计算机学院实验报告

实验题目： Catmull-Clark 细分		学号：202300130183
日期：2025/4/7	班级：23 级智能班	姓名：宋浩宇

Email：2367651943@qq.com
202300130183@mail.sdu.edu.cn
zhitian0420@gmai.com

实验目的：
希望同学们通过本实验锻炼系统能力，实验具体模块如下：
1，功能模块：数据录入，数据结构、细分算法、可视呈现；
2，数据录入：obj 格式，实现基本的文件读写功能；
3，数据结构：就是一个图结构，点和边分别存储，需要用到查询一邻域的方法；
4，细分算法：Catmull 细分方法；
5，可视呈现：基于 opencv 绘制线段，在二维屏幕上呈现细分的迭代过程，或其他任何可视
化手段（libigl, matlab, 等等都可以）；

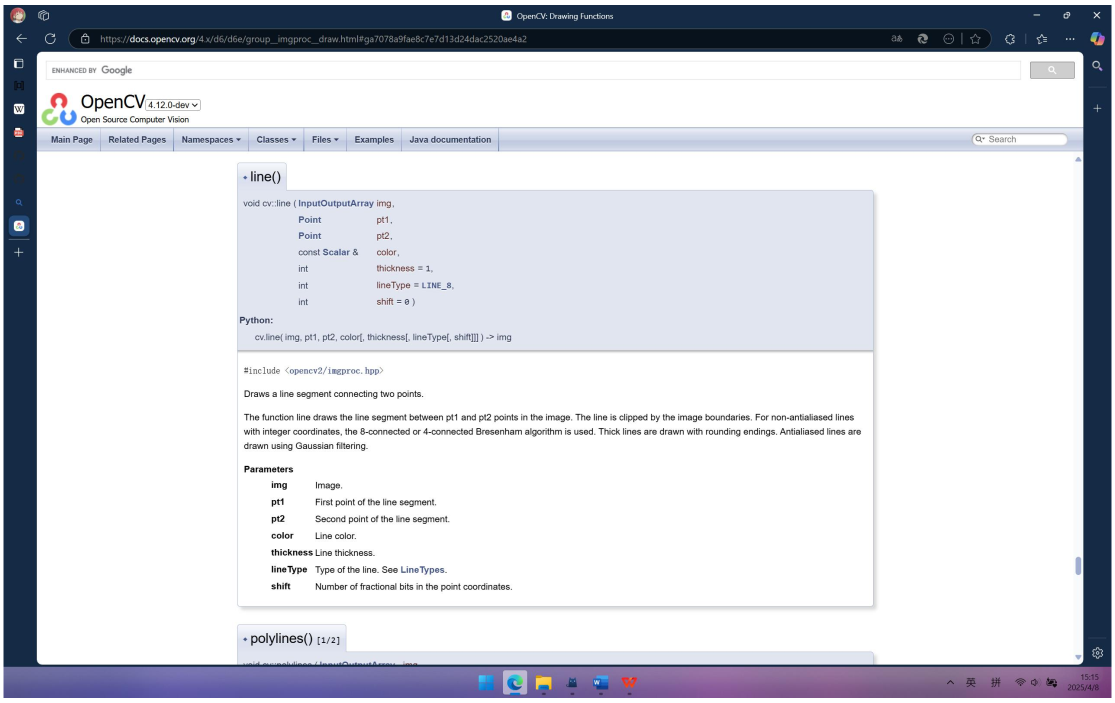
实验环境介绍：
软件环境：
主系统：Windows 11 家庭中文版 23H2 22631.4317
虚拟机软件：Oracle Virtual Box 7.1.6
虚拟机系统：Ubuntu 18.04.2 LTS
编辑器：Visual Studio Code
编译器：gcc 7.3.0
计算框架：Eigen 3.3.7

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：

本实验的解决思路如下：

首先我们需要熟悉 opencv 提供的绘制点和直线和函数：


OpenCV: Drawing Functions
https://docs.opencv.org/4.x/d6/d6e/group__imgproc__draw.html#ga7078a9fae8c7e7d13d24dac2520ae4a2
ENHANCED BY Google
OpenCV 4.12.0-dev
Open Source Computer Vision
Main Page
Related Pages
Namespaces
Classes
Files
Examples
Java documentation
Search

◆ line()

```
void cv::line ( InputOutputArray img,
                Point            pt1,
                Point            pt2,
                const Scalar &   color,
                int              thickness = 1,
                int              lineType = LINE_8,
                int              shift = 0 )
Python:
    cv.line( img, pt1, pt2, color[, thickness[, lineType[, shift]]] ) -> img
```

`#include <opencv2/imgproc.hpp>`

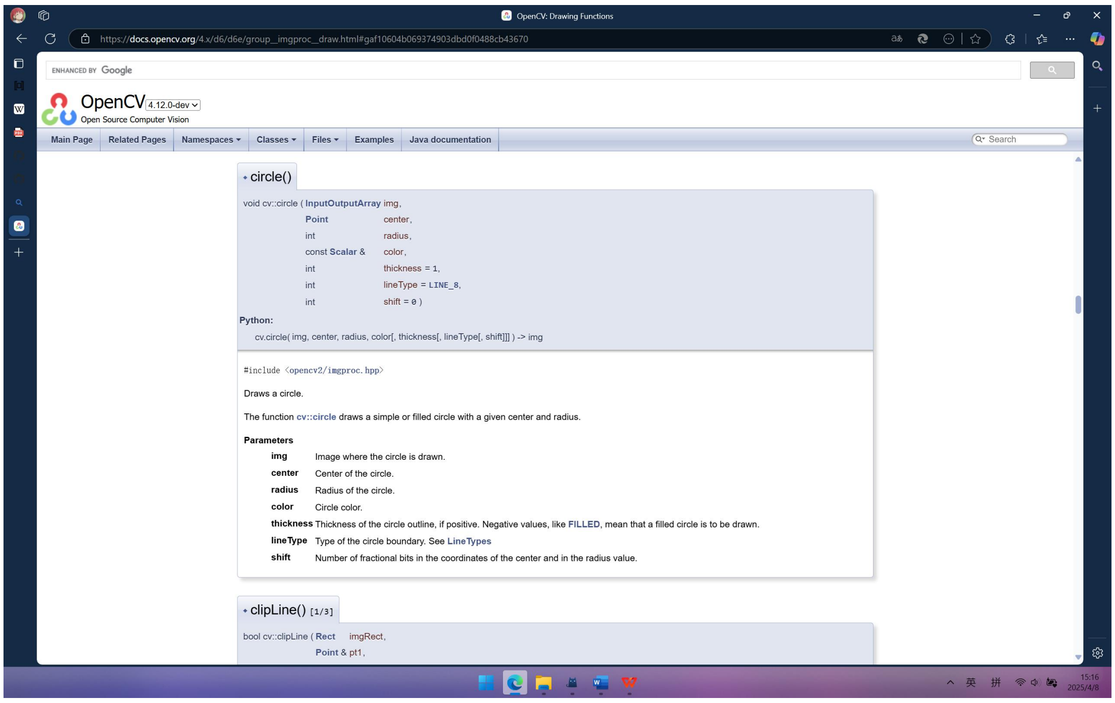
Draws a line segment connecting two points.

The function line draws the line segment between pt1 and pt2 points in the image. The line is clipped by the image boundaries. For non-antialiased lines with integer coordinates, the 8-connected or 4-connected Bresenham algorithm is used. Thick lines are drawn with rounding endings. Antialiased lines are drawn using Gaussian filtering.

Parameters

img	Image.
pt1	First point of the line segment.
pt2	Second point of the line segment.
color	Line color.
thickness	Line thickness.
lineType	Type of the line. See LineTypes.
shift	Number of fractional bits in the point coordinates.

◆ polylines() [1/2]


OpenCV: Drawing Functions
https://docs.opencv.org/4.x/d6/d6e/group__imgproc__draw.html#gaf10604b069374903dbd0f0488cb43670
ENHANCED BY Google
OpenCV 4.12.0-dev
Open Source Computer Vision
Main Page
Related Pages
Namespaces
Classes
Files
Examples
Java documentation
Search

◆ circle()

```
void cv::circle ( InputOutputArray img,
                  Point            center,
                  int              radius,
                  const Scalar &   color,
                  int              thickness = 1,
                  int              lineType = LINE_8,
                  int              shift = 0 )
Python:
    cv.circle( img, center, radius, color[, thickness[, lineType[, shift]]] ) -> img
```

`#include <opencv2/imgproc.hpp>`

Draws a circle.

