = \*lpSrc;

}

}

// 复制变换后的图像

memcpy(lpDIBBits, lpNewDIBBits, lLineBytes \* lHeight);

// 返回

return TRUE;

}

* 1. 图像旋转

HGLOBAL WINAPI Rotate(LPSTR lpDIB, LPSTR lpDIBBits, LONG lWidth, LONG lHeight, int angle)

{

// 指向复制图像的指针

LPSTR lpNewDIB;

LPSTR lpNewDIBBits;

// 指向源像素的指针

LPSTR lpSrc;

// 指向新像素的指针

LPSTR lpDst;

// 循环变量

LONG i;

LONG j;

LONG i0;

LONG j0;

//旋转后的新宽度和高度

LONG lNewWidth;

LONG lNewHeight;

// 图像每行的字节数

LONG lLineBytes;

LONG lNewLineBytes;

// 计算原图像每行的字节数

lLineBytes = WIDTHBYTES(lWidth \* 8);

// 原图四个角的坐标（以图像中心为坐标系原点）

float fSrcX1, fSrcY1, fSrcX2, fSrcY2, fSrcX3, fSrcY3, fSrcX4, fSrcY4;

// 旋转后四个角的坐标（以图像中心为坐标系原点）

float fDstX1, fDstY1, fDstX2, fDstY2, fDstX3, fDstY3, fDstX4, fDstY4;

float fRotateAngle = (angle \* PI \* 1.0) / 180;//角度转换为弧度

float fSina = (float)sin((double)fRotateAngle);

float fCosa = (float)cos((double)fRotateAngle);

//原图的四个点坐标

fSrcX1 = (float)(-(lWidth - 1) / 2);

fSrcY1 = (float)((lHeight - 1) / 2);

fSrcX2 = (float)((lWidth - 1) / 2);

fSrcY2 = (float)((lHeight - 1) / 2);

fSrcX3 = (float)((lWidth - 1) / 2);

fSrcY3 = (float)(-(lHeight - 1) / 2);

fSrcX4 = (float)(-(lWidth - 1) / 2);

fSrcY4 = (float)(-(lHeight - 1) / 2);

//新图的四个点坐标

fDstX1 = fSrcX1 \* fCosa + fSrcY1 \* fSina;

fDstY1 = -fSrcX1 \* fSina + fSrcY1 \* fCosa;

fDstX2 = fSrcX2 \* fCosa + fSrcY2 \* fSina;

fDstY2 = -fSrcX2 \* fSina + fSrcY2 \* fCosa;

fDstX3 = fSrcX3 \* fCosa + fSrcY3 \* fSina;

fDstY3 = -fSrcX3 \* fSina + fSrcY3 \* fCosa;

fDstX4 = fSrcX4 \* fCosa + fSrcY4 \* fSina;

fDstY4 = -fSrcX4 \* fSina + fSrcY4 \* fCosa;

//新图像的长度和宽度

lNewWidth = (LONG)(max(fabs(fDstX1 - fDstX3), fabs(fDstX2 - fDstX4)) + 0.5);

lNewHeight = (LONG)(max(fabs(fDstY1 - fDstY3), fabs(fDstY2 - fDstY4)) + 0.5);

//映射公式中的两个常数

float f1 = (float)(-0.5 \* (lNewWidth - 1) \* fCosa - 0.5 \* (lNewHeight - 1) \* fSina + 0.5 \* (lWidth - 1));

float f2 = (float)(0.5 \* (lNewWidth - 1) \* fSina - 0.5 \* (lNewHeight - 1) \* fCosa + 0.5 \* (lHeight - 1));

//新图像每行的字节数

lNewLineBytes = WIDTHBYTES(lNewWidth \* 8);

HDIB hDIB;

hDIB = (HDIB) ::GlobalAlloc(GHND, lNewLineBytes \* lNewHeight + \*(LPDWORD)lpDIB + ::PaletteSize(lpDIB));

if (hDIB == NULL) {

return NULL;

}

lpNewDIB = (char\*) ::GlobalLock((HGLOBAL)hDIB);

memcpy(lpNewDIB, lpDIB, \*(LPDWORD)lpDIB + ::PaletteSize(lpDIB));

lpNewDIBBits = ::FindDIBBits(lpNewDIB);

