

# Lab 1 报告

蒋锦豪 2400011785

2025 年 10 月 2 日

## Task 1: Image Dithering

### Uniform Random

先调用 `random` 库生成均匀分布的随机数, 然后对每个像素点进行处理, 若加上随机数后大于 0.5 则设为黑色, 否则设为白色. 代码实现如下:

```
1 void DitheringRandomUniform(  
2     ImageRGB &      output,  
3     ImageRGB const & input) {  
4     // your code here:  
5     std::random_device rd;  
6     std::mt19937 gen((unsigned int)rd());  
7     std::uniform_real_distribution<float> dis(-0.5,0.5);  
8     for (std::size_t x = 0; x < input.GetSizeX(); ++x)  
9         for (std::size_t y = 0; y < input.GetSizeY(); ++y) {  
10             glm::vec3 color = input.At(x, y);  
11             int graysacle = (color.r + dis(gen)) > 0.5 ? 1 : 0;  
12             output.At(x, y) = { graysacle, graysacle, graysacle, };  
13         }  
14 }
```

效果图如下:



图 1: Uniform Random Dithering 的运行结果

## Blue Noise Random

将 `input` 和 `noise` 两张图像的像素值的平均值与 0.5 进行比较, 若大于 0.5 则设为黑色, 否则设为白色. 实际运算时直接将像素值之和与 1 比较即可. 代码实现如下:

```

1 void DitheringRandomBlueNoise(
2     ImageRGB &      output,
3     ImageRGB const & input,
4     ImageRGB const & noise) {
5     // your code here:
6     for (std::size_t x = 0; x < input.GetSizeX(); ++x)
7         for (std::size_t y = 0; y < input.GetSizeY(); ++y) {
8             glm::vec3 color_input = input.At(x, y);
9             glm::vec3 color_noise = noise.At(x, y);
10            int         graysacle   = (color_input.r + color_noise.r) > 1 ? 1 : 0;
11            output.At(x, y)        = {
12                graysacle, graysacle, graysacle, };
13        }
14 }

```

效果图如下:



图 2: Blue Noise Random Dithering 的运行结果

## Ordered

将原图的像素值  $i$  扩大为  $9i$ , 于是颜色取值范围变为  $[0, 9]$ . 将  $9i$  与  $3 \times 3$  的抖动矩阵 **mat** 的每个元素进行比较, 如果大于该元素的值则将对对应位置设为黑色, 否则设为白色. 代码实现如下:

```

1 void DitheringOrdered(
2     ImageRGB &      output,
3     ImageRGB const & input) {
4     // your code here:
5     int mat[3][3] = {
6         { 6, 8, 4 }, { 1, 0, 3 }, { 5, 2, 7 }
7     };
8     for (std::size_t x = 0; x < input.GetSizeX(); ++x)
9         for (std::size_t y = 0; y < input.GetSizeY(); ++y) {
10             glm::vec3 color = input.At(x, y);
11             float      color_judge = color.r * 9;
12             for (int i = 0; i < 3; i++) {
13                 for (int j = 0; j < 3; j++) {
14                     int color_draw = color_judge >= mat[i][j] ? 1 : 0;
15                     output.At(3 * x + i, 3 * y + j) = {
16                         color_draw, color_draw, color_draw, };
17                 }
18             }
19         }
20 }

```

效果图如下:



图 3: Ordered Dithering 的运行结果

## Error Diffuse

先将 `input` 的像素值存入一个二维数组 `colormat` 中, 然后从左到右, 从上到下遍历每个像素点, 将 `colormat` 存储的像素值与 0.5 比较后将 `output` 对应位置的像素设为黑色或白色, 并计算出误差 `delta`. 按照 Floyd-Steinberg 抖动矩阵和误差值 `delta` 在 `colormat` 中更新右方和下方的像素值; 同时需要注意边界处像素的处理. 代码实现如下:

```

1 void DitheringErrorDiffuse(
2     ImageRGB & output,
3     ImageRGB const & input) {
4     // your code here:
5     std::size_t row = input.GetSizeX(), col = input.GetSizeY();
6     std::vector<std::vector<float>> colormat(row, std::vector<float>(col, 0));
7     for (std::size_t x = 0; x < row; ++x) {
8         for (std::size_t y = 0; y < col; ++y) {
9             glm::vec3 color = input.At(x, y);
10            colormat[x][y] = color.r;

