可视计算与交互概论-Lab1

xTryer

2024/9/27

1 Task 1: Image Dithering

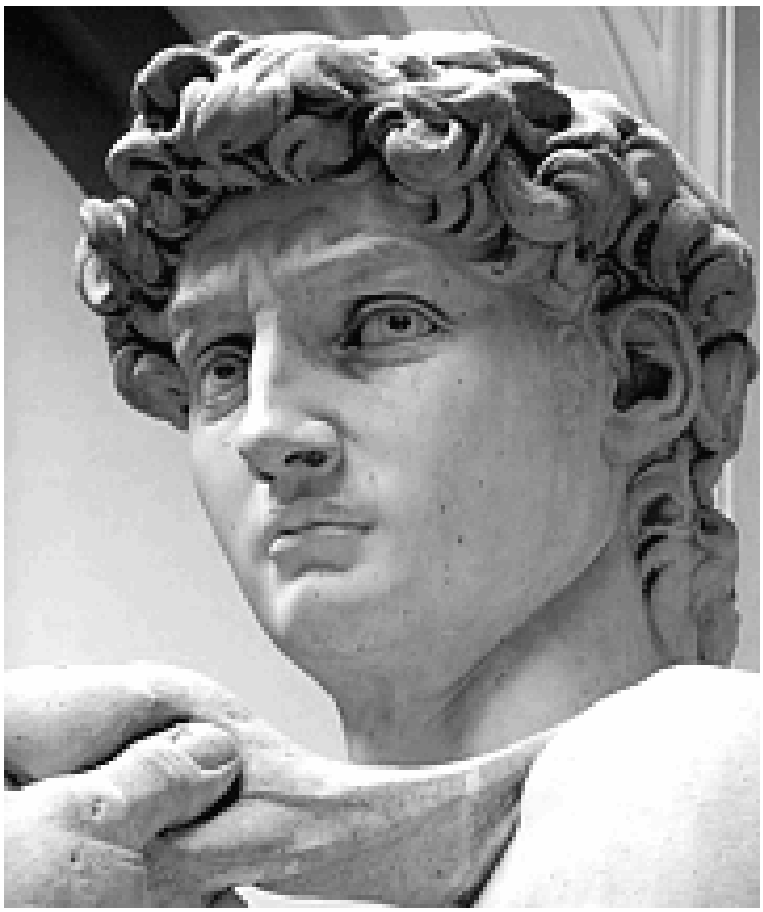


图 1: T1-original

1.1 Threshold

算法思路：遍历图像上每一个像素点，以 0.5 为阈值进行二值化操作。

```
void DitheringThreshold(ImageRGB & output,
                        ImageRGB const & input) {
for (std::size_t x = 0; x < input.GetSizeX(); ++x)
    for (std::size_t y = 0; y < input.GetSizeY(); ++y) {
        glm::vec3 color = input.At(x, y);
        output.At(x, y) = {
            color.r > 0.5 ? 1 : 0,
            color.g > 0.5 ? 1 : 0,
```

```
                color.b > 0.5 ? 1 : 0,
            };
        }
    }
```

图 2: T1.1-Threshold

1.2 Uniform_random

算法思路：生成一个随机数生成器，在-0.5 到 0.5 之间均匀生成随机数，遍历图像上每一个像素点，给每个像素点加上一个随机的扰动，再以 0.5 为阈值进行二值化操作。

```
    void DitheringRandomUniform(
        ImageRGB &        output,
        ImageRGB const & input) {
        // your code here:
        std::random_device my_rd;
        std::mt19937 my_gen(my_rd());
        std::uniform_real_distribution<float> dis(-0.5, 0.5);
        for (std::size_t x = 0; x < input.GetSizeX(); ++x)
            for (std::size_t y = 0; y < input.GetSizeY(); ++y) {
                glm::vec3 color = input.At(x, y);
```

```
                float my_rand_noise = dis(my_gen);
                output.At(x, y) = {
                    color.r+my_rand_noise > 0.5 ? 1 : 0,
                    color.g+my_rand_noise > 0.5 ? 1 : 0,
                    color.b+my_rand_noise > 0.5 ? 1 : 0,
                };
            }
    }
```

图 3: T1.2-Uniform_random

1.3 BlueNoise_random

算法思路：遍历图像上每一个像素点，给每个像素点加上一个随机的蓝噪音，由于相当于叠了两个像素，以 1.0 为阈值进行二值化操作。

```
        void DitheringRandomBlueNoise(
            ImageRGB &       output,
            ImageRGB const & input,
            ImageRGB const & noise) {
            // your code here:
            for (std::size_t x = 0; x < input.GetSizeX(); ++x)
```

```
            for (std::size_t y = 0; y < input.GetSizeY(); ++y) {
                glm::vec3 color = input.At(x, y);
                glm::vec3 noise_color = noise.At(x, y);
                float blue_noise = noise_color.r;
                output.At(x, y) = {
                    color.r + blue_noise > 1 ? 1 : 0,
                    color.g + blue_noise > 1 ? 1 : 0,
                    color.b + blue_noise > 1 ? 1 : 0,
                };
            }
        }
```