The function cv::circle draws a simple or filled circle with a given center and radius.

Parameters

img	Image where the circle is drawn.
center	Center of the circle.
radius	Radius of the circle.
color	Circle color.
thickness	Thickness of the circle outline, if positive. Negative values, like FILLED, mean that a filled circle is to be drawn.
lineType	Type of the circle boundary. See LineTypes
shift	Number of fractional bits in the coordinates of the center and in the radius value.

◆ clipLine() [1/3]

```
bool cv::clipLine ( Rect    imgRect,
                    Point & pt1,
```

这决定了我们把结果可视化的方式。

然后就是 Catmull-Clark 算法实现的具体方式了。

Catmull-Clark 算法可以概括为以下几个步骤：

1. 计算面点。
2. 计算边点。
3. 更新原有点。
4. 用新点构成新的边和面。

5. 更新全局点集、边集、面集。

我们依次来讲这些部分都是怎么计算的。

1. 计算面点

   对于每一个面 F，假设构成这个面的点集合为$\{x|x \in F\}$，那我们的面点P_F就是：

$$P_{F_i} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

2. 计算边点

   对于每一个边 E，假设构成这个边的点集合为$\{x|x \in E\}$，和这个边相邻的面为$\{F|E \in F\}$，易得面最多有两个。我们用P_{Em}表示这个边的中点，那我们可以得到边点：