LPBITMAPINFOHEADER lpbmi = (LPBITMAPINFOHEADER)lpNewDIB;

LPBITMAPCOREHEADER lpbmc = (LPBITMAPCOREHEADER)lpNewDIB;

lpbmi->biWidth = lNewWidth;

lpbmi->biHeight = lNewHeight;

for (i = 0; i < lNewHeight; i++)

{

for (j = 0; j < lNewWidth; j++)

{

// 指向新DIB第i行，第j个象素的指针

lpDst = (char\*)lpNewDIBBits + lNewLineBytes \* (lNewHeight - 1 - i) + j;

**//映射公式**

**i0 = (LONG)(-((float)j) \* fSina + ((float)i) \* fCosa + f2 + 0.5);**

**j0 = (LONG)( ((float)j) \* fCosa + ((float)i) \* fSina + f1 + 0.5);**

if ((j0 >= 0) && (j0 < lWidth) && (i0 >= 0) && (i0 < lHeight))

{

lpSrc = (char\*)lpDIBBits + lLineBytes \* (lHeight - 1 - i0) + j0;

\*lpDst = \*lpSrc;

}

else {

\*lpDst = 255;

}

}

}

// 返回

return hDIB;

}

* 1. 旧照片效果

BOOL WINAPI Stylize1(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight, int lpSrcBitCount) {

unsigned char\* lpSrc;

// 图像每行的字节数

LONG lLineBytes = WIDTHBYTES(lWidth \* 24);

// 调整颜色通道，添加暗角和光晕效果，并应用老化纹理和噪点

for (int i = 0; i < lHeight; i++) {

for (int j = 0; j < lWidth; j++) {

lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* i + j \* 3;

int red = lpSrc[2];

int green = lpSrc[1];

int blue = lpSrc[0];

// 调整颜色通道，偏向黄色和绿色

int newRed = static\_cast<int>(0.7 \* red + 0.3 \* green);

int newGreen = static\_cast<int>(0.6 \* green + 0.4 \* red);

int newBlue = static\_cast<int>(0.6 \* blue);

// 添加暗角效果

double distance = std::sqrt((j - lWidth / 2.0) \* (j - lWidth / 2.0) + (i - lHeight / 2.0) \* (i - lHeight / 2.0));

double vignette = 1.0 - 0.8 \* (distance / (lWidth / 2.0));

newRed = static\_cast<int>(newRed \* vignette);

newGreen = static\_cast<int>(newGreen \* vignette);

newBlue = static\_cast<int>(newBlue \* vignette);

// 添加光晕效果

double glow = 1.0 - 0.5 \* (distance / (lWidth / 2.0));

newRed += static\_cast<int>(100 \* glow);

newGreen += static\_cast<int>(80 \* glow);

newBlue += static\_cast<int>(50 \* glow);

// 确保像素值在 0-255 范围内

if (newRed < 0) {

newRed = 0;

}

else if (newRed > 255) {

newRed = 255;

}

if (newGreen < 0) {

newGreen = 0;

}

else if (newGreen > 255) {

newGreen = 255;

}

if (newBlue < 0) {

newBlue = 0;

}

else if (newBlue > 255) {

newBlue = 255;

}

// 应用老化纹理

if (rand() % 100 < 30) {

// 30% 的概率添加纹理效果

int texture = rand() % 50 - 25;

newRed += texture;

newGreen += texture;

newBlue += texture;

}

// 添加噪点

if (rand() % 100 < 5) {

// 5% 的概率添加噪点

int noise = rand() % 50 - 25;

newRed += noise;

newGreen += noise;

newBlue += noise;

}

lpSrc[2] = static\_cast<unsigned char>(newRed);

lpSrc[1] = static\_cast<unsigned char>(newGreen);

lpSrc[0] = static\_cast<unsigned char>(newBlue);

}

}

return TRUE;

}

* 1. 描边效果（使用opencv库）

BOOL WINAPI Stylize3(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight, int lpSrcBitCount) {

// 定义每行像素所占的字节数

int LineBytes = WIDTHBYTES(lWidth \* lpSrcBitCount);

// 创建OpenCV Mat对象来处理图像

unsigned char\* tempBits = lpDIBBits;

cv::Mat frame(lHeight, lWidth, CV\_8UC3, tempBits);