图 4: T1.3-BlueNoise_random

1.4 Ordered

算法思路：先确定一个 3x3 的抖动矩阵，然后根据原图像素的亮度来分别对新图对应的 9 个位置的像素进行二值化赋值, 从而达到一种灰度渐变的效果。

```
        void DitheringOrdered(
            ImageRGB & output,
            ImageRGB const & input) {
```

```
        // your code here:
        int shake_lst[9] = { 6, 8, 4, 1, 0, 3, 5, 2, 7 };
        for (int i = 0; i < input.GetSizeX(); i++)
        {
            for (int j = 0; j < input.GetSizeY(); j++)
            {
                for (int r_x = 0; r_x < 3; r_x++)
                {
                    for (int r_y = 0; r_y < 3; r_y++)
                    {
                        int x_index = i * 3 + r_x;
                        int y_index = j * 3 + r_y;
                        float check_ = (glm::vec3(input.At(i, j))).r * 9;
                        output.At(x_index, y_index) = (check_ > (float) shake_lst[r_x * 3
+ r_y]) ? glm::vec3({ 1, 1, 1 }) : glm::vec3({ 0, 0, 0 });
                    }
                }
            }
        }
    }
```

图 5: T1.4-Ordered

1.5 Error_Diffuse

算法思路：先创建一个足够大的二维数组用于存储传送的误差信息，自上而下，从左向右地遍历整张图片上的像素，以 0.5 为阈值二值化处理像素，将产生地误差按要求传送给误差矩阵 out_record, 在处理像素时需要用到对应误差矩阵的值加上原像素的值。考虑到我们只需要同时存储两行的信息，因此只开了 10000*2 的数组，节省了空间。

```
        void DitheringErrorDiffuse(
            ImageRGB &       output,
            ImageRGB const & input) {
            // your code here:

            float out_record[10000][2];
            memset(out_record, 0, sizeof(out_record));

            int line_index = 0;
            for (int j = 0; j < input.GetSizeY(); j++) {
                for (int i = 0; i < input.GetSizeX(); i++) {

                    float output_former = out_record[i][line_index];
                    out_record[i][line_index] = 0;

                    float dither_ans = ((glm::vec3(input.At(i, j)).r+output_former) > 0.5) ?
    1.0 : 0.0;
                    output.At(i, j)    = glm::vec3({ dither_ans, dither_ans, dither_ans });

                    float dither_noise = glm::vec3(input.At(i, j)).r+output_former -
    dither_ans;

                    if (i+1<input.GetSizeX()) {
                        float tmp_          = dither_noise * (float) 7 / (float) 16;
                        out_record[i + 1][line_index] += tmp_;

                        if (j + 1 < input.GetSizeY())
                        {
                            float tmp_2             =  dither_noise * (float) 1 / (float) 16;
                            out_record[i][1 - line_index] += tmp_2;
                        }
                     }

                    if (j + 1 < input.GetSizeY())
                    {
                         float tmp_          =  dither_noise * (float) 5 / (float) 16;
                        out_record[i][1 - line_index]+=tmp_;
                         if (i != 0)
                         {
                             float tmp_2             =  dither_noise * (float) 3 / (float) 16;
                             out_record[i - 1][1 - line_index] += tmp_2;
                         }
                    }

                }
```

```
44                 line_index = 1 - line_index;
45             }
46         }
```