$$P_{E_i} = \frac{\frac{P_{F_1} + P_{F_2}}{2} + P_{Em_i}}{2}$$

3. 更新原有点

   对于每一个点，我们可以得到它相邻的面和边，分别为$\{F|P \in F\}$、$\{E|P \in E\}$

   为了方便公式的书写，我们使用 F 表示和它相邻的面的面点的均值，用 R 表示和它相邻的边的中点的均值，用公式表示为：

$$F = \frac{\sum_{i=1}^{n} P_{F_i}}{n}$$

$$R = \frac{\sum_{i=1}^{n} P_{Em_i}}{n}$$

4. 构成新的面

   用更新后的点、面点、边点构成新的面。

实验步骤与实验结果：

首先是实现 OBJ 文件数据的输入。

我们将程序写为了可以通过启动参数来设置导入的文件。

然后为了绘制出来的点能够看起来更清楚，我们给这些点都设置上 scale 和 offset，以便于将点绘制在画布靠中间的位置并且能让他们不至于距离太近。

然后就是实现 Catmull-Clark 细分算法。

然后为了让程序更灵活一些，我们加入可调整读入文件的部分，以及可以控制细分次数的部分。

以下为代码：

```
#include <algorithm>
#include <cstdlib>
#include <cstring>
#include <ctime>
```

```
#include <fstream>
#include <iostream>
#include <opencv2/highgui.hpp>
#include <opencv2/opencv.hpp>
#include <ostream>
#include <sstream>
#include <stddef.h>
#include <string>
#include <vector>
using namespace cv;

class my_point
{
public:
  double x, y;
  size_t v_id;
};
class my_edge
{
public:
  size_t edge_id;
  size_t v1_id, v2_id;
};
class my_face
{
public:
  size_t face_id;
  std::vector<size_t> points;
  std::vector<size_t> edges;
};
std::vector<my_point> all_points;
std::vector<my_edge> all_edges;
std::vector<my_face> all_faces;
#define SCALE 200.0
#define OFFSET 100.0
// // 设置窗口
//    // 注意 opencv 的坐标系原点在左上角
// Mat img = Mat::zeros(Size(800, 800), CV_8UC3);
// img.setTo(255);              // 设置屏幕为白色
//
// Point p1(100, 100);          // 点 p1
// Point p2(758, 50);           // 点 p2
//
// // 画直线函数
// line(img, p1, p2, Scalar(0, 0, 255), 1);   // 红色
// line(img, Point(300, 300), Point(758, 400), Scalar(0, 255,
255), 1); //
// 黄色
//
//    // 画点 p1
// circle(img, p1, 3, Scalar(0, 255, 0), -1);
//    // 画点 p2
// circle(img, p2, 3, Scalar(120, 120,sum 120), -1);
//
// imshow("画板", img);
// waitKey(0);
void read_file(const char* filename)
{
  std::ifstream file(filename);
  if (!file.is_open())
  {
    std::cout << "Error: cannot open file " << filename <<
std::endl;
    return;
  }
  size_t point_index = 0;
  size_t edge_index = 0;
  size_t face_index = 0;
```

```
  char type;
  while (file >> type)
  {
    if (type == 'v')
    {
      my_point p;
      point_index++;
      file >> p.x >> p.y;
      p.v_id = point_index;
      p.x *= SCALE;
      p.y *= SCALE;
      p.x += OFFSET;
      p.y += OFFSET;
      all_points.push_back(p);
    }
    if (type == 'e')
    {
      my_edge e;
      edge_index++;
```

```cpp
      file >> e.v1_id >> e.v2_id;
      e.edge_id = edge_index;
      all_edges.push_back(e);
    }
    if (type == 'f')
    {
      my_face f;
      face_index++;
      std::string str;
      std::getline(file, str);
      std::stringstream ss(str);
      // size_t v1, v2, v3, v4;
      // file >> v1 >> v2 >> v3 >> v4;
      // f.face_id = face_index;
      // f.points.push_back(v1);
      // f.points.push_back(v2);
      // f.points.push_back(v3);
      // f.points.push_back(v4);
      // for (auto& i : all_edges)
      // {
      //   if (i.v1_id == v1 && i.v2_id == v2)
      //   {
      //     f.edges.push_back(i.edge_id);
      //   }
      //   if (i.v1_id == v2 && i.v2_id == v3)
      //   {
      //     f.edges.push_back(i.edge_id);
      //   }
      //   if (i.v1_id == v3 && i.v2_id == v4)
      //   {
      //     f.edges.push_back(i.edge_id);
      //   }
      //   if (i.v1_id == v4 && i.v2_id == v1)
      //   {
      //     f.edges.push_back(i.edge_id);
      //   }
      // }
      f.face_id = face_index;
      size_t buf;
      while (ss >> buf)
      {
        // std::cout << buf << std::endl;
        f.points.push_back(buf);
      }
      for (size_t i = 0; i < f.points.size(); i++)
      {
        auto edge = std::find_if(
            all_edges.begin(), all_edges.end(),
            [f, i](const my_edge& e)
            {
              return (e.v1_id == f.points[i] &&
                      e.v2_id == f.points[(i + 1) %
f.points.size()]) ||
                     (e.v1_id == f.points[(i + 1) %
f.points.size()] &&
                      e.v2_id == f.points[i]);
            });
        f.edges.push_back(edge->edge_id);
      }
      all_faces.push_back(f);
    }
  }
}
void draw_all(Mat& img)
{
  // 画点
  for (auto& i : all_points)
  {
    circle(img, Point(i.x, i.y), 3, Scalar(0, 255, 0), -1);
  }
  // 画边
  for (auto& i : all_edges)
  {
    auto point_1 =
        std::find_if(all_points.begin(), all_points.end(),
                     [i](const my_point& p) { return p.v_id ==
i.v1_id; });
    auto point_2 =
        std::find_if(all_points.begin(), all_points.end(),
                     [i](const my_point& p) { return p.v_id ==
i.v2_id; });
    line(img, Point(point_1->x, point_1->y),
Point(point_2->x, point_2->y),
         Scalar(0, 0, 255), 1);
  }
  // 画面
}
void display_all()
```

```
{
  for (auto& i : all_points)
  {
    std::cout << "点" << i.v_id << "(" << i.x << "," << i.y <<
")" << std::endl;
  }
  for (auto& i : all_edges)
  {
    std::cout << "边" << i.edge_id << "(" << i.v1_id << "," <<
i.v2_id << ")"
              << std::endl;
  }
  for (auto& i : all_faces)
  {
    std::cout << "面" << i.face_id << std::endl;
    for (auto& j : i.points)
    {
      std::cout << "点" << j << " " << std::endl;
    }
    for (auto& j : i.edges)
    {
      std::cout << "边" << j << " " << std::endl;
    }
  }
}
void display_faces(std::vector<my_face>& faces)
{
  std::cout << "共有" << faces.size() << "个面" << std::endl;
  for (auto& i : faces)
  {
    std::cout << "面" << i.face_id << std::endl;
    std::cout << "点" << i.points.size() << std::endl;
    for (auto& j : i.points)
    {
      auto point = std::find_if(all_points.begin(),
all_points.end(),
                                [j](const my_point& p) { return
p.v_id == j; });
      std::cout << "(" << point->x << "," << point->y << ")"
<< std::endl;
    }
    std ::cout << "边" << i.edges.size() << std::endl;
    for (auto& j : i.edges)
    {
      auto edge =
          std::find_if(all_edges.begin(), all_edges.end(),
                       [j](const my_edge& e) { return e.edge_id
== j; });
      std::cout << "(" << edge->v1_id << "," << edge->v2_id <<
")" << std::endl;
    }
  }
}
void display_points(Mat& img, std::vector<my_point>& points)
{
  std::srand(std::time(0));
  double r = std::rand() % 256;
  double g = std::rand() % 256;
  double b = std::rand() % 256;
  for (auto& i : points)
  {
    circle(img, Point(i.x, i.y), 3, Scalar(b, g, r), -1);
  }
}
void bind_edges()
{
  for (auto& f : all_faces)
  {
    for (size_t i = 0; i < f.points.size(); i++)
    {
      auto edge = std::find_if(
          all_edges.begin(), all_edges.end(),
          [f, i](const my_edge& e)
          {
            return (e.v1_id == f.points[i] &&
                    e.v2_id == f.points[(i + 1) %
f.points.size()]) ||
                   (e.v1_id == f.points[(i + 1) %
f.points.size()] &&
                    e.v2_id == f.points[i]);
          });
      f.edges.push_back(edge->edge_id);
    }
  }
}
void catmull_clark()
{
  // 面点
  std::vector<my_point> face_points;
  for (auto& i : all_faces)
  {
    double sum_x = 0.0;
    double sum_y = 0.0;
    size_t cnt = i.points.size();
    for (auto& j : i.points)
    {
      auto point = std::find_if(all_points.begin(),
all_points.end(),
                                [j](const my_point& p) { return
p.v_id == j; });
      sum_x += point->x;
      sum_y += point->y;
    }
    my_point p;
    p.x = sum_x / cnt;
    p.y = sum_y / cnt;
    p.v_id = i.face_id;
    face_points.push_back(p);
  }
  // std::cout << face_points.size() << std::endl;
  // for (auto& i : face_points)
  // {
  //   std::cout << "点" << i.v_id << "(" << i.x << "," << i.y
<< ")" <<
  //   std::endl;
  // }
  // 边中点
  std::vector<my_point> edge_avg_points;
  for (auto& i : all_edges)
  {
    auto point_1 =
        std::find_if(all_points.begin(), all_points.end(),
                     [i](const my_point& p) { return p.v_id ==
