计算机学院实验报告

实验题目： 线段裁剪和光栅化		学号：202300130183
日期：2025/4/14	班级：23 级智能班	姓名：宋浩宇

Email：2367651943@qq.com
202300130183@mail.sdu.edu.cn

实验目的：
作业 5：线段裁剪和光栅化
具体要求：
1，线段裁剪采用 Liang-baskey 算法
2，光栅化采用中点画线算法
3，在实验 2 作业框架的基础上，或其他框架的基础上；
4，算法结果需要可视化呈现在屏幕上；
5，支持键盘输入线段的任意起点和终点坐标；
6，实验周期为 1 周；

实验环境介绍：
软件环境：
主系统：Windows 11 家庭中文版 23H2 22631.4317
虚拟机软件：Oracle Virtual Box 7.1.6
虚拟机系统：Ubuntu 18.04.2 LTS
编辑器：Visual Studio Code
编译器：gcc 7.3.0
计算框架：Eigen 3.3.7

硬件环境：
CPU：13th Gen Intel(R) Core(TM) i9-13980HX 2.20 GHz
内存：32.0 GB (31.6 GB 可用)
磁盘驱动器：NVMe WD_BLACKSN850X2000GB
显示适配器：NVIDIA GeForce RTX 4080 Laptop GPU

解决问题的主要思路：

1. 本次实验主要使用 Liang-Berskey 截取算法和中点画线光栅化算法。我们选择不使用之前实验的代码框架，自行编写代码。
2. 我们自行编写 CMakeList.txt 用于构建程序。
3. 首先是 Liang-Berskey 算法，

考虑直线的参数方程：

$$x = x_0 + t(x_1 - x_0) = x_0 + t\Delta x,$$

$$y = y_0 + t(y_1 - y_0) = y_0 + t\Delta y.$$

点在裁剪窗内，若

$$x_{\min} \leq x_0 + t\Delta x \leq x_{\max}$$

且

$$y_{\min} \leq y_0 + t\Delta y \leq y_{\max},$$

其可用 4 个不等式表达：

$$tp_i \leq q_i, \quad i = 1,2,3,4,$$

其中

$$p_1 =- \Delta x, \quad q_1 = x_0 - x_{\min}, \quad (左)$$

$$p_2 = \Delta x, \quad q_2 = x_{\max} - x_0, \quad (右)$$

$$p_3 =- \Delta y, \quad q_3 = y_0 - y_{\min}, \quad (下)$$

$$p_4 = \Delta y, \quad q_4 = y_{\max} - y_0. \quad (上)$$

计算最终线段：

1. 与裁剪窗平行的直线在平行的边界上有 $p_i = 0$
2. 若对于这样的 $iq_i < 0$，则线段全部在裁剪窗的外面，可以被消除
3. 当 $p_i < 0$ 时，线从裁剪窗外向内走；$p_i > 0$
4. 对非零的 p_k，$u = q_i/p_i$
5. 对每条线，计算 u_1 和 u_2。对 u_1 检查 $p_i < 0$ 的边界（即从外向内）。令 u_1 为 $\{0, q_i/p_i\}$ u_2 检查 $p_i > 0$ 的边界（即从内向外）。令 u_2 为 $\{1, q_i/p_i\}$ $u_1 > u_2$

4. 然后是中点画线算法。

我们知道直线方程的一般形式为：

$$f(x,y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0(1)$$

在这里我们假设$x_1 > x_0$,这里为什么要这样假设，请看下文。 我们先来考虑直

线斜率$k \in (0,1)$的情况。在这种情况下，直线是以沿x轴方向的变化要快于沿y轴方向的变化。此时的直线是一条从左向右平缓上升的直线。即图 1 所示，不难想象的是，这时我们要光栅化一条直线的话，从左下角的起点出发，每次要"点亮"的像素的x 坐标都是上一个像素的x坐标加 1，而y坐标则是根据某种条件，保持不变或者加 1，如此反复直至终点。

5. 最后，实现一下循环输入线段和输出图片即可。

实验步骤与实验结果：

首先先完成 CMakeList.txt 的编写

```
cmake_minimum_required(VERSION 3.0)
project(liang-berskey-line-clipping)

find_package(OpenCV REQUIRED)
include_directories(${OpenCV_INCLUDE_DIRS})

add_executable(liang-berskey-line-clipping
liang-berskey-line-clipping.cpp)

# 链接 OpenCV 库
target_link_libraries(liang-berskey-line-clipping ${OpenCV_LIBS})
```