图 6: T1.5-Error_Diffuse

2 Task 2: Image Filtering

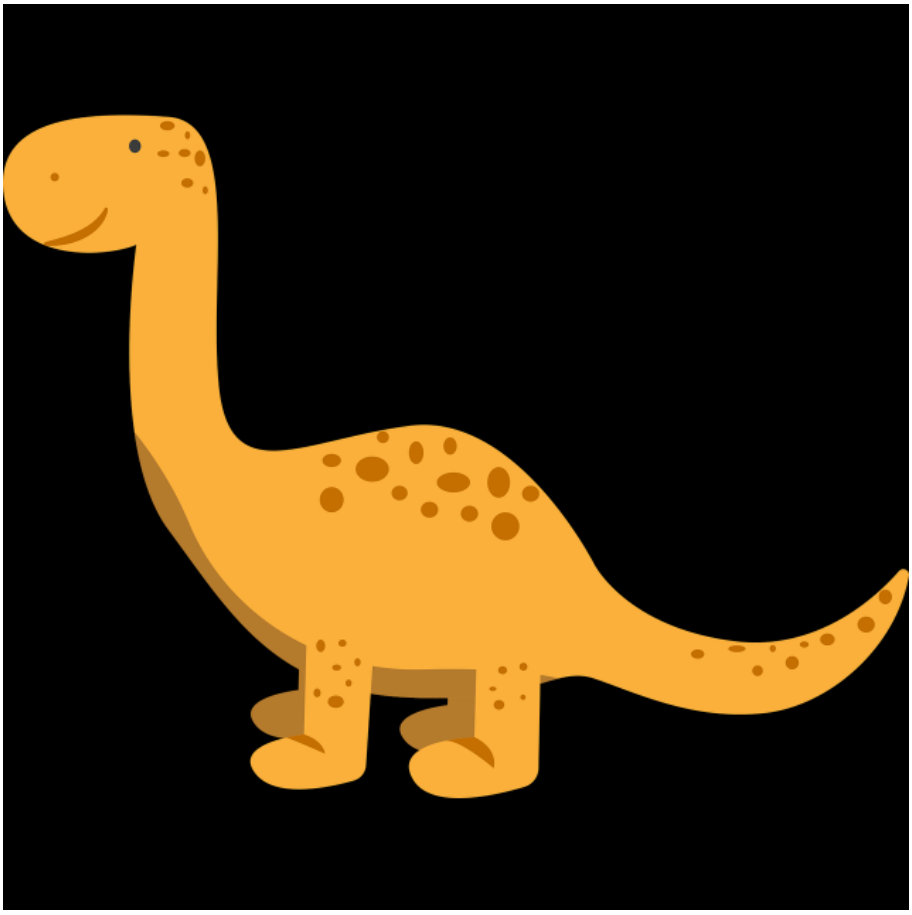


图 7: T2-original

2.1 Blur

算法思路：遍历图像上每一个像素点，以 3*3 的 kernel 进行卷积操作，对目标像素点与周围一圈的像素取加权平均，记录参与加权的像素个数，最后对三个通道分别求一个平均值代表处理后的像素。

```
1        void Blur(
2            ImageRGB &       output,
3            ImageRGB const & input) {
4            // your code here:
5            for (int i = 0; i < output.GetSizeX(); i++)
6            {
7                for (int j = 0; j < output.GetSizeY(); j++)
8                {
9                    float tmp_cnt = 0;
10                   float tar_pixel_color_r = 0;
11                   float tar_pixel_color_g = 0;
12                   float tar_pixel_color_b = 0;
13                   for (int i_x = -1; i_x <= 1; i_x++)
14                   {
15                       for (int j_y = -1; j_y <= 1; j_y++)
16                       {
17                           int tar_x = i + i_x;
18                           int tar_y = j + j_y;
19                           if (tar_x != -1 && tar_x != output.GetSizeX() && tar_y != -1 &&
    tar_y != output.GetSizeY())
```

```
                {
                    tmp_cnt++;
                    tar_pixel_color_r += (input.At(tar_x, tar_y).r);
                    tar_pixel_color_g += (input.At(tar_x, tar_y).g);
                    tar_pixel_color_b += (input.At(tar_x, tar_y).b);
                }
            }
        }
        output.At(i, j)   = { tar_pixel_color_r / tmp_cnt, tar_pixel_color_g / tmp_cnt, tar_pixel_color_b / tmp_cnt };
    }
  }
}
```