i.v1_id; });
    auto point_2 =
        std::find_if(all_points.begin(), all_points.end(),
                     [i](const my_point& p) { return p.v_id ==
i.v2_id; });
    my_point p;
    p.x = (point_1->x + point_2->x) / 2.0;
    p.y = (point_1->y + point_2->y) / 2.0;
    p.v_id = i.edge_id;
    edge_avg_points.push_back(p);
  }
  // 边点
  std::vector<my_point> edge_points;
  for (auto& i : all_edges)
  {
    auto start = all_faces.begin();
    my_point sum;
    size_t cnt = 0;
    auto face_1 = std::find_if(
        start, all_faces.end(),
        [i](const my_face& f)
        {
          return find_if(f.edges.begin(), f.edges.end(),
[i](const size_t& e)
                              { return e == i.edge_id; }) !=
f.edges.end();
        });
    start = face_1 + 1;
    auto face_2 = std::find_if(
        start, all_faces.end(),
        [i](const my_face& f)
        {
          return find_if(f.edges.begin(), f.edges.end(),
[i](const size_t& e)
                              { return e == i.edge_id; }) !=
f.edges.end();
        });
    if (face_1 != all_faces.end())
    {
      auto point = std::find_if(face_points.begin(),
face_points.end(),
                                     [face_1](const my_point& p)
                                     { return p.v_id ==
face_1->face_id; });
      // std::cout << "面的中点" << face_1->face_id << "(" <<
point->x << ","
      //            << point->y << ")" << std::endl;
      sum.x = point->x;
      sum.y = point->y;
      cnt++;
    }
    if (face_2 != all_faces.end())
    {
      auto point = std::find_if(face_points.begin(),
face_points.end(),
                               [face_2](const my_point& p)
                               { return p.v_id ==
face_2->face_id; });
      // std::cout << "面的中点" << face_2->face_id << "(" <<
point->x << ","
      //          << point->y << ")" << std::endl;
      sum.x += point->x;
      sum.y += point->y;
      cnt++;
    }
    sum.x /= cnt;
    sum.y /= cnt;
    sum.x *= 0.5;
    sum.y *= 0.5;
    sum.x +=
        std::find_if(edge_avg_points.begin(),
edge_avg_points.end(),
                     [i](const my_point& e) { return i.edge_id
== e.v_id; })
            ->x *
        0.5;
    sum.y +=
        std::find_if(edge_avg_points.begin(),
edge_avg_points.end(),
                     [i](const my_point& e) { return i.edge_id
== e.v_id; })
            ->y *
        0.5;
    sum.v_id = i.edge_id;
    edge_points.push_back(sum);
  }
  // 更新点坐标
  std::vector<my_point> new_points;
  for (auto& i : all_points)
  {
    // 接触的面的个数
    long long n = 0;
    std::vector<my_face> faces;
    for (auto& j : all_faces)
    {
      if (std::find_if(j.points.begin(), j.points.end(),
[i](const size_t& p)
```

```
                          { return p == i.v_id; }) !=
j.points.end())
      {
        n++;
        faces.push_back(j);
      }
    }
    // std::cout << "点" << i.v_id << "接触面的个数" << n <<
std::endl;
    // 面点均值
    my_point face_sum;
    face_sum.x = 0.0;
    face_sum.y = 0.0;
    for (auto& j : faces)
    {
      auto point =
          *std::find_if(face_points.begin(),
face_points.end(),
                          [j](const my_point& p) { return p.v_id
== j.face_id; });
      face_sum.x += point.x;
      face_sum.y += point.y;
      // std::cout << "面的点" << j.face_id << "(" << point.x
<< "," << point.y
      //           << ")" << std::endl;
    }
    face_sum.x /= n * 1.0;
    face_sum.y /= n * 1.0;
    // std::cout << "面的中点" << face_sum.v_id << "(" <<
face_sum.x << ","
    //           << face_sum.y << ")" << std::endl;
    // 边中点均值
    my_point edge_sum;
    edge_sum.x = 0.0;
    edge_sum.y = 0.0;
    long long cnt = 0;
    std::vector<my_edge> edges;
    for (auto& j : all_edges)
    {
      if (j.v1_id == i.v_id || j.v2_id == i.v_id)
      {
        edges.push_back(j);
        cnt++;
      }
    }
    for (auto& j : edges)
    {
      auto point =
          *std::find_if(edge_avg_points.begin(),
edge_avg_points.end(),
                        [j](const my_point& p) { return p.v_id
== j.edge_id; });
      edge_sum.x += point.x;
      edge_sum.y += point.y;
    }
    edge_sum.x /= cnt * 1.0;
    edge_sum.y /= cnt * 1.0;
    // std::cout << "边的中点" << edge_sum.v_id << "(" <<
edge_sum.x << ","
    //            << edge_sum.y << ")" << std::endl;
    // 新的点坐标
    my_point new_point;
    // std::cout << "旧的点" << i.v_id << "(" << i.x << "," <<
i.y << ")"
    //            << std::endl;
    new_point.x = (n - 3) * i.x + 2.0 * edge_sum.x + face_sum.x;
    new_point.y = (n - 3) * i.y + 2.0 * edge_sum.y + face_sum.y;
    new_point.x /= n * 1.0;
    new_point.y /= n * 1.0;
    new_point.v_id = i.v_id;
    // std::cout << "新的点" << new_point.v_id << "(" <<
new_point.x << ","
    //            << new_point.y << ")" << std::endl;
    if (cnt != n)
    {
      new_point.x = i.x;
      new_point.y = i.y;
    }
    new_points.push_back(new_point);
  }
  // for (auto& i : new_points)
  // {
  //   std::cout << "新的点" << i.v_id << "(" << i.x << "," <<
i.y << ")"
  //              << std::endl;
  // }
  // for (auto& i : all_points)
  // {
```

```
  //   std::cout << "旧的点" << i.v_id << "(" << i.x << "," <<
i.y << ")"
  //              << std::endl;
  // }
  // for (auto& i : all_faces)
  // {
  //   std::cout << "面的点的索引" << i.face_id << std::endl;
  //   for (auto& j : i.points)
  //   {
  //     std::cout << j << " ";
  //   }
  //   std::cout << std::endl;
  // }
  // for (auto& i : face_points)
  // {
  //   std::cout << "面点" << i.v_id << "(" << i.x << "," << i.y
<< ")"
  //              << std::endl;
  // }
  // Mat img = Mat::zeros(Size(800, 800), CV_8UC3);
  // img.setTo(255); // 设置屏幕为白色
  // display_points(img, new_points);
  // display_points(img, face_points);
  // display_points(img, edge_points);
  // imshow("花瓣", img);
  // waitKey(0);
  // 更新点边和面，按面更新
  std::vector<my_point> new_points_tmp;
  std::vector<my_edge> new_edges;
  std::vector<my_face> new_faces;
  new_points_tmp.push_back(all_points[0]);
  new_faces.push_back(all_faces[0]);
  new_edges.push_back(all_edges[0]);
```

```
  for (auto& i : all_faces)
  {
    if (i.points.size() == 4)
    {
      auto point_1 = *std::find_if(new_points.begin(),
new_points.end(),
                                   [i](const my_point& p)
                                   { return p.v_id ==
i.points[0]; });
      auto point_2 = *std::find_if(new_points.begin(),
new_points.end(),
                                [i](const my_point& p)
                                { return p.v_id ==
i.points[1]; });
      auto point_3 = *std::find_if(new_points.begin(),
new_points.end(),
                                [i](const my_point& p)
                                { return p.v_id ==
i.points[2]; });
      auto point_4 = *std::find_if(new_points.begin(),
new_points.end(),
                                [i](const my_point& p)
                                { return p.v_id ==
i.points[3]; });
      // std::cout << "四边形面" << i.face_id << std::endl;
      // std::cout << "point_1:" << "(" << point_1.x << "," <<
point_1.y << ")"
      //           << std::endl;
      // std::cout << "point_2:" << "(" << point_2.x << "," <<
point_2.y << ")"
      //           << std::endl;
      // std::cout << "point_3:" << "(" << point_3.x << "," <<
point_3.y << ")"
      //           << std::endl;
      // std::cout << "point_4:" << "(" << point_4.x << "," <<
point_4.y << ")"
      //           << std::endl;
      auto edge_1_2_index = *std::find_if(
          all_edges.begin(), all_edges.end(),
          [point_1, point_2](const my_edge& e)
          {
            return (e.v1_id == point_1.v_id && e.v2_id ==
point_2.v_id) ||
                   (e.v1_id == point_2.v_id && e.v2_id ==
point_1.v_id);
          });
      auto edge_point_1_2 =
