Lab1-Report

王想 2100013146

Task 1: Image Dithering

1 Uniform Random

用两层 for 循环遍历图像上的每一个点, 对 *output* 的每一个像素点 (x, y) 进行如下操作:

取 *input* 图像上坐标为 (x, y) 的像素点的 RGB 值 *color*, 利用 std 的库函数产生一个位于 $[-0.5, 0.5]$ 区间均匀分布的随机浮点数 d。考虑到输入是灰度图,RGB 三个通道的值相同, 把上面得到的 *color* 的 RGB 三个通道的值分别加上同一个随机浮点数 d, 这样得到的新的 RGB 三个通道的值仍然相同。把得到的新的 RGB 值设置为 *output* 上像素点 (x, y) 的 RGB 值。

循环结束后再把上面得到的 *output* 作为参数传递给 $DitheringThreshold$ 函数进行处理, 最终得到下面的结果:

图 1: Uniform Random

2 Blue Noise Random

用两层 for 循环遍历图像上的每一个点, 对 *output* 的每一个像素点 (x, y) 进行如下操作:

取 *input* 图像上坐标为 (x, y) 的像素点的 RGB 值 *color_input*, 取 *noise* 图像上坐标为 (x, y) 的像素点的 RGB 值 *color_noise*, 把 *output* 图像上坐标为 (x, y) 的像素点的 RGB 值设置为上述二者的平均数 $(color\_input + color\_noise)/2$。这里不能简单相加, 因为相加的结果很可能大于 1, 导致最终输出图像上出现大片的白块。

循环结束后再把上面得到的 *output* 作为参数传递给 $DitheringThreshold$ 函数进行处理, 最终得到下面的结果:

图 2: BlueNoise

3 Ordered

Ordered 算法核心思路是使用 3x3 的有规律的黑白像素分布表示原图的一个灰度像素, 所以输出图像宽高大小必然会是输入的 3 倍, 先考虑建立输入图像的像素点与输出图像的像素块中心坐标之间的映射

$$(x, y) \to (3x + 1, 3y + 1),$$

那么输出图像上对应输入图像上坐标为 (x, y) 的像素点的 9 个像素点的坐标就可以利用中心点的坐标分别求出。将灰度范围 $[0, 1]$ 划分成 10 个小区间 $[i, i+0.1), i = 0, 0.1, 0.2, \ldots, 0.9$, 对应这十个区间设置相应的 3x3 黑白像素块的表示。

用两层 for 循环遍历图像上的每一个点, 对 *output* 的每一个像素点 (x, y) 进行如下操作:

取 *input* 图像上坐标为 (x, y) 的像素点的 RGB 值 *color*, 根据 *color* 的 RGB 取值范围, 把 *output* 上中心坐标为 $(3x + 1, 3y + 1)$ 的 3x3 黑白像素块设置成对应规则的像素块

循环结束后再把上面得到的 *output* 作为参数传递给 *DitheringThreshold* 函数进行处理, 最终得到下面的结果:

图 3: Ordered

4 Error Diffuse

开辟三个大小为 $(input.GetSizeY()+2)*2$ 的一维 float 数组, 滚动存放 *error* 的 RGB 值, 数组中每一个元素都初始化为 0。*error* 数组可以视作一个二维数组, 每一层的大小为 $dy = input.GetSizeY()+2$, 每一层的 $error[0]$ 和 $error[input.GetSizeY()+1]$ 用于防止越界, 便于统一处理, 以免每次进行是否越界的检查。每次处理完一行像素点后, 对应层的 *error* 数组内数据就没用了, 对这一层的每一个元素清零, 用来存储新的一行 error 数据。

在循环开始前设置一个计数器 $cnt = 0$, 用于 *error* 数组的读取与滚动更新。*cnt* 会在处理完一行像素点后进行加 1 并模 2 的操作, 于是对于每一个坐标 (x, y), $error[y+1+cnt*dy]$ 对应坐标 (x, y) 的 error 值, $error[y+1+(xnt+1)\%2*dy]$ 对应坐标 $(x+1, y)$ 的 error 值。

用两层 for 循环遍历图像上的每一个点, 对 *output* 的每一个像素点 (x, y) 进行如下操作:

取 $input$ 图像上坐标为 (x, y) 的像素点的 RGB 值 $color$, 取 $error[y + 1 + cnt * dy]$ 存储的 errorRGB 值, 将二者三个通道的值分别相加得到 $sumRGB$, 根据 $sumRGB$ 的三个通道的值设置 $output$ 图像上坐标为 (x, y) 的像素点的 RGB 值, 若大于 0.5 则设置为 1, 否则设置为 0。