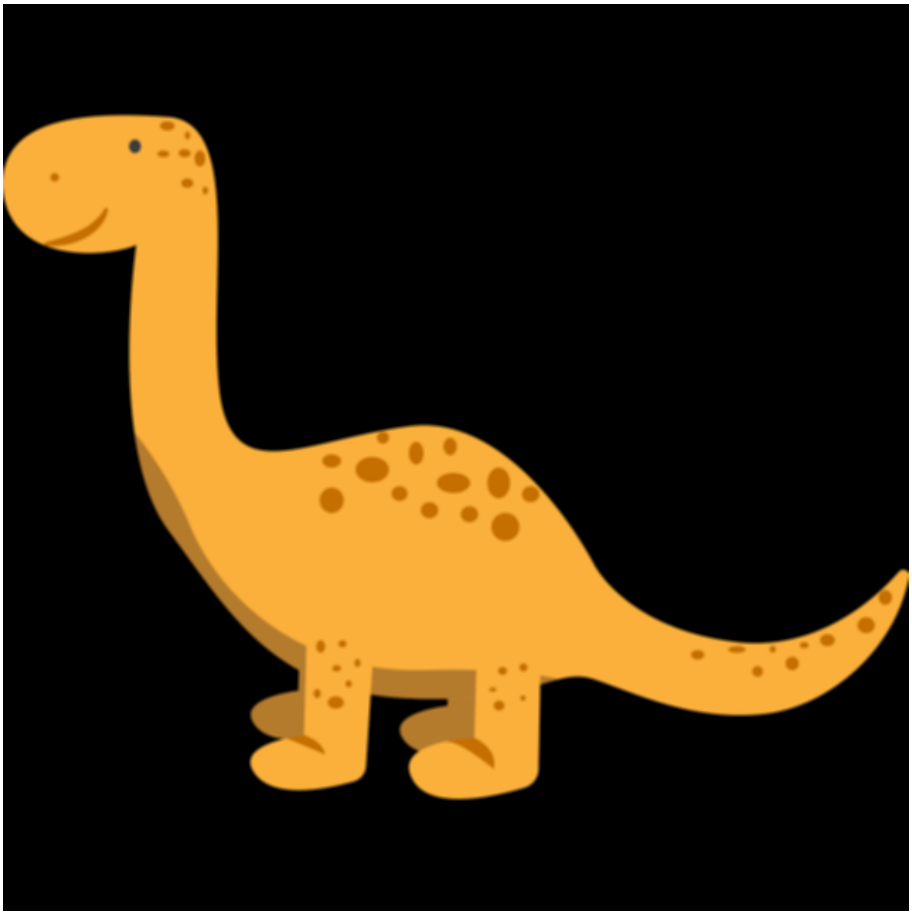


图 8: T2.1-Blur

2.2 Edge

算法思路：先设置水平方向上的 x_kernel 和竖直方向上的 y_kernel 分别用于计算原图像在目标像素点的水平梯度与竖直梯度，然后依次遍历图像上的像素点，对每个像素点以 x_kernel 与 y_kernel 进行卷积操作求对应的水平与竖直梯度，最后以二者合成的梯度的模长作为最终像素值。

```
void Edge(
ImageRGB &        output,
ImageRGB const & input) {
// your code here:
```

```
        float x_kernel[9] = {-1, 0, 1,-2, 0, 2, -1,  0,  1 };
        float y_kernel[9] = { 1, 2, 1, 0, 0, 0, -1, -2, -1 };
        for (int i = 0; i < output.GetSizeX(); i++) {
            for (int j = 0; j < output.GetSizeY(); j++) {
                float tmp_cntx           = 0;
                float tar_pixel_color_rx = 0;
                float tar_pixel_color_gx = 0;
                float tar_pixel_color_bx = 0;

                float tmp_cnty           = 0;
                float tar_pixel_color_ry = 0;
                float tar_pixel_color_gy = 0;
                float tar_pixel_color_by = 0;

                for (int i_x = -1; i_x <= 1; i_x++) {
                    for (int j_y = -1; j_y <= 1; j_y++) {
                        int tar_x = i + i_x;
                        int tar_y = j + j_y;
                        int tar_index = (j_y + 1) * 3 + (i_x + 1);
                        if (tar_x != -1 && tar_x != output.GetSizeX() && tar_y != -1 && tar_y
!= output.GetSizeY()) {
                            tmp_cntx++;
                            tar_pixel_color_rx += (input.At(tar_x, tar_y).r)*x_kernel[
tar_index];
                            tar_pixel_color_gx += (input.At(tar_x, tar_y).g) * x_kernel[
tar_index];
                            tar_pixel_color_bx += (input.At(tar_x, tar_y).b) * x_kernel[
tar_index];
                            tmp_cnty++;
                            tar_pixel_color_ry += (input.At(tar_x, tar_y).r) * y_kernel[
tar_index];
                            tar_pixel_color_gy += (input.At(tar_x, tar_y).g) * y_kernel[
tar_index];
                            tar_pixel_color_by += (input.At(tar_x, tar_y).b) * y_kernel[
tar_index];
                        }
                    }
                }
                output.At(i, j) = { sqrt(pow(tar_pixel_color_rx, 2) + pow(tar_pixel_color_ry,
2)), sqrt(pow(tar_pixel_color_gx, 2) + pow(tar_pixel_color_gy, 2)),sqrt(pow(
tar_pixel_color_bx, 2) + pow(tar_pixel_color_by, 2)) };
            }
        }
    }
```

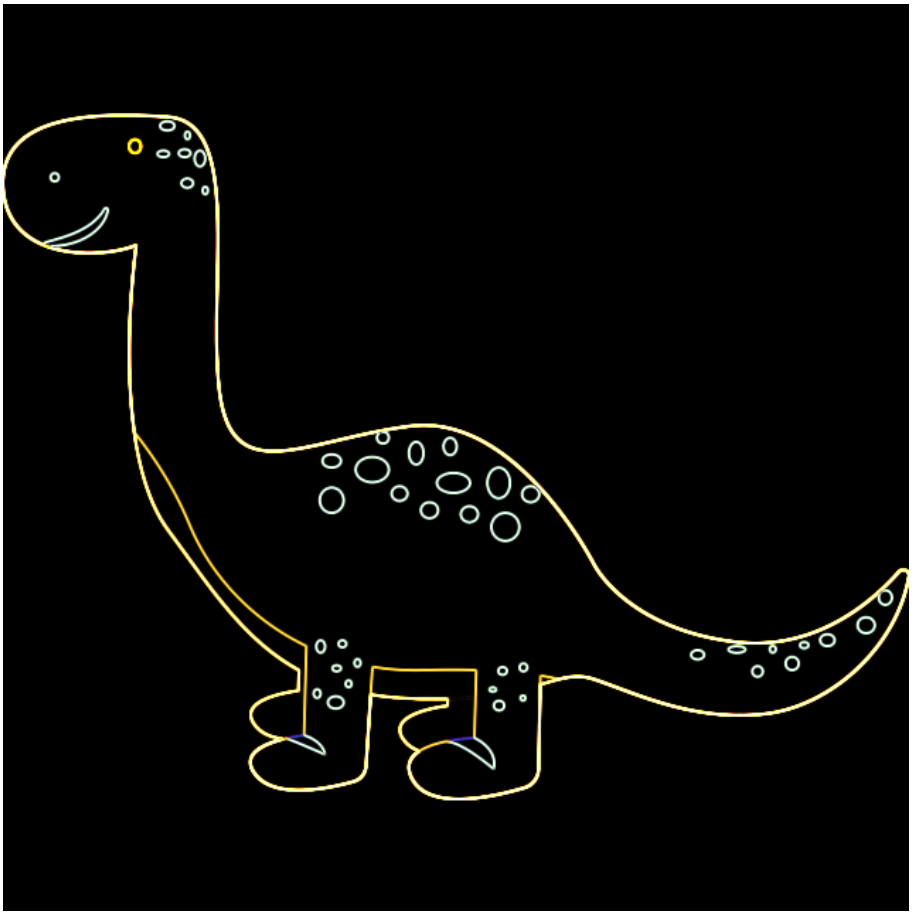


图 9: T2.2-edge

3 Task 3: Image Inpainting

图 10: T3-original

算法思路：我们的目标是使用 Poisson Editing 实现两张图片的无缝融合。我们需要确定 Laplace 矩阵的边界数值，然后再通过 Jacobi 迭代法求解。由于我们需要保持边界值的稳定性，因此我们将边界像素值直接取为 Back 图与 Front 图的差值，这样在最后生成图像的时候就能做到贴图与底图的适应。然后进行 Jacobi

