**Digital Media**

**第一次作业 实验报告**

# Question1

## 算法描述

（当前图片已经被转换为每个元素为一个像素点RGB值的二维矩阵）

1. 设置变量radius为0，记录当前圆形的半径
2. 当半径radius<图片对角线长度的1/2时，循环执行下面步骤：
   1. Radius = radius + 1
   2. 两层循环嵌套，按列遍历图片像素点
      1. 如果像素坐标与中心像素坐标的距离 < radius
         1. 使用临时变量pixel获得第二幅图片像素点的RGB
         2. 使用位运算将rgb变为r通道灰度值
            1. pixel &= 0xff0000
            2. pixel |= pixel >> 16
            3. pixel |= pixel >> 8
      2. 否则显示第一幅图片对应的r通道灰度值（计算灰度值的方法和i中相同）
   3. 调用repaint函数在面板上刷新图像

**伪代码如下**

WHILE radius < previousImage.height/2\*sqrt(2)

DO radius + 1

FOR (I = 0; I < previousImage.width; i++)

FOR (j = 0; I < previousImage.height; j++)

IF ((i-image1.width/2)^2 + (j- previousImage.height/2)^2 < radius ^ 2)

THEN pixel = subsequentImage.RGB(I,j)

pixel &= 0xff0000

pixel |= pixel >> 16

pixel |= pixel >> 8

displayImage.RGB(i,j) = pixel

ELSE pixel = previousImage.RGB(I,j)

pixel &= 0xff0000

pixel |= pixel >> 16

pixel |= pixel >> 8

displayImage.RGB(i,j) = pixel

Reload display panel

## 实验环境

1. **语言：**Java
2. **软件版本：**JDK1.8.0\_181
3. **使用类库：**java.awt.image.BufferedImage, java.awt.Graphics, javax.swing.JFrame, javax.swing.JPanel, java.io.\*, javax.imageio.ImageIO;

## 实现效果



**结果分析：**

由于是使用Java JPanel中repaint函数实现的动态效果，为了保持匀速应当每次刷新所用时间（即算法复杂度）一致可保持圆形匀速扩大。

# Question2

## 算法描述

1. 利用类库内函数将图像信息转化为一维数组data，每个像素信息使用4个数组元素分别代表RGBA
2. 统计图象颜色信息
   1. 创建colorsCount数组：数组元素为key-value，key = `r,g,b`（即用逗号分割的r,g,b三个通道的数值），value为key对应的rgb颜色在图像中出现的次数
   2. 以index作为下标遍历数组data，统计每种颜色出现的次数
      * 1. Key赋值为`data[index], data[index+1], data[index+2]`
        2. 如果colorsCount数组存在该元素，则colorsCount [key]++
        3. 否则colorsCount [key]赋值为1
        4. Index = index+4
   3. 创建colorsArray数组将colorsCount数组的键值对转化为对象数组，每个数组元素为一个对象，包含r, g, b. amount四个属性；创建tempRGB变量以临时记录colorsCount每个元素由键生成的r,g,b三色二进制数值信息
      1. 以key遍历colorsCount数组
      2. 将key值作为字符串使用’,’拆分，并存入tempRGB
      3. 在colorsArray的最后增加存储当前颜色信息的对象：
         1. R: tempRGB[0]对应的十进制
         2. G: tempRGB[1]对应的十进制
         3. B: tempRGB[2]对应的十进制
         4. Amount: colorsCount[key]对应的value
3. 使用颜色信息生成cube
   1. 确认矩阵颜色所属类型type（r/g/b）作为之后中值选择以及分割的依据

（考虑到颜色范围越广，即对应通道所有数值的极差越大，该颜色对应cube需要被更为精细地划分）

* 1. 统计R, G, B三个通道上数值的极差，遍历的同时统计cube上颜色总数total（用于分割完毕后计算每个cube的代表颜色）
     1. 以i遍历colorsArray数组
        1. Total = total + colorsArrayp[i].amount
        2. minR = min(minR, colorsArrayp[i].r). max和其他两个通道的算法同理
  2. 将极差最大者作为该cub的色彩类型
     1. 如果R通道极差 > G通道极差且R通道极差 > B通道极差，type赋值为R
     2. 如果G通道极差 > R通道极差且G通道极差 > B通道极差，type赋值为G
     3. 如果B通道极差 > R通道极差且B通道极差 > G通道极差，type赋值为B
  3. 将colorsArray作为转化为包含colorsArray, total, type三个属性的cube对象，且该对象作为cubes数组中的一个元素