为了方便我们描述线段的参数方程和一般方程，我们定义两个类：

```
class LineSegmentParametricEquation2f
{
private:
  Eigen::Vector2f p0, t;
  float upper_bound, lower_bound;

public:
  LineSegmentParametricEquation2f(const Eigen::Vector2f& p0,
                                  const Eigen::Vector2f& p1)
      : p0(p0)
  {
    float dx = p1.x() - p0.x();
    float dy = p1.y() - p0.y();
    t = Eigen::Vector2f(dx, dy);
```

```
    lower_bound = 0.0f;
    upper_bound = 1.0f;
  }
  LineSegmentParametricEquation2f(const Eigen::Vector3f& p0,
                                  const Eigen::Vector3f& p1)
  {
    this->p0[0] = p0[0];
    this->p0[1] = p0[1];
    float dx = p1[0] - p0[0];
    float dy = p1[1] - p0[1];
    t = Eigen::Vector2f(dy, dx);
    lower_bound = 0.0f;
    upper_bound = 1.0f;
  }
  Eigen::Vector2f operator()(float s) const { return p0 + s * t; }
  void set_upper_bound(float upper) { upper_bound = upper; }
  void set_lower_bound(float lower) { lower_bound = lower; }
  Eigen::Vector2f get_p0() const { return p0; }
  Eigen::Vector2f get_p1() const { return p0 + t * upper_bound; }
  void display() const
  {
    std::cout << "p0: " << p0.transpose() << std::endl;
    std::cout << "t: " << t.transpose() << std::endl;
    std::cout << "upper_bound: " << upper_bound << std::endl;
    std::cout << "lower_bound: " << lower_bound << std::endl;
  }
};
class LineSegmentNormalEquation2f
{
private:
  float a, b, c;
  float upper_bound, lower_bound;
public:
  LineSegmentNormalEquation2f(const Eigen::Vector2f& p0,
                              const Eigen::Vector2f& p1)
  {
    a = p0[1] - p1[1];                  // y1 - y2
    b = p1[0] - p0[0];                  // x2 - x1
    c = p0[0] * p1[1] - p1[0] * p0[1]; // x1*y2 - x2*y1
    upper_bound = p1.x();
    lower_bound = p0.x();
  }
  float operator()(float s) const
  {
```

```
    if (lower_bound == upper_bound)
    {
      return 0;
    }
    return -(a * s + c) / b;
  }
  float operator()(float x, float y) const
  {
    if (b == 0)
      return 0;
    return a * x + b * y + c;
  }
  LineSegmentNormalEquation2f(const
LineSegmentParametricEquation2f& l)
  {
    Eigen::Vector2f p0 = l.get_p0();
    Eigen::Vector2f p1 = l.get_p1();
    a = p0[1] - p1[1];
    b = p1[0] - p0[0];
    c = p0[0] * p1[1] - p1[0] * p0[1];
    upper_bound = l.get_p1().x();
    lower_bound = l.get_p0().x();
  }
  float get_a() const { return a; }
  float get_b() const { return b; }
  float get_c() const { return c; }
  float get_k() const { return -a / b; }
  Eigen::Vector2f get_p0() const
  {
    return Eigen::Vector2f(lower_bound,
this->operator()(lower_bound));
  }
  Eigen::Vector2f get_p1() const
  {
    return Eigen::Vector2f(upper_bound,
this->operator()(upper_bound));
  }
};
```