// 转换为灰度图像

cv::Mat gray;

cv::cvtColor(frame, gray, cv::COLOR\_BGR2GRAY);

// 边缘提取

cv::Mat edges;

cv::Canny(gray, edges, 120, 240);

// 生成边缘掩膜

cv::Mat mask;

cv::threshold(edges, mask, 120, 255, cv::THRESH\_BINARY\_INV);

// 进行下采样

cv::Mat downsampled;

cv::pyrDown(frame, downsampled);

cv::pyrDown(downsampled, downsampled);

// 双边滤波

cv::Mat filtered;

cv::bilateralFilter(downsampled, filtered, 5, 30, 20);

// 进行下采样

cv::pyrDown(filtered, downsampled);

cv::pyrDown(downsampled, downsampled);

// 恢复图像大小

cv::Mat result;

cv::pyrUp(downsampled, result);

cv::pyrUp(result, result);

// 降低图像饱和度

cv::Mat hsv;

cv::cvtColor(result, hsv, cv::COLOR\_BGR2HSV);

cv::Mat hsvChannels[3];

cv::split(hsv, hsvChannels);

cv::Mat saturationReduced;

cv::subtract(hsvChannels[1], cv::Scalar(20), saturationReduced);

cv::threshold(saturationReduced, saturationReduced, 0, 255, cv::THRESH\_TRUNC);

cv::add(hsvChannels[1], saturationReduced, hsvChannels[1]);

cv::merge(hsvChannels, 3, result);

cv::cvtColor(result, result, cv::COLOR\_HSV2BGR);

// 掩膜叠加

cv::Mat output;

frame.copyTo(output, mask);

// 将结果复制回lpDIBBits

tempBits = lpDIBBits;

std::memcpy(tempBits, output.data, LineBytes \* lHeight);

return TRUE;

}

* 1. 油画效果

BOOL WINAPI Stylize2(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight, int lpSrcBitCount) {

unsigned char\* lpSrc;

LONG lLineBytes = WIDTHBYTES(lWidth \* 24);

int brushSize = 3;

int intensityLevels = 7;

double textureScale = 0.8;

unsigned char\* lpTempBits = new unsigned char[lLineBytes \* lHeight];

memcpy(lpTempBits, lpDIBBits, lLineBytes \* lHeight);

unsigned char\* lpTextureBits = new unsigned char[lLineBytes \* lHeight];

// 填充纹理图像数据

for (int y = 0; y < lHeight; y++) {

for (int x = 0; x < lWidth; x++) {

int redTotal = 0;

int greenTotal = 0;

int blueTotal = 0;

int count = 0;

for (int i = -brushSize; i <= brushSize; i++) {

for (int j = -brushSize; j <= brushSize; j++) {

int nx = x + j;

int ny = y + i;

if (nx >= 0 && nx < lWidth && ny >= 0 && ny < lHeight) {

lpSrc = lpTempBits + ny \* lLineBytes + nx \* 3;

redTotal += lpSrc[2];

greenTotal += lpSrc[1];

blueTotal += lpSrc[0];

count++;

}

}

}

int avgRed = redTotal / count;

int avgGreen = greenTotal / count;

int avgBlue = blueTotal / count;

int intensityStep = 255 / (intensityLevels - 1);

int newRed = (avgRed / intensityStep) \* intensityStep;

int newGreen = (avgGreen / intensityStep) \* intensityStep;

int newBlue = (avgBlue / intensityStep) \* intensityStep;

int textureX = x \* textureScale;

int textureY = y \* textureScale;

lpSrc = lpTextureBits + textureY \* lLineBytes + textureX \* 3;

newRed = (newRed \* lpSrc[2]) / 255;

newGreen = (newGreen \* lpSrc[1]) / 255;

newBlue = (newBlue \* lpSrc[0]) / 255;

lpSrc = lpDIBBits + y \* lLineBytes + x \* 3;

lpSrc[2] = newRed;

lpSrc[1] = newGreen;

lpSrc[0] = newBlue;

}

}

delete[] lpTempBits;

delete[] lpTextureBits;

return TRUE;

}

* 1. 区域计数

int WINAPI countRegions(unsigned char \* lpDIBBits, LONG lWidth, LONG lHeight, int connection, int color)

{

int count = 0;

// 指向源像素的指针

unsigned char\* lpSrc;

unsigned char\* lpNew;