```

```

11     }
12 }
13 for (std::size_t y = 0; y < col; ++y) {
14     for (std::size_t x = 0; x < row; ++x) {
15         int color      = colormat[x][y] > 0.5 ? 1 : 0;
16         output.At(x, y) = {
17             color, color, color, };
18         float delta = colormat[x][y] - (colormat[x][y] > 0.5 ? 1 : 0);
19         if (x < row - 1) { colormat[x + 1][y] += delta * 7 / 16; }
20         if (y < col - 1) {
21             colormat[x][y + 1] += delta * 5 / 16;
22             if (x > 0) { colormat[x - 1][y + 1] += delta * 3 / 16; }
23             if (x < row - 1) { colormat[x + 1][y + 1] += delta * 1 / 16; }
24         }
25     }
26 }
27 }

```

效果图如下:



图 4: Error Diffuse Dithering 的运行结果

## Task 2 Image Filtering

### Blur

使用  $3 \times 3$  的均值滤波器

$$K = \frac{1}{9} \begin{vmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{vmatrix}$$

对图像进行滤波. 具体实现时, 遍历每个像素, 将该像素及其周围 8 个像素的值相加后除以 9 即为滤波后的像素值. 对于图像边缘的像素, 如果越界则不计入求和和计数. 代码实现如下:

```

1 void Blur(
2     ImageRGB & output,
3     ImageRGB const & input) {
4     // your code here:
5     std::size_t row = input.GetSizeX(), col = input.GetSizeY();
6     for (std::size_t x = 0; x < row; ++x)
7         for (std::size_t y = 0; y < col; ++y) {
8             int ave_counter = 0;
9             glm::vec3 color(0.0f, 0.0f, 0.0f);
10            for (int i = -1; i <= 1; i++) {
11                for (int j = -1; j <= 1; j++) {
12                    if (x + i >= 0 && x + i < row && y + j >= 0 && y + j < col) {
13                        ave_counter++;
14                        color += input.At(x + i, y + j);
15                    }
16                }
17            }
18            output.At(x, y) = {
19                color.r / ave_counter, color.g / ave_counter, color.b / ave_counter,
20            };
21        }
22    }

```

效果图如下:

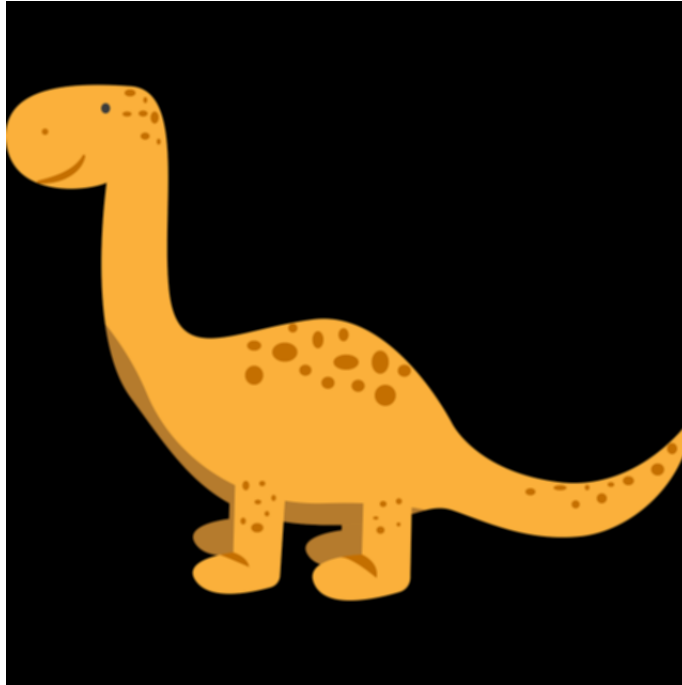


图 5: Blur(使用 Box Filter) 的运行结果

## Edge Detect

使用 Sobel 算子

$$\mathbf{K}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad \mathbf{K}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