然后实现 liang-berskey 算法

```
LineSegmentParametricEquation2f Liang_Barsky(Eigen::Vector2f&
begin,
                                             Eigen::Vector2f&
end)
{
```

```cpp
LineSegmentParametricEquation2f line(begin, end);
// 计算参数方程
float t_x, t_y, x_0, y_0;
x_0 = begin.x();
y_0 = begin.y();
// defining variables
float p1, p2, p3, p4, q1, q2, q3, q4;
p1 = -(end.x() - x_0);
p2 = -p1;
p3 = -(end.y() - y_0);
p4 = -p3;
q1 = x_0 - x_min;
q2 = x_max - x_0;
q3 = y_0 - y_min;
q4 = y_max - y_0;
float r1, r2, r3, r4;
std::vector<float> posarr;
std::vector<float> negarr;
posarr.push_back(1);
negarr.push_back(0);
// 线段全部在屏幕外
if ((p1 == 0 && q1 < 0) || (p3 == 0 && q3 < 0))
{
  line.set_upper_bound(0);
  return line;
}
// 对于非 0 的 pk, uk = qk/pk
if (p1 != 0)
{
  r1 = q1 / p1;
  r2 = q2 / p2;
  if (p1 < 0)
  {
    posarr.push_back(r2);
    negarr.push_back(r1);
  }
  else
  {
    posarr.push_back(r1);
    negarr.push_back(r2);
  }
}
if (p3 != 0)
{
```

```
    r3 = q3 / p3;
    r4 = q4 / p4;
    if (p3 < 0)
    {
      posarr.push_back(r4);
      negarr.push_back(r3);
    }
    else
    {
      posarr.push_back(r3);
      negarr.push_back(r4);
    }
  }
  float xn1, xn2, yn1, yn2;
  float rn1, rn2;
  rn1 = *std::max_element(negarr.begin(), negarr.end());
  rn2 = *std::min_element(posarr.begin(), posarr.end());
  // 线段在屏幕外
  if (rn1 > rn2)
  {
    line.set_upper_bound(0);
    return line;
  }
  xn1 = x_0 + rn1 * p2;
  yn1 = y_0 + rn1 * p4;
  xn2 = x_0 + rn2 * p2;
  yn2 = y_0 + rn2 * p4;

  line = LineSegmentParametricEquation2f(Eigen::Vector2f(xn1,
yn1),
                                         Eigen::Vector2f(xn2,
yn2));
  return line;
}
```

然后再实现中点画线法的光栅化算法

```
// 画矩形框
void draw_rectangle(Mat& img, Point p1, Point p2, Scalar color)
{
  auto x1 = p1.x, y1 = p1.y, x2 = p2.x, y2 = p2.y;
  line(img, Point(x1, y1), Point(x2, y1), color, 1);
  line(img, Point(x1, y2), Point(x2, y2), color, 1);
  line(img, Point(x1, y1), Point(x1, y2), color, 1);
  line(img, Point(x2, y1), Point(x2, y2), color, 1);
}
```

```
void draw_framework(Mat& img)
{
  auto offset_x = (WINDOW_WIDTH - FRAMEWORK_WIDTH) / 2;
  auto offset_y = (WINDOW_HEIGHT - FRAMEWORK_HEIGHT) / 2;
  draw_rectangle(img, Point(offset_x, offset_y),
                Point(offset_x + FRAMEWORK_HEIGHT, offset_y +
FRAMEWORK_WIDTH),
                Scalar(255, 255, 255));
}
// 中点画线法画线段
void draw_line(Mat& img, LineSegmentParametricEquation2f l,
Scalar color)
{
  // float x1, y1, x2, y2;
  // x1 = l.get_p0().x();
  // y1 = l.get_p0().y();
  // x2 = l.get_p1().x();
  // y2 = l.get_p1().y();
  // line(img, Point(x1, y1), Point(x2, y2), color, 1);
  auto ll = LineSegmentNormalEquation2f(l);
  if (ll.get_k() < 0)
  {
    ll = LineSegmentNormalEquation2f(l.get_p1(), l.get_p0());
  }
  if (abs(ll.get_k()) > 1)
  {
    int x1, y1, x2, y2;
    x1 = ll.get_p0().x();
    y1 = ll.get_p0().y();
    x2 = ll.get_p1().x();
    y2 = ll.get_p1().y();
    while (y1 <= y2)
    {
      img.at<Vec3b>(y1, x1) = cv::Vec3b(color[0], color[1],
color[2]);
      auto mid_x = x1 + 0.5;
      auto q = ll(mid_x, y1 + 1);
      if (q > mid_x)
      {
        y1 = y1 + 1;
        x1 = x1 + 1;
      }
      else
      {
```

```
        y1 = y1 + 1;
      }
    }
  }
  else
  {
    int x1, y1, x2, y2;
    x1 = ll.get_p0().x();
    y1 = ll.get_p0().y();
    x2 = ll.get_p1().x();
    y2 = ll.get_p1().y();

    while (x1 <= x2)
    {
      img.at<Vec3b>(y1, x1) = cv::Vec3b(color[0], color[1],
color[2]);
      auto mid_y = y1 + 0.5;
      auto q = ll(x1 + 1);
      if (q > mid_y)
      {
        x1 = x1 + 1;
        y1 = y1 + 1;
      }
      else
      {
        x1 = x1 + 1;
      }
    }
  }
}
```