          *std::find_if(edge_points.begin(),
edge_points.end(),
                        [edge_1_2_index](const my_point& p)
                        { return p.v_id ==
edge_1_2_index.edge_id; });
      auto edge_2_3_index = *std::find_if(
          all_edges.begin(), all_edges.end(),
          [point_2, point_3](const my_edge& e)
          {
            return (e.v1_id == point_2.v_id && e.v2_id ==
point_3.v_id) ||
                   (e.v1_id == point_3.v_id && e.v2_id ==
point_2.v_id);
          });
      auto edge_point_2_3 =
          *std::find_if(edge_points.begin(),
edge_points.end(),
                        [edge_2_3_index](const my_point& p)
                        { return p.v_id ==
edge_2_3_index.edge_id; });
      auto edge_3_4_index = *std::find_if(
          all_edges.begin(), all_edges.end(),
          [point_3, point_4](const my_edge& e)
          {
            return (e.v1_id == point_3.v_id && e.v2_id ==
point_4.v_id) ||
                   (e.v1_id == point_4.v_id && e.v2_id ==
point_3.v_id);
          });
      auto edge_point_3_4 =
          *std::find_if(edge_points.begin(),
edge_points.end(),
                        [edge_3_4_index](const my_point& p)
                        { return p.v_id ==
edge_3_4_index.edge_id; });
      auto edge_4_1_index = *std::find_if(
          all_edges.begin(), all_edges.end(),
          [point_4, point_1](const my_edge& e)
          {
            return (e.v1_id == point_4.v_id && e.v2_id ==
point_1.v_id) ||
                   (e.v1_id == point_1.v_id && e.v2_id ==
point_4.v_id);
          });
      auto edge_point_4_1 =
          *std::find_if(edge_points.begin(),
edge_points.end(),
                        [edge_4_1_index](const my_point& p)
                        { return p.v_id ==
edge_4_1_index.edge_id; });
      auto face_point =
          *std::find_if(face_points.begin(),
face_points.end(),
                        [i](const my_point& p) { return p.v_id
== i.face_id; });
      point_1.v_id = new_points_tmp.size();
      new_points_tmp.push_back(point_1);
      point_2.v_id = new_points_tmp.size();
      new_points_tmp.push_back(point_2);
      point_3.v_id = new_points_tmp.size();
      new_points_tmp.push_back(point_3);
      point_4.v_id = new_points_tmp.size();
      new_points_tmp.push_back(point_4);
      edge_point_1_2.v_id = new_points_tmp.size();
      new_points_tmp.push_back(edge_point_1_2);
      edge_point_2_3.v_id = new_points_tmp.size();
      new_points_tmp.push_back(edge_point_2_3);
      edge_point_3_4.v_id = new_points_tmp.size();
      new_points_tmp.push_back(edge_point_3_4);
      edge_point_4_1.v_id = new_points_tmp.size();
      new_points_tmp.push_back(edge_point_4_1);
      face_point.v_id = new_points_tmp.size();
      new_points_tmp.push_back(face_point);
      my_face new_face_1;
      my_face new_face_2;
      my_face new_face_3;
      my_face new_face_4;
      my_edge new_edge_1_1;
      my_edge new_edge_1_2;
      my_edge new_edge_1_3;
      my_edge new_edge_1_4;
      my_edge new_edge_2_1;
      my_edge new_edge_2_2;
      my_edge new_edge_2_3;
      my_edge new_edge_2_4;
      my_edge new_edge_3_1;
      my_edge new_edge_3_2;
      my_edge new_edge_3_3;
      my_edge new_edge_3_4;
      my_edge new_edge_4_1;
      my_edge new_edge_4_2;
      my_edge new_edge_4_3;
      my_edge new_edge_4_4;
      // 面 1(a,edge_point_ab,face_point,edge_point_da)
      // 边 1(a,edge_point_ab)
```

```
      new_edge_1_1.v1_id = point_1.v_id;
      new_edge_1_1.v2_id = edge_point_1_2.v_id;
      // 边 2(edge_point_ab,face_point)
      new_edge_1_2.v1_id = edge_point_1_2.v_id;
      new_edge_1_2.v2_id = face_point.v_id;
      // 边 3(face_point,edge_point_da)
      new_edge_1_3.v1_id = face_point.v_id;
      new_edge_1_3.v2_id = edge_point_4_1.v_id;
      // 边 4(edge_point_da,a)
      new_edge_1_4.v1_id = edge_point_4_1.v_id;
      new_edge_1_4.v2_id = point_1.v_id;
      // 新增边
      new_edge_1_1.edge_id = new_edges.size();
      new_edges.push_back(new_edge_1_1);
      new_edge_1_2.edge_id = new_edges.size();
      new_edges.push_back(new_edge_1_2);
      new_edge_1_3.edge_id = new_edges.size();
      new_edges.push_back(new_edge_1_3);
      new_edge_1_4.edge_id = new_edges.size();
      new_edges.push_back(new_edge_1_4);
```

```
      // 新增面
      new_face_1.points = {point_1.v_id, edge_point_1_2.v_id,
face_point.v_id,
                          edge_point_4_1.v_id};
      // new_face_1.edges = {new_edge_1_1.edge_id,
new_edge_1_2.edge_id,
      // new_edge_1_3.edge_id, new_edge_1_4.edge_id};
      new_face_1.face_id = new_faces.size();
      new_faces.push_back(new_face_1);
```

```
      // 面 2(b,edge_point_bc,face_point,edge_point_ab)
      // 边 1(b,edge_point_bc)
      new_edge_2_1.v1_id = point_2.v_id;
      new_edge_2_1.v2_id = edge_point_2_3.v_id;
      // 边 2(edge_point_bc,face_point)
      new_edge_2_2.v1_id = edge_point_2_3.v_id;
      new_edge_2_2.v2_id = face_point.v_id;
      // 边 3(face_point,edge_point_ab)
      new_edge_2_3.v1_id = face_point.v_id;
      new_edge_2_3.v2_id = edge_point_1_2.v_id;
      // 边 4(edge_point_ab,b)
      new_edge_2_4.v1_id = edge_point_1_2.v_id;
      new_edge_2_4.v2_id = point_2.v_id;
      // 新增边
      new_edge_2_1.edge_id = new_edges.size();
      new_edges.push_back(new_edge_2_1);
      new_edge_2_2.edge_id = new_edges.size();
      new_edges.push_back(new_edge_2_2);
      new_edge_2_3.edge_id = new_edges.size();
      new_edges.push_back(new_edge_2_3);
      new_edge_2_4.edge_id = new_edges.size();
      new_edges.push_back(new_edge_2_4);
```

```
      // 新增面
      new_face_2.points = {point_2.v_id, edge_point_2_3.v_id,
face_point.v_id,
                          edge_point_1_2.v_id};
      // new_face_2.edges = {new_edge_2_1.edge_id,
new_edge_2_2.edge_id,
      //                    new_edge_2_3.edge_id,
new_edge_2_4.edge_id};
      new_face_2.face_id = new_faces.size();
      new_faces.push_back(new_face_2);
      // 面 3(c,edge_point_cd,face_point,edge_point_bc)
      // 边 1(c,edge_point_cd)
      new_edge_3_1.v1_id = point_3.v_id;
      new_edge_3_1.v2_id = edge_point_3_4.v_id;
      // 边 2(edge_point_cd,face_point)
      new_edge_3_2.v1_id = edge_point_3_4.v_id;
      new_edge_3_2.v2_id = face_point.v_id;
      // 边 3(face_point,edge_point_bc)
      new_edge_3_3.v1_id = face_point.v_id;
      new_edge_3_3.v2_id = edge_point_2_3.v_id;
      // 边 4(edge_point_bc,c)
      new_edge_3_4.v1_id = edge_point_2_3.v_id;
      new_edge_3_4.v2_id = point_3.v_id;
      // 新增边
      new_edge_3_1.edge_id = new_edges.size();
      new_edges.push_back(new_edge_3_1);
      new_edge_3_2.edge_id = new_edges.size();
      new_edges.push_back(new_edge_3_2);
      new_edge_3_3.edge_id = new_edges.size();
      new_edges.push_back(new_edge_3_3);
      new_edge_3_4.edge_id = new_edges.size();
      new_edges.push_back(new_edge_3_4);
      // 新增面
      new_face_3.points = {point_3.v_id, edge_point_3_4.v_id,
face_point.v_id,
                              edge_point_2_3.v_id};
      // new_face_3.edges = {new_edge_3_1.edge_id,
new_edge_3_2.edge_id,
      // new_edge_3_3.edge_id, new_edge_3_4.edge_id};
      new_face_3.face_id = new_faces.size();
      new_faces.push_back(new_face_3);
      // 面 4(d,edge_point_da,face_point,edge_point_cd)
      // 边 1(d,edge_point_da)
      new_edge_4_1.v1_id = point_4.v_id;
      new_edge_4_1.v2_id = edge_point_4_1.v_id;
      // 边 2(edge_point_da,face_point)
      new_edge_4_2.v1_id = edge_point_4_1.v_id;
      new_edge_4_2.v2_id = face_point.v_id;
      // 边 3(face_point,edge_point_cd)
      new_edge_4_3.v1_id = face_point.v_id;
      new_edge_4_3.v2_id = edge_point_3_4.v_id;
      // 边 4(edge_point_cd,d)
      new_edge_4_4.v1_id = edge_point_3_4.v_id;
      new_edge_4_4.v2_id = point_4.v_id;
      // 新增边
      new_edge_4_1.edge_id = new_edges.size();