对图像进行滤波, 然后以结果的均方根作为输出像素值. 具体实现时, 遍历每个像素, 将该像素及其周围 8 个像素的值与  $\mathbf{K}_x$  和  $\mathbf{K}_y$  对应位置的值相乘后求和, 结果分别记作 `color_x` 和 `color_y`, 再对每个颜色通道分别计算它们的平方和开根号, 结果即滤波后的像素值. 对于图像边缘的像素, 与 **Blur** 的处理方法相同, 如果越界则不计入求和. 代码实现如下:

```

1 void Edge(
2     ImageRGB &      output,
3     ImageRGB const & input) {
4     // your code here:
5     float G_x[3][3] = {
6         { -1.0f, 0.0f, 1.0f },
7         { -2.0f, 0.0f, 2.0f },
8         { -1.0f, 0.0f, 1.0f }
9     };
10    float G_y[3][3] = {

```

```
11     { 1.0f, 2.0f, 1.0f },
12     { 0.0f, 0.0f, 0.0f },
13     { -1.0f, -2.0f, -1.0f }
14 };
15 std::size_t row = input.GetSizeX(), col = input.GetSizeY();
16 for (std::size_t x = 0; x < row; ++x)
17     for (std::size_t y = 0; y < col; ++y) {
18         glm::vec3 color_x(0.0f, 0.0f, 0.0f);
19         glm::vec3 color_y(0.0f, 0.0f, 0.0f);
20         for (int i = -1; i <= 1; i++) {
21             for (int j = -1; j <= 1; j++) {
22                 if (x + i >= 0 && x + i < row && y + j >= 0 && y + j < col) {
23                     glm::vec3 color_cur = input.At(x + i, y + j);
24                     color_x += color_cur * G_x[i + 1][j + 1];
25                     color_y += color_cur * G_y[i + 1][j + 1];
26                 }
27             }
28         }
29         output.At(x, y) = {
30             sqrt(pow(color_x.r, 2) + pow(color_y.r, 2)),
31             sqrt(pow(color_x.g, 2) + pow(color_y.g, 2)),
32             sqrt(pow(color_x.b, 2) + pow(color_y.b, 2)),
33         };
34     }
35 }
```

效果图如下:



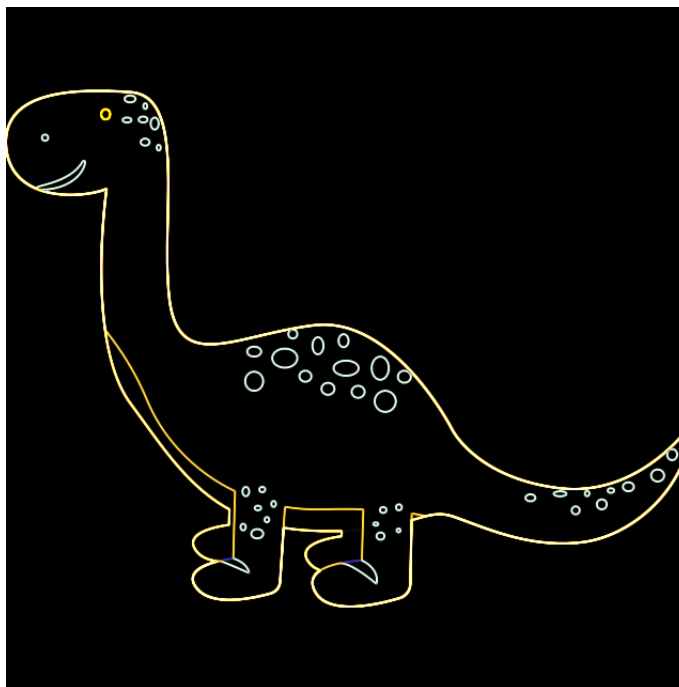


图 6: Edge Detect(使用 Sobel Filter) 的运行结果

### Task 3: Image Inpainting

采用任务指示中给出的记号, 将修改前的图片记作  $f$ , 即传入的参数 `inputFront`, 编辑量记作  $g$ , 编辑后的图像记作  $f + g$ , 即需要输出的图片 `output` 中对应 `inputFront` 的区域. 于是不难得出