最终得到下面的结果:

图 4: ErrorDiffuse

Task 2: Image Filtering

1 Blur

用两层 for 循环遍历图像上的每一个点, 对 *output* 的每一个像素点 (x, y) 进行如下操作:

取 *input* 图像上中心坐标为 (x, y) 的 3x3 像素块,（特别处理边界条件）, 将这 9 个像素点的 RGB 值取平均值作为 *output* 图像上坐标为 (x, y) 的像素点的 RGB 值。也就是这里使用的 $filter$ 是如下矩阵:

$$\begin{pmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{pmatrix}$$

最终得到下面的结果:

图 5: Blur

2 Edge

用两层 for 循环遍历图像上的每一个点, 对 *output* 的每一个像素点 (x, y) 进行如下操作:

取 $input$ 图像上中心坐标为 (x, y) 的 3x3 像素块,（特别处理边界条件）, 分别用两种 $filter$ 进行卷积, 得到纵、横两种 RGB 梯度, 由于这里得到的 RGB 梯度可能为负值, 对二者取绝对值得到两个梯度的绝对值, 取二者的最大值作为 $output$ 图像上坐标为 (x, y) 的像素点的 RGB 值。这里使用的两种 $filter$ 分别是如下两个矩阵:

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

最终得到下图所示的结果:

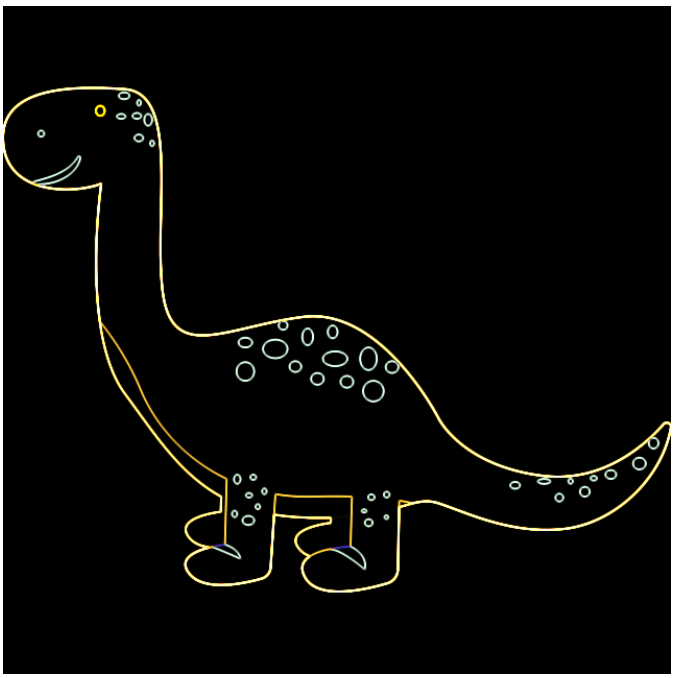

图 6: EdgeDetection

Task 3: Image Inpainting

对于这个函数, 最终的目标是在 $f|_{\partial\Omega} = f^*|_{\partial\Omega^*}$ 的条件下使得 $\int\int_\Omega \|\nabla f - g\|$ 达到最小。所以初始化边界时, 边界上的每一点 (x, y) 应该设置成 $inputBack$ 对应像素点的 RGB 值减去 $inputFront$ 对应像素点的 RGB 值。

在循环迭代之后得到下面的结果:

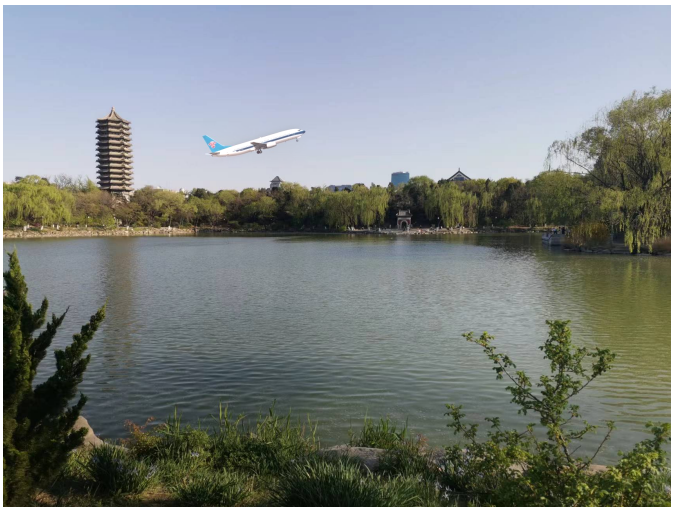

图 7: ImageInpainting

Task 4: Line Drawing

先根据 x 坐标的大小对 $p0, p1$ 两点进行排序, 得到 $p2, p3$, 使得 $p2.x <= p3.x$, 这样一来斜率 $slope$ 就只有四种情况,