迭代，实现内部参数的求解，最后输出得到无缝融合的新图片。

```
    void Inpainting(
        ImageRGB &          output,
        ImageRGB const &    inputBack,
        ImageRGB const &    inputFront,
        const glm::ivec2 & offset) {
        output              = inputBack;
        std::size_t width   = inputFront.GetSizeX();
        std::size_t height  = inputFront.GetSizeY();
        glm::vec3 * g       = new glm::vec3[width * height];
        memset(g, 0, sizeof(glm::vec3) * width * height);
        // set boundary condition
        for (std::size_t y = 0; y < height; ++y) {
            // set boundary for (0, y), your code: g[y * width] = ?
            g[y * width] = glm::vec3(inputFront.At(0, y)) * glm::vec3(-1) + glm::vec3(
inputBack.At(offset.x, offset.y+y));

            // set boundary for (width - 1, y), your code: g[y * width + width - 1] = ?
            g[y * width + width - 1] = glm::vec3(inputFront.At(width - 1, y)) * glm::vec3
(-1) + glm::vec3(inputBack.At(offset.x + width - 1, y + offset.y));
        }
        for (std::size_t x = 0; x < width; ++x) {

            // set boundary for (x, 0), your code: g[x] = ?
            g[x] = glm::vec3(inputFront.At(x, 0)) * glm::vec3(-1) + glm::vec3(inputBack.At
(offset.x+x, offset.y));
            // set boundary for (x, height - 1), your code: g[(height - 1) * width + x] =
?
            g[x + (height - 1) * width] = glm::vec3(inputFront.At(x, height - 1)) * glm::
vec3(-1) + glm::vec3(inputBack.At(offset.x+x, offset.y + height - 1));
        }


        // Jacobi iteration, solve Ag = b
        for (int iter = 0; iter < 8000; ++iter) {
            for (std::size_t y = 1; y < height - 1; ++y)
                for (std::size_t x = 1; x < width - 1; ++x) {
                    g[y * width + x] = (g[(y - 1) * width + x] + g[(y + 1) * width + x] +
g[y * width + x - 1] + g[y * width + x + 1]);
                    g[y * width + x] = g[y * width + x] * glm::vec3(0.25);
                }
        }

        for (std::size_t y = 0; y < inputFront.GetSizeY(); ++y)
            for (std::size_t x = 0; x < inputFront.GetSizeX(); ++x) {
                glm::vec3 color = g[y * width + x] + inputFront.At(x, y);
                output.At(x + offset.x, y + offset.y) = color;
            }
        delete[] g;
    }
```