$$g = (f + g) - f = \text{output} - \text{inputFront}$$

最终要求  $\nabla^2 g = 0$ . 按照 Poisson Editing 的思路, 需要将设置边界处的  $g$ , 使得在边界上总有 `output` 与背景 `inputBack` 相同. 于是边界部分的  $g$  应当按

$$g = \text{output} - \text{inputFront} = \text{inputBack} - \text{inputFront}$$

设置, 该部分代码如下:

```

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

```

```

7  }
8  for (std::size_t x = 0; x < width; ++x) {
9      // set boundary for (x, 0), your code: g[x] = ?
10     g[x] = (glm::vec3) inputBack.At(x + offset.x, offset.y) - (glm::vec3) inputFront.At(x, 0);
11     // set boundary for (x, height - 1), your code: g[(height - 1) * width + x] = ?
12     g[(height - 1) * width + x] = (glm::vec3) inputBack.At(x + offset.x, offset.y + height - 1) - (
13         glm::vec3) inputFront.At(x, height - 1);
14 }

```

效果图如下:

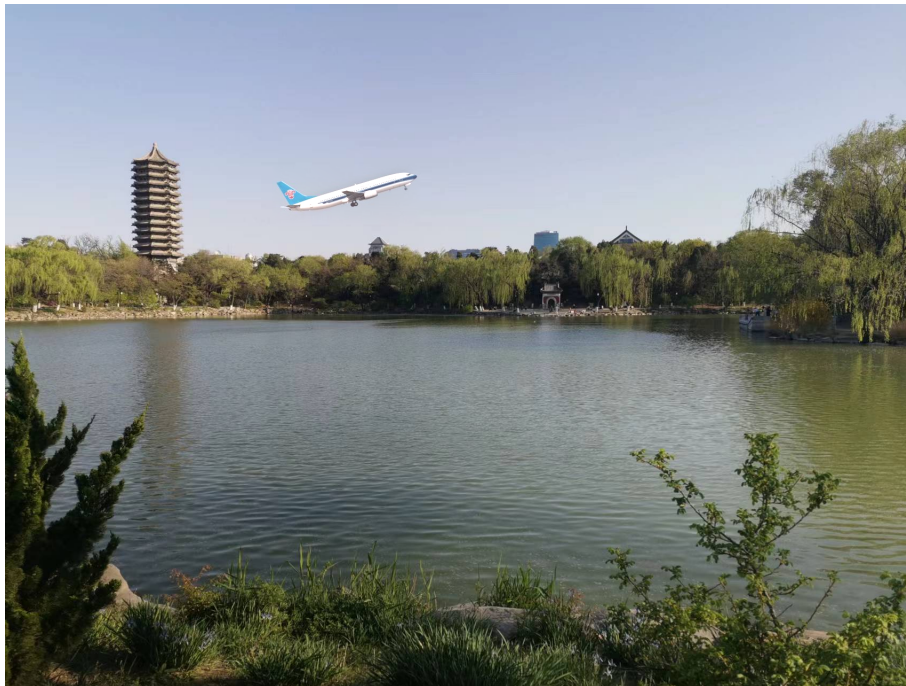


图 7: Image Inpainting 的运行结果

## Task 4: Line Drawing

按照 Bresenham 算法实现直线绘制. 具体实现时, 先判断直线的斜率是否大于 1, 若是, 则将  $x_0$  与  $y_0$ ,  $x_1$  与  $y_1$  交换, 并在后续绘制时将横纵坐标互换回来; 然后保证  $x_0 \leq x_1$ , 若不满足则交换两端点; 接着按照 Bresenham 算法计算  $dx$  和  $dy$ , 以及判断函数  $F$  和  $d$ ; 最后从  $x_0$  遍历到  $x_1$ , 每次根据决策参数  $F$  决定是否改变  $y$  (对  $y$  的增减取决于直线斜率的正负, 在确定  $x_1 > x_0$  的情况下, 如果  $y_1 > y_0$  说明斜率为正, 每次将  $y$  加 1; 否则就每次将  $y$  减 1.), 并更新判断函数  $F$ . 代码实现如下:

```

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   int x0 = p0.x, y0 = p0.y, x1 = p1.x, y1 = p1.y;
8   int x, y, dx, dy, d, F, cx, cy;
9   bool f = std::abs(x1 - x0) < std::abs(y1 - y0);
10  if (f) { std::swap(x0, y0); std::swap(x1, y1); }
11  if (x0 > x1) { std::swap(x0, x1); std::swap(y0, y1); }
12  y = y0;
13  dx = 2 * (x1 - x0); dy = 2 * std::abs(y1 - y0);
14  d = dy - dx; F = dy - dx / 2;
15  for (x = x0; x <= x1; x++) {
16      cx = f ? y : x; cy = f ? x : y;
17      canvas.At(cx, cy) = { color.r, color.g, color.b, };