      new_edges.push_back(new_edge_4_1);
      new_edge_4_2.edge_id = new_edges.size();
      new_edges.push_back(new_edge_4_2);
      new_edge_4_3.edge_id = new_edges.size();
      new_edges.push_back(new_edge_4_3);
      new_edge_4_4.edge_id = new_edges.size();
      new_edges.push_back(new_edge_4_4);
      // 新增面
      new_face_4.points = {point_4.v_id, edge_point_4_1.v_id,
face_point.v_id,
                              edge_point_3_4.v_id};
      // new_face_4.edges = {new_edge_4_1.edge_id,
new_edge_4_2.edge_id,
      // new_edge_4_3.edge_id, new_edge_4_4.edge_id};
      new_face_4.face_id = new_faces.size();
      new_faces.push_back(new_face_4);
      // std::cout << "新增边 new_edge_1_1:\n"
      //          << "\tfrom\t" << new_edge_1_1.v1_id <<
"\tto\t"
      //          << new_edge_1_1.v2_id << std::endl;
      // std::cout << "新增边 new_edge_1_2:\n"
      //          << "\tfrom\t" << new_edge_1_2.v1_id <<
```

```
"\tto\t"
      //           << new_edge_1_2.v2_id << std::endl;
      // std::cout << "新增边 new_edge_1_3:\n"
      //           << "\tfrom\t" << new_edge_1_3.v1_id <<
"\tto\t"
      //           << new_edge_1_3.v2_id << std::endl;
      // std::cout << "新增边 new_edge_1_4:\n"
      //           << "\tfrom\t" << new_edge_1_4.v1_id <<
"\tto\t"
      //           << new_edge_1_4.v2_id << std::endl;
      // std::cout << "新增边 new_edge_2_1:\n"
      //           << "\tfrom\t" << new_edge_2_1.v1_id <<
"\tto\t"
      //           << new_edge_2_1.v2_id << std::endl;
      // std::cout << "新增边 new_edge_2_2:\n"
      //           << "\tfrom\t" << new_edge_2_2.v1_id <<
"\tto\t"
      //           << new_edge_2_2.v2_id << std::endl;
      // std::cout << "新增边 new_edge_2_3:\n"
      //           << "\tfrom\t" << new_edge_2_3.v1_id <<
"\tto\t"
      //           << new_edge_2_3.v2_id << std::endl;
      // std::cout << "新增边 new_edge_2_4:\n"
      //           << "\tfrom\t" << new_edge_2_4.v1_id <<
"\tto\t"
      //           << new_edge_2_4.v2_id << std::endl;
      // std::cout << "新增边 new_edge_3_1:\n"
      //           << "\tfrom\t" << new_edge_3_1.v1_id <<
"\tto\t"
      //           << new_edge_3_1.v2_id << std::endl;
      // std::cout << "新增边 new_edge_3_2:\n"
      //           << "\tfrom\t" << new_edge_3_2.v1_id <<
"\tto\t"
      //           << new_edge_3_2.v2_id << std::endl;
      // std::cout << "新增边 new_edge_3_3:\n"
      //           << "\tfrom\t" << new_edge_3_3.v1_id <<
"\tto\t"
      //           << new_edge_3_3.v2_id << std::endl;
      // std::cout << "新增边 new_edge_3_4:\n"
      //           << "\tfrom\t" << new_edge_3_4.v1_id <<
"\tto\t"
      //           << new_edge_3_4.v2_id << std::endl;
      // std::cout << "新增边 new_edge_4_1:\n"
      //           << "\tfrom\t" << new_edge_4_1.v1_id <<
```

```
"\tto\t"
      //            << new_edge_4_1.v2_id << std::endl;
      // std::cout << "新增边 new_edge_4_2:\n"
      //            << "\tfrom\t" << new_edge_4_2.v1_id <<
"\tto\t"
      //            << new_edge_4_2.v2_id << std::endl;
      // std::cout << "新增边 new_edge_4_3:\n"
      //            << "\tfrom\t" << new_edge_4_3.v1_id <<
"\tto\t"
      //            << new_edge_4_3.v2_id << std::endl;
      // std::cout << "新增边 new_edge_4_4:\n"
      //            << "\tfrom\t" << new_edge_4_4.v1_id <<
"\tto\t"
      //            << new_edge_4_4.v2_id << std::endl;
    }
    else if (i.points.size() == 3)
    {
      auto point_1 = *std::find_if(new_points.begin(),
new_points.end(),
                                   [i](const my_point& p)
                                   { return p.v_id ==
i.points[0]; });
      auto point_2 = *std::find_if(new_points.begin(),
new_points.end(),
                                   [i](const my_point& p)
                                   { return p.v_id ==
i.points[1]; });
      auto point_3 = *std::find_if(new_points.begin(),
new_points.end(),
                                   [i](const my_point& p)
                                   { return p.v_id ==
i.points[2]; });
      // std::cout << "三角形面" << i.face_id << std::endl;
      // std::cout << "point_1:" << "(" << point_1.x << "," <<
point_1.y << ")"
      //            << std::endl;
      // std::cout << "point_2:" << "(" << point_2.x << "," <<
point_2.y << ")"
      //            << std::endl;
      // std::cout << "point_3:" << "(" << point_3.x << "," <<
point_3.y << ")"
      //            << std::endl;
      auto edge_1_2_index = *std::find_if(
          all_edges.begin(), all_edges.end(),
          [point_1, point_2](const my_edge& e)
          {
            return (e.v1_id == point_1.v_id && e.v2_id ==
point_2.v_id) ||
                   (e.v1_id == point_2.v_id && e.v2_id ==
point_1.v_id);
          });
      auto edge_point_1_2 =
          *std::find_if(edge_points.begin(),
edge_points.end(),
                        [edge_1_2_index](const my_point& p)
                        { return p.v_id ==
edge_1_2_index.edge_id; });
      auto edge_2_3_index = *std::find_if(
          all_edges.begin(), all_edges.end(),
          [point_2, point_3](const my_edge& e)
          {
            return (e.v1_id == point_2.v_id && e.v2_id ==
point_3.v_id) ||
                   (e.v1_id == point_3.v_id && e.v2_id ==
point_2.v_id);
          });
      auto edge_point_2_3 =
          *std::find_if(edge_points.begin(),
edge_points.end(),
                        [edge_2_3_index](const my_point& p)
                        { return p.v_id ==
edge_2_3_index.edge_id; });
      auto edge_3_1_index = *std::find_if(
          all_edges.begin(), all_edges.end(),
          [point_3, point_1](const my_edge& e)
          {
            return (e.v1_id == point_3.v_id && e.v2_id ==
point_1.v_id) ||
                   (e.v1_id == point_1.v_id && e.v2_id ==
point_3.v_id);
          });
      auto edge_point_3_1 =
          *std::find_if(edge_points.begin(),
edge_points.end(),
                        [edge_3_1_index](const my_point& p)
                        { return p.v_id ==
edge_3_1_index.edge_id; });
      auto face_point =
            *std::find_if(face_points.begin(),
face_points.end(),
                              [i](const my_point& p) { return p.v_id
== i.face_id; });
        // 新增点
```

```
        point_1.v_id = new_points_tmp.size();
        new_points_tmp.push_back(point_1);
        point_2.v_id = new_points_tmp.size();
        new_points_tmp.push_back(point_2);
        point_3.v_id = new_points_tmp.size();
        new_points_tmp.push_back(point_3);
        edge_point_1_2.v_id = new_points_tmp.size();
        new_points_tmp.push_back(edge_point_1_2);
        edge_point_2_3.v_id = new_points_tmp.size();
        new_points_tmp.push_back(edge_point_2_3);
        edge_point_3_1.v_id = new_points_tmp.size();
        new_points_tmp.push_back(edge_point_3_1);
        face_point.v_id = new_points_tmp.size();
        new_points_tmp.push_back(face_point);
        my_face new_face_1;
        my_face new_face_2;
        my_face new_face_3;
        my_edge new_edge_1_1;
        my_edge new_edge_1_2;
        my_edge new_edge_1_3;
        my_edge new_edge_1_4;
        my_edge new_edge_2_1;
        my_edge new_edge_2_2;
        my_edge new_edge_2_3;
        my_edge new_edge_2_4;
        my_edge new_edge_3_1;
        my_edge new_edge_3_2;
        my_edge new_edge_3_3;
        my_edge new_edge_3_4;
        // 面 1(a, edge_point_ab, face_point, edge_point_ca)
        // 边 1(a, edge_point_ab)
        new_edge_1_1.v1_id = point_1.v_id;
        new_edge_1_1.v2_id = edge_point_1_2.v_id;
        // 边 2(edge_point_ab, face_point)
        new_edge_1_2.v1_id = edge_point_1_2.v_id;
        new_edge_1_2.v2_id = face_point.v_id;
        // 边 3(face_point, edge_point_ca)
        new_edge_1_3.v1_id = face_point.v_id;
```

```
      new_edge_1_3.v2_id = edge_point_3_1.v_id;
      // 边 4(edge_point_ca, a)