// 图像每行的字节数

LONG lLineBytes;

lLineBytes = WIDTHBYTES(lWidth \* 8);

//给前景赋值

int FOREGROUND = 0;

if (color == 0) FOREGROUND = 0;//黑

else FOREGROUND = 255;//白

//邻域相对坐标

int dx[] = { -1, 1, 0, 0 , -1, -1, 1, 1};

int dy[] = { 0, 0, -1, 1 , -1, 1 ,1, -1};

int NEIGHBORNUM = 0;

if (connection == 0) {

NEIGHBORNUM = 8;

}

else {

NEIGHBORNUM = 4;

}

// 创建一个标记图像的数组并初始化为全零

std::vector<std::vector<bool>> visited = createVisitedImage(lWidth, lHeight);

// 计算图像中的物体数量

for (int y = 0; y < lHeight; y++) {

for (int x = 0; x < lWidth; x++) {

lpSrc = (unsigned char \*)lpDIBBits + lLineBytes \* (lHeight - 1 - y) + x;

// 如果是前景像素且未被访问过，则表示找到了一个新的物体

if (\*lpSrc == FOREGROUND && !isVisited(visited, x, y)) {

count++;

int area = 0;

// 使用广度优先搜索来标记与当前像素相连的所有前景像素

std::queue<std::pair<int, int>> q;

q.push(std::make\_pair(x, y));

markVisited(visited, x, y);

while (!q.empty()) {

area++;

int currentX = q.front().first;

int currentY = q.front().second;

q.pop();

// 检查相邻的前景像素

for (int i = 0; i < NEIGHBORNUM; i++) {

int newX = currentX + dx[i];

int newY = currentY + dy[i];

// 检查像素是否在图像范围内且未被访问过

lpNew = (unsigned char\*)lpDIBBits + lLineBytes \* (lHeight - 1 - newY) + newX;

if (newX >= 0 && newX < lWidth && newY >= 0 && newY < lHeight && \*lpNew == FOREGROUND && !isVisited(visited, newX, newY)) {

q.push(std::make\_pair(newX, newY));

markVisited(visited, newX, newY);

}

}

}

//太小的物体不计入

if (area < 5) {

count--;

}

}

}

}

return count;

}

* 1. 计算物体几何特征

ObjectProperties WINAPI calculateObjectProperties(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight, int connection, int color, unsigned char\* lpDstImageData) {

ObjectProperties properties = {0, 0, 0, 0, 0, 0, 0, 0};

int count = 0;

// 指向源像素的指针

unsigned char\* lpSrc;

unsigned char\* lpNew;

unsigned char\* lpDst;

// 图像每行的字节数

LONG lLineBytes;

lLineBytes = WIDTHBYTES(lWidth \* 8);

//给前景赋值

int FOREGROUND = 0;

if (color == 0) FOREGROUND = 0;//黑

else FOREGROUND = 255;//白

//邻域相对坐标

int dx[] = { -1, 1, 0, 0 , -1, -1, 1, 1 };

int dy[] = { 0, 0, -1, 1 , -1, 1 ,1, -1 };

int NEIGHBORNUM = 0;

if (connection == 0) {

NEIGHBORNUM = 8;

}

else {

NEIGHBORNUM = 4;

}

// 创建一个标记图像的数组并初始化为全零

std::vector<std::vector<bool>> visited = createVisitedImage(lWidth, lHeight);

for (int y = 0; y < lHeight; y++) {

for (int x = 0; x < lWidth; x++) {

lpSrc = (unsigned char\*)lpDIBBits + lLineBytes \* (lHeight - 1 - y) + x;

// 如果是前景像素且未被访问过，则表示找到了一个新的物体

if (\*lpSrc == FOREGROUND && !isVisited(visited, x, y)) {

// 使用广度优先搜索来计算物体的属性

int startX = x;

int startY = y;

int area = 0;

int minX = lWidth - 1;

int maxX = 0;

int minY = lHeight - 1;

int maxY = 0;

std::queue<std::pair<int, int>> q;

q.push(std::make\_pair(x, y));

markVisited(visited, x, y);

while (!q.empty()) {

int currentX = q.front().first;

int currentY = q.front().second;

q.pop();