18      if (F < 0) { F = F + dy; }
19      else { y += (y1 > y0) ? 1 : -1; F = F + d; }
20  }
21 }

```

效果图如下:

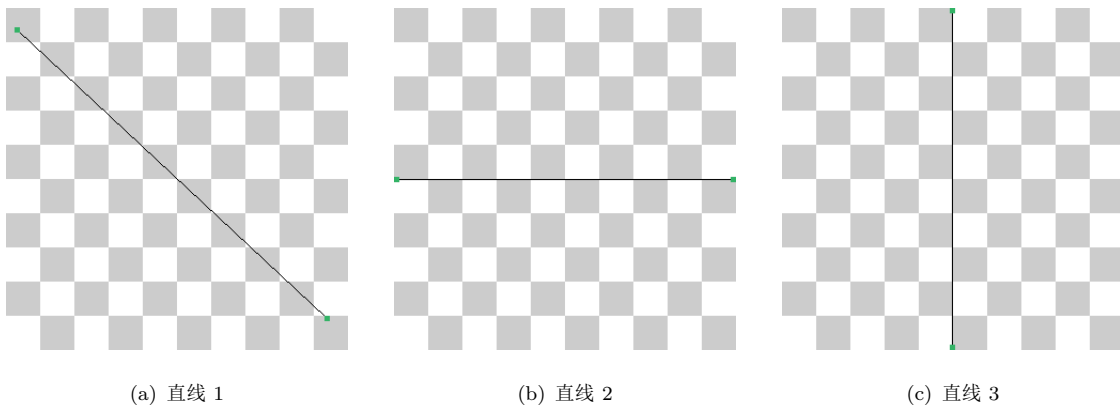


图 8: Line Drawing 的运行结果

## Task 5: Triangle Drawing

按照扫描线算法实现三角形填充. 具体实现时, 先将三个顶点按横坐标从小到大排序, 然后以位居中间的点为界分两段处理. 对于左半段, 先计算两条线段的斜率, 然后从左到右遍历每列像素, 按照 DDA 算法更新该列填充的上下边界 **ymin** 和 **ymax**, 并对该列上在上下边界之间的像素点进行填充; 对于右半段, 同理, 只是从右到左遍历每个像素点. 代码实现如下:

```

1  void DrawTriangleFilled(
2      ImageRGB &      canvas,
3      glm::vec3 const color,
4      glm::ivec2 const p0,
5      glm::ivec2 const p1,
6      glm::ivec2 const p2) {
7      // your code here:
8      int      x0 = p0.x, y0 = p0.y, x1 = p1.x, y1 = p1.y, x2 = p2.x, y2 = p2.y;
9      float      ymin, ymax;
10     float      k1, k2;
11     if (x2 < x1) { std::swap(x1, x2); std::swap(y1, y2); }
12     if (x1 < x0) { std::swap(x0, x1); std::swap(y0, y1); }
13     if (x0 != x1 && x2 != x0) {
14         ymin = 1.0 * y0; ymax = 1.0 * y0;
15         k1 = 1.0 * (y1 - y0) / (x1 - x0); k2 = 1.0 * (y2 - y0) / (x2 - x0);
16         if (k1 > k2) { std::swap(k1, k2); }
17         for (int x = x0; x <= std::min(x1, x2); x++) {
18             for (int y = std::round(ymin); y < ymax+0.5; y++) {
19                 canvas.At(x, y) = { color.r, color.g, color.b, };
20             }
21             ymin += k1; ymax += k2;
22         }
23     }
24     if (x1 != x2 && x2 != x0) {
25         ymin = 1.0 * y2; ymax = 1.0 * y2;
26         k1 = 1.0 * (y1 - y2) / (x2 - x1); k2 = 1.0 * (y0 - y2) / (x2 - x0);
27         if (k1 > k2) { std::swap(k1, k2); }
28         for (int x = x2; x >= std::max(x1, x0); x--) {
29             for (int y = std::round(ymin); y < ymax+0.5; y++) {
30                 canvas.At(x, y) = { color.r, color.g, color.b, };
31             }
32             ymin += k1; ymax += k2;
33         }
34     }
35 }