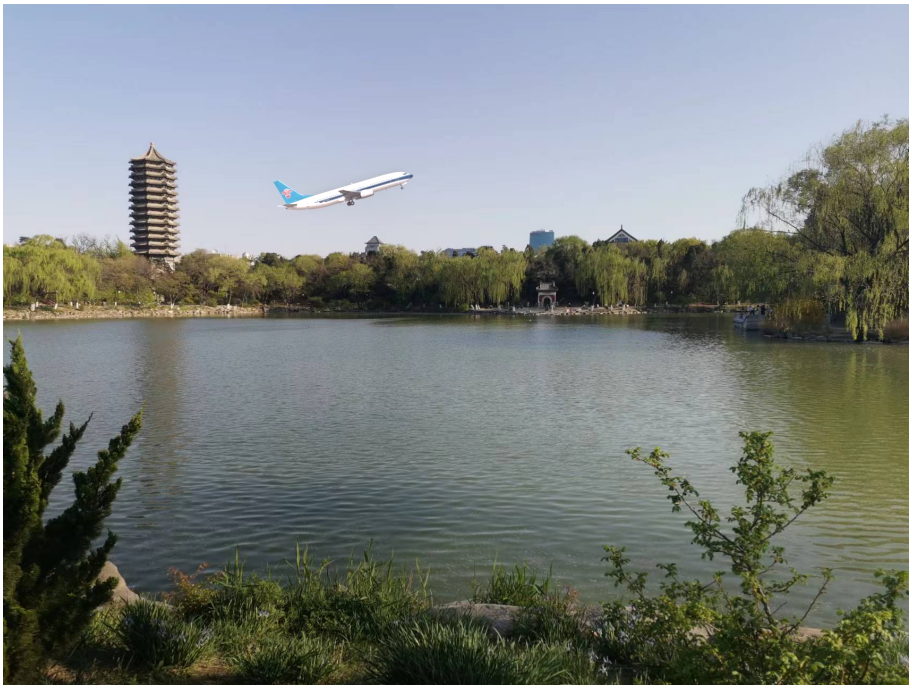


图 11: T3-inpainting

4 Task 4:Line Drawing

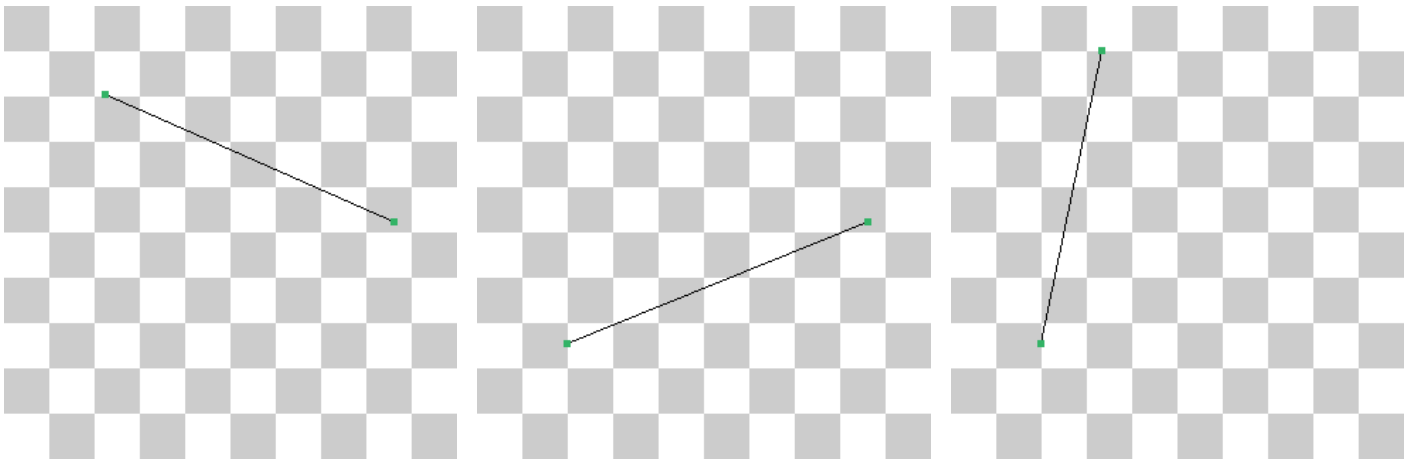
算法思路：首先先处理起点和终点 x 坐标相等或 y 坐标相等的特殊情况。然后规定 p_start 为 x 值较小的点，p_finish 为 x 值较大的点，用二者坐标来计算 k 值，然后根据 k 值进行分类讨论。

- 1.k 的绝对值小于等于 1, 此时直线较靠近 x 轴, 以 x 轴为基准, 使用 Bresenham 算法进行绘制
  - 1.1 k>0 x 递增, 考虑 y 何时 +1
  - 1.2 k<0 x 递增, 考虑 y 何时-1
- 2.k 的绝对值大于 1, 此时直线较靠近 y 轴, 以 y 轴为基准, 使用 Bresenham 算法进行绘制
  - 2.1 k>0 y 递增, 考虑 x 何时 +1
  - 2.2 k<0 y 递减, 考虑 x 何时 +1

```
1        void DrawLine(
2            ImageRGB &       canvas,
3            glm::vec3 const  color,
4            glm::ivec2 const p0,
5            glm::ivec2 const p1) {
6            // your code here:
7            if (p0.x == p1.x)
8            {
9                for (int i = p0.y; i <= p1.y; i++) canvas.At(p0.x, i) = color;
10           }
11           else if (p0.y == p1.y)
```

```
12              {
13                  for (int i = p0.x; i <= p1.x; i++) canvas.At(i,p0.y) = color;
14              }
15              else
16              {
17                  glm::ivec2 p_start=p0, p_finish=p1;
18                  if (p0.x > p1.x)
19                  {
20                      p_start = p1;
21                      p_finish = p0;
22                  }
23                  double x_start = p_start.x;
24                  double x_finish = p_finish.x;
25                  double y_start = p_start.y;
26                  double y_finish = p_finish.y;
27                  float k = (y_finish - y_start) / (x_finish - x_start);
28                  int   dx       = (x_finish - x_start)*2;
29                  int   dy       = (y_finish - y_start)*2;
30                  if (abs(k) <= 1)
31                  {
32                      if (k > 0)
33                      {
34                          int dydx = dy - dx, F = dy - dx / 2;
35                          int y = y_start;
36                          for (int x = x_start; x <= x_finish; x++)
37                          {
38                              canvas.At(x, y) = color;
39                              if (F < 0) F += dy;
40                              else {
41                                  y++;
42                                  F += dydx;
43                              }
44                          }
45                      }
46                      else
47                      {
48                          int dydx = dy + dx, F = dy + dx / 2;
49                          int y = y_start;
50                          for (int x = x_start; x <= x_finish; x++) {
51                              canvas.At(x, y) = color;
52                              if (F > 0) F += dy;
53                              else {
54                                  y--;
55                                  F += dydx;
56                              }
57                          }
58                      }
59                  }
60                  else
61                  {
62                      if (k > 0)
```

```
63                  {
64                      int dydx = dy - dx, F = dy / 2 - dx;
65                      int x = x_start;
66                      for (int y = y_start; y <= y_finish;y++)
67                      {
68                          canvas.At(x, y) = color;
69                          if (F > 0) F -= dx;
70                          else
71                          {
72                              x++;
73                              F += dydx;
74                          }
75                      }
76                  }
77                  else
78                  {
79                      int dydx = dy + dx, F = dy / 2 + dx;
80                      int x = x_start;
81                      for (int y = y_start; y != y_finish-1; y--) {
82                          canvas.At(x, y) = color;
83                          if (F < 0) F += dx;
84                          else {
85                              x++;
86                              F += dydx;
87                          }
88                      }
89                  }
90              }
91          }
92      }
```