// 更新物体的边界框

minX = min(minX, currentX);

maxX = max(maxX, currentX);

minY = min(minY, currentY);

maxY = max(maxY, currentY);

// 更新物体的面积

area++;

// 检查相邻的前景像素

for (int i = 0; i < NEIGHBORNUM; i++) {

int newX = currentX + dx[i];

int newY = currentY + dy[i];

// 检查像素是否在图像范围内且未被访问过

lpNew = (unsigned char\*)lpDIBBits + lLineBytes \* (lHeight - 1 - newY) + newX;

if (newX >= 0 && newX < lWidth && newY >= 0 && newY < lHeight && \*lpNew == FOREGROUND && !isVisited(visited, newX, newY)) {

q.push(std::make\_pair(newX, newY));

markVisited(visited, newX, newY);

lpDst = (unsigned char\*)lpDstImageData + (WIDTHBYTES(lWidth \* 24) \* (lHeight - 1 - newY)) + newX \* 3;

lpDst[2] = 0;

lpDst[1] = 100;

lpDst[0] = 100;

}

}

}

// 更新最大物体的属性

if (area > properties.area) {

properties.area = area;

//矩形度和重心坐标

properties.startX = startX;

properties.startY = startY;

int centerX = (minX + maxX) / 2;

int centerY = (minY + maxY) / 2;

properties.centerX = centerX;

properties.centerY = centerY;

double rectangularity = area \* 1.0 / ((maxX - minX + 1) \* (maxY - minY + 1));

properties.rectangularity = rectangularity;

}

}

}

}

//边界追踪求周长

properties.perimeter = traceBoundary(lpDIBBits, lWidth, lHeight, properties.startX, properties.startY, FOREGROUND, connection, lpDstImageData);

// 计算物体的圆形度

properties.circularity = (4 \* 3.14159 \* properties.area) / (properties.perimeter \* properties.perimeter);

return properties;

}

* 1. 边界追踪

int traceBoundary(const unsigned char\* image, int width, int height, int startX, int startY, int FOREGROUND, int connection, unsigned char\* lpDstImageData) {

// 指向像素的指针

unsigned char\* lpSrc;

unsigned char\* lpDst;

// 图像每行的字节数

LONG lLineBytes;

lLineBytes = WIDTHBYTES( width \* 8);

// 定义方向

int d4x[] = { 1, 0, -1, 0 };

int d4y[] = { 0, -1, 0, 1 };

int d8x[] = { 1, 1, 0, -1, -1,-1, 0, 1};

int d8y[] = { 0, -1,-1,-1, 0, 1, 1, 1};

// 记录起始点和当前点

int currentX = startX;

int currentY = startY;

startX = currentX;

startY = currentY;

// 初始化周长

int perimeter = 0;

// 开始边界跟踪

int direction = 0;

do {

int nextX = startX;

int nextY = startY;

if (connection == 4) {

nextX = currentX + d4x[direction];

nextY = currentY + d4y[direction];

}

else {

nextX = currentX + d8x[direction];

nextY = currentY + d8y[direction];

}

lpSrc = (unsigned char\*)image + lLineBytes \* (height - 1 - nextY) + nextX;

// 检查下一个点是否在图像范围内且为前景像素

if (nextX >= 0 && nextX < width && nextY >= 0 && nextY < height && \*lpSrc == FOREGROUND) {

// 更新当前点为下一个点

currentX = nextX;

currentY = nextY;

lpDst = (unsigned char\*)lpDstImageData + (WIDTHBYTES(width \* 24) \* (height - 1 - currentY)) + currentX \* 3;

lpDst[2] = 0;

lpDst[1] = 0;

lpDst[0] = 255;

// 更新周长

perimeter++;

if (connection == 4) {

// 重置方向为逆时针旋转90度

direction = (direction + 1) % 4;

}

else {

// 重置方向为逆时针旋转90度

direction = (direction + 1) % 8;

}

}

else {

if (connection == 4) {

// 顺时针旋转90度

direction = (direction - 1 + 4) % 4;

}

else {

// 顺时针旋转45度

direction = (direction - 1 + 8) % 8;

}

}

} while (currentX != startX || currentY != startY); // 回到起始点时停止

return perimeter;

}

* 1. Canny边缘检测

BOOL WINAPI cannyEdgeDetection(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight)