$$case1 : 0 <= slope <= 1;$$

$$case2 : \quad slope > 1;$$

$$case3 : -1 <= slope < 0;$$

$$case4 : \quad slope < -1;$$

利用 $Bresenham\ Line\ Algorithm$ 分别处理这四种情况。

最终得到下图的结果, 并且这种算法运行效率较高, 随着鼠标拖动控制点可以立即绘制相应的直线。

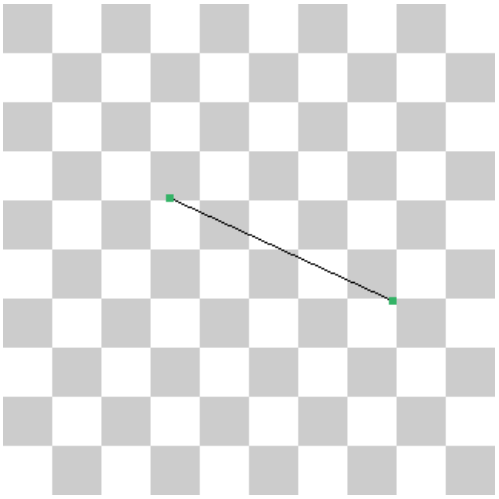

图 8: Line

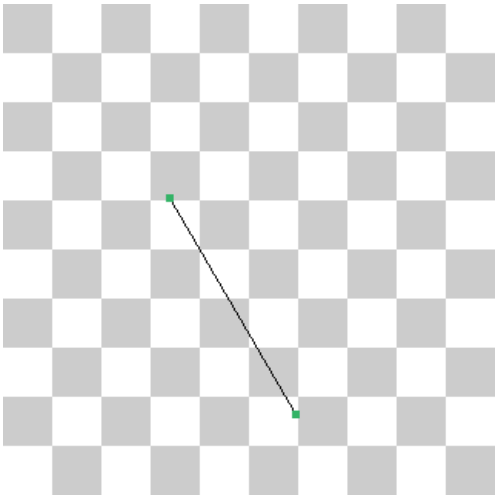

图 9: Line

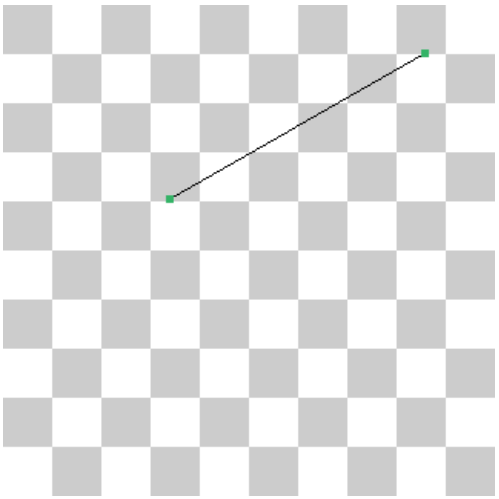

图 10: Line

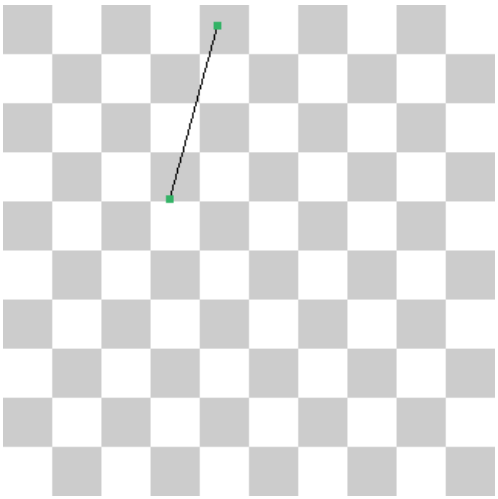

图 11: Line

Task 5: Triangle Drawing

先根据 y 坐标的大小对 $p0, p1, p2$ 三点进行排序, 得到 $p3, p4, p5$, 使得 $p3.y >= p4.y >= p5.y$, 利用 $Triangle\ Sweep - Line\ Algorithm$ 进行绘制, 以 y 为轴一步步扫描,x 每次增加 $slope$(即 dx/dy), 对满足 $p5.y <= y <= p3.y$ 的每一个 y, 对满足 $xLeft <= x <= xRight$ 的每一个 x,$Draw(x, y)$。