```

效果图如下:

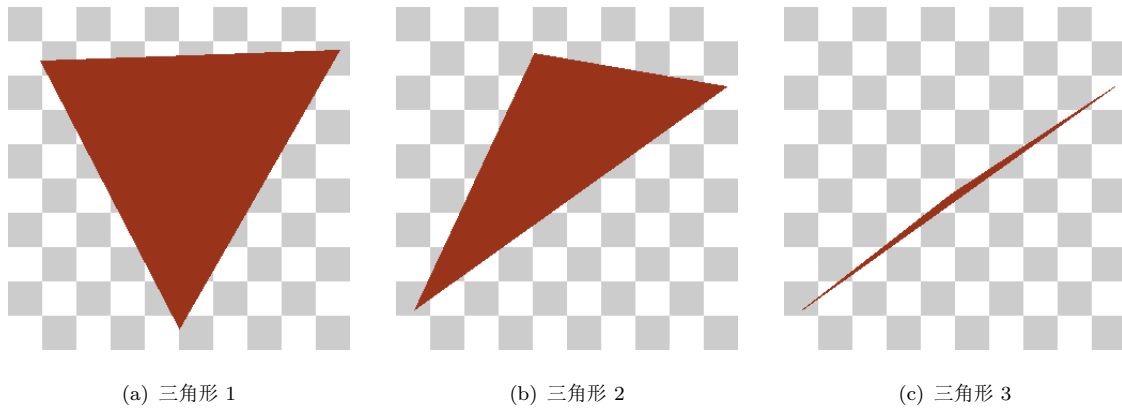


图 9: Triangle Drawing 的运行结果

## Task 6: Image Supersampling

使用双线性插值对图像进行放大, 然后对放大的图像进行均值滤波. 具体实现时, 先创建一个二维数组 `colormatrix` 存储放大后的图像的像素值, 然后遍历放大后图像的每个像素点, 计算出其在原图中对应的浮点坐标  $(sx, sy)$ , 并找出其周围的 4 个像素点  $(x0, y0)$ ,  $(x1, y0)$ ,  $(x0, y1)$ ,  $(x1, y1)$ , 以及它们与  $(sx, sy)$  的距离  $dx$  和  $dy$ , 最后按照双线性插值公式计算出该像素点的颜色并存入 `colormatrix`. 接着遍历输出图像的每个像素点, 将 `colormatrix` 中对应的  $r \times r$  个像素点的颜色值相加后除以  $r^2$  即为输出图像该像素点的颜色值. 代码实现如下:

```

1 void Supersample(
2     ImageRGB & output,
3     ImageRGB const & input,
4     int rate) {
5     int output_row = output.GetSizeX(),
6         output_col = output.GetSizeY(),
7         input_row = input.GetSizeX(),
8         input_col = input.GetSizeY();
9     std::vector<std::vector<glm::vec3>> colormatrix(output_row * rate, std::vector<glm::vec3>(
10         output_col * rate, { 0.0f, 0.0f, 0.0f }));
11     for (int x = 0; x < output_row * rate; x++) {
12         for (int y = 0; y < output_col * rate; y++) {
13             float sx = (x + 0.5f) * input_row / (output_row * rate);
14             float sy = (y + 0.5f) * input_col / (output_col * rate);
15             int x0 = std::clamp((int) std::floor(sx), 0, input_row - 1);
16             int y0 = std::clamp((int) std::floor(sy), 0, input_col - 1);
17             int x1 = std::clamp(x0 + 1, 0, input_row - 1);
18             int y1 = std::clamp(y0 + 1, 0, input_col - 1);

