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](mailto:2367651943@qq.com)**  **[202300130183@mail.sdu.edu.cn](mailto:202300130183@mail.sdu.edu.cn)**  **[zhitian0420@gmai.com](mailto: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：      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，那我们的面点就是：   1. 计算边点   对于每一个边E，假设构成这个边的点集合为，和这个边相邻的面为，易得面最多有两个。我们用表示这个边的中点，那我们可以得到边点：   1. 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：   1. 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++;        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";      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：  带三角网格不细分  图 1 带三角的网格不细分  带三角网格细分一次  图 2 带三角的网格细分一次  带三角网格细分两次  图 3 带三角的网格细分两次  最简易三角网格不细分  图 4 最简单的三角网格不细分  最简易三角网格细分一次  图 5 最简单的三角网格细分一次  最简易带三角网格细分两次  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：  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。 | | |