最终得到下图的结果, 并且这种算法运行效率较高, 随着鼠标拖动控制点可以立即绘制出相应的三角形。

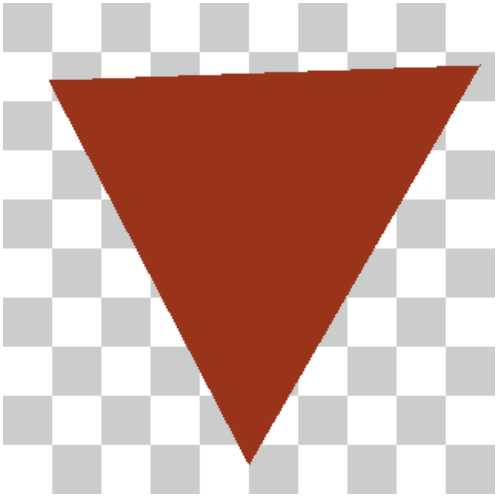

图 12: Triangle

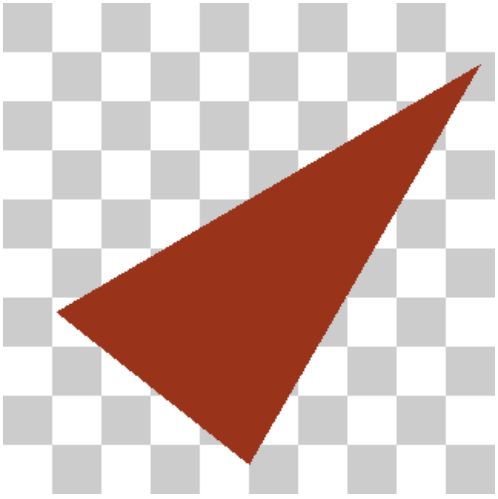

图 13: Triangle

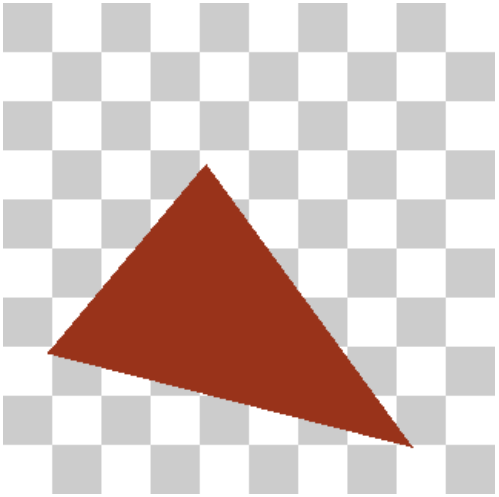

图 14: Triangle

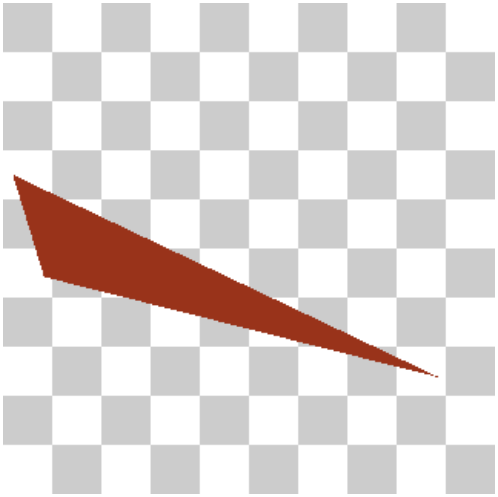

图 15: Triangle

Task 6: Image Supersampling

先确定缩小的比例

$$times_x = (double)input.GetSizeX()/output.GetSizeX(),$$

$$times_y = (double)input.GetSizeY()/output.GetSizeY().$$

然后建立从输出图像像素坐标到原始图像像素坐标的映射

$$(x, y) \to (x * times_x, y * times_y)$$

用两层 for 循环遍历图像上的每一个点, 对 $output$ 的每一个像素点 (x, y) 进行如下操作:

根据采样率 $rate$ 的大小随机选取 (tx, ty), $x * times_x <= tx < (x + 1) * times_x$, $y * times_y <= ty < (y + 1) * times_y$, 总计选取 $rate$ 个随机像素点, 取这些像素点的 RGB 平均值作为 $output(x, y)$ 的 RGB 值。