实际上应该还有关于线框绘制等步骤，此处省略。

最后实现循环输入和可选的输出图像文件即可，我们这里使用一个新线程的方式来进行输入，比和绘制图形一起放在循环里灵活一些。

以下是完整代码：

```
#include <algorithm>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <eigen3/Eigen/Core>
#include <fstream>
#include <iostream>
#include <opencv2/core/types.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/opencv.hpp>
```

```cpp
#include <ostream>
#include <pthread.h>
#include <sstream>
#include <stddef.h>
#include <string>
#include <thread>
#include <utility>
#include <vector>
using namespace cv;
#define WINDOW_HEIGHT 800
#define WINDOW_WIDTH 800
#define FRAMEWORK_HEIGHT 600
#define FRAMEWORK_WIDTH 600
constexpr auto y_max =
    (WINDOW_HEIGHT - FRAMEWORK_HEIGHT) / 2 + FRAMEWORK_HEIGHT;
constexpr auto x_max = (WINDOW_WIDTH - FRAMEWORK_WIDTH) / 2 +
FRAMEWORK_WIDTH;
constexpr auto y_min = (WINDOW_HEIGHT - FRAMEWORK_HEIGHT) / 2;
constexpr auto x_min = (WINDOW_WIDTH - FRAMEWORK_WIDTH) / 2;
class LineSegmentParametricEquation2f
{
private:
  Eigen::Vector2f p0, t;
  float upper_bound, lower_bound;
```

```cpp
public:
  LineSegmentParametricEquation2f(const Eigen::Vector2f& p0,
                                  const Eigen::Vector2f& p1)
      : p0(p0)
  {
    float dx = p1.x() - p0.x();
    float dy = p1.y() - p0.y();
    t = Eigen::Vector2f(dx, dy);
    lower_bound = 0.0f;
    upper_bound = 1.0f;
  }
  LineSegmentParametricEquation2f(const Eigen::Vector3f& p0,
                                  const Eigen::Vector3f& p1)
  {
    this->p0[0] = p0[0];
    this->p0[1] = p0[1];
    float dx = p1[0] - p0[0];
    float dy = p1[1] - p0[1];
    t = Eigen::Vector2f(dy, dx);
```

```
    lower_bound = 0.0f;
    upper_bound = 1.0f;
  }
  Eigen::Vector2f operator()(float s) const { return p0 + s *
t; }
  void set_upper_bound(float upper) { upper_bound = upper; }
  void set_lower_bound(float lower) { lower_bound = lower; }
  Eigen::Vector2f get_p0() const { return p0; }
  Eigen::Vector2f get_p1() const { return p0 + t *
upper_bound; }
  void display() const
  {
    std::cout << "p0: " << p0.transpose() << std::endl;
    std::cout << "t: " << t.transpose() << std::endl;
    std::cout << "upper_bound: " << upper_bound << std::endl;
    std::cout << "lower_bound: " << lower_bound << std::endl;
  }
};
class LineSegmentNormalEquation2f
{
private:
  float a, b, c;
  float upper_bound, lower_bound;
public:
  LineSegmentNormalEquation2f(const Eigen::Vector2f& p0,
                              const Eigen::Vector2f& p1)
  {
    a = p0[1] - p1[1];                // y1 - y2
    b = p1[0] - p0[0];                // x2 - x1
    c = p0[0] * p1[1] - p1[0] * p0[1]; // x1*y2 - x2*y1
    upper_bound = p1.x();
    lower_bound = p0.x();
  }
  float operator()(float s) const
  {
    if (lower_bound == upper_bound)
    {
      return 0;
    }
    return -(a * s + c) / b;
  }
  float operator()(float x, float y) const
  {
    if (b == 0)
```

```
      return 0;
    return a * x + b * y + c;
  }
  LineSegmentNormalEquation2f(const
LineSegmentParametricEquation2f& l)
  {
    Eigen::Vector2f p0 = l.get_p0();
    Eigen::Vector2f p1 = l.get_p1();
    a = p0[1] - p1[1];
    b = p1[0] - p0[0];
    c = p0[0] * p1[1] - p1[0] * p0[1];
    upper_bound = l.get_p1().x();
    lower_bound = l.get_p0().x();
  }
  float get_a() const { return a; }
  float get_b() const { return b; }
  float get_c() const { return c; }
  float get_k() const { return -a / b; }
  Eigen::Vector2f get_p0() const
  {
    return Eigen::Vector2f(lower_bound,
this->operator()(lower_bound));
  }
  Eigen::Vector2f get_p1() const
  {
    return Eigen::Vector2f(upper_bound,
this->operator()(upper_bound));
  }
};
LineSegmentParametricEquation2f
Liang_Barsky(Eigen::Vector2f& begin,
                                             Eigen::Vector2f&
end)
{
  LineSegmentParametricEquation2f line(begin, end);
  // 计算参数方程
  float t_x, t_y, x_0, y_0;
  x_0 = begin.x();
  y_0 = begin.y();
  // defining variables
  float p1, p2, p3, p4, q1, q2, q3, q4;
  p1 = -(end.x() - x_0);
  p2 = -p1;
  p3 = -(end.y() - y_0);
```

```
p4 = -p3;
q1 = x_0 - x_min;
q2 = x_max - x_0;
q3 = y_0 - y_min;
q4 = y_max - y_0;
float r1, r2, r3, r4;
std::vector<float> posarr;
std::vector<float> negarr;
posarr.push_back(1);
negarr.push_back(0);
// 线段全部在屏幕外
if ((p1 == 0 && q1 < 0) || (p3 == 0 && q3 < 0))
{
  line.set_upper_bound(0);
  return line;
}
// 对于非0的pk, uk = qk/pk
if (p1 != 0)
{
  r1 = q1 / p1;
  r2 = q2 / p2;
  if (p1 < 0)
  {
    posarr.push_back(r2);
    negarr.push_back(r1);
  }
  else
  {
    posarr.push_back(r1);
    negarr.push_back(r2);
  }
}
if (p3 != 0)
{
  r3 = q3 / p3;
  r4 = q4 / p4;
  if (p3 < 0)
  {
    posarr.push_back(r4);
    negarr.push_back(r3);
  }
  else
  {
    posarr.push_back(r3);
```