{

// 图像每行的字节数

LONG lLineBytes = WIDTHBYTES(lWidth \* 8);

// 高斯平滑

float coff = 1.0 / 16;

float fpArray[9] = { 1,2,1,2,4,2,1,2,1 };

Template(lpDIBBits, lWidth, lHeight, fpArray, coff, 0);

// 计算梯度幅值和方向

int\* gradientMagnitude = new int[lLineBytes \* lHeight];

int\* gradientDirection = new int[lLineBytes \* lHeight];

computeGradient(lpDIBBits, lWidth, lHeight, gradientMagnitude, gradientDirection);

// 非极大值抑制

unsigned char\* edges = new unsigned char[lLineBytes \* lHeight];

nonMaxSuppression(gradientMagnitude, gradientDirection, lWidth, lHeight, edges);

// 双阈值处理

int lowThreshold = 50;

int highThreshold = 150;

doubleThreshold(edges, lWidth, lHeight, lowThreshold, highThreshold);

// 边缘跟踪

edgeTracking(edges, lWidth, lHeight);

// 复制变换后的图像

memcpy(lpDIBBits, edges, lLineBytes \* lHeight);

// 清理内存

delete[] gradientMagnitude;

delete[] gradientDirection;

delete[] edges;

return TRUE;

}

void computeGradient(unsigned char\* image, int width, int height, int\* gradientMagnitude, int\* gradientDirection)

{

unsigned char\* lpSrc;

// 图像每行的字节数

LONG lLineBytes = WIDTHBYTES(width \* 8);

// 使用Sobel算子计算梯度

int sobelX[3][3] = { {-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1} };

int sobelY[3][3] = { {-1, -2, -1}, {0, 0, 0}, {1, 2, 1} };

for (int y = 1; y < height - 1; ++y)

{

for (int x = 1; x < width - 1; ++x)

{

int gx = 0;

int gy = 0;

// 计算x方向梯度

for (int i = -1; i <= 1; ++i)

{

for (int j = -1; j <= 1; ++j)

{

lpSrc = (unsigned char\*)image + lLineBytes \* (y + i) + (x + j);

int pixelValue = \*lpSrc;

gx += sobelX[i + 1][j + 1] \* pixelValue;

}

}

// 计算y方向梯度

for (int i = -1; i <= 1; ++i)

{

for (int j = -1; j <= 1; ++j)

{

lpSrc = (unsigned char\*)image + lLineBytes \* (y + i) + (x + j);

int pixelValue = \*lpSrc;

gy += sobelY[i + 1][j + 1] \* pixelValue;

}

}

// 计算梯度幅值和方向

gradientMagnitude[y \* lLineBytes + x] = sqrt(gx \* gx + gy \* gy);

gradientDirection[y \* lLineBytes + x] = atan2(gy, gx) \* 180 / PI;

}

}

}

// 非极大值抑制

void nonMaxSuppression(int\* gradientMagnitude, int\* gradientDirection, int width, int height, unsigned char\* edges)

{

LONG lLineBytes = WIDTHBYTES(width \* 8);

for (int y = 1; y < height - 1; ++y)

{

for (int x = 1; x < width - 1; ++x)

{

int angle = gradientDirection[y \* lLineBytes + x];

// 根据梯度方向进行非极大值抑制

int pixelValue = gradientMagnitude[y \* lLineBytes + x];

int n1, n2;

// 根据梯度方向确定抑制方向

if ((angle <= 22.5 && angle >= -22.5) || (angle >= 157.5 && angle <= -157.5))

{

n1 = gradientMagnitude[y \* lLineBytes + x + 1];

n2 = gradientMagnitude[y \* lLineBytes + x - 1];

}

else if ((angle > 22.5 && angle < 67.5) || (angle < -112.5 && angle > -157.5))

{

n1 = gradientMagnitude[(y + 1) \* lLineBytes + x + 1];

n2 = gradientMagnitude[(y - 1) \* lLineBytes + x - 1];

}

else if ((angle >= 67.5 && angle <= 112.5) || (angle <= -67.5 && angle >= -112.5))

{

n1 = gradientMagnitude[(y + 1) \* lLineBytes + x];

n2 = gradientMagnitude[(y - 1) \* lLineBytes + x];