最终得到下面的结果:

图 16: rate=1

图 17: rate=4

图 18: rate=8

图 19: rate=12

Task 7: Bezier Curve

开辟两个长度为控制点数目（记为 len）的数组 xs, ys 分别用来记录每一轮的插值点的 x, y 坐标, $xs[i], ys[i]$ 初始化为控制点 i 的 x, y 坐标。

接下来每一轮循环都对这个两个数组进行更新, 记录新一轮的插值点的 x, y 坐标, 具体操作为 $for\ i = 0\ to\ len - 2$

$$xs[i] = (1 - t) * xs[i] + t * xs[i + 1]$$

$$ys[i] = (1 - t) * ys[i] + t * ys[i + 1]$$

由于每一轮循环之后得到的点都比上一轮少一个, 在 for 循环结束后 $len = len - 1$。重复上述步骤直到 $len == 1$, 也就是只剩下一个点, 这就是所需的 $BezierPoint$。

最终得到下面的结果:

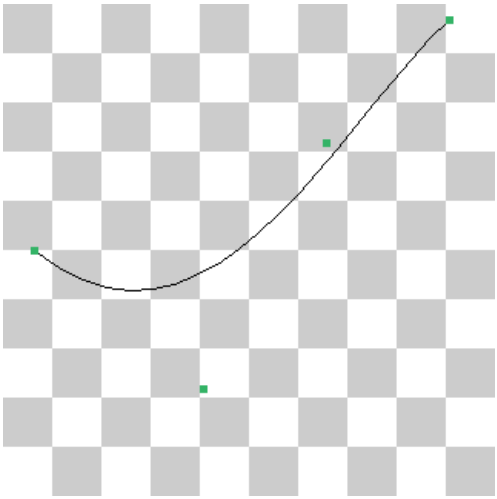

图 20: BezierCurve

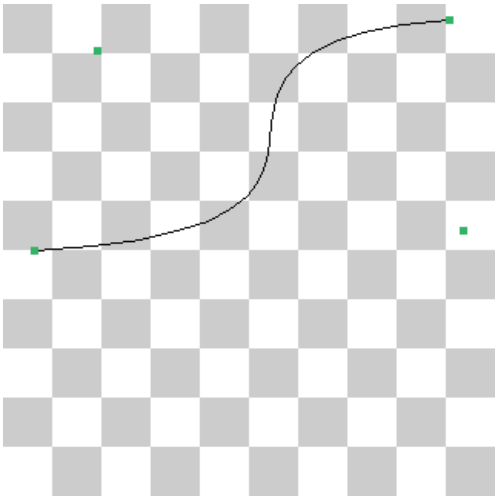

图 21: BezierCurve

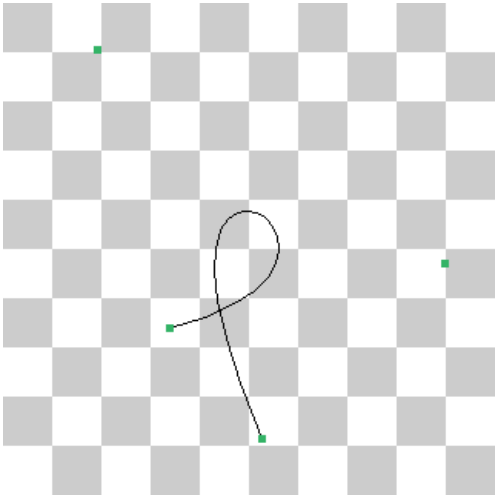

图 22: BezierCurve

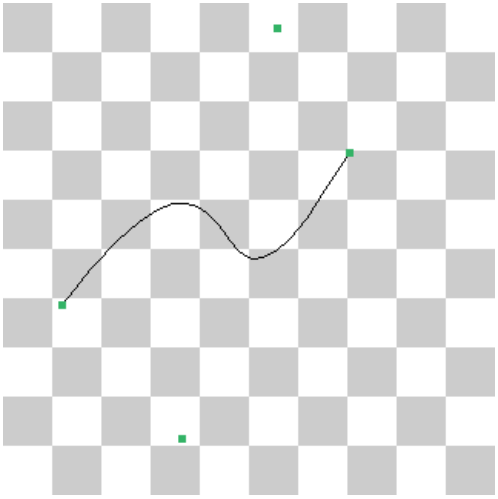

图 23: BezierCurve