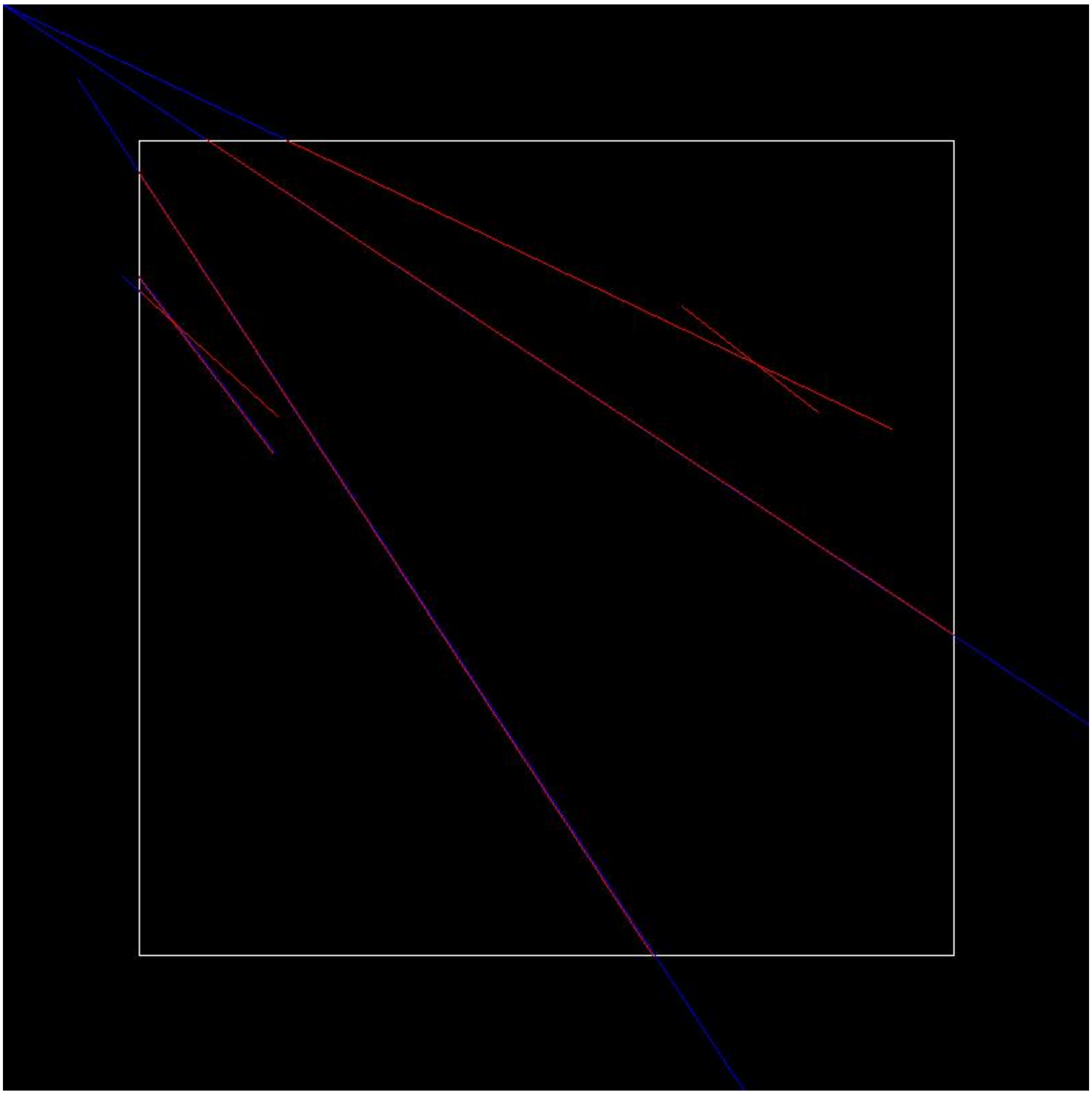
```
      negarr.push_back(r4);
    }
  }
  float xn1, xn2, yn1, yn2;
  float rn1, rn2;
  rn1 = *std::max_element(negarr.begin(), negarr.end());
  rn2 = *std::min_element(posarr.begin(), posarr.end());
  // 线段在屏幕外
  if (rn1 > rn2)
  {
    line.set_upper_bound(0);
    return line;
  }
  xn1 = x_0 + rn1 * p2;
  yn1 = y_0 + rn1 * p4;
  xn2 = x_0 + rn2 * p2;
  yn2 = y_0 + rn2 * p4;
  line =
LineSegmentParametricEquation2f(Eigen::Vector2f(xn1, yn1),
                                       Eigen::Vector2f(xn2,
yn2));
  return line;
}
// 画矩形框
void draw_rectangle(Mat& img, Point p1, Point p2, Scalar color)
{
  auto x1 = p1.x, y1 = p1.y, x2 = p2.x, y2 = p2.y;
  line(img, Point(x1, y1), Point(x2, y1), color, 1);
  line(img, Point(x1, y2), Point(x2, y2), color, 1);
  line(img, Point(x1, y1), Point(x1, y2), color, 1);
  line(img, Point(x2, y1), Point(x2, y2), color, 1);
}
void draw_framework(Mat& img)
{
  auto offset_x = (WINDOW_WIDTH - FRAMEWORK_WIDTH) / 2;
  auto offset_y = (WINDOW_HEIGHT - FRAMEWORK_HEIGHT) / 2;
  draw_rectangle(img, Point(offset_x, offset_y),
                 Point(offset_x + FRAMEWORK_HEIGHT, offset_y +
FRAMEWORK_WIDTH),
                 Scalar(255, 255, 255));
}
// 中点画线法画线段
void draw_line(Mat& img, LineSegmentParametricEquation2f l,
Scalar color)
```

```
{
  // float x1, y1, x2, y2;
  // x1 = l.get_p0().x();
  // y1 = l.get_p0().y();
  // x2 = l.get_p1().x();
  // y2 = l.get_p1().y();
  // line(img, Point(x1, y1), Point(x2, y2), color, 1);
  auto ll = LineSegmentNormalEquation2f(l);
  if (ll.get_k() < 0)
  {
    ll = LineSegmentNormalEquation2f(l.get_p1(), l.get_p0());
  }
  if (abs(ll.get_k()) > 1)
  {
    int x1, y1, x2, y2;
    x1 = ll.get_p0().x();
    y1 = ll.get_p0().y();
    x2 = ll.get_p1().x();
    y2 = ll.get_p1().y();
    while (y1 <= y2)
    {
      img.at<Vec3b>(y1, x1) = cv::Vec3b(color[0], color[1],
color[2]);
      auto mid_x = x1 + 0.5;
      auto q = ll(mid_x, y1 + 1);
      if (q > mid_x)
      {
        y1 = y1 + 1;
        x1 = x1 + 1;
      }
      else
      {
        y1 = y1 + 1;
      }
    }
  }
  else
  {
    int x1, y1, x2, y2;
    x1 = ll.get_p0().x();
    y1 = ll.get_p0().y();
    x2 = ll.get_p1().x();
    y2 = ll.get_p1().y();
```