}

else

{

n1 = gradientMagnitude[(y + 1) \* lLineBytes + x - 1];

n2 = gradientMagnitude[(y - 1) \* lLineBytes + x + 1];

}

// 进行非极大值抑制

if (pixelValue >= n1 && pixelValue >= n2)

{

edges[y \* lLineBytes + x] = (unsigned char)(pixelValue);

}

else

{

edges[y \* lLineBytes + x] = 0;

}

}

}

}

// 双阈值处理

void doubleThreshold(unsigned char\* edges, int width, int height, int lowThreshold, int highThreshold)

{

LONG lLineBytes = WIDTHBYTES(width \* 8);

for (int y = 0; y < height; ++y)

{

for (int x = 0; x < width; ++x)

{

unsigned char pixelValue = edges[y \* lLineBytes + x];

// 根据阈值进行分割

if (pixelValue >= highThreshold)

{

edges[y \* lLineBytes + x] = 255; // 强边缘

}

else if (pixelValue >= lowThreshold)

{

edges[y \* lLineBytes + x] = 128; // 弱边缘

}

else

{

edges[y \* lLineBytes + x] = 0; // 背景

}

}

}

}

// 边缘跟踪

void edgeTracking(unsigned char\* edges, int width, int height)

{

LONG lLineBytes = WIDTHBYTES(width \* 8);

unsigned char\* tempEdges = new unsigned char[lLineBytes \* height];

std::memcpy(tempEdges, edges, lLineBytes \* height \* sizeof(unsigned char));

for (int y = 1; y < height - 1; ++y)

{

for (int x = 1; x < width - 1; ++x)

{

unsigned char pixelValue = tempEdges[y \* width + x];

if (pixelValue == 128)

{

// 检查8邻域是否存在强边缘

if (tempEdges[(y - 1) \* lLineBytes + (x - 1)] == 255 || tempEdges[(y - 1) \* lLineBytes + x] == 255 ||

tempEdges[(y - 1) \* lLineBytes + (x + 1)] == 255 || tempEdges[y \* lLineBytes + (x - 1)] == 255 ||

tempEdges[y \* lLineBytes + (x + 1)] == 255 || tempEdges[(y + 1) \* lLineBytes + (x - 1)] == 255 ||

tempEdges[(y + 1) \* lLineBytes + x] == 255 || tempEdges[(y + 1) \* lLineBytes + (x + 1)] == 255)

{

edges[y \* lLineBytes + x] = 255;

}

else

{

edges[y \* lLineBytes + x] = 0;

}

}

}

}

delete[] tempEdges;

}

* 1. Usm锐化

BOOL WINAPI usmSharpening(unsigned char\* lpDIBBits, LONG lWidth, LONG lHeight, float amount, float threshold)

{

// 图像每行的字节数

LONG lLineBytes = WIDTHBYTES(lWidth \* 8);

// 创建临时缓冲区

unsigned char\* blurred = new unsigned char[lLineBytes \* lHeight];

memcpy(blurred, lpDIBBits, lLineBytes \* lHeight);

unsigned char\* mask = new unsigned char[lLineBytes \* lHeight];

unsigned char\* sharpened = new unsigned char[lLineBytes \* lHeight];

// 计算模糊图像

// 高斯平滑

float coff = 1.0 / 16;

float fpArray[9] = { 1,2,1,2,4,2,1,2,1 };

Template(blurred, lWidth, lHeight, fpArray, coff, 0);

// 计算掩膜和锐化图像

for (int y = 0; y < lHeight; y++)

{

for (int x = 0; x < lWidth; x++)

{

int diff = lpDIBBits[y \* lLineBytes + x] - blurred[y \* lLineBytes + x];

mask[y \* lLineBytes + x] = (abs(diff) > threshold) ? 255 : 0;

sharpened[y \* lLineBytes + x] = lpDIBBits[y \* lLineBytes + x] + amount \* mask[y \* lLineBytes + x] \* diff / 255;

//sharpened[y \* lLineBytes + x] = lpDIBBits[y \* lLineBytes + x] - amount \* mask[y \* lLineBytes + x] \* diff / 255可以获得模糊效果

}

}

// 更新原始图像

memcpy(lpDIBBits, sharpened, lLineBytes \* lHeight);

// 释放临时缓冲区

delete[] blurred;

delete[] mask;

delete[] sharpened;

return TRUE;

}