

# Lab01 Report

---

## Task1 Image Dithering

---

### Uniform Random

这一部分简单地按部就班即可，在上面Threshold的代码基础上加上同一个随机噪声r即可。



### Blue Noise Random

这一部分只需要把给定的noise上的像素灰度值按照点点对应加到input上，然后归一化即可。



### Ordered

对于这个task，根据讲义中给出的内容，每个像素都被映射成了一个 $3 \times 3$ 的新像素，具体根据其灰度值决定。此处我才用了比较暴力的写法，即对 $3 \times 3$ 宫格的每一个位置都写一个判断，而每一个位置由于图例的不同，其亮与暗的阈值都不同，因此写了九个判断。看上去比较丑陋，但确实有效。



## Error Diffuse

对于这个task，一个比较直接的想法是在处理像素的过程中把error按照讲义中给出的比例直接加到input上。但是input是const，无法被修改，结合error diffuse需要实时更改input的内容的特点，我采取的方法是创建一份可更改的input的副本，然后不看input，直接把inputcopy作为input来处理，这样就可以在处理过程中实时更改其内容。这里有一个实现上的细节，就是判断一个像素左边的像素是否越界，不能使用 $x-1 \geq 0$ ，因为x是unsigned， $x-1$ 就会变为INT\_MAX，此时判断恒成立，起不到判断是否越界的效果。



## Task2 Image Filtering

### Blur

这一部分相当于是用一个元素都是 $1/9$ 的三阶方阵的卷积核来对这张照片做卷积，标准的做法是超出边界的置零，但在这里由于超出边界的部分处理起来有些麻烦，所以换了一个差不太多的方法，就是统计一下 $3 \times 3$ 的范围内有多少合法格子，然后对这些格子的灰度值做平均。当像素点在图像中间时，这样的做法和标准做法完全一致。从结果上可以看到，它也确实做到了模糊图像的结果。