(a) T4-Line1 (b) T4-Line2 (c) T4-Line3

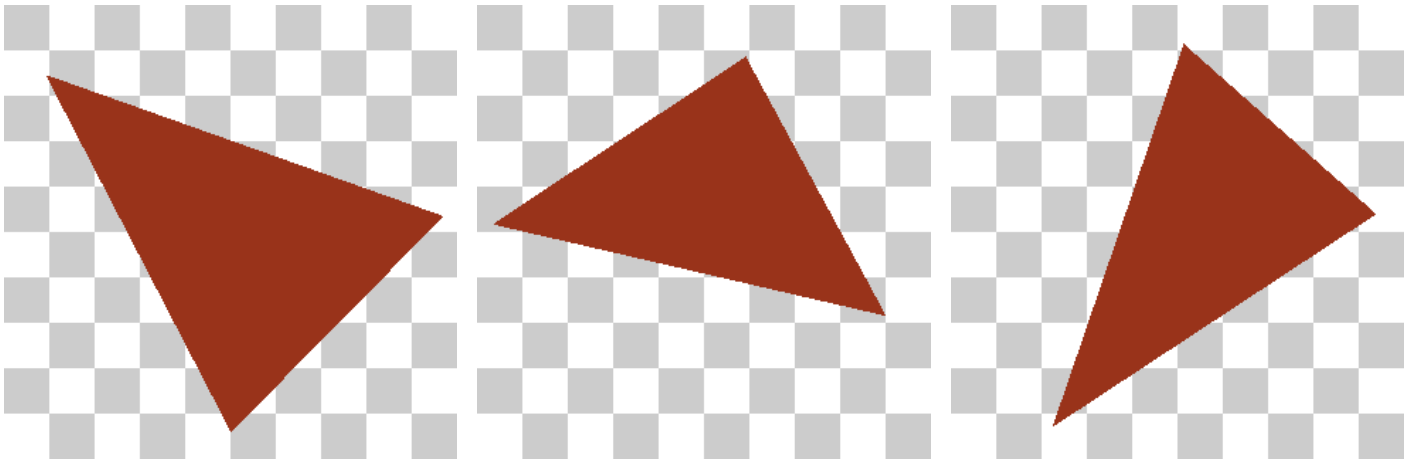
图 12: T4-Lines

5 Task 5: Triangle Drawing

算法思路：首先对三个点进行按照 x 坐标大小升序排序，然后从 x 值最小的点往中间点以扫描线的方式去进行填涂，类似画线做法，从这个起始位置分出两条指向另外两点的线，分别计算两条线的斜率然后以依次累加的方式计算每个 x 坐标下这两条线的 y 坐标 y1 与 y2，再将像素 (x,y1) 与 (x,y2) 之间的像素填涂。这样左边一半的三角形就填涂完成了，再从 x 值最大的点出发往中间点以扫描线的方式进行填涂，过程同理。

```
        bool my_p_cmpx(const glm::ivec2 & a, const glm::ivec2 & b) {
            return a.x < b.x;
        }
        void DrawTriangleFilled(
            ImageRGB &       canvas,
            glm::vec3 const  color,
            glm::ivec2 const p0,
            glm::ivec2 const p1,
            glm::ivec2 const p2) {
            // your code here:
            glm::ivec2 p_lstx[3] = { p0, p1, p2 };
            std::sort(p_lstx, p_lstx + 3, my_p_cmpx);

            int min_x = p_lstx[0].x;
            int max_x = p_lstx[2].x;

            DrawLine(canvas, color, p0, p1);
            DrawLine(canvas, color, p0, p2);
            DrawLine(canvas, color, p1, p2);

            float y1      = p_lstx[0].y;
            float y2      = p_lstx[0].y;
            float k01     = ((float) p_lstx[1].x - (float) p_lstx[0].x == 0) ? 1000 : (((float) p_lstx[1].y - (float) p_lstx[0].y) / ((float) p_lstx[1].x - (float) p_lstx[0].x));
            float k12     = ((float) p_lstx[1].x - (float) p_lstx[2].x == 0) ? 1000 : (((float) p_lstx[1].y - (float) p_lstx[2].y) / ((float) p_lstx[1].x - (float) p_lstx[2].x));
            float k02     = ((float) p_lstx[2].x - (float) p_lstx[0].x == 0) ? 1000 : (((float) p_lstx[2].y - (float) p_lstx[0].y) / ((float) p_lstx[2].x - (float) p_lstx[0].x));
            for (int i = min_x; i <= p_lstx[1].x; i++)
            {
                for (int j = std::min(y1, y2); j <= std::max(y1, y2);j++)
                {

                    canvas.At(i, j) = color;
                }
                y1 += k01;
                y2 += k02;
            }
            y1 = p_lstx[2].y;
            y2 = p_lstx[2].y;
            for (int i = max_x; i >= p_lstx[1].x; i--) {
                for (int j = std::min(y1, y2); j <= std::max(y1, y2); j++) {
                    canvas.At(i, j) = color;
                }
```

```
                y1 -= k02;
                y2 -= k12;
            }
        }
```