```

```

18     float    dx = sx - x0, dy = sy - y0;
19     glm::vec3 color_cur = (1 - dx) * (1 - dy) * (glm::vec3)(input.At(x0, y0))
20         + dx * (1 - dy) * (glm::vec3) input.At(x1, y0)
21         + dy * (1 - dx) * (glm::vec3) input.At(x0, y1)
22         + dx * dy * (glm::vec3) input.At(x1, y1);
23     colormatrix[x][y] = color_cur;
24 }
25 }
26 float inv = 1.0f / (rate * rate);
27 for (int x = 0; x < output_row; x++) {
28     for (int y = 0; y < output_col; y++) {
29         glm::vec3 color { 0.0f, 0.0f, 0.0f };
30         for (int i = 0; i < rate; i++) {
31             for (int j = 0; j < rate; j++) {
32                 color += colormatrix[x * rate + i][y * rate + j];
33             }
34         }
35         color *= inv;
36         output.At(x, y) = { color.r, color.g, color.b, };
37     }
38 }
39 }

```

效果图如下:



(a) 不进行 SSAA



(b) SSAA×2



(c) SSAA×5

图 10: SSAA 的运行结果

## Task 7: Bezier Curve

使用 De Casteljau 算法计算 Bezier 曲线上的点. 具体实现时, 先将传入的 4 个控制点存入一个数组 `p` 中 (传入的 `span` 类型存储的是各个点的地址, 不能直接修改), 然后进行 3 次插值, 每次插值后 `p` 中存储的点数减 1, 直到最后 `p` 中只剩下一个点, 该点即为所求. 代码实现如下:

```

1 glm::vec2 CalculateBezierPoint(
2     std::span<glm::vec2> points,
3     float const      t) {
4     std::vector<glm::vec2> p = { points[0],
5                                   points[1],
6                                   points[2],
7                                   points[3] };
8     for (int i = 1; i <= 3; i++) {
9         for (int j = 3; j >= i; j--) {
10             p[j] = t * p[j] + (1 - t) * p[j - 1];
11         }
12     }
13     return p[3];
14 }

```

效果图如下:

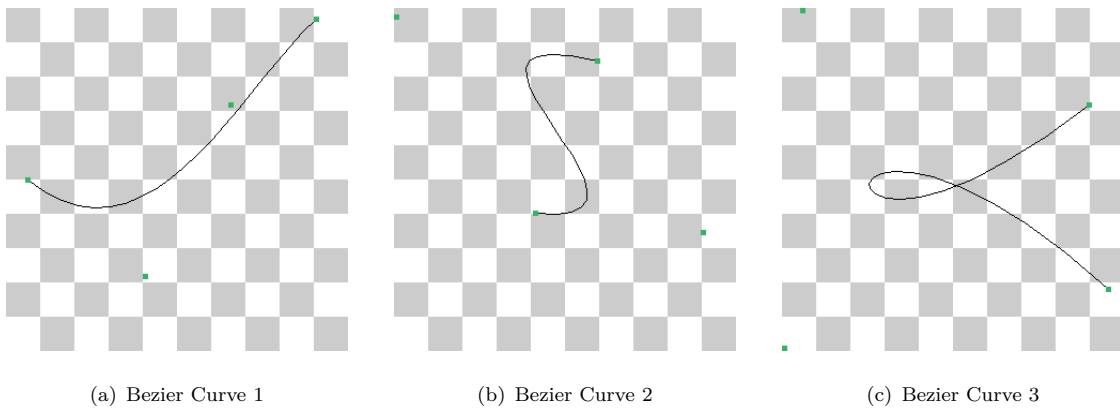


图 11: Bezier Curve 的运行结果