唐无忌 7.19-7.25

本周主要复习了 C++的基础知识，包括预处理指令，类的基本使用，内联函数，操作符重载等，之后学习了 Eigen 库和 opencv 库的简单使用，并依此在 visual studio 上实现了图形学课程里的一个代码作业：将一个三角形光栅化并绘制在屏幕上。

预处理指令

#define，#ifndef，#endif，其中#表示这是一条预处理命令，凡是#开头的都是预处理命令，预处理命令分为三种：宏定义、文件包含、条件编译。宏定义分为有参数和无参数两种。

无参宏定义：

#define 标识符 字符串

define 为宏定义命令，“标识符”是定义的宏名，“字符串”可以是常数、表达式、格式串等，要终止其作用域可以使用#undef。

有参宏定义：

#define 宏名（形参表） 字符串

调用的时候要用实参去替换形参。

条件编译：

#ifndef，#endif

#表示预处理命令；ifndef 是 if not define 的缩写，是与处理功能的条件编译，目的是防止头文件的重复包含和编译。

在 c++中用#ifndef 可以避免出现以下错误：在.hpp 头文件中定义了全局变量，一个.cpp 文件中多次包含这个.hpp 文件，加上#ifndef 可以避免重复定义的错误。

#pragma once

除了用#ifndef 防止 h 文件的代码重复引用，还可以用#pragma once 指令，在想要保护的文件开头写入。相比#ifndef（灵活，兼容性好），#pragma once 编译时间相对较短，操作简单相率高，是一个非标准但被广泛支持的方式。

Struct 结构体

struct 结构体，默认公有继承，是 public 的。C++中的 struct 不同于 C 中的，C 中的 struct 不能定义函数只能定义数据成员，数据和操作是分开的；C++中的 struct 可以包含成员函数，可修改 public / private / protected，可以直接用{}初始化。

inline 内联函数

栈空间：也叫栈内存，可以理解成程序放置局部数据的内存空间。栈空间是有限的，频繁大量的使用会导致程序出错，例如某函数的死循环的尽头就是将栈内存耗尽。为了解决一些频繁调用的小函数大量消耗栈内存的问题，引入了 inline 修饰符，表示为内联函数。

```
#include<iostream>
using namespace std;

inline string test(int a)
{
    return (a % 2 > 0) ? "奇数" : "偶数";
}

int main()
{
    int i = 0;
    for (i = 1; i < 100; i++)
    {
        cout << test(i) << endl;
    }
    return 0;
}
```

上述例子就是把调用到 test(i)的地方都换成了 (a % 2 > 0) ? "奇数" : "偶数"，避免频繁调用函数。但是，inline 的使用对编译器来说只是一个建议，编译器可以选择忽略这个建议，取决于函数的长短；当你将一个 1000 多行的函数定位 inline，编译器还会按照普通函数进行运行。

Operator 运算符重载

operator 是 C++的一个关键字，和运算符一起使用（例如：operator+），表示一个运算符重载函数，可将其视为一个函数名。C++提供的运算符只支持基本数据类型和标准库中提供的类操作，对于自己定义的 class 类，想要实现运算符操作（比大小、判断是否相等）就需要自己定义运算符的实现过程。

```
class person
{
private:
    int age;
public:
    person(int nage)
    {
        this->age = nage;
    }

    bool operator==(const person& ps)
    {
        if (this->age == ps.age)
        {
            return true;
        }
        return false;
    }
};
```

图形学作业：

get_view_matrix()，将相机移动到坐标原点并调整好朝向。在本作业里，相机就已经朝向-z 了，则只需要移动就行。

```
Eigen::Matrix4f get_view_matrix(Eigen::Vector3f eye_pos)
{
    Eigen::Matrix4f view = Eigen::Matrix4f::Identity();

    Eigen::Matrix4f translate;
    translate << 1, 0, 0, -eye_pos[0], 0, 1, 0, -eye_pos[1], 0, 0, 1,
        -eye_pos[2], 0, 0, 0, 1;

    view = translate * view;

    return view;
}
```

get_model_matrix()，将三角形绕 z 轴旋转，直接左乘旋转矩阵就行。

```
Eigen::Matrix4f get_model_matrix(float rotation_angle)
{
    Eigen::Matrix4f model = Eigen::Matrix4f::Identity();
    Eigen::Matrix4f rot;
    float angle = rotation_angle / 180 * MY_PI;
    rot <<
        cos(angle), -sin(angle), 0, 0,
        sin(angle), cos(angle), 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1;
    model = rot * model;
    return model;
}
```

get_projection_matrix()，透视投影变换，可以先缩放再正交投影，即左乘 perspective->orthographic 矩阵，再左乘 orthographic 矩阵。

```
    //透视->正交 perspective->orthographic
    Eigen::Matrix4f pertoorth;
    pertoorth << n, 0, 0, 0,
        0, n, 0, 0,
        0, 0, n + f, -n * f,
        0, 0, 1, 0;
    //正交——移动
    Eigen::Matrix4f orth;
    orth <<
        2 / (r - l), 0, 0, -(r + l) / 2,
        0, 2 / (t - b), 0, -(t + b) / 2,
        0, 0, 2 / (n - f), -(n + f) / 2,
        0, 0, 0, 1;
    //正交——缩放
    projection = orth * pertoorth;//注意矩阵顺序，变换从右往左依次进行
    return projection;
```

进行了 mvp 变换之后，就可以将得到的三角形光栅化了。

首先要给三角形取包围盒 bouding_box，就是三顶点坐标的最大最小的 x 和 y。

```
int min_x = std::min(std::min(v[0].x(), v[1].x()), v[2].x());
int min_y = std::min(std::min(v[0].y(), v[1].y()), v[2].y());
int max_x = std::max(std::max(v[0].x(), v[1].x()), v[2].x());
int max_y = std::max(std::max(v[0].y(), v[1].y()), v[2].y());
```

之后对每个像素点，判断像素中心(x+0.5,y+0.5)是否在三角形内部。用函数 insideTriangle()，判断点是否在三角形内。对 P 点，若 AP×AB，BP×BC，CP×CA，三组向量叉乘同号则 P 在三角形内，否则在三角形外。

```
static bool insideTriangle(int x, int y, const Vector3f* _v)
{
    //.head(2)用来取出每个顶点的前两个值(x,y)
    Eigen::Vector2f AB, BC, CA, AP, BP, CP, p;
    float a, b, c;
    p << x, y;
    AB = _v[1].head(2) - _v[0].head(2);
    AP = p - _v[0].head(2);
    BC = _v[2].head(2) - _v[1].head(2);
    BP = p - _v[1].head(2);
    CA = _v[0].head(2) - _v[2].head(2);
    CP = p - _v[2].head(2);
    a = AB[0] * AP[1] - AB[1] * AP[0];
    b = BC[0] * BP[1] - BC[1] * BP[0];
    c = CA[0] * CP[1] - CA[1] * CP[0];
    if (a > 0 && b > 0 && c > 0) return true;
    else if (a < 0 && b < 0 && c < 0) return true;
    return false;
}
```

若要抗锯齿，可将一个像素分成四个像素，即判断(x+0.25,y+0.25)，(x+0.25,y+0.75)，(x+0.75,y+0.25)，(x+0.75,7+0.75)四个点，根据在三角形内点的数量来确定深度再绘制颜色。