1. 中值分割算法
   1. 选取cubes数组中total（颜色出现总数）最大的元素，并用maxTotal变量辅助记录当前最大元素，用index记录下一步要分割的cube所在cubes数组的下标

以i为下标编列cubes数组：如果cubes[i].total > maxTotal且cubes[i].colors.length（即cube内颜色的种类数）不为1，将index赋值为i

* 1. 取得中值
     1. 读取选定cube[index]的颜色种类，并将其排序
     2. 通过cubes[index].colors.length/2四舍五入获得中位数下标medianInex
  2. 进行分割

（使用cube1, cube2记录cube分割后的结果）

* + 1. 以i遍历cubes[index].colors数组（已排序）
       1. 如果颜色下标i< medianInex，cube1末尾加入元素cubes[index].colors
       2. 否则cube2末尾加入元素cubes[index].colors
    2. 对分割所得cube进行步骤3，加入type，total等信息
  1. 更新cubes数组
     1. 遍历cubes，将下标不为index的cubes数组元素存入新数组result，并将本次中值分割新生成的数组加入result末尾
     2. 使用result数组替代原cube数组
  2. 判断遍历结束
     1. 如果cubes.length（即当前划分出的颜色种类数） < 所期望的颜色种类数，则继续进行步骤4
     2. 否则当前cubes数组即位分割后的结果

1. 计算代表颜色

（使用加权平均数计算每个cube的R, G, B，并用长度为cubes.length的数组aveColors记录）

* 1. 以i遍历cube数组，用临时变量aveColor记录当前r, g, b的值\*出现频数
     1. 令aveColor = {r:0, g:0, b:0}
     2. 以j遍历cubes[i].colors[i]
        1. aveColor.r += cubes[i].colors[j].r\*cubes[i].colors[j].amount
        2. aveColor.g += cubes[i].colors[j].g\*cubes[i].colors[j].amount
        3. aveColor.b += cubes[i].colors[j].b\*cubes[i].colors[j].amount
     3. 令aveColors[i] = {r: round(aveColor.r / cubes[i].total), g: round(aveColor.g / cubes[i].total), b: round(aveColor.b / cubes[i].total) }

1. 将图片数据data使用对应cube的代表颜色替换
   1. 令pixelColors为存放key-value的数组，key为`r,g,b`字符串，value为该颜色所在cube的代表颜色对象
      1. 以i遍历cubes数组
         1. 以j遍历cubes[i].colors数组
            1. 令key = ` cubes[i].colors[j].r, cubes[i].colors[j].g, cubes[i].colors[j].b`
            2. pixelColors[key]为{r: aveColors[i].r, g: aveColors[i].g, b: aveColors[i].b}
      2. 以index遍历data数组
         1. 令data[index]为pixelColors[key].r
         2. 令data[index+1]为pixelColors[key].g
         3. 令data[index+2]为pixelColors[key].b
         4. Index = index+4

## 实验环境

1. **语言：**html, css, JavaScript
2. **软件：**Firefox浏览器（亲测Chrome由于安全问题不支持）

## 实验效果

1. 2色



1. 16色



1. 64色



1. 256色



**结果分析：**

2色和16色图像效果的确不尽人意，但是64色和256色的效果已经十分相似了，因而倘若不追求很高质量的效果又要求节省空间的情况下，64色（6bit）已经能满足基本视觉需求。

**红色层次多的原因：**

红色区域划分更细致的原因是由算法的特性决定的，在统计的时候由于红色出现的次数远高于其他颜色，而在划分cubes数组时总是选择cube内颜色出现的频数最多的进行划分，也就是红色，因而划分结果中会导致红色被划分了更多的分层。