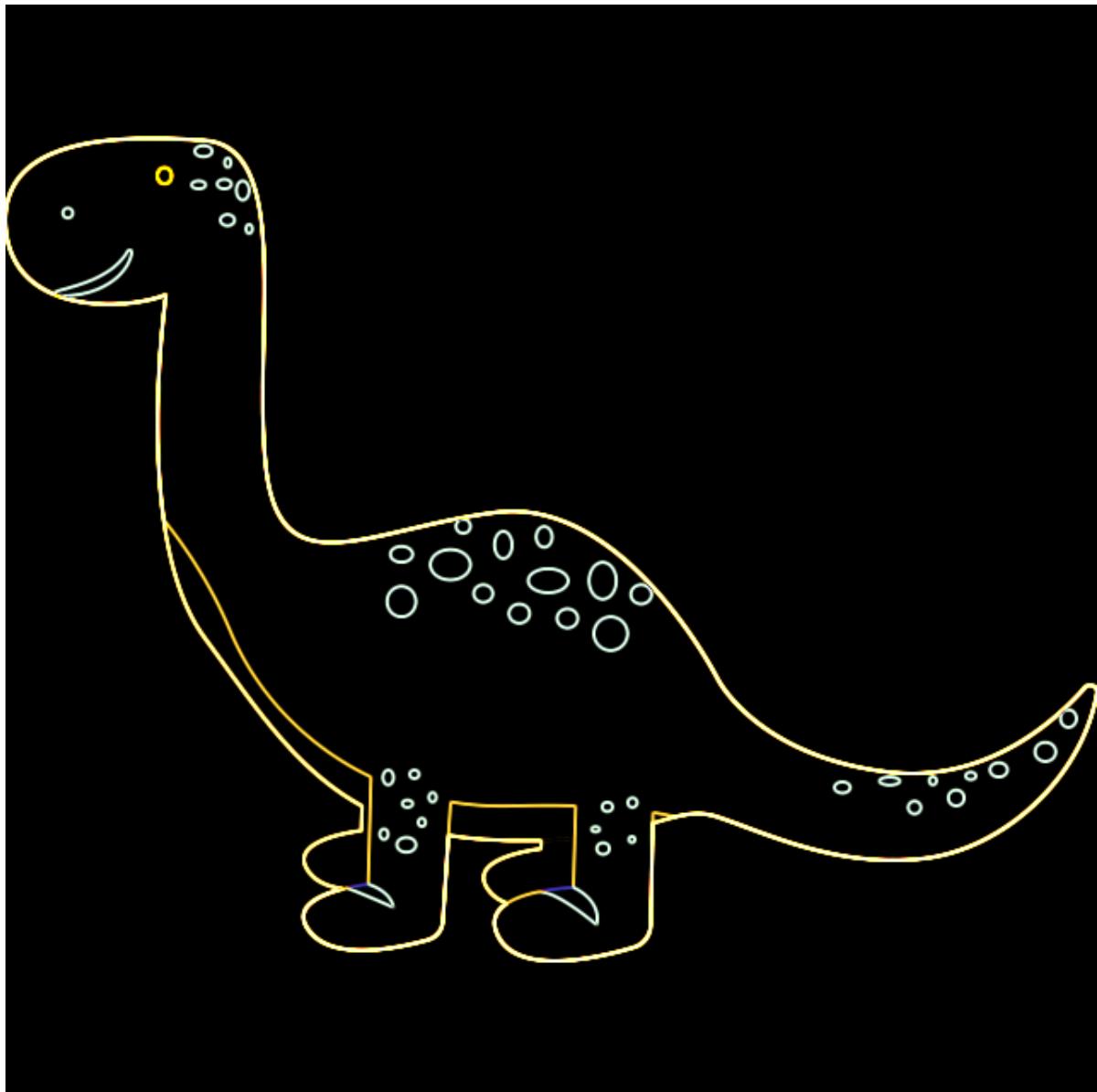


### Edge

这一部分相当于就是先求 $G_x$ 和 $G_y$ 两个近似偏导数，然后将两者通过平方和加起来，得到最终每个像素与外界的偏差，得到边缘。实现上，代表 $G_x$ 的矩阵可以通过一个小技巧来缩减代码量，也就是对于位置为 $(i, j)$ 的元素， $G_x[i, j]$ 实际上就是

```
(j == 0 ? 0 : (j > 0 ? 1 : -1)) * (i == 0 ? 2 : 1)
```

$G_y$ 也是同理，这样就可以很方便地表示矩阵的元素以及他们的点乘。



### Task3 Image Inpainting

注意到在laplace方程迭代时本来应该做这一计算

$$g(x, y) \approx \frac{1}{4}(g(x + 1, y) + g(x - 1, y) + g(x, y + 1) + g(x, y - 1) - \Delta f(x, y))$$

但是代码中给出的公式却是

$$g(x, y) = \frac{1}{4}(g(x + 1, y) + g(x - 1, y) + g(x, y + 1) + g(x, y - 1))$$

这意味着在初始化 $g$ 的时候就应该把前景的像素考虑进去。因为原本的公式中减掉了前景的像素，所以我们在实现的时候也应该预先把前景的像素减去。于是 $g$ 是一个融合的结果，因此边界条件就应该设置为

```

for (std::size_t y = 0; y < height; ++y) {
    g[y*width]=inputBack.At(offset.x,offset.y+y)-inputFront.At(0,y);
    g[y*width+width-1]=inputBack.At(offset.x+width-1,offset.y+y)-
    inputFront.At(width-1,y);
}
for (std::size_t x = 0; x < width; ++x) {
    g[x]=inputBack.At(offset.x+x,offset.y)-inputFront.At(x,0);
    g[(height-1)*width+x]=inputBack.At(offset.x+x,offset.y+height-1)-
    inputFront.At(x,height-1);
}

```

这样就得到了我们想要的抠图结果了。



## Task4 Line Drawing

这一部分如果在了解了Bresenham's Algorithm的情况下，其实相比斜率 $0 < k < 1$ 的情况只需要做一些小修正就可以转化成任意情况。先看源码：

```

int dx=abs(p1.x-p0.x), dy=abs(p1.y-p0.y);
int sx=p0.x<p1.x?1:-1, sy=p0.y<p1.y?1:-1;
int err=dx-dy;
int cx=p0.x, cy=p0.y;
while(1){
    canvas.At(cx,cy)=color;
    if(cx==p1.x&&cy==p1.y) break;
    int err2=err*2;
    if(err2>-dy)