```
    while (x1 <= x2)
    {
      img.at<Vec3b>(y1, x1) = cv::Vec3b(color[0], color[1],
color[2]);
      auto mid_y = y1 + 0.5;
      auto q = ll(x1 + 1);
      if (q > mid_y)
      {
        x1 = x1 + 1;
        y1 = y1 + 1;
      }
      else
      {
        x1 = x1 + 1;
      }
    }
  }
}
std::vector<LineSegmentParametricEquation2f> lines;
std::vector<std::pair<Point, Point>> lines_without_clip;
bool is_end = false;
void input_lines()
{
  while (true)
  {
    float x1, y1, x2, y2;
    if (std::cin.peek() != EOF)
    {
      std::string s;
      std::getline(std::cin, s);
      std::stringstream ss(s);
      ss >> x1 >> y1 >> x2 >> y2;
      lines_without_clip.push_back(
          std::make_pair(Point(x1, y1), Point(x2, y2)));
      Eigen::Vector2f begin(x1, y1);
      Eigen::Vector2f end(x2, y2);
      LineSegmentParametricEquation2f l = Liang_Barsky(begin,
end);
      lines.push_back(l);
    }
    if (is_end)
    {
      break;
    }
```

```cpp
  }
}
int main(int argc, char** argv)
{
  std::string output_file_name;
  for (int i = 0; i < argc; i++)
  {
    if (strcmp(argv[i], "-o") == 0)
    {
      output_file_name = std::string(argv[i + 1]);
    }
  }
  Mat img = Mat::zeros(WINDOW_HEIGHT, WINDOW_WIDTH, CV_8UC3);
  std::thread input_thread(input_lines);
  input_thread.detach();
  while (true)
  {
    img.setTo(Scalar(0, 0, 0));
    draw_framework(img);
    for (auto& l : lines_without_clip)
    {
      line(img, l.first, l.second, Scalar(255, 0, 0));
    }
    for (auto& l : lines)
    {
      draw_line(img, l, Scalar(0, 0, 255));
    }
    imshow("line", img);
    if (waitKey(1000 / 60) == 27)
    {
      break;
    }
  }
  is_end = true;
  if (!output_file_name.empty())
  {
    imwrite(output_file_name, img);
  }
  return 0;
}
```

实验结果如下图：



关于这张图片，白色线框划定的区域是我们截取线段的区域，蓝线是这个线段被截取之前的绘制情况，使用的是 OpenCV 的 line 函数，红线是截取后绘制的部分，使用的是自己实现的中点画线法光栅化函数，紫色是像素在渲染的时候贴的过近而又不重合的视觉效果，旨在表明中点画线法和 line 函数使用的光栅化渲染方式的不同之处。

实验中存在的问题及解决：

问题 1：我们在对 OpenCV 的像素进行着色的时候，为什么 img.at<Vec3b>(x1, y1) = cv::Vec3b(color[0], color[1], color[2]);绘制的位置和我们真实想要的位置是关于 y=x 这条直线对称的？

回答 1：因为历史遗留原因（？），或者说 OpenCV 定义的原因，在指定像素的时候是先指定行数，再指定列数的，而不像绘制点那样是按照 X 轴 Y 轴的顺序，因此想要正确绘制出我们想要的像素，需要写成 img.at<Vec3b>(y1, x1) = cv::Vec3b(color[0], color[1], color[2]);