(a) T5-Triangle1 (b) T5-Triangle2 (c) T5-Triangle3

图 13: T5-Triangles

6 Task 6: Image Supersampling

算法思路：先分别计算出输入图和输出图的长、宽的比率，使我们后续能够输出完整的图像。然后我们依次对输出图像的各个像素点进行求值，求值的方法采取超采样去走样方法，按照给定的 rate 对图像的对应像素周围 rate*tate 个像素求加权平均 (超出边界的像素不纳入计算)，得出对应输出像素点的像素值。

```
    void Supersample(
        ImageRGB &       output,
        ImageRGB const & input,
        int              rate) {
        // your code here:
        int x_rate = input.GetSizeX() / output.GetSizeX();
        int y_rate = input.GetSizeY() / output.GetSizeY();

        for (int i = 0; i < output.GetSizeX(); i++)
        {
            for (int j = 0; j < output.GetSizeY(); j++)
            {
                glm::vec3 color_ave = { 0.0, 0.0, 0.0 };
                int       cnt       = 0;
                for (int i_x = 0; i_x < rate; i_x++)
                {
                    for (int i_y = 0; i_y < rate; i_y++)
                    {
                        int samplex = (i+4) * x_rate + i_x;
                        int sampley = (j+4) * y_rate + i_y;
                        if (samplex >= input.GetSizeX() || sampley >= input.GetSizeY())
continue;
```

```
                    cnt++;
                    color_ave += input.At(samplex, sampley);
                }
            }
            color_ave       = { color_ave.r / (float) cnt, color_ave.g / (float) cnt,
color_ave.b / (float) cnt };
            output.At(i, j) = color_ave;
        }
    }
}
```


(a) T6-original


(b) T6-super_rate=4


(c) T6-super_rate=12

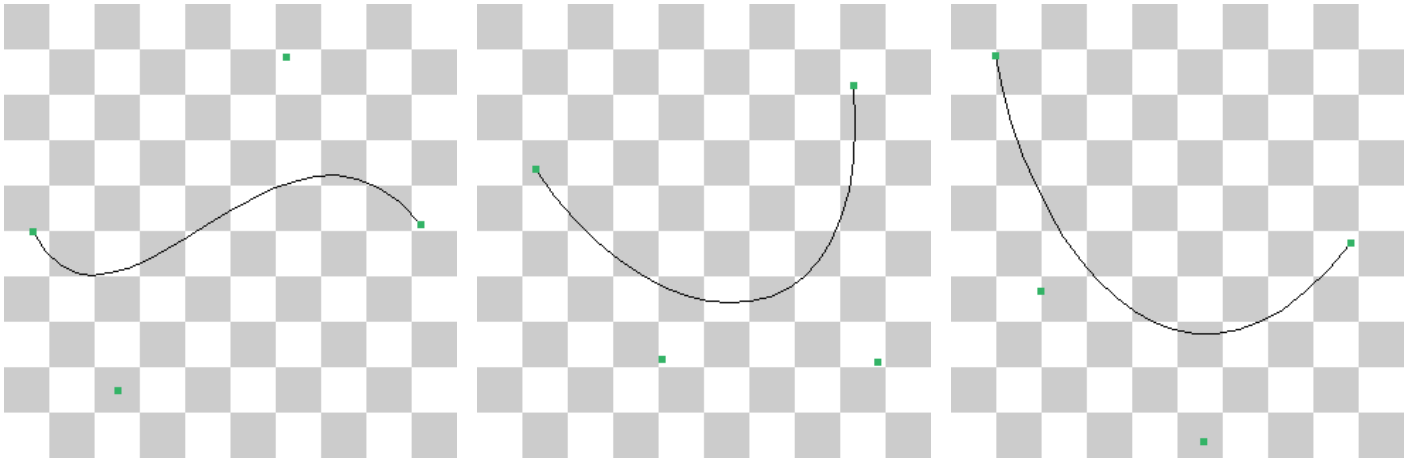
图 14: T6-Supersampling

7 Task 7: Bezier Curve

算法思路：我们不难发现，对于每个固定的 t 值，每进行一次线性插值计算后，待操作的 points 数目都会-1，最后等到待操作的 points 只剩下两个时，再进行一次插值计算就可以得到目标 Q 点的坐标。于是我们设计一个 points_vec 的 vector 来记录待操作的 points, 进行两重循环，内层循环表示将相邻两个待操作点进行插值运算得到一个新点，外层循环表示对所有产生的新点进行新一轮的插值计算，注意外层操作轮数由初始点个数决定，而每完成一轮外层操作，点个数会减少一个，使新的一轮内层循环轮数减一。最后输出 points_vec[0] 表示最后一次操作得到的目标点坐标。

```
glm::vec2 CalculateBezierPoint(
std::span<glm::vec2> points,
float const          t) {
// your code here:
int ori_points_num = 0;

std::vector<glm::vec2> points_vec;
for (auto i = points.begin(); i != points.end(); i++)
{
    points_vec.push_back(*i);
    ori_points_num++;
}
```

```
    for (int i = 1; i < ori_points_num; i++)
    {
        for (int j = 0; j < ori_points_num - i; j++)
        {
            points_vec[j] = (1 - t) * points_vec[j] + t * points_vec[j + 1];
        }
    }
    return points_vec[0];
    }
```



(a) T7-Bezier_Curve1 (b) T7-Bezier_Curve2 (c) T7-Bezier_Curve3

图 15: T7-Bezier_Curve