      new_edge_1_4.v1_id = edge_point_3_1.v_id;
      new_edge_1_4.v2_id = point_1.v_id;
      // 新增边
      new_edge_1_1.edge_id = new_edges.size();
      new_edges.push_back(new_edge_1_1);
      new_edge_1_2.edge_id = new_edges.size();
      new_edges.push_back(new_edge_1_2);
      new_edge_1_3.edge_id = new_edges.size();
      new_edges.push_back(new_edge_1_3);
      new_edge_1_4.edge_id = new_edges.size();
      new_edges.push_back(new_edge_1_4);
      // 新增面
      new_face_1.points = {point_1.v_id, edge_point_1_2.v_id,
face_point.v_id,
                           edge_point_3_1.v_id};
      // new_face_1.edges = {new_edge_1_1.edge_id,
new_edge_1_2.edge_id,
      // new_edge_1_3.edge_id};
      new_face_1.face_id = new_faces.size();
      new_faces.push_back(new_face_1);
      // 面 2(b, edge_point_bc, face_point, edge_point_ab)
      // 边 1(b, edge_point_bc)
      new_edge_2_1.v1_id = point_2.v_id;
      new_edge_2_1.v2_id = edge_point_2_3.v_id;
      // 边 2(edge_point_bc, face_point)
      new_edge_2_2.v1_id = edge_point_2_3.v_id;
      new_edge_2_2.v2_id = face_point.v_id;
      // 边 3(face_point, edge_point_ab)
      new_edge_2_3.v1_id = face_point.v_id;
      new_edge_2_3.v2_id = edge_point_1_2.v_id;
      // 边 4(edge_point_ab, b)
      new_edge_2_4.v1_id = edge_point_1_2.v_id;
      new_edge_2_4.v2_id = point_2.v_id;
      // 新增边
      new_edge_2_1.edge_id = new_edges.size();
      new_edges.push_back(new_edge_2_1);
      new_edge_2_2.edge_id = new_edges.size();
      new_edges.push_back(new_edge_2_2);
      new_edge_2_3.edge_id = new_edges.size();
      new_edges.push_back(new_edge_2_3);
      new_edge_2_4.edge_id = new_edges.size();
      new_edges.push_back(new_edge_2_4);
      // 新增面
      new_face_2.points = {point_2.v_id, edge_point_2_3.v_id,
face_point.v_id,
                           edge_point_1_2.v_id};
      // new_face_2.edges = {new_edge_2_1.edge_id,
new_edge_2_2.edge_id,
      // new_edge_2_3.edge_id};
      new_face_2.face_id = new_faces.size();
      // 面 3(c, edge_point_ca, face_point, edge_point_bc)
      // 边 1(c, edge_point_ca)
      new_faces.push_back(new_face_2);
      new_edge_3_1.v1_id = point_3.v_id;
      new_edge_3_1.v2_id = edge_point_3_1.v_id;
      // 边 2(edge_point_ca, face_point)
      new_edge_3_2.v1_id = edge_point_3_1.v_id;
      new_edge_3_2.v2_id = face_point.v_id;
      // 边 3(face_point, edge_point_bc)
      new_edge_3_3.v1_id = face_point.v_id;
      new_edge_3_3.v2_id = edge_point_2_3.v_id;
      // 边 4(edge_point_bc, c)
      new_edge_3_4.v1_id = edge_point_2_3.v_id;
      new_edge_3_4.v2_id = point_3.v_id;
      // 新增边
      new_edge_3_1.edge_id = new_edges.size();
      new_edges.push_back(new_edge_3_1);
      new_edge_3_2.edge_id = new_edges.size();
      new_edges.push_back(new_edge_3_2);
      new_edge_3_3.edge_id = new_edges.size();
      new_edges.push_back(new_edge_3_3);
      new_edge_3_4.edge_id = new_edges.size();
      new_edges.push_back(new_edge_3_4);
      // 新增面
      new_face_3.points = {point_3.v_id, edge_point_3_1.v_id,
face_point.v_id,
                           edge_point_2_3.v_id};
      // new_face_3.edges = {new_edge_3_1.edge_id,
new_edge_3_2.edge_id,
      //                     new_edge_3_3.edge_id};
      new_face_3.face_id = new_faces.size();
      new_faces.push_back(new_face_3);
      // std::cout << "新增边 new_edge_1_1:\n"
      //           << "\tfrom\t" << new_edge_1_1.v1_id <<
"\tto\t"
      //           << new_edge_1_1.v2_id << std::endl;
      // std::cout << "新增边 new_edge_1_2:\n"
      //             << "\tfrom\t" << new_edge_1_2.v1_id <<
"\tto\t"
      //             << new_edge_1_2.v2_id << std::endl;
      // std::cout << "新增边 new_edge_1_3:\n"
      //             << "\tfrom\t" << new_edge_1_3.v1_id <<
"\tto\t"
      //             << new_edge_1_3.v2_id << std::endl;
      // std::cout << "新增边 new_edge_1_4:\n"
      //             << "\tfrom\t" << new_edge_1_4.v1_id <<
"\tto\t"
      //             << new_edge_1_4.v2_id << std::endl;
      // std::cout << "新增边 new_edge_2_1:\n"
      //             << "\tfrom\t" << new_edge_2_1.v1_id <<
"\tto\t"
      //             << new_edge_2_1.v2_id << std::endl;
      // std::cout << "新增边 new_edge_2_2:\n"
      //             << "\tfrom\t" << new_edge_2_2.v1_id <<
"\tto\t"
      //             << new_edge_2_2.v2_id << std::endl;
      // std::cout << "新增边 new_edge_2_3:\n"
      //             << "\tfrom\t" << new_edge_2_3.v1_id <<
"\tto\t"
      //             << new_edge_2_3.v2_id << std::endl;
      // std::cout << "新增边 new_edge_2_4:\n"
      //             << "\tfrom\t" << new_edge_2_4.v1_id <<
"\tto\t"
      //             << new_edge_2_4.v2_id << std::endl;
      // std::cout << "新增边 new_edge_3_1:\n"
      //             << "\tfrom\t" << new_edge_3_1.v1_id <<
"\tto\t"
      //             << new_edge_3_1.v2_id << std::endl;
      // std::cout << "新增边 new_edge_3_2:\n"
      //             << "\tfrom\t" << new_edge_3_2.v1_id <<
"\tto\t"
      //             << new_edge_3_2.v2_id << std::endl;
      // std::cout << "新增边 new_edge_3_3:\n"
      //             << "\tfrom\t" << new_edge_3_3.v1_id <<
"\tto\t"
      //             << new_edge_3_3.v2_id << std::endl;
      // std::cout << "新增边 new_edge_3_4:\n"
      //             << "\tfrom\t" << new_edge_3_4.v1_id <<
"\tto\t"
      //             << new_edge_3_4.v2_id << std::endl;
```

```
    }
  }
  // 最终结果
  new_faces.erase(new_faces.begin());
  new_edges.erase(new_edges.begin());
  new_points_tmp.erase(new_points_tmp.begin());
  all_faces = new_faces;
  all_edges = new_edges;
  all_points = new_points_tmp;
  bind_edges();
  // for (auto& i : all_points)
  // {
  //   std::cout << "点" << i.v_id << "(" << i.x << "," << i.y
<< ")" << std::endl;
  // }
  // for (auto& i : all_edges)
  // {
  //   std::cout << "边" << i.edge_id << " from " << i.v1_id
<< " to " << i.v2_id
  //              << std::endl;
  // }
  // for (auto& i : all_faces)
  // {
  //   std::cout << "面" << i.face_id << std::endl;
  //   std::cout << "\t 点\t";
  //   for (auto& j : i.points)
  //   {
  //     std::cout << j << " ";
  //   }
  //   std::cout << std::endl;
  //   std::cout << "\t 边\t";
  //   for (auto& j : i.edges)
  //   {
  //     std::cout << j << " ";
  //   }
  //   std::cout << std::endl;
  // }
}
void write_all(const char* path)
{
  std::ofstream ofs(path);
  for (auto& i : all_points)
  {
    ofs << "v " << (i.x - OFFSET) / SCALE << " " << (i.y - OFFSET)
```

```
/ SCALE
        << "\n";
  }
  for (auto& i : all_edges)
  {
    ofs << "e " << i.v1_id << " " << i.v2_id << "\n";
  }
  for (auto& i : all_faces)
  {
    ofs << "f ";
    for (auto& j : i.points)
    {
      ofs << j << " ";
    }
    ofs << "\n";
  }
}
int main(int argc, char** argv)
{
  std::string filepath = "obj1.txt";
  if (argc > 1)
  {
    filepath = argv[1];
  }
  size_t loop_times = 1;
  if (argc > 2)
  {
    loop_times = std::stoull(argv[2]);
  }
  read_file(filepath.c_str());
  Mat img = Mat::zeros(Size(800, 800), CV_8UC3);
  img.setTo(255); // 设置屏幕为白色
  draw_all(img);
  imshow("画板", img);
  waitKey(0);
  for (size_t i = 0; i < loop_times; i++)
  {
    // if (i > 0)
    // {
    //   read_file(".temp.obj");
    // }
    std::cout << "\n\n\n";
    std::cout << "第" << i + 1 << "次细分" << std::endl;
    std::cout << "\n\n\n";
```

```cpp
    img.setTo(255);
    catmull_clark();
    draw_all(img);
    imshow("画板", img);
    waitKey(0);
    // write_all(".temp.obj");
  }
  // system("rm .temp.obj");
  if (loop_times == 0)
  {
    img.setTo(255);
    draw_all(img);
    imshow("画板", img);
    waitKey(0);
  }
  // 输出 png
  imwrite("output.png", img);
  if (argc > 3)
  {
    if (strcmp(argv[3], "-o") == 0)
    {
      std::string output_filepath = argv[4];
      write_all(output_filepath.c_str());
    }
  }
  return 0;
}
```
实验结果的截图如下：