```

```

{
    err-=dy;
    cx+=sx;
}
else
{
    err+=dx;
    cy+=sy;
}
}

```

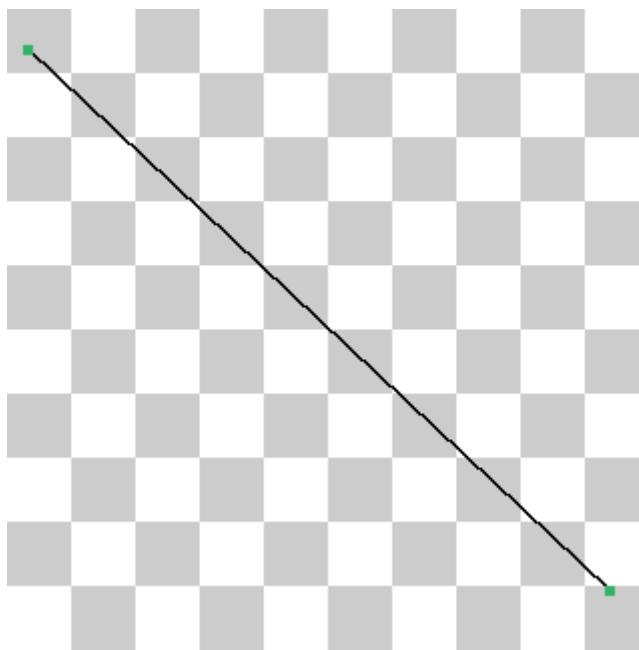
在这里设直线方程是

$$y = mx + c, m = \frac{\Delta y}{\Delta x}$$

定义

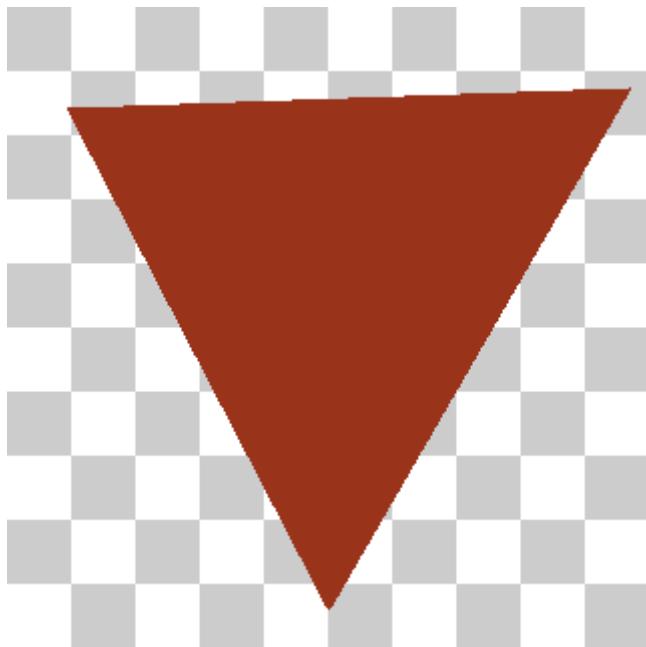
$$err = \Delta x - \Delta y, err2 = 2 \cdot err$$

表示目前x增加一步的情况下，当前点和直线的偏差量。如果 $err2 > -\Delta y$ ，那么说明x方向误差更大，需要在x方向前进一步。这和课件上给出的判断公式逻辑是一致的。反之那么在y方向上前进一步。注意此处的“前进一步”未必是+1，依据起点终点的位置，前进一步可能+1也可能-1，具体就是代码里面的sx与sy。



## Task5 Triangle Drawing

参考课件，这里采取的方法是利用平行于x轴的扫描线进行逐行填色。具体实现上，按照y的大小给三个点排序，然后分别给三角形的上半和下半填色即可。给每一行进行填色时，先按直线方程或者向量（反正就是数学工具）计算出x的左右边界，然后用一个for循环填色即可。这里比较容易崩溃的点就是如果有两个顶点的y值一样，那么会导致除以0的数学错误，因此在代码里面需要特判。



## Task6 Image Supersampling

在超采样的时候，需要先确定目标图的给定位置上的像素是由原图中的哪些像素来得到的。对于这个问题，我们需要输入和输出图的数据，随后由这个公式进行换算：

```
float fx=(x+(dx*1.0)/rate)*inw/outw;
float fy=(y+(dy*1.0)/rate)*inH/outH;
```

这是精确的像素位置，但是由于尺寸不一定是配好的，它大概率会得到一个非整数的结果，这个时候就需要做插值了。这里采用了线性插值，具体实现方法就是用一个运算结果的小数部分衡量它靠近整数像素的程度，然后用这个作为权重，和它的相邻像素进行加权平均做一次插值。对x和y都做一次这样的操作，因此实际上是双插值。具体实现如下：

```
int ix=int(fx),iy=int(fy);
float tx=fx-ix,ty=fy-iy;//weight
int ix1=std::min(ix+1,int(inw)-1);
int iy1=std::min(iy+1,int(inH)-1);
glm::vec3 c00=input.At(ix,iy),c10=input.At(ix1,iy);
glm::vec3 c01=input.At(ix,iy1),c11=input.At(ix1,iy1);
glm::vec3 c=c00*(1-tx)+c10*tx,c1=c01*(1-tx)+c11*tx;
glm::vec3 c=c0*(1-ty)+c1*ty;
colorSum+=c;
```

最后用colorSum除以rate的平方就得到了超采样的结果了。

观察到1倍到3倍的效果非常明显，



但是再往上就区别不大了，如4~6倍的图片如下：



可见超采样方法不需要特别高的rate。

---

## Task7 Bezier Curve

这里我不太懂为什么原本给出的代码是

```
return glm::vec2 {0, 0};
```

这里我把这一句注释了。实现方法是，该函数是一个递归函数，直到控制点+端点只剩一个的时候，这个点就是我们要画的点，直接return这个点即可。否则先计算出一组新的经过加权平均的点，然后

```
return CalculateBezierPoint(newpoints, t);
```

这也是课件中给出的标准bezier曲线的画法。