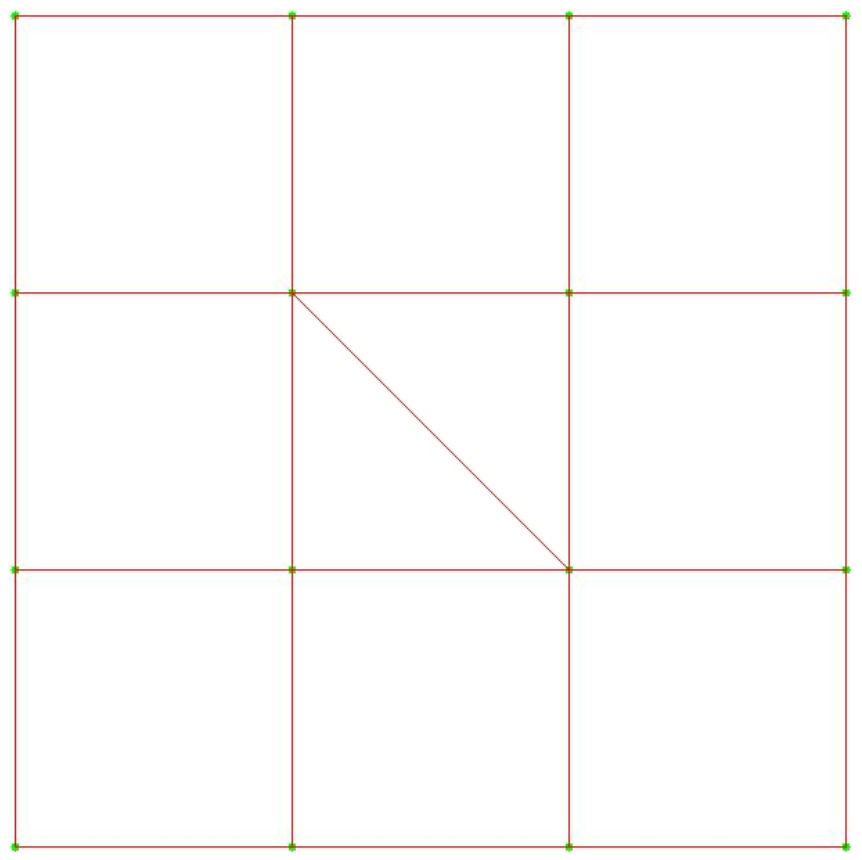


图 1 带三角的网格不细分

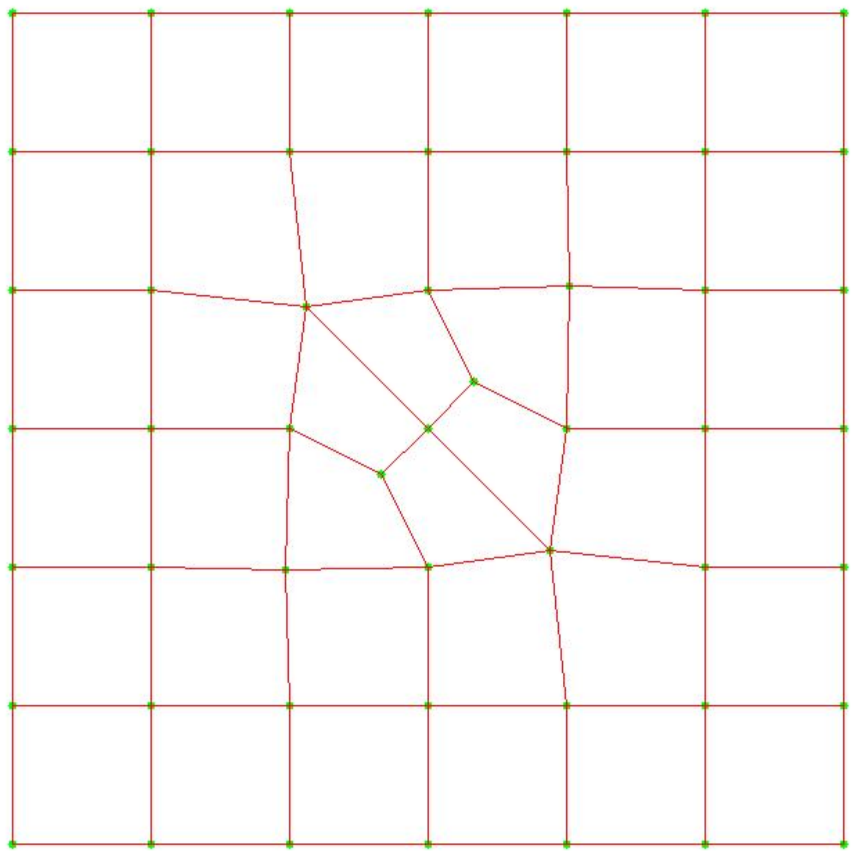


图 2 带三角的网格细分一次

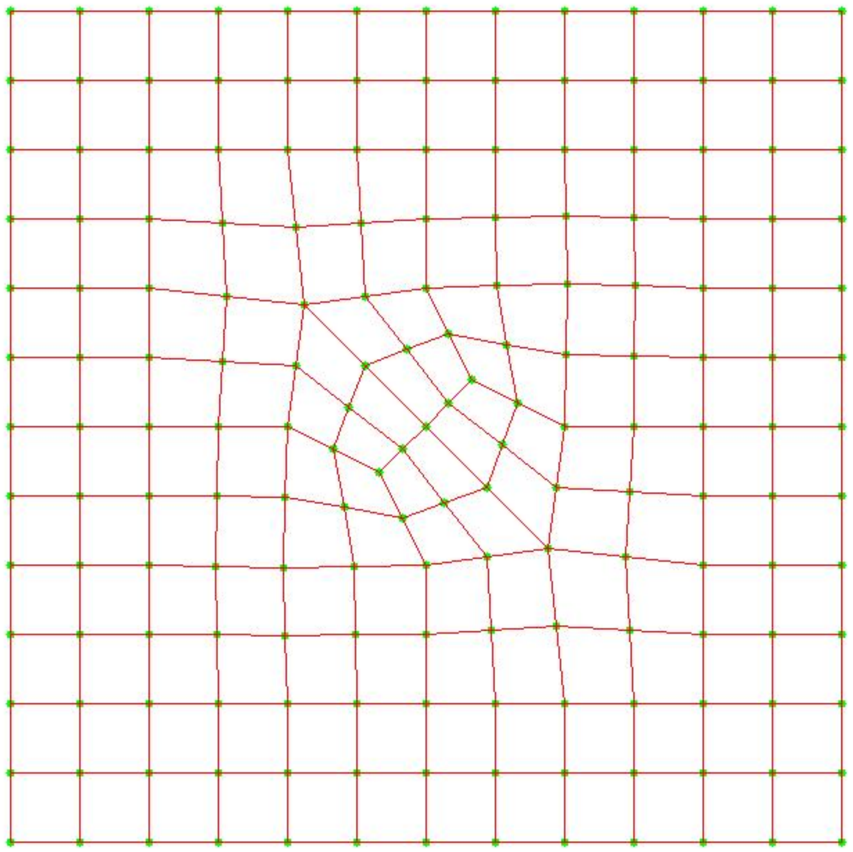


图 3 带三角的网格细分两次

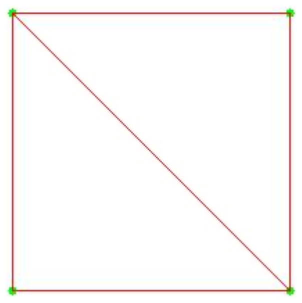


图 4 最简单的三角网格不细分

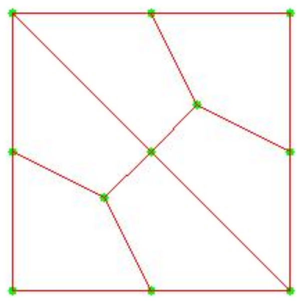


图 5 最简单的三角网格细分一次

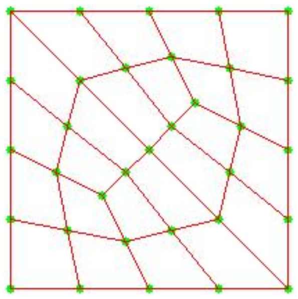


图 6 最简单的三角网格细分两次

实验中存在的问题及解决：

问题 1：Catmull-Clark 算法最初设计是三维闭合图形的细分，但是本次实验要求的是对二维非闭合平面图形进行细分，应该怎么办？

回答 1：只需要对边界情况进行特殊处理即可，在确认一个点是边界点（即图形边缘的点，数学的描述就是相邻面和边的个数不一致）之后，只需要在更新原始点坐标的时候不更新这个点的坐标即可。

问题 2：实验指导书给出的点坐标更新算法是分别处理了奇异点和非奇异点，有没有通用的办法？

回答 2：有的兄弟有的，Catmull-Clark 算法的提出者给出了通用计算公式，即无论该点有多少个相邻面都可以按这个方式计算：

$$\frac{F + 2R + (n-3)P}{n}$$

问题 3：实验指导书给的边存储面以及面存储边和点的方式是储存索引，而实验指导书使用 vector 这种线性结构来存储所有点和边和面，这样是不是会麻烦很多，降低效率？有没有更好的选择？

回答 3：我全程按照实验指导书给的类（结构体）定义以及用 vector 来作为全局容器来完成这个实验，全程体验下来我可以很清楚地说：用 vector 存储没有任何好处，甚至全是坏处。首先这会让存索引，准确的说是存编号这种方式变得意义不明，为了迎合这种存储方式，在很多地方需要使用一些难以维护的 find 函数，也难以使用 C++的运算符重载等特性，并且因为 vector 这种线性容器 O(n)的查询时间复杂度，当点的个数变多之后（实测下来当点的个数达到 6 万个左右的时候延迟就会达到秒级别）。只能说使用 vector 是完完全全不如使用 unordered_map 的，而且使用唯一编号做索引这种方式，因为这个编号是 size_t(unsigned long long)类型，他是非常适合使用哈希表这种数据结构的，效率又高，维护起来又容易。不过为了按照实验指导书的做法完成实验，我并没有使用。